- -

This file, drivers.doc, describes the interface between Ghostscript and device drivers.

For an overview of Ghostscript and a list of the documentation files, see README.

****
******** Adding a driver ********
****

To add a driver to Ghostscript, all you need to do is edit devs.mak in two places. The first is the list of devices, in the section headed

```
# ------------------------------- Catalog ------------------------------- #
```

Pick a name for your device, say smurf, and add smurf to the list. (Device names must be 1 to 8 characters, consisting of only letters, digits, and underscores, of which the first character must be a letter. Case is significant: all current device names are lower case.)
The second is the section headed

```
# ---------------------------- Device drivers --------------------------- #
```

Suppose the files containing the smurf driver are called joe and fred. Then you should add the following lines:

```
# ------ The SMURF device ------ #

smurf_=joe.$(OBJ) fred.$(OBJ)
smurf.dev: $(smurf_)
	$(SHP)gssetdev smurf $(smurf_)

joe.$(OBJ): joe.c ...and whatever it depends on

fred.$(OBJ): fred.c ...and whatever it depends on
```

If the smurf driver also needs special libraries, e.g., a library named gorf, then the gssetdev line should look like

```
	$(SHP)gssetdev smurf $(smurf_)
	$(SHP)gsaddmod smurf -lib gorf
```

****
******** Keeping things simple
****

If you want to add a simple device (specifically, a black-and-white printer), you probably don't need to read the rest of this document; just use the code in an existing driver as a guide. The Epson and BubbleJet drivers (gdevepsn.c and gdevbj10.c) are good models for dot-matrix printers, which require presenting the data for many scan lines at once; the DeskJet/LaserJet drivers (gdevdjet.c) are good models for laser printers, which take a single scan line at a time but support data compression. For color printers, the DeskJet 500 C driver (gdevcdj.c) is a good place to start.

On the other hand, if you're writing a driver for some more esoteric device, or want to do something like add new settable attributes (besides page size and resolution), you probably do need at least some of the information in the rest of this document. It might be a good idea for you to read it in conjunction with one of the existing drivers.

****
******** Driver structure ********
****

A device is represented by a structure divided into three parts:

- procedures that are shared by all instances of each device;

- parameters that are present in all devices but may be different for each device or instance; and

- device-specific parameters that may be different for each instance.

Normally, the procedure structure is defined and initialized at compile time. A prototype of the parameter structure (including both generic and device-specific parameters) is defined and initialized at compile time, but is copied and filled in when an instance of the device is created.

The gx_device_common macro defines the common structure elements, with the intent that devices define and export a structure along the following lines:

```
	typedef struct smurf_device_s {
		gx_device_common;
		... device-specific parameters ...
	} smurf_device;
	smurf_device gs_smurf_device = {
		sizeof(smurf_device),		* params_size
		{ ... procedures ... },		* procs
		... generic parameter values ...
		... device-specific parameter values ...
	};
```

The device structure instance *must* have the name gs_smurf_device, where smurf is the device name used in devs.mak.

All the device procedures are called with the device as the first argument. Since each device type is actually a different structure type, the device procedures must be declared as taking a gx_device * as their first argument, and must cast it to smurf_device * internally. For example, in the code for the "memory" device, the first argument to all routines is called dev, but the routines actually use md to reference elements of the full structure, by virtue of the definition

```
#define md ((gx_device_memory *)dev)
```

(This is a cheap version of "object-oriented" programming: in C++, for example, the cast would be unnecessary, and in fact the procedure table would be constructed by the compiler.)

Structure definition

This essentially duplicates the structure definition in gxdevice.h.

```
typedef struct gx_device_s {
	int params_size;		/* size of this structure */
	gx_device_procs *procs;		/* pointer to procedure structure */
	char *name;			/* the device name */
	int width;			/* width in pixels */
	int height;			/* height in pixels */
	float x_pixels_per_inch;	/* x density */
	float y_pixels_per_inch;	/* y density */
	gs_rect margin_inches;		/* margins around imageable area, */
					/* in inches */
	gx_device_color_info color_info;	/* color information */
	int is_open;			/* true if device has been opened */
} gx_device;
```

The name in the structure should be the same as the name in devs.mak.

gx_device_common is a macro consisting of just the element definitions.

