27 CHAPTER

CUSTOMIZING OPENPAIGE

This section discusses the low-level "hook" structure of OpenPaige and general guidelines of customizing the look and feel of text display.

27.1 Standard Procs

```
(void) pgSetStandardProcs (pg_globals_ptr globals);
```

This function initializes all the standard low-level function pointers for styles, paragraph formats and general OpenPaige hooks.

The *globals* parameter should be a pointer to the same globals given to *pgInit* and *pgNew*. (Actually, globals can be a pointer to any *pg_global* record. For example, an application may want to set up different "versions" of globals or copies of globals; but the normal use of this function would be to reset the standard functions in the globals record currently in use).

The intended purpose of this function is to reinitialize all the standard functions after temporarily changing them to something else.

OpenPaige “Add-on” Extensions

For future OpenPaige extensions, *pgSetStandardProcs* will still work correctly even if some of the low-level functions require special function pointers from the extension(s) as the “standard.”

This is accomplished by the *extend_proc* which OpenPaige will now call every time *pgSetStandardProcs* is invoked (and after the standard functions are set). For this purpose the following values are possible for the *verb* parameter when *extend_proc* is called:

```
typedef enum
 {
   pg_std_procs,       /* Standard procs have been initialized */
   pg_new,                                /* pgNew has been called */
   pg_dispose                          /* About to do a pgDispose */
};
```

27.2 Style Functions

If you want to do anything at all with text styles or paragraph formatting that is beyond the scope (or different) than the way OpenPaige will normally handle formatting, you must first understand the following basic design concept on which OpenPaige is founded:

1. Every *style_info* record contains an extensive set of function pointers that, when called, perform all the associated tasks for that style, including text measurement (determining width of characters), display, height computation (ascent, descent, etc.), and information about characters (which char is a control code, which one is a line breaking char, etc.).

2. All new *style_info* records, by default, are initialized with OpenPaige standard functions: every function pointer gets set with the appropriate default procedure for that aspect of the style, none are null.
3. To alter the way a style behaves or draws, you simply replace the appropriate function pointer(s) within that *style_info* record with your own.

The end result of this method is that any section of text "knows how to measure and draw" itself.

27.3 Standard Low-Level Function Access

A subset of the above design concept is the public availability of OpenPaige standard low-level style functions.

An OpenPaige user has access to all the standard low-level functions that are normally placed into a *style_info* record by default. The reason this can be important is because a custom style function, which you create, has the ability to make a call to the "standard" function any time when it becomes necessary to facilitate your feature.

For example, suppose you had a requirement to create a "boxed" style which draws an outline around the text. To draw the text itself, you simply call the standard OpenPaige low-level function that draws text for any given style; then you would draw the outlining box. A great deal of coding time can be saved with this feature. See "Creating a simple custom style" on page 30-37 for a sample implementation.

The typical use for calling one of these functions is for situations where you only want to ALTER something only slightly from the standard and you want to avoid the necessity of writing huge code to replace the standard handling.

Another obvious example is using the *line_measure* function to build a line of text. As we have frankly stated in our own documentation, using this function to completely replace OpenPaige's standard handling is a mistake due to its enormous complexity. However, this function can become useful if you replace it with your own function that FIRST CALLS THE STANDARD FUNCTION to build the line; from that point you could alter the results. See "Anatomy of Text Blocks" on page 36-1 for additional information on line structures.

To call a standard low-level style function, simply include *"defprocs.h"*. The standard functions are listed with their corresponding hook below.

27.4 Setting Style Functions

Every *style_info* record contains the following structure as one of its components:

```
typedef struct
 {
   style_init_proc init;                          /* Initialize style_info */
   install_font_procinstall;          /* Set up "current" font & style */
   measure_proc measure;              /* Measure char positions */
   merge_proc    merge;                  /* Substitute other text */
   char_info_procchar_info;           /* Return info about a char */
   text_draw_procdraw;                /* Draw the character(s) */
   dup_style_procduplicate;           /* Style will get duplicated */
   delete_style_procdelete_style;     /* Style will get deleted */
   alter_style_procalter_style;       /* Style will get altered */
   save_style_procsave_style;         /* Style about to be written to
                                      disk */
   copy_text_proccopy_text;    /* Text of style will be copied */
   delete_text_procdelete_text;/* Text of style will be deleted */
   setup_insert_procinsert_proc;            /* Set up for insert */
   track_control_proctrack_ctl;    /* Track "control" type style*/
   style_activate_procactivate_proc;  /* Activate/deactivate */
}
 pg_style_hooks;
```

In each field shown above, a pointer to an appropriate function exists. Every style must contain these function pointers, but any (or all) styles can be different functions.

Hence, to customize a style, you need to decide which of these functions you need to "override," then change the pointer(s) in the style record so OpenPaige will call your function for that task.

NOTE (Windows 3.1 Users): Function pointers you use to override the OpenPaige functions must be created with *MakeProcInstance()*.

27.5 Setting / Changing Function Pointers

There are three general methods of changing function pointer(s) for styles.

1. Use pgSetStyleInfo

The first way is to simply use *pgSetStyleInfo*: set your mask parameter to all zeros except for the style functions you wish to change, and place pointers to your functions into the new *style_info* record (see "style_info" on page 30-21 for details about *pgSetStyleInfo*). Doing it this way you will literally apply your customization to whatever range(s) of text you prefer.

2. Set the style procs

The second method is to use the following function(s):

```
void pgSetStyleProcs (pg_ref pg, pg_style_hooks PG_FAR *procs,
    style_info_ptr match_style, style_info_ptr mask_style, style_info_ptr
    AND_style, long user_data, long user_id, pg_boolean inval_text,
    short draw_mode);

(void) pgSetParProcs (pg_ref pg, pg_par_hooks PG_FAR *procs,
    par_info_ptr match_style, par_info_ptr mask_style, par_info_ptr
    AND_style, long user_data, long user_id, pg_boolean inval_text,
    short draw_mode)
```

This function applies procs to all styles that match a specified criteria, as follows: each field of a *style_info* record in *pg* is compared to the same field in *match_style* if the corresponding field in *mask_style* is nonzero; in other words, only nonzero fields in *mask_style* are compared. Before each comparison is performed, the field in pg's *style_info* record is temporarily AND'd with the corresponding field in *AND_style* before the comparison.

Either *match_style*, *mask_style* or *AND_style* can be null, in which case the following occurs: if *match_style* is null, then all styles in *pg* are changed (no comparisons are made). If mask_style is null, every field is compared and must match; if *AND_style* is null, no AND'ing is performed.

Only styles that completely match the comparison criteria based on *match_style*, *mask_style* and *AND_style* are changed and, if so, all functions in procs are placed into that style.

Additionally, for every style that is changed, *user_data* and *user_id* are placed in the *style_info* record (see "style_info" on page 30-21 for more information about the *style_info* record).

If *inval_text* is TRUE, the text for which the changed styles apply is "invalidated" (tagged to require re-word-wrapping and line calculations).

If *draw_mode* is nonzero the text is redrawn (see"Draw Modes" on page 2-30).

This function is mainly used to restore all custom function pointers for "opened" files.

pgSetParProcs is 100% identical to *pgSetStyleProcs* except paragraph style records are changed.

3. Initialize function pointers

The third method is to initialize function pointers while a file is being "read". This is done using the file handler functions as described in "File Handlers" on page 34-1 (see also "Special "Initializing" Handlers" on page 34-23).

CAUTION: Warning: Style functions can not be null, ever. They must point to a valid function even if that function does nothing. As a general rule, if you don't need to change a style function, leave the default that OpenPaige has initialized.

4. Set “default” for all styles

This method should be used when you want to set one or more style functions to be used for all styles.

This is done by simply setting up the default style in pg_globals with the appropriate function pointer(s). The default style is called “*def_style*” and is a *style_info* record which OpenPaige will copy for every new style insertion.

NOTE (Windows 3.1 Users): Function pointers you use to override the OpenPaige functions must be created with *MakeProcInstance()*.

27.6 Function Definitions

The following is a definition and description of each of these functions, what each parameter means and general comments on how you might use the function to create special features.

style_init_proc

PURPOSE: To initialize the *style_info* record.

STANDARD FUNCTION: The default function, *pgStyleInitProc*, figures out the text ascent, descent and default leading; for the Mac version, it also figures out a 16-bit “style” word it can give to *QuickDraw* for the text style; for Windows it builds a *Font Object HANDLE*.

```
PG_FN_PASCAL (void, style_init_proc) (paige_rec_ptr pg, style_info_ptr
    style, font_info_ptr font);
```

The *style* record is to be initialized; the font parameter is a pointer to the *font_info* record associated with this style.

NOTE 1. Important: OpenPaige determines character heights solely from the information in the *style_info* record; specifically, *style_info*.ascent, *style_info*.descent and *style_info*.leading. Additionally, *top_extra*, *bot_extra* and superscript/subscript values will affect the height of characters. It is therefore important that you initialize these fields to reflect the appropriate vertical "bounding" areas of text drawn in that format.

NOTE 2. When computing the height for offscreen bitmap display, OpenPaige uses only the ascent, descent and leading fields. You can therefore avoid excessive memory usage for bitmap drawing by using *top_extra* and *bot_extra* to define "non text" boundaries such as picture frames.

install_font_proc

PURPOSE: To set the font and style as the "current font" for subsequent drawing.

STANDARD FUNCTION: The default function, *pgInstallFont*, sets the current device to the font, style, color and *char_extra* values all taken from the *style_info* and *font_info* records. If *include_offscreen* is TRUE, all the same values are set in the internal offscreen port as well. If scaling is enabled (not 1:1), the point size is scaled accordingly and the scaled point size is used instead.

```
PG_FN_PASCAL (void, install_font_proc) (paige_rec_ptr pg,
    style_info_ptr style,font_info_ptr font, style_info_ptr composite_style,
    pg_boolean include_offscreen);
```

The *pg* parameter is the *paige_rec* affected by this font. The *style* and *font* parameters point to the style and font record to install, respectively; neither will ever be a null pointer.

If *composite_style* is nonnull, this function must copy the contents of the *style_info* it actually used to set up the device (in certain cases, the original *style_info* may be temporarily altered, in the case of "superimpose" style variations, ALL CAPS, etc.).

If *include_offscreen* is TRUE, OpenPaige will want to use the same format when and if it draws to an offscreen bitmap during for the next text display (see "text_draw_proc" on page 27-501).

measure_proc

PURPOSE: To obtain the width of character(s).

STANDARD FUNCTION: The default function, *pgMeasureProc*, determines the character pixel positions as if each were drawn from left to right beginning at **positions*. Special flags are set in each corresponding element in **types*.

```
PG_FN_PASCAL (void, measure_proc) (paige_rec_ptr pg,
    style_walk_ptr walker, pg_char_ptr data, long length, pg_short_t slop,
    long *positions,short *types, short measure_verb, long current_offset,
    pg_boolean scale_widths, short call_order);
```

The walker parameter points to a *style_walk* record that contains pointers to the style and font to be measured.

NOTE: The paragraph formatting pointers in walker can be NULL or uninitialized.

data and *length* — define the text and length (in bytes), respectively.

slop — contains the number of "extra" pixels to include in the measurement, i.e., how much extra space to add for justification purposes (as in full justification).

positions —parameter points to an array of longs; the first long in that array will already be set to a number, which is the starting position, in pixels, for the first character in **data*; the remaining elements in **positions* will be uninitialized. The job of this function is to fill in the remaining longs with the position for all remaining characters as

they would appear on the screen if drawn from **positions* pixel position, including the ending position of the last byte.

For example, if the text had 3 bytes, each of them 8 pixels wide, and the first element in positions was 70, you would fill in the long's in positions as follows:

```
70, 78, 86, 92
```

types — parameter is a pointer to an array of shorts; what you must fill in into this array is the character type of each byte in data. The character type is defined by setting any (or all, or none) of the following bits:

```
#define SOFT_HYPHEN_BIT         0x00000008          /* Soft hyphen char */
#define CTL_BIT                 0x00000040      /* Char is a control code */
#define PAR_SEL_BIT             0x00000100     /* Char breaks a paragraph */
#define LINE_SEL_BIT            0x00000200 /* Char breaks a line (soft CR) */
#define TAB_BIT                 0x00000400       /* Char performs a TAB */
#define CONTAINER_BRK_BIT       0x00004000        /* Break-container bit */
#define PAGE_BRK_BIT            0x00008000         /* Page breaking char */
```

For each character, the corresponding **types* element should contain either zero (meaning the character is none of the above), or a combination of these bits.

EXAMPLE:

If there are 4 bytes to measure consisting of "a, b, c" followed by <CR>, the *types elements should be set to:

```
0, 0, 0, PAR_SEL_BIT | CTL_BIT
```

NOTE: types uses only the lower half of these #defines.

The *measure_verb* parameter indicates one of two reasons *measure_proc* is being called, which are:

```
typedef enum
{
   measure_width_locs,          /* Measure for char widths only */
   measure_draw_locs            /* Measure for relative drawing */
};
```

The difference between these two verbs is that *measure_width_locs* calls for the locations of characters to define the cascading width of each character, while *measure_draw_locs* calls for the literal locations (relative to horizontal screen **positions*) that characters will appear on the screen. For standard Roman text these two measurements are identical. However, for right-to-left scripts they are not identical.

For example, Arabic characters aligned as “abcd” in memory will display as “cdba” on the screen. Hence, if each character were 10 pixels wide, the relative screen locations should be 40, 30, 20, 10, 0; if measuring purely for character widths, however, the **positions* elements must be 0, 10, 20, 30, 40.

current_offset —parameter indicates the current offset into all text, which will match with the first byte of **data*. For example, if OpenPaige is measuring text at absolute byte offset 1200, *current_offset* will be 1200.

scale_widths — parameter indicates whether or not the text is currently scaled: if TRUE, the measured text will eventually be used to convert a screen point to a character position that is scaled to something other than 1:1.

NOTE: If *scale_widths* is TRUE that does not mean the character position must be scaled, rather that the function must compensate for potential errors in scaling.

For example, suppose a document is currently scaled by 50% and the text to be measured is 24 point. However, the relative screen locations of scaled-down 12 point text do not necessarily correspond to exactly 50% of 24-point text. Hence, there will be an inaccuracy if the text were being measured to compute a character from a scaled coordinate.

If *scale_widths* is TRUE, the Macintosh function computes the same scaled text it would normally draw in scaled mode, then upscales **positions*.

EXAMPLE:

If scaling is currently 50% and point size is 12, then 6 point text is measured then each resulting character position is multiplied X 2.

call_order — parameter indicates whether there will be one more *measure_proc* call for the current block of text or if this section of text is the end of the block: if *call_order* is -1 the text to be measured is the last part of the block.

NOTE: The "block" in this case is not necessarily all one style and in fact can be many styles (and potentially many different *measure_proc* functions). For Macintosh WorldScript measuring, it is necessary to know if *measure_proc* is being called for the last time in a sequence of bytes in order to set the last location of right-to-left script correctly. (Normally you won't care what the value of *call_order* is).

Multibyte Characters

The data given to *measure_proc* is always at a byte level, regardless of the character type. If you write your own *measure_proc*, it is your responsibility to return the correct responses anyway.

For example, suppose data had 10 bytes, but each byte represented 1/2 of a two-byte character. If each double-byte character were 8 pixels wide, you would fill *positions with something like this:

```
0, 0, 8, 8, 16, 16, 24, 24, 32, 32, 32
```

NOTE (Macintosh): The *style*, *font* and *point* size will have already been set in the current *GrafPort* before *measure_proc* is called.

NOTE: The *measure_proc* never gets called if the text is hidden (*styles[hidden_text_var]* is TRUE and "hide hidden text" is on).

PURPOSE: To temporarily substitute the text for this style with something else.

