28 Embedding Non-Text Characters

28.1 Inserting graphics & user items

DEFINITION: A non-text character is a graphic display embedded into the text stream of an OpenPaige document, such as a picture, box or special string (such as a page number, footnote, etc.). It is not an ASCII byte as such, but otherwise looks and behaves like an ordinary character. It can be clicked, deleted, cut, copied, and pasted.

The purpose of this chapter is to explain the built-in, high-level support for these special characters.

DISCLAIMER

There are several undocumented references in `pgEmbed.h`. If anything in that header file is not explained in this chapter, it is *not supported*. The purpose of these definitions is for possible future enhancement and/or custom development by DataPak Software, Inc.

Description

OpenPaige provides a certain degree of built-in support for graphic characters. For the Macintosh version, `PicHandles` (pictures) can be inserted into the text stream with practically no support required from your application. For the Windows version, meta files can be inserted in the same way.

For other graphic types and/or "user items" (custom characters), OpenPaige supports a variety of user-defined non-text character insertions; your application can then handle the display and other rendering through a single callback function.

All the functions documented in this chapter are prototyped in pgEmbed.h. You therefore need to include this header file to use the structures, functions and callbacks.

28.2 The embed_ref

The first step to embedding a non-text character is to create an `embed_ref`:

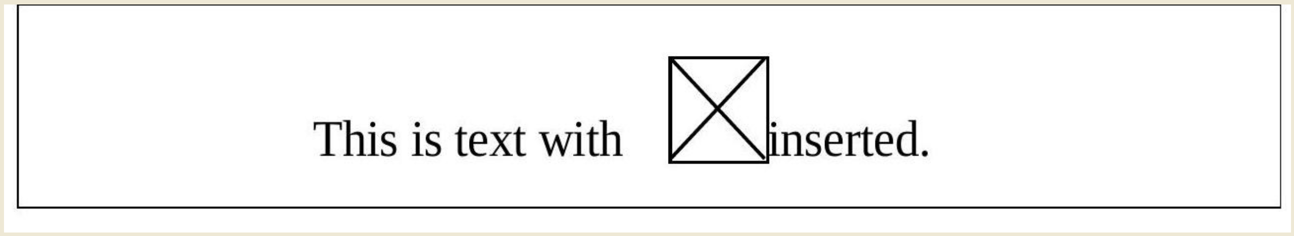
```
embed_ref pgNewEmbedRef (pgm_globals_ptr mem_globals, long item_type, void
PG_FAR *item_data, long modifier, long flags, pg_fixed, vert_pos, long
user_refcon, pg_boolean keep_around)
```

This function returns a special memory_ref that can be subsequently inserted into a pg_ref as a "character". Once you have created an embed_ref, call pgInsertEmbedRef() below.

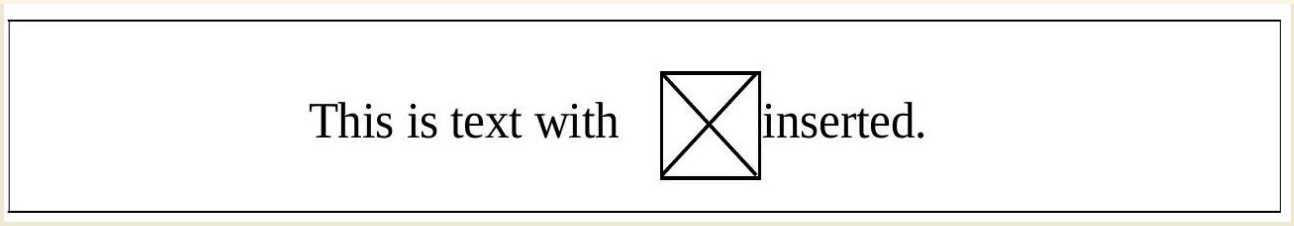
- mem_globals - must be a pointer to your memory globals (same structure that was also given to pgMemStartup and pgInit).
- item_type - indicates the kind of object you want to create. This value can be any value shown in "Embed_Ref Types”.
- item_data, modifiers - What you provide in item_data and modifiers depends on the item_type; these are also described in "Embed Reference Types."
- flags - should be set to zero (with some unusual exceptions - "Special Cases").
- vert_pos - Its purpose is to indicate a descent value for the object you will be inserting. By *descent* is meant the amount the item should be offset vertically below the baseline.
  If vert_pos is positive, it is considered to be a percent of the item's total height. If vert_pos is negative, it is considered to be a pixel value. Carefully note that in both cases, vert_pos is a Fixed value - the high-order word is the whole part and the low-order word is the fraction.

  For example, if vert_pos is 0x000A0000, the embed_ref will be offset 10% of its total height from the text baseline. If vert_pos is 0xFFFEFFFF (negative 0x00020000), the item will be offset 2 pixels below the text baseline, etc.

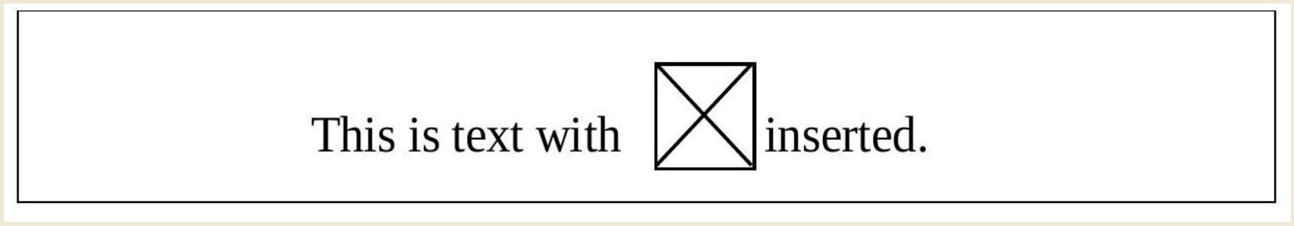
  The following illustrations show a typical embed_ref's descent values for different vert_pos values:

  The above shows the result of an embed_ref with vert_pos = 0 (no descent from baseline).



The above shows the result of an `embed_ref` with `vert_pos` = 50.00 (descent is 50% of height).



The above shows the result of an `embed_ref` with `vert_pos` = -3 (descent is 3 pixels)

- `user_refcon` - is saved inside the `embed_ref` itself, and can be anything.
- `keep_around` - indicates whether or not the `embed_ref` can be automatically disposed once it is no longer being used within any existing `pg_ref`. If this value is `FALSE` then OpenPaige is authorized to dispose of it once it is no longer being used by any `pg_ref`; a `TRUE` value tells OpenPaige it must never dispose it even if no `pg_ref` contains the embed_ref.

  To understand the full meaning of `keep_around`, the deceloper should realise that an `embed_ref` can be "shared" by multiple positions in a document, and even between different documents. For example, if the user performs multiple copy/paste operations on a single `embed_ref`, OpenPaige won't actually duplicate the `embed_ref`; rather, it simply creates multiple pointers to its data.

However, once the last remaining shared copy of the reference is deleted, OpenPaige will dispose the `memory_ref` (if `keep_around` = `TRUE`). Normally, this is what you would want.

NOTE: If `keep_around` is `FALSE` you should never dispose the `embed_ref` (OpenPaige will dispose of it at the appropriate time). If `keep_around` is TRUE then you need to eventually dispose the reference with `pgDisposeEmbedRef()`.

28.3 Inserting the `embed_ref`

```
pg_boolean pgInsertEmbedRef (pg_ref pg, embed_ref ref, long position, short
stylesheet_option, embed_callback callback, long callback_refcon, short
draw_mode);
```

This function inserts an `embed_ref` as a "character" into `pg` at the specified text location. The position parameter indicates the text offset (relative to zero) to insert the `embed_ref`; the position parameter can also be `CURRENT_POSITION` which causes the insertion to occur at the current insertion point.

- `stylesheet_option` - is an optional stylesheet ID that gets automatically applied to the `embed_ref` "character". If you merely want to use whatever style applies to the text position, pass zero for `stylesheet_option`, otherwise you need to create a new stylesheet and provide that stylesheet ID.
- `callback` - is a pointer to a function (in your application) that will be called for various occurrences; the purpose of this callback is to handle custom characters and/or to modify the default behavior of standard `embed_refs`. However, if you want OpenPaige to handle the `embed_ref` in the default way, pass `NULL` for `callback`.

  NOTE: Only certain `embed_ref` types are supported with a "default behaviour", and therefore only those types will work correctly if you pass NULL for callback; see section 28.4, *Embed Ref Types*. below.
- `callback_refcon` - value can be anything you want; this same value will be given to your callback function. If you have not supplied a callback function (`callback = NULL`), this parameter does not matter.
- `draw_mode` - indicates whether or not to redraw the text after the `embed_ref` is inserted; the value you pass for this parameter is identical to all other OpenPaige functions that accept a `draw_mode`.

Notes and Cautions

- Value `callback_refcon` given to `pgInsertEmbedRef` is *not* to be confused with `user_refcon` given to `pgNewEmbedRef`; they are entirely *separate*. The `user_refcon` given to `pgNewEmbedRef` is stored within the `embed_ref` itself, and the same `embed_ref` can exist as multiple copies throughout a document; the `callback_refcon` is specific to the insertion itself and can be different
  for all occurrences of the `embed_ref`. The `callback_refcon` is stored in the `style_info` that is applied to the insertion, and is

also passed to your callback function (if one exists).

The user_refcon will always be whatever value you passed to pgInsertEmbedRef() in the callback_refcon parameter except for EMBED_READ_DATA and EMBED_WRITE_DATA, in which case the user_refcon will be the original user_refcon value given to pgNewEmbedRef().

- Do not insert the same embed_ref more than once unless you have created it with keep_around = TRUE. Otherwise, OpenPaige can dispose it prematurely and your program will crash. Once it has been inserted, however, it is OK to copy and paste that character to as many documents as memory permits.
- Once the embed_ref has been inserted, OpenPaige "owns" its memory, i.e. you *must not* dispose of it as long as it exists in any pg_ref (and, if you passed FALSE for keep_around in pgNewEmbedRef(), you *must not* dispose of it at all, at any time).

The Style Sheet Option

Any embed_ref has the option to alter the style and/or font of the text it applies to.

For example, a string-type embed_ref (embed_alternate_char) will normally assume the style and font of the text where it was embedded; however, by passing a non-zero stylesheet_id number in stylesheet_option in pgInsertEmbedRef or pgSetEmbedRef, the style or font can be overridden.

A stylesheet_id is obtained by first creating a new style_info record and adding it to the pg_ref as a new style sheet (see chapter 31, "Style Sheets"). The stylesheet_id is then given to the stylesheet_option parameter, in which case the text for which the embed_ref applies will assume that style.

However, a style sheet applied to an embed_ref works slightly differently than normal styles: only the non-zero items in the style_info record of the stylesheet are applied.

For example, let us suppose that a new style sheet is created with every field in its style_info record set to zero except for the italic attribute. If this style sheet is applied to an embed_ref, the text is forced to italic but retains all other attributes (same point size as before, same font, etc.).

The following is an example of applying italic to an embed_ref; note that the embed_ref text retains all style and font characteristics except that it is italicised:

```
style_info new_sheet;
short stylesheet_id;

pgFillBlock(&new_sheet, sizeof(style_info), 0); // Fill with all zeros
style_info.styles[italic_var] = -1;       // Set for italic
stylesheet_id = pgNewStyle(pg, &new_sheet, NULL);

// Now include in "stylesheet_option":

pgSetEmbedRef(pg, ref, NULL, stylesheet_id, 0, best_way);
```

If you want to change only the font in the embed_ref text, use the same example as above except omit changing the italic attribute and pass a font_info record instead of NULL for pgNewStyle.

28.4 embed_ref *Types*

The following table describes each possible embed_ref data type and what you should pass in the item_data and modifier parameters for pgNewEmbedRef(). The *Support* column indicates which OpenPaige platform supports the data type. All the items listed are supported to some extent; *i.e.*, none of them require a callback function to render a default behavior.

NOTE: *If the data type is not listed, there is no current OpenPaige support for the type.* (You can, of course, support your own using the callback function.)

Table 5. embed_ref **data types**

Data Type	**Support**	*item_data **parameter**	**modifier**	**Note(s)**
embed_rectangle	all platforms	rectangle_ptr	pen size (pixels)	
embed_oval	all platforms	rectangle_ptr	pen size (pixels)	
embed_roundrectangle	all platforms	rectangle_ptr	round corner + pen size	1
embed_control	Macintosh only	ControlHandle	not used	2
embed_polygon	all platforms	memory_ref of polygon	pen size (pixels)	3
embed_mac_pict	Macintosh only	PicHandle	not used	
embed_mac_vm_pict	Macintosh only	memory_ref of Pict data	not used	7
embed_meta_file	Windows only	metafile_ptr	not used	4
embed_alternate_char	all platforms	cstring (any size)	not used	5
embed_user_data	all platforms (limited)	pointer to data	data size (bytes)	6
embed_dynamic_string	all platforms	cstring (any size)	max size (bytes)	
embed_user_box	all platforms (limited)	rectangle_ptr	pen size (pixels)	1

-

- embed_roundrectangle – the low-order word of modifier is the pen size; the high-order word is the "rounded corner" value, e.g. FrameRoundRect(rect, value, value).
- embed_user_box – The default behavior for embed_user_box is identical to embed_rectangle. To modify the default behaviour, use the callback function.
- ControlHandle – is detached from any Window before it gets inserted.
- data – is a memory_ref that *must* contain the following structure:

```
typedef struct
{
  short width;      // Width of polygon in pixels
  short height;     // Height of polygon in pixels
  short rsrv;       // (Here for future enhancement)
  short num_points;   // Number of points that follow
  co_ordinate points[1];  // One or more points for the drawing
}
pg_poly_rec, PG_FAR *pg_poly_ptr;
```

- width and height - members should contain the width and height of the bounding area of the polygon. The num_points member should contain the number of connecting points in the points[] member array.
- points - are represented by a series of co_ordinate pairs; the first pair is a line, the second pair is another line, etc.
  1. The points array must therefore be in PAIRS.
  2. A Windows meta file must be represented by the following structure (pointed to by the item_data parameter);

-

```
struct metafile_struct
{
  long metafile;    // Metafile data (HANDLE if Windows)
  long mapping_mode;  // Mapping mode (Windows only)
  short x_ext;    // Original X-extent
  short y_ext;    // Original Y-extent
  rectangle bounds; // Source bounding rect
};
```

-

- metafile - The meta file HANDLE should be in the metafile member; the mapping mode for the meta file should be in mapping_mode. For most meta files the mapping mode is MM_ANISOTROPIC.
- x_ext and y_ext - Members should contain the mapping mode-specific *X* and *Y* extents, respectively. You can also set these to zero (in which case the default width and height of the meta file will be used, taken from the bounds member). Most often, the mapping_mode and the *x* and *y* extents are taken from clipboard information.
- bounds - Member defines the meta file's dimensions in screen coördinates.