For sophisticated developers only

If for any reason you need to change the definition of the basic device structure, or add procedures, you must change the following places:

- This document and NEWS (if you want to keep the documentation up to date).
- The definition of gx_device_common and/or the procedures in gxdevice.h.
- The null device in gsdevice.c. (Note that this device does not allow procedure defaulting.)
- The tracing "device" in gstdev.c. (Ditto.)
- The command list "device" in gxclist.c. (Ditto.)
- The clip list accumulation and clipping "devices" in gxcpath.c. (Ditto.)
- The "memory" devices in gdevmem.h and gdevmem*.c. (Ditto.)
- The generic printer device macros in gdevprn.h.
- The generic printer device code in gdevprn.c.
- All the real devices in the standard Ghostscript distribution, as listed in devs.mak. (Most of the printer devices are created with the macros in gdevprn.h, so you may not have to

          edit the source code for them.)
      - Any other drivers you have that aren't part of the standard
          Ghostscript distribution.

You may also have to change the code for gx_default_get_props and/or gx_default_put_props (in gsdevice.c). Note that if all you are doing is adding optional procedures, you do NOT have to modify any device drivers other than the ones specifically listed above; Ghostscript will substitute the default procedures properly.

****
******** Coding conventions ********
****

While most drivers (especially printer drivers) follow a very similar template, there is one important coding convention that is not obvious from reading the code for existing drivers: Driver procedures must not use malloc to allocate any storage that stays around after the procedure returns. Instead, they must use gs_malloc and gs_free, which have slightly different calling conventions. (The prototypes for these are in gs.h, which is included in gx.h, which is included in gdevprn.h.) This is necessary so that Ghostscript can clean up all allocated memory before exiting, which is essential in environments that provide only single-address-space multi-tasking (specifically, Microsoft Windows).

```
char *gs_malloc(uint num_elements, uint element_size,
  const char *client_name);
```

Like calloc, but unlike malloc, gs_malloc takes an element count and an element size. For structures, num_elements is 1 and element_size is sizeof the structure; for byte arrays, num_elements is the number of bytes and element_size is 1.

The client_name is used for tracing and debugging. It must be a real string, not NULL. Normally it is the name of the procedure in which the call occurs.

```
void gs_free(char *data, uint num_elements, uint element_size,
  const char *client_name);
```

Unlike free, gs_free demands that num_elements and element_size be supplied. It also requires a client name, like gs_malloc.

****
******** Types and coordinates ********
****

Coordinate system

Since each driver specifies the initial transformation from user to device coordinates, the driver can use any coordinate system it wants, as long as a device coordinate will fit in an int. (This is only an issue on MS-DOS systems, where ints are only 16 bits. User coordinates are represented as floats.) Typically the coordinate system will have (0,0) in the upper left corner, with X increasing to the right and Y increasing toward the bottom. This happens to be the coordinate system that all the currently supported devices use. However, there is supposed to be nothing in the rest of Ghostscript that assumes this.

Drivers must check (and, if necessary, clip) the coordinate parameters given to them: they should not assume the coordinates will be in bounds. The fit_fill and fit_copy macros in gxdevice.h are very helpful in doing this.

Color definition

Ghostscript represents colors internally as RGB or CMYK values. In communicating with devices, however, it assumes that each device has a palette of colors identified by integers (to be precise, elements of type gx_color_index). Drivers may provide a uniformly spaced gray ramp or color cube for halftoning, or they may do their own color approximation, or both.

