Libnetbpm Utility Functions

Libnetpbm Utility Functions

Updated: 27 August 2006

Table Of Contents

NAME

libpm - netpbm utility functions

DESCRIPTION

These library functions are part of Netpbm.

This page documents functions in the Netpbm subroutine library that are not directly related to image data.

For introductory and general information using **libnetpbm**, see Libnetpbm User's Guide.

The most commonly used **libnetpbm** functions are those that read and write and process Netpbm images. Those are documented in Libnetpbm Netpbm Image Processing Manual

To use these services, #include **pam.h**.

Functions

Initialization

Overview

void pm_init(int ***argcP*, **char ***argv[]);**

Description

All Netpbm programs must call **pm_init()** just after startup, before they process their arguments. **pm_init()**, among other things, processes Netpbm universal arguments and removes them from the argument list.

A program that isn't a Netpbm program, but just uses **libnetpbm** services, need not invoke **pm_init**.

File Or Image Stream Access

Overview

FILE *pm_openr(char * *name***)**

FILE *pm_openw(char * *name* **);**

FILE *pm_openr_seekable(const char * *name* **);**

FILE *pm_close(FILE * *fp* **);**

void pm_tell2(FILE * *fileP*, **pm_filepos *** *fileposP*, **unsigned int** *fileposSize* **);**

unsigned int pm_tell(FILE * *fileP* **);**

void pm_seek2(FILE * *fileP*, **const pm_filepos *** *fileposP*, **unsigned int** *fileposSize* **);**

void pm_seek(FILE * *fileP*, **unsigned long** *filepos* **);**

char *pm_read_unknown_size(FILE * *fp*, **long *** *nread* **);**

Description

An image stream is just a file stream (represented in the standard C library as type **FILE ***).

These routines work on files > 2 GiB if the underlying system does, using the standard large file interface. Before Netpbm 10.15 (April 2003), though, they would fail to open any file that large or process any offset in a file that could not be represented in 32 bits.

pm_openr() opens the given file for reading, with appropriate error checking. A filename of **-** is taken to mean Standard Input. **pm_openw()** opens the given file for writing, with appropriate error checking. **pm_close()** closes the file descriptor, with appropriate error checking.

pm_openr_seekable() appears to open the file just like **pm_openr()**, but the file thus opened is guaranteed to be seekable (you can use ftell() and fseek() on it). **pm_openr_seekable()** pulls this off by copying the entire file to a temporary file and giving you the handle of the temporary file, if it has to. If the file you name is a regular file, it's already seekable so **pm_openr_seekable()** just does the same thing as **pm_openr()**. But if it is, say, a pipe, it isn't seekable. So **pm_openr_seekable()** reads the pipe until EOF into a temporary file, then opens that temporary file and returns the handle of the temporary file. The temporary file is seekable.

The file **pm_openr_seekable()** creates is one that the operating system recognizes as temporary, so when you close the file, by any means, it gets deleted.

You need a seekable file if you intend to make multiple passes through the file. The only alternative is to read the entire image into memory and work from that copy. That may use too much memory. Note that the image takes less space in the file cache than in a buffer in memory. As much as 96 times less space! Each sample is an integer in the buffer, which is usually 96 bits. In the file, a sample may be as small as 1 bit and rarely more than 8 bits.

pm_tell2() returns a handle for the current position of the image stream (file), whether it be the header or a row of the raster. Use the handle as an argument to **pm_seek2()** to reposition the file there later. The file must be seekable (which you can ensure by opening it with **pm_openr_seekable()**) or this may fail.

The file position handle is of type **pm_filepos**, which is intended to be opaque, i.e. used only with these two functions. In practice, it is a file offset and is 32 bits or 64 bits depending upon the capability of the underlying system. For maximum backward and forward compatibility, the functions that take or return a **pm_filepos** have a *fileposSize* argument for the size of the data structure. In C, simply code **sizeof(pm_filepos)** for that.

pm_seek() and **pm_tell** are for backward compatibility only. Do not use them in new code. These functions are not capable of handle positions in files whose byte offset cannot be represented in 32 bits.

pm_tell2() and **pm_seek2()** replaced **pm_tell()** and **pm_seek()** in Netpbm 10.15 (April 2003).

pm_read_unknown_size() reads an entire file or input stream of unknown size to a buffer. It allocates more memory as needed. The calling routine has to free the allocated buffer with **free()**.

pm_read_unknown_size() returns a pointer to the allocated buffer. The **nread** argument returns the number of bytes read.