STANDARD FUNCTION: The default function, *pgMergeProc*, does nothing. This particular function is mainly for application features for "mail merge" and similar functionality.

```
PG_FN_PASCAL (short, merge_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);
```

This function gets called only when the application has called *pgMergeText* (see "Mail Merging" on page 29-1).

style — parameter points to the style for which this call is intended; *text_data* will point to the first character affected by the style and length the number of bytes given in *text_data*.

merged_data — parameter will be an empty *memory_ref*. The job of this function is to fill *merged_data* with whatever new text it wishes to insert instead of the text that already exists. If *merged_data* is set to zero size, the "old" text is deleted with no new substitute.

ref_con — parameter will contain a value provided by the application.

FUNCTION RESULT: If this function returns FALSE, the existing text in this style is not affected (stays the same), otherwise the existing text is replaced with the text in *merged_data* even if *merged_data* is empty.

See "Mail Merging" on page 29-1.

PURPOSE: To return specific information about a byte of text.

STANDARD FUNCTION: The default function, *pgCharInfoProc*, returns the various bit settings listed below for standard ASCII, Roman and non-Roman Macintosh character sets.

```
PG_FN_PASCAL (long, char_info_proc) (paige_rec_ptr pg,
    style_walk_ptr style_walker, pg_char_ptr data, long block_offset, long
    offset_begin, long offset_end, long char_offset, long mask_bits);
```

This function gets called at various times for OpenPaige to determine the nature of an arbitrary byte of text. Upon entry, the *walker* parameter points to a *style_walk_ptr* containing the following information:

```
typedef struct
{
    long                current_offset;   /* Current style offset position */
    style_info_ptr      cur_style;                    /* Current text style */
    par_info_ptr        cur_par_style;      /* Current paragraph style */
    font_info_ptr       cur_font;                   /* Current font record */
    style_run_ptr       next_style_run;       /* Next style run record */
    style_run_ptr       next_par_run;    /* Next paragraph run record */
    style_run_ptr       prev_style_run;          /* Previous style run */
    style_run_ptr       prev_par_run; /* Previous paragraph style run */
    style_info_ptr      style_base;                     /* used internally */
    par_info_ptr        par_base;                      /* used internally */
    font_info_ptr       font_base;                      /* used internally */
    pg_short_t          last_font;                      /* used internally */
    long                t_length;  /* Text size from original pg_ref */
    style_info          superimpose;/* Composite superimpose style */
}
style_walk;
```

The *style_walk* record is provided for this function's reference only; the *style_walk* structure is described in more detail under "Style Walkers" on page 32-1.

CAUTION 1: You must not alter the contents of walker directly.

CAUTION 2: The paragraph format pointers might be NULL.

data — parameter will point to one or more bytes; the byte in question is at offset *char_offset*. For example, if *char_offset* is 7, the byte being questioned is data[7]. This function only gets called to learn about a single byte at a time; for multibyte characters you must respond with the appropriate bits set to indicate what part of a character the byte in question represents.

block_offset — parameter will be the absolute offset, relative to all text in *pg*, for the first byte of the *text_block* record from which the text is derived. For example, if *block_offset* were 1900, the data can be found at the *text_block* record, within pg, whose begin member is 1900; the purpose of this parameter is for special-case character testing for multiple language scripts (*block_offset* helps *char_info* find out if the text prior to data is a different script).

However, if *block_offset* is NO_BLOCK_OFFSET (value -1), the data does not belong to any text block. Some examples of when this can occur would be a byte stream that is about to be inserted but not yet part of a *pg_ref*, or a group of characters within an *undo_ref*, etc.

offset_begin and *offset_end* — parameters indicate the range of byte(s) within data that are valid for purposes of looking "ahead" and "before" *data[char_offset]*. Usually, *offset_begin* will be zero and *offset_end* will be some *value >= char_offset*. The purpose of these parameters is for checking adjacent character(s) for determining double versus single byte status of a character.

mask_bits — parameter indicates which character type(s) need to be checked; the bits set in *mask_bits* correspond one for one to the function result bits given below.

For example, if *mask_bits* contained BLANK_BIT and CTL_BIT, then the only characteristic that needs to be checked for *data[local_offset]* is whether or not the

character is blank and whether or not it is a control character. The purpose of *mask_bits* is to optimize the performance of *char_info_proc* by providing a hint as to which data types can be ignored.

FUNCTION RESULT: The appropriate combination of the following must be the function result (but the bits don't need to be set if they were not set in *mask_bits*):

```
#define BLANK_BIT                0x00000001          /* Character is blank */
#define WORD_BREAK_BIT           0x00000002          /* Word breaking char */
#define WORD_SEL_BIT             0x00000004          /* Word select char */
#define SOFT_HYPHEN_BIT          0x00000008          /* Soft hyphen char */
#define INCLUDE_BREAK_BIT        0x00000010 /* Word brk but include w/ word */
#define INCLUDE_SEL_BIT          0x00000020 /* Select brk but include w/ word */
#define CTL_BIT                  0x00000040          /* Char is a control code */
#define INVIS_ACTION_BIT         0x00000080/* Char implies command or action*/
#define PAR_SEL_BIT              0x00000100       /* Char breaks a paragraph */
#define LINE_SEL_BIT             0x00000200     /* Char breaks a line (soft CR) */
#define TAB_BIT                  0x00000400          /* Char performs a TAB */
#define FIRST_HALF_BIT           0x00000800     /* 1st half of a multi-byte char */
#define LAST_HALF_BIT            0x00001000   /* Last half of a multi-byte char */
#define MIDDLE_CHAR_BIT          0x00002000      /* Middle of a multi-byte char */
#define CONTAINER_BRK_BIT        0x00004000            /* Break-container bit */
#define PAGE_BRK_BIT             0x00008000        * Break-repeating-shape bit */
#define NON_BREAKAFTER_BIT       0x00010000//Char must stay with char(s)after it
#define NON_BREAKBEFORE_BIT      0x00020000        /* Char must stay with char(s)
                                                              before it */
#define NUMBER_BIT               0x00040000             /* Char is numeric */
#define DECIMAL_CHAR_BIT         0x00080000       /* Char is decimal (for decimal
                                                                          tab) */
#define UPPER_CASE_BIT           0x00100000          /* Char is UPPER CASE */
#define LOWER_CASE_BIT           0x00200000           /* Char is lower case */
#define SYMBOL_BIT               0x00400000            /* Char is a symbol */
#define EUROPEAN_BIT             0x00800000         /* Char is ASCII-European */
#define NON_ROMAN_BIT            0x01000000        /* Char is not Roman script */
#define NON_TEXT_BIT             0x02000000          /* Char is not really text */
#define FLAT_QUOTE_BIT           0x04000000          /* Char is a "flat" quote */
#define SINGLE_QUOTE_BIT         0x08000000       /* Quote char is single quote */
#define LEFT_QUOTE_BIT           0x10000000           /* Char is a left quote */
#define RIGHT_QUOTE_BIT          0x20000000          /* Char is a right quote */
#define PUNCT_NORMAL_BIT         0x40000000       /* Char is normal punctuation */
#define OTHER_PUNCT_BIT          0x80000000        /* Char is other punctuation */
```

NOTE: Important: OpenPaige, as a whole, knows nothing at all about "character types" and therefore relies utterly on the information provided by *char_info_proc*:

For an example see "Control characters don't draw" on page 24-12.

text_draw_proc

PURPOSE: To draw the text for a style.

FUNCTION RESULT: The default function, *pgDrawProc,* draws the text using the standard styles and color as defined in the *style_info* record, plus any other styles that are appropriate (such as strikeout, word underline, etc.).

```
PG_FN_PASCAL (void, text_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);
```

walker — parameter contains information about the text style, font and paragraph formatting (for more information about the *style_walk* record see "char_info_proc" on page 27-498 and "Style Walkers" on page 32-1).

data, *offset* and *length* — provide the pointer to text, offset into that text to start, and the number of bytes to draw, respectively. All text parameters are in bytes; if the text is something other than bytes it is this function's responsibility to work that out.

The text is guaranteed to never be zero length, and is guaranteed to not contain any "blanks" that should not be drawn (such as tabs, CR's, etc.). Only text for this style will be given: the data will never cross style boundaries nor will the text ever cross line boundaries (it will always be given for a whole line or a portion of a line).

The *draw_position* is a pointer to a record that defines exactly where to draw the text as follows:

```
typedef struct
{
    co_ordinate     from;                                   /* Draw from */
    co_ordinate     to;                                     /* Draw to */
    long            real_offset;        /* Actual offset into all text */
    long            ascent;           /* Distance from baseline to top*/
    long            descent;     /* Distance to baseline to bottom */
    long            line_offset;         /* Offset where line begins */
    long            compensate_h;// Amount of x-axis compensation
    point_start_ptr  starts;           /* Current point_start record */
    shape_ref       bitmap_exclude;    /* Exclusion rects or NULL */
    co_ordinate     bitmap_offset;         /* Offset to "real" point */
    co_ordinate     vis_offset;                 /* Total amount point
                                                starts were offset */
    text_block_ptr  block;                     /* Current text block */
}
draw_points, *draw_points_ptr;
```

The text is to be drawn at coordinate *draw_position->from*; after drawing, *draw_position->to* must be set to the ending "pen" position (ending coordinate where the text stops). The *real_offset* field contains the actual byte position (within *pg*) of the text to be drawn relative to all text in the *pg_ref*.

The *ascent* and *descent* — fields define the line's height from the current vertical position (which will be *draw_position->from.v*).

compensate_h — field is currently not used (but reserved for future enhancement).

starts — field is a pointer to the current *point_start* record. The value in *starts->flags* can be useful for special drawing as any of the following might be set in this field:

```
#define LINE_BREAK_BIT              0x8000      /* Line ends here */
#define PAR_BREAK_BIT               0x4000      /* Paragraph ends here */
#define SOFT_PAR_BIT                0x2000      /* Soft CR ends line */
#define RIGHT_DIRECTION_BIT         0x1000      /* Text is right to left */
#define LINE_GOOD_BIT               0x0800      /* This line requires no recalc */
#define NEW_LINE_BIT                0x0400      /* New line starts here */
#define NEW_PAR_BIT                 0x0200      /* New paragraph starts here */
#define WORD_HYPHEN_BIT             0x0100      /* Draw a hyphen after this text */
#define TAB_BREAK_BIT               0x0080      /* Tab char terminates this line */
#define HAS_WORDS_BIT               0x0040      /* One or more word breaks exist */
#define CUSTOM_CHARS_BIT            0x0020      /* Style(s) are custom */
#define SOFT_BREAK_BIT              0x0010      /* Start breaks on soft hyphen */
#define BREAK_CONTAINER_BIT         0x0008      /* Line breaks for next container */
#define BREAK_PAGE_BIT              0x0004      /* Line broke on page break */
#define LINE_HIDDEN_BIT             0x0002      /* Line is invisible (hidden) */
#define NO_LINEFEED_BIT             0x0001      /* Line does not advance vertically */
#define TERMINATOR_BITS             0xFFFF      /* ALL ONES if terminator record */
```

bitmap_offset — field indicates how far the display is offset from its screen position to draw within the offscreen bitmap (if that is the drawing mode).

vis_offset — field indicates how much was added to the display to obtain the real screen position; usually, this is the "scroll position" of the text.

block — field contains a pointer to the current *text_block* record.

The two parameters in *text_draw_proc* — þ`extra` and *draw_mode* — þdefine the amount of space (in pixels) to make up for justification and the current drawing mode, respectively.

NOTE (Macintosh): On scaling for Macintosh: If text is scaled, the fields *draw_position* will already reflect the scaling (i.e., the coordinates will already be scaled) and the point size will also have been scaled and set appropriately. Therefore you do not need to do anything "special" for scaling in this function.

NOTE: On blank characters: THE SIZE OF TEXT WILL ALWAYS EXCLUDE "INVISIBLE" CHARACTERS, including trailing spaces. Example: If a 4-byte line of text were 3 bytes followed by a carriage return, the *length* parameter

passed to *text_draw_proc* will be 3 (not 4). Also, if a line of text has embedded tab characters (or other similar controls), multiple calls to *text_draw_proc* will be made, each of which omit the tab characters. The function is not called at all if omitting non-displaying chars results in zero length.

NOTE: More about blank characters: "Blank" characters are determined to be so as a response from *char_info_proc*. OpenPaige knows nothing about the nature of characters. Hence, if you embed a stream of characters for special purposes, you can declare each of them as "visible" (or not) simply by writing your own *char_info_proc*.

Rather than actually drawing a character, sometimes all you need do is change the character attributes. See "Control characters don't draw" on page 24-12.

dup_style_proc
delete_style_proc

PURPOSE: To duplicate any structure(s) in a *style_info* record, or to delete any structure(s) in the *style_info* record.

STANDARD FUNCTION: The default functions, *pgDupStyleProc* and *pgDeleteStyleProc*, do nothing (there are normally no allocated structures within a *style_info* record). These functions are mainly for special application features.

```
PG_FN_PASCAL (void, dup_style_proc) (paige_rec_ptr src_pg,
    paige_rec_ptr target_pg, short reason_verb, format_ref all_styles,
    style_info_ptr style);
PG_FN_PASCAL (void, delete_style_proc) (paige_rec_ptr pg,
    pg_globals_ptr globals,short reason_verb, format_ref all_styles,
    style_info_ptr style);
```

Whenever OpenPaige duplicates (copies) a *style_info* record internally, *dup_style_proc* is called. This would give the application a chance to duplicate any structures it may have placed in the *style_info* record. For duplication, both source and target *paige_rec's* are given. The source will be the original OpenPaige document from

which the item is being copied; the target is the *paige_rec* for which the copy is being applied. Either might be NULL depending on the reason the function is being called.

Whenever OpenPaige is about to delete a *style_info* record, *delete_style_proc* is called giving the application a chance to dispose any structures it may have placed in the *style_info* record.

globals is sometimes used in cases where *pg* can be NULL if style records are getting disposed from an *undo_ref*, but Windows version must have a pointer to the globals when a style is deleted.

For both duplicate and delete functions, a *verb* parameter is given to indicate the reason the function is being called, which will be one of the following:

```
enum
{
    prepare_undo_text_reason,
    prepare_undo_style_reason,
    prepare_undo_typing_reason,
    undo_delete_reason,
    undo_style_reason,
    copy_reason,
    paste_reason,
    for_next_insert_reason,
    new_stylesheet_reason,
    internal_clone_reason,
    not_used_reason,
    pgdispose_reason,
    disposeundo_reason,
    delete_text_reason,
    pg_new_reason
}
```

A good example of this would be if a picture structure were kept in one of the *style_info* fields, and if OpenPaige made a copy of the style, you would want to also duplicate the picture; for deletion, you would want to delete the picture, and so forth.

Neither function gets called when you “set” a style for the first time, but rather it is called for other situations such as Copy, Paste and deleting all the text for a style.

The parameter *all_styles* is the *memory_ref* containing all the *style_info* records (of which the *style* parameter is a part of). The purpose of this parameter is to provide the array of style records for special situations where *pg* is a null pointer (see below).

CAUTION: In these particular functions, *pg* can be a null pointer. This is because both of these functions can get called in relation to “undo” and “redo” functions that are not associated with a *pg_ref* at the time they are required. If you need to examine the array of *style_info* records, use the *all_styles* parameter.

Source and destination pg_refs when duplicating or deleting

I'm doing some custom style work and I'm trying to follow the logic of how the hooks are called for duplicate and delete, etc. I understand that sometimes the pg_ref to these hooks is passed as NULL. However, it seems to me that the duplicate call should have BOTH the source and destination pg_refs, and BOTH the source and destination style records.

I can best answer all these questions by explaining what the duplicate/delete functions are all about.

The only real purpose of the duplicate and delete function hooks in a style is to handle memory structures that have been embedded in the style itself. Otherwise these hooks are not needed and shouldn't be used.

The style_info record is always “copied” or “removed” by OpenPaige as needed, without you doing anything. However, if you have embedded some kind of Handle or Pointer or *memory_ref*, etc. inside the *style_info* you would need to be informed of duplications/deletions so you can handle the memory allocation(s).