The color_info member of the device structure defines the color and gray-scale capabilities of the device. Its type is defined as follows:

```
typedef struct gx_device_color_info_s {
      int num_components;           /* 1 = gray only, 3 = RGB, */
                              /* 4 = CMYK */
      int depth;              /* # of bits per pixel */
      gx_color_value max_gray;      /* # of distinct gray levels -1 */
      gx_color_value max_rgb;       /* # of distinct color levels -1 */
                              /* (only relevant if num_comp. > 1) */
      gx_color_value dither_gray;   /* size of gray ramp for halftoning */
      gx_color_value dither_rgb;    /* size of color cube ditto */
                              /* (only relevant if num_comp. > 1) */
} gx_device_color_info;
```

The following macros (in gxdevice.h) provide convenient shorthands for initializing this structure for ordinary black-and-white or color devices:

```
#define dci_black_and_white { 1, 1, 1, 0, 2, 0 }
#define dci_color(depth,maxv,dither) { 3, depth, maxv, maxv, dither, dither }
```

The idea is that a device has a certain number of gray levels (max_gray +1) and a certain number of colors (max_rgb +1) that it can produce directly. When Ghostscript wants to render a given RGB color as a device color, it first tests whether the color is a gray level. (If num_components is 1, it converts all colors to gray levels.) If so:

- If max_gray is large (>= 31), Ghostscript asks the device to approximate the gray level directly. If the device returns a gx_color_value, Ghostscript uses it. Otherwise, Ghostscript assumes that the device can represent dither_gray distinct gray levels, equally spaced along the diagonal of the color cube, and uses the two nearest ones to the desired color for halftoning.

If the color is not a gray level:

- If max_rgb is large (>= 31), Ghostscript asks the device to approximate the color directly. If the device returns a gx_color_value, Ghostscript uses it. Otherwise, Ghostscript assumes that the device can represent dither_rgb * dither_rgb * dither_rgb distinct colors, equally spaced throughout the color cube, and uses two of the nearest ones to the desired color for halftoning.

Types

Here is a brief explanation of the various types that appear as parameters or results of the drivers.

gx_color_value (defined in gxdevice.h)

This is the type used to represent RGB color values. It is currently equivalent to unsigned short. However, Ghostscript may use less than the full range of the type to represent color values: gx_color_value_bits is the number of bits actually used, and gx_max_color_value is the maximum value (equal to 2^gx_max_color_value_bits - 1).

gx_device (defined in gxdevice.h)

This is the device structure, as explained above.

gs_matrix (defined in gsmatrix.h)

This is a 2-D homogenous coordinate transformation matrix, used by many Ghostscript operators.

gx_color_index (defined in gxdevice.h)

This is meant to be whatever the driver uses to represent a device color. For example, it might be an index in a color map. Ghostscript doesn't ever do any computations with these values: it gets them from map_rgb_color or map_cmyk_color and hands them back as arguments to several other procedures. The special value gx_no_color_index (defined as (gx_color_index)(-1)) means "transparent" for some of the procedures. The type definition is simply:

```
typedef unsigned long gx_color_index;
```

gs_prop_item (defined in gsprops.h)

This is an element of a property list, which is used to read and set attributes in a device. See the comments in gsprops.h, and the description of the get_props and put_props procedures below, for more detail.

gx_bitmap (defined in gxbitmap.h)

This structure type represents a bitmap to be used as a tile for filling a region (rectangle). Here is a copy of the relevant part of the file:

```
/*
 * Structure for describing stored bitmaps.
 * Bitmaps are stored bit-big-endian (i.e., the 2^7 bit of the first
 * byte corresponds to x=0), as a sequence of bytes (i.e., you can't
 * do word-oriented operations on them if you're on a little-endian
 * platform like the Intel 80x86 or VAX).  Each scan line must start on
 * a (32-bit) word boundary, and hence is padded to a word boundary,
 * although this should rarely be of concern, since the raster and width
 * are specified individually.  The first scan line corresponds to y=0
 * in whatever coordinate system is relevant.
```

```
 *
 * For bitmaps used as halftone tiles, we may replicate the tile in
 * X and/or Y, but it is still valuable to know the true tile dimensions.
 */
typedef struct gx_bitmap_s {
      byte *data;
      int raster;             /* bytes per scan line */
      gs_int_point size;            /* width, height */
      gx_bitmap_id id;
      ushort rep_width, rep_height; /* true size of tile */
} gx_bitmap;
```

