Permuted Index

NAME

d3as – DSP32 assembler

SYNOPSIS

d3as *[options] source_files...*

DESCRIPTION

Filenames ending with **.s** or **.i** are assumed to be DSP32 assembly source files. Each specified source file is assembled, and a corresponding object file is created with a **.o** suffix. The valid options are:

-V	Print the version number and exit.
-N	Produce DSP32 object code. (Default mode)
-Q	Produce DSP32C object code.
-C	Retain comments through preprocessor (useful only with -P).
-P	Preprocess the named files and store them in corresponding files with the .i suffix.
-D*n*	Define *n* to the preprocessor with value 1.
-D*n*=*v*	Define *n*, an identifier, to the preprocessor as if by #define and give it value *v*.
-U*n*	Undefine *n* by removing any initial definition of *n*.
-I*dir*	The #include files whose names do not begin with / (\ on *MS-DOS*) should be searched for in *dir*, before looking in the directories on the standard list. Thus, #include files whose names are enclosed in " " are searched for first in the directory of the *filename* argument, then in directories named in -I options, and last in directories on a standard list. For #include files whose names are enclosed in <>, the directory of the *filename* argument is not searched.
-l*n*	(Lower-case L). Produce listing of assembly file. The *n*, if specified is the page length (default is 66 lines).
-l*file*	(Lower-case L). Produce listing of assembly file and store in *file*.l. If *file* is not specified, the source file names are used (with a .l extension).
-n	Generate parity bits for the DSP32 device. **Note:** This option has the opposite effect that it had in previous versions of the assembler.
-W	Turn off warning messages.
-F	Treat certain programming violations as warnings, rather than fatal errors. See section **3.4** of the *DSP32 and DSP32C Support Software Library DSP32 and DSP32C C Language Compiler*
-A	Do not invoke the C preprocessor.
-p	Whenever possible, translate each goto statement to a pc relative goto statement (**pcgoto**). Note that is option does not translate call statements to pc relative call statements (**pccall**).
-o *file*	Place output object file in *file*.

DIRECTIVE

The assembler supports the following directives:

.rsect section_name
This assembler directive allows the user to set up a relocatable program section. The one argument to .rsect is a legal identifier enclosed in quotes which is the name of the section.

.= The .= directive is followed by a constant expression. It sets the current section's location counter to the constant value that is on the right of the equal sign. The expression cannot be external.

.align This directive is used to assure that an instruction or data occurs on a legal boundary. It is usually used when data space is allocated. The directive has one argument, and integer constant that is used to determine that correct alignment.

.global Once an identifier is used, it is known from that point on in the file. Therefore, every identifier in a file must be unique. The identifiers are not known across file boundaries. The .global directive is followed by a list of identifiers, separated by commas, that are to be made known across file boundaries. The identifiers on the directive line must be defined in that file, but are then available to other files that are linked with it.

.extern If an identifier is listed as external, it is defined and listed as global in another file, but is known throughout the local file. The .extern directive is followed by a list of identifiers, separated by commas.

.list Turn on listing. For use with the **-l** flag.

.nolist Turn off listing. For use with the **-l** flag.

.page Skip to the top of a new page. For use with the **-l** flag.

EXAMPLES

The command

```
d3as test.s
```

will produce a file test.o which contains the relocatable object code produced by assembling test.s .

The command

```
d3as -l test.s
```

will produce an assembly listing written to the file test.l. This command also produces a relocatable object file, test.o.

NEW FEATURES

A new form for an unconditional branch instruction is supported for both the DSP32 and the DSP32C. This instruction is:

```
pcgoto label
```

The assembler will produce a pc-relative goto which can be dynamically relocated without affecting the branch. Presently, there is a restriction that the label must be within the same section as the **pcgoto** instruction using it and within the same file. These restrictions may be lifted at a later time.

SEE ALSO

DSP32 C Support Software Library User Manual
DSP32 and DSP32C Support Software Library DSP32 and DSP32C C Language Compiler
d3sim(1)
d3cc(1)
d3ld(1)

NAME

d3cc - DSP32 C language compiler

SYNOPSIS

d3cc *options source_files*

DESCRIPTION

The valid options are:

–N	Produce DSP32 object code. (Default mode)
–Q	Produce DSP32C object code.
–P	Invoke the C preprocessor only. For each *file*.**c**, this generates a *file*.**p** containing the preprocessed C source code.
–S	Invoke the preprocessor and compiler only. This generates assembly source files (**.s** extension) from C source files.
–i	Invoke the compiler and optimizer only. This generates optimized assembly files (**.i** extension) from C or assembly source files (**.c** or **.s** extension, respectively).
–c	Invoke the compiler, optimizer, and assembler only. This generates object files (**.o** extension) from C or assembly source files (**.c** or **.s** extension, respectively).
–l	Generate a listing file (**.l** extension) of assembled files. The listing is useful for assembly-level debugging.
–t *textseg*	Causes the compiler to load all the program text in the compiled files in a section called *textseg* instead of the default section **.text**.
–d *dataseg*	Causes the compiler to load all the global and static data in the compiled files in a section called *dataseg* instead of the default section **.data**.
–m *mapfile*	Specifies an alternate memory configuration file (ifile) for use by the linker. The default ifiles are **mem32.map** (for the DSP32) and **mem32c.map** (for the DSP32C) in the directory $DSP32SL/lib.
–s *startfile*	Specifies an alternate start-up file for use by the linker. The default start-up files are **crt0_32.o** (for the DSP32) and **crt0_32c.o** (for the DSP32C) in the directory $DSP32SL/lib.
–o *outfile*	Specifies the name of the output file. The default output file is **a.out**.
–l*xx*	Includes the library **lib***xx***32.a** or **lib***xx***32c.a**, depending on whether DSP32 or DSP32C code is being generated.
–W*c*,*arg1*,[*arg2* ...]]	Passes the specified argument(s) (*arg1* ...) to pass *c*, where *c* is one of {**p**, **c**, **o**, **a**, or **l**} indicating the preprocessor, compiler, optimizer, assembler, or linker, respectively.
–D*n*	Define *n* to the preprocessor with value 1.
–D*n*=*v*	Define *n*, an identifier, to the preprocessor as if by #define and give it value *v*.
–U*n*	Undefine *n* by removing any initial definition of *n*.
–I*dir*	The #include files whose names do not begin with / (\ on *MS-DOS*) should be searched for in *dir*, before looking in the directories on the standard list. Thus, #include files whose names are enclosed in " " are searched for first in the directory of the *filename* argument, then in

directories named in -I options, and last in directories on a standard list. For #include files whose names are enclosed in <>, the directory of the *filename* argument is not searched.

-n Generate parity bits for the DSP32 device. **Note:** This option has the opposite effect that it had in previous versions of the assembler.

-T Trace program execution. d3cc prints command lines used to invoke the preprocessor, compiler, optimizer, assembler, and linker. Useful for debugging problems with d3cc command strings.

SEE ALSO

DSP32 C Language Compiler User Manual **d3as(1)**
d3sim(1)
d3ld(1)

NAME

d3ld - DSP32 link editor

SYNOPSIS

d3ld *[options] [ifile] obj_files...*

DESCRIPTION

The **d3ld** command links the named *obj_files* object files, produced by **d3as** or **d3cc,** and puts the resulting object file into **a.out** unless otherwise specified. The *ifile* is an ASCII file containing directives.

The valid options are:

-a Produces an absolute, executable file; gives warnings for undefined references. Relocation information is stripped from the output file unless the -r option is given. The -r option is needed only when an absolute file should retain its relocation information (not the normal case). If neither -a nor -r is given, -a is assumed.

-f *fill* Sets the default fill pattern for "holes" within an output section as well as initialized *bss* sections. The argument *fill* is a two-byte constant.

-lx Searches a library **lib**x**.a**, where *x* is up to nine characters. A library is searched when its name is encountered, so the placement of a -l is significant. By default, libraries are located in the directory **lib** within the directory specified by the environment variable DSP32SL.

-m Produces a map or listing of the input/output sections (including holes) on the standard output.

-o *outfile* Produces an output object file by the name *outfile*. The default name of the object file is **a.out**.

-r Retains relocation entries in the output object file. Relocation entries must be saved if the output file is to become an input file in a subsequent **ld** run. The link editor does not complain about unresolved references, and the output file is not executed.

-s Strips line number entries and symbol table information from the output object file. This function can also be performed using the utility **d3strip.**

-u *symname* Enters *symname* as an undefined symbol in the symbol table. This is useful for loading entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

-x Does not preserve local (non-.global) symbols in the output symbol table; enters external and static symbols only. This option saves some space in the output file.

-L *dir* Changes the algorithm of searching for **lib**x**.a** to look in *dir* before looking in **DSP32SL/lib.** This option is effective only if it precedes the -l option on the command line.

-M Outputs a message for each multiply-defined external definition. However, if the objects loaded include debugging information, extraneous output is produced.

-N Puts the data section immediately following the text in the output file.

-V Outputs a message giving information about the version of **d3ld** being used.

-n Generate parity bits for DSP32 device. **Note:** This option has the

opposite effect that it had in previous versions of the linker.

Flags can be combined with file names on both the command line and in an *ifile*. The ordering of flags is insignificant with the exception of the **-l** flag for specifying libraries. Libraries are searched as they are encountered for any undefined external references.

SEE ALSO

DSP32 C Support Software Library User Manual
d3as(1)
d3sim(1)
d3cc(1)

NAME

d3sim - DSP32 link editor

SYNOPSIS

d3sim *[options] file*

DESCRIPTION

The **file** is the DSP32 executable program file that is being simulated. The valid options are:

-c The -c option must be used with programs that were compiled by the DSP32/DSP32C C compiler. This option allows breakpoints to operate correctly, allows the **printf** function to be used in the program, and also allows registers r14, r18, and r19 to be referred to as **sp** (stack pointer), **rp** (return pointer), and **ir** (increment register), respectively. The **ftrace** command (trace function calls) is also available.

-T Time run. The time taken to run the simulation on the host computer is displayed (in seconds) at the end of a simulation run. (This is not the time that the physical device takes to run the same program).

-e Exec file. This option causes a text file of commands to be executed. The name of the file is assumed to be file.ex, where file is the name of the program file. It is useful to produce such a file and put into it definitions of any functions that would be useful in debugging that particular program.

-m*n* Memory Mode Specification. This option specifies the memory mode. The value of *n* can be 0 through 3 for DSP32 programs or 0 through 7 for DSP32C programs. If it is not specified, the mode defaults to 2 for the DSP32 or 6 for the DSP32C. If no mode flag is present, d3sim allows writes to ROM with no complaint.

-l Log Commands. This option causes "command logging" to be turned on. If input is from a terminal, it gets written to a file called log.cmd. If input is from a file, then as it is read, it gets written to standard output. Thus, each line of output is preceded by the command line that caused it, which can be useful in identifying unexpected results.

-b Turn off breakpoint verbose mode. No message is printed when a breakpoint occurs. **Note:** This option has the opposite effect that it had in previous versions of the simulator.

-d# Development system specifier. Sets the simulator in development system mode. The # specifies the number of the DSP32 development system that is being controlled (see the *WE®DSP32-DS Development System User Manual* for details).

-D/dev/*alt_port*

Device Driver Select. ***UNIX* SYSTEM ONLY**
The /dev/*alt-port* is used if the DSP32 development system is connected to a port other than the user's tty port. The /dev/*alt_port* is the *UNIX* System device driver of the port to which the development system is connected (see the *WE®DSP32-DS Digital Signal Processor Development System User Manual* for details).

-C *file*.cfg

DSP32C Development System in use. Refer to the *WE®DSP32C Development System User Manual* for complete details.

-Z Disables "dirty-zero" checking in the DAU. By default, a dirty-zero error occurs when a number has a nonzero exponent and a mantissa of zero.

-p	Turn on profiling. The profiling feature requires a large amount of memory, which may cause problems on some systems. Therefore, it is not active by default.
-R	DSP32C mode only. Run DSP32C as a ROM device.
-P	Set default pcw value to 0. Normal default is 0x3f.
-A*n*	DSP32C mode only. Set number of wait states for external memory bank A to *n*.
-B*n*	DSP32C mode only. Set number of wait states for external memory bank B to *n*.
-w *n*	Sets the number of conflict wait states to remember to *n*. This determines the number of conflict wait states that are displayed when the **waits** command is issued.
-S #	Enable stack range checking. The argument # is an upper bound the stack pointer should not exceed.

Prior to accepting user command input, **d3sim** loads memory from the given *file* and initializes as if a chip reset has occured.

SEE ALSO

DSP32 C Support Software Library User Manual
d3as(1)
d3cc(1)
d3ld(1)

NAME

devcc – C compiler for Pixel Machine programs using DEVtools

SYNOPSIS

devcc <d3cc options> [–pixel | –pipe] <source files>

DESCRIPTION

devcc is the DSP32 compiler used with DEVtools programs. It is the same as **d3cc** but it knows about Pixel Machine specific files. In addition to the directories searched by **d3cc** for **include** files, **devcc** also searches **$HYPER_PATH/devtools/include**. **devcc** also passes the correct startup file and loader directive file (*ifile*) to **d3ld** as well as the **$HYPER_PATH/devtools/lib/libpm.a** library.

devcc takes all the options that **d3cc** does plus **–pixel** (default) and **–pipe** options.

The **–pipe** option is used to link pipe programs and causes **crt0_pipe.o** and **pipe_ifile** to be used.

The **–pixel** option is for pixel programs and causes **crt0_pixel.o** and **pixel_ifile** to be used.

NOTES

If users want to use **printf** with **d3sim**, they should include **$DSP32SL/include/printf.c** on the **devcc** command line to prevent loading the **printf** that is included in **libpm.a**.

SEE ALSO

DSP32 C Language Compiler User Manual
DSP32 Support Software Library and DSP32 C Language Compiler Version 1.3.1 Addendum

NAME

devdisp – download an image from a file to a Pixel Machine.

SYNOPSIS

devdisp [–p initx inity] [–s npixels nlines] [–o xoffset yoffset] [–b buffer] [–d] [–v] [–u] file

DESCRIPTION

devdisp is used to download an image from a file to a Pixel Machine. The *file* specified must be in DEVtools image format as specified in **DEVimage_header(4)**.

The following options are supported:

–p *initx inity* — the image download will begin at pixel *(initx, inity)*. Default is (0,0) (upper left hand corner of the screen).

–s *npixels nlines* — a rectangular section of pixels specified by *(npixels, nlines)* will be downloaded. Default is the size of the image as specified in the file.

–o *xoffset yoffset* — if specified, *xoffset* pixels and *yoffset* lines are skipped in the image file before downloading. This is used to download only a portion of the image file. Default is 0, 0.

–d — the image download will begin at the pixel specified in the image file. This option is useful when an image was saved from a specific location on the screen and the user wishes to display it at the same location. This option overrides the **–p** option.

–b *buffer* — the image will be downloaded to the specified portion of the frame buffer. Valid values for **buffer** are:

front – pixels are downloaded to the front (currently displayed) buffer (default).

back – pixels are downloaded to the back (currently non–displayed) buffer.

vram0 – pixels are downloaded to VRAM0.

vram1 – pixels are downloaded to VRAM1.

zram – pixels are downloaded to ZRAM.

–v — verbose output will be written to the standard output.

–u — print usage information.

RETURNS

The exit code will be 0 upon success, non–zero on failure.

NOTES

devdisp downloads code into the pipe and pixel nodes to perform the image download, consequently any programs that had been downloaded will be overwritten.

SEE ALSO

DEVimage_header(4)
DEVput_scan_line(3H)
devsave(1)
picdisp(1) in the PIClib *Reference Manual*
raydisp(1) in the RAYlib *Reference Manual*

NAME

devprint – a host server program for Pixel Machine code that uses the print routines

SYNOPSIS

devprint [**-d** *node* **all**] [**-g** *node* **all**] [**-u**] [**-i**] [**-n**]

DESCRIPTION

devprint is a program that runs on the host system that polls a selected set of pipe and/or pixel nodes and performs the host processing required by any *system messages* sent from the nodes, usually the messages for **PMhost_exit**, **PMsiodir** and **printf**.

The following options may be used:

-d *node* poll pixel node *node* for print messages

-dall poll all pixel nodes for print messages

-g *node*
poll pipe node *node* for print messages

-gall poll all pipe nodes for print messsages

-i print node identification information for node **printf** commands

-n causes **devprint** to poll all nodes specified, but discards all the messages except from the first pipe and pixel node specified on the command line. This is used for debugging when it is not necessary to see the output of all of the nodes, but they must be polled so they do not hang waiting for the host to read a message. Care must be taken when using this option because commands executed on the other nodes will not function properly if they expect a response from the host.

-u print command usage format

If no node specification is provided, all pipe and pixel nodes are polled for print messages.

EXAMPLES

devprint – poll and print for all nodes

devprint -n – only prints output for Pipe and Pixel node 0

devprint -gall -dall -n – only prints output for Pipe and Pixel node 0

devprint -n -g8 -d5 – prints for pipe #8 and Pixel #5

SEE ALSO

DEVpoll_nodes(3S)
PMhost_exit(3N)
PMsiodir(3X)
printf(3N)

NAME

devsave – upload an image from a Pixel Machine to a file

SYNOPSIS

devsave [–p initx inity] [–s npixels nlines] [–b buffer] [–m mode] [–v] [–u] file

DESCRIPTION

devsave is used to upload an image that exists in the frame buffer of a Pixel Machine into a file on the host computer. The image is stored in *file* in the format specified by *mode*. The uploaded file will contain an initial DEVtools header. See the **DEVimage_header(4)** manual page for a description of the Pixel Machines image header. Each pixel component (red, green, blue and alpha) consumes 8 bits and is byte aligned. *file* will be overwritten if it exists.

The following options are supported:

–p *initx inity* — the image upload will begin at pixel *(initx, inity)*. Default is (0,0) (upper left hand corner of the screen).

–s *npixels nlines* — a rectangular section of pixels specified by *(npixels, nlines)* will be uploaded. Default is the size of the screen.

–b *buffer* — the image will be uploaded from the specified portion of the frame buffer. Valid values for *buffer* are:

front – pixels are uploaded from the front (currently displayed) buffer (default).

back – pixels are uploaded from the back (currently non–displayed) buffer.

vram0 – pixels are uploaded from VRAM0.

vram1 – pixels are uploaded from VRAM1.

zram – pixels are uploaded from ZRAM.

–m *mode* — the image will be uploaded according to the format specified in *mode*. Valid values for *mode* are:

rgba – pixels are stored in red, green, blue, alpha format (default).

rgb – pixels are stored in red, green, blue format.

a – only the alpha component of the pixel is stored.

b – only the blue component of the pixel is stored.

g – only the green component of the pixel is stored.

r – only the red component of the pixel is stored.

a_mono – the alpha component of the pixel is stored, and the image header is set to mono (for later monochrome display).

b_mono – the blue component of the pixel is stored, and the image header is set to mono (for later monochrome display).

g_mono – the green component of the pixel is stored, and the image header is set to mono (for later monochrome display).

r_mono – the red component of the pixel is stored, and the image header is set to mono (for later monochrome display).

mono – pixels will be read 8 bits at a time from ZRAM only.

16 – pixels will be read 16 bits at a time from ZRAM only.

dsp – pixels will be read 32 bits at a time from ZRAM only.

ieee – DSP floats will be converted to IEEE floats in ZRAM and uploaded. After the floats are uploaded, the values in ZRAM are converted back to DSP floats.

-v verbose output will be written to the standard output.
-u print usage information.

RETURNS

The exit code will be 0 upon success, non-zero on failure.

NOTES

devsave downloads code into the pipe and pixel nodes to perform the image upload, consequently any programs that had been downloaded will be overwritten.

SEE ALSO

DEVimage_header(4)
DEVget_scan_line(3H)
devdisp(1)
picsave(1) in the PIClib *Reference Manual*
raysave(1) in the RAYlib *Reference Manual*

NAME

DEVbswapl – convert between DSP32 long integer and host long integer

SYNOPSIS

```
#include <host/devtools.h>

long DEVbswapl(number)
long   number;
```

DESCRIPTION

DEVbswapl *(byte swap long)* converts a long integer in DSP32 format to a long integer in the host format, and vice–versa. **DEVbswapl** is implemented as a macro which returns the value of *number* with the bytes in reverse order.

SEE ALSO

DEVswap_long(3S)
DEVbswapl(3S)

NAME

DEVbswaps – convert between DSP32 short integer and host short integer

SYNOPSIS

```
#include <host/devtools.h>

short DEVbswaps(number)
short    number;
```

DESCRIPTION

DEVbswaps *(byte swap short)* converts a short integer in DSP32 format to short integers in the host format and vice–versa. **DEVbswaps** is implemented as a macro which returns the value of *number* with the high and low bytes swapped.

SEE ALSO

DEVswap_short(3S)
DEVbswaps(3S)

NAME

DEVclose – closes the Pixel Machine

SYNOPSIS

void DEVclose()

DESCRIPTION

DEVclose closes the Pixel Machine designated by the environment variable HYPER_UNIT. Closing the device consists of closing the file associated with the VME device, releasing the memory blocks that were mapped to the device, and removing the lock file.

The system status file is updated to reflect any changes that may have occurred during the execution of the program since the device was opened.

NOTES

DEVexit rather than **DEVclose** is usually used. **DEVopen** and **DEVclose** are provided for users that require lower level control of the system.

SEE ALSO

DEVexit(3H)
DEVopen(3S)

NAME

DEVdsp_ieee – convert from the DSP32 floating–point format to the IEEE floating–point format

SYNOPSIS

#include <host/devtools.h>
float DEVdsp_ieee(n)
long n;

DESCRIPTION

The host and the DSP32 use different formats for floating point numbers. **DEVdsp_ieee** converts a single 32 bit floating point number in DSP32 format to the IEEE floating point format used by the host. The number to be converted is stored in the 32 bit long *n*. The contents of *n* must be in the correct host byte order. A value read from the Pixel Machine must be converted using **DEVbswapl()** or **DEVswap_long()** before calling **DEVdsp_ieee().**

RETURNS

DEVdsp_ieee returns a floating point number with the same value as the DSP32 floating point number.