  The metafile HANDLE should be in the metafile member; the bounds member must define the bounding area of the metafile (the enclosing rectangle as the metafile was recorded).
- embed_alternate_char and embed_dynamic_string - draw a whole string to represent a single "character".

The embed_dynamic_string, however, can be dynamically altered (changed or "swapped" with a different string) in the callback function for display and character measuring purposes.

- embed_alternate_char and embed_dynamic_string - are treated as a single character and will therefore not wrap or word break in the middle.
- embed_user_data - the item is considered custom (generally handled by your callback function), but OpenPaige will save and retrieve your data automatically when saving to files. The data is assumed to be a contiguous byte stream in *item_data.
- item_data for embed_mac_vm_pict - must be a memory_ref containing the data from a PicHandle. This type behaves exactly the same as embed_mac_pict except that the memory_ref provides virtual memory to the picture.

28.5 The Callback Function

Any custom user type embed_ref, embed_refs that are not supported, or items that require modification(s) to the default behaviour will require a callback function. The callback function is a pointer to some code (that you write) that gets called for a number of occurrences:

```
PG_PASCAL (long) EmbedCallback (paige_rec_ptr pg, pg_embed_ptr embed_ptr,
long embed_type, short command, long user_refcon, long param1, long param2);
```

Each embed_ref you have inserted can have its own callback function (or they can all share the same callback if you so choose). The callback is set by passing the function pointer to the "callback" parameter in pgInsertEmbedRef().

NOTE: For *Windows 3.1*, you need to set a callback function that has been created with MakeProcInstance().

Upon entry, pg is the OpenPaige record structure that owns the embed_ref; the embed_ptr parameter is a pointer to the embed_ref record structure *infra*, and embed_type is the data type (same one you gave to pgNewEmbedRef() when it was initially created).

- command - indicates why the function is being called, and param1/param2 will contain different values depending on what value is in command (see "Command Messages” *infra*).

- user_refcon - will be whatever value you passed to pgInsertEmbedRef() in the callback_refcon parameter except for EMBED_READ_DATA and EMBED_WRITE_DATA, in which case the user_refcon will be the original user_refcon value given to pgNewEmbedRef().

Command Messages

When the callback function is called, the value in command will be one of the values below. Depending on the command, param1 and param2 contain additional data. In each case, the embed_ptr will point to the embed_ref structure (see "The Embed Record").

- EMBED_INIT - occurs during a pgReadDoc() function (file read). The purpose of this command is to initialize an embed_ref that has been read from a file (typically, to set a callback function specific to the associated text style). See pgInitEmbedProcs regarding the first pgReadDoc callback function.

  Upon entry, param1 indicates the number of times EMBED_INIT has been sent to the callback function during pgReadDoc for this particular embed_ref. (Since the same embed_ref can be shared by many places in the text, your initialisation code might want to know this information so the embed_ref data is initialised only once). On the first callback for this embed_ref, param1 is zero.

  The param2 will be a style_info pointer that is associated to the embed_ref.

  The callback function result is ignored.

- EMBED_DRAW - occurs when the embed_ref should be drawn.
- param1 - Upon entry, this is a rectangle_ptr (an OpenPaige rectangle) that defines the exact drawing bounds of the embedded item (which includes scrolled position and scaling).
- param2 - is a draw_points_ptr containing additional information for drawing (see draw_points_ptr in OpenPaige manual and/or in Paige.h).

  The callback function result is ignored.

  NOTE: On *Windows*, the device context that you should draw to can be obtained as follows:

  ```
  HDC hdc;
  hdc = (HDC)pg → globals → current_port.machine_ref;
  ```

Do not assume that OpenPaige is drawing to the current window (it can be drawing to a bitmap DC or a printer DC, etc.). When the callback is called, the above code is *guaranteed* to return a valid device context to use for drawing.

NOTE: On *Macintosh*, the GrafPort you should draw to is set as the current port before the callback is called. Do not assume that drawing will occur to the pg_ref window (it can occur to an offscreen bitmap port).

- EMBED_MEASURE – occurs when OpenPaige wants to know the character width(s) of the embedded item.

  Upon entry, param1 is a pg_embed_measure_ptr and param2 is not used. (See “The Measure Record”).

  NOTE: This callback is only used to obtain the object's width. Its height must be initialized either before inserting the embed_ref or in response to EMBED_VMEASURE.

  Before returning from this function, you should set the embed_ptr -> width to the embed_ref's width, in pixels.

  NOTE: OpenPaige will determine the width automatically for embed types that are fully supported (requiring no callback).

  The function result from the callback function is ignored.

- EMBED_VMEASURE – occurs when OpenPaige wants to know the height of the embedded item.

  Upon entry, param1 is the style_info_ptr that "owns" the embed_ref and param2 is not used.

  Before returning from this function you should set the embed_ptr → height to the embed_ref's height, in pixels.

  NOTE: OpenPaige will determine the height automatically for embed types that are fully supported (requiring no callback).

  The function result from the callback function is ignored.

- EMBED_SWAP – occurs (only) when the item type is embed_dynamic_str. The callback function is used to "swap out" (substitute) a string of bytes with something else. For example, this type of embed_ref can be used to indicate a date or time, page number, or footnote, all of which might change dynamically.

  Upon entry, param1 is a pointer to the existing string (or empty buffer), and param2 is a long value indicating the maximum number of bytes that can be put into the buffer. The job of the callback function, in this case, is to fill the buffer pointed to by param1.

The function result of this callback must be the number of bytes placed in *param1 (i.e., the string length).

- EMBED_CURSOR - occurs when the mouse is on top of the embed_ref. The purpose of the callback is to let your application change the cursor, if desired.

  NOTE: This callback will never occur unless you call pgPtInEmbed().

  Upon entry, param1 will be a co_ordinate_ptr for the mouse point, and param2 is a rectangle_ptr to the enclosing bounds of the embed_ref.

  The callback function result is ignored.

- EMBED_MOUSEDOWN – occurs during a pgDragSelect() with mouse_down verb, when an embed_ref has been clicked.

  Upon entry, param1 is a pointer to a pg_embed_click record with additional info (see "Click Record" *infra*).

  The function result from the callback should be any non-zero value if you want to continue tracking the embed_ref like a control button, otherwise return zero. (By tracking like a control button is meant that OpenPaige will not try to drag-select surrounding characters, rather subsequent mouse movements will be passed to your callback function with EMBED_MOUSEMOVE and EMBED_MOUSEUP commands.

  CAUTION: To achieve a "push-button control" effect, mouse-click behaviour may not appear to work correctly unless you include EMBED_CONTROL_FLAG in the flags parameter for pgNewEmbedRef(). See "Acting Like a Control" in this chapter.

- EMBED_MOUSEMOVE – occurs during a pgDragSelect() with mouse_move given as the verb, **if** you returned non-zero from the previous callback for EMBED_MOUSEDOWN. The parameters are identical to EMBED_MOUSEDOWN.

  CAUTION: To achieve a "push-button control" effect, mouse-click behavior may not appear to work correctly unless you include EMBED_CONTROL_FLAG in the flags parameter for pgNewEmbedRef(). See "Acting Like a Control".

- EMBED_MOUSEUP – occurs during a pgDragSelect() with mouse_up given as the verb, **if** you returned non-zero from the previous callback for EMBED_MOUSEMOVE.

  The parameters are identical to EMBED_MOUSEDOWN.

- EMBED_DOUBLECLICK – occurs during a pgDragSelect() with mouse_down and modifier containing WORD_MOD_BIT (“double click"). The parameters are identical to the callback for EMBED_MOUSEDOWN.
- EMBED_DESTROY – occurs when the embed_ref is about to be disposed. Upon entry, param1 and param2 are not used. The function result is ignored.

  NOTE: You will not receive this message if you dispose your own embed_ref (e.g., made a call to pgEmbedDispose()). The only time you will receive this callback command is when OpenPaige disposes the embed_ref; this happens when the last occurrence an embed_ref has been deleted (and you gave FALSE for keep_around when the embed_ref was created).

  CAUTIONS:

  1. If you have created your own data and have placed it in embed_ptr → data, you must first dispose it (if appropriate) then set that member to NULL. However, do not dispose the data if you gave that data to item_data and the data type is embed_user_data.
  2. If the embed_ref data is not supported (i.e. fully custom), *do not* call the default callback function when command is EMBED_DESTROY.
  3. Do not dispose the embed_ref itself. You should only dispose memory structures that you created.
- EMBED_COPY – Occurs when a style_info containing an embed_ref is duplicated.

  This callback only occurs for embed_refs that contain NOT_SHARED_FLAG (see “Special Cases”).

  Upon entry, param1 and param2 are not used. The intended purpose of EMBED_COPY is to duplicate any memory structures you might have stored in the embed_ref.
- EMBED_WRITE_DATA – Occurs when an embed_ref is saved during pgSaveDoc().

  When OpenPaige saves an embed_ref to a file, all the "default" information is saved before this command is given to your callback. The intended purpose of EMBED_WRITE_DATA is for you to prepare any additional data that needs to be written to the file; this same data will then be retrieved when the file is read and issued to your callback as EMBED_READ_DATA.

Essentially, when you get the EMBED_WRITE_DATA command, you don't need to do anything unless there is extra data you have stored in the embed_ref that OpenPaige won't know about; all the other embed_ref contents are saved otherwise.

Upon entry, param1 is a memory_ref of zero byte size, and param2 is not used. To save any additional data associated to the embed_ref, insert the bytes into this memory_ref. When the function returns, OpenPaige will write GetMemorySize(param1) bytes to the file; later when the file is opened, these same bytes will be read from the file and given to your callback with EMBED_READ_DATA as the command.

When the callback returns, if the memory size of (memory_ref) param1 is zero, no extra data is saved.

The function result from the callback is ignored.

NOTE: The EMBED_WRITE_DATA callback will only occur once for each embed_ref. In other words, if multiple "shared" copies of the embed_ref exist in the document, you will only be asked to save extra data once.

- EMBED_READ_DATA - Occurs when an embed_ref is read from a file during pgReadDoc(). This command will always get sent for every embed_ref that is read even if you saved no extra data (from EMBED_WRITE_DATA).

  Upon entry, param1 is a pointer to the same data bytes, if any, that you saved when the command was EMBED_WRITE_DATA, and param2 is the byte count.

  The function result from this callback is ignored.

  NOTE: The EMBED_READ_DATA callback will only occur once for each embed_ref. In other words, if multiple "shared" copies of the embed_ref exist in the document, you will only be asked to process the data once.

28.6 Default Callback

You should always call the default function, pgDefaultEmbedCallback, from your callback code if you do not handle the command (some exceptions-see caution *infra*).

For example, you might create a callback function only for the purpose of changing the cursor when the mouse is over the embed_ref. In this case, you would not want to handle any other command; rather, you want the default handling. To do so, make a call to pgDefaultEmbedCallback().

```
pascal long MyCallback (paige_rec_ptr pg, pg_embed_ptr embed_ptr, long
embed_type, short command, long callback_refcon, long param1, long param2);
{
  if (i_dont_want_to_handle)
    return pgDefaultEmbedCallback(pg, embed_ptr, embed_type, command,
callback_refcon, param1, param2);
  // else handle the command
}
```

CAUTION: Never call `pgDefaultEmbedCallback()` for `EMBED_DESTROY` if you have placed your own data in `embed_ptr → data`. If you have not directly altered the `data` field in any way, it is OK to call the default.

28.7 Acting Like a Control

In many cases, user-defined `embed_refs` need to act like a "control". For example, once the user clicks in the `embed_ref`, the mouse needs to be "tracked" as if the `embed_ref` were a push-button control.

To make your embedded item behave this way, include the following value in the `flags` parameter for `pgNewEmbedRef()`:

```
#define EMBED_CONTROL_FLAG  0x00100000    // Acts like a control
```

Creating `embed_ref` that acts like a "control"

```
/* This sample was taken from the Mac demo but can apply to ANY embed_ref
inserted for any platform: */

static void insert_embedded_pict(doc_rec *doc, PicHandle picture)
{
  embed_ref ref;

  ref = pgNewEmbedRef(&mem_globals, embed_mac_pict, (void*) picture, 0,
EMBED_CONTROL_FLAG, 0, 0, FALSE);
  pgInsertEmbedRef(doc → pg, ref, CURRENT_POSITION, 0, NULL, 0, best_way);
}
```

28.8 The Embed Record

```
struct pg_embed_rec {
  short       version;      // Version of embedded
  short       reserved;     // reserved
  long        type;       // Type of item embedded
  long        width;        // Drawing width, in pixels
  long        minimum_width;    // Minimum width
  long        height;       // Drawing height, in pixels
  long        descent;      // Distance bottom is below text line bottom
  long        draw_flags;     // Drawing attributes (see table below)
  long        modifier;     // Extra data for certain objects
  long        empty_width;    // Width of item when empty (applies to dynamic
types).
  long        alignment_flags;  // Alignment (subrefs only)
  short       top_extra;      // Extra "whitespace" at the top
  short       bot_extra;      // Extra "whitespace" at the bottom
  short       left_extra;     // Extra "whitespace" at the left edge
  short       right_extra;    // Extra "whitespace" at the right edge
  void PG_FAR     *data;        // The item's data, if appropriate
  memory_ref      embed_represent;  // Optional embed that represents
unsupported type
  memory_ref      rtf_text_data;    // Original preamble text from
unsupported RTF import
  union
  {
    pg_pic_embed  pict_data;      // Special picture data
    pg_horiz_line line_info;      // Special line data
    pg_date     date_info;      // Date info
    pg_time     time_info;      // Time info
    pg_char     alt_data[ALT_SIZE + BOM_HEADER];  // Alternate data (for
ALT_SIZE or less chars)
    pg_char     book_data[BOOKMARK_SIZE];// Bookmark data
  } uu;
  long        border_info;    // Border control (revised for 3.01)
  long        border_color;   // The border color
  long        shading;      // Background color shading
  style_info_ptr    style;        // The style associated to this item
  long        user_refcon;    // What app put with this embed
  long        user_data;      // App can also use this field
  long        style_refcon;   // Refcon saved in styles
  long        lowlevel_index;   // Used by low level functions
  long        used_ctr;     // Count of shared access (maintained internally)
};
```

The above structure is what all `embed_refs` look like internally. Most of the fields are maintained by OpenPaige and you must neither alter them nor assume they are valid at any time, except as further elucidated.

