

PROforma documentation

PROGS, Professional & Graphical Software

Dr. Frans Hemerijckxlaan 13 /1

2650 Edegem

BELGIUM

tel : +32 (0)3/ 457 84 88 fax : +32 (0)3/ 458 62 07 e-mail : joachim@club.innet.be

www : http://www.club.innet.be/~year2827

1. [Introduction](#)
 1. [What is PROforma](#)
 2. [This manual](#)
 3. [Present, Past an Future](#)
 4. [Installation](#)
2. [Configuration](#)
3. [PROforma Concepts](#)
4. [Imaging Model](#)
5. [Graphics](#)
 1. [Gstate](#)
 2. [Drivers](#)
 3. [Transformation matrix](#)
 4. [Drawing parameters](#)
 5. [Clipping path](#)
 6. [Building a path](#)
 7. [Controlling the visible area](#)
 8. [Controlling the page](#)
 9. [Displaying pictures](#)
 10. [Windowing aids](#)
6. [Font Management](#)
 1. [Font Loading](#)
 2. [Font Information](#)
 3. [Available Fonts](#)
 4. [Text display](#)
 5. [String handling](#)
 6. [Supported character set](#)

PROforma documentation

PROGS, Professional & Graphical Software

Dr. Frans Hemerijckxlaan 13 /1

2650 Edegem

BELGIUM

tel : +32 (0)3/ 457 84 88 fax : +32 (0)3/ 458 62 07 e-mail :

joachim@club.innet.be

www : <http://www.club.innet.be/~year2827>

1. [Introduction](#)
 1. [What is PROforma](#)
 2. [This manual](#)
 3. [Present, Past an Future](#)
 4. [Installation](#)
2. [Configuration](#)
3. [PROforma Concepts](#)
4. [Imaging Model](#)
5. [Graphics](#)
 1. [Gstate](#)
 2. [Drivers](#)
 3. [Transformation matrix](#)
 4. [Drawing parameters](#)
 5. [Clipping path](#)
 6. [Building a path](#)
 7. [Controlling the visible area](#)
 8. [Controlling the page](#)
 9. [Displaying pictures](#)
 10. [Windowing aids](#)
6. [Font Management](#)
 1. [Font Loading](#)
 2. [Font Information](#)
 3. [Available Fonts](#)
 4. [Text display](#)

5. [String handling](#)
6. [Supported character set](#)
7. [Cache handling](#)
8. [Charpaths](#)
7. [PROforma sessions](#)
8. [Functions](#)
9. [Writing your own ...](#)
 1. [... bitmap drivers](#)
 2. [... printer drivers](#)
 3. [... picture drivers](#)

PROGS, Professional & Graphical Software
last edited February 14, 1996

PROforma introduction

- [What is PROforma](#)
 - [This manual](#)
 - [Present, Past and Future](#)
 - [Installation](#)
-

What is PROforma

PROforma is short for 'PROGS Font & Raster Manager', and it does exactly what this name suggest. It is a library of routines to manage and display vector graphics and fonts on (raster) devices like screens and printers.

The availability of a separate program to manage graphics and fonts has several advantages. It allows application developers to create output of equal quality (resolution permitting) on several devices, and they can share resources. In short this means that the PROforma library only has to be loaded once, independent of the number of applications which use it. Also fonts only have to be loaded once, and can be shared between applications.

PROforma was originally developed as the graphics library for LINEdesign. That does not mean that this is the only kind of application for which PROforma is of use. PROforma is perfectly suitable as well for desktop publishers, word processors, business graphics and all applications which want high quality output (which must be just about every application except compilers and games). Actually, even at the time of writing there are things which are possible with PROforma and can't be accessed through LINEdesign.

More recently PROforma has been redesigned to a great extent, to make it even more future proof, easier to extend (both internally, and by writing drivers). There have been some changes to make it easier to write a window manager (for ProWesS) and complete support of colour has been added.

As a library, PROforma has the form of dynamic link library (DLL) (if you don't know what that is, don't worry).

This manual

This manual is intended to explain in detail what PROforma is about, how it operates, how it should be used and how it can be extended. For some specific details like possible errors of the access routines, we would like to refer to the *PROforma_ddf* DATAdesign file.

We (and everybody who uses this manual) would like it very much if you could send us any comments about this manual, like

- omissions
- inaccuracies or mistakes
- typing and/or spelling mistakes
- making this manual better English
- anything else (positive comments are also always appreciated)

At the bottom of each page is mentioned when the HTML document was last modified. I will try to keep this date correct, however it is only meant to indicate changes in the information provided, I will not change that date when correcting spelling mistakes or HTML errors.

Present, Past and Future

PROforma is originally developed as the graphics library for LINEdesign. When we started developing LINEdesign v2, we felt that the graphics routines we used were too slow, and also too restrictive. On the other hand, LINEdesign v1 was quite greedy on memory. Therefore, we threw away all the old routines, and started writing a new, more powerful and faster set of routines. During this development, we even introduced some concepts (like the clipping path), which are not used in LINEdesign. On the other hand, the graphics library was expanded to allow efficient editing on screen.

So what do we have now ??

We have a system that can efficiently render and display fonts. All fonts can be shared among applications. A font cache is used to speed up the handling of fonts. Even the font cache and everything in it is shared amongst applications. Fonts are rendered using proper hinting (if the font includes the hints).

The system can draw lines and curves either stroked (with given accuracy and thickness), or filled (using either in/out or winding rule).

Anything can be displayed in any gray shade or colour. If wanted, everything can also be clipped by regular or irregular shapes. Transformation matrices can be applied on the page.

The user can define which part of the coordinate space is actually visible on page (or screen).

Bitmapped pictures can be directly displayed. This allows the user to include screens in his or her output.

Possibilities for the future ??

We want to improve the control over how colour is produced, allowing the user to define how the colour patterns are formed. We also want to make it possible to use a pattern (drawn using PROforma of course) to be used as "colour". Also, we want to add dashed lines, and some variations on line caps, line joins and maybe even some kinds calligraphic lines.

PROforma already contains three kinds of drivers : bitmap drivers, printer (or screen) drivers, and picture drivers. At this point the bitmap drivers are only used internally in PROforma, but we are thinking of making them accessible from outside PROforma, so that general bitmap graphics routines can be written using the primitive commands in the bitmap drivers.

Although PROforma already uses hinting when displaying the fonts, we would like to examine whether we can further improve the quality of hinted

fonts at very small sizes (especially when displayed on screens, e.g. in ProWesS).

Of course we continually try to improve PROforma's speed.

Installation

PROforma is a job which makes itself available to clients in the form of a Dynamic Link Library (a thing with an efficient access method).

Some extensions have to be loaded for PROforma to run : the dynamic link library manager and syslib.

PROforma has the shape of a job, and loads its configuration file ("PROforma_cfg") when it starts. A parameter can be given to PROforma to specify the path where the configuration file can be found (e.g. "win1_pf;flp1" to search on "win1_pf_" and "flp1_" in that order). If no parameter is given or the configuration file is still not found, then first the program default and then the data default devices will be searched.

The fact that PROforma has the form of a job (and not a resident extension as most libraries like the Menu Extensions), has certain advantages. Jobs can always be loaded (if you have enough memory), and jobs can always be removed. When loading a job it is possible to pass a parameter (like where to find the configuration file), which is particularly useful. Also, no memory is wasted if PROforma is loaded while a copy was already running. So if you want to release the memory which is used by PROforma, you can just remove the job. Of course the disadvantage of this scheme is that you can accidentally remove the PROforma job, which is dangerous as all programs which use PROforma will also be removed, so you could loose data that way (in fact, as ProWesS uses PROforma, all jobs which use ProWesS would also be removed).

PROforma Configuration

1. [Configuration file](#)
 2. [Dynamic configuration](#)
-

Configuration file

PROforma reads the initial configuration information from a special file called `PROforma_cfg`. You can specify the directory where this file should be searched when executing PROforma.

Each line in the configuration file is interpreted as a configuration command. Empty lines are discarded as comments. All the other lines are divided in two types, commands and definitions of configuration constants. The lines with a command have a fixed format : the first character is the actual command, the second character should be a space, and the rest of the line is the parameter. All lines which don't have a space as second character are considered as configuration constants.

The configuration commands currently supported by PROforma are :

'%' and ';' :

the line is considered as comment and is discarded.

'S'

the parameter is now the searchpath for fonts. If the first character of the new path is a plus sign, then the path will be added at the end of the existing path.

's'

to set the searchpath for drivers. If the first character of the new path is a plus sign, then the path will be added at the end of the existing path.

'D'

will load the give PROforma [driver](#). It is not necessary to know what kind of driver it is. The names of PROforma driver files normally end in

'_pfd'. The file will be searched on the current searchpath for drivers (cfr 's').

'M'

allow you to specify the maximum amount of memory which can be used by PROforma as buffer to render a page in. If the amount given is negative, then that is the amount of memory which has to remain free (both in bytes).

'C'

specify the size of the [font cache](#). This consists of two numbers, the actual size of the font cache, and the minimum number of different font/size combinations that can be in the cache (one more combination can be in the cache for each [gstate](#)). Each combination of font & size uses about 1.5kB of memory, so this number should not be too big, however, if you use a large fontcache, this number should also be increased.

'c'

Define the size for the colour cache. In PROforma each gstate keeps a few colours which were last used to make sure that the pattern which is used to estimate the colour does not have to be recalculated all the time. This causes a very big speed increase in some operations, especially for drawing pictures. You can choose how many colours are retained in the colour cache. The value is restricted to stay inside the 1..256 range. The default value is 8.

'R'

load a [font](#) file as resident font. A [resident font](#) will always remain in memory (unless PROforma is removed). The first resident font is considered to be the [builtin font](#) (which is essential for proper functioning). The characters from the builtin font are (also) displayed when that character is not available in the current font. It is therefore recommended that the builtin font is as complete as possible. The parameter is the name of the fontfile, which is searched on the searchpath for fonts. If you also want to be able to choose the resident fonts in the fontmap, then you should also include a 'P' command.

'P'

this command adds a font to the [fontmap](#). The fontmap is a matching between font names and their filename. The fontmap is also used to figure out which fonts are available. The command has two parameters,

separated by a semicolon (';'), there should be no spaces before and after the semicolon. The first parameter is the name of the font (which has to be an exact match, including case). The second parameter is the name of the font file. PROforma font files normally end in '_pff'.