SEE ALSO

DEVieee_dsp(3S)
DEVbswapl(3S)
DEVswap_short(3S)
DEVswap_long(3S)

NAME

DEVerror – generate an error message on standard error

SYNOPSIS

```
#include <host/devtools.h>
#include <host/deverror.h>

void DEVerror(msg)
char    *msg;

char DEVerror_msg[];
int DEVerrno;
```

DESCRIPTION

DEVerror is the DEVtools equivalent of the UNIX system **perror**() function. It is used to generate an error message on standard error describing the last error that occurred during a call to a DEVtools host function.

A message of the form:

msg: `error message`

is generated.

Error messages can also be formatted by user programs by accessing the global variable DEVerror_msg. User programs can check for specific errors by comparing the global variable DEVerrno with the symbolic names defined in the **deverror.h** include file.

NOTES

It is possible for some DEVtools routines to fail because of errors returned from system calls. When this occurs, **DEVerrno** contains the value DEV_ERR_SYSTEMERR, and the contents of DEVerror_msg is undefined. Therefore, user error message handlers should not display DEVerror_msg for system errors.

SEE ALSO

perror on host system

NAME

DEVexit – halts processors, closes Pixel Machine device

SYNOPSIS

void DEVexit()

DESCRIPTION

DEVexit halts the processors, closes the device associated with the Pixel Machine, and restores the default handling of signals intercepted by **DEVinit. DEVexit** should always be called before exiting any host program that uses **DEVinit.**

NOTES

DEVexit does not wait for the Pixel Machine to finish any outstanding commands. Use **DEVwait_exit** to guarantee that the pixel nodes are done.

SEE ALSO

DEVclose(3S)
DEVinit(3H)
DEVwait_exit(3H)

NAME

DEVfifo_parallel – configure a pipe board to operate in parallel mode

SYNOPSIS

```
#include <host/devtools.h>

void DEVfifo_parallel(system, fifo)
DEVpixel_system  *system;
int fifo;
```

DESCRIPTION

DEVfifo_parallel configures a pipe board to operate in parallel mode. This mode can only be used in systems with two pipe boards. A call to this function must be made for each pipe card in the system. *fifo* is the number of the pipe board whose FIFO is to be configured in parallel.

NOTES

DEVfifo_parallel is automatically called by **DEVinit** and **DEVopen** on dual parallel pipe systems as specified by the HYPER_MODEL and HYPER_PIPE environment variables.

SEE ALSO

DEVfifo_serial(3S)
DEVinit(3H)
DEVopen(3S)

NAME

DEVfifo_read – read a block of four byte values from a pipe feedback FIFO

SYNOPSIS

#include <host/devtools.h>

int DEVfifo_read(input, input_flags, buffer, nwords)
DEVulong *input;
DEVbyte *input_flags;
DEVulong *buffer;
int nwords;

DESCRIPTION

DEVfifo_read reads a block of four byte values from a pipe node feedback FIFO. This is done by copying the data from the memory mapped address of the feedback FIFO.

input is a pointer to the memory mapped area that the data is to be read from. *input_flags* is a pointer to the memory mapped location of the input flags of the pipe board.

buffer is a pointer to the location into which the data is to be read. *nwords* is the number of four byte values to be read.

NOTES

THE **DEVcread** macros should be used for most applications.

DEVfifo_read always returns zero.

DEVfifo_read cannot be used on a system without pipe boards.

SEE ALSO

DEVfifo_write(3S)
DEVwrite(3H)

NAME

DEVfifo_reset – resets all FIFOs on a pipe board

SYNOPSIS

```
#include <host/devtools.h>

void DEVfifo_reset(pixel_system,fifo)
DEVpixel_system *pixel_system;
int fifo;
```

DESCRIPTION

DEVfifo_reset resets all the FIFOs on a pipe board. *fifo* is the number of the pipe board to be reset.

NOTES

Resetting a pipe board empties all of its FIFOs.

DEVfifo_reset is automatically called by **DEVinit** and **DEVopen**.

NAME

DEVfifo_serial – configure a pipe board to operate in serial mode

SYNOPSIS

```
#include <host/devtools.h>

void DEVfifo_serial(pixel_system,fifo)
DEVpixel_system *pixel_system;
int fifo;
```

DESCRIPTION

DEVfifo_serial configures a pipe board to operate in serial mode. This mode can be used in systems with two pipe boards. A call must be made for each pipe card in the system. *fifo* is the number of the pipe card.

NOTES

DEVfifo_serial is automatically called by **DEVinit** and **DEVopen** on dual serial pipe systems as specified by the HYPER_MODEL and HYPER_PIPE environment variables.

SEE ALSO

DEVfifo_parallel(3S)

NAME

DEVfifo_write – write a block of four byte values to a pipe FIFO

SYNOPSIS

```
#include <host/devtools.h>

int DEVfifo_write(output, output_flags, buffer, nwords)
DEVulong *output;
DEVbyte *output_flags;
DEVulong *buffer;
int nwords;
```

DESCRIPTION

DEVfifo_write writes a block of four byte values to a pipe FIFO. This is done by copying the data to the memory mapped address of the FIFO.

output is a pointer to the memory mapped area that the data is to be written to. *output_flags* is a pointer to the memory mapped location of the output flags of the pipe board.

buffer is a pointer to the data to be written. *nwords* is the number of four byte values to be written.

NOTES

DEVfifo_write always returns zero.

DEVfifo_write cannot be used to write directly to the broadcast bus FIFO.

The **DEVwrite** macros provide a more efficient mechanism to write to a pipe FIFO.

SEE ALSO

DEVfifo_read(3S)
DEVwrite(3H)

NAME

DEVget_color_map - read the color tables from video controller board and returns value

SYNOPSIS

```
#include <host/devtools.h>

void DEVget_color_map(pixel_system, r, g, b)
DEVpixel_system         *pixel_system;
int                     r[DEV_VIDEO_TABLE];
int                     g[DEV_VIDEO_TABLE];
int                     b[DEV_VIDEO_TABLE];
```

DESCRIPTION

DEVget_color_map reads the color tables from the video controller board and returns the values to the caller. Each color table contains 256 entries; each entry is a 10-bit value (0-1023).

SEE ALSO

DEVput_color_map(3S)

NAME

DEVget_image_header – read the Pixel Machine image header from a file

SYNOPSIS

```
#include <stdio.h>
#include <host/devtools.h>
#include <host/devimage.h>
#include <host/deverror.h>

int DEVget_image_header(file, image_header, optional_header)
FILE                *file;
DEVimage_header     *image_header;
DEVbyte             **optional_header;
```

DESCRIPTION

DEVget_image_header reads the DEVimage_header and the optional header (if one exists) from the specified file and returns them to the caller.

file is a file descriptor obtained from a previous call to *fopen(3)*. The file must have been successfully opened for reading and the file pointer should be pointing to the beginning of the file (i.e., no previous reads have been issued). Upon return from **DEVget_image_header**, the file pointer will be set to the beginning of the pixel data (i.e., past the image and optional headers).

DEVget_image_header will read in the first DEV_IMAGE_HEADER_SIZE bytes from the file, convert them from ASCII into unsigned longs and place them into the correct locations in the structure pointed to by *image_header*. Except for the *magic* and *optional_header_size* fields, none of the information in the header is checked for validity.

If an optional header is present (image_header->optional_header_size is not 0), memory will be allocated (via *malloc(3)*) and *image_header->optional_header_size* bytes will be read. A pointer to the allocated memory will be returned in **optional_header*. If no optional header is present, **optional_header* will be set to NULL.

RETURNS

DEVget_image_header returns 0 upon success and −1 on failure. **DEVget_image_header** will set **DEVerrno** to indicate the reason for failure:

DEV_ERR_BAD_MAGIC: the magic number is not DEV_IMAGE_MAGIC.

DEV_ERR_READ_ERR: an error was returned by the *fread(3)* system call while reading either the image header or the optional header.

SEE ALSO

DEVimage_header(4)
DEVput_image_header(3S)

NAME

DEVget_pixel, DEVget_pixels – read a pixel from the frame buffer

SYNOPSIS

```
#include <host/devtools.h>

void DEVget_pixel(system, buffer, x, y, r, g, b, o)
DEVpixel_system *system;
int buffer, x, y;
short *r, *g, *b, *o;

void DEVget_pixels(system, buffer, x, y, r, g, b, o, npixl)
DEVpixel_system *system;
int buffer, x, y;
short *r, *g, *b, *o;
int npixl;
```

DESCRIPTION

DEVget_pixel reads a pixel from the frame buffer. By using this routine, a program can read the Pixel Machine frame buffer without having to deal with the details of how the frame memory is organized on different models of the system.

system is a pointer to the system description information returned by **DEVinit**. *buffer* indicates which frame buffer is to be updated (must be the value 0 or 1). *x* is the x coordinate, *y* is the y coordinate. *r*, *g*, *b*, and *o* are pointers to the locations into which the values of the red, green, blue, and overlay values from the frame buffer are to be stored.

DEVget_pixels reads a sequence of pixels for a single scan line. *npixl* is the number of pixels to be read. *r*, *g*, *b*, and *o* point to the locations into which the values of red, green, blue, and overlay values to be stored.

NOTES

DEVget_scan_line provides a more efficient and versatile way to upload images.

SEE ALSO

DEVpixel_read(3S)
DEVget_scan_line(3H)
DEVinit(3H)

NAME

DEVget_scan_line – read one or more scan lines from a frame buffer

SYNOPSIS

```
#include <host/devtools.h>
#include <host/devimage.h>

int DEVget_scan_line(system, x, y, npixl, nlines, mode, pixels)
DEVpixel_system *system;
unsigned int x, y;
unsigned int npixl, nlines, mode;
DEVbyte *pixels;
```

DESCRIPTION

DEVget_scan_line reads one or more scan lines from the frame buffer and packs the pixels into *pixels* according to the mode specified by *mode*. By using this routine, a program can read scan lines from a Pixel Machine frame buffer without having to deal with the details of how the frame memory is organized on different models of the system.

system is a pointer to the system description information returned by **DEVinit**. *x* is the starting x screen coordinate, *y* is the starting y screen coordinate.

DEVget_scan_line reads a sequence of pixels for one or more scan lines. *npixl* is the number of pixels to be read from each scan line, *nlines* scan lines will be read. *pixels* points to the location into which the pixel values will be stored.

The buffer pointed to by *pixels* must be large enough to store (*npixl* * *nlines* * pixel size) bytes, where the pixel size is determined by the *mode* argument as described below. In all cases, pixels will be stored in *pixels* in the following order: (x,y), $(x+1,y)$, ..., $(x+npixl-1,y)$, $(x,y+1)$, ..., $(x+npixl-1,y+nlines-1)$.

The *mode* argument is used to specify two independent pieces of information: how the pixels will be stored in the array pointed to by *pixels*, and which portion of Pixel Machine memory the data should be copied from. These two values are or'ed into the *mode* argument. Valid values for *mode* and their results are:

DEV_RGBA_PACKED_PIXELS: pixels will be stored in *pixels*, 4 bytes to a pixel, in the following order: red, green, blue, alpha.

DEV_RGB_PACKED_PIXELS: pixels will be stored in *pixels*, 3 bytes to a pixel, in the following order: red, green, blue.

DEV_MONO_R_PIXELS: pixels will be stored in *pixels*, 1 byte to a pixel, with the red component of the pixel actually being stored.

DEV_MONO_G_PIXELS: pixels will be stored in *pixels*, 1 byte to a pixel, with the green component of the pixel actually being stored.

DEV_MONO_B_PIXELS: pixels will be stored in *pixels*, 1 byte to a pixel, with the blue component of the pixel actually being stored.

DEV_MONO_A_PIXELS: pixels will be stored in *pixels*, 1 byte to a pixel, with the alpha (overlay) component of the pixel actually being stored.

DEV_MONO_PIXELS: pixels will be stored in *pixels*, 1 byte to a pixel. This option is only available when reading from DEV_ZRAM_BUFFER.

DEV_MONO_16_PIXELS: pixels will be stored in *pixels*, 2 bytes to a pixel. This option is only available when reading from DEV_ZRAM_BUFFER.

DEV_DSP_FLOAT_PIXELS: pixels will be stored in *pixels*, 4 bytes to a pixel. This option is only available when reading from DEV_ZRAM_BUFFER.

DEV_IEEE_FLOAT_PIXELS: DSP floating point values in ZRAM will be converted to IEEE floating point pixels in ZRAM, then uploaded 4 bytes to a pixel. When the upload operation is finished, the IEEE floats in ZRAM will be converted back to DSP floats. This double conversion can result in rounding errors. This option is only available when reading from DEV_ZRAM_BUFFER.

The following values are or'ed into the *mode* argument to specify which portion of Pixel Machine memory to upload from:

DEV_FRONT_BUFFER: Upload pixels from the front (currently displayed) portion of VRAM.

DEV_BACK_BUFFER: Upload pixels from the back (currently non-displayed) portion of VRAM.

DEV_VRAM0_BUFFER: Upload pixels from the VRAM0 portion of VRAM.

DEV_VRAM1_BUFFER: Upload pixels from the VRAM1 portion of VRAM.

DEV_ZRAM_BUFFER: Upload pixels from ZRAM.

The sizes of the above buffers vary depending on the type of Pixel Machine being used as defined in the following table:

Model	FRONT	BACK	VRAM0	VRAM1	ZRAM
916	1024x1024	1024x1024	-	-	1024x1024
920	1280x1024	1280x1024	-	-	1280x1024
932	1024x1024	1024x1024	1024x2048	1024x2048	1024x2048
940	1280x1024	1280x1024	1280x2048	1280x2048	1280x2048
964	2048x1024	2048x1024	2048x2048	2048x2048	2048x2048
964X	2048x1024	2048x1024	2048x2048	2048x2048	2048x2048

Note that when uploading from ZRAM, the number of "pixels" per scan line varies with the size of a pixel. For example, on a 964, a scan line of DEV_MONO_PIXELS is 8192 (4*2048) pixels wide, a scan line of DEV_MONO_16_PIXELS is 4096 (2*2048) pixels wide.

RETURNS

DEVget_scan_line returns 0 upon success and -1 on failure. **DEVget_scan_line** also sets **DEVerrno** and **DEVerr_msg** upon failure. If **DEVget_scan_line** fails, **DEVerrno** will be set to one of the following values:

DEV_ERR_INVPARAMETER: one or more of the parameters passed to **DEVget_scan_line** is invalid.

DEV_ERR_NORESPONSE: **DEVget_scan_line** sent a system command to the Pixel Machine to begin uploading but received no response from the pixel nodes. Typically this means that the pixel node programs did not call **PMenable()** to allow processing of the system command or the system command was not passed through the pipe nodes.

NOTES

DEVget_scan_line sends a system command to all pixel nodes to initiate uploading of the scan line. Pixel node programs must be prepared to receive this command or **DEVget_scan_line** will fail. The pixel node program should call **PMenable** with the PM_ENABLE_GET_SCAN_LINE,

PM_ENABLE_GET_VRAM or PM_ENABLE_GET_ZRAM argument during its initialization and should call **PMgetcmd** in its main processing loop. **PMgetcmd** will recognize the system command and call the appropriate routine to upload the scan line(s). In addition, the pipe node programs must make sure the system command is forwarded through each of the pipe nodes. The **PMgetop** function will transparently pass these system commands through to the pixel nodes.

DEVget_scan_line is an optimized version of **DEVget_pixels** for operations like image upload and image processing.

DEVget_scan_line will be slightly faster if the scan line starts and ends on a subscreen boundary (i.e., ((*x* % **DEVx_scale(system)**) == 0) and ((*x*+*npixl* % **DEVx_scale(system)**) == 0)).

SEE ALSO

DEVget_pixels(3S)
DEVinit(3H)
PMenable(3N)
PMgetcmd(3X)

NAME

DEVinit - opens and initializes Pixel Machine device

SYNOPSIS

#include <host/devtools.h>

DEVpixel_system *DEVinit()

DESCRIPTION

DEVinit opens and initializes the device associated with the Pixel Machine. It should always be included at the start of any DEVtools host program.

DEVinit performs the following operations: opens the device, sets global variables that can be used to access the system configuration information, halts the processors, resets the processors and configures the pipes, and sets the pixel mode register.

DEVinit handles the signals **SIGHUP**, **SIGINT**, and **SIGTERM**. If any of these signals are received, the processors are stopped, the FIFOs are reset, and the pipe is restored to its original configuration.

DEVinit returns a pointer to the system descriptor if all of the operations complete successfully. If the operation fails, it returns NULL.

SEE ALSO

DEVopen(3S)

NAME

DEVload_color_tables – reads file of gamma calibration values and sets color lookup tables

SYNOPSIS

```
#include <host/devtools.h>

void DEVload_color_tables(pixel_system, filename)
DEVpixel_system        *pixel_system;
char                   *filename;
```

DESCRIPTION

DEVload_color_tables reads a file of gamma calibration values and sets the color lookup tables appropriately. The gamma file consists of a series of lines of the format:

x.x *y.y*

Where:

x.x is a calibration level
y.y is the measured video output

DEVload_color_tables computes color table values by interpolating the input values.

NOTES

DEVload_color_tables is automatically called by **DEVinit** and **DEVopen** when the HYPER_GAMMA environment variable is set.

SEE ALSO

DEVinit(3S)
DEVopen(3S)

NAME

DEVlock - manage Pixel Machine locks

SYNOPSIS

#include <host/devtools.h>

int DEVlock(key,device)
int key;
DEVpixel_device *device;

DESCRIPTION

DEVlock is used to manage the locks for the Pixel Machine to prevent more than one user from accessing the machine at the same time. *key* designates the action desired; it must have one of the following values:

DEV_KEYLOCK_ASSIGN: assigns the device to a user

DEV_KEYLOCK_UNASSIGN: clears a previous assignment

DEV_KEYLOCK_LOCK: locks the device for a user

DEV_KEYLOCK_UNLOCK: unlocks the device

Locking and assigning are similar processes, differing only in that locking has higher precedence. Locking is used by the **hyplock** and **hypfree** commands, while assigning is used by the **DEVopen** and **DEVclose** functions. The difference in precedence levels allows a user to lock a system using the **hyplock** command, run one or more programs that use **DEVopen** and **DEVclose** and still have the system locked upon completion of the programs. This may be useful to avoid having the contents of the screen corrupted, even after the program that created the image has completed.

NAME

DEVopen, DEVopen_system – make a Pixel machine available to a user program

SYNOPSIS

#include <host/devtools.h>

DEVpixel_system *DEVopen()

DEVpixel_system *DEVopen_system(options)
int options;

DESCRIPTION

DEVopen makes a Pixel Machine available to a user program. The environment variables HYPER_UNIT, HYPER_ADDRESS, HYPER_MODEL, HYPER_PIPE HYPER_GAMMA, and HYPER_VIDEO are used to determine which machine is to be used and the configuration of the system.

If the device is already open, it is closed before **DEVopen** attempts to reopen it. **DEVopen** looks for a lock file for the device being requested. If the device is already locked, **DEVopen** returns NULL. Otherwise, a lock file is created to prevent the device from being accessed by another user.

If the open operation is successful, **DEVopen** returns a pointer to a system description block, otherwise NULL is returned.

The actual process of opening the device consists of:

creating a lock file for the desired device

opening the VME bus device associated with the Pixel Machine designated by the environment variable HYPER_UNIT

allocating a memory area that is mapped to the device that has been opened

initializing a system description block that contains the memory map addresses for each of the boards and each of the processors in the Pixel Machine

configuring the pipes based on the contents of the HYPER_PIPE environment variable

initializing the pixel mode registers on the pixel boards

configuring the video controller based on the contents of the HYPER_MODEL, HYPER_GAMMA and HYPER_VIDEO environment variables.

The following system status information is updated by **DEVopen**:

– The color tables are updated based on the HYPER_GAMMA environment variable. If HYPER_GAMMA is set and is not null, it is used as the the name of a file that contains a gamma correction table. If HYPER_GAMMA is not set or is null, a linear ramp is loaded into the color tables. If HYPER_GAMMA does not contain an absolute pathname, it is used as a filename in the **$HYPER_PATH/crts** directory. Relative pathnames are not supported.

– The video control parameters are set based on the HYPER_MODEL and HYPER_VIDEO environment variables. The HYPER_VIDEO variable contains a string that is parsed to produce a value that is passed to **DEVset_video_options()**. The string in HYPER_VIDEO must be of the format:

```
sync_source={int,ext}
```

```
sync_on_green={on,off}
```

The value after the equal sign must be one of the values listed in braces. The first value is the default; spaces in the string are ignored.

EXAMPLES

HYPER_VIDEO="sync_source=ext sync_on_green=off"

HYPER_VIDEO="sync_source = int"

NOTES

DEVinit is ordinarily used instead of **DEVopen**. **DEVopen** is provided for users who require lower level control of the Pixel Machine.

DEVopen_system is identical to **DEVopen**, with the exception that an option parameter is provided to override certain default actions described above.

options must be zero or the value DEV_OPEN_NOCONFIG. Setting the *noconfig* option causes **DEVopen_system** to suppress the steps that set the configuration of the machine. The steps omitted are:

- configuring the pipes
- initializing the pixel mode registers
- loading the color tables
- setting the video options

The *noconfig* option is used by commands like **devprint** and **hypstat** that need to access the Pixel Machine without altering the mode that the machine is running.

This function should only be used for applications that require lower level access to the machine.

SEE ALSO

DEVload_color_tables(3S)

NAME

DEVpipe_boot – load a Pixel Machine executable into specified set of pipe nodes

SYNOPSIS

```
#include <host/devtools.h>

int DEVpipe_boot(pixel_system, filename, first_node, last_node,
                 load_table, options)
DEVpixel_system      *pixel_system;
char                 *filename;
int                  first_node;
int                  last_node;
int                  load_table[];
int                  options;
```

DESCRIPTION

DEVpipe_boot determines whether the specified Pixel Machine executable file has been loaded into the Pixel Machine. If the file has not been loaded, **DEVpipe_boot** loads it into the specified set of nodes.

pixel_system is a pointer to the system structure of the system to be loaded. *first_node* and *last_node* specify the range of nodes to be loaded. Setting *first_node* to DEV_ALL causes all of the pipe nodes to be loaded. *load_table* is a pointer to an array of boolean values that indicate for each node whether or not the node should be loaded. If *first_node* <= *node* <= *last_node* and *load_table[node]* is true, then the node is loaded. The load table feature is supplied to make it possible to load the same program into an arbitrary group of nodes while only reading the executable file once. If the load table feature is not needed, a null pointer can be used as the argument.

options is used to specify certain optional processing. This value must be zero or a bitwise or of one or more of the following values:

DEV_BOOT_VERBOSE: causes a description of the actions being performed to be displayed

DEV_BOOT_FORCE: causes the file to always be loaded regardless of the contents of the system status file

DEV_BOOT_CHECK_TIME: causes the modification time of the file to be compared with the modification time of the file currently loaded into the node (if the filenames are the same). If the times are not the same, the file is reloaded.