For example, let's say you create a custom style that contains as one of its fields a *PicHandle*. You initially created this style something like:

```
style_info      my_style;
style_info.user_var = (long) MyPicture; /* put in a Picture handle*/
```

As long as this *style_info* sits inside the *pg_ref* as-is, no problem. Later on, however, the user does *pgCopy* which causes

OpenPaige to make a duplicate of this particular *style_info*. In such a case, you would wind up with the same PicHandle in each copy of the style -- which would be a disaster. Hence, you need to be told that OpenPaige just made a copy of your *style_info* record and you would need to make a unique *PicHandle* to avoid this problem, something like:

```
HandToHand((Handle*) &style_info.user_var);
```

If *"user_var"* were not a memory allocation you would NOT CARE that OpenPaige made a copy.

The same is true for deletions: if OpenPaige decides to dispose a *style_info* record (which ALWAYS occurs during *pgDispose*) you would need to be told so you could remove memory allocations, as:

```
KillPicture((PicHandle) style_info.user_var);
```

That is really all there is to it.

As to when and why OpenPaige calls these functions, it doesn't matter. There are lots of reasons and I've only named the obvious ones. All you need to know is:

1. The "duplicate" function gets called after OpenPaige has made an exact copy of a *style_info* for whatever reason.

2 The "delete" function gets called just before OpenPaige removes a *style_info* record for whatever reason.

Now, how can a *pg_ref* be "NULL" when these functions get called? That only occurs when the target *pg_ref* does not exist. The only situation where this exists is setting up an *undo_ref*, in which case there isn't any target *pg_ref*.

PURPOSE: To alter the style as necessary as it is about to be modified as "offspring" of some other file. For example, if the user selected text that was currently in Helvetica 12-point text, then changed the style to bold, OpenPaige creates a new style_info record exactly the same as Helvetica 12-point but with the bold attribute set. In this way, new styles can be "offspring" of parent styles and *alter_style_proc* gets called with the original style and offspring so custom styles can be altered appropriately.

STANDARD FUNCTION: The default function, *pgAlterStyleProc*, does nothing. This function is normally used for custom styles if the customized elements need to be changed before becoming offspring.

```
PG_FN_PASCAL (void, alter_style_proc) (paige_rec_ptr pg,
    style_info_ptr old_style, style_info_ptr new_style,
    style_info_ptr mask);
PG_FN_PASCAL (void, alter_par_proc) (paige_rec_ptr pg, par_info_ptr
    old_par, par_info_ptr new_par);
```

Upon entry, *old_style* is the "parent" (original) style and *new_style* is the altered "offspring" (new) style.

For *alter_style_proc*, the mask will indicate which field(s) of the *style_info* are actually being altered. Only the nonzero fields as indicated in mask will actually be changed.

A typical reason for using *alter_style_proc* is the following scenario: A custom style is developed by the application that places a "box" around text. The box effect is accomplished purely by changing the *draw_proc* to draw a frame around the surrounding text. The user, however, selects a range of text to become "plain," in which case you might want to clear out your custom function pointers. The *alter_style* proc would be best suited for this purpose: you could check the *new_style* to see if it is your custom "box" style and, if so, reinitialize all the function pointers to their defaults (so your box doesn't draw any more).

alter_par_proc is identical to *alter_style_proc* except it does not have a *"mask"* parameter and is called for paragraph formats.

copy_text_proc
delete_text_proc

PURPOSE: To duplicate any structure(s) in within the text stream associated with a *style_info* record, or to delete any structure(s) in the text for *style_info* record.

NOTE: Standard text characters are automatically copied or deleted by OpenPaige. The purpose of this function is to copy anything that really isn't text, such as embedded picture allocations, controls, etc.

STANDARD FUNCTION: The default functions, *pgCopyTextProc* and *pgDeleteTextProc*, do nothing (there are normally no allocated structures within the text stream). These functions are mainly for special application features.

```
PG_FN_PASCAL (void, copy_text_proc) (paige_rec_ptr src_pg,
    paige_rec_ptr target_pg, short reason_verb, style_info_ptr style, long
    style_position, long text_position, pg_char_ptr text, long length);
PG_FN_PASCAL (void, delete_text_proc) (paige_rec_ptr pg, short
    reason_verb, style_info_ptr style, long style_position, long
    text_position, pg_char_ptr text, long length);
```

These functions are similar to *dup_style_proc* and *delete_style_proc* except they are intended for structures the application might have embedded into the text stream for the "style" in question. They get called for exactly the same reasons (and occurrences) as the *dup_style_proc* and *delete_style_proc*, but for the text stream.

On entry, *style_position* is the byte offset where the style begins (byte offset relative to all text within the *pg_ref*). The *text_position* parameter is the offset of the text in relation to all text in the *pg_ref* (this is not the offset of the text given in the text parameter, rather the absolute position from the first byte in *pg*). The *text* parameter is a pointer to the text to be copied or deleted and the *length* parameter is the number of bytes that *text* is pointing to.

The *reason_verb* for both functions indicates the reason for the function call; this value will be one of the values listed above for *dup_style_proc* and *delete_style_proc*.

For *copy_text_proc*, the text is being copied from *src_pg* to *target_pg*.

NOTE: Important: Neither of these functions get called unless "REQUIRES_COPY_BIT" is set in the *class_bits* field of the *style_info* record.

CAUTION: In these particular functions, the *paige_rec_ptr* param(s) can be a null pointer. This is because both of these functions can get called in relation to "undo" and "redo" functions that are not associated with a *pg_ref* at the time they are required.

setup_insert_proc

PURPOSE: To set up for the next text insert. Custom style features sometimes require a "warning" before a new text insertion.

STANDARD FUNCTION: The default function, *pgSetupInsertProc*, does nothing. This function is mainly intended for custom style features, but OpenPaige will alter this hook when/if an *embed_ref* is inserted.

```
PG_FN_PASCAL (short, setup_insert_proc) (paige_rec_ptr pg,
    style_info_ptr to_be_inserted, long position);
```

The main reason for this low-level function is to avoid "extending" a style beyond the range of what makes sense. A perfect example of this is when pictures have been embedded into the text stream as a "style". If the user selected the style at the end of the picture, you would not want the style to extend along with the text; rather, you would want to null out the "picture" part of the style and force OpenPaige to insert a normal, non-picture format from that point forward. The *setup_insert_proc* give you a chance to do that.

Upon entry, *to_be_inserted* is the style that would affect the next text, if you did nothing and *position* is the text position for the text insertion (relative to all text in pg).

If you change anything in *to_be_inserted* (which you are allowed to do), return TRUE from the function result. If *to_be_inserted* is untouched, return FALSE.

track_control_proc

PURPOSE: To track a "mouse" across a "control" type style.

NOTE: A style is considered a control simply if its *class_bits* field contains "*STYLE_IS_CONTROL*" setting.

STANDARD FUNCTION: Usually, the default function, *pgTrackCtlProc,* does nothing, because this function only gets called if *class_bits* indicate it is a control style.

```
PG_FN_PASCAL (long, track_control_proc) (paige_rec_ptr pg, short
    verb,  t_select_ptr first_select, t_select_ptr last_select, style_walk_ptr
    styles, pg_char_ptr associated_text, point_start_ptr bounds_info,
    short modifiers, long track_refcon);
```

This function gets called by OpenPaige only from within *pgDragSelect* (text click/drag) and works precisely as follows: when text is clicked for the first time (*pgDragSelect* called with *verb = mouse_down*), the *class_bits* field of the style applied to the character being "clicked" is examined for *STYLE_IS_CONTROL*; if that bit is set, *track_control_proc* gets called. Then, if *track_control* returns any nonzero result, OpenPaige "remembers" to make subsequent calls to *track_control* during every subsequent call to *pgDragSelect* for *announcers* until (and including) *mouse_up* is indicated.

NOTE: If *track_control* returns zero from its initial (*mouse_down*) call, it will not get called again until the next time *pgDragSelect* receives a *mouse_down* and the style's *class_bits* contain *STYLE_IS_CONTROL*.

Conversely, if *track_control_proc* returns a nonzero response during mouse_down (indicating that it is indeed a "control" as far as the application is concerned), *pgDragSelect* will neither track nor highlight nor "auto scroll" any text in the normal fashion. Rather, each new call to *pgDragSelect* will simply determine the location of

the new selection point and pass that information to *track_control_proc*. In other words, once this function indicates the style is a "control" the application becomes responsible for handling the *mouse_down*, *mouse_moved* and *mouse_up* activity.

For every *track_control_proc* call, the *verb* parameter contains the same verb given to *pgDragSelect*. The *first_select* and *last_select* parameters will contain selection information for the initial *mouse_down* point and the new (most recent) point, respectively (see record structure below).

styles — parameter is a pointer to a *style_walk* record containing the style information for the first, original selection point (which will be, of course, the same style for which *track_control_proc* is being called).

The text for which the (original) style is applied is given in *associated_text*.

NOTE: Associated text is a pointer to the first byte of text for which the style applies, not necessarily the character that is being "clicked."

bounds_info — parameter is a pointer to a *point_start* record that defines the boundaries of the portion of line originally selected. Among the fields in bounds_info that is probably the most useful is *bounds_info->bounds*, which is a rectangle defining the precise bounding area for the top and bottom of the text line and the left and right side of the character(s) enclosed by the style. However, this information is neither scrolled nor scaled.

modifiers and *track_refcon* — parameters are one and the same values given in the most recent call to *pgDragSelect*.

FUNCTION RESULT: The initial call to *track_control_proc* (*pgDragSelect* got called for *mouse_down*) should return any nonzero result if the selection point is indeed a control and the application wishes to continue tracking it through more calls to *track_control_proc*.

For every call to *track_control_proc*, *pgDragSelect* returns the same function result. Hence your application can detect when a "control" has been clicked by what is returned from *pgDragSelect*.

Selection Information

To obtain information about the first selection point (or most recent selection point) you can examine *first_select* or *last_select*, both which point to the following structure:

```
typedef struct
{
    long            offset;                 /* Absolute text offset */
    select_pair     word_offsets;           /* Original word offsets if
                                               applicable */
    co_ordinate     original_pt;        /* Original point of selection */
    pg_short_t      line;                       /* Point start number */
    short           flags;              /* Contains internal attributes */
    long            control_offset;     /* Offset for purposes of tracking
                                               control */
    long            section_num;    /* Section ID (reserved for future) */
    long            primary_caret;  /*Relative primary direction caret*/
    long            secondary_caret;    /* Relative secondary caret */
}
t_select;
```

Among the fields in *t_select*, probably the two of interest to the *track_control_proc* are *control_offset* and *original_pt*: the *control_offset* field contains the text position that corresponds to the selection coordinate. The *original_pt* contains the *co_ordinate* that was used to determine the selection point.

NOTE: Normally the *"offset"* field contains the selection position of text; when the *track_control_proc* is in progress, however, you should examine *control_offset* instead as it may be different than the actual text position.

NOTE: Important: The *original_pt* is not necessarily one and the same value given to *pgDragSelect* since it is backwards adjusted to the document's current scrolled position and scaling values.

By "backwards adjusted" is meant the following: Before *pgDragSelect* determines the *text* selection that corresponds with the *co_ordinate* given to it, the original *co_ordinate* values are first offset by the *(positive) vertical* and *horizontal* scrolled positions of the document; then the *co_ordinate* is reverse scaled to the document's

scaling factor, e.g. if the document is currently scaled by 1/2, the *co_ordinate* is upscaled by 2/1, or if the document is currently scaled by 2/1 the *co_ordinate* is reversely scaled 1/2, etc.

Hence to obtain the "real" *co_ordinate* of, say, *first_select -> original_pt* from your *track_control_proc*, perform the following:

```
co_ordinatereal_pt;

    real_pt = first_select -> original_pt;
    pgScaleLong( - pg -> scale_factor.scale, pg->scale_factor.origin.h,
    &real_pt.h);
    pgScaleLong( - pg->scale_factor.scale, pg->scale_factor.origin.v,
    &real_pt.v);
    pgOffsetPt(&real_pt, pg->scroll_pos.h, pg -> scroll_pos.v);
```

Checking for Arrow Selection(s)

OpenPaige performs a "shift-arrow" selection by emulating a shift-click internally. When using the control tracking hook, your application can become confused by this action since the hook will be called if the user performs shift-arrow over a "control" style.

To avoid this problem, the modifiers parameter will contain the following attribute if arrow keys are causing the selection (instead of the mouse):

ARROW_ACTIVE_BIT

```
#define ARROW_ACTIVE_BIT0x8000/* Arrow key is actively down */
```

PURPOSE: To activate or deactivate "control" styles when the *pg_ref* changes its highlight state.

STANDARD FUNCTION: The standard function, *pgActivateStyleProc*, does nothing.

```
PG_FN_PASCAL (void, style_activate_proc) (paige_rec_ptr pg,
    style_info_ptr style, select_pair_ptr text_range, pg_char_ptr text,
    short front_back_state, short perm_state, pg_boolean show_hilite);
```

Whenever the active/deactive state changes within a *pg_ref*, this function gets called for every style that has ACTIVATE_ENABLE_BIT set in *class_bits*. Note that the style is usually a "control" style that your application defined but does not really need to be a control.

The situation that causes this function to be called is (a) the document changes active/deactive states by virtual of *pgSetHiliteStates*, and (b) the style's *class_bits* contain ACTIVATE_ENABLE_BIT.

Upon entry, the *style* parameter is a pointer to the *style_info* record.

ext_range _ parameter defines the beginning and ending range of text for which this style applies, and text is a pointer to the first byte in the text stream for which this style applies.

f*ont_back_state* and *perm_state* — are the same values as given in *pgSetHiliteStates*; note that *pgSetHiliteStates* is the only way a *pg_ref* can change from active to inactive or visa versa.

If *show_hilite* is TRUE — the new highlight state is to be drawn, otherwise this function should not change anything on the screen.

```
PG_FN_PASCAL (void, style_activate_proc) (paige_rec_ptr pg,
    style_info_ptr style, select_pair_ptr text_range, pg_char_ptr text,
    short front_back_state, short perm_state, pg_boolean show_hilite);
```

This hook gets called during *pgSaveDoc*, just before *style_to_save* gets written to the file.

The intended purpose of this function is to change something in *style_to_save* before each of its fields get written to the file; when the *style_info* record is read later, the field(s) will contain the altered contents, if any.

An example of using this function would be to replace a memory structure that is stored in the *style_info* record with something that can be recognized to restore that structure later when the file is opened. If a pointer to a picture were placed in one of the *style_info* fields, for instance, the application might want to change that to some type of "picture ID" reference so it can restore the appropriate picture later.

NOTE: The actual *style_info* record within the *pg_ref* is not altered, only the record that is written to the file is altered.

STANDARD FUNCTION: The standard *save_style_proc* does nothing.

27.7 Paragraph Style Functions

Paragraph formats —þ*par_info* records —þalso contain their own set of low-level function pointers. For each *par_info*, OpenPaige sets default functions in the following record:

```
typedef struct
{
    line_glitter_procline_glitter;          /* Draw ornaments, line */
    tab_measure_proctab_width;          /* Return the tab position */
    tab_draw_proctab_draw;                      /* Draws leaders */
    dup_par_proc duplicate;             /* Style will get duplicated */
    delete_par_procdelete_par;          /* Style will get deleted */
    alter_par_proc alter_par;           /* Style will get altered */
}
pg_par_hooks;
```

PURPOSE: To draw "ornaments" such as lines over paragraphs, line numbers, paragraph numbers, etc.

```
PG_PASCAL (void) line_glitter_proc (paige_rec_ptr pg, style_walk_ptr
    walker, long line_number, long par_number, text_block_ptr block,
    point_start_ptr first_line, point_start_ptr last_line, point_start_ptr
    previous_first, point_start_ptr previous_last, co_ordinate_ptr
    offset_extra, rectangle_ptr vis_rect, short call_verb);
```

This function gets called after each text line is drawn; information is available from the function parameters to determine whether or not the line is the beginning of a paragraph, the ending of a paragraph, and whether or not a page breaking characters terminates the line.

walker — parameter points to a *style_walk* record which contains pointers to the current text and paragraph style (see "Style Walkers" on page 32-1).

line_number and *par_number* — will contain the line and paragraph number that just displayed, respectively. Both of these numbers are one-based, i.e. the top line or paragraph of *pg* is 1, the second line is 2, etc.