Endian I/O

Entry Points

void pm_readchar(FILE * *in*, **char *** *sP* **);**

void pm_writechar(FILE * *out*, **char** *s* **);**

int pm_readbigshort(FILE * *in*, **short *** *sP* **);**

int pm_writebigshort(FILE * *out*, **short** *s* **);**

int pm_readbiglong(FILE * *in*, **long *** *lP* **);**

int pm_writebiglong(FILE * *out*, **long** *l* **);**

int pm_readlittleshort(FILE * *in*, **short *** *sP* **);**

int pm_writelittleshort(FILE * *out*, **short** *s* **);**

int pm_readlittlelong(FILE * *in*, **long *** *lP* **);**

int pm_writelittlelong(FILE * *out*, **long** *l* **);**

void pm_readcharu(FILE * *in*, **char *** *sP* **);**

void pm_writecharu(FILE * *out*, **char** *s* **);**

int pm_readbigshortu(FILE * *in*, **short *** *sP* **);**

int pm_writebigshortu(FILE * *out*, **short** *s* **);**

int pm_readbiglongu(FILE * *in*, **long *** *lP* **);**

int pm_writebiglongu(FILE * *out*, **long** *l* **);**

int pm_readlittleshortu(FILE * *in*, **short *** *sP* **);**

int pm_writelittleshortu(FILE * *out*, **short** *s* **);**

int pm_readlittlelongu(FILE * *in*, **long *** *lP* **);**

int pm_writelittlelongu(FILE * *out*, **long** *l* **);**

Description

pm_readchar(), **pm_writechar**(), **pm_readbigshort**(), **pm_writebigshort**(), **pm_readbiglong**(), **pm_writebiglong**(), **pm_readlittleshort**(), **pm_writelittleshort**(), **pm_readlittlelong**(), and **pm_writelittlelong()** are routines to read and write 1-byte, 2-byte, and 4-byte pure binary integers in either big- or little-endian byte order. Note that a "long int" C type might be wider than 4 bytes, but the "long" routines still read and write 4 bytes.

pm_readbiglongu(), etc. (names ending in **u**) are the same except they work on unsigned versions of the type.

The routines with declared return values always return 0. Before Netpbm 10.27 (March 2005), they returned -1 on failure, including EOF. Now, they issue an error message to Standard Error and abort the program if the I/O fails or encounters EOF.

The 1-byte routines were new in Netpbm 10.27 (March 2005). The unsigned versions were new somewhere around Netpbm 10.21 (2004).

Maxval Arithmetic

Entry Points

int pm_maxvaltobits(int *maxval* **);**

int pm_bitstomaxval(int *bits* **);**

unsigned int pm_lcm(unsigned int *x*, **unsigned int** *y*, **unsigned int** *z*, **unsigned int** *limit* **);**

Description

pm_maxvaltobits() and **pm_bitstomaxval()** convert between a maxval and the minimum number of bits required to hold it.

pm_lcm() computes the least common multiple of 3 integers. You also specify a limit and if the LCM would be higher than that limit, **pm_lcm()** just returns that limit.

Gamma Arithmetic

Entry Points

float pm_gamma(float *intensity* **);**

float pm_ungamma(float *brightness* **);**

Description

In graphics processing, there are two common ways of representing numerically the intensity of a pixel, or a component of a pixel.

The obvious way is with a number that is directly proportional to the light intensity (e.g. 10 means twice as many milliwatts per square centimeter as 5). There are two problems with this:

> To the human eye, a 1 milliwatt per square centimeter difference in a bright image is much less apparent than a 1 milliwatt per square centimeter difference in a dark image. So if you have a fixed number of bits in which to store the intensity value, you're wasting resolution at the bright end and skimping on it at the dark end.
>
> Monitor inputs and camera outputs aren't directly proportional to the light intensity they project or detect.

For these reasons, light intensities are often represented in graphics processing by an exponential scale. The transfer function is called a gamma function and the resulting numbers are called gamma-corrected or gamma-adjusted. There are various gamma functions. The Netpbm formats specify that intensities are represented by gamma-adjusted numbers of a particular gamma transfer function.

These functions let you convert back and forth between these two scales, using the same gamma transfer function that is specified in the Netpbm format specifications.

pm_gamma709 converts from an intensity-proportional intensity value to a gamma-adjusted intensity value (roughly proportional to brightness, which is the human subjective perception of intensity), using the ITU-R Recommendation BT.709 gamma transfer function.

pm_ungamma709 is the inverse of **pm_gamma709**.

Messages

Overview

void pm_message(char * *fmt***,**

void pm_setusermessagefn(pm_usermessagefn * *function***);**

Description

pm_message() is a **printf()** style routine to write an informational message to the Standard Error file stream. **pm_message()** suppresses the message, however, if the user specified the **-quiet** option on the command line. See the initialization functions, e.g. **pnm_init()**, for information on the **-quiet** option. Note that Netpbm programs are often used interactively, but also often used by programs. In the interactive case, it is nice to issue messages about what the program is doing, but in the program case, such messages are usually undesirable. By using **pm_message()** for all your messages, you make your program usable in both cases. Without any effort on your part, program users of your program can avoid the messages by specifying the **-quiet** option.

Netpbm distinguishes between error messages and information messages; **pm_message()** is just for informational messages. To issue an error message, see **pm_errormsg()**.

pm_setusermessagefn registers a handler for informational messages, called a user message routine. Any library function (including **pm_message()**) that wants to issue an informational message in the future will call that function with the message as an argument instead of writing the message to Standard Error.

The argument the user message routine gets is English text designed for human reading. It is just the text of the message; there is no attempt at formatting in it (so you won't see any newline or tab characters).

To capture error messages in addition to informational messages, see **pm_setusererrormsgfn()**.

You can remove the user message routine, so that the library issues future informational messages in its default way (write to Standard Error) by specifying a null pointer for *function*. Example:

```
static pm_usermessagefn logfilewrite;
static void
logfilewrite(const char * const msg) {
   fprintf(mymsglog, "Netpbm message: %s", msg);
}
pm_setusermessagefn(&logfilewrite);

pm_message("Message for the message log");
```

System Utilities

- pm_system
- pm_tmpfile

Keyword Matching

Entry Points

void pm_keymatch();

Description

This subroutine is obsolete. It used to be used for command line option processing. Today, you can do better option processing more easily with the shhopt facility. See any recent program in the Netpbm package for an example. **pm_keymatch()** does a case-insensitive match of **str** against **keyword**. **str** can be a leading substring of **keyword**, but at least **minchars** must be present.

Table Of Contents