RETURNS

DEVpipe_boot returns zero if the operation was successful, −1 if an error occurred. The following error codes can be generated by **DEVpipe_boot**:

DEV_ERR_LDFILEOPEN: the specified file could not be opened

DEV_ERR_LDFILERR: the specified file is not a valid object file

DEV_ERR_OTHER: miscellaneous error while loading the program

NAME

DEVpipe_get – read a stream of bytes from the PIR of a pipe DSP

SYNOPSIS

```
#include <host/devtools.h>

int DEVpipe_get(pixel_system, node, buffer, nbytes, timeout)
DEVpixel_system *pixel_system;
int node;
DEVbyte *buffer;
int nbytes;
int timeout;
```

DESCRIPTION

DEVpipe_get reads a stream of bytes from the PIR of a pipe DSP. This function differs from **DEVpipe_read** in that it requires a program running on the DSP to load data into the PIR register. The implementation differs in that:

it does not use DMA

the address from which the data is to be read cannot be supplied

a *timeout* parameter must be supplied

pixel_system is a pointer to the system descriptor, *node* is the number of the pipe node from which the data is to be read. *buffer* points to the location into which the data is to be read. *nbytes* is the number of bytes of data to be read. *nbytes* should always be an even number. If *nbytes* is odd, *nbytes*+1 bytes of data will be read. *timeout* contains the number of times, for each two bytes transferred, that the PCR register is to be tested to see if the data has been sent successfully.

No byte order translation is performed. The data read will be in the same byte order as it is in the DSP memory.

As a result of this operation, the parallel communications modes are altered to set the interrupt vector to 16–bit mode.

NOTES

DEVpipe_get returns the number of characters read.

The *timeout* parameter contains the number of loop iterations to be attempted before giving up. Because the execution rate depends on the system load, this could yield different results under different system load conditions. Also, because there is no **sleep** involved, the host process could consume a great deal of CPU time if the delay for each character is significant.

NAME

DEVpipe_get_msg – read a message from the PIR of a pipe DSP

SYNOPSIS

```
#include <host/devtools.h>

int DEVpipe_get_msg(pixel_system, node, buffer, nbytes, swap)
DEVpixel_system *pixel_system;
int node;
DEVbyte *buffer;
int nbytes;
int swap;
```

DESCRIPTION

DEVpipe_get_msg reads a message from the PIR of a pipe DSP. This function is similar to **DEVpipe_get** with the following exceptions:

a timeout parameter is not supplied

a byte swapping parameter is provided to allow mapping of DSP values into host values

Like **DEVpipe_get**, **DEVpipe_get_msg** does not use DMA and requires that a program running on the DSP load the data into the PIR.

pixel_system is a pointer to the system descriptor, *node* is the number of the pipe node from which the data is to be read. *buffer* points to the location into which the data is to be read. *nbytes* is the number of bytes of data to be read. *nbytes* should always be an even number. If *nbytes* is odd, *nbytes*+1 bytes of data will be read. *swap* must be one of the following values:

DEV_SWAP_NONE – no byte order conversion

DEV_SWAP_SHORT – the buffer is treated as a collection of 2-byte values and the bytes are ordered as required

DEV_SWAP_LONG – the buffer is treated as a collection of 4-byte values and the bytes are ordered as required.

If *swap* is DEV_SWAP_LONG, *nbytes* should be a multiple of 4, because a multiple of 4 bytes will always be read.

As a result of this operation, the parallel communications modes are altered to set the interrupt vector to 16-bit mode.

NOTES

DEVpipe_get_msg returns the number of characters read.

This routine will hang and use a lot of CPU time if the process on the DSP does not load the expected data into the PIR.

NAME

DEVpipe_get_pir - read the PIR register of a pipe DSP

SYNOPSIS

```
#include <host/devtools.h>

DEVushort DEVpipe_get_pir(pixel_system,node)
DEVpixel_system *pixel_system
int node;
```

DESCRIPTION

DEVpipe_get_pir reads the PIR register of a pipe DSP. This function is a special version of **DEVpipe_get_msg** that always fetches two bytes without adjusting the byte order.

Like **DEVpipe_get_msg**, **DEVpipe_get_pir** does not use DMA and it requires that a program running on the DSP load the PIR.

pixel_system is a pointer to the system descriptor, *node* is the number of the pipe node from which the data is to be read.

As a result of this operation, the parallel communications modes are altered to set the interrupt vector to 16-bit mode.

DEVpipe_get_pir returns the contents of the DSP's PIR register as an unsigned short integer.

NOTES

This routine will hang and use a lot of CPU time if the process on the DSP does not load the expected data into the PIR.

NAME

DEVpixel_halt – halt a pixel node processor

SYNOPSIS

```
#include <host/devtools.h>

int DEVpixel_halt(pixel_system,node)
DEVpixel_system *pixel_system;
int node;
```

DESCRIPTION

DEVpixel_halt halts a pixel node processor. After the processor has halted, the parallel communications modes are altered to:

enable interrupts

enable DMA

set PAR to be autoincremented on DMA

set the interrupt vector to 16-bit mode

RETURNS

DEVpixel_halt returns DEV_ERR_OK if the operations succeeds, DEV_ERR_FAIL otherwise.

NAME

DEVpixel_id_check – check status of node's ID

SYNOPSIS

```
#include <host/devtools.h>
#include <host/crt0.h>

int DEVpixel_id_check(system,node,id)
DEVpixel_system *system;
int node;
DEVcrt0_id *id;
```

DESCRIPTION

DEVpixel_id_check is used to check whether a node's ID has been corrupted.

system is a pointer to the system description information returned by **DEVopen**. *node* is the number of the node to which the ID is to be written, and is also used as a node identification number.

DEVpixel_id_check uses the parameter *id* to return the node ID information to the caller. *id* is a pointer to a node identification block.

RETURNS

This function returns DEV_ERR_OK if the operation is successful, otherwise an error value is returned. The possible error values are:

DEVERR_ID: Node ID information is invalid

DEVERR_NODE: Node number is invalid

SEE ALSO

DEVpixel_read(3S)
DEVpixel_write(3S)

NAME

DEVpixel_id_print – read and print the node ID of a processor

SYNOPSIS

```
#include <host/devtools.h>
#include <host/crt0.h>

int DEVpixel_id_print(system,node,id)
DEVpixel_system *system;
int node;
DEVcrt0_id *id;
```

DESCRIPTION

DEVpixel_id_print reads and prints the node ID of a processor's memory, and the node status information from the system status file and displays the information on standard output. **DEVpixel_id_print** reads the node ID from a processor and displays the information on standard output.

system is a pointer to the system descriptor. *node* is the number of the node to which the ID is to be written and is also used as a node identification number.

The checksum information in the node is compared with the value stored in the system status file on the host. If the checksum values do not match the message Node checksum does not match is printed beneath the program name.

This function returns DEV_ERR_OK if the operation is successful, otherwise an error value is returned. The possible error values are:

DEV_ERR_ID: Node ID information is invalid

DEV_ERR_NODE: Node number is invalid

EXAMPLE

Pixel node 0 identification data:

```
node id:       0
crt0 format: DEVtools
x nodes:       5
y nodes:       4
x offset:      0
y offset:      0
program:      /usr/xyz/prog.dsp
semaphore:     0
```

SEE ALSO

DEVpixel_write(3S)
DEVpixel_read(3S)

NAME

DEVpipe_put – write a block of data to a pipe DSP's PDR register

SYNOPSIS

```
#include <host/devtools.h>

int DEVpipe_put(pixel_system, node, buffer, nbytes, timeout)
DEVpixel_system *pixel_system;
int node;
DEVbyte *buffer;
int nbytes;
int timeout;
```

DESCRIPTION

DEVpipe_put writes a block of data to a pipe DSP's PDR register. This function differs from **DEVpipe_write** in that it requires a program running on the DSP to read data from the PDR register and store it in the appropriate memory location. The implementation differs in that:

- it does not use DMA
- the address to which data is to be sent is not supplied
- a *timeout* parameter must be supplied

pixel_system is a pointer to the system descriptor, *node* is the number of the pipe node from which the data is to be written. *buffer* points to the data to be sent. *nbytes* is the number of bytes of data to be written. *nbytes* should always be an even number. If *nbytes* is odd, *nbytes*+1 bytes of data will be written. *timeout* contains the number of times, for each two bytes transferred, that the PCR register is to be tested to see if the data has been sent successfully.

No byte order translation is performed. The data sent will be in the same byte order as it is in *buffer*.

As a result of this operation, the parallel communications modes are altered to:

- disable DMA
- set PAR to not be autoincremented on DMA
- set the interrupt vector to 16-bit mode.

DEVpipe_put returns the number of characters written.

NOTES

The *timeout* parameter contains the number of loop iterations to be attempted before giving up. Because the execution rate depends on the system load, this could yield different results under different system load conditions. Also, because there is no **sleep** involved, the host process could consume a great deal of CPU time if the delay for each character is significant.

NAME

DEVpipe_run – begin execution of programs loaded into specified pipe nodes

SYNOPSIS

```
#include <host/devtools.h>

void DEVpipe_run(pixel_system, first_node, last_node, options)
DEVpixel_system      *pixel_system;
int                  first_node;
int                  last_node;
int                  options;
```

DESCRIPTION

DEVpipe_run begins execution of the programs loaded into the specified pipe nodes.

pixel_system is a pointer to the system structure of the system whose node is to be started. *first_node* and *last_node* specify the range of nodes.

options is used to specify certain optional processing. This value should be zero or the value DEV_RUN_VERBOSE, which causes **DEVpipe_run** to provide additional information.

RETURNS

DEVpipe_run returns zero if execution was started successfully, −1 if an error occurred. The following error code can be generated by **DEVpipe_run**:

DEV_ERR_STARTERR: the program loaded in the node could not be started

NOTES

DEVrun can be used to begin execution on all pipe and pixel nodes.

SEE ALSO

DEVrun(3H)

NAME

DEVpipe_write – write a buffer to a pipe DSP

SYNOPSIS

```
#include <host/devtools.h>

int DEVpipe_write(pixel_system, node, addr, buffer, nbytes)
DEVpixel_system *pixel_system;
int node;
DEVushort addr;
DEVbyte *buffer;
int nbytes;
```

DESCRIPTION

DEVpipe_write writes a buffer to a pipe DSP. The data is transferred using parallel DMA.

pixel_system is a pointer to the system descriptor, *node* is the number of the pipe node from which the data is to be written. *addr* is the location in the DSP address space to which the data is to be sent. *buffer* points to the data to be sent. *nbytes* is the number of bytes of data to be written. *nbytes* should always be an even number. If *nbytes* is odd, *nbytes*+1 bytes of data will be written.

The data sent will be in the same byte order as it is in *buffer*. No byte order translation is performed.

DEVpipe_write uses parallel DMA I/O to transfer the data. As a result, the parallel control register is updated by this routine. The parallel communications modes are altered to:

enable DMA

set PAR to be autoincremented on DMA

set the interrupt vector to 16 bit mode

RETURNS

DEVpipe_write should always return zero.

If *nbytes* is odd, **DEVpipe_write** will write *nbytes*+1 bytes of data and return -1 as its return value. The return value should probably be the number of bytes written, not zero.

NAME

DEVpixel_boot – load a Pixel Machine executable into specified set of pixel nodes

SYNOPSIS

```
#include <host/devtools.h>

int DEVpixel_boot(pixel_system, filename, first_node, last_node,
                  load_table, options)
DEVpixel_system     *pixel_system;
char                *filename;
int                 first_node;
int                 last_node;
int                 load_table[];
int                 options;
```

DESCRIPTION

DEVpixel_boot determines whether the specified Pixel Machine executable file has been loaded into the Pixel Machine. If the file has not been loaded, **DEVpixel_boot** loads it into the specified set of nodes.

pixel_system is a pointer to the system structure of the system to be loaded. *first_node* and *last_node* specify the range of nodes to be loaded. If *first_node* is set to DEV_ALL, all pixel nodes will be loaded. *load_table* is a pointer to an array of boolean values that indicate for each node whether or not the node should be loaded. If *first_node* <= *node* <= *last_node* and *load_table[node]* is true, then the node is loaded. The load table feature is supplied to make it possible to load the same program into an arbitrary group of nodes, while only reading the executable file once. If the load table feature is not needed, a null pointer can be used as the argument.

options is used to specify certain optional processing. This value must be zero or a bitwise or of one or more of the following values:

DEV_BOOT_VERBOSE: causes a description of the actions being performed to be displayed

DEV_BOOT_FORCE: causes the file to always be loaded regardless of the contents of the system status file

DEV_BOOT_CHECK_TIME: causes the modification time of the file to be compared with the modification time of the file currently loaded into the node (if the filenames are the same). If the times are not the same, the file is reloaded.

RETURNS

DEVpixel_boot returns zero if the operation was successful, −1 if an error occurred. The following error codes can be generated by **DEVpixel_boot**:

DEV_ERR_LDFILEOPEN: the specified file could not be opened

DEV_ERR_LDFILERR: the specified file is not a valid object file

DEV_ERR_OTHER: miscellaneous error while loading the program

NAME

DEVpixel_buffer - selects the frame buffer to be displayed

SYNOPSIS

```
#include <host/devtools.h>
#include <host/pixel.h>

void DEVpixel_buffer(system,buffer)
DEVpixel_system *system;
DEVushort buffer;
```

DESCRIPTION

DEVpixel_buffer selects the frame buffer to be displayed.

system is a pointer to the system description information returned by **DEVopen**. *buffer* indicates which frame buffer is to be displayed, and must be one of the following values:

DEV_VBUF0:	Display frame buffer 0
DEV_VBUF1:	Display frame buffer 1

NOTES

Because this function updates the pixel node flag registers, it should only be used when the Pixel Machine is halted.

NAME

DEVpixel_get – read a stream of bytes from a pixel DSP's PIR register

SYNOPSIS

```
#include <host/devtools.h>

int DEVpixel_get(pixel_system, node, buffer, nbytes, timeout)
DEVpixel_system *pixel_system;
int node;
DEVbyte *buffer;
int nbytes;
int timeout;
```

DESCRIPTION

DEVpixel_get reads a stream of bytes from a pixel DSP's PIR register. This function differs from **DEVpixel_read** in that it requires a program running on the DSP to load data into the PIR register. The implementation differs in that:

- it does not use DMA
- the address from which the data is to be read cannot be supplied
- a *timeout* parameter must be supplied

pixel_system is a pointer to the system descriptor, *node* is the number of the pixel node from which the data is to be read. *buffer* points to the location into which the data is to be read. *nbytes* is the number of bytes of data to be read. *nbytes* should always be an even number. If *nbytes* is odd, *nbytes*+1 bytes of data will be read. *timeout* contains the number of times, for each two bytes transferred, that the PCR register is to be tested to see if the data has been sent successfully.

No byte order translation is performed. The data read will be in the same byte order as it is in the DSP memory.

As a result of this operation the parallel communications modes are altered to set the interrupt vector to 16-bit mode.

RETURNS

DEVpixel_get returns the number of characters read.

NOTES

The *timeout* parameter contains the number of loop iterations to be attempted before giving up. Because the execution rate depends on the system load, this could yield different results under different system load conditions. Also, because there is no **sleep** involved, the host process could consume a great deal of CPU time if the delay for each character is significant.

NAME

DEVpixel_get_msg – read a message from a pixel DSP's PIR register

SYNOPSIS

```
#include <host/devtools.h>

int DEVpixel_get_msg(pixel_system, node, buffer, nbytes, swap)
DEVpixel_system *pixel_system;
int node;
DEVbyte *buffer;
int nbytes;
int swap;
```

DESCRIPTION

DEVpixel_get_msg reads a message from a pixel DSP's PIR register. This function is similar to **DEVpixel_get** with the following exceptions:

a timeout parameter is not supplied

a byte swapping parameter is provided to allow mapping of DSP values into host values

Like **DEVpixel_get**, **DEVpixel_get_msg** does not use DMA and requires that a program running on the DSP load the data into the PIR.

pixel_system is a pointer to the system descriptor, *node* is the number of the pixel node from which the data is to be read. *buffer* points to the location into which the data is to be read. *nbytes* is the number of bytes of data to be read, and it should always be an even number. If *nbytes* is odd, *nbytes*+1 bytes of data will be read. *swap* must be one of the following values:

DEV_SWAP_NONE:
no byte order conversion

DEV_SWAP_SHORT:
the buffer is treated as a collection of 2-byte values and the bytes are ordered as required

DEV_SWAP_LONG:
the buffer is treated as a collection of 4-byte values and the bytes are ordered as required.

If *swap* is DEV_SWAP_LONG, *nbytes* should be a multiple of 4 because a multiple of 4 bytes will always be read.

As a result of this operation, the parallel communications modes are altered to set the interrupt vector to 16-bit mode.

RETURNS

DEVpixel_get_msg returns the number of characters read.

NOTES

This routine will hang and use a lot of CPU time if the process on the DSP does not load the expected data into the PIR.

NAME

DEVpixel_get_pir - read the PIR register of a pixel DSP

SYNOPSIS

```
#include <host/devtools.h>

DEVushort DEVpixel_get_pir(pixel_system,node)
DEVpixel_system *pixel_system;
int node;
```

DESCRIPTION

DEVpixel_get_pir reads the PIR register of a pixel DSP. This function is a special version of **DEVpixel_get_msg**, and it always fetches two bytes without adjusting the byte order.

Like **DEVpixel_get_msg**, **DEVpixel_get_pir** does not use DMA and it requires that a program running on the DSP load the PIR.

pixel_system is a pointer to the system descriptor, *node* is the number of the pixel node from which the data is to be read.

As a result of this operation, the parallel communications modes are altered to set the interrupt vector to 16-bit mode.

DEVpixel_get_pir returns the contents of the DSP's PIR register as an unsigned short integer.

NOTES

This routine will hang and use a lot of CPU time if the process on the DSP does not load the expected data into the PIR.

NAME

DEVpipe_halt – halt a pipe node processor

SYNOPSIS

```
#include <host/devtools.h>

int DEVpipe_halt(pixel_system,node)
DEVpixel_system *pixel_system;
int node;
```

DESCRIPTION

DEVpipe_halt halts a pipe node processor. After the processor has halted, the parallel communications modes are altered to:

enable interrupts

enable DMA

set PAR to be autoincremented on DMA

set the interrupt vector to 16-bit mode

NOTES

DEVpipe_halt returns DEV_ERR_OK if the operations succeeds, DEV_ERR_FAIL otherwise.

NAME

DEVpipe_id_check – check status of node's ID

SYNOPSIS

```
#include <host/devtools.h>
#include <host/crt0.h>

int DEVpipe_id_check(system,node,id)
DEVpixel_system *system;
int node;
DEVcrt0_id *id;
```

DESCRIPTION

DEVpipe_id_check is used to check whether a node's ID has been corrupted.

system is a pointer to the system description information returned by **DEVopen**. *node* is the number of the node to which the ID is to be written and is also used as a node identification number.

DEVpipe_id_check uses the parameter *id* to return the node ID information to the caller. *id* is a pointer to a node identification block.

This function returns DEV_ERR_OK if the operation is successful, otherwise an error value is returned. The possible error values are:

DEVERR_ID: Node ID information is invalid

DEVERR_NODE: Node number is invalid

SEE ALSO

DEVpipe_write(3S)
DEVpipe_read(3S)

NAME

DEVpipe_id_print – read and print the node ID of a processor

SYNOPSIS

```
#include <host/devtools.h>
#include <host/crt0.h>

int DEVpipe_id_print(system,node,id)
DEVpixel_system *system;
int node;
DEVcrt0_id *id;
```

DESCRIPTION

DEVpipe_id_print reads the node ID from the processor's memory and the node status information from the system status file, and displays the information on standard output. **DEVpipe_id_print** reads the node ID from a processor and displays the information on standard output.

system is a pointer to the system descriptor. *node* is the number of the node to which the ID is to be written and is also used as a node identification number.

The checksum information in the node is compared with the value stored in the system status file on the host. If the checksum values do not match, the message Node checksum does not match is printed beneath the program name.

This function return DEV_ERR_OK if the operation is successful, otherwise an error value is returned. The possible error values are:

DEV_ERR_ID: Node ID information is invalid

DEV_ERR_NODE: Node number is invalid

EXAMPLE

Pipe node 0 identification data:

```
node id:      0
crt0 format: DEVtools
x nodes:      5
y nodes:      4
x offset:     0
y offset:     0
program:     /usr/xyz/prog.dsp
semaphore:   0
```

SEE ALSO

DEVpipe_write(3S)
DEVpipe_read(3S)

NAME

DEVpixel_id_write – write a node id block to a reserved location in a pixel node DSP's memory

SYNOPSIS

```
#include <host/devtools.h>
#include <host/crt0.h>

int DEVpixel_id_write(system,node,name)
DEVpixel_system *system;
int node;
char *name;
```

DESCRIPTION

DEVpixel_id_write writes a node identification block to a reserved location in pixel node memory. The memory used to hold the node ID is allocated by the routine crt0, therefore pixel_crt0.o must be linked as part of the executable code running on the processor in order to use **DEVpixel_id_write**.

system is a pointer to the system description information returned by **DEVopen**. *node* is the number of the node to which the ID is to be written, and is also used as a node identification number. *name* is a pointer to the name that is to be assigned to the node.

RETURNS

This function returns DEV_ERR_OK if the operation is successful, otherwise an error value is returned. The possible error values are:

DEVERR_ID: Node ID information is invalid

DEVERR_NODE: Node number is invalid

SEE ALSO

DEVpixel_write(3S)

NAME

DEVpixel_mode_init - initialize pixel board mode register

SYNOPSIS

#include <host/devtools.h>

void DEVpixel_mode_init(system,omode)
DEVpixel_system *system;
DEVpixel_modereg omode;

DESCRIPTION

DEVpixel_mode_init sets overlay mode and initializes the gate bits, video shift rate, and the serial I/O connector selection fields in the pixel mode register on each pixel node board. A copy of the pixel mode register is maintained on the host because the board's pixel mode register cannot be read. As a result, **DEVpixel_mode_init** must be called during the initialization process, otherwise when a call is made that updates the pixel mode register (**DEVserial_direction** for example), it will load the register with an uninitialized value.

system is a pointer to the system description information returned by **DEVopen**. *omode* must contain one of the following values:

DEV_OVERLAY_OFF: Uses the values in rgb

DEV_OVERLAY_ON: If any overlay bit is on, the overlay value is used for the red, green, and blue values. If all of the overlay bits are on, the inverse of rgb is used.