'd'

selects the default printer [driver](#). The driver can be given either as the driverid number (in ASCII, this starts with a minus sign as driverid's are negative) or as the full (case sensitive) printer driver name.

'v'

this command should not be used in the PROforma configuration file. The parameter is the minimum version number you want to use. An error (ERR_ISYN) will be returned when the version of PROforma is older than the version requested. An example of the use of this is v 1.14.

The configuration constants are only passed to the last loaded PROforma [printer driver](#) (or if you just selected the default printer driver, than that driver will get the configuration constants). Most printer drivers will normally understand the following configuration constants :

DEFAULT-DEVICE

The parameter if the default device for the printer driver. Some examples are ser1hr or pard. Note that PROforma only prints raw data, so translates should be switched off, hence the 'r' in ser1hr and the 'd' in pard.

PRINTABLE-AREA-SIZE

Allows you to set the size of the *printable area* for your printer. This is the area where output can be visible on the page. The parameters are in typographical points, which has a unit if 1/72 inch or approx. .35 mm.

PRINTABLE-AREA-ORIGIN

Allows you to set the origin of the *printable area* for your printer, or to put it differently, the offset of the printable area from the left and top of the page. The parameters are in typographical points, which has a unit if 1/72 inch or approx. .35 mm.

Dynamic configuration

PROforma can also be configured further while it is already active. PROforma contains a special entry point which allows you to pass configuration lines which are then processed.

The 'PROforma DLL' thing has a CNFG extension which is used for this purpose. This extension accesses a function which accepts a character array as parameter ("char *"). This string is handled as if it was a line in the configuration file. It is thus possible (as mentioned in the previous section) to add printer drivers or fonts, change the default driver etc.

The call to this routine can be done as follows :

```
#include "thing_h"
#include "PROforma_h"

...
    Error err;    /* the error returned by the config routine */
    char *str;    /* the config line which is passed */
    ...
    err=THINGCall(PF_THING_NAME,PF_THING_CNFG,1,str);
    if (err) ... /* error handling */
...

```

PROGS, Professional & Graphical Software
last edited 4 December, 1997

Concepts

- [Graphics State - Gstate](#)
- [Driver & Device](#)
- [Path](#)
- [Subpath](#)
- [Path segment](#)
- [Bezier curve](#)
- [Clipping path](#)
- [Transformation matrix](#)
- [User space](#)
- [Device space](#)
- [Current point](#)
- [PageBbox & PageOrigin](#)
- [WindowSub](#)
- [Font](#)
- [Resident font](#)
- [Builtin font](#)
- [Fontmap](#)
- [Fontlist](#)
- [Font caching](#)
- [Extended character set](#)
- [Kerning](#)
- [Ligatures](#)
- [Tracking](#)
- [Display mode](#)

Graphics State - Gstate

All operations in PROforma need some kind of entry point, just to let PROforma know which device has to be used, what parameters are currently valid, how big the drawing board is etc. To prevent the client (that is the user, or the application which wants to use PROforma) from having to pass these

details with every command, with all possibilities of mistakes, these parameters are combined into a general, internal structure for PROforma. This structure is a 'graphics state' or 'Gstate'. A gstate contains information about :

- the device which is used (screen, printer) and the size of the usable area, specification about how to draw on that device,...
- All parameters about the current drawing methods like colour, line thickness,...
- All information about the fontlist, current font,...
- The current transformation matrix (CTM), list of saved CTM's, current point,...
- All information about the current state of the iterators like device iterator, fontmap iterator, charpath iterator,...
- Information about the current clipping path.

Driver & Device

A driver is a set of characteristics and routines which describe the behaviour of a certain output device (like a printer, or the screen). This usually includes details as size, resolution, available colours... On the other hand you can probably attach your printer both to a serial port, or a parallel port, or maybe you just want your image to output to a file. Therefore, you always have to specify the driver (how to draw), and a device (where to draw) when you allocate a gstate.

PROforma actually works with three kinds of drivers, and one of these has two variants. The most important kind is a *printer driver*, which is used to actually make everything visible (create the output). A variant of this is a *screen driver*. This is similar to a printer driver with the only difference that the output is produced by your monitor instead of your printer. (In this manual the term printer driver always includes the screen drivers, and the word page can be replaced by screen when a screen driver is used).

Another type of driver is a *bitmap driver*, which is used by PROforma to draw in a buffer which is later copied to the actual output device.

Picture drivers are the third and last kind of drivers. They are used to display bitmapped pictures in PROforma. They are a separate kind of driver because there are so many graphics formats in the world.

In this manual, when the general term *driver* is used, without specifying which kind of driver, it normally means a printer driver.

On devices : we strongly recommend the use of the parallel printer port and not the serial port. Serial ports are extremely slow and the amount of data which has to be sent to a printer can be huge. Of course we try to send as little data as possible, but not too many printers can handle compressed data. You should also be aware that serial to parallel converters do NOT speed the transfer of data up. The serial port can handle a certain speed and not more. For instance try sending an A4 page of 300 dpi data on a 9600 baud serial port (standard). This A4 page would need about 966k of data and this would take at least 13 minutes without control bits or correction of control bits (and without handshaking). In short, it will take MORE than 13 minutes to send this data. Luckily, PROforma will normally send less than 966k.

Path

There are actually two meanings for this term, a device interpretation and a graphical interpretation.

- device : a device name, possibly including directory, where files can be found. In PROforma we also allow semicolons in a path name to distinguish between several paths to form a searchpath (that is all paths are tested from left to right until the requested file is found). For example win1_fonts_;flp1_ will search the file first on win1_fonts_font_pff and later (if not found) on flp1_font_pff
- graphical : a collection of subpaths.

Subpath

A move (to define the origin of the subpath) followed by a sequence of path

segments. A subpath can be open, or closed.

Path segment

A path segment is either a line or a bezier curve. Circular arcs are converted to bezier path segments.

Bezier curve

Bezier is a French mathematician who works for Renault and who "invented" a description/display method for curves based on Bernstein polynomials.

In PROforma we only use cubic bezier curves. That is curves which consist of four points: the two endpoints (which are on the curve), and two controlpoints (which are off the curve).

Clipping path

A clipping path is a special path which is not actually drawn, but which is used as a mask for all drawing operation (except text when the cache is used, see later). So the path itself is not drawn, but instead only the places which would be coloured by drawing the path are candidates for all future drawings until the clipping path is cleared.

Transformation matrix

This is a structure (actually a matrix), which explains how the coordinates which are passed to PROforma will be transformed to default user space.

User space

User space is the coordinate system which is used to tell PROforma where and how to display path or text objects. The user space is converted into default user space by PROforma. This user space divides an inch in 72 equal parts, the axes are horizontal (x) and vertical (y). The origin is at the top left, and the axes extend right and down. The unit of 1/72 inch is called a point (pt). Note that a point can be defined slightly different depending on the source: some say there are 72.27 points in an inch, others say 72.307 points per inch.

Please note that PROforma allows to scale the default user space. This would allow the user to specify all coordinates in inches, or centimetres,...

Device space

Internally, PROforma transforms all coordinates from user space to device space. This resembles the position of the picture elements (pixels) of the device. Thus PROforma can decide which pixels to turn on or off.

Current point

The current point is very important when building a path. All path construction commands start at the current point, and set the current point to their endpoint.

It is also the start position on the baseline for text, and set to the end of the text. And it is also the position where a bitmap can be placed.

However, the current point is not always stable. For instance, the current point can not handle changes in the CTM. To avoid this kind of problems, see "PROforma sessions."

PageBbox & PageOrigin

PROforma has a special view on how things have to be visualised on the

chosen device. For starters there is the "page." This entity contains all path and text objects which have to be drawn. The actual image of the object depends on the CTM.

On the other hand, the page has to be visualised on the chosen device. Two things are important for this, the PageBbox which gives the origin and size (on the device) where the page (or part of) will be visualised. Which part of the page will be shown is determined by the PageOrigin, which is the coordinate (in default user space) of the point in the top-left corner of the PageBbox.

WindowSub

Especially for writing such things as window manager which uses PROforma, the concept of a subwindow is introduced. The concept is quite similar to a pagebbox except that a pagebbox is used to redraw part of something bigger (e.g. in interactive applications), and a subwindow is used to draw an independent part of the screen. If the concept of a subwindow would not exist in PROforma, you would have to open a Gstate for each part of the window where you need to draw. Subwindows allow you to reduce a lot of the overhead by using one Gstate for lots of smaller parts in a window.



Font

Collection of graphical shapes, which can usually be combined to give readable text. The font files currently have a lot of similarity with the Adobe Type I font format (slightly adopted for easier access, which also makes them a bit shorter). However this may change in future if we choose to add a different hinting scheme (as the hinting used in type I files is quite obscure, and our current implementation quite unsatisfactory).

Fonts are handled quite efficiently. Each font will only be in memory once. Clients have to state which font they want to use (load), or no longer want to use (unload). Fonts are always referenced by their name. The name of the

font and where to find it are stored in the "fontmap." The fontmap is read when PROforma is loaded. If a font is not in the fontmap, then it can't be used.

PROforma automatically releases a font when there are no gstates which have loaded it. Special routines are included to make sure this is always true (even when a job is force removed). When a font is loaded it is placed in the "fontlist" for that gstate.

Resident font

Normally, fonts are loaded when they are first requested to be usable by a gstate, and they will be discarded when the font is no longer in use by any gstate.

However, PROforma also contains the concept of a resident font. A resident font remains loaded from when it is specified until PROforma is removed (or a reset). A resident font is always available by all gstates, they don't have to say that they are going to use that font. It is most practical when a user often uses the same fonts. They are then always immediately accessible.

Builtin font

PROforma always needs at least one resident font, which is called the builtin font (the first resident font). This font has a special purpose as it will always be used when no other font is selected. It is also intended that you should use a complete font for the builtin. When PROforma displays a character, it will get the shape from the current font. However, it is quite possible that the current font does not contain a shape for the requested character. In that case, PROforma will try to get the shape from the builtin font.

Fontmap

PROforma always keeps a table of all known fonts. This table is used to map a fontname to a fontfile. If a client tries to access a font which is not in the

fontmap, then an error is returned.

The fontmap can not change after PROforma has been loaded (except by removing the PROforma job and loading it again, alas this also removes all clients of PROforma).

Naturally, the fontmap can be examined to find out which fonts can be loaded (if the fontfile is available or fonts is already loaded of course).

Fontlist

Each gstate also keeps a list of the fonts which it can already access. A gstate can only access fonts which are actually loaded. Therefore, when the client request to load a font, it is added to the fontlist of the gstate. The fontlist can be examined to find out which fonts can already be used by a gstate.