The following fields can be altered (and in some cases need to be initialised) by your application:

- width, height - Define the width and height of the object. The width member gets set in the callback function when the command is EMBED_MEASURE; if the item type is unsupported or custom, the height member must be initialised before inserting the embed_ref.
- minimum_width - Define the minimum width (smallest size) allowed for the embed_ref. Your application needs to set this; otherwise, it is zero.
- descent - Defines the distance the object should draw below the text baseline. You may alter this value for a descent other than the default.
- top_extra through right_extra - Define optional extra white space on the top, left, bottom and right sides of the object. The default for each of these members is zero; if you want something else, you should modify them before inserting the embed_ref.
- data - You may place whatever data your application requires into this member. However, please observe the following cautions:
  1. Do NOT alter the data field directly for embed_user_data type or any of the supported types listed above.
  2. If you place anything directly in the data member, do not call the default callback function when the command is EMBED_DESTROY.
  3. You must dispose your own data, if appropriate. Letting OpenPaige handle it as a default can result in a crash.

28.9 The Measure Record

```
typedef struct
{
  style_walk_ptr  walker;        // Style information
  pg_char_ptr     text;          // "Text" pointer
  long            text_size;     // "Text" size, in bytes
  pg_short_t      slop;          // Extra amount for full-justify
  long PG_FAR     *positions;    // Width locations of "text" bytes
  short PG_FAR    *types;        // Character types
  short           scale_verb;    // Whether or not to scale results
  short           measure_verb;  // Measurement verb
  long            current_offset; // Current offset to measure
  short           call_order;    // The call order
};

pg_embed_measure, PG_FAR *pg_embed_measure_ptr;
```

The measure record is passed as a pointer in `param1` for `EMBED_MEASURE` commands. Usually, you won't need to use any of these values, but they are listed here for the sake of clarity.

28.10 The Click Record

```
typedef struct
{
  t_select_ptr first_select;  // Start of selection
  t_select_ptr last_select; // End of selection
  co_ordinate point;      // Mouse point
  rectangle bounds;     // Frame around the item
  short modifiers;      // Modifiers from pgDragSelect
}
pg_embed_click, PG_FAR *pg_embed_click_ptr;
```

A pointer to the above structure is provided in `param1` for `EMBED_MOUSEDOWN`, `EMBED_MOUSEMOVE`, `EMBED_MOUSEUP`, and `EMBED_DOUBLECLICK` commands.

The `first_select` and `last_select` members represent the current beginning and ending selection point(s) of the drag-select. (For `EMBED_MOUSEDOWN` these are typically the same). The `point` and `modifiers` members will contain the `co_ordinate` value and modifiers given to `pgDragSelect()`, respectively. The `bounds` member will contain the WYSIWYG bounding rectangle of the `embed_ref` that is being clicked.

28.11 Special Cases

The `flags` parameter for `pgNewEmbedRef` has been briefly mentioned earlier. Normally, the value for `flags` should be zero. There are two possible bit settings you can provide for this parameter for specific situations, as follows:

- `NOT_SHARED_FLAG` - If this bit is set, the `embed_ref`'s data is always duplicated for any copy/paste operation. Normally, when an `embed_ref` is copied in the text stream, its contents are not actually copied; rather, only a pointer to its reference is copied. In essence, only one "real" `embed_ref` exists in the document even though there could be many occurrences of the reference throughout the text stream.

However, this may be undesirable in situations where copied embed_ref(s) must be unique (as opposed to pointer "clones" of the original). If this is the case, set NOT_SHARED_FLAG.

- EMBED_CONTROL_FLAG – If this bit is set, the embed_ref responds like a control (such as a button). Otherwise, the embed_ref acts like a character. Note that the only significant difference between a *control* and a *character* is the way OpenPaige highlights the embed_ref when it is clicked. As a "control", the entire embed_ref is selected from a single click.
- NO_FORCED_IDENTITY – If this bit is set, when the embed_ref is inserted, OpenPaige scans the document for any embed_ref that matches its type, its width and height, and its user_refcon value. If such a match is found, the (new) embed_ref is discarded and the matching (older) embed_ref is used in its place.

  The purpose of NO_FORCED_IDENTITY is to minimise the amount of memory used by repeated insertions of the same embed_ref type.

  For example, suppose your application is designed to insert a mathematical symbol (that can't otherwise be represented by a text character). To achieve this, an embed_ref is created to draw the symbol and it is inserted in many different places. Normally (without NO_FORCED_IDENTITY set), OpenPaige will create a unique style_info record and embed_ref for every insertion. If NO_FORCED_IDENTITY is set, however, only one record of this symbol would exist even though it may be inserted and display in many different text positions.

28.12 Tips and Tricks

For all user items and custom or non-supported embed_refs, you must initialise at least the height of the embed_ref before you insert it. Otherwise, OpenPaige has no idea how tall the object is (but it will get the object's width from the callback function). To initialise the height, do the following:

```
pg_embed_ptr embedPtr;
embedPtr = UseMemory(ref); // where "ref" is the newly created embed_ref
embedPtr -> height = HeightOfMyItem;
UnuseMemory(ref);
```

NOTE: For supported types that require no special callback function, you do not need to initialise the height - OpenPaige initialises it for you. You would only need to change the height if you wanted something other than the default.

If you need to create a custom embed_ref that requires a block of data larger than a long word, the recommended choice is to use embed_user_data because OpenPaige will at least store the data, present a pointer to it for your callback, and save/read the data for files. This minimal support assumes that nothing in your data stream needs to be de-referenced, *i.e.*, if you have pointers inside of pointers, OpenPaige has no way of knowing how to save them.

To create an embed_ref of type embed_user_data, pass a pointer to the data in item_data and the byte count in modifier; OpenPaige will then make a copy of the data (so you can then dispose the pointer, etc.).

CAUTION: If you let OpenPaige store the data, you should neither alter nor dispose it. Let the default callback function handle the dispose (see "The Callback Function”).

For all embed_refs (both supported items and custom/user items), OpenPaige normally keeps only one embed_ref around and creates pointers to the original when the text is copied/pasted. If this default behavior is unworkable for any particular feature, pass NOT_SHARED_FLAG for the flags field in pgNewEmbedRef() (see “Special Cases”).

28.13 Applying to Existing Text

In certain cases, you might want to apply an embed_ref to existing characters (as a "style") as opposed to inserting a new "character" by itself. One example of this would be to support hypertext links that apply to existing key words in the document; for such a feature, you will probably want to connect an embed_ref to an existing group of characters instead of inserting a new one. If this is the case, you should use the following function instead of pgInsertEmbedRef:

```
void pgSetEmbedRef (pg_ref pg, embed_ref ref, select_pair_ptr selection,
short stylesheet_option, embed_callback callback, long callback_refcon, short
draw_mode);
```

This function's parameters are identical to pgInsertEmbedRef(), except the embed_ref is applied to existing text as a style. Hence, the selection parameter can be a pointer to a range of characters, or NULL if you want to apply the reference to the current text selection.

NOTE: Relying on the default behaviour of the embed_ref in this case can render the text "invisible". This is because the text within the specified selection becomes literally a custom style and the standard text drawing function within OpenPaige will no longer get called for those characters.

You can handle this by setting a callback function that responds to EMBED_DRAW, at which time you can call the standard text drawing function (see "The Callback Function").

28.14 Non-sharing Embeds

By default, multiple occurrences of the same embed_ref are shared. For example, if you created a single embed_ref and inserted it as a character, subsequent copy/paste operations might duplicate the reference several times; yet, only one embed_ref is maintained by OpenPaige. Each copy is merely a pointer to the same (shared) memory.

In special cases, however, an application might need to force unique occurrences for each copy. For example, suppose the user is allowed to edit an embedded picture (such as changing its size or content). If multiple copies exist in the text, changing one of them would change the appearance of all—which may not be a desirable feature.

The work-around is to pass the following value in the flags parameter when pgNewEmbedRef() is called:

```
#define NOT_SHARED_FLAG0x00080000 // Embed_ref not shared
```

Setting flags to this value tells OpenPaige that for each copy/paste operation, the embed_ref needs to be newly created. Hence, each copy will be a unique reference and not shared.

28.15 File Saving

Unless you call the function below immediately after calling pgSaveDoc, embed_refs contained in a document *do not* automatically get saved to an OpenPaige file:

```
pg_error pgSaveAllEmbedRefs ( $\mathrm{pg}$ _ref pg, file_io_proc io_proc,
file_io_proc data_io_proc, long PG_FAR *file_position, file_ref filemap);
```

This function writes all embed_refs in pg to the file specified. The pg, io_proc, file_position and filemap are the same parameters you just gave to pgSaveDoc() for pg, write_proc, file_position, and filemap, respectively. The data_io_proc should be NULL (it is only used in very specialised cases).

This function is safe to call even if there are no embed_refs contained in pg (if that is the case, nothing gets written to the file).

The reason this function is separate, as opposed to OpenPaige saving embed_refs automatically, is that some OpenPaige developers will not be using the embed_ref extension, so the required library to handle this feature might not exist in every application.

For each embed_ref that is saved, the callback function will be called with EMBED_WRITE_DATA as the command.

The pgSaveAllEmbedRefs is to be used to save embed_refs already existing in pg; if you have embed_refs around that are not inserted anywhere, you need to save them discretely using the following function:

```
pg_error pgSaveEmbedRef (pg_ref pg, embed_ref ref, long element_info,
file_io_proc io_proc, file_io_proc data_io_proc, long PG_FAR *file_position,
file_ref filemap);
```

The above function is similar to pgSaveAllEmbedRefs except a single embed_ref is saved to the file. The element_info value can be anything, and that value is returned to a read handler when the data is read later. If this function is successful, zero (NO_ERROR) is returned.

NOTE: You do not need to call this function unless you need to save an embed_ref that you have kept around that isn't inserted into a document.

28.16 File Reading

Since OpenPaige can not make the assumption that the embed_ref extension library is available in all applications, you must tell the file I/O mechanism that an OpenPaige file being read might contain embed_refs. You do so by calling the following function at least once before calling pgReadDoc:

```
void pgInitEmbedProcs (pg_globals_ptr globals, embed_callback callback,
app_init_read init_proc);
```

This initialises the embed_ref read handler so it can process any embed_ref within the text stream during pgReadDoc. You only need to call this function once, some time after pgInit and before the first pgReadDoc.

The callback parameter should be a pointer to a callback function that you want to set, as the default callback, for all embed_refs that are read. This function should either be NULL (for no callback) or a pointer to the same kind of function used for callback when inserting an embed_ref. The reason you need to provide this parameter when reading a file is the newly created embed_refs won't have callback functions (hence there would be no way to examine the incoming data). Additionally, OpenPaige sets the callback given in pgInitEmbedProcs to become the callback for all the embed_refs read from the file.

An embed_ref is read from the file and processed as follows:

1. The embed_ref is created and the default contents are read;
2. The callback function is called with EMBED_READ_DATA, giving your application a chance to append additional data that might have been saved;
3. OpenPaige walks through all the style_info records and attaches the embed_ref to all appropriate elements; for each style_info that contains the embed_ref, the callback is called once more with EMBED_INIT.

The init_proc is an optional function pointer that will be called after an embed_ref is retrieved during file reading; the primary purpose for this function is to initialise an embed_ref that is not attached to the document. Normally you won't need to use this callback function so just pass NULL; but if for some reason you have saved an embed_ref discretely (using pgSaveEmbedRef()) and it is not applied to any character(s), the init_proc might be the only way you can get called back to initialise the embed_ref data.

The init_proc gets called immediately after an embed_ref has been read from a file:

```
PG_PASCAL (void) init_read(paige_rec_ptr pg, memory_ref ref);
```

When init_read is called, the newly read embed_ref will be given in ref.

28.17 Additional Support

Checking the Cursor

```
embed_ref pgPtInEmbed (pg_ref pg, co_ordinate_ptr point, long PG_FAR
*ext_offset, style_info_ptr associated_style, pg_boolean do_callback);
```

This function returns an embed_ref, if any, that contains point. If no embed_ref contains point, MEM_NULL (zero) is returned.

If text_offset is non-NULL and an embed_ref containing point is found, *text_offset is set to the text position for that ref. Likewise, if associated_style is non-NULL, then *associated_style is initialised to the style_info for that ref.

If do_callback is TRUE, the callback function for the embed_ref is called with EMBED_CURSOR command when and if the point is contained in an embed_ref. (See "The Callback Function" and EMBED_CURSOR command in "Command Messages").

```
embed_ref pgGetEmbedJustClicked (pg_ref pg, long drag_select_result);
```

Returns the embed_ref that was clicked during the last call to pgDragSelect. If no embed_ref was clicked from the last pgDragSelect, the function returns MEM_NULL (zero).

The drag_select_result should be whatever value was returned from the last call to pgDragSelect (which is actually how pgGetEmbedJustClicked knows which embed_ref was clicked).

Finding/Searching

```
embed_ref pgFindNextEmbed (pg_ref pg, long PG_FAR *text_position, long
match_refcon, long AND_refcon);
```

This function does a search through all the embed_refs in pg and returns the first one that matches the criteria specified. The search begins at *text_position. If an embed_ref is found, it is returned and *text_position is set to the text offset for that ref. Otherwise, MEM_NULL is returned and *text_position is set to the end of the document.

For example, to search for an embed_ref starting at the document's beginning, set a long to 0 and pass a pointer to it as text_position.

Essentially, the function searches for the first occurrence of an embed_ref whose callback_refcon (the value given to pgInsertEmbedRef) matches match_refcon; the callback refcon value in the embed_ref is first ANDed with AND_refcon, then compared to match_refcon. If the comparison is equal, that embed_ref is considered a true match and it is returned.

For example, if you wanted to find the next embed_ref that had a 1 set for the low-order bit of the callback refcon, you would pass 1 for both match_refcon and AND_refcon.

If you simply want to find the first occurrence of any embed_ref, pass 0 for both match_refcon and AND_refcon.

To find an exact, specific embed_ref (per value in callback refcon), pass that exact refcon value in match_refcon and -1 for AND_refcon.

```
embed_ref pgGetExistingEmbed (pg_ref pg, long user_refcon);
```

Returns the embed_ref currently in pg, if any, that contains user_refcon. The user_refcon being searched for is the same value given to pgNewEmbedRef originally.

NOTE: The user_refcon is the value that was given to pgNewEmbedRef(), which can be different to the callback refcon.

If one is not found that matches user_refcon, this function returns MEM_NULL.

```
long pgNumEmbeds (pg_ref pg, select_pair_ptr selection);
```

Returns the total number of embed_refs contained in the specified selection of pg. If selection is a null pointer, the current selection is used.