DEV_OVERLAY_FORCE: The overlay value is always used.

DEV_OVERLAY_MASK: If overlay bit 7 is on, the overlay value is used for red, green, and blue; otherwise rgb is used.

DEVpixel_mode_init initializes the other components of the pixel mode register to default values. The defaults are:

Mode Bits - DEV_GATES_SYNC | DEV_GATES_FIFO

Video shift rate - Appropriate value based on the system type

Serial I/O - Serial I/O connector/direction zero selected

NOTES

In order for overlaying to be performed, the overlay flags must be set in both pixel node boards' pixel mode registers and in the individual processor's flag registers.

SEE ALSO

DEVpixel_overlay(3S)
PMoverlay(3X)

NAME

DEVpixel_mode_overlay - set overlay mode in the pixel mode register

SYNOPSIS

#include <host/devtools.h>

void DEVpixel_mode_overlay(system,omode)
DEVpixel_system *system;
DEVpixel_modereg omode;

DESCRIPTION

DEVpixel_mode_overlay sets the overlay mode in the pixel mode register of each pixel node board. Other fields of the pixel mode register are not affected.

system is a pointer to the system description information returned by **DEVopen.** *omode* must contain one of the following values:

DEV_OVERLAY_OFF: Uses the values in rgb

DEV_OVERLAY_ON: If any overlay bit is on, the overlay value is used for the red, green, and blue values. If all of the overlay bits are on, the inverse of rgb is used.

DEV_OVERLAY_FORCE: The overlay value is always used.

DEV_OVERLAY_MASK: If overlay bit 7 is on, the overlay value is used for red, green, and blue; otherwise rgb is used.

NOTES

In order for overlaying to be performed, the overlay flags must be set in both the pixel node boards' pixel mode register and in the individual processor's flag registers.

SEE ALSO

DEVpixel_overlay(3S)
PMoverlay(3X)

NAME

DEVpixel_overlay – update overlay mode in all pixel processor's flag registers

SYNOPSIS

```
#include <host/devtools.h>
#include <host/pixel.h>

void DEVpixel_overlay(system,mode)
DEVpixel_system *system;
DEVushort mode;
```

DESCRIPTION

DEVpixel_overlay updates the overlay mode associated with each of the individual pixel processor's flag registers. The overlay mode must be set both in the pixel node board's pixel mode register and for the individual processors.

system is a pointer to the system description information returned by **DEVopen**. *mode* contains the new contents of the overlay flag and must be one of the following values:

DEV_OVERLAY: Set the overlay flag

0 (zero): Clear the overlay flag

NOTES

Because this function updates the pixel node flag registers, it should only be used when the Pixel Machine is halted.

The **PMoverlay** function should be used to set the overlay mode during execution.

SEE ALSO

PMoverlay(3X)

NAME

DEVpixel_put – send a block of data to a pixel DSP's PDR register

SYNOPSIS

#include <host/devtools.h>

int DEVpixel_put(pixel_system, node, buffer, nbytes, timeout)
DEVpixel_system *pixel_system;
int node;
DEVbyte *buffer;
int nbytes;
int timeout;

DESCRIPTION

DEVpixel_put sends a block of data to a pixel DSP's PDR register. This function differs from **DEVpixel_write** in that it requires that a program running on the DSP read data from the PDR register and store it in the appropriate memory location. The implementation differs in that:

it does not use DMA

the address to which data is to be sent is not supplied

a *timeout* parameter must be supplied

pixel_system is a pointer to the system descriptor, *node* is the number of the pixel node from which the data is to be written. *buffer* points to the data to be sent. *nbytes* is the number of bytes of data to be written. *nbytes* should always be an even number. If *nbytes* is odd, *nbytes*+1 bytes of data will be written. *timeout* contains the number of times, for each two bytes transferred, that the PCR register is to be tested to see if the data has been sent successfully.

No byte order translation is performed. The data sent will be in the same byte order as it is in *buffer*.

As a result of this operation, the parallel communication modes are altered to:

disable DMA

set PAR to not be autoincremented on DMA

set the interrupt vector to 16-bit mode

RETURNS

DEVpixel_put returns the number of characters written.

NOTES

The *timeout* parameter contains the number of loop iterations to be attempted before giving up. Because the execution rate depends on the system load, this could yield different results under different system loadconditions. Also, because there is no **sleep** involved, the host process could consume a great deal of CPU time if the delay for each character is significant.

NAME

DEVpixel_read – read a block of memory from a pixel DSP

SYNOPSIS

```
#include <host/devtools.h>

int DEVpixel_read(pixel_system, node, addr, buffer, nbytes)
DEVpixel_system *pixel_system;
int node;
DEVushort addr;
DEVbyte *buffer;
int nbytes;
```

DESCRIPTION

DEVpixel_read reads a block of memory from a pixel DSP. The data is retrieved from DSP memory using parallel DMA.

pixel_system points to the system descriptor, *node* is the number of the pixel node from which the data is to be read. *addr* is the location in the DSP address space that contains the data to be read. *addr* must be an even memory location, aligned on a 16-bit word boundary. *buffer* points to the location into which the data is to be read. *nbytes* is the number of bytes of data to be read. *nbytes* should always be an even number. If *nbytes* is odd, *nbytes*+1 bytes of data will be read.

No byte order translation is performed. The data read will be in the same byte order as it is in the DSP memory.

DEVpixel_read uses parallel DMA I/O to transfer the data. As a result, the parallel control register is updated by this routine. The parallel communications modes are altered to:

enable DMA

set PAR to be autoincremented on DMA

set the interrupt vector to 16-bit mode

RETURNS

DEVpixel_read should always return zero.

If *nbytes* is odd, **DEVpixel_read** reads *nbytes*+1 bytes of data and returns −1 as its return value. The return value should be the number of bytes written, not zero.

SEE ALSO

DEVpixel_get(3S)

NAME

DEVpixel_run – begin execution of programs loaded into specified pixel nodes

SYNOPSIS

```
#include <host/devtools.h>

void DEVpixel_run(pixel_system, first_node, last_node, options)
DEVpixel_system     *pixel_system;
int                 first_node;
int                 last_node;
int                 options;
```

DESCRIPTION

DEVpixel_run begins execution of the programs loaded into the specified pixel nodes.

pixel_system is a pointer to the system structure of the system whose node is to be started. *first_node* and *last_node* specify the range of nodes.

options is used to specify certain optional processing. This value should be zero or the value DEV_RUN_VERBOSE, which will cause **DEVpixel_run** to provide additional information.

RETURNS

DEVpixel_run returns zero if execution started successfully, −1 if an error occurred. The following error code can be generated by **DEVpixel_run**:

DEV_ERR_STARTERR: the program loaded in the node could not be started

NOTES

DEVrun can be used to begin execution on all pipe and pixel nodes.

NAME

DEVpixel_system DEVpipe_nodes, DEVlast_pipe, DEVpixel_nodes, DEVlast_pixel, DEVx_nodes, DEVy_nodes, DEVx_scale, DEVy_scale, DEVx_screen, DEVy_screen, DEVmodel_code, DEVvideo_code, DEVpipe_code – macros used to fetch system description information from the system descriptor

SYNOPSIS

```
#include <host/devtools.h>

int DEVpipe_nodes(pixel_system)
DEVpixel_system *pixel_system;

int DEVlast_pipe(pixel_system)
DEVpixel_system *pixel_system;

int DEVpixel_nodes(pixel_system)
DEVpixel_system *pixel_system;

int DEVlast_pixel(pixel_system)
DEVpixel_system *pixel_system;

int DEVx_nodes(pixel_system)
DEVpixel_system *pixel_system;

int DEVy_nodes(pixel_system)
DEVpixel_system *pixel_system;

int DEVx_scale(pixel_system)
DEVpixel_system *pixel_system;

int DEVy_scale(pixel_system)
DEVpixel_system *pixel_system;

int DEVx_screen(pixel_system)
DEVpixel_system *pixel_system;

int DEVy_screen(pixel_system)
DEVpixel_system *pixel_system;

DEVushort DEVmodel_code(pixel_system)
DEVpixel_system *pixel_system;

DEVushort DEVvideo_code(pixel_system)
DEVpixel_system *pixel_system;

DEVushort DEVpipe_code(pixel_system)
DEVpixel_system *pixel_system;
```

DESCRIPTION

These macros are used to fetch system description information from the system descriptor. These macros should always be used to access this information. *Direct use of the fields of the system structure is unsupported.*

The following describes the value returned by each macro:

DEVpipe_nodes: the number of pipe node processors (0, 9, or 18).

DEVlast_pipe: the number of the last pipe node. Useful for calling routines such as DEVpoll_nodes.

DEVpixel_nodes: the number of pixel node processors (16, 20, 32, 40, or 64).

DEVlast_pixel: the number of the last pixel node. Useful for calling routines such as DEVpoll_nodes.

DEVx_nodes: the number of nodes in the X dimension (4, 5, 8, or 10)

DEVy_nodes: the number of nodes in the Y dimension (4 or 8)

DEVx_scale: the number of virtual nodes in the X dimension (8, or 10)

DEVy_scale: the number of virtual nodes in the Y dimension (8)

DEVx_screen: the screen width in pixels

DEVy_screen: the screen hight in pixels

DEVmodel_code: the system model

- DEV_MODEL_916
- DEV_MODEL_920
- DEV_MODEL_932
- DEV_MODEL_940
- DEV_MODEL_964
- DEV_MODEL_964X

DEVvideo_code – the video mode in use

- DEV_MODEL_VIDEO_HIRES
- DEV_MODEL_VIDEO_NTSC
- DEV_MODEL_VIDEO_PAL

DEVpipe_code – the pipe mode in use

- DEV_MODEL_PIPE_SINGLE
- DEV_MODEL_PIPE_PARALLEL
- DEV_MODEL_PIPE_SERIAL
- DEV_MODEL_PIPE_NONE

NAME

DEVpixel_write – write a buffer to a pixel DSP

SYNOPSIS

```
#include <host/devtools.h>

int DEVpixel_write(pixel_system, node, addr, buffer, nbytes)
DEVpixel_system *pixel_system;
int node;
DEVushort addr;
DEVbyte *buffer;
int nbytes;
```

DESCRIPTION

DEVpixel_write writes a buffer to a pixel DSP. The data is transferred using parallel DMA.

pixel_system is a pointer to the system descriptor, *node* is the number of the pixel node from which the data is to be written. *addr* is the location in the DSP address space to which the data is to be sent. *addr* must be an even memory location, aligned on a 16-bit word boundary. *buffer* points to the data to be sent. *nbytes* is the number of bytes of data to be written. *nbytes* should always be an even number. If *nbytes* is odd, *nbytes*+1 bytes of data will be written.

No byte order translation is performed. The data sent will be in the same byte order as it is in *buffer*.

DEVpixel_write uses parallel DMA I/O to transfer the data. As a result, the parallel control register is updated by this routine. The parallel communications modes are altered to:

enable DMA

set PAR to be autoincremented on DMA

set the interrupt vector to 16-bit mode

RETURNS

DEVpixel_write should always return zero.

If *nbytes* is odd, **DEVpixel_write** writes *nbytes*+1 bytes of data and returns −1 as its return value. The return value should be the number of bytes written, not zero.

NAME

DEVpoll_nodes – poll DSP processors for messages

SYNOPSIS

#include <host/devtools.h>

int DEVpoll_nodes(pixel_system, firstpipe, lastpipe, firstpixel, lastpixel, iter_cnt, sleep)
DEVpixel_system *pixel_system;
int firstpipe, lastpipe;
int firstpixel, lastpixel;
int iter_cnt;
int sleep;

DESCRIPTION

DEVpoll_nodes is used to poll one, several, or all of the DSP processors to see if they have a *user* or *system message* to be served. **DEVpoll_nodes** *must* be called if the user has calls to **printf, PMusermsg, PMsiodir** or **PMhost_exit,** in a pipe or pixel node program.

firstpipe and *lastpipe* are the node numbers of the lowest and highest pipe node processors to be polled. *firstpixel* and *lastpixel* are the node numbers of the lowest and highest pixel node processors to be polled. The lowest node on a system is always zero; the highest number is the number of nodes minus one. If either the pipe or pixel nodes are not to be polled, DEV_NONE should be supplied for both the first and last values.

iter_cnt is the number of times the designated processors are to be polled. If *iter_cnt* is DEV_FOREVER, the polling process continues until an *exit* message is sent from one of the polled processors or until the host program is interrupted. An exit message can be sent from a processor by calling the **PMhost_exit** function.

sleep is the amount of time to sleep between each time the processors are polled. All of the processors are polled before the system sleeps. If *sleep* is DEV_NONE, no sleep call is made. The *sleep* value is passed to the **usleep** system call. DEV_NONE should only be used for applications that require very fast response to Pixel Machine message requests because it causes the host to consume a large amount of CPU time.

RETURNS

DEVpoll_nodes returns after all of the specified processors have been polled *iter_cnt* times, or when an exit message is received from any of the polled nodes. The return value is 1 if an exit message was received, 0 if the specified number of iterations have been completed.

EXAMPLE

```
#include <host/devtools.h>

main()
{
        if ((DEVinit() == NULL)) {
                exit(1);
        }
        DEVrun(DEVsystem);
        DEVpoll_nodes(DEVsystem, 0, DEVlast_pipe(DEVsystem),
                        0, DEVlast_pixel(DEVsystem), DEV_FOREVER, DEV_NONE);
        DEVexit();
}
```

SEE ALSO

DEVexit(3H)
DEVinit(3H)
PMhost_exit(3N)
printf(3N)
PMusermsg(3N)
PMsiodir(3X)
usleep(2) on the host system

NAME

DEVput_color_map - update color tables from video controller board and return the value

SYNOPSIS

```
#include <host/devtools.h>

void DEVput_color_map(pixel_system, r, g, b)
DEVpixel_system         *pixel_system;
int                     r[DEV_VIDEO_TABLE];
int                     g[DEV_VIDEO_TABLE];
int                     b[DEV_VIDEO_TABLE];
```

DESCRIPTION

DEVput_color_map updates the color tables from the video controller board and returns the values to the caller. Each color table contains 256 entries; each entry is a 10-bit value (0-1023).

SEE ALSO

DEVget_color_map(3S)

NAME

DEVput_image_header – write a Pixel Machine image header to a file

SYNOPSIS

```
#include <stdio.h>
#include <host/devtools.h>
#include <host/devimage.h>
#include <host/deverror.h>

int DEVput_image_header(file, image_header, optional_header)
FILE                *file;
DEVimage_header     *image_header;
DEVbyte             *optional_header;
```

DESCRIPTION

DEVput_image_header writes the DEVimage_header and the optional user header (if one exists) to the specified file.

file is a file descriptor obtained from a previous call to *fopen(3)*. The file must have been successfully opened for writing and the file pointer should be pointing to the beginning of the file (i.e., no previous writes have been issued). Upon return from **DEVput_image_header**, the file pointer will be set to where the pixel data should start (i.e., past the image and optional headers).

DEVput_image_header will convert the DEVimage_header structure pointed to by *image_header* into a string of decimal ASCII characters and write it to the file pointed to by *file*. If the *magic* structure member is 0, it will be set to DEV_IMAGE_MAGIC before being written. If *magic* is non–zero, it will be written as is.

If *optional_header* is non–zero, the characters pointed to it will be written to *file* immediately after the image header. *image_header–>optional_header_size* bytes will be written.

RETURNS

DEVput_image_header returns 0 upon success and –1 on failure. **DEVput_image_header** will set **DEVerrno** to indicate the reason for failure:

DEV_ERR_BAD_MAGIC: The magic number is not DEV_IMAGE_MAGIC.

DEV_ERR_WRITE_ERR: An error was returned by the *fwrite(3)* system call while writing either the image header or the optional header.

NOTES

No value in the **DEVimage_header** should be greater than 100,000,000.

SEE ALSO

DEVimage_header(4)
DEVget_image_header(3S)

NAME

DEVput_pixel, DEVput_pixels – write pixels into the frame buffer

SYNOPSIS

```
#include <host/devtools.h>

void DEVput_pixel(system,buffer,x,y,r,g,b,o)
DEVpixel_system *system;
int buffer, x, y;
short r, g, b, o;

void DEVput_pixels(system,buffer,x,y,r,g,b,o,npixl)
DEVpixel_system *system;
int buffer, x, y;
short *r, *g, *b, *o;
int npixl;
```

DESCRIPTION

DEVput_pixel writes pixels into a frame buffer. Through this routine, a program can update the Pixel Machine frame buffer without having to deal with the details of how the frame memory is organized on different models of the system.

system is a pointer to the system description information returned by **DEVinit**. *buffer* indicates which frame buffer is to be updated (must be the value 0 or 1). *x* is the x coordinate, *y* is the y coordinate. *r*, *g*, *b*, and *o* are the values to be stored in the red, green, blue, and overlay values in the frame buffer.

DEVput_pixels writes a sequence of pixels for a single scan line. *npixl* is the number of pixels to be written. *r*, *g*, *b*, and *o* point to a sequence of red, green, blue, and overlay values to be written.

NOTES

DEVput_scan_line provides a more efficient and flexible facility for dowloading image data.

SEE ALSO

DEVpixel_write(3S)
DEVput_scan_line(3H)

NAME

DEVput_scan_line - download an image or a portion of an image to a Pixel Machine

SYNOPSIS

```
#include <host/devtools.h>
#include <host/devimage.h>

int DEVput_scan_line(system, x, y, npixels, nlines, mode, pixel_buffer)
DEVpixel_system *system;
unsigned int x, y;
unsigned int npixels, nlines, mode;
DEVpixel *pixel_buffer;
```

DESCRIPTION

DEVput_scan_line transfers an image or a portion of an image from the host to a Pixel Machine. The data is transferred from the host memory area specified by *pixel_buffer* according to the mode specified by *mode*.

system is a pointer to the system description information returned by **DEVinit**. *x* is the starting x screen coordinate, *y* is the starting y screen coordinate.

The buffer pointed to by *pixel_buffer* must contain (*npixels* * *nlines* * pixel size) bytes, where the pixel size is determined by the *mode* argument as described below. In all cases, pixels will be accessed in *pixel_buffer* in the following order: (*x*,*y*), (*x*+1,*y*), ..., (*x*+*npixels*-1,*y*), (*x*,*y*+1), ..., (*x*+*npixels*-1,*y*+*nlines*-1).

The *mode* argument is used to specify two pieces of information: how the pixels are stored in the array pointed to by *pixel_buffer*, and which portion of Pixel Machine memory the data should be copied to. These values are or'ed into the *mode* argument. Valid pixel format values for *mode* are:

DEV_RGBA_PACKED_PIXELS: pixels are stored in *pixel_buffer*, 4 bytes to a pixel, in the following order: red, green, blue, alpha.

DEV_RGB_PACKED_PIXELS: pixels are stored in *pixel_buffer*, 3 bytes to a pixel, in the following order: red, green, blue.

DEV_MONO_R_PIXELS: pixels are stored in *pixel_buffer*, 1 byte to a pixel, with the red component of the pixel actually being stored. This option is not available when downloading to DEV_ZRAM_BUFFER.

DEV_MONO_G_PIXELS: pixels are stored in *pixel_buffer*, 1 byte to a pixel, with the green component of the pixel actually being stored. This option is not available when downloading to DEV_ZRAM_BUFFER.

DEV_MONO_B_PIXELS: pixels are stored in *pixel_buffer*, 1 byte to a pixel, with the blue component of the pixel actually being stored. This option is not available when downloading to DEV_ZRAM_BUFFER.

DEV_MONO_A_PIXELS: pixels are stored in *pixel_buffer*, 1 byte to a pixel, with the alpha (overlay) component of the pixel actually being stored. This option is not available when downloading to DEV_ZRAM_BUFFER.

DEV_MONO_PIXELS: pixels are stored in *pixel_buffer*, 1 byte to a pixel. When downloading to VRAM, the 1 byte pixel is written to the red, green, and blue component of a pixel.

DEV_MONO_16_PIXELS: pixels are stored in *pixel_buffer*, 2 bytes to a pixel. This option is only available when downloading to DEV_ZRAM_BUFFER.

DEV_DSP_FLOAT_PIXELS: pixels are stored in *pixel_buffer*, 4 bytes to a pixel. This option

is only available when downloading to DEV_ZRAM_BUFFER.

DEV_IEEE_FLOAT_PIXELS: pixels are stored in *pixel_buffer*, 4 bytes to a pixel. During the download operation, the pixels are converted from IEEE format floating point to DSP floating point. This option is only available when downloading to DEV_ZRAM_BUFFER.

The following values are or'ed into the *mode* argument to specify which portion of Pixel Machine memory to download to:

DEV_FRONT_BUFFER: Download pixels to the front (currently displayed) portion of VRAM.

DEV_BACK_BUFFER: Download pixels to the back (currently non-displayed) portion of VRAM.

DEV_VRAM0_BUFFER: Download pixels to the VRAM0 portion of VRAM.

DEV_VRAM1_BUFFER: Download pixels to the VRAM1 portion of VRAM.

DEV_ZRAM_BUFFER: Download pixels to ZRAM.

The sizes of the above buffers vary depending on the type of Pixel Machine being used as defined in the following table:

Model	FRONT	BACK	VRAM0	VRAM1	ZRAM
916	1024x1024	1024x1024	-	-	1024x1024
920	1280x1024	1280x1024	-	-	1280x1024
932	1024x1024	1024x1024	1024x2048	1024x2048	1024x2048
940	1280x1024	1280x1024	1280x2048	1280x2048	1280x2048
964	2048x1024	2048x1024	2048x2048	2048x2048	2048x2048
964X	2048x1024	2048x1024	2048x2048	2048x2048	2048x2048

Note that subscreens are not used when downloading to ZRAM.

RETURNS

DEVput_scan_line returns 0 upon success and -1 on failure. **DEVput_scan_line** also sets **DEVerrno** and **DEVerr_msg** upon failure.