Font caching

To increase the drawing speed of text, often used characters are also kept in an internal format which can be displayed much faster than the standard representation on the font. This is called the font cache. There are two limitations imposed by the font cache. The font cache is not capable to display fonts with clipping. Only characters which are not slanted or rotated (so only scaled) can be handled by the font cache. This actually means that some fonts can never be cached (fonts which are internally slanted or rotated). The font cache is also not used for the characters which are partly invisible.

Because the font cache has a limited size, a replacement algorithm must be used. In the case of PROforma, we make sure that only the least recently used characters are removed from the font cache. PROforma makes sure that the capacity of the font cache is not reduced because of fragmentation.

Unfortunately, the font cache doesn't use a magic trick. Although a cached characters draw at least four times faster than a character which was not

cached, you can only gain speed if the character which is cached is used again before it is removed. So if you now in advance that a certain character will only be displayed once, switch off the cache ! This should be done because actually placing a character in the font cache can be hard work !

Extended character set

Because typography uses many characters, PROforma uses a special extended character set, which contains much more characters than the standard character set which is supported by the operating system.

All character strings which are used to display text use an extended character set, unicode. In unicode all characters are a word long (two bytes instead of one).

Actually unicode is a character encoding, while PROforma needs a glyph encoding. This means that some things are not supported by unicode which PROforma needs and vice versa (e.g. ligatures). So PROforma uses only a subset with some extra characters (the ff and fi ligatures). The characters which PROforma considers as "supported" all have a proper character name (which can be found in the PFCharNameTable).

Kerning

To increase the cohesion of a combination of characters, it is often not enough to position all characters side by side, put some character combinations have to be put closer together (or further apart) to make sure that they are visually equally spaced (same amount of whitespace between characters). This process is called kerning. A typical example is the word "AWAY."

Ligatures

Another typesetting feature is that some characters sequences like "ff", "fi",

"fl", "ffi", "ffl" should be replaced by special characters which look better. Ligatures are supported in the Extended Character Set and can therefore be used by the client.

Tracking

Sometimes it may be interesting to add some extra space between all characters. This is called tracking, and can be particularly useful for logo's.

Display mode

PROforma does all drawing inside a buffer. Only when the `PFPageShow` command is called, is the buffer actually displayed to the user.

However, for interactive use, this is not an ideal situation. In fact, users find it very annoying to wait for the drawing to complete, and time only passes very slowly when you are waiting. Therefore, PROforma (and specifically the screen driver) allows you to change the display mode from the default behaviour, to an *update* mode where the screen is continually refreshed with the current state of the drawing buffer. This refreshing is cancelled when the `PFPageShow` command is called.

PROGS, Professional & Graphical Software
last edited June 21, 1996

Imaging Model

PROforma has its own specific way to look at pixels and pages, the two basic entities in this system.

- [Pages](#)
 - [Pixels](#)
 - [CRT screen](#)
 - [dot matrix printer](#)
 - [inkjet or bubblejet printer](#)
 - [laser printer](#)
-

Pages


Because not all devices are capable of changing their output (printers for example), PROforma uses a buffered approach. So instead of drawing on a page, all operators actually draw in a buffer, and this buffer can then be displayed on the actual page (using `PFPageShow`). However, such a buffer can be quite large (typically 1MB for a 300dpi mono A4 page), and there may not be enough memory available for the entire page. Therefore, an actual page can be split into several pieces, and transferring the buffer will only display part of the page.

So pages are built in passes. The client knows how many passes are necessary for each page and has to call the display operators for all visible objects on the page once for each pass. When transferring the buffer to the printer, PROforma immediately makes sure the buffer is ready for the next pass of that page, or, if this was the last pass of the page, it makes sure the buffer is ready for the first pass of the next page. The buffer is however not cleared. This is done to allow small changes to be made in the buffer without redrawing all the other stuff (which is only relevant if the page is produced in one pass and can be particularly useful for interactive use and mailmerging).


The buffered approach is actually taken one step further in PROforma. It also applies to [paths](#). Although all parameter about how to draw the path have to be known in advance, the path is not actually drawn while it is built. The path is only drawn when you call a command to do so.

To be 100% correct, we must state that some device drivers (possibly in some versions and with some parameters) may actually bypass the buffer(s). However, this can only explain some 'unexpected' behaviour in some cases (like marks on the page when the page or path is not drawn). It should never be assumed. In fact you could consider the commands to draw a page or path as end of page or end of path markers, they have to be there !

Pixels

PROforma has it's own convention on pixels. It assumes that pixels are rectangles, and that they are positioned between the grid lines. 

In this picture you see the grid lines, the pixel centres, and the actual pixels. On the right, there is a filled triangle drawn. As you can see, pixels are only drawn when the centre lies inside the triangle. A boundary situation occurs when the edge of the triangle coincides with a pixel centre. In this case the edge is shifted to the right over in infinitely small amount.

This also means that areas which have a thickness of less than a pixel may be (partly) invisible if no pixel centres fall inside the path.  In the picture you see a line which is less than a pixel wide (and not hairline), and which pixels would be drawn.

The same rules apply to stroked paths. However when the linewidth is less than one pixel, the path will be drawn hairline. A hairline is a line with a uniform width of one pixel.

Unfortunately, the view that PROforma has on pixels is ideal and does not conform with most output devices (none probably) There are two differences possible.

For starters, some devices don't draw their pixels as PROforma does it, but at

the actual crossing of the grid lines. This is no problem as it only means there is a shift of half a pixel for the entire page. This causes no problems at all.

On the other hand, pixels are usually round, and they often overlap. To make matters even worse, some printers don't even have a consistent pixel size. We will just explain what the problems are with a few types of devices.

CRT screen

These are the common monitors, and we are lucky. Monitors draw in white, which has the effect that white pixels are larger than black pixels. However, the difference in size is not too large. The average size of the dots is slightly bigger than the addressable resolution. This is quite a good approximation of the PROforma model.

dot matrix printer

Dot matrix printer have round dots which are always equal in size. Dots are usually much larger than the resolution at which they are positioned. Although this produces smoother results, it also meant that output is usually more black than is intended. For instance the difference between a one or two pixel wide line can be very small, even if this is a relatively big difference in user coordinates.

Another problem often encountered in dot matrix printers is banding. This means that there is a regular repetition of lighter and darker horizontal bands. This is mainly caused by the use of ink ribbons. They are also used for printing text and therefore the area in the middle of the ribbon is used more than the top or bottom. The less used area produces darker dots. On the other hand the ribbon also rotates horizontally, and this may also cause a difference in darkness (some parts were used more than others).

inkjet or bubblejet printer

This is generally speaking the same as a dot matrix printer. However, the ink is fluid now, and it is usually absorbed by the paper. This causes an additional problem as the size of the dots now also depends on the type of paper. The shape of each dot can also change, and this also depends on the paper (very local). Inkjet or bubblejet printer usually suffer a lot less of banding (unless one of the jets is blocked). A major advantage of inkjet printers is that they are very good at filling black regions, although the paper may bend because of the wet ink.

laser printer

Laser printers either draw their page in black (most often) or in white (as copiers do). This has certain effects on the result (making it either darker or lighter), and pixels don't always have the same size (especially in corners, this is sometimes corrected or used by the printer (so called resolution enhancement)).

Because of the technology used (toner which sticks to charged particles) laser printers have got problems with small (or thin) areas (like hairline paths, which fade away), and with large black areas (which become lighter in the middle). On the other hand, laser printer have the highest real resolution (smallest dots), and gives the highest quality output. Actually, a 300 dpi laser printer giver better, crisper output than a 300 or 360 dpi dot matrix or inkjet printer.

PROGS, Professional & Graphical Software
last edited February 13, 1996

Font Management

A very important part of PROforma concerns the manipulation and displaying of [vector fonts](#). The vector fonts which are used in PROforma are a direct descendant of the Adobe Type I font format, but optimised for efficient access, and low memory consumption. Programs exist to convert Adobe Type 1 fonts for PROforma (pfb2pff).

- [Font Loading](#)
 - [Font Information](#)
 - [Available Fonts](#)
 - [Text display](#)
 - [String handling](#)
 - [Supported character set](#)
 - [Cache handling](#)
 - [Charpaths](#)
-

Font Loading

Fonts have to be loaded upon request of the client, and can also be released from memory by the client. Fonts only take a little more memory than they occupy on disk. Fonts are always referenced by their name. *Font names are case dependant !* If a font is loaded by several [gstates](#), it is only kept in memory once. So a font is not released from memory if there is at least one gstate which has that font loaded (or the font is resident - [resident fonts](#) are never released). A font which is not resident is always removed from memory when there are no more gstates which have loaded that font.

Resident fonts never have to be loaded by a gstate. They are always available (so they are always in the fontlist).

PFFontLoad

Load the given font file into memory, to make that font available to the

client. This command will automatically select the just loaded font (the current fontsize is not affected by this call). This command needs a fontname. The corresponding filename is searched in the [fontmap](#). So only fonts which are in the fontmap can be used. Font files are searched 'as is', and on the configured [path](#).

PFFontUnLoad

Stop using the given font. After PFFontUnLoad the current font is no longer defined. If the font is not loaded by any other [gstate](#) (and the font is not resident), then the memory which this font occupies will be released. This actually allows PROforma to release the font. The client can no longer use that font, unless it is loaded again.

PFFontSelect

Select the font with given name as current font. The new font will be at the current fontsize.

PFFontScale

Scale the current font to the given point size. Scaling a font is always relative with the [CTM](#).

Font Information

PFFontNameGet

Get name of current font.

PFFontFamilyGet

Get name of font family of current font.

PFFontVersionGet

Get version of current font.

PFFontWeightGet

Get weight of current font.

PFFontNoticeGet

Get notice of current font. This usually includes details about creator and/or copyright on the font.

PFCharAvailable

Routine to allow the client to know whether a character (with given unicode) is available in the current font. Note that the character is only searched in the current font. It may actually be displayed when the character is displayed because the character may be extracted from the

builtin font.

PFWidth, PFStringWidth

Get the width of a string.

PFWidthKern, PFStringWidthKern

Get the width of a string, when displayed with [kerning](#).

PFFontBbox, PFFontBbox

Get the FontBbox. This allows you to find out the maximum amount a character can extend to the right and top, and to the left and bottom.