****
******** Driver procedures ********
****

All the procedures that return int results return 0 on success, or an appropriate negative error code in the case of error conditions. The error codes are defined in gserrors.h. The relevant ones for drivers are as follows:

gs_error_invalidfileaccess
    An attempt to open a file failed.

gs_error_limitcheck
    An otherwise valid parameter value was too large for the implementation.

gs_error_rangecheck
    A parameter was outside the valid range.

gs_error_VMerror
    An attempt to allocate memory failed. (If this happens, the procedure should release all memory it allocated before it returns.)

If a driver does return an error, it should use the return_error macro rather than a simple return statement, e.g.,

```
      return_error(gs_error_VMerror);
```

This macro is defined in gx.h, which is automatically included by gdevprn.h but not by gserrors.h.

Most of the procedures that a driver may implement are optional. If a device doesn't supply an optional procedure <proc>, the entry in the procedure structure may be either gx_default_<proc>, e.g. gx_default_tile_rectangle, or NULL or 0. (The device procedure must also call the gx_default_ procedure if it doesn't implement the function for particular values of the arguments.) Since C compilers supply 0 as the value for omitted structure elements, this convention means that statically initialized procedure structures will continue to work even if new (optional) members are added.

Life cycle

Ghostscript "opens" and "closes" drivers explicitly; a driver can assume that no output operations will be done through it while it is closed.

Ghostscript keeps track of whether a given driver is open, so a driver will never be opened when it is already open, or closed when it is already closed.

The following are the only driver procedures that may be called when the driver is closed:

    open_device
    get_initial_matrix
    get_props
    put_props

Open/close/sync

```
int (*open_device)(P1(gx_device *)) [OPTIONAL]
```

Open the device: do any initialization associated with making the device instance valid. This must be done before any output to the device. The default implementation does nothing.

```
void (*get_initial_matrix)(P2(gx_device *, gs_matrix *)) [OPTIONAL]
```

Construct the initial transformation matrix mapping user coordinates (nominally 1/72" per unit) to device coordinates. The default procedure computes this from width, height, and x/y_pixels_per_inch on the assumption that the origin is in the upper left corner, i.e.

```
xx = x_pixels_per_inch/72, xy = 0,
yx = 0, yy = -y_pixels_per_inch/72,
tx = 0, ty = height.
```

```
int (*sync_output)(P1(gx_device *)) [OPTIONAL]
```

Synchronize the device. If any output to the device has been buffered, send / write it now. Note that this may be called several times in the process of constructing a page, so printer drivers should NOT implement this by printing the page. The default implementation does nothing.

```
int (*output_page)(P3(gx_device *, int num_copies, int flush)) [OPTIONAL]
```

Output a fully composed page to the device. The num_copies argument is the number of copies that should be produced for a hardcopy device. (This may be ignored if the driver has some other way to specify the number of copies.) The flush argument is true for showpage, false for copypage. The default definition just calls sync_output. Printer drivers should implement this by printing and ejecting the page.

```
int (*close_device)(P1(gx_device *)) [OPTIONAL]
```

Close the device: release any associated resources. After this, output to the device is no longer allowed. The default implementation does nothing.

Color mapping

A given driver normally will implement either map_rgb_color or map_cmyk_color, but not both; black-and-white drivers do not need to implement either one.

```
gx_color_index (*map_rgb_color)(P4(gx_device *, gx_color_value red,
  gx_color_value green, gx_color_value blue)) [OPTIONAL]
```

Map a RGB color to a device color. The range of legal values of the RGB arguments is 0 to gx_max_color_value. The default algorithm uses the map_cmyk_color procedure if the driver supplies one, otherwise returns 1 if any of the values exceeds gx_max_color_value/2, 0 otherwise.