NOTES

DEVput_scan_line sends a series of system commands to the pipe and pixel nodes to perform the download operation. Pixel node programs must be prepared to receive this command or **DEVput_scan_line** will fail. The pipe node programs must use **PMgetop** to read command opcodes. The download commands are implicitly copied through the pipe by **PMgetop**. The pixel node program should call **PMenable** during its initialization and should call **PMgetcmd** in its main processing loop. **PMgetcmd** will recognize the system command and call the appropriate routine to display the scan line(s).

DEVput_scan_line is an optimized version of **DEVput_pixels** for operations like image upload and image processing.

SEE ALSO

DEVput_pixels(3S)
DEVinit(3H)
PMenable(3N)
PMgetcmd(3X)
PMgetop(3P)

NAME

DEVread_z – read a buffer of bytes from the Z memory of a pixel node

SYNOPSIS

```
#include <host/devtools.h>

void DEVread_z(pixel_system, node, x, y, buffer, n)
DEVpixel_system *pixel_system;
int node;
int x, y;
DEVbyte *buffer;
int n;
```

DESCRIPTION

DEVread_z reads a buffer of bytes from the Z memory of a pixel node. *pixel_system* is a pointer to the memory mapped control block of the processor whose memory is to be read. *x* and *y* are the coordinates in the Z memory where the read operation starts. *buffer* is a pointer to the area into which the data is to be read. *n* is the number of bytes to be read.

The Z memory is organized as 256 rows of 256 32-bit words. "x" is the row from which the data is to be read, "y" is the word offset of the data to be read. An even number of bytes is always read.

Transfers must not attempt to wrap past the end of a row, or, in other words, the offset in bytes (y * 4) plus the number of bytes read (*n*) must not exceed the number of bytes per row (1024).

NOTES

This routine does not perform any byte order changes.

SEE ALSO

DEVpixel_read(3S)
DEVget_scan_line(3H)

NAME

DEVrelease_pipe_semaphore, DEVrelease_pixel_semaphore - clear the software semaphore in the memory of one of the DSP processors

SYNOPSIS

```
#include <host/devtools.h>

void DEVrelease_pipe_semaphore(pixel_system,node)
DEVpixel_system *pixel_system;
int node;

void DEVrelease_pixel_semaphore(pixel_system,node)
DEVpixel_system *pixel_system;
int node;
```

DESCRIPTION

These routines are used to clear the software semaphore in the memory of one of the DSP processors. *pixel_system* is a pointer to the system descriptor, *node* is the number of the pipe or pixel node whose semaphore is to be reset.

The semaphore can be set by a program running on one of the nodes by calling the **PMsetsem** routine.

These routines are used by the message serving system, but may also be used by user applications that do not make use of the message serving routines. They should never be called by routines that serve message requests from the Pixel Machine, as this would effect the synchronization between the Pixel Machine and host system.

NAME

DEVrun – begin execution of all pipe and pixel nodes

SYNOPSIS

void DEVrun(pixel_system)
DEVpixel_system *pixel_system;

DESCRIPTION

DEVrun is used to begin execution of the programs loaded into all pipe and pixel node processors. *pixel_system* is the system pointer returned by **DEVinit**. **DEVinit** must be called before calling **DEVrun**. If **DEVpipe_boot** and **DEVpixel_boot** are used, they must be called before calling **DEVrun**.

SEE ALSO

DEVinit(3H)
DEVpipe_boot(3H)
DEVpixel_boot(3H)

NAME

DEVserial_direction - updates the serial I/O link direction

SYNOPSIS

#include <host/devtools.h>

int DEVserial_direction(system, direction)
DEVpixel_system *system;
int direction;

DESCRIPTION

DEVserial_direction updates the serial I/O link direction.

system is a pointer to the system description information returned by **DEVopen.** *direction* indicates the direction in which data is to be transferred, and must be one of:

DEV_NORTH
DEV_EAST
DEV_SOUTH
DEV_WEST

Based on the system type, the appropriate calls to **DEVpixel_mode_serial** are executed to configure the system for the desired serial I/O direction.

RETURNS

Returns 0 on success.

SEE ALSO

devprint(1)
DEVpoll_nodes(3M)
PMsiodir(3X)

NAME

DEVshadow_off - turns off updating of color lookup tables from shadow tables

SYNOPSIS

```
#include <host/devtools.h>

void DEVshadow_off(pixel_system)
DEVpixel_system        *pixel_system;
```

DESCRIPTION

DEVshadow_off turns off updating of the color lookup tables from the shadow tables. To avoid flickering caused by partially updated color tables, this function should be called before updating the lookup tables.

SEE ALSO

DEVshadow_on(3S)

NAME

DEVswap_long – convert from DSP32 long integers to host long integers

SYNOPSIS

```
#include <host /devtools.h>

void
DEVswap_long(buffer, nbyte)
DEVbyte *buffer;
int    nbyte;
```

DESCRIPTION

DEVswap_long converts an array of long integers in DSP32 format to long integers in the host format(and vice–versa). The pointer to the array is passed in the argument *buffer*. The size of the array in bytes is passed in the argument *nbyte*. *nbyte* is not the number of elements in the array.

The conversion is done in place.

SEE ALSO

DEVswap_short(3S)
DEVdsp_ieee(3S)
DEVieee_dsp(3S)
DEVsswapl(3S)

NAME

DEVswap_pipe – switch primary and alternate pipes of a dual pipe system

SYNOPSIS

#include <host/devtools.h>

void DEVswap_pipe()

DESCRIPTION

On a dual pipe system, with the pipes operating in parallel mode, one pipe is the primary pipe and the other is the alternate pipe. **DEVswap_pipe** reverses the functions of the two pipes. This is used to balance the load between the two pipes.

DEVswap_pipe sends a *system* command to the primary pipe to perform the broadcast bus arbitration. The command is passed through each of the pipe nodes until it reaches the last pipe node. When the last pipe node processes the swap–pipe command, it releases the broadcast bus to the alternate pipe. It then requests the bus and waits for bus access to be granted.

NOTES

Programs in pipe nodes 8 and 17 must have called **PMenable**(PM_ENABLE_SWAP_PIPE) in order to correctly respond to the *system* command that **DEVswap_pipe()** sends.

Pipe node programs must use **PMgetop** to read command opcodes. The swap–pipe commands are implicitly copied through the pipe by **PMgetop**.

Pipe node programs can control the broadcast bus independently using the **PMswap_pipe** function.

SEE ALSO

PMenable(3N)
PMgetop(3P)
PMbus_wait(3P)
PMswap_pipe(3P)

NAME

DEVswap_short – convert from DSP32 short integers to host short integers

SYNOPSIS

```
#include <host/devtools.h>

void
DEVswap_short(buffer, nbyte)
DEVbyte *buffer;
int    nbyte;
```

DESCRIPTION

DEVswap_short converts an array of short integers in DSP32 format to short integers in the host format (and vice–versa). The pointer to the array is passed in the argument *buffer*. The size of the array in bytes is passed in the argument *nbyte*. *nbyte* is not the number of elements in the array.

The conversion is done in place.

SEE ALSO

DEVswap_long(3S)
DEVdsp_ieee(3S)
DEVieee_dsp(3S)

NAME

DEVuser_msg_enable – define a message code and specify functions to be called

SYNOPSIS

#include <host/devtools.h>
#include <host/msgserve.h>

int DEVuser_msg_enable(code, pipefunction, pixelfunction)
int code;
int (*pipefunction)(),
(*pixelfunction)();

DESCRIPTION

DEVuser_msg_enable allows a program to define a message code that is to be recognized by the polling routine, and to specify the functions that are to be called to service the message.

code is the user message code. It must be greater than zero, but must be less than the value DEV_HIGHEST_USER_MESSAGE (defined in host/msgserve.h).

When a user message with the value *code* is received from a DSP, the polling routine will call **pipefunction** if the message is from a pipe node, or **pixelfunction** if the message is from a pixel node.

pipefunction must be defined as:

```
int pipefunction(opcode, pixel_system, node)
int   opcode;
DEVpixel_system   *pixel_system;
int   node;
```

pixelfunction must be defined as:

```
int pixelfunction(opcode, pixel_system, node)
int   opcode;
DEVpixel_system, *pixel_system;
int   node;
```

opcode is the value of *code*; this allows one function to service several codes. *pixel_system* is the system descriptor. *node* is the node number of the processor that sent the message.

SEE ALSO

DEVpoll_nodes(3H)
PMusermsg(3N)

NAME

DEVwait_exit – wait for pixel nodes to signal completion, then call **DEVexit**

SYNOPSIS

void DEVwait_exit()

DESCRIPTION

DEVwait_exit sends a *system* command to all pixel nodes informing them that the host wishes to exit. The pixel node programs must have called **PMenable** with the PM_ENABLE_WAIT_EXIT argument at initialization in order to process the *system* command correctly.

Upon receipt of the *system* command, the pixel nodes perform a **PMpsync** operation to ensure all nodes have finished, then sends a message to the host. When the host sees this message, it automatically calls **DEVexit** before returning to the user.

SEE ALSO

DEVclose(3S)
DEVinit(3H)
DEVexit(3H)
PMenable(3N)

NAME

DEVwrite, DEVcwrite, DEVwriten, DEVcwriten, DEVwrite_alt, DEVcread, DEVreadn, DEVreadn_alt, – macros to write to the Pixel Machines pipelines and read commands back from the feedback FIFO

SYNOPSIS

```
#include <host/devtools.h>
#include <host/devcommand.h>

DEVulong DEVcommand(opcode,length)
short    opcode;
short    length;

short DEVcommand_opcode(command)
long     command;

short DEVcommand_length(command)
long     command;

void DEVcwrite0(command)
long     command;

void DEVcwrite0_alt(command)
long     command;

void DEVcwrite1(command,type,x)
long     command;
/* type is the type name of the remaining arguments */
type     x;

void DEVcwrite1_alt(command,type,x)
long     command;
/* type is the type name of the remaining arguments */
type     x;

void DEVwrite1(type,x)
/* type is the type name of the remaining arguments */
type     x;

void DEVwrite1_alt(type,x)
/* type is the type name of the remaining arguments */
type     x;

void DEVcwrite2(command,type,x,y)
long     command;
/* type is the type name of the remaining arguments */
type     x, y;

void DEVcwrite2_alt(command,type,x,y)
long     command;
/* type is the type name of the remaining arguments */
type     x, y;
```

```
void DEVwrite2(type,x,y)
/* type is the type name of the remaining arguments */
type    x, y;

void DEVwrite2_alt(type,x,y)
/* type is the type name of the remaining arguments */
type    x, y;

void DEVcwrite3(command,type,x,y,z)
long    command;
/* type is the type name of the remaining arguments */
type    x, y, z;

void DEVcwrite3_alt(command,type,x,y,z)
long    command;
/* type is the type name of the remaining arguments */
type    x, y, z;

void DEVwrite3(type,x,y,z)
/* type is the type name of the remaining arguments */
type    x, y, z;

void DEVwrite3_alt(type,x,y,z)
/* type is the type name of the remaining arguments */
type    x, y, z;

void DEVcwrite4(command,type,x,y,z,w)
long    command;
/* type is the type name of the remaining arguments */
type    x, y, z, w;

void DEVcwrite4_alt(command,type,x,y,z,w)
long    command;
/* type is the type name of the remaining arguments */
type    x, y, z, w;

void DEVwrite4(type,x,y,z,w)
/* type is the type name of the remaining arguments */
type    x, y, z, w;

void DEVwrite4_alt(type,x,y,z,w)
/* type is the type name of the remaining arguments */
type    x, y, z, w;

void DEVcwrite5(command,type,a,b,c,d,e)
long    command;
/* type is the type name of the remaining arguments */
type    a, b, c, d, e;

void DEVcwrite5_alt(command,type,a,b,c,d,e)
long    command;
/* type is the type name of the remaining arguments */
type    a, b, c, d, e;
```

```
void DEVwrite5(type,a,b,c,d,e)
/* type is the type name of the remaining arguments */
type    a, b, c, d, e;

void DEVwrite5_alt(type,a,b,c,d,e)
/* type is the type name of the remaining arguments */
type    a, b, c, d, e;

void DEVcwrite6(command,type,a,b,c,d,e,f)
long    command;
/* type is the type name of the remaining arguments */
type    a, b, c, d, e, f;

void DEVcwrite6_alt(command,type,a,b,c,d,e,f)
long    command;
/* type is the type name of the remaining arguments */
type    a, b, c, d, e, f;

void DEVwrite6(type,a,b,c,d,e,f)
/* type is the type name of the remaining arguments */
type    a, b, c, d, e, f;

void DEVwrite6_alt(type,a,b,c,d,e,f)
/* type is the type name of the remaining arguments */
type    a, b, c, d, e, f;

void DEVcwrite7(command,type,a,b,c,d,e,f,g)
long    command;
/* type is the type name of the remaining arguments */
type    a, b, c, d, e, f, g;

void DEVcwrite7_alt(command,type,a,b,c,d,e,f,g)
long    command;
/* type is the type name of the remaining arguments */
type    a, b, c, d, e, f, g;

void DEVcwrite8(command,type,a,b,c,d,e,f,g,h)
long    command;
/* type is the type name of the remaining arguments */
type    a, b, c, d, e, f, g, h;

void DEVcwrite8_alt(command,type,a,b,c,d,e,f,g,h)
long    command;
/* type is the type name of the remaining arguments */
type    a, b, c, d, e, f, g, h;

void DEVcwrite9(command,type,a,b,c,d,e,f,g,h,i)
long    command;
/* type is the type name of the remaining arguments */
type    a, b, c, d, e, f, g, h, i;

void DEVcwrite9_alt(command,type,a,b,c,d,e,f,g,h,i)
long    command;
```

```
/* type is the type name of the remaining arguments */
type    a, b, c, d, e, f, g, h, i;

void DEVwrite9(type,a,b,c,d,e,f,g,h,i)
long    command;
/* type is the type name of the remaining arguments */
type    a, b, c, d, e, f, g, h, i;

void DEVwrite9_alt(type,a,b,c,d,e,f,g,h,i)
long    command;
/* type is the type name of the remaining arguments */
type    a, b, c, d, e, f, g, h, i;

void DEVcwriten(command,type,block,length)
long    command;
/* type is the type name of block */
type    block[];
int     length;

void DEVcwriten_alt(command,type,block,length)
long    command;
/* type is the type name of block */
type    block[];
int     length;

void DEVwriten(type,block,length)
/* type is the type name of block */
type    block[];
int     length;

void DEVwrite_alt(type,block,length)
/* type is the type name of block */
type    block[];
int     length;

void DEVcread0(command)
long    command;

void DEVcread0_alt(command)
long    command;

void DEVreadn(type,block,length)
/* type is the type name of block */
type    block[];
int     length;

void DEVreadn_alt(type,block,length)
/* type is the type name of block */
type    block[];
int     length;
```

DESCRIPTION

These macros are used to write commands to the Pixel Machine pipelines and to read commands back from the feedback FIFO.

Each command consists of a command code, an operand count, and a list of 32-bit operands. The operands can be integers, host floating point numbers, or Pixel Machine floating point numbers. The interpretation of the contents of the operands is the responsibility of the user written code on the Pixel Machine that interprets the commands.

Macros that end with the string **_alt** write to the alternate pipe of a multi-pipe system, the routines without **_alt** write to the primary pipe. **_alt** macros must not be used on single pipe systems or on multi-pipe systems whose pipes are configured in parallel.

DEVcommand is used to encode an opcode and parameter count into a 32-bit command code. The *command* argument of the **DEVcwrite** macros is usually a call to **DEVcommand.**

DEVcommand_opcode and **DEVcommand_length** are used with the **DEVreadn** macros to extract the opcode and length from the encoded value.

The **DEVcwrite0** through **DEVcwrite9** macros are used to write commands and a number of operands that match the last character of the macro name. **DEVwrite0** through **DEVwrite9** macros write only operands, they do not output a command code.

The read and write macros contain a *type* argument. This indicates the type of the arguments being read or written. The storage class of the *type* argument must be such that sizeof(type) == 4 bytes and *type* is word aligned. All of the argument types in a given macro invocation must be the same. To create a command with four arguments, the first two of which are floats and the last two of which are ints, the following sequence of commands must be used:

```
DEVcwrite2(DEVcommand(opcode, 4), float, x, y);
DEVwrite2(int, i, j);
```

The **DEV_writen** and **DEV_readn** macros are used to write and read a block of operands. *block* is an array of values to be used as operands. *length* is the number of elements of *block* to be used. *length* must be less then or equal to 64.

NOTES

In a pipeless Pixel Machine, the **DEVwrite** macros write directly to the broadcast bus FIFOs. The **DEVread** and **_alt** macros should not be used in a pipeless Pixel Machine.

NAME

DEVwrite_z - writes a buffer of bytes into the Z memory of a pixel node

SYNOPSIS

```
#include <host/devtools.h>

void DEVwrite_z(pixel_system,node, x, y, buffer, n)
DEVpixel_system *pixel_system;
int node;
int x, y;
DEVbyte *buffer;
int n;
```

DESCRIPTION

DEVwrite_z writes a buffer of bytes into the Z memory of a pixel node. *pixel_system* is a pointer to the system description information returned by **DEVopen()**. *x* and *y* are the coordinates in the Z memory where the write operation starts. *buffer* is a pointer to the data to be written. *n* is the number of bytes to be written.

The Z memory is organized as 256 rows of 256 32-bit words. "x" is the row to which the data is to be written, "y" is the word offset of the data to be written. An even number of bytes is always written.

Transfers must not attempt to wrap past the end of a row, or, in other words, the offset in bytes (y * 4) plus the number of bytes written (*n*) must not exceed the number of bytes per row (1024).

NOTES

This routine does not perform any byte order changes.

SEE ALSO

DEVput_scan_line(3H)
DEVopen(3S)
DEVpixel_write(3S)

NAME

PMapply – apply a function to all subscreens

SYNOPSIS

#include <pxm.h>

void PMapply(function [,arg] ...)
void (*function)(PMsubscrn *scrn ...);

DESCRIPTION

PMapply provides a convenient method of calling a rendering *function* once for each subscreen, independent of the Pixel Machine model the code is being run on. *function* must take a pointer to a subscreen structure as its first argument, which is inserted by **PMapply**; the other *args* given to **PMapply** are passed on unchanged in each call to *function* made by **PMapply**.

EXAMPLES

To set a pixel node's image memory to a specified color using the DEVtools routine **PMclear**:

```
PMpixeltype color;

PMapply(PMclear, 0, 0, PMimax, PMjmax, &color);
```

Without **PMapply** the above call would have to be written:

```
PMclear(PMscrns[0],0,0,PMimax,PMjmax,&color);
if (PMmx)
{
      PMclear(PMscrns[1],0,0,PMimax,PMjmax,&color);
      if (PMmy)
      {
            PMclear(PMscrns[2],0,0,PMimax,PMjmax,&color);
            PMclear(PMscrns[3],0,0,PMimax,PMjmax,&color);
      }
}
```

Of course, if the user is not concerned with portability across different models of the Pixel Machine, neither **PMapply** nor the *if* statements are needed. In this case, specify 1, 2 or 4 calls to the required function (in this example **PMclear**) with the corresponding subscreen argument, depending on the number of subscreens in the model.

NOTES

PMapply is only useful in calling routines that do not modify their arguments and whose return value is not needed.

NAME

PMclear – fill a rectangular region of the screen

SYNOPSIS

```
#include <pxm.h>

void PMclear(scrn, imin, jmin, imax, jmax, color)
PMsubscrn *scrn;
short imin, jmin;
short imax, jmax;
PMpixeltype *color;
```

DESCRIPTION

PMclear fills a retangular section of a pixel node's subscreen memory with *color*. *scrn* is a pointer to an initialized PMsubscrn structure.

imin, *jmin*, *imax* and *jmax* are subscreen coordinates with the legal ranges:

i [0, PMimax]
j [0, PMjmax]

PMimax and **PMjmax** are automatically initialized to the appropriate value for the current model (see the *DEVtools User's Guide* for more information on subscreen ranges).

Values beyond these ranges will generate unpredictable results.

color is a pointer to a PMpixeltype structure containing the red, green, blue and overlay components to **PMclear** the region to. Each pixel within the region bounded by *imin*, *imax*, *jmin*, and *jmax* will be set to these values.

NOTES

Refer to **PMzbrk(3X)** for page register use.

NAME

PMcolor_float - macro that converts internal color value to floating point number

SYNOPSIS

```
#include <pxm.h>

float PMcolor_float(color)
int       color;
```

DESCRIPTION

PMcolor_float is a macro that converts an internal color value to a floating point number in the range 0.0 - 1.0.

SEE ALSO

PMint_color(3N)
PMcolor_int(3N)
PMfloat_color(3N)

NAME

PMcolor_int - macro that converts internal color value to an integer

SYNOPSIS

```
#include <pxm.h>

int PMcolor_int(color)
int     color;
```

DESCRIPTION

PMcolor_int is a macro that converts an internal color value to an integer in the range 0 - 255.

SEE ALSO

PMcolor_float(3N)
PMfloat_color(3N)
PMint_color(3N)

NAME

PMcopy_f – fast but dangerous 32 bit D/VRAM copy

SYNOPSIS

void PMcopy_f(to, from, count)
register float *to, *from;
register int count;

DESCRIPTION

PMcopy_f copies *count* words (4 bytes each) using a sequence of the longword-copy instruction:

```
a0 = (r3++ = r4++) * a0;
```

to reduce loop overhead.

to and *from* are any kind of pointer as long as they are 4 byte aligned. They can be pointers that use page registers. They will work properly as long as the appropriate page registers were correctly initialized.

For copying VRAM, it is necessary to call **PMcopy_f** twice, once with RG pointers and once with BO pointers.

This is the most efficient copy available. **PMcopy_f** calls *mover* which can copy up to 64 words with no overhead. *mover* resides in BANK 1 to eliminate conflict wait states in most cases.

For VRAM or DRAM to VRAM or DRAM copy, each 32-bit copy takes 550ns including clock stretching. For VRAM or DRAM to SRAM (and vice-versa) each copy is 375ns. For SRAM it takes 200ns plus any possible conflict wait states. If both pointers point to BANK 0 (.text section or automatic data), there are no wait states. If one pointer is in BANK 1, there is one 50ns conflict wait state, two if both pointers point to BANK 1. All global and static data generated by the C compiler reside in BANK 1 by default. Loop overhead is only encountered every 64 words.