CAUTION: The line and paragraph numbers will be incorrect unless you have set COUNT_LINES_BIT as one of the *attributes flags* in the *pg_ref*.

block — parameter points to the *text_block* record containing the text for the line (if you need to access the text, see "Accessing Text" below).

first_line and *last_line* — parameters point to the first and last *point_start* records that compose the line (the format of an OpenPaige line is composed of one or more records called a *point_start* —þsee "Line Records" on page 36-4 for information on this structure). Most of the information your application needs to know about the line is contained in one of these two parameters —þsee "Determining Line Type" below).

previous_first and *previous_last* — parameters point to the previous line's beginning and ending point_start records, *or they are null pointers* if there is no previous line.

offset_extra — parameter points to a *co_ordinate* record whose h and v fields indicate the distance, in pixels, the text was adjusted when drawn; given the horizontal and vertical positions of the line as defined in its point_start record(s), OpenPaige will have added *offset_extra->h* and *offset_extra->v* to those locations when it drew the text (see "Determining Bounding Rectangle" below).

call_verb — indicates the nature of the function call, which will be one of the following

:

```
enum
{
    glitter_bitmap_draw,
    glitter_post_bitmap_draw,
    glitter_normal_draw
};
```

glitter_bitmap_draw —þThe line has been drawn through OpenPaige's offscreen bitmap; the bits have not yet been stamped to the screen.

glitter_post_bitmap_draw —þThe *line_glitter* function has been called a *second time*, after it stamped the bits to the screen in offscreen drawing mode.

glitter_normal_draw —þThe line has been drawn directly to the screen.

NOTE: If *call_verb = glitter_post_bitmap_draw*, the function would have been called once already for the same line. The purpose of this *call_verb* is to give the application a chance to draw directly to the screen after OpenPaige has displayed the offscreen bits. If *call_verb = glitter_normal_draw*, however, offscreen drawing did not occur.

Determining Line Type

first_start and *last_start* — parameters point to the beginning and ending records that make up a line of OpenPaige text. Each of these are type *point_start*:

```
typedef struct
{
   pg_short_t     offset;                    /* Position into text */
   short          extra;          /* Tab record if &0xC000 == 0 */
   short          baseline;     /* Distance from bottom to draw */
   pg_short_t     flags;                /* Various attributes flags */
   long           r_num;/* Wrap rectangle record where this sits */
   rectangle      bounds;    /* Point(s) that enclose text piece
                                          exactly */
 }
 point_start,  *point_start_ptr;
```

It is possible that *first_start* and *last_start* will be the same (both might point to the same record). For a single line of text, for example, that contains no style changes and no tab characters, OpenPaige maintains only one *point_start*; if the line contains style changes and/or tabs, a *point_start* record separates each style or tab separation.

The field you will probably need to examine the most often within a *point_start* is *flags*, which contains various bit settings that indicate almost everything you would need to know about the line, such as whether the line begins a new paragraph, whether it ends a paragraph, whether or not it breaks on a CR character or a page break character, etc.

The following bit values that exist in *"flags"* for a *point_start* record indicate the anatomy of the line:

```
#define LINE_BREAK_BIT          0x8000      /* Line ends here */
#define PAR_BREAK_BIT           0x4000      /* Paragraph ends here */
#define SOFT_PAR_BIT            0x2000      /* CR ends line */
#define RIGHT_DIRECTION_BIT     0x1000      /* Text in this start is right to left */
#define LINE_GOOD_BIT           0x0800      /* This line requires no recalc */
#define NEW_LINE_BIT            0x0400      /* New line starts here */
```

```
#define NEW_PAR_BIT             0x0200      /* New paragraph starts here */
#define WORD_HYPHEN_BIT         0x0100      /* Draw a hyphen after this text */
#define TAB_BREAK_BIT           0x0080      /* Tab char terminates this line */
#define HAS_WORDS_BIT           0x0040      /* One or more word separators exist */
#define CUSTOM_CHARS_BIT        0x0020      /* Style(s) are known only to app
                                               (so don't play games with display) */
#define SOFT_BREAK_BIT          0x0010      /* Start breaks on soft hyphen */
#define BREAK_CONTAINER_BIT     0x0008      /* Line breaks for next container */
#define BREAK_PAGE_BIT          0x0004      /* Line broke for whole repeater shape */
#define LINE_HIDDEN_BIT         0x0002      /* Line is invisible (hidden text) */
#define NO_LINEFEED_BIT         0x0001      /* Line does not advance vertically */
#define TERMINATOR_BITS         0xFFFF      /* Flagged only as terminator record */
```

Some of these bits might be set in either *first_start* or *last_start*, while others will only be set in *first_start*, still others will only be in *last_start*.

The following code examples demonstrate common tests for flags:

Testing for new paragraph

```
if (first_start->flags & NEW_PAR_BIT)
{
                    // Line begins a paragraph
}
```

Testing end (last line) of paragraph:

```
if (last_start->flags & PAR_BREAK_BIT)
{
                    // Last line of paragraph
}
```

Testing for soft page break

```
if (last_start->flags & BREAK_PAGE_BIT)
{
                    // Page break comes after this line (SOFT page break
char)
}
```

Testing for hyphenated word

```
if (last_start->flags & WORD_HYPHEN_BIT)
{
        // Line is hyphenated (see note below)
}
```

Regarding WORD_HYPHEN_BIT: This bit will only be set if your application has provided a hyphenate hook and that hook has indicated a hyphenation break; except for "soft hyphen" characters, OpenPaige will never automatically hyphenate.

- or, for soft hyphens

```
if (last_start->flags & SOFT_BREAK_BIT)
{
        // Line ends on soft hyphen character
}
```

Regarding WORD_HYPHEN_BIT: This bit will only be set if your application has provided a hyphenate hook and that hook has indicated a hyphenation break; except for "soft hyphen" characters, OpenPaige will never automatically hyphenate.

Determining Bounding Rectangle

It is common to want the bounding rectangle and/or the top or bottom dimensions of a line. This is accomplished by examining the bounds field of either *first_start* or *last_start*, or both.

When doing so, always add *offset_extra* to the side(s) of the bounding area to determine the exact drawing location. The following code examples show typical methods of determining bounding dimensions:

Line's top

```
lineTop = first_start->bounds.top_left.v + offset_extra->v;
```

NOTE: Both *first_start* and *last_start* will have the same top and the same bottom, each reflecting the line's top and bottom

Line's Bottom

```
lineBottom = first_start->bounds.bot_right.v + offset_extra->v;
```

Line's left side

```
lineLeftSide = first_start->bounds.top_left.h + offset_extra->h;
```

Line's right side

```
lineRightSide = last_start->bounds.bot_right.h + offset_extra->h;
```

Paragraph(s) versus Line(s)

Your application might use the line glitter hook to place ornaments around, or on the top of "paragraphs," yet this might not appear immediately intuitive since this hook is line-oriented (gets called for every line of text).

However, since an OpenPaige paragraph is merely composed of one or more lines, by interrogating the flags field in *first_start* or *last_start* you can immediately learn what portion of the paragraph the line belongs to.

For example, if NEW_PAR_BIT is set in *first_start->flags*, the line is first in a paragraph, while if *last_start->flags* has PAR_BREAK_BIT set, the line is last in a paragraph.

NOTE: Both NEW_PAR_BIT and PAR_BREAK_BIT can be set if the "paragraph" is composed of only one line.

Finding "Real" Text Position

The feature you are implementing with line glitter might require you to learn the actual text position of the line, relative to the beginning of all text in the document. To do so, simply add *first_start->offset* to the text block's begin field.

Real text position

```
longtextPosition;
    textPosition = (long) first_start->offset; // compilers often need this
    coercing
    textPosition += block->begin;  // Add the block's begin, = real text
    position
```

Line's Text Length (or ending position)

To learn how many bytes compose the line, subtract the offset value in the element AFTER *last_start* from the offset value in *first_start*.

Line length

```
pg_short_t      lineTextSize;
lineTextSize = last_start[1].offset - first_start->offset;
```

NOTE: *last_start[1]* is guaranteed to exist even if *last_start* defines the end of the whole document, because OpenPaige always appends at least one *point_start* defining the ending position of all text.

To obtain the "real" text offset of the line's end, add the element after *last_start* to the block's "begin" value:

Offset of line end

```
longendTextPosition;

    endTextPosition = (long)last_start[1].offset;
    endTextPoistion += block->end;
```

ACCESSING TEXT

If it is necessary to examine the actual text of a line, you can do so by getting a pointer to *block->text* and its first byte in *first_line->offset*:

```
// The following example shows how to look at the text in the line:
    pg_char_ptr    text;

    text = UseMemory(block->text);
    text += first_start->offset;// Points to FIRST BYTE of line
// .. be sure when you are through with “text” you call:
    UnuseMemory(block->text);
|
```

tab_measure_proc
tab_draw_proc

PURPOSE: To measure the distance required for a tab and to draw “tab leaders.”

STANDARD FUNCTION: The default function for *tab_measure_proc*, *pgTabMeasureProc*, returns the appropriate distance for every tab; the default *tab_draw_proc*, *pgTabDrawProc*, draws tab leaders if they exist.

```
PG_FN_PASCAL (long, tab_measure_proc) (paige_rec_ptr pg,
    style_walk_ptr walker, long cur_pos, long cur_text_pos, long line_left,
    pg_char_ptr text, pg_short_t text_length, long *char_positions,
    pg_short_t PG_FAR *tab_rec_info);
PG_FN_PASCAL (void, tab_draw_proc) (paige_rec_ptr pg,
    style_walk_ptr walker, tab_stop_ptr tab, draw_points_ptr
    draw_position);
```

For *tab_measure_proc*, *walker* is the current *style* and paragraph format; *cur_pos* is the current text width of the characters in the line preceding the tab; *current_text_pos* is the text position within *pg* that contains the tab. The *line_left* parameter indicates the position where the line started, in pixels (which would include paragraph indentation, etc.).

text — parameter is a pointer to the text character immediately following the tab being measured and *text_length* holds the number of bytes remaining in that text. The *char_positions* parameter is a pointer to the character pixel positions that correspond to the text bytes (see *"measure_proc" on page 27-493* for more information about character positions).

tab_info_rec — parameter points to the extra field of the current *point_start* record (see *"line_glitter_proc" on page 27-517* for information about *point_start*). If the tab position does not correspond to any physical *tab_stop* record, **tab_info_rec* should get cleared to zero; otherwise **tab_info_rec* should be set to *tab_stop* element number OR'd with 0x8000. (Example: If tab corresponds to element 3 in the *tab_stop* array, **tab_info_rec* should be set to 0x8003). This function should return the "width" of the tab character that, if added to *cur_pos*, would hit the appropriate tab position.

For *tab_draw_proc*, *walker* is the current style/paragraph info, *tab* is the *tab_stop* record and *draw_position* will contain the screen positions for the end of the text just drawn and the start of the tab position (hence, the two points to draw a tab leader). See "text_draw_proc" on page 27-501 for a description of a *draw_points* record.

dup_par_proc
delete_par_proc

PURPOSE: To duplicate any memory allocations, if any, that are present in the *par_info* record.

NOTE: The record itself is automatically duplicated by OpenPaige. The purpose of this function is to make copies of memory allocations that are embedded in the record.

STANDARD FUNCTION: In *pgDupParProc* the *memory_ref* containing a list of tabs is duplicated. In, *pgDeleteParProc*, they are deleted. Nothing else is cop-

ied or deleted (since there are no other memory structures in a standard *par_info* record).

```
PG_FN_PASCAL(void, dup_par_proc) (paige_rec_ptr src_pg,
    paige_rec_ptr target_pg, short reason_verb, par_ref all_pars,
    par_info_ptr par_style);
PG_FN_PASCAL (void, delete_par_proc) (paige_rec_ptr pg, short
    reason_verb, par_ref all_pars, par_info_ptr par_style);
```

These functions do exactly the same thing as *dup_style_proc* and *delete_style* proc except for *par_info* records. The *all_pars* parameter is the *memory_ref* containing all the paragraph formats (of which the *style* parameter is a part).

CAUTION: In this function it is possible that the *paige_rec_ptr(s)* will be null. This will happen if either function is getting called from the "undo" and "redo" function for which there is no *pg_ref* associated. If you need to examine all the *par_info* records use *all_pars*.

27.8 "Global" OpenPaige Low-level Hooks

OpenPaige also has an additional set of low-level hooks that apply to all text and styles for that OpenPaige object; you can also replace any of these to enhance customized features (or, if you want all *pg_ref's* to assume various functions other than the standard, you can change the default functions in *pg_globals*):

```
typedef struct
{
   line_init_proc          line_init;              /* Initialize line measure */
   line_measure_proc       line_proc;              /* Measure a line */
   line_adjust_proc        adjust_proc;            /* Adjust a line */
   line_validate_proc      validate_line;          /* Validate a line */
   line_parse_proc         parse_line;             /* Change length for parsing */
   hyphenate_proc          hyphenate;              /* Hyphenate word */
   hilite_rgn_proc         hilite_rgn;             /* Make highlight region */
   draw_hilite_proc        hilite_draw;            /* Draw (invert) highlight */
   text_load_proc          load_proc;              /* Load text for text_block */
   text_break_proc         break_proc;             /* Find break in text block  */
   draw_cursor_proc        cursor_proc;            /* Draw a caret */
   pt2_offset_proc         offset_proc;            /* Find offset of point */
   font_init_proc          font_proc;              /* Set up font_info */
   special_char_proc       special_proc;           /* Special character draw */
   auto_scroll_proc        auto_scroll;  // Called for auto-scrolling  during drag
   scroll_adjust_proc      adjust_scroll;          /* Adjust for scroll */
   draw_scroll_proc        draw_scroll;            /* Draw for scroll */
   draw_page_proc          page_proc;              /* Called to draw "page" */
   bitmap_modify_proc      bitmap_proc;            /* Modify offscreen drawing */ wait_proc;
   wait_proc;              /* Called for long crunches */
   enhance_undo_proc       undo_enhance;           /* Custom undo's */
   par_boundary_proc       boundary_proc;          /* Find par/word  boundary */
   change_container_proc   container_proc;         /* Alter container */
   smart_quotes_proc       smart_quotes;           /* Do smart quotes */
   post_paginate_proc      paginate_proc;   /* Called after a block paginates */
```

```
    increment_text_proc     text_increment;    /* Called when text is
                                                    inserted or deleted */
    click_examine_proc          click_proc;/* Called to look at clicked item */
    set_device_proc             set_device;      /* Called before, after using a
                                                    display device */
    page_modify_proc        page_modify; /* Called to let app modify each page
                                                    rect */
    wordbreak_info_proc     wordbreak_proc;/* Called to decide on word break
                                                    chars */
    charclass_info_proc     charclass_proc; /* Called to modify pgCharInfo() for
                                                    chars */
    key_insert_queryinsert_query;  // Called to get yes, no to buffer insertion
}
pg_hooks;
```

Setting pg_hooks

```
void pgSetHooks (pg_ref pg, pg_hooks PG_FAR *hooks, pg_boolean
    inval_text);
void pgGetHooks (pg_ref pg, pg_hooks PG_FAR *hooks);
```

To get the current *pg_hooks*, call *pgGetHooks* and the function pointers are copied to hooks.

To set new ones, call *pgSetHooks* with hooks containing new function pointer(s). The hooks <only in> *pg* are changed. If *inval_text* is TRUE, all text in *pg* is "invalidated" (marked to recalculated, reword wrap).

NOTE (Windows 3.1 Users): Hooks must be set from the result of *MakeProcInstance()* unless the function exists within a DLL.

NOTE: You can set the hook for all *pg_refs* by changing OpenPaige globals *"def_hooks"*.

CAUTION: No function pointers can be null, all must be valid. If you or OpenPaige attempts to access a proc that is NULL you will crash.