Once you know how many embed_refs are present in the specified range of text, you can access individual occurrences using pgGetIndEmbed (*infra*).

```
embed_ref pgGetIndEmbed (pg_ref pg, select_pair_ptr selection, long index,
long PG_FAR *text_position, style_info_ptr associated_style);
```

Returns the *n*th embed_ref within the specified selection. If selection is a null pointer, the current selection is used.

If text_position *is not* a null pointer, then *text_position gets set to the (zero-indexed) text position of the embed_ref.

If associated_style is non-NULL, the style_info is initialised to the style the embed_ref is attached.

If the index embed_ref does not exist, the function returns MEM_NULL (and neither *text_position nor *associated_style is set to anything).

NOTE: The index value is *one-based*, i.e. the first embed_ref is 1 (not zero).

28.18 Miscellaneous Support

```
long pgGetEmbedBounds (pg_ref pg, long index, select_pair_ptr index_range,
rectangle_ptr bounds, short PG_FAR *vertical_pos, co_ordinate_ptr
screen_extra);
```

This function returns the bounding dimensions of the embed_ref represented by index within the index_range; if index_range is NULL, the whole document is used.

The index is zero-based (first embed_ref in the document is zero). You can determine how many embed_ref exist by calling pgNumEmbeds().

This function returns the text position of the embed_ref (what character it applies to relative to the 0th char); the bounding rectangle of the ref is returned in *bounds and the *vertical_pos parameter returns the item's descent value (distance from baseline to bottom).

NOTE: Any or all of these parameters can be NULL if you don't need the information.

The rectangle returned in *bounds will be the enclosing box of the embed_ref *not* scrolled, i.e. where it would be on the screen, were pg's scroll position (0, 0). If screen_extra is non-NULL then it will be set to the amount of pixels you would need to offset the bounding rectangle in order to obtain the physical location of its bounds. Hence, if you offset *bounds by screen_extra → h and screen_extra → v you obtain the WYSIWYG rectangle.

```
long pgEmbedStyleToIndex(pg_ref pg, style_info_ptr embed_style);
```

Returns the index value of the embed_ref attached to embed_style, if any. This function is useful for obtaining an "index number" for an embed_ref where only the style_info is known. If no embed_ref exists for embed_style, zero is returned; otherwise assume the function result is the related index.

This index value can then be used for functions that require it such as pgGetEmbedBounds, pgGetIndEmbed, etc.

```
void pgSetEmbedBounds(pg_ref pg, long index, select_pair_ptr index_range,
rectangle_ptr bounds, short PG_FAR *vertical_pos, short draw_mode);
```

This function changes the bounding dimensions and/or the baseline position (descent) of an embed_ref within a document.

The index parameter specifies which embed_ref to change (one-indexed), and index_range indicates the range of text to consider. If index_range is NULL the current selection is used.

For example, if the current selection contained two embed_refs, an index of 1 would indicate the first embed_ref within that selection and a 2 would indicate the second embed_ref. The physical order of embed_refs is the order in which they appear in the text (not necessarily the order they were inserted).

The bounds rectangle indicates the embed_ref's new width and height. Note that width and height are taken from the rectangle dimensions—the physical top-left location of the embedded object is not altered. If bounds is NULL the embed_ref dimensions remain unaltered.

The vert_pos parameter should point to a value that indicates the amount of descent, in pixels, that the embed_ref should be drawn relative to the text baseline. This is a positive value, i.e. a value of 3 will cause the embed_ref to draw three pixels below the text baseline.

The vert_pos parameter can also be NULL, in which case the object's descent remains unchanged.

If draw_mode is non-zero, the text in pg (including the changes to the embed_ref) is redrawn.

28.19 Undo Support

You can prepare for undoing an embed_ref insertion by calling pgPrepareUndo(), passing undo_embed_insert as the undo verb. You should do this just before inserting an embed_ref.

Otherwise, there is no specific undo support required for embed_ref (because after they are inserted, all the normal undo operations will work -undo for Cut, Paste, format changes, etc.).

28.20 Relationship to Style Info

OpenPaige stores embed_refs directly in the style_info record. The following style_info fields contain embed_ref information (from style_info struct):

```
long embed_entry;       /* App callback rfunction for embed_refs */
long embed_style_refcon;  /* Used by embed object extension */
long embed_refcon;      /* Used by embedded object extension */
long embed_id;        /* Used by embedded object extension */
memory_ref embed_object;  /* Used by embedded object extension */
```

The callback function is stored in embed_entry; embed_style_refcon is the callback refcon and embed_refcon is the user refcon (see refcon values for pgNewEmbedRef and for pgInsertEmbedRef).

The embed_id will contain a unique ID number generated by OpenPaige; this value has no direct meaning except that it is created to keep style_infos and embed_refs from running together.

The embed_ref itself is in embed_object.

28.21 Examples

Windows Example

The following is an example of inserting a metafile as a "character" into a pg_ref. It also shows how to prepare for an Undo.

```
/* This function embeds a meta file into the text. The x and y extents are
given in x_ext and y_ext. Note, the x and y extents should be given in device
units. The user_refcon param is whatever you want it to be for application
reference. The callback param will become the embed callback, or NULL if you
want to use the default. */

void InsertMetafile (pg_ref pgRef, HMETAFILE meta, int x_ext, int y_ext, long
user_refcon, embed_callback callback)
{
  metafile_struct metafile;
  embed_ref ref;
  void PG_FAR *the_data;

  /* It is a good idea to fill struct with zeros for future compatibility. */

  memset(&objData, '\0', sizeof(PAIGEOBJECTSTRUCT));
  metafile. metafile = (long)meta;
  metafile.bounds.top_left.h = metafile.bounds.top_left.v;
  metafile.bounds.bot_right.h = x_ext;
  metafile.bounds.bot_right.v = y_ext;
  metafile.mapping_mode = MM_ANISOTROPIC;
  metafile.x_ext = x_ext;
  metafile.y_ext = y_ext;
  the_data = (void PG_FAR *) &metafile;
  ref = pgNewEmbedRef(&mem_globals, embed_meta_file, the_data, 0, 0, 0,
user_refcon, FALSE);
  pgInsertEmbedRef(pgRef, ref, CURRENT_POSITION, 0, callback, 0, best_way);
};
```

Macintosh Example

```
/* The following example shows inserting a PicHandle as a "character". The
picture's descent (distance below baseline) is 20% of its height. We also
prepare for an Undo. */

void InsertPicture (pg_ref pg, PicHandle picture)
{
  embed_ref ref;
  undo_ref undoStuff;

  ref = pgNewEmbedRef(&mem_globals, embed_mac_pict, (void*) picture, 0, 0,
(pg_fixed) 20<<16, 0, FALSE);

  undoStuff = pgPrepareUndo(pg, undo_embed_insert, NULL);
  pgInsertEmbedRef(pg, ref, CURRENT_POSITION, 0, NULL, 0, best_way);
}
```

A Custom `embed_ref`

This example shows how to create and manipulate a custom `embed_ref`. In this case we are creating a simple box for which we draw a frame, and we respond in some way if the user double-clicks in this box.

For purposes of demonstration, we also attach a data struct to the custom `embed_ref`. While this example doesn't do anything with that data, it shows how you would save and read your data to an OpenPaige file.

```
/* Insertion of a custom ref into a pg_ref "pg". Upon entry, width and height
define the dimensions of the box; data is a pointer to some arbitrary data
structure that gets attached to the ref (and eventually saved to the
OpenPaige file) and dataSize is the size of that data. The callbackProc param
is a pointer to our callback function (almost mandatory for any custom
embeds). The refCon value becomes the callback refcon. */

void makeCustomRef (pg_ref pg, short width, short height, char *data, long
dataSize, embed_callback callbackProc, long refCon)
{
  embed_ref ref;
  pg_embed_ptr embed_ptr;

  /* Create a custom ref, but if we specify embed_user_data then OpenPaige
will attach the data to the ref. */

  ref = pgNewEmbedRef(&mem_globals, embed_user_data, (void*) data, dataSize,
0, 0, 0, FALSE);
```

```
  /* The following code is vital for a "custom" user type since OpenPaige has
no idea how tall our embed item is, nor does it know how wide it is: */

  embed_ptr = UseMemory(ref); // Get the embed struct
  embed_ptr -> height = height;
  embed_ptr -> width = width;
  UnuseMemory(ref);

  // Insert the ref. (Also add pgPrepareUndo() here if desired).
  pgInsertEmbedRef(pg, ref, CURRENT_POSITION, 0, callBackProc, refCon,
best_way);
}

/* The following code is the callback function for the embed_ref. OpenPaige
calls this with various "messages". */

PG_PASCAL (long) callBackProc (paige_rec_ptr pg, pg_embed_ptr embed_ptr, long
embed_type, short command, long user_refcon, long param1, long param2)
{
  memory_ref specialData;
  Rect theBox;
  char *extraBytes;
  long result = 0 ; // Default function result

  switch (command)
  {
    case EMBED_DRAW:
      // In this example we frame the box.
      // param1 is a rectangle_ptr of the box
      RectangleToRect((rectangle_ptr)param1, NULL, &theBox);
      FrameRect(&theBox);
      break;
    case EMBED_MOUSEDOWN:
    case EMBED_MOUSEMOVE:
    case EMBED_MOUSEUP:
    case EMBED_DOUBLECLICK:
      result = pgDefaultEmbedCallback(paige_rec_ptr pg, pg_embed_ptr
embed_ptr, long embed_type, short command, long user_refcon, long param1,
long param2);
      if (command == EMBED_DOUBLECLICK)
        HandleMyDoubleClick(pg, user_refcon);
        // The "HandleMyDoubleClick() is whatever...
        break;
    case EMBED_DESTROY:
      /* Important note: Since our embed_ref type is embed_user_data, we can
let OpenPaige dispose the data. However if we attached our own data directly
we would NOT call the standard callback, or we would crash! */
      result = pgDefaultEmbedCallback(paige_rec_ptr pg, pg_embed_ptr
embed_ptr, long embed_type, short command, long user_refcon, long param1,
long param2);
      break;
    case EMBED_WRITE_DATA:
      /* NOTE, since our embed type is embed_user_data, OpenPaige will save
that data automatically, so we don't need to do anything for this message.
But purely for the sake of demonstration we will save two extra bytes to the
file to show how it is done: */
      specialData = (memory_ref) param1;
      SetMemorySize(specialData, sizeof(char) * 2);
      extraBytes = UseMemory(specialData);
      extraBytes[0] = myCustomChar1;
      extraBytes[1] = myCustomChar2;
      UnuseMemory(specialData);
      break;
      /* NOTE, since our embed type is embed_user_data, OpenPaige will read
that data automatically, so we don't need to do anything for this message.
But purely for the sake of demonstration we will read the two extra bytes
from the file that we saved in EMBED_READ_DATA: */
      extraBytes = (char*) param1; // Pointer to data
      myCustomChar1 = extraBytes[0];
      myCustomChar2 = extraBytes[1];
      break;
    default:
      result = pgDefaultEmbedCallback(paige_rec_ptr pg, pg_embed_ptr
embed_ptr, long embed_type, short command, long user_refcon, long param1,
long param2);
      break;
    }
  return result;
}
```

29 MAIL MERGING

"Mail merging" is a feature in which specific portions of text can be temporarily swapped with text from other sources. We are referring to it as "mail merge" because this feature is typically used to substitute special embedded symbols or fields within a document with data from a database for form letters, mailing labels, statements, etc.

29.1 How merging works

In OpenPaige, merging is essentially a custom style. For more about custom styles in general see "Creating a simple custom style" and "Customizing OpenPaige". Specifically, the merge feature is accomplished as follows:

1. A merge "symbol" is simply a specific style (set by the application) which is applied to a portion of text. It differs from other styles simply by the existence of a merge_proc other than the default (see point 2 below). Otherwise, such "symbols" can be any kind of characters, words or phrases the application wishes to embed in the text stream to convey "merge fields".

For the sake of discussion, we shall refer to this style attribute as a merge style.

2. A merge style must have the `merge_proc` function pointer set to an application-defined function (see "`merge_proc`").
3. By itself, a merge style does nothing and text set to a merge style remains unchanged until the application calls `pgMergeText` (below). OpenPaige will then call the appropriate `merge_procs`, at which time the application makes the decision for substituting text (or not).
4. When `pgMergeText` is called, the text for which all merge styles applied is temporarily *pushed* (saved) into an internal `memory_ref` within the OpenPaige object. Later, when the application wishes to revert from "merge mode", the document can be completely restored to its original state, prior to any text substitutions.