PFFontBbox works slightly different. It tries to approximate the fontbbox in pixels by determining a bbox in which the letters 'W', 'f', 'g', 'm' fit. As it is possible for some characters to extend further in any direction as these letters, you can also add an extra character to the approximation. In general, this returns a bbox which fits most characters (capitals with accents probably not though). □

Available Fonts

Information about all the available fonts. This allows the client to know which fonts are loaded and can be used.

PFFontCount

Get the number of fonts which are loaded (in the [fontlist](#)).

PFFontNext

Select the next font in the list of fonts as the current font. If the last font in the list was the current, an error will be returned (and the current font is not changed).

And you can enquire about all the fonts which are in the fontmap.

PFFontCountAll

Get the number of fonts in the fontmap. Also makes sure that the next enquiry with `PFNextFontName` will return the first font in the [fontmap](#).

PFFontNextName

Get the name of the next font in the fontmap.

Text display

PROforma always uses [UniCode character codes](#). These character code are "short" in size (and not char). A char just didn't allow enough possible characters.

All the routines to display text (or get the width thereof) are available in two variants. The first variant uses unicode characters directly (a unicode string is the same as a c string, `\0` is used as end of string marker). The other variant (the commands which start with "PFString") use strings in the local character set (ordinary strings).

The requested characters are retrieved from the current font at the current fontsize. If the character is not available in the current font, the character will be retrieved from the [builtin font](#). If the builtin font also doesn't contain the character, then an empty character (no width, not visible) is displayed.

PFShow, PFStringShow

Show the requested string at the current position. Characters are placed proportional. The current position will now be the place where the next character should be placed.

PFShowKern, PFStringShowKern

Show the requested string at the current position with [kerning](#). The current position will now be the place where the next character should be placed.

PFShowX, PFStringShowX

Show the requested string at the current position. The advance width of the characters is overwritten with the given values. The vertical advance width is taken as zero. The current position will now be the place where the next character should be placed.

PFShowXY, PFStringShowXY

Show the requested string at the current position. The advance width of the characters is overwritten with the given values. The current position will now be the place where the next character should be placed.

PFShowTrack, PFStringShowTrack

Show the requested string at the current position. The current font size is used, but characters are positioned in such a way to make sure that the

string has the given width (using [tracking](#)).

PFSHOWJust, PFSTRINGSHOWJust

Same as PFSHOW (resp. PFSTRINGSHOW), except that some whitespace is added or subtracted at the space characters, to make sure that the text is displayed with full justification. The requested width of the line has to be specified.

PFSHOWKernJust, PFSTRINGSHOWKernJust

Same as PFSHOWKern (resp. PFSTRINGSHOWKern), except that some whitespace is added or subtracted at the space characters, to make sure that the text is displayed with full justification. The requested width of the line has to be specified.

Normally, PROforma always attempts to be completely device independent. Unfortunately, when displaying text on screen, you often prefer quality above device independence. The main problem is that spaces are usually too small to separate characters properly. Therefore, there are some special routines which force the distance between all characters to be consistent (one pixel). These commands should only be used on screen, as it would remove all character spacing on devices with a higher resolution.

As these commands are only meant to be used on screen devices, they use the PFP prefix. In fact, the string width commands actually return the width in pixels.

PFPShow, PFPStringShow

Show the string using one pixel spaces between all characters.

PFPWidth, PFPStringWidth

Get the width a string occupies when displayed with PFPShow or PFPStringShow commands. The width is given in pixels.

String handling

PFEExtraEOS

All strings which are passed to PROforma are '\0' terminated c strings. However, it may in some cases be useful to have an extra symbol which can be interpreted as end of string marker. Such an extra marker can be

specified with this command. You can reset the extra marker by setting it to '\0'

PFPrintEscape

When strings which have to be displayed are passed using the local character set, not all of the supported character set can be accessed. Therefore, PROforma allows an escape sequence to specify all the characters in the supported character set when normal strings are used. You are then allowed to specify the character by putting the (case dependent) name or (decimal) [unicode](#) between backslashes. However, as this behaviour is not always wanted, you can switch it on or off.

Supported character set

PFCharNameTable

To be able to use the [supported character set](#), PROforma gives you access to a table which contains all the supported characters. Both the name and unicode of each character is given. Please note that the name of the supported characters is case dependent (for example to allow both Egrave and egrave as character names).

PFASCII2UnicodeTable

Although PROforma allows you to use the local character set for most of the font handling commands, it may sometimes be necessary to convert characters in the local character set to unicode. Therefore, there is this table which gives the unicode value for each character in the local character set.

Cache handling

Normally all font display operators try to use the [font cache](#). However, as there are some problems with the usage of the font cache, the use of the font cache can also be switched off. This can be done because all font display operators work as if there is no clipping path when the cache is switched on !

PFCacheUse

Tell PROforma whether or not the fontcache should be used.

Charpaths

For drawing programs like LINEdesign it is interesting to be able to extract the outline of a character from a font as this allows the user to make some individual changes to the characters. Therefore this feature is supported by PROforma. And consists of a loop like :

```
select character which should be converted
DO
    get the next path operator
    process it
UNTIL end of character
```

Note: the PFCharPathInit and PFCharPathEl commands should only be used in such a loop, just as all the other iterators in PROforma (no other font operators should be used inside the loop).

PFCharPathInit

Select the character for which the outline will be extracted. Selects that character from the current font at the current size of that font.

PFCharPathEl

Get the next path element for that character. This can be a move, line, curve, width or end of character command (which can be interpreted as a PFPathDraw). The coordinates which are returned are absolute coordinates, only the character origin has to be added.

PROforma sessions

If you want to use PROforma, there are certain rules you have to follow to assure correct operation. The most important rules concern the order in which the operators have to be used. For instance, you should not change the transformation matrix during the building and drawing of a path. This is done because PROforma is built for efficiency. Everything has to be fast and flexible. This unfortunately means that robustness is not one of PROforma's strong points. Doing unexpected things may cause unexpected results.

To try to make things easier we will now give an outline of a common procedure for programs which generate their output using PROforma. That is for programs which are not interactive. A program which is interactive does not produce pages as suggested here. Generally speaking, the scheme will be very similar though.

The scheme listed here doesn't include two groups of operators. The operators which request information from the system, as these can be used between drawing commands, as they don't change anything in the system. Also some commands like `PFPageScroll` and `PFWindowMove` are not listed. These commands can also be used at about any moment.

There are some general notation used in the following scheme : `command` Execute the command. `{ command }` The command can be repeated zero or more times. `[command]` The command can be executed at most once. `command1 | command2` Either `command1` or `command2` will be executed. `< command >` Composite command, elaborated somewhere else in the scheme. Of course these rules can be combined to form a more elaborate scheme.

```
< PROforma session > =
  PFInitGstate
  [ PFCopies ]
  /* FOR each page */
  {
    < Draw Page >
  }
  PFRemoveGstate
```

```

< Draw Page > =
  /* FOR each pass */
  {
    [ PFPaperColourGray | PFPaperColourRGB | PFPaperColourCMY
    PFPageClear
    < Draw Pass >
    PFPageShow
  }

< Draw Pass > =
  /* FOR each object */
  {
    < Draw Object >
  }

< Draw Object > =
  [ PFPageScale ]
  [ PFPageBboxSet | PFPageOriginSet ]
  [ < Set Draw Parameters > ]
  [ PFSaveCTM ]
  {
    < Draw Path > | < Draw Text > | < Draw Picture >
  }
  [ PFRestoreCTM | PFResetCTM ]
  [ PFPageBboxReset | PFPageBboxRestore ]
  [ PFClearClip ]

< Set Draw Parameters > =
  [ < Change CTM > ]
  [ PFLineWidth ]
  [ PFColourGray | PFColourRGB | PFColourCMYK ]
  [ PFFlatness ]
  [ PFPathMethod | PFPathStroked | PFPathFilled | PFPathEOFille

< Change CTM > =
  {
    PFCTMSet | PFCTMMove | PFCTMScale | PFCTMXScale | PFCTMYS
  }

< Draw Path > =
  < Draw First Subpath >
  {
    [ < Draw Subpath > ]
  }
  PFPathDraw | PFPathClip | PFPathClear

< Draw First Subpath > =

```

```

PFMoveTo
{
    PFLineTo | PFLiner | PFCurveTo | PFCurveR
}
[ PFPathClose ]

< Draw Subpath > =
[ PFMoveTo | PFMoveR ]
{
    PFLineTo | PFLiner | PFCurveTo | PFCurveR
}
[ PFPathClose ]

< Draw Text > =
[ PFFontLoad ]
[ < Set Text Parameters > ]
[ PFCacheUse ]
[ PFPrintEscape ]
[ PFExtraEOS ]
PFMoveTo
{
    PFSHOW          | PFStringShow
    PFSHOWKERN      | PFStringShowKern
    PFSHOWX         | PFStringShowX
    PFSHOWXY        | PFStringShowXY
    PFSHOWTRACK     | PFStringShowTrack
    PFSHOWJUST      | PFStringShowJust
    PFSHOWKERNJUST  | PFStringShowKernJust
}
[ PFFontUnLoad ]
[ PFCacheUse ]

< Set Text Parameters > =
[ PFFontSelect ]
[ PFFontScale ]

< Draw Picture > =
PFMoveTo
.....

```

PROGS, Professional & Graphical Software
last edited February 13, 1996

Functions

For a list of all the functions which are part of PROforma, including the prototype, exact behaviour and possible errors, you should have a look at the PROforma_ddf DATAdesign file.

Two kinds of PROforma access functions

There are two kinds of PROforma access functions. Most of them use coordinates which are passed as [fixpoint](#) values in [user space](#). These functions all start with PF (for PROforma).

However, there are also commands which need parameters which are given directly in [device space](#) or pixels. These functions always start with the PFP prefix (for PROforma Pixels). Some commands are only available in one of the two flavours, and some in both. The PFP commands are mostly provided for efficient access from a window manager (like ProWesS). The functions which work in pixels, usually require you to pass the coordinates as integers. However, for accuracy, the drawing operators will always work in [fixpoint](#) representation.

Parameters

PROforma is a c (c68) programming library. However, it can also be called from other languages (e.g. assembler). The calling and register saving conventions for c68 are valid. This means that parameters are actually passed on the stack. All parameters are four bytes long and contain either an integer (int), a pointer to a string, or a fixpoint number.

Strings

Contrary to the approach which is usually taken by the operating system, we

use null terminated strings (instead of preceding them by the length).