The names of the standard functions used by OpenPaige can be found in "Calling Standard Functions".

Setting a proc

The following is an example of setting a *pg* function pointer for *"wait_proc"* but using the defaults for all other function pointers.

```
/* This is an example of setting a single function pointer for a pg_ref. The
   parameter "the_proc" is a pointer to a wait_proc function.  */

void set_wait_proc (pg_ref pg, wait_process_proc the_proc)
{
   pg_hooks       hooks;

   pgGetHooks(new_doc.pg, &hooks);
   hooks.wait_proc = the_proc;
   pgSetHooks(new_doc.pg, &hooks, FALSE);
}
```

line_measure_proc

PURPOSE: To compute a line of text by setting up an array of *point_start* records.

STANDARD FUNCTION: The default function, *pgLineMeasureProc*, computes a line of text, breaks on the appropriate word break and handles any "exclusion" that may exist (overlapping portions with *exclude_area*).

```
PG_FN_PASCAL (void, line_measure_proc) (paige_rec_ptr pg,
    pg_measure_ptr measure);
```

NOTE: This function requires highly complex handling and **we do not recommend you use it**. There are many alternative methods to force a line to calculate for customizing effects.

See also, "Anatomy of Text Blocks" on page 36-1 and "Standard Low-Level Function Access" on page 27-487 for better understanding of this area.

For alternatives to this see the next function pointer, *line_adjust_proc*, below.

line_adjust_proc

PURPOSE: To adjust a line (by adjusting an array of *point_start* records) after the line has been calculated. The intended purpose of this function is move an entire line somewhere else, for widows and orphan, or "keep- paragraphs-together" features, for instance. (See "Advanced Text Placement" on page 37-1).

STANDARD FUNCTION: The default function, *pgLineAdjustProc*, adjusts the line for non-left justification. Applications can use this function to move *point_start* records around for any purpose.

CAUTION: The *point_start* records must only be moved: new records must not be "inserted" nor can records be "deleted."

```
PG_FN_PASCAL (void, line_adjust_proc) (paige_rec_ptr pg,
    pg_measure_ptr measure, point_start_ptr starts, pg_short_t
    num_starts, rectangle_ptr line_fit, par_info_ptr par_format);
```

measure — parameter contains all the information about the line that was just built.

starts — parameter is a pointer to the first *point_start* for the line just computed, and *num_starts* indicates how many *point_starts* are in that array (which represent the whole line). The *line_fit* parameter contains the maximum rectangle that the line had to fit inside, and *par_format* is the current paragraph format.

NOTE: If you alter the bounding dimensions or location of the line in any way, be sure to also change *measure_info->actual_rect* to reflect the change.

TECH NOTE

Line leading

I need to implement the space between lines differently than OpenPaige does it now. How can I do this?

The only way I can think of to bypass line leading is to use the *"line_adjust_proc"* and change the physical baseline(s) of each line record once a line is figured out.

The default *line_adjust_proc* hook is used to alter the line for *justification*, but you can also use it to change the line leading.

When this gets called, you should:

1. First call *pgLineAdjustProc* itself (which is prototyped in *defprocs.h*) so OpenPaige can do its thing.

2. Then walk through *line_starts* for *num_starts* elements and make adjustments you need.

   The *"line_starts"* param points to one or more *point_start* records (it points to *num_starts* records). The line "leading" will be reflected in either *line_starts->bounds* (which defines the enclosing rectangle for that part of the line), and/or in *line_starts -> baseline* (which defines the distance from *bounds.bot_right.v* where the text baseline sits).

3. For convenience, *par_format* is the current *par_info* for this line. You can examine fields in this format if you need to.

4. The *line_fit* param contains the overall rectangle for the whole *line*. MAKE SURE you adjust its height to the same dimension you changed the *line_starts -> bounds*, if any. This is important, because when you return from the hook, OpenPaige uses the bottom of *line_fit* to know where the next line begins vertically.

PURPOSE: To verify that a built line of text is "valid," requiring no change of location, dimension or any other form and if so return TRUE. A result of FALSE tells OpenPaige to compute the text line over again. Hence, this low-level hook can be used to alter the form of a line under various conditions.

STANDARD FUNCTION: *pgLineValidate* verifies that the new line fits within the boundaries of the current part of the *page_area* and does not intersect with any part of the exclusion area. If line fails to meet this criteria and *line_validate_proc* can't correct the line by mere adjustment, the function alters the line's parameters as necessary (such as adjusting the maximum width, changing to a new vertical position, etc.) and tells OpenPaige to recalculate the line by returning FALSE.

```
PG_FN_PASCAL (pg_boolean, line_validate_proc) (paige_rec_ptr pg,
    pg_measure_ptr measure_info);
```

The typical purpose of this hook would be to alter the form of a line after the "normal" line has been calculated.

For example, suppose the inclusion of a special character causes a line to dynamically change its physical location and/or its maximum allowable width. Using *validate_line_proc*, the necessary adjustments can be made, and if the function returns FALSE, OpenPaige will rebuild the line with the new information.

The *measure_info* parameter is a pointer to a large record structure which offers all available information about the line of text just computed; these are the fields you would

alter should the line need to be recalculated from new information. The *pg_measure* record is defined as follows:

```
typedef struct
{
    short           previous_flags;          /* Ending flags at last line's end */
    short           prv_prv_flags;           /* Previous flags before above */
    short           wrap_dimension;          /* Bits defining how complex wrap is */
    pg_boolean      repeating;               /* TRUE if shape is a repeater */
    rectangle       extra_indents;           /* Any extra indents for hooks to alter */
    long            line_text_size;          /* Total use of text in line */
    long            max_text_size;           /* Maximum text line can use */
    long            extra_width;             /* Excess width not used by line */
    style_walk_ptr          styles;          /* Pointer to the style_walk */
    text_block_ptr          block;           /* Current text block */
    point_start_ptr         starts;          /* Next point_start *record */
    pg_short_t              starts_ctr;      /* Number of starts remaining */
    pg_short_t              num_starts;      /* # starts this line */
    long PG_FAR             *char_locs;      /* Current character locations */
    short PG_FAR            *char_types;     /* Current character types */
    long PG_FAR             *positions;      /* Original character locations */
    short PG_FAR            *types;          /* Original character types */
    rectangle               fit_rect;        /* Rect in which line must fit */
    rectangle               actual_rect;     /* Actual rect enclosing line */
    rectangle               wrap_bounds;     /* Bounding rect for wrap area */
    line_ref                starts_ref;      /* Memory_ref of starts */
    memory_ref              tab_info;        /* Contans tab_width_info */
    rectangle_ptr           wrap_r_base;     /* Base for shape (first rect) */
     rectangle_ptr          wrap_r_begin;    /* Top wrap rectangle */
     rectangle_ptr          wrap_r_end;      /* End wrap rectangle */
    long                    r_num_begin;     /* Current wrap-target rectangle */
    long                    r_num_end;       /* Ending rect of line */
    long                    end_r;           /* Ending record for wrap rects */
    memory_ref              exclude_ref;     /* Holds "exclusion" rectangles */
    long                    wrap_r_save;     /* Saves old bottom */
    co_ordinate             repeat_offset;   /* Amount to add for repeat */
    rectangle               prev_bounds;     /* Previous start's bounds */
    long                    hook_refcon;     /* Custom hooks can use this */
    long                    minimum_left;    /* Minimum left side */
    long                    maximum_right;   /* Maximum right side */
```

```
    pg_boolean              quick_paginate;                 /* Only moving lines */
    pg_short_t              old_offset;         * Ending offset from old line end */
}
pg_measure, PG_FAR *pg_measure_ptr;
```

For the sake of simplicity and clarity we will discuss only the fields that would most likely apply to a custom *line_validate_proc*.

repeating — is TRUE if the *page_area* in *pg* is a repeating shape.

extra_indents — extra pixel amounts to inset to the top, left, bottom and right of the line. By default, these are all zero but can be changed by hook(s) to adjust a line's bounding dimensions. For example, to force a line to fit within a smaller width, *extra_indents.top_left.h* and/or *extra_indents.bot_right.h* could be changed.

NOTE: *extra_indents* are inset values, i.e. *extra_indents.top_left* is added to the potential line's top-left bounds and *extra_indents.bot_right* is subtracted from the potential line's bottom-right.

line_text_size — þnumber of text bytes in this line.

max_text_size — the maximum number of text bytes the line can use from the main stream of text. This is not necessarily the total text bytes available, rather it is an optional maximum for special features to restrict all lines to, say, 80 characters. The *max_text_size* field might be useful if your *line_validate_proc* decided the line should be smaller: the *max_text_size* field could be reduced and the line forced to recalculate.

styles — pointer to the current *style_walk* which will hold information for the current style and paragraph formats.

block — pointer to the current *text_block* record (for which this line belongs).

starts — pointer to the next *point_start* record after the line being validated.

NOTE: A "line" in OpenPaige consists of one or more *point_start_records*; the starts field will be the next *point_start* record should the next line be calculated.

(To get the first *point_start* of the line being validated, subtract *measure_info -> num_starts* from the *measure_info->starts* pointer).

num_starts — number of *point_start* records in the line being validated. Since the starts field (above) points to the NEXT (not current) *point_start*, the first *point_start* of the line in question is *measure_info->starts - measure_info -> num_starts*.

fit_rect, actual_rect — contain the maximum rectangle for which the line must be contained and the actual bounding rectangle of the line after it was computed, respectively. These fields could prove useful in determining the potential and actual bounding rectangles for the line and/or to change the maximum dimensions and force a recalculation.

wrap_r_base — a pointer to the list of rectangle in the page area. For example, if the document had three "container" rectangles,

measure_info->wrap_r_base — would point to an array of those three rectangles.

r_num_begin — the rectangle index of the page area this line is contained in. By "rectangle index" is meant the nth rectangle of the page area shape. For repeating shapes, the index is a modulo value representing the rectangle number of the shape X page number (example: for a 3-column page area, a value of "0" represents the first column of the first page; a value of "3" would be the first column of the second (repeating) page, etc.). The rectangle index is zero-based. The actual rectangle that contains the line just calculated can be determined by first determining how many rectangles there are in the page area and indexing the array of rectangles:

```
measure_info->wrap_r_base[measure_info->r_num_begin %
    num_rects];
(where num_rects is number of rectangles in the page area).
```

hook_refcon — can be used for anything you choose. OpenPaige initially sets this to zero and does not alter it while lines are being calculated.

NOTES:

1. If you alter the bounding dimensions or location of the line in any way, be sure to also change *measure_info->actual_rect* to reflect the change.

2. *measure_info->fit_rect's* height is often undetermined, i.e. OpenPaige only cares about its top, left and right sides; *measure_info->actual_rect*, on the other hand, will contain the true dimensions of the line.
3. If you want to force a different maximum bounding area and/or the line's vertical position, change *pg_measure->fit_rect* (not *pg_measure-> actual_rect*). If you return FALSE from line_validate_proc, OpenPaige will recalculate the line based on the (new) information in *pg_measure-> fit_rect.*

hyphenate_proc

PURPOSE: To compute a word break at the end of a line.

STANDARD FUNCTION: The default function, *pgHyphenateProc*, figures out where to break a word, including handling soft hyphen characters if they exist. (A "soft hyphen" is a control character imbedded in the text stream that is normally invisible, but defines a word break if the enclosing word will wrap).

```
PG_FN_PASCAL (pg_boolean, hyphenate_proc) (paige_rec_ptr pg,
    text_block_ptr block, style_walk_ptr styles, pg_char_ptr block_text,
    long line_begin, long *line_end, long *positions, short *char_types,
    long PG_FAR *line_width_extra, pg_boolean zero_length_ok);
```

This function gets called whenever a word at the end of a line will not fit. However, the term "word" in this case really means the next character, if added to the line of text, would overflow the maximum allowed width; there might not be any real "word" at all.

Nonetheless, it is the responsibility of this function to return the word break whether or not there are "real words" and whether or not any hyphenation is to be implemented (see Function Result below regarding actual hyphenation).

Upon entry, block is the current *text_block* record and styles is a pointer to a *style_walk* record which will be set to the style affecting the first byte following the character that overflowed the line.

The *block_text* parameter is a pointer to all the text in block, and *line_begin* contains the offset into that text where the line begins, while *line_end* points to the offset of text after the first byte that caused the line to overflow.

EXAMPLE : If the text “abcdefg” overflows the maximum line width after the “e”, then **line_end* will contain the offset of “f” (first byte after “e”).

NOTE: *line_end* will always be the offset after only one byte of overflow; hence, correct word breaking and/or hyphenation can assume that **line_end - 1* is the maximum text position for which the line can end.

When this function returns, it must have set **line_end* to the correct location.

The *positions* and *char_types* parameters point to the character positions and the character types (both obtained from the *measure_proc*) for all character in text (the **position* for start of the line would be *positions[line_begin]*). See “measure_proc” on page 27-493.

If *zero_length_ok* is TRUE, it is acceptable to return a “word break” that results in no text at all; i.e. the word won’t fit on a line and cannot be divided. However, if *zero_length_ok* is FALSE, this function must break the text so at least one character exists on the line.

**line_width_extra* value should be set by your function to the pixel width of the hyphenation character “-” if any.

NOTE: OpenPaige uses *pg_globals.hyphen_char* as the hyphenation character.

FUNCTION RESULT: If you want the line to be drawn with a “-” (hyphenation char), return TRUE, otherwise return FALSE. Note that you must still break the word by setting **line_end*: the function result simply indicates whether or not the line must now include a hyphenation symbol to be drawn.

hilite_rgn_proc
draw_hilite_proc
draw_cursor_proc

PURPOSE: To compute the highlight region, draw a highlight region and to draw a “caret,” respectively.

STANDARD FUNCTION: The default functions *pgHiliteProc*, *pgDrawHiliteProc*, and *pgDrawCursorProc* do each of the above.

```
PG_FN_PASCAL (void, hilite_rgn_proc) (paige_rec_ptr pg, t_select_ptr
    selections, pg_short_t select_qty, shape_ref rgn);
PG_FN_PASCAL (void, draw_hilite_proc) (paige_rec_ptr pg, shape_ref
    rgn);
PG_FN_PASCAL (void, draw_cursor_proc) (paige_rec_ptr pg,
    t_select_ptr select, short verb);
```

For *hilite_rgn_proc* — selections contains an array of *select_qty t_select* record pairs from which to compute the highlight region). The region must be returned in *rgn*, which is a standard OpenPaige shape.

For *draw_hilite_proc* — the highlighting in *rgn* is to be drawn; this function must adjust the region for any scrolled positions and scaling factors (neither of those have been considered when computing *rgn*).

For *draw_cursor_proc* — selections contains the information as to where the cursor should go and *verb* is the cursor drawing mode. The cursor position and height needs to be computed from the information in select.

text_load_proc

PURPOSE: To initialize the text within a *text_block*.

STANDARD FUNCTION: The default function, *pgTextLoadProc*, does nothing. The intended purpose of this low-level hook is to implement "text paging" from a file.

```
PG_FN_PASCAL (void, text_load_proc) (paige_rec_ptr pg,
    text_block_ptr text_block);
```

This function gets called any time OpenPaige wants to do a *UseMemory* on

text_block->text. Hence, a text-paging feature is given the chance to load the text for this block.

A text block record follows:

```
typedef struct
{
    long                begin;          /* Relative offset beginning */
    long                end;            /* Relative offset ending */
    rectangle           bounds;         /* Entire area this includes */
    text_ref            text;           /* Actual text data */
    line_ref            lines;          /* Point_start run for lines */
    pg_short_t          flags;          /* Used internally by OpenPaige */
    short               extra;          /* (reserved for future) */
    pg_short_t          num_lines;      /* Number of lines */
    pg_short_t          num_pars;       /* Number of paragraphs */
    long                first_line_num; /* First line number */
    long                first_par_num;  /* First par number */
    point_start         end_start;      /* Copy of ending point_start in block */
    memory_ref          isam_end_ref;   /* (Reserved for DSI) */
    tb_append_t         user_var;       /* Can be used for anything */
}
text_block, *text_block_ptr;
```

For more information about text blocks, see “Anatomy of Text Blocks” on page 36-1.