Sample merge text proc

This is called when the styles need to be initialized. Usually at the beginning of the program. This sets the merge style `procs` and `user_id` and the mask makes it so only the two desired `procs`, `merge_text_proc` and `setup_insert`, get set to our custom ones following:

```
void InitMergeStyles(pg_ref pg)
{
  style_info style, mask;
  pg_style_hooks style_functions;
  pgInitStyleMask(&mask, 0);

  style.user_id = STYLE_IS_MERGE;
  mask.user_id = -1;

  // The idea is to change only the styles that have pictures:

  InitStyleProcsToDefaults(&style_functions); // Init standard procs first.
  style_functions.merge = (pg_proc) merge_text_proc;
  style_functions.insert_proc = (pg_proc) setup_insert;
  pgSetStyleProcs(pg, &style_functions, &style, &mask, NULL, 0,
STYLE_IS_MERGE, FALSE, draw_none);
}
```

Mail merge fields are inserted into text

```
/* This function inserts my "mail merge" fields into the text. I shall use
only a couple of style hooks to make this work. */

void insert_merge_fields (doc_rec *doc)
{
  style_info style, mask;
  short index, size_of_fld;
  Str255 name_of_fld;
  for (index = 0; index < NUM_MERGE_FLDS; ++index)
  {
    GetIndString(name_of_fld, MERGE_STRINGS, index + 1);
    size_of_fld = name_of_fld[0];
    pgGetStyleInfo(doc -> pg, NULL, FALSE, &style, &mask);
    pgInitStyleMask(&mask, 0);

    // Set up everything I want in the style_info record:

    style.user_id = STYLE_IS_MERGE;
    style.class_bits |= (STYLE_IS_CUSTOM | GROUP_CHARS_BIT);
    style.char_bytes = 0;
    style.user_data = index;
    mask.user_id = -1;
    mask.user_data = -1;
    mask.class_bits = -1;
    mask.char_bytes = -1;
    // Set desired function pointers:

    style.procs.merge = (pg_proc) merge_text_proc;
    style.procs.insert_proc = (pg_proc) setup_insert;
    mask.procs.merge = (pg_proc) - 1;
    mask.procs.insert_proc = (pg_proc) - 1;
    pgSetStyleInfo(doc → pg, (pg_char_ptr) &name_of_fld[1], size_of_fld,
CURRENT_POSITION, data_insert_mode, 0, draw_none);
  }

  InvalRect(&doc → w_ptr → portRect);
  DoAllUpdates();
}
```

Sample `setup_insert` ***hook for merging***

```
/* This is the hook that gets called when OpenPaige saves off the next style
to apply from the next insert. The reason I need this for merge "characters"
is because I don't want the user to "type" or extend text if the caret sits
on one of my merge styles. Hence, this function must remove my own hooks from
the style so it becomes just a regular style. */

static pascal short setup_insert (paige_rec_ptr pg, style_info_ptr style,
long position)
{
  pgInitStyleProcs(&style → procs);
  // This sets all the standard procs
  style → class_bits = 0;
  style → user_data = style → user_id = 0;

  return TRUE;
  /* Won't call me again (because I just nuked my own function ptr */
}
```

`merge_text_hook`

```
// This gets called by page to swap out text during pgMergeText.
static pascal short merge_text_proc (paige_rec_ptr pg, style_info_ptr style,
pg_char_ptr text_data, pg_short_t length, text_ref merged_data, long ref_con)
{
  short field_size
  char *str_to_merge;
```

```
   field_size = *merge_text[style -> user_data];
   if (!merged_data)
     return TRUE;
   SetMemorySize (merged_data, field_size);
   if (!field_size)
     return TRUE;
   str_to_merge = (char*) merge_text[style -> user_data];
   ++str_to_merge;
   BlockMove(str_to_merge, UseMemory(merged_data), field_size);
   UnuseMemory(merged_data);
   return TRUE;
 }
```

29.2 Merge mode

Assuming all your merge styles have been set up (all the desired merge areas have a `merge_proc` set in their `style_info` record), placing the OpenPaige object in "merge mode" is accomplished by calling the following:

```
(pg_boolean) pgMergeText (pg_ref pg, style_info_ptr matching_style,
style_info_ptr mask, style_info_ptr AND_mask, long ref_con, short draw_mode);
```

For every `style_info` that matches a specified criteria (based on the contents of `matching_style`, `mask` and `AND_mask` as described below), has its `merge_proc` called, at which time text can be substituted in place of the text that currently exists for each style involved.

Before any text is substituted, however, the "old" text is saved temporarily within the OpenPaige object. This is intended to allow the application to "revert" to the original document at some later time.

Styles that are affected by this call (in which the `merge_proc` gets called) are determined on the following bases:

- The fields in each `style_info` record in OpenPaige is compared to each field in `matching_style` if the corresponding field in `mask` is non-zero.
- Before the comparison, the corresponding field in `AND_mask` is temporarily `AND`'d with the target `style_info` field in OpenPaige before it is compared.
- If all fields that are checked in this way match exactly, the style is "accepted" and the `merge_proc` gets called.

Any of the three comparison styles – matching_style, mask and AND_mask can be null pointers to control the comparison procedure, in which case the following occurs:

- If matching_style is null, then all styles in pg are considered "valid" with no comparisons made, hence all merge_procs are called.
- If mask is null, all fields in each style are compared to matching_style (none are skipped).
- If AND_mask is null, no AND occurs before the field comparisons (instead, the fields are compared as-is).

Using the various combinations of matching_style, mask and AND_mask, you can selectively "merge" various styles based on a nearly infinite set of criteria.

The ref_con parameter can be anything; this value gets passed to the merge_proc.

NOTE: The MERGE_MODE_BIT will be set in pg's attributes when the document has been "merged" in the above fashion.

You can check the attributes using pgGetAttributes.

FUNCTION RESULT: This function returns TRUE if anything merged at all; FALSE is returned if no text has been substituted from any merge_proc (hence the document remains unchanged).

NOTES

1. If you intend to revert to the original document using pgRestoreMerge *infra*, you must not insert any new text or allow any kind of editing by the user until you revert. It is OK, however, to do multiple pgMergeText calls before reverting the document.
2. The original document is saved only once; subsequent pgMergeText calls will not save the merge styled text again. Hence, you can make multiple pgMergeText calls before reverting, then pgRestoreMerge will revert the document to its state before the first merge.
3. Even if you intend not to revert the text, you need to call pgRestoreMerge anyway, otherwise a memory leak can result.

Restore merge

```
(void) pgRestoreMerge (pg_ref pg, pg_boolean revert_original, short
draw_mode);
```

This function "reverts" pg to its original state, prior to the first pgMergeText call if revert_original == TRUE.

If revert_original == FALSE, the previous text that has been saved within pg is simply disposed and the document is not reverted. The purpose of this parameter is to allow a document to "convert" to a merged state, but to keep it that way.

If draw_mode is non-zero, pg is re-displayed. draw_mode can be the values as described in *Draw Modes* under section 2.11:

```
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
```

This function only needs to be called once, even after multiple pgMergeText calls. Once you have reverted, however, a subsequent call to pgRestoreMerge will do nothing (unless you have done another pgMergeText).

This function also clears the MERGE_MODE_BIT from pg's attributes.

NOTE: Even if you do not wish to revert the text, you should call pgRestoreMerge anyway (with revert_original as FALSE) if anything has merged to dispose the saved text.

Show Merged Items

```
{
  init_merge_strings();
  if (pgGetAttributes(doc -> pg) & MERGE_MODE_BIT)
  {
    pgSetHiliteStates(doc -> pg, no_change_verb, activate_verb, FALSE);
    pgRestoreMerge(doc -> pg, TRUE, draw_none);
  }
```

```
    else
    {
      pgMergeText(doc → pg, NULL, NULL, NULL, 0, draw_none);
      pgSetHiliteStates(doc → pg, no_change_verb, deactivate_verb, FALSE);
    }
    InvalRect(&doc → w_ptr → portRect);
    DoAllUpdates();
  }
```

TECH NOTE: `STYLE_IS_CUSTOM` bit set incorrectly

I looked into your code and found that you are correct that setting up the `style_info` record is the problem. You need to *remove* the `class_bits` setting, `STYLE_IS_CUSTOM`. That's what is forcing the merge field to not draw.

`STYLE_IS_CUSTOM` tells OpenPaige that only your app knows how to draw the style and measure its characters. Hence, if you call the standard draw/measure functions (which you are), they will *do nothing*. I not only noticed the fields were invisible, on my machine, but the char widths would result in random garbage text sizes (which is *correct* since the standard measuring does nothing for `STYLE_IS_CUSTOM`).

Technically, the style is not "custom" at all–it has regular text chars and it draws like any other text. By strict definition, a `STYLE_IS_CUSTOM` means OpenPaige can't understand the "text" stream, such as an embedded `PicHandle` or `ControlHandle`, etc.

TECH NOTE: Merge fields and blank lines

Regarding merge fields and blank lines (and how to remove them) in items like addresses, I am not sure I have a perfect answer for that. I don't think you dare try deleting anything from within a hook, you will probably get a debugger break (because `memory_refs` for the text and styles will be locked and "in use").

The only thing I can think of is to detect this situation and, after all is merged, go back and delete the "blank lines".

CAUTION: If you do this, I am not sure `pgRestoreMerge` will work correctly, because it assumes you have not edited the document.

We had another customer doing extensive altering of a merged document for similar reasons, and he had to simply restore the original doc without using `pgRestoreMerge`. Rather, he would copy the document and then do `pgUndo`.

TECH NOTE: Restore-all not yet implemented and the work-around

If I understand you correctly, the reason you need to throw away each document and reread it-as opposed to relying on "restore merge" feature-is due to your extra editing of the document and the fact that "restore merge" just restores the merge styles.

The supreme work-around would be, of course, for us to add "restore all" to the merge features-which, incidentally, is not a bad idea. Sooner or later, someone else will encounter the same problem.

In the meantime (since that feature is not currently available in `pgMerge`), I would suggest starting with a single `pg_ref`, as you are now, but use `pgCopy` to duplicate the doc, given that `pgCopy` can produce what you thought `pgDuplicate` did.

Here are some precautions/hints:

- To duplicate a whole document, simply use `pgCopy` with a selection parameter for whole text range. (Remember that `pgCopy` returns a new `pg_ref`-which is exactly what you want).
- You might have a problem displaying the copied `pg_ref`; I do not believe the exact `vis` and `page` areas are copied. In that case you might need to set those shapes before drawing (or printing) the merged document. I would get the shape(s) from the master document then do `pgSetAreas` to the copy. Even faster would be to get the master shapes once at the beginning, with `pgGetAreas`, then set them for each merge.
- I do know for sure that a copied `pg_ref`, from `pgCopy`, will have no `graf_device` associated with it. I do not know if this is a problem if you intend to print, since `pgPrintPage` accepts a `graf_device` (which would be a print port). But if you need to draw the merged doc to a window, you will need to set a window port using `pgSetDefaultDevice`, **or** you will need to specify a `graf_device_ptr` in `pgDisplay`. Otherwise, the drawing will be "invisible" and you will think you are going crazy. I believe our manual explains this (if not, I will be happy to provide more details).

- In OpenPaige's current stage, I do not believe anyone has yet to display (or print) a copied `pg_ref` returned from `pgCopy`. Usually, they just paste with it. In that case, you may have unforeseen problems. However, all such cases (other than the precautions listed above) I would consider a bug, so be sure to let us know so the problem(s) can be corrected. We will make sure that a copied `pg_ref` displays correctly, one way or the other.
- You may encounter some slowness with this work-around. However, that will probably improve during future updates.

30 ADVANCED STYLES

This chapter unveils all of the style and font setting abilities within OpenPaige. For easier and quicker implementation of style setting, you will want to look at "Style Basics".

NOTE: As used in this chapter, the term *font* generally refers only to a typeface, or typeface name, unlike a Windows "font" that defines the whole composite format of text. OpenPaige considers a *font* to simply be a specific family such as Times, Courier, Helvetica, etc. while distinguishing other formatting properties such as bold, italic, underline, etc. as the text *style*.

30.1 Data Structures

For the sake of clarity and easier implementation of text formatting, the exact structure definitions and descriptions for style, font and paragraph formats are given at the end of this section. While you will need to set up these structures to effectively change text styles, they have been placed at the end for easier reference.

To understand the functions, however, let it suffice to declare the type for each of the four formats, as follows:

Record Type	Pointer (to the record)	Description
style_info	style_info_ptr	Structure defining a style
font_info	font_info_ptr	Structure defining a font
par_info	par_info_ptr	Structure defining paragraph format
style_run	style_run_ptr	Structure designating a style run.°

° A series of style_run records is maintained by OpenPaige to define all the style changes and associated text offsets. This record is much smaller than either style_info or par_info, thus requiring only one style_info record for every identical style change throughout the text and one par_info record for every identical paragraph format throughout the text. The style_run record is defined at the end of this section; most of the time you will not need to access style_runs.

30.2 More About Style Runs

For both style_info and par_info changes throughout the text, OpenPaige maintains a list of style_run records. There is one style_run array for style_info changes and one array for par_info changes.

The last element in a style_run array is a "dummy" entry whose offset field will be greater than the total text size of the pg_ref. For example, if the total text size of a pg_ref is 100 bytes, the final element in the array of style_run records will contain a value in style_run.offset of > 100.

30.3 Style Basics

To simply set a style, font, size or paragraph format, see "Style Basics". The following information is for those developers wanting more precise control of style, font and paragraph format setting.

30.4 Changing Fonts and Styles together

This sets the font and style at the same time.

```
(void) pgSetStyleAndFont (pg_ref pg, select_pair_ptr selection,
style_info_ptr the_style, style_info_ptr style_mask, font_info_ptr font,
font_info_ptr font_mask, short draw_mode);
```

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

If you do give a pointer to selection, it must point 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;
```

- 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 (first offset is zero).
- info and mask - parameters must point to style_info records; info is the new style to apply to the text and mask defines which of the info fields to apply. For every non-zero field in mask, the corresponding field in info gets applied to the text.

- mask - parameter allows only partial style changes, which is almost always what you want to accomplish. For instance, if the user highlighted some text and changed it to **bold**, all you want to change in the text range is the **bold** attribute, not anything else such as colour, leading, or any other formatting. To do so, you would set info's style element for **bold** and the same field in mask to non-zero.
- font and font_mask - is almost identical to the similar style parameters, except in that a font_info record is used for font and font_mask.
- info, mask, font and font_mask - None of these can be a null pointer.
- draw_mode - parameter indicates whether or not to redraw the text once the style has changed: if draw_mode is non-zero, that drawing mode is used to redisplay the text.
- If draw_mode is non-zero, pg is re-displayed. draw_mode can be the values as described in *Draw Modes* under section 2.11:

```
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
```

OpenPaige will only re-draw the text, however, if it is appropriate: if nothing changed (same styles applied as already exist), the text is not drawn, nor is it drawn if the new style applies only to an insertion point.

NOTE: The mask fields that indicate what to change must be set to -1 (all ones). The reason is that the internal comparison function–which checks the mask settings–does one word at a time, hence the fields longer than 16 bits will not change correctly.

Preparing OpenPaige formats from a LOGFONT *(Windows)*

```
/* The following example function converts a LOGFONT into a font_info,
style_info and "mask" record that can be given to pgSetStyleInfo(): */
```

```
static void convert_log_font (pg_ref pg, pg_globals_ptr paige_globals,
LOGFONT PG_FAR *log_font, font_info_ptr font, style_info_ptr style,
style_info_ptr stylemask)
{
  // Initialise the style to OpenPaige default:
  *style = paige_globals -> def_style;
  // Initialise other structs to zeros or -1's, etc.:
  pgFillBlock(font, sizeof(font_info), 0);
  pgFillBlock(stylemask, sizeof(style_info), 0);
  pgFillBlock(stylemask -> styles, MAX_STYLES * sizeof(short), -1);
  stylemask -> point = (pg_fixed) - 1;
  CToPString(log_font -> lfFaceName, font -> name);
  // (OpenPaige wants a pascal string)
  font -> family_id = log_font -> lfPitchAndFamily;
  font -> machine_var[PG_OUT_PRECISION] = log_font -> lfOutPrecision;
  font -> machine_var[PG_CLIP_PRECISION] = log_font -> lfClipPrecision;
  font -> machine_var[PG_QUALITY] = log_font -> lfQuality;
  font -> machine_var[PG_CHARSET] = log_font -> lfCharSet;

  if ((style -> point = (pg_fixed)log_font -> lfHeight) < 0)
    style -> point = -style -> point;
  style->point <<= 16; // Make sure point size is 0x000n0000
  // Convert pointsize to fit the screen resolution
  style->point = pgScreenToPointsize (pg, style->point);
  if (log_font -> lfWeight == FW_BOLD)
    style -> styles[bold_var] = -1;
  if (log_font -> lfItalic)
    style -> styles[italic_var] = -1;
  if (log_font -> lfUnderline)
    style -> styles[underline_var] = -1;
  if (log_font -> lfStrikeOut)
    style -> styles[strikeout_var] = -1;
}
```

30.5 Easier "Mask" Setups

Masks

The easiest way to initialise a style_info, font_info, or par_info record for a "mask" is to call one of the following:

```
(void) pgInitStyleMask (style_info_ptr mask, short filler);
(void) pgInitFontMask (font_info_ptr mask, short filler);
(void) pgInitParMask (par_info_ptr mask, short filler);
```

These function fill mask with filler.

To set a section of text to a style, call the following:

```
(void) pgSetStylelnfo (pg_ref pg, select_pair_ptr selection, style_info_ptr
info, style_info_ptr mask, short draw_mode);
```

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

If you do give a pointer to selection, it must point 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;
```

- begin and end fields of a select_pair - define the beginning and ending text position. Both values are byte positions (not necessarily character positions, e.g. multilingual text can have double-byte characters, etc.). Text positions in OpenPaige are zero-indexed (first offset is zero).
- info and mask - must point to style_info records; info is the new style to apply to the text and mask defines which of the info fields to apply. For every non-zero field in mask, the corresponding field in info gets applied to the text.
- mask - allows only partial style changes, which is almost always what you want to accomplish. For instance, if the user highlighted some text and changed it to **bold**, all you want to change in the text range is the **bold** attribute, not anything else such as colour, leading, or any other formatting. To do so, you would set info's style element for bold and the same field in mask to non-zero.