Ghostscript assumes that for devices that have color capability (i.e., color_info.num_components > 1), map_rgb_color returns a color index for a gray level (as opposed to a non-gray color) iff red = green = blue.

```
gx_color_index (*map_cmyk_color)(P5(gx_device *, gx_color_value cyan,
  gx_color_value magenta, gx_color_value yellow, gx_color_value black))
  [OPTIONAL]
```

Map a CMYK color to a device color. The range of legal values of the CMYK arguments is 0 to gx_max_color_value. The default algorithm calls the map_rgb_color procedure, with suitably transformed arguments.

Ghostscript assumes that for devices that have color capability (i.e., color_info.num_components > 1), map_cmyk_color returns a color index for a gray level (as opposed to a non-gray color) iff cyan = magenta = yellow.

```
int (*map_color_rgb)(P3(gx_device *, gx_color_index color,
  gx_color_value rgb[3])) [OPTIONAL]
```

Map a device color code to RGB values. The default algorithm returns (0 if color==0 else gx_max_color_value) for all three components.

Drawing

All drawing operations use device coordinates and device color values.

```
int (*fill_rectangle)(P6(gx_device *, int x, int y,
  int width, int height, gx_color_index color))
```

Fill a rectangle with a color. The set of pixels filled is {(px,py) | x <= px < x + width and y <= py < y + height}. In other words, the point (x,y) is included in the rectangle, as are (x+w-1,y), (x,y+h-1), and (x+w-1,y+h-1), but *not* (x+w,y), (x,y+h), or (x+w,y+h). If width <= 0 or height <= 0, fill_rectangle should return 0 without drawing anything.

```
int (*draw_line)(P6(gx_device *, int x0, int y0, int x1, int y1,
  gx_color_index color)) [OPTIONAL]
```

Draw a minimum-thickness line from (x0,y0) to (x1,y1). The precise set of points to be filled is defined as follows. First, if y1 < y0, swap (x0,y0) and (x1,y1). Then the line includes the point (x0,y0) but not the point (x1,y1). If x0=x1 and y0=y1, draw_line should return 0 without drawing anything.

Bitmap imaging

Bitmap (or pixmap) images are stored in memory in a nearly standard way.

The first byte corresponds to (0,0) in the image coordinate system: bits (or polybit color values) are packed into it left-to-right. There may be padding at the end of each scan line: the distance from one scan line to the next is always passed as an explicit argument.

```
int (*copy_mono)(P11(gx_device *, const unsigned char *data, int data_x,
  int raster, gx_bitmap_id id, int x, int y, int width, int height,
  gx_color_index color0, gx_color_index color1))
```

Copy a monochrome image (similar to the PostScript image operator). Each scan line is raster bytes wide. Copying begins at (data_x,0) and transfers a rectangle of the given width at height to the device at device coordinate (x,y). (If the transfer should start at some non-zero y value in the data, the caller can adjust the data address by the appropriate multiple of the raster.) The copying operation writes device color color0 at each 0-bit, and color1 at each 1-bit: if color0 or color1 is gx_no_color_index, the device pixel is unaffected if the image bit is 0 or 1 respectively. If id is different from gx_no_bitmap_id, it identifies the bitmap contents unambiguously; a call with the same id will always have the same data, raster, and data contents.

This operation is the workhorse for text display in Ghostscript, so implementing it efficiently is very important.

```
int (*tile_rectangle)(P10(gx_device *, const gx_bitmap *tile,
  int x, int y, int width, int height,
  gx_color_index color0, gx_color_index color1,
  int phase_x, int phase_y)) [OPTIONAL]
```

Tile a rectangle. Tiling consists of doing multiple copy_mono operations to fill the rectangle with copies of the tile. The tiles are aligned with the device coordinate system, to avoid "seams". Specifically, the (phase_x, phase_y) point of the tile is aligned with the origin of the device coordinate system. (Note that this is backwards from the PostScript definition of halftone phase.) phase_x and phase_y are guaranteed to be in the range [0..tile->width) and [0..tile->height) respectively.

If color0 and color1 are both gx_no_color_index, then the tile is a color pixmap, not a bitmap: see the next section.