Another problem is that we use two kinds of strings. We have the C standard null terminated strings which contain character codes using the local character set (one byte per character), and we have unicode strings, in which each character is represented by a word (2 bytes). The use of unicode allows us to have character sets which contain more than 256 characters, and have all characters at a fixed spot. Most general characters (about codes 32 to 126) are at about the same position as in the ASCII character set.

fixpoint - pt

This is actually just a representation of a decimal character which can be processed faster than standard floating points. This is done by dividing the actual representation in two parts: an integer part and a fractional part. Each part occupies two bytes in a long word. The most significant bytes are the integer part, and the least significant bytes are the fractional part, expressed as multiples of 1/65536. In binary representation, this means that there is an imaginary dot between the two words in a long word. □

Fixpoint numbers can be added and subtracted just as normal integers. A special routine (`fixmul`) has to be used when you want to multiply them. Some macros are available in "PROforma_h" to manipulate fixpoint values more easily.

```
/*
    most parameters which are passed to PROforma are fixpoints,
    as they give more accuracy than integers, and more speed
    than floats or doubles. A fixpoint number is actually a long,
    with an imaginary dot between the two words. This convention
    allows for fast adding and multiplication, without loosing
    too much accuracy.
*/

#include "err_h"

typedef long pt;
#define pt_one 0x10000
#define pt_half 0x8000
#define pt_quarter 0x4000
```

```

#define pt_hundred 0x640000
#define ptmin 0xc0000000
#define ptmax 0x3fffffff
#define ptmagic 36045          /* .55 in fixpoint */
#define ptcigam 29491         /* .45 in fixpoint */

/*
    Most operations can be done directly on fixed-points :
    addition, subtraction, shifting, multiplication or
    division by integer constants; assignment, assignment
    with zero; comparison, comparison against zero.
    Multiplication and division by floats is OK if the result
    is explicitly cast back to fixed.
    Conversion to and from int and float types must be done
    explicitly. Note that if we are casting a fixed to a
    float in a context where only ratios and not actual values
    are involved, we don't need to take the scale factor into
    account: we can simply cast to float directly.
*/

#define long2pt(x) ((pt)((x)<<16))
#define short2pt(x) ((pt)(x)<<16)
#define pt2short(x) ((short)((x)/65536))
#define pt2rshort(x) pt2short((x)+pt_half)
#define pt2long(x) ((long)((x)/65536))
#define pt2rlong(x) pt2long((x)+pt_half)
#define double2pt(x) ((pt)((x)*(double)65536.0))
#define pt2double(x) ((double)((x)/(65536.0)))

/* Rounding and truncation on fixeds */
#define pt_trunc(x) ((x)&(0xffff0000))
#define pt_round(x) pt_trunc(x+pt_half)
#define pt_ceiling(x) pt_trunc(x+pt_one)
#define pt_fraction(x) ((x)&0xffff)
#define pt_center(x) (pt_trunc(x)+pt_half)

/* special multiplication routine for fixpoint coordinates */
pt fixmul(pt x, pt y); /* returns x*y */

```

PROGS, Professional & Graphical Software
last edited June 28, 1996

Graphics

This document gives an review of what is possible with PROforma when you do not want to use text. Some of these things also have their consequences when displaying text, but we have chosen to make the font management a [separate document](#) as there is so much to say about it.

Note that the PROforma functions (they all return an error code) use a consistent naming scheme (as introduced in syslib). The name always starts with the general module name in capitals ("PF" for PROforma), followed by some words defining the action. Each word starts with a capital, and the most important words (indicating a group of commands) are given first. For example "PFLineWidth" is a PROforma command (PF), concerning lines (Line), and specifically the width (Width).

- [Gstate](#)
- [Drivers](#)
- [Transformation matrix](#)
- [Drawing parameters](#)
- [Clipping path](#)
- [Building a path](#)
- [Controlling the visible area](#)
- [Controlling the page](#)
- [Displaying pictures](#)
- [Windowing aids](#)

Gstate

All commands in PROforma actually require a [gstate](#) as an access point to PROforma. Therefore we have introduced some commands to create and delete gstates.

To make sure that you can draw part of a picture without affecting the current

graphics state, you can save the graphics state, so that it can be restored later. The graphics state which is saved includes the [CTM](#), linewidth, paper and drawing colour, [clipping path](#), current font (when it remains loaded), fontsize, [pagebbox and origin](#), flatness and [current position](#), path drawing method, [usecache](#), extraeos, printescape,...

PFGstateInit

Create a new gstate. The client has to specify which driver to use and the device which should be used for that driver. The client also determines the size of the page and gets to know how many passes are necessary to render a complete page.

Drivers can be specified either by name, or using a driverid. The driverid's are not fixed and depend on the configuration. Positive driverid's are interpreted as a pointer to a string containing the driver name (case dependent). Some driverid's and names are reserved : "default driver", "screen driver" and "dummy driver" are the reserved name, and the values are PF_DRIVER_DEFAULT, PF_DRIVER_SCREEN and PF_DRIVER_DUMMY.

You can also specify the device to which the driver should send it's output. However is NULL or "" is passed, then the output will be sent to the default device (as can be configured). Note that some drivers are only capable of sending their image to one device.

The size of the maximum area depends on the type of the driver. A printer driver will typically have a 595x842pt area, matching an A4 page (excluding marging forced by the printer). The screen driver has a maximum size of 720x540pt. This is approximately a 12inch screen (10x7.5 inch).

The client can specify the requested page size and position on the device. However, this area is clipped to the actually usable area. If the resulting area is non-existing (so no part of the requested area is usable on that printer), an error is reported.

When a gstate is opened, PROforma assumes you want to use the entire page. So the PageOrigin is set to the position of the top left pixel of the

[PageBbox](#) on the device.

Most internal structures are initialised properly by PROforma like the flatness, drawing colour (black), paper colour (white),... However some things are not initialised like the linewidth, and the path type (so this should be set or a path will not become visible).

PFGstateRemove

Remove the given gstate from memory. This releases all the fonts which are not used by other gstates (and not resident), and also releases the current path,...

PFGstateSave

Save the current graphics state. This command does not affect anything, but allows the gstate to be restored later. The graphics states are store in lifo (last in, first out) stack. This command should not be called while building a path which has to be drawn.

PFGstateRestore

Restore the graphics state to an earlier saved version. This command should not be called while building a path which has to be drawn.

Drivers

PROforma also includes routines to enquire about the available [printer drivers](#). This can be necessary because driverid's are not fixed. They can vary between versions and/or configurations of PROforma. You need a [gstate](#) to able to query the available drivers. For this purpose you could initialise a gstate with PF_DRIVER_DUMMY as driver (the dummy driver does not allow you to draw anything and is specificaly intended to query PROforma).

PFDriverCount

Get the number of available drivers. This count does not include the screen driver as this has a fixed driverid (PF_DRIVER_SCREEN). This command also assures that the next call to PFDriverNext for this gstate will return the first driver in the list.

PFDriverNext

Get the driverid and name of the next driver in the list. This list is not

sorted !

Transformation matrix

To make it easy for a client to produce moved, slanted, scaled, rotated,... images, PROforma uses a [transformation matrix](#). This matrix converts given coordinates to coordinates in default [user space](#) (which are then, internally, converted to [device space](#)). There is always a current transformation matrix. The default does nothing (unity matrix).

To allow the client to set the matrix to a certain state, which can be altered and later recovered, it is possible to save and restore the CTM.

PFCTMMove

Move the origin of the current transformation matrix. This means that all objects are actually moved over the given distance. This command could be simulated by adding the given coordinates to all following absolute coordinates. This routine is a macro for PFCTMSet.

PFCTMScale

Scale the current transformation matrix. The origin remains in the same position. All following objects are enlarged by the given factor. This command maintains the ratio, so everything is scaled by an equal amount in all directions. This routine is a macro for PFCTMSet.

PFCTMXScale

Scale the current transformation matrix along the x axis. The origin remains in the same position. All following objects are enlarged along the x axis by the given factor. This command does not maintain the ratio, scaling is only along the x axis (which can be rotated) ! This routine is a macro for PFCTMSet.

PFCTMYScale

Scale the current transformation matrix along the y axis. The origin remains in the same position. All following object are enlarged along the y axis by the given factor. This command does not maintain the ratio, scaling is only along the y axis (which can be rotated) ! This routine is a macro for PFCTMSet.

PFCTMSet

Set the current transformation matrix to the given value, which is relative to the previous value of the CTM. This command should not be performed during the building of a path as that would give problems when closing the current subpath (just as the other commands which change the CTM).

PFCTMReset

Reset the current transformation matrix to the standard values, being all measurements in default user space, the origin at the top left, and axes extending right and down. This command clears the list of CTM's which are saved. Default user space is normally 1/72 inch (approx .35mm), but this can be changed with `PFPageScale`.

PFCTMSave

This command allows the client to save the values of the CTM, so that it can later be recovered. Can be called as often as needed (memory permitting, not that it uses much).

PFCTMRestore

Restore the CTM to a previously saved CTM. You should consider the list of CTM's as a last in, first out (LIFO) stack. `PFCTMRestore` removes the last topmost CTM from the stack and makes it the current.

PFCTMRestoreKeep

This command is functionally equivalent to

```
PFCTMRestore(gstate);  
PFCTMSave(gstate);
```

It restores the last saved CTM, but doesn't remove that value from the stack of saved CTMs.

Drawing parameters

Because PROforma actually attempts to produce pages as fast and efficient as possible, the client has to know how the path has to be visualised before it is actually built.

PFPathMethod

Set the drawing method for the following paths. This is a general routine

to set this drawing method, with a parameter. Especially useful when the path is defined in a data structure. Macros are provided to the individual cases (given below). If an invalid drawing method is passed, the drawing method is undefined (nothing is drawn).

PFPathStroked

Notify PROforma that all future path construction commands will apply to stroked paths. Stroked paths have a thickness (as specified by `PFLinewidth`), and always have round caps and round joins. Stroked lines are always visible. This means that even zero width lines are drawn one pixel wide (also called hairline). This is done to make sure that they don't disappear from the final result. Note however that hairlines can be so thin on certain high resolution devices that they may be invisible. Also some devices (like some laser printers) are very bad at colouring small areas, which can have the same result.

PFPathFilled

Notify PROforma that all future path construction commands will apply to paths which are filled using the winding rule. Please note that areas which are less than a pixel wide or tall can disappear from the final result. To determine whether an area is filled by the winding rule. Initialise the winding counter to zero and draw a line to infinity. For every edge which is crossed you should add one to the winding counter if the edge goes up (left if the edge is horizontal). If the edge goes down (right for horizontal), then you should subtract one from the counter. The area will be filled when the winding counter is not zero. All areas have to be closed for this rule to work. The direction of the path is very important as can be seen in the picture.