  Neither info nor mask can be a null pointer.
- draw_mode - indicates whether or not to redraw the text once the style has changed: if draw_mode is non-zero, that drawing mode is used to re-display the text. draw_mode can be the values as described in *Draw Modes* under section 2.11:

```
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
```

HERMES Paige will only redraw the text, however, if it is appropriate: if nothing changed (same styles applied as already exist), the text is not drawn, nor is it drawn if the new style applies only to an insertion point.

NOTE: The `mask` fields that indicate what to change must be set to `-1` (all ones). The reason is that the internal comparison function-which checks the mask settings-does one word at a time. Hence, the fields longer than 16 bits will not change correctly.

NOTE: To convert a `LOGFONT` into a `style_info` and `mask` on **Windows**, see code example earlier in this chapter.

Set some text to "bold" (Macintosh)

```
/* This function sets the current selection to Bold (Macintosh) */
void set_to_bold (pg_ref pg)
{
  style_info mask, info;
  pgInitStyleMask(&info, 0); // Sets all to zero
  pgInitStyleMask(&mask, 0); // Sets all to zero
  info.styles[bold_var] = -1; // sets styles[bold_var] to force bold
  mask.styles[bold_var] = -1;
  pgSetStyleInfo(pg, NULL, \&info, \&mask, best_way);
}
```

While the styles each contain `shorts` to indicate bold, italic, etc., this is generally done for future expansion. When OpenPaige was designed, new fonts were being created which would use "degrees of boldness", etc. Generally, this is not implemented in OpenPaige 1.0 for Mac and Windows except for the following style elements:

- style_info → styles[small_caps_var] - The value in this style element indicates a percentage of the original point size to display lower case characters that get converted to ALL CAPS. Or, if this value is -1 , the default is used (which is 75% of the original style).

  For example, if style_info → styles[small_caps_var] is 50 and style_info point size is 24, the lower case text is converted to uppercase 12 point; if style_info → styles[small_caps_var] is -1, the lower case text is converted to 18 (which is 75% of 24).

- style_info→styles[relative_point_var] - The value in this style element indicates a point size to display the text which is a ratio relative to 12 point times the original point size. The ratio is computed as: style_info → styles[relative_point_var] / 12. (The "original point size" is taken from the point field in style_info).

  For example, if style_info → styles[relative_point_var] is 6 and the original point size is 12, the point size that displays is 12 * (6/12) = 6 point. If style_info → styles[relative_point_var] is 6 and the original point size is 24, the point size that displays is 24 * (6/12)= 12 point.

  NOTE: The relative_point_var element must not be "-1" as there is no default.

- style_info → styles[super_impose_var] - If non-zero, the value represents a stylesheet ID that gets "superimposed" over the existing style. What this means is all fields in the stylesheet style_info -> styles[super_impose_var] that are non-zero are temporarily forced into style_info to form a composite style of both.

  For example, if style_info → styles[super_impose_var] record had all fields set to zero *except for* the bold_var element, the resulting style would be whatever the original style_info contained but with **boldface** text.

  style_info→styles[super_impose_var] can only be zero or a positive number representing a stylesheet ID that actually exists in the pg_ref.

See the chapter "Style Sheets" for more information.

Insertion Point Changes

If pgSetStyleInfo is called and the specified selection is a single insertion point, the style change occurs on the next pgInsert. Furthermore, a processed mouse-click for change of selection invalidates the style_info set to the previous insertion point (i.e., the new style setting is lost).

Exception: Applying a style to a completely empty pg_ref forces that style_info to become the default style for that pg_ref.

The point field in style_info is a Fixed type value, i.e. the integral part of the point size is the high-order word and the fractional part is the low-order word. For example, 12 point is (pg_fixed) (12<<16) or 0x000C0000.

TECH NOTE: Changing point size

I am having some difficulty in setting the point size of the font within OpenPaige.

Your code doesn't work because the point size in style_info is a Fixed value, which means the whole-number point size needs to be in the high-order word—and you're just setting a long integer (which is putting it in the low-order word). You must have skipped quite a few OpenPaige versions because that change has been there for a while.

So, your code is fine *except* you need to put the point size in the high-order word, and it will work. Something like:

```
theStyle. point = fontSize;
theStyle.point <<= 16;
```

In case you're curious, OpenPaige only looks at the high-word of the point size, so setting only the low word results in "zero point", i.e., the default—12 point—which is why it never changed.

30.7 Changing Fonts

Changing the font applied to text range(s) requires a separate function call since fonts are maintained separate from text styles within a pg_ref.

NOTE: As used in this chapter, the term *font* generally refers only to a typeface, or typeface name, unlike a Windows "font" that defines the whole composite format of text. OpenPaige considers a *font* simply to be a specific family such as Times, Courier, Helvetica, etc. while distinguishing other formatting properties such as bold, italic, underline, etc. as the text *style*.

To set a section of text to a new font, call the following:

```
(long) pgSetFontInfo (pg_ref pg, select_pair_ptr selection, font_info_ptr
info, font_info_ptr mask, short draw_mode);
```

This function is almost identical to pgSetStyleInfo except in that a font_info record is used for info and mask.

- selection and draw_mode - are functionally identical to the same parameters in pgSetStyleInfo. The same rules apply regarding insertion points versus selection range(s).
- draw_mode can be the values as described in *Draw Modes* under section 2.11:

  ```
  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
  ```

Detailed information on font_info records-and what fields you should set up-is available at the end of this section. There is one important one you should be sure to set correctly, however: environs.

When you set a font_info record, only the name and environs fields should be changed; this is because OpenPaige initialises all the other fields when the font is applied to a pg_ref.

For **Macintosh** version, the font_info.name should be a pascal string terminated with the remaining bytes in font_info.name set to zero; the font_info.environs field should likewise be set to zero. For an example, read on.

For Windows version, the font_info.name can be initially set to either a pascal string or a cstring, with all remaining bytes in font_info.name set to zero. Usually, due to Windows programming conventions, you will set the name to a cstring. In this case, before passing the font_info record to pgSetFontInfo, you must set font_info.environs to NAME_IS_CSTR (see following example).

CAUTION: On Windows, OpenPaige converts font_info.name to a pascal string and clears the NAME_IS_CSTR bit when the font is stored in the pg_ref. This is done purely for cross-platform portability. This is important to remember, because if you examine the font thereafter with pgGetFontInfo, the font name will now be a pascal string (the first byte indicating the string length), not a cstring.

Setting font_info (Windows)

```
/* This example assumes we got a "LOGFONT" struct from a ChooseFont dialog or
similar. */

LOGFONT log;
font_info font, mask;
pgFillBlock(&font, sizeof(font_info), 0); // clear all to zeros
pgFillBlock(&mask, sizeof(font_info), -1); // Set to all 1's
lstrcpy((LPSTR)font.name, log.IfFaceName);

/* IMPORTANT! The following line is an absolute MUST or your code will fail:
*/

font_info.environs |= NAME_IS_CSTR;

/* Apply to the text: */

pgSetFontInfo(pg, NULL, &font, &mask, best_way);
```

Responding to font menu (Macintosh)

```
/* In this example, we assume a "Font" menu whose MenuHandle is FontMenu, and
"item" is the menu item selected by the user. */

font_info font, mask;
pgFillBlock(&font, sizeof(font_info), 0); // clear all to zeros
pgFillBlock(&mask, sizeof(font_info), -1); // Set to all 1's
GetItem(FontMenu, item, (StringPtr)font.name);
pgSetFontInfo(pg, NULL, &font, &mask, best_way);
```

30.8 Obtaining Current Text Format(s)

```
(long) pgGetStyleInfo (pg_ref pg, select_pair_ptr selection, pg_boolean
set_any_match, style_info_ptr info, style_info_ptr mask);
(long) pgGetParInfo (pg_ref pg, select_pair_ptr selection, pg_boolean
set_any_match, par_info_ptr info, par_info_ptr mask);
(long) pgGetFontInfo (pg_ref pg, select_pair_ptr selection, pg_boolean
set_any_match, font_info_ptr info, font_info_ptr mask);
```

The three functions above return information for a specific range of text about its style or paragraph format, or font, respectively.

For all functions, 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 point to a `style_info`, `par_info`, or `font_info` record; neither of the former can be a null pointer. When the function returns, both `info` and `mask` will be filled with information you can examine to determine which 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 non-zero 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 pgGetStyleInfo, mask.styles[bold_var] has a non-zero value. That means that whatever value has been set in info.styles[bold_var] is the same for every character in the selected text. Hence if info.styles[bold_var] is -1, then *every* character is **bold**; if info.styles[bold_var] is 0 , then *no* character is **bold**.

On the other hand, suppose after calling pgGetStyleInfo, mask.styles[bold_var] is set to zero. That means that some of the characters in the selected text are **bold** and some are *not*. In this case, whatever value happens to be in info.styles[bold_var] is, mathematically speaking, *undefined* (think zero divided by zero).

Essentially, a non-zero-valued mask is saying, "Whatever is in info for this field is applied to every character in the text," and a zero-valued 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."

Pass FALSE for set_any_mask to find out which styles, paragraph formats or fonts do and do not apply to the entire selection.

If set_any_mask is TRUE, all fields in mask get set to nonzero if the corresponding field value in info appears *anywhere* within the selected text. In this case, you must first set all the info fields that you want to check before making the call.

The purpose for setting set_any_mask to TRUE is to find out what item(s) in info exist anywhere in the selected text (as opposed to finding out what items are the same *throughout* the text).

NOTE: If you pass FALSE for set_any_mask, OpenPaige sets the info_fields; if you pass TRUE for set_any_mask, you set the info_fields before calling pgGetStyleInfo, pgGetParInfo or pgGetFontInfo. This is a *significant* difference.

For example, suppose you want to find out if **bold** exists anywhere in the selected text. To do so, you would set info.styles[bold_var] to a non-zero value, then call pgGetStyleInfo() passing TRUE for set_any_mask. Upon return, if mask.styles[bold_var] is TRUE (non-zero), that means **bold** exists *somewhere* in the selected text. On the other hand, had the function returned and mask.styles[bold_var] was FALSE, that would have meant that **bold** exists *nowhere* in the text.

Usually, the reason you would want to pass TRUE for set_any_mask is to make some kind of notation on a UI element (e.g. a menu or dialogue box) as to which style(s) appear anywhere in a selection but do not necessarily apply to the *entire* selection.

FUNCTION RESULT: Each function returns the text offset (which is a byte offset) of the first text that is examined. For example, if the current selection range in pg was offsets 100-500, pgGetStyleInfo would return 100; for the same selection range pgGetParInfo would return the text offset of the beginning of the first paragraph (which would often be less than 100).

NOTE: If you want to get information about tabs, it is more efficient (and less code) to use the functions in the section below, See also, "Tab Support".

30.9 Getting Info Recs

An additional way to obtain the current font that applies to a text range is to first obtain the style information that applies using pgGetStyleInfo, then get the font record by calling the following function:

```
(void) pgGetFontInfoRec (pg_ref pg, short font_index, font_info_ptr info);
```

- font_index - should be whatever is in the font_index field in the style_info record (which you received from pgGetStyleInfo). The font record is put into info (which must not be a null pointer).

This function is used to fill in the whole font record if you already know its font index number, which you do after doing a pgGetStyleInfo.

Styles and fonts have the same functions that will fill in the appropriate record.

```
(void) pgGetStyleInfoRec (pg_ref pg, short style_item, style_info_ptr
format);
(void) pgGetParInfoRec (pg_ref pg, short style_item, par_info_ptr format);
```

These functions take the style_item value from a style_run record and return a par_info or style_info record.

NOTE: This is a low-level function that you will rarely need but has been provided for documentation purposes. See style_run information at "More About Style Runs".

Obtaining a font record

```
/* This function is to obtain a font record that is "attached" to a
style_info record. For example, you could get the whole font record after
doing pgGetStyleInfo as follows: */

style_info info, mask;
font_info font;
pgGetStyleInfo(pg, NULL, FALSE, &info, &mask);
pgGetFontInfoRec(pg, info.font_index, &font);
```

30.10 Other Style, Font and Paragraph Utilities

Set Insertion Styles

This function provides a convenient way to set both a style record and font for a single insertion point.

```
(void) pgSetInsertionStyles (pg_ref pg, style_info_ptr style, font_info_ptr
font);
```

The style parameter will be the style that will apply to the next pgInsert; the font parameter will be the font that will apply to the next pgInsert. Neither parameter can be null.

NOTE: This function is intended for single insertion points and will fail to work correctly if there is a selection range in pg.

TECH NOTE: pgSetInsertionStyles is a convenience

Is pgSetInsertionStyles just a convenience function? Or should I be using this to set font/style info when there is only an insertion point (no selection), i.e., can I simply always use pgSetStyleInfo and pgSetFontInfo, and never use pgSetInsertionstyles?

This is only a convenience function; you will probably never use it. pgSetStyleInfo handles this for you. It checks the selection and, if only a "caret", it calls pgSetInsertionStyles for you.