NOTES

This copy is so blindingly fast that it may interfere with the video shift register load temporarily messing up the display. This problem only occurs in VRAM; it is perfectly safe in SRAM.

RETURNS

Results are undefined if the *to* and *from* pointers overlap.

If *count* < 1 it will be treated as a 1.

SEE ALSO

PMcopy_s(3X)
PMcopy_v(3X)

NAME

PMcopy_s – safe 32–bit DRAM or VRAM copy

SYNOPSIS

void PMcopy_s(to, from, count)
register float *to, *from;
register int count;

DESCRIPTION

PMcopy_s copies *count* words (4 bytes each) using a 2 instruction loop.

to and *from* are any kind of pointer, but they must be 4 byte aligned. They can be pointers that use page registers. They will work properly as long as the appropriate page registers were correctly initialized.

For copying VRAM, it is necessary to call **PMcopy_s** twice, once with RG pointers and once with BO pointers.

This copy is a little slower than **PMcopy_f**, but is guaranteed not to cause any video flashing problems. It also resides in BANK 1 to eliminate conflict wait states in most cases.

For VRAM or DRAM to VRAM or DRAM copy, each 32–bit copy takes 550ns including clock stretching. For VRAM or DRAM to SRAM (and vice-versa) each copy is 375ns. For SRAM it takes 200ns plus any possible conflict wait states. If both pointers point to video bank 0 (.text section or automatic data), there are no wait states. If one pointer is in video bank 1, there is one 50ns conflict wait state, two if both pointers point to video bank 1. All global and static data generated by the C compiler reside in video bank 1 by default. Add to these times 200ns for loop overhead per word.

NOTES

Results are undefined if the *to* and *from* pointers overlap or are not 4 byte aligned.

If *count* < 1 **PMcopy_s** will return immediately.

SEE ALSO

PMcopy_f(3X)
PMcopy_v(3X)

NAME

PMcopy_v – 32–bit copy with variable increments

SYNOPSIS

```
void PMcopy_v(to, from, to_inc, from_inc, count)
register float *to, *from;
register int  to_inc, from_inc;
register int count;
```

DESCRIPTION

PMcopy_v is similar to **PMcopy_s** but it allows the user to specify the increments for both the *to* and *from* pointers. The 32–bit copy and both increments are all accomplished in one DSP32 instruction, plus one more instruction for loop control.

to and *from* can be any kind of pointer, but must be 4 byte aligned. *to_inc* and *from_inc* are the increments to be added to the pointers after each 32–bit copy.

count is the number of 4 byte words to copy.

NOTES

Results are undefined if the *to* and *from* pointers overlap, or are not 4 byte aligned.

RETURNS

If *count* < 1 **PMcopy_v** returns immediately.

SEE ALSO

PMcopy_s(3X)

NAME

PMcopycmd - copy opcode, parameter count, and data from input to output FIFO of a pipe node

SYNOPSIS

#include <pxm.h>

void PMcopycmd()

DESCRIPTION

PMcopycmd copies the opcode, count, and parameters of a pipe command to the output FIFO. The parameters are copied directly from the input FIFO, but the opcode and count are copied from the **PMcommand** structure (which is initialized by a previous call to **PMgetop**).

NOTES

PMcopycmd can only be called from a pipe node program.

SEE ALSO

PMcommand(4N)
PMgetop(3P)
PMgetcmd(3X)
PMgetdata(3P)
PMputop(3P)
PMputdata(3P)

NAME

PMcopyftob - copy front to back

SYNOPSIS

```
#include <pxm.h>

void PMcopyftob(scrn, i, j, npix, nline)
PMsubscrn       *scrn;
int             i, j;
int             npix, nline;
```

DESCRIPTION

PMcopyftob copies a block of video memory from the front buffer to the back buffer.

scrn is a pointer to an initialized PMsubscrn structure. *i* and *j* are the starting location of the block to be copied. *npix* is the number of pixels and *nline* is the number of scan lines to be copied.

i and *j* are in the range of [0-PMimax] and [0-PMjmax], respectively. *npix* and *nlines* are in the range of [1-PMimax+1] and [1-PMjmax+1], respectively.

This function also works in single buffer mode.

NOTES

Values outside these ranges will generate unpredictable results.

To copy from the back to front buffer call **PMswapback** before and after the call to **PMcopyftob**.

PMcopyftob saves and restores page registers.

SEE ALSO

PMswapback(3X)
PMcopyvtov(3X)

NAME

PMcopyvtov – copy blocks of VRAM

SYNOPSIS

```
#include <pxm.h>

void PMcopyvtov(bank_from, bank_to, i, j, ni, nj, di, dj)
int     bank_from, bank_to;
int     i, j;
int     ni, nj;
int     di,dj;
```

DESCRIPTION

PMcopyvtov copies a block of video memory from the specified video bank *bank_from* to *bank_to*.

The banks can be PM_VRAM0_BUFFER or PM_VRAM1_BUFFER. You can copy from either bank to itself, or to the other bank.

i and *j* are the starting location of the block to be moved.

ni and *nj* are the number of pixels in the i and j directions, respectively, to be copied.

di and *dj* are the destination coordinates of the block.

i, *j*, *di* and *dj* are all in the range [0-255]; *ni* and *nj* are in the range [1-256].

NOTES

A value of less than 1 is treated as 1. Values outside these ranges will generate unpredictable results.

SEE ALSO

PMcopy_s(3N)

NAME

PMcopyvtoz – copy video RAM to DRAM

SYNOPSIS

```
#include <pxm.h>

void PMcopyvtoz(scrn, start_i, start_j, len_i, len_j, dest_i, dest_j, mode)
PMsubscrn *scrn;
int start_i;
int start_j;
int len_i;
int len_j;
int dest_i;
int dest_j;
int mode;
```

DESCRIPTION

PMcopyvtoz copies a rectangular section of the VRAM buffer, *len_i* pixels by *len_j* pixels, from the processor space coordinates, *start_i* and *start_j*, to the Z (DRAM) buffer. *dest_i* and *dest_j* are the destination coordinates in DRAM, and correspond to *start_i* and *start_j*, respectively. The section that is copied depends on the value of *mode*. If the value is the defined constant PM_FRONT_BUFFER, the image will be copied from the front, or visible, buffer. If the value is the defined constant PM_BACK_BUFFER, the image will be copied from the back, or invisible, buffer. Images in VRAM 1 can be copied by or'ing in the value PM_VRAM1_BUFFER.

The pixel data is organized in ZRAM so that the color data is placed into four adjacent bytes, in the order red, green, blue, overlay. Pixels with the same value of y are stored in the same row of memory; those with the same value of x and same color, are stored in the same column. Each pixel that is owned by a processor is adjacent to the next pixel owned by that processor, regardless of subscreen. For example, ZRAM for a 964 will contain data for every eighth pixel in both x and y, while ZRAM for a 916 will contain the data for every fourth pixel in both x and y. Each pixel is copied to a different, well defined, location. Pixels do not overwrite each other.

NOTES

It does not make sense to use PM_VRAM1_BUFFER on models 916 or 920 because screen pixels are already stored in both sections of VRAM.

This function can be called using **PMapply()**, which will call the function for all subscreens for each of the processor coordinates chosen. However, if the data to be copied does not align so that the upper left hand pixel falls on node 0 and subscreen 0, and the lower right hand pixel falls on the highest processor and highest subscreen, the function will need to be called more selectively. In that case the processor coordinates for each subscreen and the processors involved will need to be calculated from screen space, and this function called within each pixel node for each subscreen structure, with the appropriate arguments.

SEE ALSO

PMcopyztov(3X)
PMcopyztoz(3X)
PMqcopyztoz(3X)

NAME

PMcopyztov – copy DRAM to video RAM

SYNOPSIS

```
#include <pxm.h>

void PMcopyztov(scrn, start_i, start_j, len_i, len_j, dest_i, dest_j, mode)
PMsubscrn *scrn;
int start_i;
int start_j;
int len_i;
int len_j;
int dest_i;
int dest_j;
int mode;
```

DESCRIPTION

PMcopyztov copies a rectangular section of DRAM, *len_i* pixels wide by *len_j* pixels high, starting at coordinates *start_i* and *start_j*, to the VRAM buffer. *dest_i* and *dest_j* are the destination coordinates in VRAM, and correspond to *start_i* and *start_j*, respectively. The section of VRAM which is copied to depends on the value of *mode*. If the value is the defined constant PM_FRONT_BUFFER, the image will be copied to the front, or visible, buffer. If the value is the defined constant PM_BACK_BUFFER, the image will be copied to the back, or invisible, buffer. These values may be or'ed with PM_VRAM1_BUFFER to copy to VRAM1.

This function is the inverse of **PMcopyvtoz()**, and assumes that the data in DRAM has the structure that **PMcopyvtoz** would impose.

NOTES

It does not make sense to use PM_VRAM1_BUFFER on models 916 or 920 because screen pixels are already stored in both sections of VRAM.

SEE ALSO

PMcopyvtoz(3X)
PMcopyztoz(3X)
PMqcopyztoz(3X)

NAME

PMcopyztoz – copy from one section of DRAM to another

SYNOPSIS

```
#include <pxm.h>

void PMcopyztoz(start_i, start_j, len_i, len_j, dest_i, dest_j)
int start_i;
int start_j;
int len_i;
int len_j;
int dest_i;
int dest_j;
```

DESCRIPTION

PMcopyztoz copies a rectangular section of DRAM, with dimensions *len_i* long words (4 byte units) by *len_j* long words, starting at coordinates *start_i* from *start_j* to another section of DRAM buffer. *dest_i* and *dest_j* are the destination coordinates, and correspond to *start_i* and *start_j*, respectively.

The *_i* arguments are in units of 4 bytes, e.g., 1 byte for each of red, green, blue and overlay, or the space for one float. Thus, if *start_i* is set to 1, and *len_i* is set to 2, 8 bytes will be copied on each row, starting at an offset of 4 bytes from the beginning of the row. In the *_j* direction, one row is copied to one row.

NOTES

This function provides a copy from one address to another, arbitrary, address. If there is no chance of overlapping copies, the function **PMqcopyztoz()** should be used, because it is faster and uses less code space.

SEE ALSO

PMqcopyztoz(3X)
PMcopyvtoz(3X)
PMcopyztov(3X)

NAME

PMcos - trigonometric function to compute the cosine of an angle

SYNOPSIS

```
#include <libmath.h>

float PMcos(theta)
float theta;
```

DESCRIPTION

PMcos returns the cosine of *theta.*

theta must be in radians and be between -Pi/2 and +Pi/2.

NAME

PMdblbuff – enable double buffering mode

SYNOPSIS

#include <pxm.h>

void PMdblbuff()

DESCRIPTION

PMdblbuff enables double buffering. Double buffering implies a distinction between a visible buffer that is displayed by the video controller and a pixel buffer in which pixels are modified. **PMswapbuff** exchanges these two buffers.

PMsnglbuff disables double buffering.

SEE ALSO

PMswapbuff(3X)
PMsnglbuff(3X)

NAME

PMdelay - do nothing for a specified time

SYNOPSIS

```
#include <pxm.h>

void PMdelay(time)
int time;
```

DESCRIPTION

PMdelay executes a delay loop for (*time* / 250) seconds.

NAME

PMenable – enable processing of selected system commands

SYNOPSIS

#include <pxm.h>
#include <syscmd.h>

void PMenable(function)

DESCRIPTION

PMenable enables reception of certain *system* commands that are sent by host programs. After calling **PMenable**, any *system* commands that are generated by the host will be correctly processed when the pixel node receives them using **PMgetcmd**.

PMenable should be called as part of the program's initialization and *must* be called with one of the following #defines:

PM_ENABLE_GET_SCAN_LINE: Enables processing of **all** *system* commands sent by the **DEVget_scan_line** host routine. This option allows upload of pixels from both VRAM and ZRAM. This option only applies to pixel nodes.

PM_ENABLE_GET_VRAM: Enables processing of *system* commands sent by the **DEVget_scan_line** host routine to upload pixels from VRAM only. This option saves space if ZRAM pixel upload is not needed. This option only applies to pixel nodes.

PM_ENABLE_GET_ZRAM: Enables processing of *system* commands sent by the **DEVget_scan_line** host routine to upload pixels from ZRAM only. This option saves space if VRAM pixel upload is not needed. This option only applies to pixel nodes.

PM_ENABLE_PUT_SCAN_LINE: Enables processing of **all** *system* commands sent by the **DEVput_scan_line** host routine. This option allows download of pixels to both VRAM and ZRAM. This option only applies to pixel nodes.

PM_ENABLE_PUT_VRAM: Enables processing of *system* commands sent by the **DEVput_scan_line** host routine to download pixels to any portion of VRAM. This option saves space if ZRAM pixel upload is not needed. This option only applies to pixel nodes.

PM_ENABLE_PUT_ZRAM: Enables processing of *system* commands sent by the **DEVput_scan_line** host routine to download pixels to any portion of ZRAM. This option saves space if VRAM pixel upload is not needed. This option only applies to pixel nodes.

PM_ENABLE_SWAP_PIPE: Enables processing of *system* commands sent by the **DEVswap_pipe** host routine. This option only applies to pipe nodes, and should only be used by the last node of each parallel pipe (nodes 8 and 17).

PM_ENABLE_WAIT_EXIT: This option allows processing of *system* commands sent by the **DEVwait_exit** host function. This option applies only to pixel nodes.

If **PMenable** is *not* called before the host sends the *system* command, the *system* command will not be processed correctly.

NOTES

PMenable is implemented as a macro.

It is important to enable only those functions that will actually be used, because each one takes up additional code space.

SEE ALSO

DEVget_scan_line(3H)
DEVput_scan_line(3H)
DEVswap_pipe(3S)
DEVwait_exit(3H)
PMgetcmd(3X)

NAME

PMfb_on - direct output commands to the feedback FIFO
PMfb_off - direct output commands to the regular output FIFO

SYNOPSIS

#include <pxm.h>

void PMfb_on()
void Pbfb_off()

DESCRIPTION

PMfb_on directs the output of subsequent **PMputop, PMputdata**, and **PMcpycmd** calls to the feedback FIFO instead of the output FIFO.

PMfb_off redirects the output to the output FIFO instead of the feedback FIFO.

NOTES

These functions must only be called from the last pipe node of each pipe board (nodes 8 and 17).

NAME

PMfdiv - perform floating point division

SYNOPSIS

#include <libmath.h>
float PMfdiv(a, b)
float a, b;

DESCRIPTION

PMfdiv computes the floating point value *a* * (1.0 / *b*). If *b* is equal to zero, **PMfdiv** returns a large value of the same sign as *a*.

NOTES

PMfdiv is intended to be called by assembly language routines.

NAME

PMfreezaddr - decrement references to a page register

SYNOPSIS

```
#include <pxm.h>

void PMfreezaddr(ptr)
char *ptr;
```

DESCRIPTION

PMfreezaddr is called to decrement the number of references to a page register. *ptr* is the pointer returned by a previous call to **PMgetzaddr**. The pointer may have been incremented and still work with **PMfreezaddr** as long as it did not get incremented past the end of the block.

Neither the contents of the page register nor the contents of the memory are changed in any way. The purpose of **PMfreezaddr** is to make the page register available for use when it is no longer needed to access this particular address, so that it may be used by a call to **PMgetzaddr** with a different PMzdesc descriptor.

NOTES

If **PMfreezaddr()** is called with the PMzdesc returned by **PMgetzaddr()**, and **PMgetzaddr()** is called again with the same PMzdesc, the value of the returned pointer may change, but the contents of the memory pointed to will not be changed.

SEE ALSO

PMgetzaddr(3X)
PMgetzdesc(3X)
PMzbrk(3X)
PMblock_reg(3X)
PMavail_reg(3X)
PMset_lowreg(3X)
PMset_hireg(3X)

NAME

PMfxtoi - map a linear function of x from screen space to processor space i

SYNOPSIS

#include <pxm.h>

PMfxtoi(scrn, a, b)
PMsubscrn *scrn;
float a, b;

DESCRIPTION

PMfxtoi converts an expression of the form $f(x)=A_{xy}\ x+B_{xy}$ to an expression of the form $f(i)=A_{ij}\ i+B_{ij}$. The macro actually modifies the values of A and B.

In the above expressions, the subscripts xy and ij are used to denote a constant in (x,y) space and a constant in (ij) space, respectively.

NOTES

PMfxtoi is implemented as a macro.

SEE ALSO

DEVtools User's Guide
PMfxytoij(3X)
PMfytoj(3X)

NAME

PMfxytoij - map a linear function of x and y from screen space to processor space i and j

SYNOPSIS

#include <pxm.h>

PMfxytoij(scrn, a, b, c)
PMsubscrn *scrn;
float a, b, c;

DESCRIPTION

PMfxyoij converts an expression of the form $f(x,y)=A_{xy}\ x+B_{xy}\ y+C_{xy}$ to an expression of the form $f(i,j)=A_{ij}\ i+B_{ij}\ j+C_{ij}$. The macro actually modifies the values of A, B and C.

In the above expressions, the subscripts xy and ij are used to denote a constant in (x,y) space and a constant in (ij) space, respectively.

NOTES

PMfxytoij is implemented as a macro.

SEE ALSO

DEVtools User's Guide
PMfxtoi(3X)
PMfytoj(3X)

NAME

PMfytoj - map a linear function of y from screen space to processor space j

SYNOPSIS

```
#include <pxm.h>

PMfytoj(scrn, a, b)
PMsubscrn *scrn;
float a, b;
```

DESCRIPTION

PMfytoj converts an expression of the form $f(y)=A_{xy}\ y+B_{xy}$ to an expression of the form $f(j)=A_{ij}\ j+B_{ij}$. The macro actually modifies the values of A and B.

In the above expressions, the subscripts xy and ij are used to denote a constant in (x,y) space and a constant in (ij) space, respectively.

NOTES

PMfytoj is implemented as a macro.

SEE ALSO

DEVtools User's Guide
PMfxtoi(3X)
PMfxytoij(3X)

NAME

PMgetcmd – load command from a pixel node FIFO

SYNOPSIS

#include <pxm.h>

short PMgetcmd()

DESCRIPTION

PMgetcmd reads an opcode, parameter count, and parameters from the input FIFO and stores them in the global **PMcommand** structure. The parameters are placed in the array pointed to by PMcommand.data_ptr. The opcode is returned.

If the received command contains a negative opcode, the command is treated as a *system* command and the appropriate *system* function is invoked. If the appropriate *system* command has not been previously initialized by a call to **PMenable**, the command is ignored. In any case, **PMgetcmd** will consume all *system* commands until a user (positive opcode) command is read from the input FIFO.

NOTES

PMgetcmd can only be called from a pixel node.

Unlike pipe nodes, pixel nodes may only *receive* commands from the FIFO.

PMgetcmd is implemented as a macro.

SEE ALSO

PMcommand(4N)
PMenable(3N)

NAME

PMgetdata - get data from a pipe node FIFO

SYNOPSIS

#include <pxm.h>

void PMgetdata()

DESCRIPTION

PMgetdata reads parameters of a command from the input FIFO. The parameters are placed in the array pointed to by PMcommand.data_ptr.

NOTES

PMgetdata can only be called from a pipe node.

PMgetdata must be preceded by a call to **PMgetop**.

SEE ALSO

PMcommand(4N)
PMgetop(3P)
PMputdata(3P)

NAME

PMgetop – get opcode and parameter count from input FIFO of a pipe node

SYNOPSIS

#include <pxm.h>

short PMgetop()

DESCRIPTION

PMgetop loads an opcode and parameter count from the input FIFO and stores them in the global **PMcommand** structure. It returns the opcode.

If the received command contains a negative opcode, the command is treated as a *system* command and the appropriate *system* function is invoked. If the appropriate *system* command has not been previously initialized by a call to **PMenable**, the command is passed on to the output FIFO of this pipe node. In any case, **PMgetop** will consume all *system* commands until a user (positive opcode) command is read from the input FIFO.

NOTES

PMgetop can only be called from a pipe node.

PMgetop must be followed by a call to **PMgetdata** if PMcommand.count is non-zero.

PMgetop is implemented as a macro.

SEE ALSO

PMcommand(4N)
PMenable(3N)
PMgetdata(3P)
PMputop(3P)

NAME

PMgetpix – read a pixel from the current buffer

SYNOPSIS

```
#include <pxm.h>

short *PMgetpix(scrn, i, j, color)
PMsubscrn *scrn;
short i, j;
PMpixeltype *color;
```

DESCRIPTION

PMgetpix reads a single pixel from the frame buffer. *scrn* is a pointer to an initialized PMsubscrn structure corresponding to the subscreen from which the pixel is read.

i and *j* are subscreen coordinates with the following legal ranges:

i [0, PMimax]
j [0, PMjmax]

PMimax and **PMjmax** are set to the appropriate value for the current model by system initialization. (see the *DEVtools User's Guide* for more information on subscreen ranges).

Values beyond these ranges will generate unpredictable results.

color is a pointer to a PMpixeltype structure whose red, green, blue and overlay components will be loaded with the pixel data contained at (i,j) in *scrn*.

PMgetpix returns a pointer to the next pixel on the given row $(i+1,j)$. This pointer can be used by **PMqget** for more efficient frame buffer access.

NOTES

Refer to **PMzbrk(3X)** for page register use.

SEE ALSO

PMputpix(3X)
PMqget(3X)

NAME

PMgetrow, PMgetcol, PMputrow, PMputcol – read or write a scanline or scancolumn from pixel memory without subscreens

SYNOPSIS

```
#include <pxm.h>

void PMgetrow(buf, row, col, npix)
PMpixeltype *buf;
int row, col, npix;

void PMgetcol(buf, row, col, npix)
PMpixeltype *buf;
int row, col, npix;

void PMputrow(buf, row, col, npix)
PMpixeltype *buf;
int row, col, npix;

void PMputcol(buf, row, col, npix)
PMpixeltype *buf;
int row, col, npix;
```

DESCRIPTION

These four functions implement reading and writing pixels in *subscreen–independent* space. That is, these routines treat pixel memory as a single block of pixels and alternate access to subscreens as needed to preserve this illusion. Thus, in a 916, for example, instead of using **PMgetscan** and calling it four times with each of the four 128 by 128 subscreens, **PMgetrow** can be called once on a 256 by 256 buffer of pixels without the use of subscreens. This abstraction is useful for working in deinterleaved pixel space (e.g., filtering code). Either rows or columns can be accessed with these four functions.