PFPathEOFilled

Notify PROforma that all future path construction commands will apply to paths which are filled using the even odd rule. Please note that areas which are less than a pixel wide or tall can disappear from the final result, (see "[PROforma Imaging Model](#)"). To determine whether an area is filled by the even odd rule. Initialise the winding counter to zero and draw a line to infinity. For every edge which is crossed you should add one to the winding counter. If the resulting winding counter is odd, then the area will be filled. All areas should be closed for this rule to work.

PFColourGray

Select the current grayshade for drawing. Grayshades are given in percentages. All devices have a few distinct grayshades. Higher resolution devices have more grayshades than low resolution devices.

PFColourRGB

Select the current colour for drawing. RGB colours use an additive colour model, where the red, green and blue components are given. Each component is a percentage, ranging from black (0) to the pure colour (100). Devices always have a native colourspace. If that is not RGB, then the colour which is given is transformed to the devices native colourspace.

PFColourCMYK

Select the current colour for drawing. CMYK colours use a subtractive colour model, where the cyan (kind of blue), magenta (kind of red), yellow and black components are given. Each component is a percentage, ranging from white (0) to the pure colour (100). The black component exists because mixing cyan, magenta and yellow inks, usually turns out more dark brown than black. Therefore the black component should be removed and given separately. Devices always have a native colourspace. If that is not CMYK, then the colour which is given is transformed to the devices native colourspace.

PFPatternMask, PFPatternMaskUser

When a colour doesn't match a solid colour on the output device, then PROforma will produce a fastpattern to approximate the colour. This is a 8x8 (one or more) pattern. You can determine how such patterns are built (how colours are spread in the pattern) with these calls.

PFPatternMask uses a choice of builtin methods, while the User variant allows you to build your own distribution method.

PFLineWidth

Set the linewidth in user coordinates for stroking. All lines and curves which are drawn stroked after this command is given will have the given linewidth until the next PFLineWidth command. The linewidth should not be changed while the path is built as this could cause unexpected results.

PFFlatness

Set the flatness. This is the amount of tolerance the approximation of the [bezier curves](#) allows. A high value increases drawing speed, but decreases the accuracy. If this value is too high, curves will degenerate

to polygons.

Clipping path

To aid in special constructions, you can define a [clipping path](#). This means that of any following drawings, only the parts which fall inside the clipping path are visible.

PFClipClear

Clear the clipping path and the current path (initial state). The parts of the path which are not inside the [PageBbox](#) will never be visualised, irrespective of the clipping path.

PFPathClip

Convert the path which was built into the current clipping path. The path will actually be clipped according to the previous clipping path. The path will also be cleared by this command, and the current point will be reset.

The path will be clipped as drawn with `PFPathFilled` unless the current drawing mode is `PFPathEOFilled`. If the linewidth is hairline (less than a pixel thick), then the clipping path will make everything invisible !

Clipping paths are a powerful tool. They allow you to look through an area. It can be viewed as if the drawing which you draw afterwards is the building of a large pattern which is used to fill the area indicated by the path which is clipped.

Thus a clipping path can for example be used if you want to produce graphics with a gradient fill (the colour tone changes). This can be achieved by using the shape of the drawing to define a clipping path, and then draw a series of blocks with slightly differing colours, which are clipped. This will produce the proper result.

Building a path

Of course you also need commands to build a [path](#). This path can then be drawn, or used as a [clipping path](#). It is strongly advised that no other operators are called while building a path, or between building the path and the actual drawing or clipping. If you do call other operators, the behaviour may be quite unexpected (see [PROforma sessions](#), especially the text show operators should not be used).

PFMoveTo

Set the [current point](#) to the given absolute position. If you were drawing a filled path which was not closed, this will be done automatically. This commands actually starts a new [subpath](#).

PFMoveR

Move the [current point](#) by the given distance. If you were drawing a filled path which was not closed, this will be done automatically. This commands actually starts a new [subpath](#).

PFLineTo

Construct a line from the [current point](#) to the given point in absolute coordinates. After this command, the endpoint will be the new current point.

PFLineR

Construct a line from the [current point](#) with the given displacements. After this command, the endpoint will be the new current point.

PFCurveTo

Construct a [bezier curve](#) from the [current point](#) to the given endpoint, using the given control points for direction (all absolute coordinates). After this command, the endpoint will be the new current point.

PFCurveR

Construct a [bezier curve](#) from the [current point](#) to the given endpoint, using the given control points for direction (all relative coordinates). After this command, the endpoint will be the new current point.

PFClosePath

Make sure the current [subpath](#) is closed. A line segment will be added from the end of the subpath (the current point) to the beginning.

PFPathDraw

Make sure the [path](#) which was built is actually rendered in the buffer.

The path will be empty after this command, and the [current point](#) reset.

PFPathClear

Clear the current [path](#) (make it empty). This also resets the [current point](#).

PROforma also provides some high level drawing routines. These actually use the routines above, but are easier than duplicating the code each time.

PFArcTo

Add a circular arc to the path, starting at the current point and ending at the given point. You have to give the coordinates of the point where the tangents cross. This routine only works properly for arcs which cover less than 90 degrees.

PFArcR

Same as PFArc but with relative coordinates.

PFCircle

Add a circle subpath to the current path. The center of the circle will be at the current point.

PFPie

With the current point as center, build a pie. You have to specify the cosine and sine of the start and end degree and the radius. You can also choose whether the pie is closed or not. A pie which is not closed is just part of a circle.

PFRectangle

Starting from the current point, build a rectangle subpath. The rectangle can have rounded corners with the given radius.

Controlling the visible area

PFPageBboxReset

Reset the [PageBbox](#) and PageOrigin to the original values when the [gstate](#) was initialised (so reset it to the entire visible page).

PFPageBboxRestore

Reset the [PageBbox](#) to the entire visible page, and set the PageOrigin to the point which is already at the top left of this area. This allows the client to restore the PageBbox after a PFPageScroll or PFPageBboxSet command.

PFPageBboxSet

Set the new size of the [PageBbox](#) and moves the origin (relative to

previous value) of visible area. All parameters to this function are in the default [user space](#).

PFPageOriginSet

Move the PageOrigin relative to the previous value. The PageOrigin is the coordinate of the top left point in the [PageBbox](#). All parameters to this function are in the default user space.

PFPageScale

This routine allows the client to set the scaling of a page. It is useful in cases where the page is zoomed in (as in LINEdesign). The default [user space](#) is actually changed from the previous value (initially 72 points/inch) to something else (e.g. to 144 pt/inch if factor was 2). Please note that this changes both the size of the [PageBbox](#) and the PageOrigin. Both are multiplied by the inverse of the scale.

PFPageScale allows the client to transparently change the imaginary size of the page. Contrary to PFCTMScale which has no effect on any parameters in default user space (such as for PFPageBboxSet).

PFPageBboxGet

Get the current size and origin of the [PageBbox](#), and the current value of the PageOrigin. All returned values are given in default [user space](#).

Controlling the page

PFPageShow

Copy the device buffer to the device. Will display the (part of) the page which has already been rendered. If the page will be built in several passes, this routine automatically adjusts the internal structures to render the next pass. In all cases it automatically adjusts the PageBbox to make the entire page visible. The client is responsible for rebuilding the page. If this was the last pass of a page, the internal structures will be adjusted for the first pass again. This command also allows multiple copies of the page to be produced. However this option will only work if the device supports this (like a laser printer). The actual image in the buffer is not cleared by this command.

PFPageScroll

Allows the client to efficiently scroll in interactive applications. It is only allowed to scroll in one direction at a time (horizontal or vertical). The [PageBbox](#) and PageOrigin are automatically adjusted to fit the newly visible space (which probably has to be cleared first).

PFPageClear

Clear the page using the current paper colour (which by default is white). This will only clear the area which falls inside the current [PageBbox](#), as this allows efficient redrawing of the screen for interactive applications.

PFPaperColourGray

Select the paper colour. Grayshades are given in percentages. All devices have a few distinct grayshades. Higher resolution devices have more grayshades than low resolution devices.

PFPaperColourRGB

Select the current paper colour. RGB colours use an additive colour model, where the red, green and blue components are given. Each component is a percentage, ranging from black (0) to the pure colour (100). Devices always have a native colour space. If that is not RGB, then the colour which is given is transformed to the devices native colour space.

PFPaperColourCMYK

Select the current paper colour. CMYK colours use a subtractive colour model, where the cyan (kind of blue), magenta (kind of red), yellow and black components are given. Each component is a percentage, ranging from white (0) to the pure colour (100). The black component exists because mixing cyan, magenta and yellow inks, usually turns out more dark brown than black. Therefore the black component should be removed and given separately. Devices always have a native colour space. If that is not CMYK, then the colour which is given is transformed to the devices native colour space.

PFDisplayMode

This command can be used to set the [display mode](#) which should be used for the page. On screen drivers, this allows you to make sure the users does not have to wait for the drawing to finish before they can see something.

Some devices (usually laser printers) can easily produce following copies

after the first one at great speed. This is supported by PROforma.

PFCopies

Set the number of copies which should be produced. Only works if the device supports it (and the driver of course). If not supported this command returns an error.

Displaying pictures

A special library call is provided to display bitmaps. This is particularly important for DTP applications. Bitmaps can be visualised in any orientation. In fact a picture can encapsulate any kind of graphical object. A picture could also consist of vector graphics and/or (possibly preformatted) text. [Picture drivers](#) could be written for all these purposes.

Pictures always have to be in memory (an image of the file on disk) before they can be used by a picture driver (IOFileLoad). PROforma can then be used to try to recognize which picture driver can display the picture. However, recognizing a picture is not always possible. Sometimes, you have to know the type of the picture in advance (or let the user choose). The picture can be loaded with code like :

```
{
    Size size;
    Channel file;
    char *base;

    IOOpenPath(file, OPEN_OLD, path, &file);
    IOLength(file, &size);
    MEMAllocate(size+sizeof(FileInfo), &base);
    IOFileInfoGet(file, (FileInfo *)base);
    IOFileLoad(file, size, base+sizeof(FileInfo));
    IOClose(file);

    PFMoveTo(gstate, xpos, ypos);
    PFPictureDisplay(gstate, driver, base,
                    xsize, xsize);
}
```

Picture drivers (like printer drivers), can be identified either by name or by

driverid. Some driverid's are also reserved for some specific drivers.