Pixmap imaging

Pixmaps are just like bitmaps, except that each pixel occupies more than one bit. All the bits for each pixel are grouped together (this is sometimes called "chunky" or "Z" format). The number of bits per pixel is given by the color_info.depth parameter in the device structure: the legal values are 1, 2, 4, 8, 16, 24, or 32. The pixel values are device color codes (i.e., whatever it is that map_rgb_color returns).

```
int (*copy_color)(P9(gx_device *, const unsigned char *data, int data_x,
  int raster, gx_bitmap_id id, int x, int y, int width, int height))
```

Copy a color image with multiple bits per pixel. The raster is in bytes, but x and width are in pixels, not bits. If the device doesn't actually support color, this is OPTIONAL; the default is equivalent to copy_mono with color0 = 0 and color1 = 1. If id is different from gx_no_bitmap_id, it identifies the bitmap contents unambiguously; a call with the same id will always have the same data, raster, and data contents.

tile_rectangle can also take colored tiles. This is indicated by the color0 and color1 arguments both being gx_no_color_index. In this case, as for copy_color, the raster and height in the "bitmap" are interpreted as for real bitmaps, but the x and width are in pixels, not bits.

Reading bits back

```
int (*get_bits)(P4(gx_device *, int y, byte *str, byte **actual_data))
  [OPTIONAL]
```

Read one scan line of bits back from the device into the area starting at str, starting with scan line y. If the bits cannot be read back (e.g., from a printer), return -1; otherwise return a value as described below. The contents of the bits beyond the last valid bit in the scan line (as defined by the device width) are unpredictable.

If actual_data is NULL, the bits are always returned at str. If actual_data is not NULL, get_bits may either copy the bits to str and set *actual_data = str, or it may leave the bits where they are and return a point to them in *actual_data. In the latter case, the bits are guaranteed to start on a 32-bit boundary and to be padded to a multiple of 32 bits; also in this case, the bits are not guaranteed to still be there after the next call on get_bits.

Properties

Devices may have an open-ended set of properties, which are simply pairs consisting of a name and a value. The value may be of various types: integer, boolean, float, string, array of integer, or array of float.

Property lists are somewhat complex. If your device has properties beyond those of a straightforward display or printer, we strongly advise using the code for the default implementation of get_props and put_props in gsdevice.c as a model for your own code.

```
int (*get_props)(P2(gx_device *dev, gs_prop_item *plist)) [OPTIONAL]
```

Read all the properties of the device into the property list at plist. Return the number of properties. See gsprops.h for more details, gx_default_get_props in gsdevice.c for an example.

If plist is NULL, just return the number of properties plus the total number of elements in all array-valued properties. This is how the getdeviceprops operator finds out how much storage to allocate for the property list.

```
int (*put_props)(P3(gx_device *dev, gs_prop_item *plist,
  int count)) [OPTIONAL]
```

Set the properties of the device from the property list at plist. Return 0 if everything was OK, an error code (gs_error_undefined/typecheck/rangecheck/limitcheck) if some property had an invalid type or out-of-range value. See gsprops.h for more details, gx_default_put_props in gsdevice.c for an example.

Changing device properties may require closing the device and reopening it. If this is the case, the put_props procedure should just close the device; a higher-level routine (gs_putdeviceprops) will reopen it.

External fonts

Drivers may include the ability to display text. More precisely, they may supply a set of procedures that in turn implement some font and text handling capabilities. These procedures are documented in another file, xfonts.doc. The link between the two is the driver procedure that supplies the font/text procedures:

```
xfont_procs *(*get_xfont_procs)(P1(gx_device *dev)) [OPTIONAL]
```

Return a structure of procedures for handling external fonts and text display. A NULL value means that this driver doesn't provide this capability.

For technical reasons, a second procedure is also needed:

```
gx_device *(*get_xfont_device)(P1(gx_device *dev)) [OPTIONAL]
```

Return the device that implements get_xfont_procs in a non-default way for this device, if any. Except for certain special internal devices, this is always the device argument.