CAUTION: In the above record, only the text field should be changed by this function (it should be filled with the appropriate text).

pt2_offset_proc

PURPOSE: To compute a text offset belonging to a specified Coordinate.

STANDARD FUNCTION: The default function, *pgPt2OffsetProc*, computes the text offset from a point received in *pgDragSelect*.

```
PG_FN_PASCAL (void, pt2_offset_proc) (paige_rec_ptr pg,
   co_ordinate_ptr point, short conversion_info, t_select_ptr selection);
```

point — is the coordinate from which to compute the offset.

conversion_info — indicates additional attributes to apply to the logic; this value can be either (or both) of the following bits:

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

If NO_HALFCHARS is set,— the offset must not shift to the character's right side unless it is completely to the right of the character. In other words, if a character were 10 pixels wide, the computed offset for NO_HALFCHARS must equal the left side of that character until the point was at least 10 pixels to its right.

If NO_BYTE_ALIGN is set, — possibly landing in the middle of a multibyte characters should be ignored. In other words, the text position should be computed as-is without any consideration to adjust or align for multibyte character boundaries.

selection — points to a *t_select* record which this function must completely initialize. The *t_select* record is defined as follows:

```
typedef struct
{
    longoffset;                                    /* Absolute text offset */
    select_pair     word_offsets;          /* Original word offsets if applicable */
    co_ordinate     original_pt;               /* Original point of selection */
    pg_short_t      line;                                  /* Point start number */
    short           flags;                          /* Contains internal attributes */
    long            control_offset;      /* Offset for purposes of tracking control */
    long            section_num;           /* Section ID (reserved for future) */
    long            primary_caret;         /* Relative primary direction caret */
    long            secondary_caret;              /* Relative secondary caret */
}
t_select;
```

Upon entry, none of the fields will be initialized. When this function returns, each field must contain the following:

offset —þThe absolute offset, in bytes, representing the point.

word_offsets —þThe function does NOT need to initialize this field.

original_pt —þShould be a copy of the *point* parameter.

line —þThe *point_start* element number within the *text_block* record that applies to offset. (Each *text_block* record contains an array of *point_start* records. The line field in *t_select* should be the element number for the appropriate *text_block*, the first line for each block being zero).

flags —þShould be set to zero.

control_offset —þShould be same as offset or, if tracking a “control” style, this should be set to whatever is appropriate for the control-tracking feature.

primary_caret —þThe pixel position relative to the *point_start*’s left bounds. In other words, caret should be the amount relative to the *point_start’s bounds.top_left.h* indicated above (line field).

secondary_caret —þThe pixel position relative to the *point_start's* left bounds for a "secondary" insertion point for mixed directional scripts. If one direction only, *secondary_caret* must be set to the same value as *primary_caret*.

font_init_proc

PURPOSE: To initialize a *font_info* record.

STANDARD FUNCTION: The default function for the **Windows** version. The font name is changed to a pascal string (if *info->environs* has *NAME_IS_CSTR* set). For the **Macintosh** version, *pgInitFont*, determines the font ID code, the script code (e.g., Roman, Kanji, etc.), the language and sets most other fields to zeros. It also converts the "name" field to a Pascal string if necessary. For **Windows-32**, the appropriate code page and language ID is determined and the name is adjusted to a pascal string, if necessary.

```
PG_FN_PASCAL (void, font_init_proc) (paige_rec_ptr pg, font_info_ptr
    info);
```

special_char_proc

PURPOSE: To draw "invisible" characters.

STANDARD FUNCTION: The default function, *pgSpecialCharProc*, draws the symbols as specified in *pg_globals* in the font specified in *pg_globals*.

```
PG_FN_PASCAL (void, special_char_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);
```

This function gets called after any text is drawn, but only if SHOW_INVIS_CHAR_BIT is set as an attribute in pg (see "Changing Attributes" on page 3-1 for information on *pgSetAttributes*).

walker — contains the current format info.

data — is a pointer to the text in the current text block and offset/length are the byte position and length of the text that has just been drawn.

draw_position, *extra* and *draw_mode* —parameters are the same parameters just given to *text_draw_proc*.

auto_scroll_proc

PURPOSE: To perform an automatic scroll during *pgDragSelect()* (mouse drag).

STANDARD FUNCTION: The default function, *pgAutoScrollProc*, performs automatic scrolling. If EXTERNAL_SCROLL_BIT is set in *pg*, only an internal adjustment is made (no visual scrolling occurs). If you need to autoscrolling in a different way than the default, use this hook to override it.

```
PG_FN_PASCAL (void, auto_scroll_proc) (paige_rec_ptr pg, short
    h_verb, short v_verb, co_ordinate_ptr mouse_point, short
    draw_mode);
```

This only gets called during *pgDragSelect(),* and then only if *auto_scroll == TRUE*. Upon entry, *h_verb* and *v_verb* indicate the direction to scroll (same possible values as given to *pgScroll* function). The *mouse_point* will be the current point given to *pgDragSelect()*.

draw_scroll_proc

PURPOSE: To draw additional items when updating a scroll region.

STANDARD FUNCTION: The default function, *pgDrawScrollProc*, does nothing. This low-level function has been provided for special features where the application needs to update something on the screen and, since OpenPaige can do autoscrolling, this provides a way to add "ornaments" to the scrolled area. The *draw_scroll_proc* now gets called twice, once before updating the scrolled area and once after updating the scrolled area.

```
PG_FN_PASCAL (void, draw_scroll_proc) (paige_rec_ptr pg, shape_ref
    update_rgn, co_ordinate_ptr scroll_pos, pg_boolean post_call);
```

This only gets called immediately after a physical scroll. The *update_rgn* contains the shape that requires updating (the "blank" part of the screen after a scroll). The *scroll_pos* parameter contains the current horizontal and vertical scrolled position (which always gets subtracted from drawing coordinates when updating the screen).

However, *draw_scroll_proc* gets called twice: after physical scrolling occurs and before any text is drawn inside the scrolled region, *draw_scroll_proc* is called and passes FALSE for *post_call*; then once all text is redrawn, *draw_scroll_proc* gets called again with *post_call* as TRUE.

NOTE: To further understand the relationship between scrolling, display and the scrolling "hooks" please see "scroll_adjust_proc" on page 27-552.

PURPOSE: To add additional graphics to the offscreen bitmap before stamping such bits to the screen.

STANDARD FUNCTION: The default function, *pgBitmapModifyProc*, does nothing. This low-level function has been provided for special features where the application needs to display something in the "background" such as a picture for which text overlays, which normally would get "erased".

```
PG_FN_PASCAL (void, bitmap_modify_proc) (paige_rec_ptr pg,
    graf_device_ptr bits_port, pg_boolean post_call, rectangle_ptr
    bits_rect, co_ordinate_ptr screen_offset, long text_position);
```

If OpenPaige does offscreen drawing, this function gets called twice: once after the bitmap is set up but before any text is drawn, and once after the text is drawn into the bitmap but before transferring to the screen.

bits_port — the offscreen bitmap port (see"Graphic Devices" on page 3-8). If *post_call* is TRUE, the function is getting called the second time (after the text is drawn into the bitmap but before transferring to the screen).

bits_rect — the target rectangle that the bits will eventually get copied to. In other words, the target rectangle is the bounding area on the screen for the text line being prepared for eventual bits transfer. The "local" rectangle for the bitmap area itself (whose top-left coordinate is typically 0, 0) is *bits_rect* offset by *screen_offset->h* and *screen_offset->v*.

text_position — parameter is the text position (relative to all text in *pg*) for the line about to be drawn or just drawn.

NOTE: This is the hook you use to do "backgrounding", i.e. to display some kind of graphic or pattern behind editable text.

The purpose of the *bitmap_modify_proc* is to alter the contents of OpenPaige's offscreen bitmap before it transfers those bits to the screen. Note that this only occurs

when OpenPaige is drawing in one of the "bits" draw modes: *bits_copy*, *bits_or*, *bits_xor*, etc. The bit transfer will always be targeted to the *graf_device* currently set in the *pg_ref*—there is no way to *change* what device will receive the bitmap other than assigning a different device to a *pg_ref* (using *pgSetDefaultDevice*) before calling any function that might draw text. (The exception to this is when the function has as one of its parameters an optional *graf_device* such as *pgPrintToPage*).

NOTE: The *bitmap_modify* function will also get called for the whole window if *bits_emulate_or*, *bits_emulate_xor* or *bits_emulate_copy* are indicated as the drawing mode. In this case, the "bitmap" is really the entire drawing area for the *pg_ref* on the screen. For maximum performance, you should use *"bits_emulate_xx"* modes for backgrounding something if the entire document is repainted on an erased window.

NOTE (Windows): The device context of the bitmap (or screen if *bits_emulate_xx* drawing mode) is *device->machine_ref*.

NOTE: bitmap transfer mode might still occur even if you did not explicitly pass one of the "bits" draw modes. If a function was called that suggested *best_way* for the drawing mode, OpenPaige often decides that bitmap transfer is *best_way* and assumes that mode.

wait_process_proc

PURPOSE: To inform the application when something that can take a bit of time is being performed.

STANDARD FUNCTION: The default function, *pgWaitProc*, does nothing. This low-level function has been provided so the application can display messages, put up "thermometers," etc. when something is going on that can take a while.

```
PG_FN_PASCAL (void, wait_process_proc) (paige_rec_ptr pg, short
    wait_verb, long progress_ctr, long completion_ctr);
```

wait_verb — defines what OpenPaige is doing and will be one of the following:

```
typedef enum {
    paginate_wait,                          /* Long pagination */
    copy_wait,                                  /* Long copy */
    insert_wait,                          /* Long paste (insert) */
    save_wait,                               /* Save file wait */
    open_wait                                /* Open file wait */
};
```

progress_counter — some number less than or equal to *completion_ctr*; however, the first time this function gets called, *progress_counter* will be zero; the final call (when the operation has completed) *progress_ctr* will equal *completion_ctr*.

A "percentage completion" can be calculated as:

```
(progress_ctr * 100) / completion_ctr
```

draw_page_proc

PURPOSE: To inform the application when the whole screen (or part of the screen) gets repainted. The purpose of this is to draw any special page or document ornaments such as page break or "margin" lines, page numbers, container outlines, headers and footers, etc.

STANDARD FUNCTION: The default function, *pgDrawPageProc*, does nothing. This low-level function has been provided so the application can draw "pagination" and other document items. See "Display Proc" on page 16-16.

```
PG_FN_PASCAL (void, draw_page_proc) (paige_rec_ptr pg, shape_ptr
    page_shape,pg_short_t r_qty, pg_short_t page_num, co_ordinate_ptr
    vis_offset, short draw_mode_used, short call_order);
```

OpenPaige calls this function only after a general display (from *pgDisplay*), after a scroll (*pgScroll* or any scrolling function), and printing (*pgPrintToPage*). This function is not called for any other display, including keyboard character insertions. The assumption is that page ornament items are "clipped" for regular typing, but require updating for general display.

Upon entry, *page_shape* is a pointer to the first rectangle of the page area in *pg*. The *r_qty* parameter will contain the number of rectangles within that shape (it will always be at least one).

page_num — indicates the "page" you are being asked to draw, the first page being zero. For a new *pg_ref* in which only the defaults are used, *page_num* will always be zero. If *pg* is set for repeating shapes (V_REPEAT_BIT or H_REPEAT_BIT set in *pg_doc_info*), *page_num* will indicate the page number (beginning at zero) you are asked to draw which can be a multiple repeat of the original shape. The *draw_page_proc* will get called for every "page" that is visible, one at a time.

vis_offset — will contain horizontal and vertical pixel amounts that should offset the rectangle(s) in *page_shape* to obtain the exact screen location for the "page" that is being drawn. In other words, if *page_shape* pointed to a single rectangle, the on-screen page dimensions are precisely **page_shape* offset horizontally by *vis_offset->h* and offset vertically by *vis_offset->v*.

For example, if an OpenPaige object is set for a repeating shape resulting in five "pages" on the screen (which is to say, the original shape repeats itself five times), *draw_page_proc* would be called five consecutive times, the first time passing zero for *page_num*, the second time a 1, then 2, 3, 4 and 5. Also, for each consecutive call, *vis_offset* would contain the appropriate pixel amount to adjust the rectangles in *page_shape* to draw each page at the correct screen location.

The *draw_mode_used* parameter indicates which drawing mode was used for text display before *draw_page_proc* was called (note that all text is drawn to the screen first, then *draw_page_proc*).

This function is not called for "pages" that fall completely out of the vis area in *pg*.

NOTE (Windows): The device context for drawing the page(s) is available as *pg->globals->current_port->machine_ref*.

CAUTION: Warning! The *page_ptr* is literally a pointer to the contents of *pg*'s *page_area* shape. Do not alter these rectangles!

CLIPPING

Upon entry, the "clip region" will be set to *pg's vis_area* boundaries. Normally, you should not need to change the clipping area.

PRINTING NOTE

The *draw_page_proc* also gets called for printing. In certain cases, you might not want to draw page ornaments (such as page break lines) while printing. To detect printing mode, check *pg->flags* as follows:

```
if (pg->flags & PRINT_MODE_BIT)
{
// is in print mode if PRINT_MODE_BIT is set.
}
```

See also, *pgSetDocInfo* in "Getting/Setting Document Info" on page 13-9.

text_break_proc

PURPOSE: To find the best place to split apart a block of text. OpenPaige does not hold a large continuous block of text, rather it breaks text up into smaller sections. The *text_break_proc* is used to determine where in a section of text is a good breaking point.

STANDARD FUNCTION: *pgTextBreakProc*: if <CR> characters exist in the text block, the closest CR to the middle of the block is returned as the best breaking point. If no CR's, the function recommends breaking on a line (word-wrap) boundary. If no lines (or one huge single line), the break occurs on a word boundary; if no words then *text_break_proc* has no other recourse than to break in the middle of text.

```
PG_FN_PASCAL (long, text_break_proc) (paige_rec_ptr pg,
    text_block_ptr block);
```

block — a pointer to the text block that must be split apart. The function result should be the relative offset to break the text (relative to the text in the block —þnot to all text in pg) .

scroll_adjust_proc

PURPOSE: To allow an application to adjust something before and after a document scrolls.

STANDARD FUNCTION: *pgScrollAdjustProc* does nothing.

```
PG_FN_PASCAL (void, scroll_adjust_proc) (paige_rec_ptr pg, long
    amount_h, long amount_v, short draw_mode);
```

Upon entry, *amount_h* and *amount_v* — are the amounts in pixels that will be scrolled horizontally and vertically, respectively. Negative amounts indicate the document contents will move upwards and/or left and positive amounts indicate the document contents will move down and/or to the right.

This function gets called twice, once before any physical scrolling occurs and once after scrolling occurs. You can detect the difference by the values in *amount_h* and *amount_v*: if *scroll_adjust_proc* is getting called after scrolling, both parameters will be zero.

draw_mode — indicates what the drawing mode will be. Note that it is possible for *draw_mode* t o be *draw_none*; it might be wise to observe that situation since it could make a difference in how you handle a scrolling adjustment.

SEQUENCE OF SCROLL HOOKS & DISPLAY

Scrolling hooks and display occurs in the following sequence:

- Application calls *pgScroll* (or any other function that causes a scroll).
- OpenPaige computes the amount to scroll, in pixels, and calls *adjust_scroll_proc* with those values.
- The window is scrolled.
- The *scroll_adjust_proc* is called again with 0,0 for "scroll amounts."

If *draw_mode* is not *draw_none*:

- The *draw_scroll_proc* is called with FALSE for *post_call*.
- Screen is refreshed (for scrolled area).
- The *draw_scroll_proc* is called once more with TRUE for *post_call*.

enhance_undo_proc

PURPOSE: To allow for custom "undo" and/or to modify an existing undo record prepared for undo.