PFPictureCount

Get the number of available picture drivers. This also assures that the next call to PFPictureNext will return the first picture driver which is available.

PFPictureNext

Get the name and id of the next printer driver in the list.

PFPictureRecognize

Let the given picture driver test whether it can recognize the given picture. If it can, the picture driver can visualise the picture. However, the picture driver is only allowed to recognize picture of which it is very sure that it can be displayed. Therefore, if the file format does not include a descriptor (e.g. the type is only indicated by a file extension), then the driver probably has to reject pictures which it can display.

So the answers are either

- YES, I can display the picture
- NO, I don't know if I can display the picture

All the other commands assume that the picture which is passed can be displayed by the chosen picture driver !

The following code can be used to recognize the driver which can display the picture.

```
/* figure out the picture driver id */
{
    int recognized=FALSE;
    int id;
    PFPictureCount(gstate, NULL);
    while (!recognized && !PFPictureNext(gstate, &id, NULL))
    {
        if (PFPictureRecognize(gstate, id, picture base)==ERR_0)
            recognized=TRUE;
    }
    if (recognized)
        the picture id is now stored in id
}
}
```

PFPictureRatio

Get the aspect ratio of the picture, if known.

PFPictureColourCount

Get the number of colours which are used in the picture. This command allows you to figure out how many colours are used in the picture, so that the user can display the picture with different colours. If changing the colours is not possible, this command will indicate that.

PFPictureColourGrayGet

PFPictureColourRGBGet

PFPictureColourCMYKGet

Get the colours which are by default used in the picture. The colours are given in the requested colour space. The colours are filled in an array of the size as returned by PFPictureColourCount.

PFPictureDisplay

PFPictureDisplayGray

PFPictureDisplayRGB

PFPictureDisplayCMYK

Display the picture at the given size at the position indicated by the [current point](#). When using PFPictureDisplay, the default colours for the picture will be used. The other commands need you to pass the colours which should be used (in the colour space which is indicated by the command).

Windowing aids

PFWindowMove

This command adjust the internal structures to a [gstate](#) to match the position of the screen window (it does nothing for non-screen gstates). This should be called if the owning job uses the Pointer Environment and the window has been moved.

PFWindowSubSet

The concept of a [WindowSub](#) can be used recursively. A WindowSub is somewhat similar to the [PageBbox](#), however, the pagebbox is used to redraw part of something, while the WindowSub is used to draw the contents of a part of the gstate, as a WindowSub on the screen. Any following PageBbox has to fall inside the current WindowSub. Any successive WindowSubs are also confined to fall inside the current WindowSub. Any call of PFWindowScale is local inside the current and subsequent WindowSubs. The new WindowSub is always relative with

the current PageBbox, and the coordinates are in user space, which should not be rotated. When the WindowSub is set, the PageOrigin is at (0,0).

PFWindowSubRestore

The concept of a WindowSub can be used recursively, so restoring the WindowSub will only restore the the changes since the matching call of PFWindowSubSet (this acts as a LIFO stack, like PFCTMSet and PFCTMRestore). The CTM and PageScale, PageBbox and PageOrigin are restored to their original values.

PROGS, Professional & Graphical Software
last edited December 4, 1996

Extending PROforma

PROforma can be extended by anybody, by adding new drivers.

bitmap drivers :

A bitmap driver is used by both printer and screen drivers to do the actual drawing in the buffers. They are a separate type because they are usually shared by many drivers and can easily be replaced by better - faster versions.

printer drivers :

Make sure the PROforma output can really be visualised on your chosen output device.

screen drivers :

Actually just a variant of a printer driver, which is intended for interactive use.

picture drivers :

Allow pictures of several types (especially bitmap pictures - but vector pictures are also possible), to be included without hassles, and without the need to know about picture formats.

They can all be written by anybody. All you have to do is actually write the drivers, and make sure PROforma knows about the driver (loads it).

- [How to write a bitmap driver](#)
- [How to write a printer driver](#)
- [How to write a picture driver](#)

How to write your own picture drivers

- [Structure of a picture driver](#)
 - [The member functions](#)
 - [Support routines](#)
 - [Example](#)
-

Structure of a picture driver

A picture driver is an external module which is loaded by PROforma. The init routine should be a data structure of type PICTdriver.

The module identifier has to be "PROforma external driver".

```
/* picture driver definition */

typedef struct _PICTdriver {
    struct _PICTdriver *next;    /* next driver in list of drivers
    int identifier;             /* identifies type of driver */

    char name[PF_MAXDRIVERNAME]; /* name of driver - the name of
                                /* should not start with "-" or "

    /* can we recognize the picture */
    Error (*Recognize)(Gstate, char *base);
    Error (*AspectRatio)(Gstate, char *base, int *xratio, int *yr

    /* display the picture */
    Error (*Display)(Gstate, char *base, pt xsiz, pt ysiz,
                    Error (*ColourSelect)(Gstate, int));

    /* info on colours used in the picture */
    Error (*ColourInfo)(Gstate, char *base, int *count, int *colo
    Error (*ColourGet)(Gstate, char *base, int which, void *colou
```



```
    /* for future extensions */
    Error Handle(int command, ...);
} PICTdriver;
```

The *next* pointer allows you to define several picture drivers in one external module. However, an external module can only contain one type of drivers (in this case picture drivers).

Each driver contains an *identifier* which indicates the type of driver. For a picture driver, the value has to be PF_PICTUREDIVER.

Each picture driver should have a (preferably unique) name. It is advisable to make these names as descriptive as possible, e.g. "QL mode 4 screen, 512x256". Note that driver names are case sensitive !

The member functions

A picture driver is quite simple, as it does not contain many member functions. The main difficulty when writing picture drivers lies in interpreting the actual picture types.

Recognize

Try to recognize the given picture as one that can be displayed by this picture driver. ERR_INAM should be returned if the picture can not be recognized. This routine is included to allow applications to automatically detect the picture driver which has to be used. However, not all picture files embed sufficient information for automatic detection, therefore it is quite allowed to reject all pictures as not recognized.

A picture which is passed to the picture driver is just a block of memory. It is intended that this block starts by the FileInfo structure of the file, followed by a copy of the picture file itself, as if it was loaded with the following code

```
#include "mem_h"
#include "io_h"
#include "PROforma_h"
#define catch(x) do { if (err=(x)) return err; } while (0)
```

```

...
{
    Error err;
    char *base;
    Channel file;
    Size size;

    catch( IOOpenPath(filename, OPEN_OLD, path, &file) );
    catch( IOLength(file, &size) );
    catch( MEMAllocate(size+sizeof(FileInfo), &base) );
    catch( IOFileInfo(file, (FileInfo *)base) );
    catch( IOLoadFile(file, size, base+sizeof(FileInfo)) );
    IOClose(file);
    ...
    PFPictureDisplay(gstate, id, base, xsize, ysize);
}

```

This is the only picture driver access function which can not assume that the picture is of a suitable type. **Pictures may only be recognized when you are quite sure it is not supposed to be displayed by another picture driver.**

AspectRatio

Try to determine the aspect ratio of the given picture. Again, some picture formats do not include the aspect ratio, and it is therefore allowed to fail (ERR_ITNF) on all pictures passed, however, in all cases a guessed aspect ratio should be filled in (4x3 for a full screen is normally a good guess).

The aspect ratio is returned as two integers. This means that when the picture is displayed with a width of *xratio*, then the height should be *yratio* to preserve the aspect ratio of the original.

Display

Actually display a picture. The picture will be displayed at the current position, and at the given size. Pictures can be rotated etc., but all that is handled by the support routine PFPictureElement. For an idea of the recommended way to implement Display, see the [example](#) below.

Please note that the *ColourSelect* parameter is invalid when the picture has fixed colours. In that case the PFColourXXX commands should be

called directly to set the drawing colour.

Before calling the `Display` routine for a picture, PROforma will first adjust the `PageOrigin` to the top left position of the picture. Also, the current graphics state is saved before and restore after the `Display` routine. This makes sure that the graphics state is not affected by drawing pictures.

ColourInfo

Get some information about the *number of colours* and the *colourspace* which is used for the default colours.

This command is used to query the default colours which can be used to display a picture. A picture driver can however choose only to support fixed colours (especially for real-colour images). In that case, zero (0) can be returned as the number of colours.

The possible value for the colourspace are `PF_COLOURSPACE_RGB`, `PF_COLOURSPACE_GRAYSHADE` or `PF_COLOURSPACE_CMYK`.

ColourGet

This function is used by PROforma to build the table with the default colours. The *which* parameter is always in the range $[0..count-1]$, where *count* is returned by `ColourInfo`. The *colour* parameter points to an area where the colour has to be filled in (using the correct colourspace).

Handle

This is a function which is provided for possible future extensions of the PROforma drivers. It should always return `ERR_NIMP`.

Support routines

The PROforma core library contains a support routine which is specifically intended for drawing pictures.

PFPictureElement

Draw a picture element, which is a filled rectangle of given size and at the given position. The size and position are relative with the current

point and in user space. The current drawing colour is used.
This routine will return ERR_ORNG if the rectangle is completely invisible.
This can be used to speed up the drawing of pictures.

Example

code

To start with, the file with all the definition of the data structures which are used by PROforma has to be loaded. Most of this doesn't concern the author of picture drivers, by you do need the definition of the picture driver structure. Accidentally, this also includes PROforma_h.