Style info of a Mouse Point

```
long pgPtToStyleInfo (pg_ref pg, const co_ordinate_ptr point, short
conversion_info, style_info_ptr style, select_pair_ptr range);
```

This function is useful for determining which style_info is applied to the character containing a specific mouse coördinate. For instance, pgPtToStyleInfo() can be used to detect what kind of text the cursor is moving through.

When this function returns, if is non-null it gets set to the range of text for which this style applies (see "Selection range" for information about select_pair record).

conversion_info is used to indicate one or two special-case alignment instructions, which can be represented by the following bits:

```
#define NO_HALFCHARS  0x0001  // Whole char only
#define NO_BYTE_ALIGN 0x0002  // No multibyte alignment
```

NO_HALFCHARS instructs the function to select the right side of a character if the point is anywhere to the right of its left side (not having NO_HALFCHARS set results in the left side of the character if the point is within its first half-width, or the right side of the character if the point is within its second half-width).

NO_BYTE_ALIGN returns the absolute byte position regardless of multibyte character status. For example, in a Kanji system that contains double-byte characters, setting NO_BYTE_ALIGN can result in the selection of 1/2 character.

FUNCTION RESULT: The function result is the text (character) position for the character found containing point. The function will always return a style and position even if the point is way beyond text (in which case the style for the last character is returned) or before text (where the first style is returned). Either style or range can be a null pointer if you don't need those values.

NOTE: This function always finds some style_info even if point is nowhere near any text. Hence, to detect "true" cursor-over-text situations you should also call pgPtInView() to learn whether or not the point is actually over text.

Font table

```
(memory_ref) pgGetFontTable (pg_ref pg);
```

FUNCTION RESULT: This function returns the memory allocation in $p g$ that contains all the fonts used for the text. The memory_ref will contain an array or one or more font_info records.

NOTE: The actual memory_ref that OpenPaige used for this pg_ref is returned, not a copy. Therefore do not dispose this allocation and do not delete any records it contains.

To learn how to access a memory_ref, see "The Allocation Mgr" on page 25-1.

30.11 Record Structures

style_info

```
#define MAX_STYLES32  // Maximum number of styles in style_info
typedef struct
{
  short font_index;          // What font this style is in
  short styles[MAX_STYLES]; // Stylisation extension
  short char_bytes;          // Number of bytes per character less 1
  short max_chars;           // Maximum chars before new style begins
  short ascent;              // This style's ascent
  short descent;             // This style's descent
  short leading;             // Regular leading
  short shift_verb;          // Super/subscript verb
  short class_bits;          // Defines selection and behaviour
  long  style_sheet_id;      // Used for style sheet features
  long  small_caps_index;    // style_info index for point size
  long  machine_var;         // Machine-specific
  long  machine_var2;        // Machine-specific
  long  left_overhang;       // Style's left overhang if any
  long  right_overhang;      // Style's right overhang if any
  long  top_extra;           // Style's top leading
  long  bot_extra;           // Style's bottom leading
  long  space_extra;         // Extra pixels for spaces (Fixed value)
  long  char_extra;          // Extra pixels for chars (Fixed value)
  long  user_id;             // Can be used by app to ID custom styles
  long  user_data;           // Add'l space for app if style is custom
  long  user_data2;          // Add'l space for app
  long  future[3];           // Reserved for future exp'n
  long  embed_entry;         // App callback function for embed_refs
```

```
    long  embed_style_refcon; // Used by embedded object ext'n
    long  embed_refcon;       // Used by embedded object ext'n
    long  embed_id;           // Used by embedded object ext'n
    long  maintenance;        // Internal use
    long  used_ctr;           // Internal use
    color_value     fg_color;     // Foreground colour
    color_value     bk_color;     // Background colour
    pg_fixed        char_width;   // Character width (not used on Mac)
    pg_fixed        point;        // Point size (do <<16 on int value)
    memory_ref      embed_object; // Used by embedded object ext'n
    style_append_t  user_var;     // Arbitrary use
    pg_style_hooks  procs;        // Contains functions on how to draw
  }
  style_info, PG_FAR *style_info_ptr;
```

Field descriptions

- font_index – The record number of the font that goes along with this style. (To obtain the actual font, see "Getting Info Recs" for information about pgGetFontInfoRec).

  NOTE: Do not change the font_index using pgSetStyleInfo. Instead, use pgSetFontInfo and the font_index values will be handled by OpenPaige appropriately.

- styles – An array of shorts that correspond to 32 possible "standard" styles. Each element of styles, if non-zero, implies that style be applied to the text. An overall style of "plain" generally means all style elements are zero.

  The standard styles supported by OpenPaige are defined by the following enumerates (each corresponding to one of the array elements):

  ```
  enum
  {
    bold_var,
    italic_var,
    underline_var,
    outline_var,
    shadow_var,
    condense_var,
    extend_var,
    dbl_underline_var,
    word_underline_var,
    dotted_underline_var,
  ```

```
    hidden_text_var,
    strikeout_var,
    superscript_var,
    subscript_var,
    rotation_var,           // future, not currently supported
    all_caps_var,
    all_lower_var,
    small_caps_var,
    overline_var,
    boxed_var,
    relative_point_var,
    super_impose_var,
    dsi_custom_var = 27,    // Internal use by HERMES
    custom_var = 28
  };
```

- `superscript, subscript` - If `styles[superscript_var]` or `styles[subscript_var]` apply, their values define the "amount" of shift.

  For example, if `styles[sub_script_var]` contains a value of 3, the text is to be shifted down by 3 points (3 pixels). If `styles[super_script_var]` were 3, the text is to be shifted upwards by 3 points. However, the `shift_verb` (*infra*) defines whether or not the super/subscript is relative to the text baseline or relative to a percentage of the style's height.

- `small_caps` - If `styles[small_caps_var]` applies, the value in this style element indicates a percentage of the original point size to display lower case characters that get converted to ALL CAPS. Or, if this value is -1, the default is used (which is 75% of the original style).

  For example, if `style_info → styles[small_caps_var]` is 50 and `style_info` point size is 24 , the lower case text is converted to uppercase 12 point; if `style_info → styles[small_caps_var]` is -1, the lower case text is converted to 18 (which is 75% of 24).

- `relative_point` - If `styles[relative_point_var]` applies, the value in this style element indicates a point size to display the text which is a ratio relative to 12 point times the original point size. The ratio is computed as: `style_info → styles[relative_point_var] / 12`. (The "original point size" is taken from the `point` field in `style_info`).

For example: If style_info → styles[relative_point_var] is 6 and the original point size is 12, the point size that displays is 12 * (6 / 12) = 6 point. If style_info → styles[relative_point_var] is 6 and the original point size is 24, the point size that displays is 24 * (6 / 12) = 12 point.

NOTE: The relative_point_var element must not be -1 as there is no default.

- super_impose - If styles[super_impose_var] applies, the value represents a stylesheet ID that gets "superimposed" over the existing style. All fields in the stylesheet style_info → styles[super_impose_var] that are non-zero are temporarily forced into style_info to form a composite style of both.

  For example, if style_info → styles[super_impose_var] record had all fields set to zero EXCEPT for the bold_var element, the resulting style would be whatever the original style_info contained but with boldface text.

  NOTE: style_info → styles[super_impose_var] can only be zero or a positive number representing a stylesheet ID that actually exists in the pg_ref.

  See "Style Sheets" for more information.

- char_bytes - Defines the number of bytes per character minus 1. For "normal" text, this field will be zero.

  NOTE: If you are a **Macintosh** user, do not confuse this with double-byte scripts such as Kanji. This field is intended for situations where all character are char_bytes + 1 in size, such as a feature in which a PicHandle is embedded as a "character". For Kanji, not every character is a double-byte so this field will always be zero.

- max_chars - **Not currently supported.** Eventually this will be used for something fancy.

- ascent, descent, leading - Define the style's ascent, descent and leading values. (For **Macintosh**, each value is obtained from the Toolbox call, GetFontInfo).

  NOTES:

  1. You do not need to set these fields for "normal" (non-custom) styles because the machine-specific portion of OpenPaige will initialise these fields according to the composite style and font.

  2. If you need to implement a "set extra leading" feature, use top_extra and bot_extra *infra*.

- shift_verb - This value is used only if styles[sub_script_var] or styles[subscript_var] are non-zero. The shift_verb can be one of two values:

  ```
  typedef enum
  {
    baseline_relative,  // Draw from line's baseline
    percent_of_style    // Draw relative to percentage of baseline
  };
  ```

  For baseline_relative, values in styles[sub_script_var] or styles[subscript_var] are considered to be point (pixel) values; for percent_of_style, the super/subscript point values are computed as a percent (value of styles[sub_script_var] or styles[subscript_var]) of the style's total height (ascent + descent + leading). Example: If style's total height is 32 and styles[subscript_var] contained 50, the point value to shift the text will be 32 * 0.50, or 16.

- class_bits - Contains a set of bits defining specific attributes and behaviors for this style. The current attributes supported by OpenPaige are as follows:

  ```
  #define CANNOT_HILITE_BIT   0x00000000  // Can not highlight text of
  this style
  #define CANNOT_BREAK        0x00000002  // Chars can not break with this
  #define STYLE_IS_CONTROL    0x00000004  // Style is "control" type item
  #define GROUP_CHARS_BIT     0x00000008  // All chars selected as one
  #define STYLE_MERGED_BIT    0x00000010  // Style has been merged
  #define STYLE_IS_CUSTOM     0x00000020  // Style is understood only by
  app
  #define HILITE_RESTRICT_BIT 0x00000040  // Can not select outside of
  style
  #define CANNOT_WRAP_BIT     0x00000080  // Can not wrap (for tables,
  etc)
  #define IS_NOT_TEXT_BIT     0x00000100  // Data is not text at all
  #define REQUIRES_COPY_BIT   0x00000200  // Text copy requires copy_proc
  callback
  #define NO_SMART_DRAW_BIT   0x00000400  // Do not second-guess line
  drawing
  #define ACTIVATE_ENABLE_BIT 0x00000800  // Causes activate_proc to be
  called
  #define CANT_UNDERLINE_BIT  0x00001000  // The OS does not support
  underline
  ```

```
#define CANT_TRANS_BIT      0x00002000  // Text can't transliterate etc
#define RIGHTLEFT_BIT       0x00004000  // Text direction is RTL
#define VERTICAL_TEXT_BIT   0x00008000  // (unsupported - for )
#define TEXT_LOCKED         0x00010000  // (unsupported)
#define NO_EXTRA_SUPER_SUB  0x00020000  // (unsupported)
#define EMBED_SUBSET_BIT    0x00040000  // (for HERMES only)
#define NO_SAVEDOC_BIT      0x00080000  // Do not save this style_info
#define EMBED_INITED_BIT    0x00100000  // Used internally by embed_refs
```

Each of the above bits, if set, cause the following (only the bits currently supported are explained):

- `CANNOT_HILITE_BIT` causes highlighting *not* to show for the characters; even if the user does a "Select All", text with this style attribute will not highlight.
- `CANNOT_BREAK` is essentially a "non-breaking" style; characters with this attribute will not break in the middle (unless the line is too large).
- `STYLE_IS_CONTROL` causes the track-control low-level function to be called when a "mouse" click is received (see "Customizing OpenPaige").
- `GROUP_CHARS_BIT` causes all text in this style to be highlighted as one, i.e. a single click selects the whole group.
- `STYLE_MERGE_BIT` gets set by OpenPaige in "mail merge mode"; *do not* set this bit yourself.
- `STYLE_IS_CUSTOM` causes the text to be invisible *if* the standard display function is used. In other words, all text with this attribute will only display if you have provided your own display function.
- `HILITE_RESTRICT_BIT` forces a click/drag on this style to stay inside the boundaries of the style.
- `CANNOT_WRAP_BIT` causes text *not* to wrap regardless of width.
- `IS_NOT_TEXT_BIT` tells OpenPaige the character(s) aren't really text. If this is set, the standard text measuring and drawing functions do nothing (hence you would need to set your own hooks for both functions).
- `REQUIRES_COPY_BIT` causes the `copy_text_proc` (hook) to get called for these character(s); otherwise OpenPaige does not call this hook.

    - NO_SMART_DRAW_BIT disables the "second-guessing" for fast character display. If this bit is set, the whole text line is drawn (instead of a partial line).
    - ACTIVATE_ENABLE_BIT causes the style_activate_proc to get called, otherwise that hook is ignored.
    - CANT_UNDERLINE_BIT informs the text drawing function that the OS will not display an underline style (used for Kanji characters in Macintosh). **Obsolete!**
    - CANT_TRANS_BIT informs the "ALL CAPS" and "small caps" functions that the text can't be translated to upper/lower case. This bit might be important for text that is not really text, e.g. a pointer or memory reference.
    - RIGHTLEFT_BIT indicates the writing direction for the text is right-to-left.
    - NO_SAVEDOC_BIT causes this style_info not to be included in pgSaveDoc(). One reason you might want to do this is for special applications that want to construct their own styles or stylesheets without saving style_info to each file.
- style_sheet_id - Contains a unique ID used by style sheet support (see "Style Sheets").
- small_caps_index - You should not alter this field; it is used by OpenPaige when small_caps_var style is set.
- fg_color, bk_color - Define the foreground and background color of the text. Both fields are structured as follows:

```
typedef struct
{
  unsigned short red;   // Red composite
  unsigned short green; // Green composite
  unsigned short blue;  // Blue composite
  pg_short_t alpha;     // Optional value (machine dependent)
}
```

  NOTE: The *background* colour applies to the **text** background, not necessarily the **window** background. For example, a line of text drawn with a *blue* background colour on a white background window will result in a blue "stripe" of line height's size with the text foreground overlaying the stripe.

- `machine_var`, `machine_var2` – *Do not* alter these values; they are used internally by HERMES Paige.
- `char_width` – On **Windows**, this becomes the `ffWidth` value when setting up a `LOGFONT` for font selection.
- `point` – The point size for this style. This field is a `Fixed` type, i.e., the high-order word of the field is the integral part and the low-order word the fractional part of the value, if any. For more on setting point, see "Setting / Getting Point Size" and "Changing point size".
- `left_overhang`, `right_overhang` – These are a form of indent for characters. These fields control how far a style overhangs to the left and/or right, the best example being italic that can overhang to the right.

  NOTE: OpenPaige sets the default for these values when the style is initialised.
- `top_extra`, `bot_extra` – Contains extra leading, in pixels, to add to the top or bottom of the style.