For a full screen image, the size of *subscreen-independent* pixel memory is:

model	cols	rows
964X	160	128
964	128	128
940/932	128	256
920/916	256	256

PMgetrow and **PMputrow** read or write a row of pixels at a time, while **PMgetcol** and **PMputcol** read or write columns.

buf is a buffer of pixels to write to pixel memory (**PMputrow, PMputcol**) or read from pixel memory (**PMgetrow, PMgetcol**). The *buf* array must be large enough to store the requested pixels.

col and *row* are coordinates in *subscreen–independent space*. The number of pixels is specified in *npix*. Note that each pixel will take up 8 bytes (`sizeof(PMpixeltype)`) so *buf* must be 8 times *npix*.

To map from *screen space* to *subscreen–independent* processor space the coordinate conversion macros **(PMilo, PMihi, PMjlo, PMjhi)**, etc. should be used with the global PMrealscrn subscreen pointer.

NOTES

Subscreen-independent space is only an abstraction on top of subscreens. Although these routines do not use PMsubscrn pointers, they read and write pixels using subscreens, alternating between subscreens when needed. In most cases using the subscreen oriented routines will be faster because pixels are accessed linearly.

Refer to **PMzbrk(3X)** for page registers used.

SEE ALSO

PMgetscan(3X)
PMputscan(3X)

NAME

PMgetscan – read a scanline from a subscreen

SYNOPSIS

```
#include <pxm.h>

void PMgetscan(scrn, buf, row, col, npix)
PMsubscrn *scrn;
PMpixeltype *buf;
short row, col, npix;
```

DESCRIPTION

PMgetscan reads a row of *npix* pixels starting at (*col*,*row*) in subscreen *scrn* into the buffer *buf*, which must be large enough to hold the pixels.

scrn is a pointer to an initialized subscreen pointer. *col* and *row* are subscreen coordinates in the following legal ranges:

col [0,*PMimax*]

row [0,*PMjmax*]

PMimax and *PMjmax* are set to the appropriate value for the current model by system initialization (see the DEVtools *User's Guide* for more information on subscreen ranges).

Values beyond these ranges can generate unpredictable results.

SEE ALSO

PMgetcol(3X)
PMgetrow(3X)
PMputscan(3X)

NAME

PMgetzaddr - load a page register and return an address to a section of DRAM

SYNOPSIS

```
#include <pxm.h>

char *PMgetzaddr(desc)
PMzdesc desc;
```

DESCRIPTION

PMgetzaddr() is called to gain access to the portion of DRAM memory allocated by **PMgetzdesc** via a pointer and page register. *desc* is the Z memory descriptor returned from a previous call to **PMgetzdesc.**

A table of available page registers is maintained by **PMgetzaddr**. Page registers 0 through 13 are available by default. Registers may be blocked by calls to the macros **PMblock_reg(), PMavail_reg(), PMset_lowreg()** and **PMset_hireg()**. The table is searched to see if the 1K row containing the memory to be accessed has been loaded into a page register. If the row has already been loaded, the number of accesses using that page register is incremented and the address is returned. If the row is not already loaded, an unaccessed page register is searched for and loaded with the page descriptor, if such a page register is found. The number of accesses to the page register is then incremented.

If no page registers are available, it will be necessary to call **PMfreezaddr** to free one up and temporarily restrict access to that block. By careful use of **PMfreezaddr** and **PMgetzaddr** and knowing how many page registers are available, it should be possible to never run out of page registers.

RETURNS

PMgetzaddr() returns a pointer to the valid memory address, if a page register can be found. NULL is returned on failure.

NOTES

If **PMfreezaddr()** is called with the PMzdesc returned by **PMgetzaddr(),** and **PMgetzaddr()** is called again with the same PMzdesc, the value of the returned pointer may change, but the contents of the memory pointed to will not be changed.

Unpredictable results can occur if the memory past the end of the allocated block is accessed.

SEE ALSO

PMfreezaddr(3X)
PMgetzdesc(3X)
PMzbrk(3X)
PMblock_reg(3X)
PMavail_reg(3X)
PMset_lowreg(3X)
PMset_hireg(3X)

NAME

PMgetzbuf - read a float value from the Z buffer

SYNOPSIS

```
#include <pxm.h>

float *PMgetzbuf(scrn, i, j, zptr)
PMsubscrn *scrn;
short i, j;
float *zptr;
```

DESCRIPTION

PMgetzbuf reads a single value from Z buffer memory. *scrn* is a pointer to an initialized PMsubscrn structure corresponding to the subscreen from which the value is to be read.

i and *j* are subscreen coordinates with the following legal ranges:

i [0, PMimax]
j [0, PMjmax]

PMimax and **PMjmax** are set to the appropriate value for the current model by system initialization (see the *DEVtools User's Guide* for more information on subscreen ranges).

Values beyond these ranges will generate unpredictable results.

zptr is a pointer to a floating point number to be written with the Z value contained at (*i*,*j*) in *scrn*.

PMgetzbuf returns a pointer to the next Z value on the given row (*i*+1,*j*). This pointer can be used by **PMqzget** for more efficient Z buffer access.

For even faster access, the pointer returned can be used directly (unlike the pointer returned from **PMgetpix**) because Z buffer memory is fully mapped.

NOTES

Refer to **PMzbrk(3X)** for page register use.

The pointer returned can be cast to other types to allow the Z memory to be used for *char*, *int* and other data types.

EXAMPLE

```
PTR=PMgetzbuf(scrn,i,j,zval
    z2=ptr++
    z3=ptr++
```

SEE ALSO

PMgetpix(3X)
PMputzbuf(3X)
PMqzget(3X)
PMzget(3X)

NAME

PMgetzdesc, PMzdesc_valid - allocate a DRAM block

SYNOPSIS

#include <pxm.h>

PMzdesc PMgetzdesc(numbytes)
int numbytes;

PMzdesc_valid(desc)
PMzdesc desc;

DESCRIPTION

PMgetzdesc() is called after **PMzbrk()** has reserved the DRAM memory resources to allocate memory in blocks up to 1024 bytes. *numbytes* is the requested number of bytes, which must be less than or equal to 1024. The allocated memory is aligned on 4 byte boundaries.

PMgetzdesc returns a memory descriptor, of type PMzdesc, that contains two elements of addressing information. One element contains the number of the 1K block that holds the first available memory, and the other contains the offset of that memory from the beginning of that block. The offset is given in units of 4 bytes.

Memory is allocated from the beginning of the section reserved by **PMzbrk()** until the end of DRAM. No block may wrap over a 1K boundary, therefore, **PMgetzdesk** may have to skip over memory to guarantee this. Because of this, it is advisable to allocate memory in chunks that divide into 1024 evenly. Once a block of memory is allocated with **PMgetzdesc** it cannot be freed, except by reinitializing with a call to **PMzbrk,** which then starts the allocation process from the beginning.

In order to actually gain access to the memory being allocated, the descriptor must be used in a subsequent call to **PMgetzaddr().**

RETURNS

If successful, **PMgetzdesc** returns a descriptor of type PMzdesc, as described above. If there is no more reserved DRAM left or if the portion left is smaller than the *numbytes* requested, both elements of the returned descriptor are zero. Validity of a descriptor can be tested with the macro **PMzdesc_valid(desc),** where *desc* is the descriptor being tested. The value is non-zero if the result is valid.

NOTES

Requesting more than 1024 bytes can produce unpredictable results.

SEE ALSO

PMgetzaddr(3X)
PMfreezaddr(3X)
PMzbrk(3X)

NAME

PMhost_exit – send a message to the host that signals the completion of a Pixel Machine program

SYNOPSIS

void PMhost_exit()

DESCRIPTION

PMhost_exit sends a message to the host that causes the **DEVpoll_nodes** function to return to the caller. This is usually used to signal the completion of a Pixel Machine program, but may also be used in other applications where the Pixel Machine may want to request that **DEVpoll_nodes** return to the caller.

NOTES

If **devprint** is running on the host, **PMhost_exit** will cause it to terminate.

SEE ALSO

devprint(1)
DEVpoll_nodes(3H)
DEVwait_exit(3H)

NAME

PMieee_dsp - convert IEEE float to DSP float

SYNOPSIS

```
#include <libmath.h>

float *PMieee_dsp(len, ptr)
int len;
float *ptr;
```

DESCRIPTION

The *len* floating point numbers in IEEE format stored at *ptr* are converted to DSP32 format. A pointer immediately following the end of the array (*ptr+len*) is returned.

SEE ALSO

PMlong_dsp(3M)

NAME

PMihi - map from screen space (xmax) to processor space (ihi)

SYNOPSIS

```
#include <pxm.h>

int PMihi(scrn, x)
PMsubscrn *scrn;
float x;
```

DESCRIPTION

PMihi performs the mapping from screen space to processor space. The domain transformation that maps from Cartesian (x,y) screen space to (i,j) processor space is as follows:

$$i = \frac{1}{Nx}(x - Ox)$$

$$j = \frac{1}{Ny}(y - Oy)$$

where Nx and Ny are the numbers of processors in the x and y directions, respectively, and Ox and Oy are the x and y offsets into the processor array, respectively. **PMihi** converts a screen space coordinate x to a processor space coordinate i that will guarantee satisfying the condition :

$$i\ Nx + Ox \leq x$$

This ensures that all i values generated will map to screen coordinates less than or equal to x. The i value is always used as the last valid pixel to be rendered by a processor.

NOTES

PMihi is implemented as a macro.

SEE ALSO

DEVtools User's Guide
PMilo(3X)
PMjlo(3X)
PMjhi(3X)

NAME

PMilo - map from screen space (xmin) to processor space (ilo)

SYNOPSIS

```
#include <pxm.h>

int PMilo(scrn, x)
PMsubscrn *scrn;
float x;
```

DESCRIPTION

This macro performs the mapping from screen space to processor space. The domain transformation that maps from Cartesian (x,y) screen space to (i,j) processor space is as follows:

$$i = \frac{1}{Nx}(x - Ox)$$

$$j = \frac{1}{Ny}(y - Oy)$$

where Nx and Ny are the numbers of processors in the x and y directions, respectively, and Ox and Oy are the x and y offsets into the processor array, respectively.

PMilo converts a screen space coordinate x to a processor space coordinate i that guarantees satisfying the condition :

$$i\ Nx + Ox \geq x$$

This ensures that all i values generated will map to screen coordinates greater than or equal to x. The i value is always used as the first valid pixel to be rendered by a processor.

NOTES

PMilo is implemented as a macro.

SEE ALSO

DEVtools User's Guide
PMihi(3X)
PMjlo(3X)
PMjhi(3X)

NAME

PMint_color - macro that converts an integer to an internal color value

SYNOPSIS

```
#include <pxm.h>

int PMint_color(i)
int     i;
```

DESCRIPTION

PMint_color is a macro that converts an integer in the range 0 - 255 to an internal color value. Results for input values outside of the supported range are undefined.

SEE ALSO

PMcolor_int(3N)
PMcolor_float(3N)
PMfloat_color(3N)

NAME

PMinterleave – interleave or deinterleave a block

SYNOPSIS

```
#include <pxm.h>
#include <sysmsg.h>

void PMinterleave(mode, dir, x, y, nx, ny, ram )
int     mode;
int     dir;
int     x, y;
int     nx, ny;
int     ram;
```

DESCRIPTION

PMinterleave() deinterleaves or interleaves a rectangular region of the screen starting at *(x,y)* in screen space, for a size of *nx* pixels by *ny* scanlines in one dimension. The values of *x* and *y* are restricted to multiples of the number of processors in the *x* and *y* directions (*PMnx, PMny*), respectively.

nx and *ny* must be multiples of *PMnx* squared and *PMny* squared, respectively.

mode is either PM_INTERLEAVE or PM_DEINTERLEAVE, and specifies if this is an interleave or deinterleave operation.

dir is the dimension, either PM_ROW_INT or PM_COL_INT for horizontal or vertical.

x and *y* are the upper left hand coordinate of the block in screen space and are in the range [0-(PMxmax-1)] and [0-(PMymax-1)].

nx and *ny* are the number of pixels in the x and y direction, respectively, and are in the range [0-PMxmax)] and [0-PMxmax)].

The *ram* parameter is one of:

PM_VRAM1_BUFFER: uses VRAM1 instead of VRAM0 on a 932 and higher.
If in double buffer mode, (i,j) must be within the correct limits, otherwise they can be larger as with PM_FRONT_BUFFER.

PM_BACK_BUFFER: the currently non-displayed buffer.

PM_FRONT_BUFFER: the currently displayed buffer. Note, however, that in an appropriately large model in single buffer mode, you can specify i,j out of bounds, e.g., on a 964 (512,512) will work.

PM_ZRAM_BUFFER: uses ZRAM without subscreens.

To interleave (deinterleave) in two dimensions call **PMinterleave()** twice with the same parameters except change *dir* from PM_ROW to PM_COL (or vice-versa).

NOTES

For **PMinterleave()** to work, the Pixel Machine must be equipped with the necessary SIO hardware.

This function changes the SIO direction. The host must be polling via a call to **DEVpoll_nodes()** or running the **devprint(1)** utility. **PMpsync()** is called internally.

PMinterleave() needs 4200 bytes available on the stack.

Saves and restores any page registers that it uses.

SEE ALSO

PMpsync(3X)
PMsiodir(3X)
PMmsg_exchange(3X)
PMmsg_setup(3X)
PMsioinit(3X)
DEVpoll_nodes(3S)
devprint(1)

NAME

PMjhi – map from screen space (ymax) to processor space (jhi)

SYNOPSIS

```
#include <pxm.h>

int PMjhi(scrn, y)
PMsubscrn *scrn;
float y;
```

DESCRIPTION

PMjhi performs the mapping from screen space to processor space. The domain transformation that maps from Cartesian (x,y) screen space to (i,j) processor space is as follows:

$$i = \frac{1}{Nx}(x - Ox)$$

$$j = \frac{1}{Ny}(y - Oy)$$

where Nx and Ny are the numbers of processors in the x and y directions, respectively, and Ox and Oy are the x and y offsets into the processor array, respectively.

PMjhi converts a screen space coordinate y to a processor space coordinate j that will guarantee satisfying the condition :

$$j\ Ny + Oy \leq y$$

This ensures that all j values generated will map to screen coordinates less than or equal to y. The j value is always used as the last valid pixel to be rendered by a processor.

NOTES

PMjhi is implemented as a macro.

SEE ALSO

DEVtools User's Guide
PMilo(3X)
PMihi(3X)
PMjlo(3X)

NAME

PMjlo - map from screen space (ymin) to processor space (jlo)

SYNOPSIS

```
#include <pxm.h>

int PMjlo(scrn, y)
PMsubscrn scrn;
float y;
```

DESCRIPTION

PMjlo performs the mapping from screen space to processor space. The domain transformation that maps from cartesian (x,y) screen space to (i,j) processor space is as follows:

$$i = \frac{1}{Nx}(x - Ox)$$

$$j = \frac{1}{Ny}(y - Oy)$$

where Nx and Ny are the numbers of processors in the x and y directions, respectively, and Ox and Oy are the x and y offsets into the processor array, respectively.

PMjlo converts a screen space coordinate y to a processor space coordinate j that will guarantee satisfying the condition :

$$j\ Ny + Oy \geq y$$

This ensures that all j values generated will map to screen coordinates greater than or equal to y. The j value is always used as the first valid pixel to be rendered by a processor.

NOTES

PMjlo is implemented as a macro.

SEE ALSO

DEVtools User's Guide
PMilo(3X)
PMihi(3X)
PMjhi(3X)

NAME

PMldot - specialized dot product for light sources

SYNOPSIS

#include <libmath.h>

float PMldot(v0, v1)
float v0[3], v1[3];

DESCRIPTION

PMldot calculates the dot product of vectors *v0* and *v1*. If the result is negative, **PMldot** returns zero, otherwise it returns the value of the dot product.

NAME

PMlong_dsp - convert an array of longs to float

SYNOPSIS

```
#include <libmath.h>

long *PMlong_dsp(len, ptr)
int len;
float *ptr;
```

DESCRIPTION

The *len* **long** numbers stored at *ptr* are converted to **float.** A pointer immediately following the end of the array (*ptr+len*) is returned.

SEE ALSO

PMieee_dsp(3M)

NAME

PMmsg_exchange - send and receive data packet over serial links

SYNOPSIS

#include <pxm.h>

void PMmsg_exchange(inbuf, outbuf, length)
float *inbuf, *outbuf;
int length;

DESCRIPTION

PMmsg_exchange sends *length* floats from *outbuf* out the serial link, then waits to receive *length* floats into *inbuf* on the link. Because of restrictions imposed by hardware, all nodes must exchange the same amount of data at the same time; the correct procedure to do this uses the **PMmsg_setup** and **PMpsync** routines as follows:

```
float inbuf[SIZE], outbuf[SIZE];

PMmsg_setup(inbuf);
PMpsync();
PMmsg_exchange(inbuf, outbuf, SIZE);
```

Any data type may be exchanged over the link, but the packet size must be a multiple of 4 bytes *(sizeof(float))*.

NOTES

The *inbuf* pointers passed to **PMmsg_setup** and **PMmsg_exchange** must be the same or **PMmsg_exchange** may never return.

PMsioinit must be called before any other use of the serial links is made.

SEE ALSO

PMmsg_setup(3X)
PMpsync(3X)
PMsioinit(3X)

NAME

PMmsg_setup - set serial DMA input pointer

SYNOPSIS

```
#include <pxm.h>

void PMmsg_setup(buffer)
float *buffer;
```

DESCRIPTION

PMmsg_setup sets the serial DMA input pointer to the supplied *buffer*. The pointer must be set and all processors synchronized using **PMpsync** before **PMmsg_exchange** functions correctly.

SEE ALSO

PMmsg_exchange(3X)
PMpsync(3X)
PMsioinit(3X)

NAME

PMmyx - test if a given screen space coordinate is in processor space

SYNOPSIS

```
#include <pxm.h>

int PMmyx(scrn, x)
PMsubscrn *scrn;
float x;
```

DESCRIPTION

PMmyx tests if the screen space coordinate *x* is in the processor subscreen *scrn* and returns TRUE or FALSE accordingly.

NOTES

PMmyx is implemented as a macro.

SEE ALSO

PMmyy(3X)
PMxat(3X)
PMyat(3X)

NAME

PMmyy - test if a given screen space coordinate is in processor space

SYNOPSIS

```
#include <pxm.h>

int PMmyy(scrn, y)
PMsubscrn *scrn;
float     y;
```

DESCRIPTION

PMmyy tests if the screen space coordinate *y* is in the processor subscreen *scrn* and returns TRUE or FALSE accordingly.

NOTES

PMmyy is implemented as a macro.

SEE ALSO

PMmyx(3X)
PMxat(3X)
PMyat(3X)

NAME

PMnorm - normalize a 3D vector and return its length

SYNOPSIS

```
#include <libmath.h>

float PMnorm(v)
float v[3];
```

DESCRIPTION

PMnorm normalizes the vector *v*, and overwrites *v* with this new value. It returns the inverse of the length of vector *v* prior to normalization.

NAME

PMoutpir – output a value to the PIR register

SYNOPSIS

void PMoutpir(val)
short val;

DESCRIPTION

PMoutpir waits until the PIR is empty and then writes *val* to it. The wait ensures that the host has read all values written with previous calls to **PMoutpir.**

This function is a low level I/O routine; most applications should use **PMusermsg()** instead.

SEE ALSO

PMusermsg(3N)

NAME

PMoverlay – turn overlay on or off

SYNOPSIS

void PMoverlay(flag)
int flag;

DESCRIPTION

PMoverlay sets the overlay bit in the pixel node flag register to turn the overlay capability on or off.

If *flag* is zero, overlay is disabled (the default). A nonzero value for *flag* turns overlay on.

NOTES

In addition to calling **PMoverlay**, **DEVpixel_mode_overlay** must also be called on the host to set the desired overlay mode.

SEE ALSO

DEVpixel_mode_overlay(3S)

NAME

PMpagereg, PMdesc, PMxlate – macros to manipulate page registers used to access video and Z memory

SYNOPSIS

```
#include <pxm.h>
#include <pixel.h>

int PMpagereg(reg_number)
int     reg_number;

int PMdesc(bank, mode)
int     bank;
int     mode;

int PMxlate(reg_number)
int     reg_number;
```

DESCRIPTION

These macros are used to manipulate the page registers used to access the video memory and Z memory.

The page registers are located in a reserved memory area. The **PMpagereg** macro is used to generate the address of a specified page register. *reg_number* is the number of the register whose address is to be supplied and is in the range [0-15].

The **PMdesc** macro is used to generate the value to be stored into a page register in order to access a given bank of memory. *bank* designates the bank of memory to be accessed and must be one of:

PM_ZMEM	– Z memory
PM_RG0	– red/green bank of VRAM0
PM_BO0	– blue/overlay bank of VRAM0
PM_RG1	– red/green bank of VRAM1
PM_BO1	– blue/overlay bank of VRAM1

mode must be either PM_FIX_ROW or PM_FIX_COL; PM_FIX_ROW is used to access the pixels of a given scan line. PM_FIX_COL is used to access the pixels of a given column. The row number (in fixed row mode) or column number (in fixed column mode) is added to the value returned by **PMdesc** to create the descriptor needed to access the desired memory row or column.

PMxlate generates a pointer than can be used to access the contents of the row or column specified by the **PMdesc** macro. Once a page register has been established, the next 1024 bytes can be accessed using the pointer generated by the **PMxlate** macro.

EXAMPLE

The following is an example of these macros. This programs turns on all of the red pixels in VRAM0 and the blue pixels in VRAM1, and turns off the green pixels in VRAM0 and the overlay pixels in VRAM1.

```
#include <pxm.h>
#include <pixel.h>

#define RGREG      6
#define BOREG      7
```

```
main()
{
        register int        i;
        register int        j;
        register int        *rgptr;
        register int        *boptr;
        register int        *rgpagereg;
        register int        *bopagereg;

        rgpagereg = (int *)PMpagereg(RGREG);
        bopagereg = (int *)PMpagereg(BOREG);

        for (j = 0; j < 255; ++j) {
              *rgpagereg = PMdesc(PM_RG0, PM_FIX_ROW) + j;
              *bopagereg = PMdesc(PM_BO1, PM_FIX_ROW) + j;
              rgptr = (int *)PMxlate(RGREG);
              boptr = (int *)PMxlate(BOREG);
              for (i = 0; i < 255; ++i) {
                    *rgptr++ = PMint_color(255);   /* Set red  */
                    *rgptr++ = PMint_color(0);     /* Clear green    */
                    *boptr++ = PMint_color(255);   /* Set blue */
                    *boptr++ = PMint_color(0);     /* Clear alpha    */
              }
        }
}
```

NOTES

The **pixel.h** include file can be used with both C and assembler source files. As a result, the macro return values are not cast as pointers. For this reason, you must cast the return value of the macros to the appropriate pointer type.