As the definition of PROforma Core routines is not included in the header files, import the PFPictureElement function.

```
#include "PFmodule.h"  
Error PFPictureElement(Gstate gstate, pt xsiz, pt ysiz, pt xorg,
```

Next up, define the actual structure of the picture data, as this is used by all the member functions. For the sake of the example, I have defined a very simple picture format, including the data needed to recognize the picture, and the aspect ratio (which is optional). The picture itself has one byte for each pixel, giving 256 distinct colours.

```
typedef struct {  
    char identifier[24];    /* "example picture format" */  
    short xsiz, ysiz;      /* size in pixels */  
    short xratio, yratio;  /* pixel aspect ratio (0 if not known)  
    unsigned char data[2]; /* start of picture data */  
} Picture;
```

Start with the real work. For starters, try to recognize a picture as being of the correct type. You should always try to build in as many checks as possible, as illustrated here by assuring that the picture has a real size, and that the aspect ratio is possible (zero indicactes that the ratio is not known).

```
static Error Recognize(Gstate gstate, char *base)  
{
```

```

FileInfo *fi=(FileInfo *)base;
Picture *pict=(Picture *)(base+sizeof(FileInfo));

if (fi->type==FILETYPE_NORMAL &&
    STRSameCD(pict->identifier,"example picture format") &&
    pict->xsiz>0 && pict->ysiz>0)
    return ERR_OK;
else
    return ERR_INAM;
}

```

Get the aspect ratio of the picture. We can assume that the picture passed is of correct type. If the aspect ratio is defined in the picture, than return that. If not, give an error, and assume the picture was a full screen.

```

static Error AspectRatio(Gstate gstate, char *base, int *xratio,
{
    Picture *pict=(Picture *)(base+sizeof(FileInfo));
    if (pict->xratio && pict->yration)
    {
        *xratio=pict->xratio*pict->xsiz;
        *yration=pict->yration*pict->ysiz;
        return ERR_OK;
    }
    else
    {
        *xratio=4; *yration=3;
        return ERR_ITNF;
    }
}

```

Get information about the number of colours used, and the colour space. In this simple example, all pictures have 256 colours, and as pictures usually originate from a screen, the colours will be given as red, green and blue components.

```

static Error ColourInfo(Gstate gstate, char *base, int *count, in
{
    *count=256;
    *space=PF_COLOURSPACE_RGB;
    return ERR_OK;
}

```

Get the default colours used for the picture. The default colours only have to be defined here (except when the colours are fixed). Although the default

colours are often embedded in the picture format, they have to be calculated in this example.

The colour is calculated by looking at the bits. Each colour component has to bits allocated to it, and the two remaining bits can increase the intensity of the colour.

```
static Error ColourGet(Gstate gstate, char *base, int which, void
{
    /* the colours are 8 bit : iirrggbb (i for intensity) */
    ColourRGB *rgb=(ColourRGB *)colour;
    int intensity=((which>>6)&3)+1;

    rgb->red  =intensity * ((which>>4)&3) * (pt_hundred/12);
    rgb->green=intensity * ((which>>2)&3) * (pt_hundred/12);
    rgb->blue =intensity * ((which  )&3) * (pt_hundred/12);
    return ERR_OK;
}
```

To make sure that a picture is always drawn as fast as possible, the background of the picture is drawn first. If this is completely invisible, you can stop immediately.

Each line is also cleared to the background colour before drawing the individual pixels (or spans). This can also indicate that a line can be discarded, especially when drawing on screen (where speed is most important).

When drawing the spans, unnecessary drawing is not done, by making sure that the colour is different from the background colour. This could give a larger speed gain if the background colour would be the most used colour in the picture. However, actually determining that colour each time the picture is displayed, would probably slow the displaying of the picture down.

```
static Error Display(Gstate gstate, char *base, pt xsiz, pt ysiz,
                    Error (*ColourSelect)(Gstate, int))
{
    Picture *pict=(Picture *)(base+sizeof(FileInfo));
    int xpix,ypix=pict->ysiz;
    int colour, backcolour=0;
    short bit=0x80;
    int length, lastcolour;
```

```

unsigned char *linestart=pict->data;
unsigned char *linepos;
int posx;
pt posy=0;

/* make sure that at least some part of the picture is visible
ColourSelect(gstate,backcolour);
if (PFPictureElement(gstate,xsiz,ysiz,0,posy)) return ERR_OK;

xsiz/=pict->xsiz;
ysiz/=pict->ysiz;

while(ypix)
{
    posx=0;
    linepos=linestart;

    /* see if this line is clipped - and set background colour
ColourSelect(gstate,backcolour);
if (!PFPictureElement(gstate,xsiz*pict->xsiz,ysiz,0,posy)
{
    length=1;
    /* get colour */
    lastcolour=*linepos;

    for(xpix=pict->xsiz-1; xpix; xpix--)
    {
        /* get colour */
        colour=*linepos++;

        if (colour!=lastcolour)
        {
            if (lastcolour!=backcolour)
            {
                ColourSelect(gstate,lastcolour);
                PFPictureElement(gstate,xsiz*length,ysiz,
            }
            /* skip pixels on page */
            posx+=length;
            lastcolour=colour;
            length=1;
        } else
            length++;
    }
    /* there may be a sequence left at the end of the line
if (colour!=backcolour)
{
    ColourSelect(gstate,colour);

```

```

        PFPictureElement(gstate, xsiz*length, ysiz, xsiz*pos
    }
    }
    posy+=ysiz;
    ypix--;
    linestart+=pict->xsiz;
}
return ERR_OK;
}

```

We also need a dummy routine, for future compatibility with possible extensions of the picture drivers.

```

Error Handle(int command, ...)
{
    return ERR_NIMP;
}

```

To finish the driver, only the actual driver definition has to be written. The structure is called `init` to make sure PROforma (the external module system to be precise) knows where to find the picture driver definition.

```

PICTdriver init = {
    NULL, PF_PICTUREDIVER,
    "example 256 colour picture",
    Recognize,
    AspectRatio,
    Display,
    ColourInfo,
    ColourGet,
    Handle
};

```

makefile

Here are the lines from the makefile which allow you to build the example given above as a genuine PROforma picture driver. Note that all occurrences of "pict_example" can be replaced by any other filename.

```

pict_example_pfd : pict_example_o core-dll_o
    ${LD} -ms -opict_example_pfd \
    pict_example_o core-dll_o \
    -lsms -sxmod

```


mkxmod pict_example_pfd \"PROforma external driver\"

PROGS, Professional & Graphical Software
last edited November 11, 1996

PROforma introduction

- [What is PROforma](#)
 - [This manual](#)
 - [Present, Past and Future](#)
 - [Installation](#)
-

What is PROforma

PROforma is short for 'PROGS Font & Raster Manager', and it does exactly what this name suggest. It is a library of routines to manage and display vector graphics and fonts on (raster) devices like screens and printers.

The availability of a separate program to manage graphics and fonts has several advantages. It allows application developers to create output of equal quality (resolution permitting) on several devices, and they can share resources. In short this means that the PROforma library only has to be loaded once, independent of the number of applications which use it. Also fonts only have to be loaded once, and can be shared between applications.

PROforma was originally developed as the graphics library for LINEdesign. That does not mean that this is the only kind of application for which PROforma is of use. PROforma is perfectly suitable as well for desktop publishers, word processors, business graphics and all applications which want high quality output (which must be just about every application except compilers and games). Actually, even at the time of writing there are things which are possible with PROforma and can't be accessed through LINEdesign.

More recently PROforma has been redesigned to a great extent, to make it even more future proof, easier to extend (both internally, and by writing drivers). There have been some changes to make it easier to write a window manager (for ProWesS) and complete support of colour has been added.

As a library, PROforma has the form of dynamic link library (DLL) (if you don't know what that is, don't worry).

This manual

This manual is intended to explain in detail what PROforma is about, how it operates, how it should be used and how it can be extended. For some specific details like possible errors of the access routines, we would like to refer to the *PROforma_ddf* DATAdesign file.

We (and everybody who uses this manual) would like it very much if you could send us any comments about this manual, like

- omissions
- inaccuracies or mistakes
- typing and/or spelling mistakes
- making this manual better English
- anything else (positive comments are also always appreciated)

At the bottom of each page is mentioned when the HTML document was last modified. I will try to keep this date correct, however it is only meant to indicate changes in the information provided, I will not change that date when correcting spelling mistakes or HTML errors.

Present, Past and Future

PROforma is originally developed as the graphics library for LINEdesign. When we started developing LINEdesign v2, we felt that the graphics routines we used were too slow, and also too restrictive. On the other hand, LINEdesign v1 was quite greedy on memory. Therefore, we threw away all the old routines, and started writing a new, more powerful and faster set of routines. During this development, we even introduced some concepts (like the clipping path), which are not used in LINEdesign. On the other hand, the graphics library was expanded to allow efficient editing on screen.

So what do we have now ??

We have a system that can efficiently render and display fonts. All fonts can be shared among applications. A font cache is used to speed up the handling of fonts. Even the font cache and everything in it is shared amongst applications. Fonts are rendered using proper hinting (if the font includes the hints).

The system can draw lines and curves either stroked (with given accuracy and thickness), or filled (using either in/out or winding rule).

Anything can be displayed in any gray shade or colour. If wanted, everything can also be clipped by regular or irregular shapes. Transformation matrices can be applied on the page.

The user can define which part of the coordinate space is actually visible on page (or screen).

Bitmapped pictures can be directly displayed. This allows the user to include screens in his or her output.

Possibilities for the future ??

We want to improve the control over how colour is produced, allowing the user to define how the colour patterns are formed. We also want to make it possible to use a pattern (drawn using PROforma of course) to be used as "colour". Also, we want to add dashed lines, and some variations on line caps, line joins and maybe even some kinds calligraphic lines.

PROforma already contains three kinds of drivers : bitmap drivers, printer (or screen) drivers, and picture drivers. At this point the bitmap drivers are only used internally in PROforma, but we are thinking of making them accessible from outside PROforma, so that general bitmap graphics routines can be written using the primitive commands in the bitmap drivers.

Although PROforma already uses hinting when displaying the fonts, we would like to examine whether we can further improve the quality of hinted

fonts at very small sizes (especially when displayed on screens, e.g. in ProWesS).

Of course we continually try to improve PROforma's speed.

Installation

PROforma is a job which makes itself available to clients in the form of a Dynamic Link Library (a thing with an efficient access method).

Some extensions have to be loaded for PROforma to run : the dynamic link library manager and syslib.

PROforma has the shape of a job, and loads its configuration file ("PROforma_cfg") when it starts. A parameter can be given to PROforma to specify the path where the configuration file can be found (e.g. "win1_pf;flp1" to search on "win1_pf_" and "flp1_" in that order). If no parameter is given or the configuration file is still not found, then first the program default and then the data default devices will be searched.

The fact that PROforma has the form of a job (and not a resident extension as most libraries like the Menu Extensions), has certain advantages. Jobs can always be loaded (if you have enough memory), and jobs can always be removed. When loading a job it is possible to pass a parameter (like where to find the configuration file), which is particularly useful. Also, no memory is wasted if PROforma is loaded while a copy was already running. So if you want to release the memory which is used by PROforma, you can just remove the job. Of course the disadvantage of this scheme is that you can accidentally remove the PROforma job, which is dangerous as all programs which use PROforma will also be removed, so you could loose data that way (in fact, as ProWesS uses PROforma, all jobs which use ProWesS would also be removed).