  NOTE: You should use these fields–*not* the ascent/descent fields–for "add extra leading" features.
- `space_extra` – The fractional amount to add to each space width. This value is a `Fixed` value (high order word is the integral part and low order word the fractional part).
- `char_extra` – The fractional amount to add to each non-space character. This value is a "fixed" fraction (high order word is the whole part and low order word the fraction part) and can be used for kerning.

  NOTE: This field is only supported on **Macintosh** if **Color QuickDraw** exists.
- `user_id`, `user_data`, `user_data2` – Use these fields for an arbitrary setting. These are of particular utility for customising styles.
- `future - an array of longs` reserved for future enhancement. Do not use these fields.
- `embed_entry`, `embed_style_refcon`, `embed_refcon`, `embed_id`, `embed_object` – Do not alter these values; they are used by the TEXT-embed extension. See chapter on "Embedding Non-Text Characters".

- user_var – This can be used for anything. It is intended mainly, however, for source code users who want to append to the style_info record.
- procs – This is a record of many function pointers that get called by OpenPaige for drawing, text measuring, etc. The array of functions literally define the way this style behaves (OpenPaige will always call one of these functions to obtain information and/or to display text in this style). These are the essence and key to implementing special styles and text types. See "Customizing OpenPaige".
- maintenance, used_ctr – Both of these are used only by OpenPaige for internal maintenance and must not be altered (actually, you cannot alter them anyway; when calling pgSetStyleInfo, bOpenPaige ignores anything you put into these two fields).

User-defined styles, setting "invisible markers"

A style_info is said to be *user-defined* if one or more fields contain information understood only by the application. Usually, in all other respects the style looks and feels like any other OpenPaige style.

For example, your application might want to "mark" various sections of text with some special attribute, but invisibly to the (human) user. You can set invisible "marks" for various sections of text by merely applying a style_info to the desired text with any of the user fields set to something your app will understand. The user fields are user_id, user_data and user_data2, each usable for any purpose whatsoever.

font_info

```
typedef struct
{
  pg_char name[FONT_SIZE];            // "Name" of font
  pg_char alternate_name[FONT_SIZE];  // Alternate if first not found
  short   environs;                   // Machine-specific attributes
  short   typeface;                   // Typography class
  short   family_id;                  // Font ID code
  short   alternate_id;               // Alternate ID code if bad font
  short   char_type;                  // Char type (machine-specific)
  long    platform;                   // The platform this font originated
  long    language;                   // Language
  long    machine_var[8];             // Machine-specific array
  font_append_t user_var;             // Arbitrary use
}
font_info, PG_FAR *font_info_ptr;
```

The font_info record is somewhat machine-dependent and what should be placed in each field may depend on the platform you are running.

When you set a font_info record, usually only the name, alternate_name, and environs fields need be changed; this is because OpenPaige will initialise all the other fields to their defaults when the font is applied to a pg_ref.

One exception to this is setting a Windows font and you require a special character set and/or special precision information (see "Additional Font Info for Windows" below).

NOTE: On **Macintosh**, the font_info.name should be a pascal string terminated with the remaining bytes in font_info.name set to zero; the font_info.environs field should also be set to zero. For an example see "Responding to font menu (Macintosh)".

NOTE: On **Windows**, the font_info.name can be initially set to either a pascal string or a cstring, with all remaining bytes in font_info.name set to zero. Usually, due to Windows programming conventions, you will set the name to a cstring. In this case, before passing the font_info record to pgSetFontInfo, you must set font_info.environs to NAME_IS_CSTR.

CAUTION: On **Windows**, OpenPaige converts font_info.name to a pascal string and clears the NAME_IS_CSTR bit when the font is stored in the pg_ref. This is done purely for cross-platform portability. This is important to remember, because if you examine the font thereafter with pgGetFontInfo, the font name will now be a pascal string (the first byte indicating the string length), not a cstring.

- name - This should contain the name of the font. This can either be a pascal string (first byte is the length) or a cstring (terminated with zero). However, the assumption is made by OpenPaige that the string is a pascal string. Hence, you need to set the environs field accordingly if name is a cstring (see below).
- alternate_name – This should contain a font name to use as a second choice if the font defined in name does not exist. If OpenPaige can't find the first font, it will try using alternate_name. If you do not have an alternative, set alternate_name to all zeros.
- environs - Additional information about the font, which contains the following bit (or not):

```
#define NAME_IS_CSTR 1 // Font name is a cstring
```

All the other fields in `font_info` are initialised by OpenPaige when you set a font.

NOTE: Fill the font name with all zeros before setting the string. This will allow applications more easily to shift between `pascal` strings and `cstrings` (because a `pascal` string will also be terminated with a zero).

NOTE: For your reference, on **Macintosh**, the `family_id` will get initialised to the font ID and `char_type` will get set to the font script code (e.g., Roman, Kanji, etc.).

- `language` - This will contain the language ID for the font. In Windows NT and 95, this contains the langID and code page.

The remaining fields are not supported for any particular purpose and might be used for future enhancements.

Additional Font Info for Windows

In certain cases, it is necessary to map certain members of the font information to obtain the appropriate character set and drawing precision. The `machine_var` field in `font_info` is used for this purpose, the first four elements of which are defined as follows:

- `machine_var[PG_OUT_PRECISION]` should contain output precision.
- `machine_var[PG_CLIP_PRECISION]` should contain clipping precision.
- `machine_var[PG_QUALITY]` should contain output quality.
- `machine_var[PG_CHARSET]` should contain the character set code.

Setting LOGFONT precision info

```
/* This code snippet shows the members of LOGFONT you should map across to
font_info: */
font->machine_var[PG_OUT_PRECISION] = log_font -> IfOutPrecision;
font->machine_var[PG_CLIP_PRECISION] = log_font -> IfClipPrecision;
font->machine_var[PG_QUALITY] = log_font -> IfQuality;
font->machine_var[PG_CHARSET] = log_font->IfCharSet;
```

`par_info`

```
struct par_info
{
  short       justification;  // How text is justified
  short       direction;      // Primary text direction
  short       class_info;     // Used to define para attributes
  pg_short_t  num_tabs;       // Number of active tabs
  tab_stop    tabs[TAB_ARRAY_SIZE]; // Tab stop information
  long        style_sheet_id; // Used for style sheet features
  pg_fixed    def_tab_space;  // Default tab space
  pg_indents  indents;        // Line spacing
  pg_fixed    leading_extra;  // Extra leading of lines
  pg_fixed    leading_fixed;  // Fixed leading (0 = auto)
  pg_fixed    top_extra;      // Extra space at top
  pg_fixed    bot_extra;      // Extra space at bottom
  pg_fixed    left_extra;     // Extra space at left
  pg_fixed    top_extra;      // Extra space at right
  long        user_id;        // Can be used by app to ID custom para sizes
  long        user_data;      // Add'l space for app if par is custom
  long        user_data2;     // More space for app
  long        partial_just;   // Partial justify (future enhancement)
  long        future[PG_FUTURE];  // Reserved for future enhancement
  par_append_t  user_var;     // Arbitrary use
  pg_par_hooks  procs;        // Function pointers
  long        maintenance;    // Internal use
  long        used_ctr;       // Internal use
}
par_info, PG_FAR *par_info_ptr;
```

Field descriptions

- `justification` - The justification (alignment) for the paragraph. This value can be any of the following:

```
typedef enum
{
  justify_left,   // Align left
  justify_center, // Align centrally
  justify_right,  // Align right
  justify_full,   // Fully justify (pad spaces)
  force_left,     // Force left (notwithstanding writing direction)
  force_right,    // Force right (notwithstanding writing direction)
}
```

force_left and force_right are used to force an alignment to one side or the other regardless of the writing direction.

- direction – Defines the writing direction (left to right or right to left), and can be one of the following:

```
typedef enum
{
  right_left_direction = -1,  // Right-to-left
  system_direction, // System-default direction
  left_right_direction  // Left-to-right
}
```

  NOTE: The direction parameter defines the writing direction of the paragraph(s) affected by the par_info style. In such paragraphs, bidirectional scripts can exist such as English and Hebrew. While each script has its own direction attribute, the writing direction defines the point of origin for all lines in the paragraph. If writing direction is right-to-left, all text is aligned to the right; if writing direction is left to right, all text is aligned to the left. In both cases, however, individual blocks of text can go either direction relative to the text they are aligned to.

- class_info – Contains various bit setting(s) defining special attributes. Currently, the following attribute bits are supported:

```
#define KEEP_PARS_TOGETHER  0x0001  // Keep paragraphs on same page
#define NO_SAVEDOC_PAR      0x0200  // Don't save par_info in
pgSaveDoc()
```

- num_tabs, tabs – Define the tab stop(s). The tabs field contains an array of tab_stop records and num_tabs contains the number of valid elements. Tabs are described in "Tab Support".
- style_sheet_id – Contains a unique ID for paragraph style sheets (see "Style Sheets").
- def_tab_space – Defines the default tab spacing (when no preset tab stops exist). You can set this to anything.

  NOTE: The initial (default) setting is taken from pg_globals (see "Changing Globals" for more information about pg_globals).

- indents – These are the paragraph indentations; for information about indents see "Set Indents" and “Get Indents”.

- spacing - Defines the line spacing for the paragraph. This value is a mixed-number Fixed type in which the integral part is in the high-order word and the fractional part in the low-order word. This value is multiplied times the current line height (ascent + descent) and the result becomes the new height.

  For example, to obtain 2*1 line spacing, the spacing value should be 0x00020000. For 1.5*1 spacing, the value should be 0x00018000 (low-order word is 1/2 of an unsigned short).

  NOTE: A spacing value of zero implies "auto" spacing (lines spaced according to their style). You would also get the same effect if spacing = 0x00010000.
- leading_extra, leading_fixed - Both of these can also control line spacing. The leading_extra field is added to the line's height. The leading_fixed field, if non-zero, is forced as the line height. Both should never be set to non-zero at the same time since that would make no sense.
- top_extra, bot_extra, left_extra, right_extra - These are all added to the top, bottom, left and right of the paragraph, respectively.

  NOTE: These values are all pixel amounts (point) and they are added to the paragraph's boundaries in addition to everything else (in addition to indentations and spacing, etc.). Use these fields to obtain "space before" and "space after" for paragraphs.
- user_id, user_data, user_data2 - Your app can use these fields for anything it wants. These come in handy for customising paragraphs.
- partial_just, future - These are reserved for future enhancement. Do not alter these fields.
- procs - This is a record of many function pointers that get called by OpenPaige for paragraph formatting. The array of functions literally define the way this format behaves. See "Customizing OpenPaige".
- user_var - This can be used for anything. It is intended mainly, however, for source code users who want to append to the par_info record.
- maintenance, used_ctr - Both of these are used only by OpenPaige for internal maintenance and must not be altered (actually, you can't alter them anyway with pgSetParInfo - OpenPaige simply ignores anything you put into these two fields)

30.12 Creating a simple custom style

One of the most important features of OpenPaige is the ability to create custom styles. There are several issues to be understood when doing custom styles. They involve customising how OpenPaige draws and measures the text. This is accomplished by using hooks, described in "Customizing OpenPaige".

However, here simple custom styles can be created by changing just a few functions. The following example shows how to create a custom style that draws a box around some text. In this case, the only thing changing is how the text is drawn.

First of all, I must set the text to my custom style and install the hooks I will need. Second, I show how to initialize my style and my drawing hook. I even get to use the default OpenPaige functions for simply drawing the characters.

Other custom styles may have to use other custom hooks, including `measure_proc`, but nearly every custom style can be built changing only three:

1. The `measure_proc`. The (replaced) function must not only measure the character widths correctly, it must also fill in the `types` pointer (see "measure_proc").
2. The `text_draw_proc`. The (replaced) function must be able to draw the text on demand (see "text_draw_proc").
3. The `style_init_proc`. The (replaced) function probably needs to determine the style's ascent, descent and leading if that functionality for the character set in question does not already exist inherently in the OS. (See "style_init_proc").

NOTE: Many improvements could be made to this code, such as drawing a single box around the text when boxes are adjacent, setting the box so the offset on the left and right of the style is not right next to the first and last character, using the `styles[var]` amount for various offsets or widths of the line or both, and implement scaling.

Set some text to a custom style (Cross platform)

```
void SetBoxStyle (pg_ref pg)
{
  style_info style={0}; // or use pgInitStyleMask
  style_info mask={0};
  /* it is zero because I don't necessarily want to set everything, only the
procs I am interested in */
  style.styles[box_var] = -1;
  style.class_bits |= NO_SMART_DRAW_BIT;
  info → procs.init = box_init_proc;
  info → procs.draw = box_draw_proc;
  mask.procs.init = (style_init_proc) -1;
  mask.procs.draw = (text_draw_proc) -1;
  mask.class_bits = -1;
  mask.styles[box_var] = -1;

  pgSetStyleInfo(pg, NULL, &style, &mask, best_way);
  /* text inserted using pgInsert is now my custom boxed style */
}
```

Drawing a box around some text hook (Cross-platform)

```
/* This does the actual box and text drawing. */
/* Note: this does not handle multiple custom styles to do that we will need
to build our own myMasterDrawProc with the major changes being 1) a huge
if/then for each styles [], 2) possibly the order in which these are called,
and 3) that the pgDrawProc be called only once. */

static PG_PASCAL (void) box_draw_proc (paige_rec_ptr pg, style_walk_ptr
walker, pg_char_ptr data, pg_short_t offset, pg_short_t length,
draw_points_ptr draw_position, long extra, short draw_mode)
style_info_ptr original_style = walker → cur_style;
pg_scale_factor scale = pg → scale_factor; // this is not implemented

Point start_pt;
Point end_pt;

pgDrawProc(pg, walker, data, offset, length, draw_position, extra,
draw_mode);
/* OpenPaige's standard draw */
start_pt.h = pgLongToShort(draw_position → from.h);
start_pt.v = pgLongToShort(draw_position → from.v);
end_pt.h = pgLongToShort(draw_position → to.h);
end_pt.v = pgLongToShort(draw_position → to.v);

draw_a_box_around_rectangle (start_pt.h, start_pt.v - original_style →
ascent + 1, end_pt.h, end_pt.v + original_style → descent - 1 );
/* on Mac use FrameRect */
}
```

Figure out new line heights due to the box (Cross platform)

```
// This sets up the required info in the style record
static PG_PASCAL (void) box_init_proc (paige_rec_ptr pg, style_info_ptr
style, font_ptr_info font)
{
  register short distance;
  pgStyleInitProc(pg, style, font); // first call standard proc
  distance = style → styles[box_var];
  style → ascent += distance;
  style → descent += distance

  // style → right_extra += distance;
  // style → left_extra += distance;

  style → class_bits |= NO_SMART_DRAW_BIT;
}
```