STANDARD FUNCTION: *pgEnhanceUndo* does nothing.

```
PG_FN_PASCAL (void, enhance_undo_proc) (paige_rec_ptr pg,
    pg_undo_ptr undo_rec, void PG_FAR *insert_ref, short
    action_to_take);
```

This function gets called from either *pgPrepareUndo* or *pgUndo*; the difference can be determined by the *action_to_take* verb which will be one of the following:

```
typedef enum
{
    enhance_prepared_undo,/* undo_rec is from pgPrepareUndo */
    enhance_performed_undo  /* undo_rec is from pgUndo */
};
```

insert_ref — will be whatever was given to the same parameter for *pgPrepareUndo* (or NULL if *enhance_undo_proc* is being called from *pgUndo*).

undo_rec — a pointer to an OpenPaige undo record as follows:

```
typedef struct
{
    short                verb;                    /* Type of undo (for app's reference) */
    short                real_verb;                        /* Internal action verb */
    pg_ref               data;                   /* Data (different for each verb) */
    pg_globals_ptr       globals;                    /* Pointer to OpenPaige globals */
    memory_ref           keyboard_ref;            /* Used for backspace-key-undo */
    format_ref           keyboard_styles;         /* Styles possibly backspaced */
    format_ref           keyboard_pars;       /* Paragraphs possibly backspaced */
    memory_ref           applied_range;                    /* Range to apply Undo */
    memory_ref           shape_data;                       /* Data for shape undo */
    memory_ref       refcon_data;          * Used to copy refcons for containers */
    memory_ref           doc_data;                         /* Used for undo doc info */
    memory_ref       rsrv;               /* Reserved for DSI extensions*/ select_pair     alt_range;
    /* Range for Undo Paste & other things */    select_pair       keyboard_delete;
    /*Delete range for backspace undo*/      long    ref_con;                        /* App can set
    this */
}
pg_undo;
```

The fields of interest (for custom undo) are as follows:

verb — Holds the original undo verb as given to *pgPrepareUndo*. However, if verb is negative then the record is intended for a "redo." Example: —*undo_paste* is "redo paste."

globals — A pointer to OpenPaige globals.

applied_range — If non-NULL this is a *memory_ref* containing *select_pair's* defining the selection range for which the undo/redo applies.

alt_range — Contains the range of text for which this undo/redo applies (if selection is simply two offsets). If more complex selection, the offsets will be in *applied_range*.

ref_con — Can be used by your application for anything.

The precise calling sequence of *enhance_undo_proc* in relationship to *pgPrepareUndo* and *pgUndo* is as follows:

- When *pgPrepareUndo* is called, the undo record is prepared with everything OpenPaige "knows" about. Then before returning from *pgPrepareUndo*, *enhance_undo_proc* is called passing the *pg_undo_ptr* (just prepared) and the *action_to_take* will be *enhanced_prepared_undo*.
- When *pgUndo* is called, everything is "undone" that OpenPaige knows about. Then before updating anything on the screen, *enhance_undo_proc* is called, passing the undo record and *action_to_take* is *enhanced_performed_undo*.

NOTE: Both *pgPrepareUndo* and *pgUndo* will only handle the verbs that it recognizes. It means that a completely foreign *undo* verb causes OpenPaige to do nothing at all —þbut it still calls *enhance_undo_proc*, passing your application an empty *pg_undo* record the "verb" field set to whatever you gave). Hence, a completely custom prepare undo/redo is possible by inventing undo verbs that OpenPaige doesn't understand.

For example, if you called *pgPrepareUndo(pg, 9000, ptr)*, OpenPaige will have no idea what "9000" is —þbut it will create an empty undo record, place "9000" in the verb field and call *enhance_undo_proc(pg, &undo_rec, ptr, enhance_prepared_undo*).

Of course you can also use *enhance_undo_proc* to simply modify existing undo operations. For example, additional information can be placed in the *ref_con* field while you let OpenPaige handle everything else.

CAUTION: While it is perfectly OK for you to set up a completely custom *undo_ref*, do not call *pgDisposeUndo* if you have set any of the fields besides the verb and *ref_con*, even if you have set an unrecognized undo verb. Instead, dispose the structure(s) yourself.

NOTE: There is no verb for *"dispose undo"* since there is not necessarily an associated *pg_ref* when an *undo_ref* is disposed (thus there is no function pointer!). Therefore you must dispose your own data structures, if any, before calling *pgDisposeUndo*.

par_boundary_proc

PURPOSE: To quickly locate the text offsets of a paragraph.

STANDARD FUNCTION: *pgParBoundaryProc* locates the beginning and ending offsets that enclose a paragraph of text.

```
PG_FN_PASCAL (void, par_boundary_proc) (paige_rec_ptr pg,
    select_pair_ptr boundary);
```

This function gets called when OpenPaige wants to know the offsets of a paragraph. Upon entry, *boundary->begin* contains the text location in question; this function should initialize *boundary->begin* and *boundary->end* to the beginning and ending offsets of the paragraph.

change_container_proc

PURPOSE: To change, modify or enhance a "container" before erasing it, displaying text, calculating text or clipping its bounding region.

STANDARD FUNCTION: *pgModifyContainerProc* erases the container if the *verb* parameter so designates, otherwise does nothing.

```
PG_FN_PASCAL (void, change_container_proc) (paige_rec_ptr pg,
    pg_short_t container_num, rectangle_ptr container, pg_scale_ptr
    scaling, co_ordinate_ptr screen_extra, short verb, void PG_FAR
    *misc_info);
```

OpenPaige calls this function to give the application a chance to modify something to achieve desired affects for different text "containers" or areas in the page shape.

For example, OpenPaige always calls *change_container_proc* to "erase" a container. If your application wanted to provide different background colors for different containers, it could use this function to set and erase the appropriate backgrounds.

Upon entry, *container_num* is the rectangle number of the "container" and container is the pointer to the rectangle. The container number is one-based, i.e. the first rectangle is 1. The rectangle itself is unscaled and not scrolled.

scaling — indicates any scaling (or can be NULL).

screen_extra — contains the amount of offset that would be required to obtain the actual rectangle on the screen. In other words, container offset by *screen_extra->h* and *screen_extra->v* is the actual container position in the scrolled document.

verb — indicates why the function is being called, which will be one of the following:

```
typedef enum
{
    clip_container_verb,                    /* clip container if desired */
    unclip_container_verb,    /* restore clip region (that changed
                                    from above) */
    erase_rect_verb,                            /* erase container */
    will_draw_verb,             /* about to draw text in container */
    will_delete_verb                /* about to delete container */
};
```

What gets passed in **misc_info* depends on the value of verb, as follows:

If *verb* is *clip_container_verb*, *misc_info* is a pointer to a long which is initially set to zero.

If *verb* is *unclip_container*, *misc_info* points to the same long as in *clip_container_verb*. The purpose of this is to let you set **misc_info* to something (such as the previous clip region) so you can use it when you unclip the area.

If *verb* is *erase_rect_verb*, *will_draw_verb* or *will_delete_verb*, *misc_info* is NULL.

NOTE: The term "containers" in this description really means a rectangle inside the page area. Complex, irregular shapes can therefore have hundreds of "containers" even though your application might think of the page shape as one object. Therefore do not confuse the logical "container" as a single unit shape or page with the OpenPaige meaning of a rectangle within a shape.

smart_quotes_proc

PURPOSE: To implement "smart quotes" for text insertions.

STANDARD FUNCTION: If the next insertion is a "flat" single or double quote character, the insertion is changed by *pgSmartQuotesProc* to the appropriate left or right quote characters per definitions in *pg_globals*.

```
PG_FN_PASCAL (void, smart_quotes_proc) (paige_rec_ptr pg, long
    insert_offset, long info_bits, pg_char_ptr char_to_insert, short
    PG_FAR *insert_length);
```

This function gets called if the byte about to be inserted into a *pg_ref* is a "quote character".

NOTE: This is determined purely by the *char_info* hook for the insertion style returning such information about the character.

Upon entry, *insert_offset* is the *byte_offset* in text that will be inserted (relative to all text). *char_to_insert* points to the first byte to be inserted (whose first character will be at least one of the quote characters as defined in the OpenPaige globals). The *info_bits* parameters contains the results of *char_info*, indicating which quote character will be inserted.

Upon entry, **insert_length* contains the number of bytes of the character to be inserted. To change the character to, say, an appropriate smart quote, this function should physically change **char_to_insert* and set **insert_length* to the new length if it is different.

NOTE: **char_to_inser*t will always be big enough to hold up to four bytes.

If it is decided that the character should change, *smart_quotes_proc* should literally alter **char_to_change*.

line_init

```
PG_FN_PASCAL(void, line_init_proc) (paige_rec_ptr pg,
    pg_measure_ptr measure_info, short init_verb);
```

This function is called once before building lines of text, once before building an individual line and once when all lines have been built.

PURPOSE: Its intended purpose is to let an application set up whatever it needs to handle subsequent calls to other hooks (such as *line_parse_proc* below).

STANDARD FUNCTION: The default proc is *pgInitLineProc* which does nothing.

Upon entry, the information about the line to be calculated is contained in *measure_info* (see *pg_measure_ptr* in "line_validate_proc" on page 27-534).

The *verb* parameter will be one of the following:

```
enum
 {
   init_measure_verb,        // Called before any lines are computed
   new_line_verb,                 // Called before a line is computed
   done_measure_verb              // Called when all lines have been
   computed
};
```

line_parse

```
PG_FN_PASCAL(long, line_parse_proc) (paige_rec_ptr pg,
   pg_measure_ptr measure_info, pg_char_ptr text, point_start_ptr
   line_start, long global_offset, long remaining_length);
```

This hook gets called repetitively when a line of text in *pg* is being built. Its intended purpose is to force a particular *point_start* record (many of which can compose a single line) to assume a certain length.

STANDARD FUNCTION: The default function is *pgLineParse* which does nothing.

One example of this would be building an array of "cells", in which an application needed to display a matrix of rows and columns for every "number" that appeared in a plain line of text. Normally, the line would be formatted as a single *point_start* record, yet a row/column display feature would require the line to break apart into one *point_start* for each "cell".

When this function is called, OpenPaige is essentially asking where the end of the current *point_start* record should break. The *measure_info* parameter is a pointer to the current position and *point_start(s)* of the line (see *pg_measure_ptr* "line_validate_proc" on page 27-534). The *line_start* parameter is the current *point_start* record (the one being asked about). The *text* parameter is the text to be included in the current *point_start* and *remaining_length* its maximum length.

What this function must do is return the new remaining length that the line builder should work with.

For example, in the cell/row application, the function would examine the text, and notice that the next "number" is 6 bytes in length. In this case, this function would return a "6" and the *point_start* for this portion of the line would be limited to 6 bytes.

The function would then be called again, after the 6-byte *point_start* had been formatted (if there is any more text available and the maximum line width permits it).

paginate_proc

```
PG_FN_PASCAL(void, post_paginate_proc) (paige_rec_ptr pg,
    text_block_ptr block, smart_update_ptr update_info, long
    lineshift_begin, long lineshift_end,
    short action_taken_verb);
```

This function is called every time after OpenPaige has paginated a block of text. By *paginate* is meant the computations of lines, their text widths and vertical/horizontal positions on the page.

STANDARD FUNCTION: The default proc, *pgPostPaginateProc,* implements widow and orphan control and "keep-paragraphs-together" if the document is set for repeating pages.

Upon entry, block is the *text_block* just paginated.

The *update_info* parameter is provided in case this function needs to change the recommended beginning of display. If so, *update_info->suggest_begin* and/or *update_info->suggest_end* can be altered

NOTE: .*update_info* might be null.

The *lineshift_begin* and *lineshift_end* values indicate a range of text that has moved vertically by virtue of pagination. These two values are text-line based (operate on line boundaries) and provide valuable information for performance purposes. For example, if

the user inserted a character causing no new wordwrapping, *lineshift_begin* will equal *lineshift_end*, which means none of the lines in the document have shifted vertically.

The *action_taken_verb* indicates what has occurred, which will be one of the following:

```
enum
{
    paginated_line_shift,                /* Only shifted line locations vertically */
    paginated_empty_block,                                  /* Built an empty block */
    paginated_hidden_block,           /* Built block whose text is all invisible. */
    paginated_fake_block,  /* Built dummy block -- sits below last container */
    paginated_partial_block,                        /* Rebuilt lines only partially */
    paginated_full_block                                      /* Rebuilt everything */
};
```

click_proc

```
PG_FN_PASCAL(void, click_examine_proc) (paige_rec_ptr pg, short
    click_verb, short modifiers, long refcon_return, t_select_ptr
    begin_select, select_ptr end_select);
```

This function gets called after *pgDragSelect* has processed a "click" but before it returns control back to the application.

STANDARD FUNCTION: The default function, *pgExamineClickProc*, does nothing.

Upon entry, *click_verb* is the verb given to *pgDragSelect* (*mouse_down*, *mouse_moved* or *mouse_up*); modifiers is the same value given in *pgDragSelect* for modifiers. (See "Clicking & Dragging" on page 2-43). The *refcon_return* field is the value that *pgDragSelect* is about to return.

The *begin_select* and *end_select* parameters indicate the beginning and ending points of the recent selection

NOTE: Until *mouse_up* has occurred, these points can be "backwards," i.e. *end_select* can reflect an *offset < begin_select* if the user clicked and dragged backwards.

text_increment

```
PG_FN_PASCAL (void, increment_text_proc) (paige_rec_ptr pg, long
    base_offset, long increment_amount);
```

This function is called for every occurrence of inserting or deleting text. The intended purpose is to let OpenPaige extensions handle external "runs" when it becomes necessary to make adjustments for text insertions and deletions.

This function is always called after (not before) an insertion or deletion.

STANDARD FUNCTION: The default proc, *pgTextIncrementProc*, does nothing.

Upon entry, if the function is being called due to an insertion, *base_offset* is the text offset where the insertion occurred and *increment_amount* is positive (indicating the number of bytes inserted).

If the function is being called after a deletion, *base_offset* is the text offset before the first byte that got deleted and *base_offset* is negative (the absolute value is the number of bytes that got deleted).

NOTE: The function is called after the deletion, so the text that got deleted will no longer exist.

```
PG_FN_PASCAL(void, set_device_proc) (paige_rec_ptr pg, short verb,
    graf_device_ptr device,
    color_value_ptr bk_color);
```

This function is called to prepare a device for drawing, clipping, and font selection(s), etc., or to report that the device is to be released. The *verb* parameter indicates whether to prepare the device or release the device.

PURPOSE: The purpose is to "prepare" or "release" a machine-specific graphics device for general use. Your application might use this hook, for instance, to prepare a special device or device characteristics. *Calls to this function can be nested*, so this function is responsible for handling multiple "prepare" and "release" situations.

STANDARD FUNCTION: If the *verb* indicates device preparation, the **Windows** version creates a device context (HDC), stores it into the *machine_ref* member of device and does a *SaveDC()*; the **Macintosh** version saves the current *GrafPort* then does a *SetPort()* with the *GrafPtr* in the *machine_var* field of device and saves all the *GrafPort* settings. If the verb indicates device release, the **Windows** version does a *RestoreDC()* and/or a *DeleteDC()* if appropriate; the **Macintosh** version restores the original settings of the port, then sets the previous *GrafPort*.

Upon entry, the *verb* parameter will indicate one of the following:

```
set_pg_device,                  /* Prepare the device */
unset_pg_device          /* Release the device */
```

The *device* parameter points to the *graf_device* structure to prepare (this function alters that structure as necessary).

If *bk_color* is nonnull, the background color should be set to that value.

NESTED CALLS

OpenPaige will make frequent calls to this function, both *set_pg_device* and *unset_pg_device*, often nesting these pairs several levels deep. The standard *set_device_proc* handles this by incrementing/decrementing a counter in the *graf_device* structure, and handling the device accordingly.

For example, when *verb == pg_set_device* the **Windows** version creates a new device context only if the counter is zero, otherwise it simply uses the previous DC (which is stored in the *graf_device*); then it increments the counter. Similarly, when *verb == pg_unset_device*, the counter is decremented, and only when it decrements to zero does the DC get deleted.

page_modify_proc

```
PG_FN_PASCAL(void, page_modify_proc) (paige_rec_ptr pg, long
    page_num, rectangle_ptr margins);
```

This function is called for each repeating "page" during the text pagination process.

PURPOSE: Its intended purpose is to allow temporary modification(s) to any of the four sides of the page. A typical reason for doing this would be to add space for headers and footers, or extra "gutter" space based on document format and page number, etc.

STANDARD FUNCTION: The default function does nothing.

NOTE: This hook will not get called unless the *pg_ref* is set for a "repeating page shape". See "Repeating Shapes" on page 13-9.

Upon entry, *page_num* will indicate the page number of the "page" about to be paginated; the first page is 1 (not 0). At this time, no text lines will have been calculated for this page, so any modifications made to the page boundary will affect the placement of text.

To "modify" the page boundary, this function should set margins to the desired amount of inset. For example, if *margins->top_left.v* were set to 20, the text would paginate on

this page beginning 20 pixels from the top; if *margins->bot_right.h* were set to 50, text would wrap 50 pixels from the right side of the page, etc.

While text is being paginated, OpenPaige calls this function before it calculates the first line on a page. Each time this function is called, all four "margin" sides will be set to zero; hence if this function merely returns (does nothing), the page dimensions remain unchanged (they will retain their original dimensions as if you were not using this hook at all).

NOTE: The *page_modify_proc* also gets called when OpenPaige is computing the clipping region.

wordbreak_info_proc

```
PG_FN_PASCAL (long, wordbreak_info_proc) (paige_rec_ptr pg,
    pg_char_ptr the_char, short charsize,
    style_info_ptr style, font_info_ptr font,
    long current_settings);
```

This hook has been provided to allow special-case word breaking for various character sets, or multilingual scripts.

PURPOSE: OpenPaige calls this hook in response to its internal *char_info* function to learn about the nature of a particular character.

For example, in Japanese most characters can be considered "words" for purposes of breaking on a line, but there are several exceptions —þcertain characters must never end on a line and must stay grouped with character(s) that fall after them. Similarly there are characters that must never begin on a line and must stay grouped with character(s) before them. The purpose of this callback function is to determine of the character in question matches the application-defined table of these exceptions.

STANDARD FUNCTION: The default function returns *current_settings*.

Upon entry, *the_char* is a pointer to a character and charsize is the byte size of that character. The current style information is provided in the *style* parameter and the font parameter provides the current font information.

NOTE (Windows): þthe current code page for the character in question is available in the font parameter as *font->code_page*; the language ID (in LCID format) is available in the "language" member of font.

NOTE (Macintosh): þthe script code is in *font->char_type*.

The *current_settings* parameter contains a combination of bit settings that define what kind of character OpenPaige already thinks is appropriate, which can be a combination of any of the bit settings for the *char_info_proc* function.

FUNCTION RESULT: The new character bit settings should be returned. If no change to the existing settings are required, return *current_settings*.

key_insert_query

```
PG_FN_PASCAL(short, key_insert_query) (paige_rec_ptr pg,
    pg_char_ptr the_char, short charsize);
```

This function is called when *pgInsert()* is called but before anything is inserted into *pg*.

PURPOSE: Its intended purpose is to speed up keyboard entry by determining if a character should be inserted immediately and redrawn, or temporarily buffered and inserted later.

STANDARD FUNCTION: The default function for **Macintosh** returns *key_insert_mode* (signifying the character should be inserted now). The Windows version checks for a pending *WM_CHAR* message and, if any, returns *key_buffer_mode* (so the character is buffered).

Upon entry, *the_char* is the character to be inserted and charsize the number of bytes for the character.

FUNCTION RESULT: If the character should be inserted immediately, return *key_insert_mode*, otherwise return *key_buffer_mode*.

```
PG_FN_PASCAL(pg_word, charclass_info_proc) (paige_rec_ptr pg,
    pg_char_ptr the_char, short charsize,
    style_info_ptr style, font_info_ptr font);
```

This function is called to determine the possible character subset or multilingual "class" of character.

PURPOSE: Its intended purpose is to determine a language or scripting break in the text for purposes of selection or wordwrapping. For all-Roman text, this functionality is not required (since OpenPaige handles text selection and word-breaking automatically). For special scripts, character classes can become more complex and demand further attention.

STANDARD FUNCTION: The default function uses OS-specific functions to determine the character type of *the_char*. For Japanese, the function will determine which subset of the current script *the_char* belongs to.

Upon entry, *the_char* will be a single or double byte character, charsize its byte size. The *style* and *font* parameters provide the current *style_info* and *font_info* for the character.

FUNCTION RESULT: The character class type should be returned. This can be any value so long as each character of the same class return the same response. (OpenPaige simply compares the responses to each other, and if one of them is different, that is considered a change in script type. This is used to select (highlight) "words" from double-clicks, etc.

NOTE (Windows): þthe current code page for the character in question is available in the font parameter as *font->code_page*; the language ID (in LCID format) is available in the "language" member of font.

NOTE (Macintosh): þthe script code is in *font->char_type*.

27.9 Warning About "Re-Entrant" Functions

OpenPaige functions are generally reentrant and can therefore be called when you execute a custom hook. However, as a general rule you should never call anything that tries to change the internal structure(s) of a *pg_ref*. While the code itself is reentrant, data structures might be LOCKED and therefore cannot be resized. A function such as *pgSetStyleInfo*, for instance, often needs to add a *style_info* record; if the array of styles is temporarily locked, a call to *pgSetStyleInfo* could crash!

You should therefore restrict data changes to the parameters given to you in the low-level function itself or, if you absolutely must change something (and know what you are doing), change the structure(s) directly without resizing memory or adding/deleting records.

NOTE: All the "Get" functions, however (*pgGetStyleInfo*, *pgGetParInfo*, etc.) are always safe to call.

One exception: It is OK (often anticipated) to call *pgSetExtraStruct* while executing a low-level hook.

27.10 Text Parsing & Word Breaks

A new callback "hook" has been provided to allow special-case word breaking for various character sets (code pages). OpenPaige calls this hook in response to its internal *char_info* function to learn about the nature of a particular character.

For example, in Japanese most characters can be considered "words" for purposes of breaking on a line, but there are several exceptions — certain characters must never end on a line and must stay grouped with character(s) that fall after them. Similarly there are characters that must never begin on a line and must stay grouped with character(s) before them. The purpose of this callback function is to determine of the character in question matches the application-defined table of these exceptions.

The prototype definition of this callback is as follows:

```
PG_PASCAL (long) pgBreakInfoProc (paige_rec_ptr pg, pg_char_ptr
    the_char,
    short charsize, style_info_ptr style, font_info_ptr font,
    long current_settings);
```

When OpenPaige is determining where to break a line (word-wrap), it calls this function to determine any special-case information that might influence the word breaking result. The default function (the one that exists if your application does not set its own) does not do anything other than use the defaults, which in the case of double-byte characters it assumes that all such characters are valid word breaks.

Upon entry, *the_char* is a pointer to (usually) a double-byte character and charsize is the byte size of that character. The current style information is provided in the style parameter and the font parameter provides the current font information.

NOTE: The current code page for the character in question is available in the *font* parameter as f*ont->code_page*; the language ID (in LCID format) is available in the "language" member of font.

The *current_settings* parameter contains a combination of bit settings that define what kind of character OpenPaige already thinks is appropriate, which can be a combination of any of the following:

```
#define BLANK_BIT           0x00000001      // Character is blank
#define WORD_BREAK_BIT      0x00000002      // Word breaking char
#define WORD_SEL_BIT        0x00000004      // Word select char
#define SOFT_HYPHEN_BIT     0x00000008      // Soft hyphen char
#define INCLUDE_BREAK_BIT   0x00000010      // Include with word break
#define INCLUDE_SEL_BIT     0x00000020      // Include with word select
#define CTL_BIT             0x00000040      // Char is a control code
#define INVIS_ACTION_BIT    0x00000080      // Char is an arrow, etc.
#define PAR_SEL_BIT         0x00000100      // Char breaks a paragraph
#define LINE_SEL_BIT        0x00000200      // Char breaks a line (soft CR)
#define TAB_BIT             0x00000400      // Char is a TAB
#define FIRST_HALF_BIT      0x00000800      // 1st half of a multi-byte char
#define LAST_HALF_BIT       0x00001000      // Last half of a multi-byte char
#define MIDDLE_CHAR_BIT     0x00002000      // Middle of a multi-byte char
```

```
#define CONTAINER_BRK_BIT       0x00004000      // Break-container bit
#define PAGE_BRK_BIT            0x00008000      // Break-repeating-shape bit
#define NON_BREAKAFTER_BIT      0x00010000      // Must stay with char(s) after
#define NON_BREAKBEFORE_BIT     0x00020000      // Must stay with char(s) before
#define NUMBER_BIT              0x00040000      // Char is numeric
#define DECIMAL_CHAR_BIT        0x00080000      // Char is decimal
#define UPPER_CASE_BIT          0x00100000      // Char is UPPER CASE
#define LOWER_CASE_BIT          0x00200000      // Char is lower case
#define SYMBOL_BIT              0x00400000      // Char is a symbol
#define EUROPEAN_BIT            0x00800000      //Char is ASCII-European
#define NON_ROMAN_BIT           0x01000000      //Char is not Roman script
#define NON_TEXT_BIT            0x02000000      //Char is not really text
#define FLAT_QUOTE_BIT          0x04000000      // Char is a "flat" quote
#define SINGLE_QUOTE_BIT        0x08000000      //Quote char is single quote
#define LEFT_QUOTE_BIT          0x10000000      // Char is a left quote
#define RIGHT_QUOTE_BIT         0x20000000      // Char is a right quote
#define PUNCT_NORMAL_BIT        0x40000000      // Char is normal punctuation
#define OTHER_PUNCT_BIT         0x80000000      // Char is other punctuation
```

For purposes of the *pgBreakInfoProc*, the following bits are usually the only settings you will care about:

```
WORD_BREAK_BIT

INCLUDE_BREAK_BIT

NON_BREAKAFTER_BIT

NON_BREAKBEFORE_BIT
```

If WORD_BREAK_BIT is set that character delineates a word for wrapping (breaking) purposes; if INCLUDE_BREAK_BIT is also set, the character is part of the word.

For example, in Roman text a space character would have WORD_BREAK_BIT set but not INCLUDE_BREAK_BIT (because the space is not part of a word). However, a "," (comma) character would have both WORD_BREAK_BIT and INCLUDE_BREAK_BIT set because it is a word break but must be included with the word.

NON_BREAKAFTER_BIT causes the character to always stay with the character(s) immediately after its position in text; NON_BREAKBEFORE causes the character to always stay with the character(s) immediately before its position in text.

FUNCTION RESULT: The callback function must return the new settings that OpenPaige should use for the character. Often, this return value will simply be whatever is in *current_settings*.

EXAMPLE:

```
PG_PASCAL (long) MyBreakInfoProc (paige_rec_ptr pg, pg_char_ptr
    the_char,
    short charsize, style_info_ptr style, font_info_ptr font,
    long current_settings)
{
    if (SettingsOK(the_char, charsize))
                        return      current_settings;
}
```

Examples for changing the settings to something else are shown below.

SETTING THE HOOK

The callback ("hook") function is embedded in the *pg_ref* itself. If you want the same callback function to always get called from every *pg_ref* you should establish the function pointer into OpenPaige globals early before any *pg_refs* are created.

EXAMPLE:

```
pg_globalspgGlobals;   // OpenPaige globals record

void InitializePAIGE (void)
{
   pgInit(&paige_globals, &mem_globals);
   mem_globals.def_hooks.wordbreak_proc = MyBreakInfoProc;
}
```

In the example above, *"MyBreakInfoProc"* is the name of your callback function (note, for 16-bit Windows you would need to call *MakeProcInstance()* to create a valid function pointer). By placing it in the globals structure, OpenPaige will initialize the wordbreaking callback for all *pg_refs*.

Examples of Parsing Multilingual Word Breaks

A typical multilingual word parsing feature would be to check the code page in the word breaking callback, then compare the character to a predefined table for that code page to determine if the character has any additional word breaking attributes.

In Japanese, for example, most characters can break on a line as a "word." In some cases, however, the character can break but must be included with the "word" immediately to the left or to the right. The following is an example of tagging these exceptions in *"MyBreakInfoProc"* (your application-defined word breaking callback):

```
#define CP_JAPANESE932// Japanese code page
#define CP_US   1252                  // United states and Latin

PG_PASCAL (long) MyBreakInfoProc (paige_rec_ptr pg, pg_char_ptr
   the_char,
   short charsize, style_info_ptr style, font_info_ptr font,
   long current_settings)
```

```
                        // If not a double byte, trust the defaults,
                        // otherwise let's present a lookup table:

                                keep_with_left_table = jp_resource1;
                                keep_with_left_table = jp_resource2;
                                }

                                break;

                        case CP_US:
                                break;  // Just trust the defaults
    }

// Check if we even care (if we have a table for comparison):

    if (keep_with_left_table != (LPSTR)NULL)
    {
      // If character found in the "keep-with-left-table"
      // then we must force it to stay with word on the left:

    if (FindCharInTable(keep_with_left_table, the_char, charsize))
                                result |= NON_BREAKBEFORE_BIT;
                                else
                                result &= (~NON_BREAKBEFORE_BIT);
    }
if (keep_with_right_table != (LPSTR)NULL)
    {
      // If character found in the "keep-with-right-table"
      // then we must force it to stay with word on the right:
    if (FindCharInTable(keep_with_right_table,
    the_char, charsize))
    result |= NON_BREAKAFTER_BIT;
                        else
                        result &= (~NON_BREAKAFTER_BIT);
    }
    return          result;
}
```

In the above example, the “table” variables can be simple resources that contain a list of characters. Usually you only care about double-byte characters (because OpenPaige handles most single byte characters correctly using its defaults — but that might change with certain languages). The *“FindCharInTable”* function would simply be a function that returns TRUE if the character in question was in the table.

27.11 Character/Language Subsets

Certain languages can have “subsets” of characters that belong together as a group — at least for purposes of double-clicking for a word selection. In Japanese script, for instance, each double-byte character can be Katakana, Hiragana or Ideograph.

Although OpenPaige handles these subsets appropriately for word-selection purposes, should it become necessary to alter the subset differences or add new ones, the following callback hook is available:

```
PG_PASCAL (pg_word) pgCharClassProc (paige_rec_ptr pg,
   pg_char_ptr the_char,
   short charsize, style_info_ptr style, font_info_ptr font);
```

OpenPaige calls this hook in addition (and after) the word break callback. The purpose of the function is to return the character class — or “subset” of the language. Upon entry, *the_char* points to the first byte of the character and charsize is the size of the character, in bytes. The style and font parameters are the current style and font of the character. Note that *font->code_page* and f*ont->language* will contain the character's code page and LCID, respectively.

The following is the default code that OpenPaige-**Windows** executes for this function (the function that gets called if you do not set your own):

```
PG_PASCAL (pg_word) pgCharClassProc (paige_rec_ptr pg,
    pg_char_ptr the_char,
    short charsize, style_info_ptr style, font_info_ptr font);
{
    WORD          types[4];

    GetStringTypeEx((LCID)font->language, CT_CTYPE3,
    (LPCSTR)the_char, (int)charsize, types);

    return        (WORD)(types[0] & (C3_KATAKANA | C3_HIRAGANA
    | C3_IDEOGRAPH | C3_HALFWIDTH | C3_FULLWIDTH));
}
```

NOTE: The default code uses *GetStringTypeEx* to determine the language subset of the character. OpenPaige does not actually examine the value or contents of the function result; rather, this function is called for adjacent characters to determine if a word selection (highlight) should continue.

For example, if the first five characters in a string returned 0x0000 from *pgCharClassProc*, then the sixth character returned 0x0001, OpenPaige would only highlight the word(s) within the first five characters if the user double-clicked.