PMpagereg should always be cast as a pointer to an *int*. **PMdesc** really does return an integer. **PMxlate** should be cast to an appropriate type based on the application. When dealing with VRAM (as opposed to Z memory), the pointer returned by **PMxlate** is usually a pointer to an *int*.

Some of the DEVtools pixel node functions set page registers automatically, and other functions rely on them. See **PMzbrk** for the list of page registers used.

Page registers 14 and 15 are reserved for use by the host for DMA.

SEE ALSO

PMzbrk(3X)

NAME

PMpixaddr – generate a pointer to a specific pixel

SYNOPSIS

```
#include <pxm.h>

short *PMpixaddr(scrn, i, j)
PMsubscrn *scrn;
short i, j;
```

DESCRIPTION

PMpixaddr generates addresses of pixels in the frame buffer. *scrn* is a pointer to an initialized PMsubscrn structure corresponding to the subscreen in which the desired pixel lies.

i and *j* are subscreen coordinates with the following legal ranges:

i [0, PMimax]
j [0, PMjmax]

PMimax and **PMjmax** are set to the appropriate value for the current model by system initialization (see the *DEVtools User's Guide* for more information on subscreen ranges).

Values beyond these ranges will generate unpredictable results.

PMpixaddr returns a pointer to the pixel at coordinates (*i,j*) in subscreen *scrn*. This pointer can be used by **PMqget** and **PMqput** for more efficient frame buffer access.

NOTES

Refer to **PMzbrk(3X)** for page register use.

SEE ALSO

PMgetpix(3X)
PMputpix(3X)
PMqget(3X)
PMqput(3X)

NAME

PMpow – power function

SYNOPSIS

```
#include <libmath.h>

float PMpow(x, y)
float x, y;
```

DESCRIPTION

PMpow returns the quantity *x* ^ *y*, where both *x* and *y* are floating point values. *x* should be of positive magnitude.

SEE ALSO

PMx_exp_n(3M)

NAME

PMpsync – wait for all pixel processors to synchronize

SYNOPSIS

void PMpsync()

DESCRIPTION

PMpsync is a processor synchronization primitive. Once called, it will not return until *all* pixel nodes have called **PMpsync**.

NOTES

PMpsync uses the PM_FLAG hardware signal; thus **PMflagled** and **PMpsync** should not be used in the same program.

SEE ALSO

PMvsync(3X)

NAME

PMputcmd - write opcode, parameter count, and parameters to the output FIFO of a pipe node

SYNOPSIS

#include <pxm.h>

void PMputcmd()

DESCRIPTION

PMputcmd copies the opcode, count, and parameters from the global **PMcommand** structure to the output FIFO.

NOTES

PMputcmd can only be called from a pipe node program.

SEE ALSO

PMcommand(4N)
PMgetdata(3P)
PMgetop(3P)
PMputdata(3P)
PMputop(3P)

NAME

PMputdata - write parameters to the output FIFO of a pipe node

SYNOPSIS

#include <pxm.h>

void PMputdata()

DESCRIPTION

PMputdata copies the parameters from the global PMcommand structure to the output FIFO.

NOTES

PMputdata can only be called from a pipe node program.

PMputdata must be preceded by a call to **PMputop**.

SEE ALSO

PMcommand(4N)
PMgetdata(3P)
PMgetop(3P)
PMputcmd(3P)
PMputop(3P)

NAME

PMputop - write opcode and parameter count to the output FIFO of a pipe node

SYNOPSIS

```
#include <pxm.h>

void PMputop()
```

DESCRIPTION

PMputop copies the opcode and parameter count from the global **PMcommand** structure to the output FIFO.

NOTES

PMputop can only be called from a pipe node program.

PMputop must be followed by a call to **PMputdata** if PMcommand.count is non-zero.

SEE ALSO

PMcommand(4N)
PMgetdata(3P)
PMgetop(3P)
PMputcmd(3P)
PMputdata(3P)

NAME

PMputpix – output a pixel to the current buffer

SYNOPSIS

```
#include <pxm.h>

short *PMputpix(scrn, i, j, color)
PMsubscrn *scrn;
short i, j;
PMpixeltype *color;
```

DESCRIPTION

PMputpix writes a single pixel to the frame buffer. *scrn* is a pointer to an initialized PMsubscrn structure corresponding to the subscreen to which the pixel is written.

i and *j* are subscreen coordinates with the following legal ranges:

i [0, PMimax]
j [0, PMjmax]

PMimax and **PMjmax** are set to the appropriate value for the current model by system initialization (see the *DEVtools User's Guide* for more information on subscreen ranges).

Values beyond these ranges will generate unpredictable results.

color is a pointer to a PMpixeltype structure whose red, green, blue and overlay components are written at (i,j) in *scrn*.

PMputpix returns a pointer to the next pixel on the given row $(i+1,j)$. This pointer may be used by **PMqput** for more efficient frame buffer access.

NOTES

Refer to **PMzbrk(3X)** for page register use.

SEE ALSO

PMgetpix(3X)
PMqput(3X)

NAME

PMputscan – write a scanline to a subscreen

SYNOPSIS

```
#include <pxm.h>

void PMputscan(scrn, buf, row, col, npix)
PMsubscrn *scrn;
PMpixeltype *buf;
short row, col, npix;
```

DESCRIPTION

PMputscan writes a row of *npix* pixels starting at (*col*,*row*) in subscreen *scrn* from the buffer *buf*.

scrn is a pointer to an initialized subscreen pointer. *col* and *row* are subscreen coordinates in the following legal ranges:

col [0,*PMimax*]

row [0,*PMjmax*]

PMimax and *PMjmax* are set to the appropriate value for the current model by system initialization (see the DEVtools *User's Guide* for more information on subscreen ranges).

Values beyond these ranges can generate unpredictable results.

SEE ALSO

PMgetscan(3X)

NAME

PMputzbuf – write a float value to the Z buffer

SYNOPSIS

```
#include <pxm.h>

float *PMputzbuf( scrn, i, j, zval )
PMsubscrn *scrn;
short i, j;
float zval;
```

DESCRIPTION

PMputzbuf writes a single value to Z buffer memory. *scrn* is a pointer to an initialized PMsubscrn structure corresponding to the subscreen from which the value is to be read.

i and *j* are subscreen coordinates with the following legal ranges:

i [0, PMimax]
j [0, PMjmax]

PMimax and **PMjmax** are set to the appropriate value for the current model by system initialization (see the *DEVtools User's Guide* for more information on subscreen ranges.

Values beyond these ranges will generate unpredictable results.

zval is a floating point value to be written at (*i*,*j*) in *scrn*.

PMputzbuf returns a pointer to the next Z value on the given row (*i*+1,*j*).

The pointer returned can be used directly (unlike the pointer returned from **PMputpix**), because Z buffer memory is fully mapped.

NOTES

Refer to **PMzbrk(3X)** for page register use.

The pointer returned can be cast to other types to allow Z memory to be used for *char*, *int*, and other data types.

EXAMPLE

```
ptr=PMputzbuf(scrn, i,j, zval);
*ptr++=zval;
*ptr++=zval;
```

SEE ALSO

PMgetzbuf(3X)
PMputpix(3X)
PMzput(3X)

NAME

PMqcopyztoz – copy from one section of DRAM to another

SYNOPSIS

```
#include <pxm.h>

void PMqcopyztoz(start_i, start_j, len_i, len_j, dest_i, dest_j)
int start_i;
int start_j;
int len_i;
int len_j;
int dest_i;
int dest_j;
```

DESCRIPTION

PMqcopyztoz copies a rectangular section of DRAM, *len_i* long words by *len_j* rows, from coordinates *start_i* and *start_j* to another section of DRAM buffer. *dest_i* and *dest_j* are the destination coordinates, and correspond to *start_i* and *start_j*, respectively. **PMqcopyztoz** is faster and takes less code space than **PMcopyztoz(3),** but cannot handle overlapping copies. While some overlapping copies may succeed, care should be taken so that the source area and destination areas of ZRAM are disjoint.

The *_i* arguments are in units of 4 byte long words, e.g., 1 byte for each of red, green, blue and overlay, or the size of one float. Thus, if *start_i* is set to 1, and *len_i* is set to 1, 4 bytes will be copied on each row, starting at an offset of 4 bytes from the beginning of the row. In the *_j* direction, one row corresponds to one row, with no multiplicative factors.

SEE ALSO

PMcopyztoz(3X)
PMcopyztov(3X)
PMcopyvtoz(3X)

NAME

PMqget – quick read of a pixel from the current buffer

SYNOPSIS

```
#include <pxm.h>

short *PMqget(color, ptr)
PMpixeltype *color;
short *ptr;
```

DESCRIPTION

PMqget reads a single pixel from the frame buffer. *ptr* is a pointer to the pixel location from which the pixel is to be read; *color* is a pointer to a PMpixeltype structure which is written with the pixel located at *ptr*.

PMqget returns a pointer to the next pixel on the given row. This value may be used in subsequent calls to **PMqget**.

NOTES

PMqget uses a pointer created by **PMgetpix, PMv0get** and other routines. **PMqget** uses the same page registers as the routine that generated the pointer. The user must ensure that the page registers are not corrupted while **PMqget** is in use.

Refer to **PMzbrk(3X)** for page register use.

SEE ALSO

PMgetpix(3X)
PMpixaddr(3X)
PMqput(3X)

NAME

PMqput - quick write of a pixel to the current buffer

SYNOPSIS

```
#include <pxm.h>

short *PMqput(color, ptr)
PMpixeltype *color;
short *ptr;
```

DESCRIPTION

PMqput writes a single pixel from the frame buffer. *ptr* is a pointer to the pixel location to which the pixel is to be written; *color* is a pointer to a PMpixeltype structure containing the pixel to be written at *ptr*.

PMqput returns a pointer to the next pixel on the given row. This value may be used in subsequent calls to **PMqput**.

NOTES

PMqput uses a pointer created by **PMputpix, PMv0put** and other routines. **PMqput** uses the same page registers as the routine that generated the pointer. The user must ensure that the page registers are not corrupted while **PMqput** is in use.

Refer to **PMzbrk(3X)** for page register use.

SEE ALSO

PMpixaddr(3X)
PMputpix(3X)
PMqget(3X)

NAME

PMrdyled - turn the PM_RDY LED on or off

SYNOPSIS

```
#include <pxm.h>

void PMrdyled(flag)
short flag;
```

DESCRIPTION

PMrdyled clears (if *flag* == 0) or sets (if *flag* != 0) the PM_RDY LED for this node.

NOTES

PMrdyled uses the PM_RDY hardware signal; thus **PMrdyled** and **PMvsync** should not be used in the same program.

SEE ALSO

PMflagled(3X)
PMvsync(3X)

NAME

PMrdyoff - turn the ready signal off

SYNOPSIS

void PMrdyoff()

DESCRIPTION

PMrdyoff turns off the DEV_FLAG signal used by **PMvsync**. It must be called some time after calling **PMvsync** and before another **PMvsync** is done.

NOTES

The purpose of separating **PMvsync** and **PMrdyoff** is to allow as much time as possible for user code after vertical retrace begins.

SEE ALSO

PMvsync(3X)

NAME

PMsetsem - set the semaphore

SYNOPSIS

void PMsetsem(value)
short value;

DESCRIPTION

PMsetsem waits for the software semaphore to be cleared by the host, then sets it to the passed *value*.

SEE ALSO

PMwaitsem(3N)

NAME

PMsin - trigonometric function

SYNOPSIS

#include <libmath.h>

float PMsin(theta)
float theta;

DESCRIPTION

PMsin returns the sine of *theta*.

theta must be in radians and be between -Pi/2 and +Pi/2.

NAME

PMsiodir – set serial I/O link direction

SYNOPSIS

```
#include <sysmsg.h>

void PMsiodir(dir)
short dir;
```

DESCRIPTION

PMsiodir sends a message to the host monitor process to set the serial I/O (SIO) link direction. *dir* must be one of:

```
PM_MSG_SERIAL_NORTH
PM_MSG_SERIAL_SOUTH
PM_MSG_SERIAL_EAST
PM_MSG_SERIAL_WEST
```

These constants are defined in **sysmsg.h.**

For it to work correctly, all the pixel nodes must call **PMsiodir**. **PMsiodir** calls **PMpsync** internally to synchronize before the host changes the link direction for all the pixel nodes.

NOTES

As with all other SIO functions, **PMsiodir** must only be called from pixel nodes.

SEE ALSO

PMmsg_exchange(3X)
PMmsg_setup(3X)
PMpsync(3X)
PMsioinit(3X)

NAME

PMsioinit - initialize serial I/O

SYNOPSIS

#include <pxm.h>

void PMsioinit()

DESCRIPTION

PMsioinit configures the serial I/O link for DMA input and polled output. It must be called only once before attempting to send messages over the serial links using **PMmsg_setup()** and **PMmsg_exchange()**.

SEE ALSO

PMmsg_setup(3X)
PMmsg_exchange(3X)
PMpsync(3X)

NAME

PMsnglbuff - disable double buffering mode

SYNOPSIS

#include <pxm.h>

void PMsnglbuff()

DESCRIPTION

PMsnglbuff disables double buffering and returns to single buffer mode. This means that all future updates using subscreen oriented functions (e.g., **PMputpix**()) will occur in the same buffer that is displayed.

PMsnglbuff only needs to be called after a call to **PMdblbuff** because it is the default mode at start up.

SEE ALSO

PMswapbuff(3X)
PMswapback(3X)
PMdblbuff(3X)

NAME

PMsqrt - sqare root function

SYNOPSIS

#include <libmath.h>

float PMsqrt(x)
float x;

DESCRIPTION

PMsqrt returns the square root of *x*. *x* must be >=0. This function is accurate to 6 significant digits.

NAME

PMswap_pipe – switch primary and alternate pipes of a dual pipe system
PMbus_wait – wait until control of the broadcast bus is granted

SYNOPSIS

#include <pxm.h>

void PMswap_pipe()
void PMbus_wait()

DESCRIPTION

On a dual pipe system, with the pipes operating in parallel mode, one pipe is the primary pipe and the other is the alternate pipe. **PMswap_pipe** reverses the functions of the two pipes. This is used to balance the load between the two pipes.

PMswap_pipe can only be called by the last node of a pipe board (node 8 or 17). When called, **PMswap_pipe** releases the broadcast bus to the alternate pipe It then requests the bus and waits for bus access to be granted.

PMbus_wait loops until control of the bus is granted. This is typically called during the initialization phase by the second pipe board, because initial control of the pipe is granted to the first pipe board.

SEE ALSO

DEVswap_pipe(3P)

NAME

PMswapback - swap meaning of back buffer

SYNOPSIS

#include <pxm.h>

void PMswapback( )

DESCRIPTION

PMswapback swaps the back and front buffer with respect to update, but does not change the visible buffer. In double buffer mode this means that the front buffer is also the update the buffer. In single buffer mode this means that the back buffer is updated. This functions will change the behavior of all functions that use a PMsubscrn argument to update the current buffer.

NOTES

PMswapback is implemented as a macro.

NAME

PMswapbuff – swap front and back pixel buffers

SYNOPSIS

void PMswapbuff()

DESCRIPTION

PMswapbuff exchanges the front (visible) and back pixel buffers; it should be called when a frame has been generated in the pixel buffer and must be displayed. **PMswapbuff** waits for vertical retrace by calling **PMvsync** before swapping and then calling **PMrdyoff**.

NOTES

Double buffering mode must be enabled with **PMdblbuff** before calling **PMswapbuff**.

SEE ALSO

PMdblbuff(3X)
PMswapback(3X)
PMsnglbuff(3X)
PMvsync(3X)
PMrdyoff(3X)

NAME

PMusermsg – send a *user message* to the host

SYNOPSIS

void PMusermsg(msg)
short msg;

DESCRIPTION

PMusermsg() sends a user defined opcode (a *user message*) to the host monitor process. *msg* must be a positive short int.

PMusermsg() checks the software semaphore to see if there was a previous **PMusermsg()** pending and, if necessary, waits. Otherwise, **PMusermsg()** returns immediately. If the message operation must complete before execution continues, **PMwaitsem()** should be called.

SEE ALSO

PMwaitsem(3N)
DEVtools*User's Guide* (section on user messages)

NAME

PMv0get – read a pixel from buffer 0

SYNOPSIS

```
#include <pxm.h>

short *PMv0get(i, j, color)
short i, j;
PMpixeltype *color;
```

DESCRIPTION

PMv0get() reads a single pixel from the frame buffer. Unlike **PMgetpix()**, the coordinate system used allows full access to frame buffer memory.

i and *j* are coordinates in the range [0, 255]. Values outside this range will generate unpredictable results.

color is a pointer to a PMpixeltype structure whose red, green, blue and overlay components are loaded with the pixel data contained at (*i*,*j*) in page 0 of pixel memory.

PMv0get() returns a pointer to the next pixel on the given row (*i*+1,*j*). This pointer can be used by **PMqget()** for more efficient frame buffer access.

NOTES

Refer to **PMzbrk(3X)** for page register use.

PMv0get() does not take into account subscreens or front and back buffers.

SEE ALSO

PMgetpix(3X)
PMqget(3X)
PMv0put(3X)
PMv1get(3X)

NAME

PMv0put – write a pixel to buffer 0

SYNOPSIS

```
#include <pxm.h>

short *PMv0put(i, j, color)
short i, j;
PMpixeltype *color;
```

DESCRIPTION

PMv0put() writes a single pixel to the frame buffer. Unlike **PMputpix()**, the coordinate system used allows full access to frame buffer memory.

i and *j* are coordinates in the range [0, 255]. Values outside this range will generate unpredictable results.

color is a pointer to a PMpixeltype structure which contains the red, green, blue and overlay components to be written at (i,j) in page 0 of pixel memory.

PMv0put() returns a pointer to the next pixel on the given row $(i+1,j)$. This pointer can be used by **PMqput()** for more efficient frame buffer access.

NOTES

Refer to **PMzbrk(3X)** for page register use.

PMv0put() does not take into account subscreens or front and back buffers.

SEE ALSO

PMputpix(3X)
PMqput(3X)
PMv0get(3X)
PMv1put(3X)

NAME

PMv1get – read a pixel from buffer 1

SYNOPSIS

```
#include <pxm.h>

short *PMv1get(i, j, color)
short i, j;
PMpixeltype *color;
```

DESCRIPTION

PMv1get() reads a single pixel from the frame buffer. Unlike **PMgetpix()**, the coordinate system used allows full access to frame buffer memory.

i and *j* are coordinates in the range [0, 255]. Values outside this range will generate unpredictable results.

color is a pointer to a PMpixeltype structure whose red, green, blue and overlay components will be loaded with the pixel data contained at (i,j) in page 1 of pixel memory.

PMv1get() returns a pointer to the next pixel on the given row $(i+1,j)$. This pointer can be used by **PMqget()** for more efficient frame buffer access.

NOTES

Refer to **PMzbrk(3X)** for page register use.

PMv1get() does not take into account subscreens or front and back buffers.

SEE ALSO

PMgetpix(3X)
PMqget(3X)
PMv1put(3X)
PMv0get(3X)

NAME

PMv1put - write a pixel to buffer 1

SYNOPSIS

```
#include <pxm.h>

short *PMv1put(i, j, color)
short i, j;
PMpixeltype *color;
```

DESCRIPTION

PMv1put() writes a single pixel to the frame buffer. Unlike **PMputpix()**, the coordinate system used allows full access to frame buffer memory.

i and *j* are coordinates in the range [0, 255]. Values outside this range will generate unpredictable results.

color is a pointer to a PMpixeltype structure which contains the red, green, blue and overlay components to be written at (*i*,*j*) in page 1 of pixel memory.

PMv1put() returns a pointer to the next pixel on the given row (*i*+1,*j*). This pointer can be used by **PMqput()** for more efficient frame buffer access.

NOTES

Refer to **PMzbrk(3X)** for page register use.

PMv1put() does not take into account subscreens or front and back buffers.

SEE ALSO

PMputpix(3X)
PMqput(3X)
PMv1get(3X)
PMv0put(3X)

NAME

PMvsync – synchronize and wait for vertical retrace

SYNOPSIS

void PMvsync()

DESCRIPTION

PMvsync is a video synchronization primitive. Once called, it will not return until both of the following conditions are true:

all pixel nodes have called **PMvsync**.

the vertical retrace period has begun.

NOTES

PMvsync uses the PM_RDY hardware signal; thus **PMrdyled** and **PMvsync** should not be used in the same program.

PMrdyoff *must* be called after calling **PMvsync** and before further calls to **PMvsync** are made. The purpose of separating **PMvsync** and **PMrdyoff** is to allow as much time as possible for user code after vertical retrace begins.

PMswapbuff uses **PMvsync** and **PMrdyoff** internally.

SEE ALSO

PMpsync(3X)
PMrdyoff(3X)
PMswapbuff(3X)

NAME

PMwaitsem – wait for semaphore to clear

SYNOPSIS

void PMwaitsem()

DESCRIPTION

PMwaitsem polls the software semaphore until it is cleared by the host. It can be used to synchronize with the host after calling **PMsetsem**, or to wait for the host to complete a *user message* or *system message* such as **printf**.

SEE ALSO

PMsetsem(3N)
PMusermsg(3N)
printf(3N)
DEVtools*User's Guide* (section on messages)

NAME

PMx_exp_n – integer power function

SYNOPSIS

```
#include <libmath.h>

float PMx_exp_n(x, n)
float    x;
short    n;
```

DESCRIPTION

PMx_exp_n returns the quantity *x* ^ *n*, where *n* is a positive integer between 1 and 20.

SEE ALSO

PMpow(3M)

NAME

PMxat - map subscreen coordinates to screen space

SYNOPSIS

```
#include <pxm.h>

float PMxat(scrn, i)
PMsubscrn *scrn;
short i;
```

DESCRIPTION

PMxat maps the subscreen coordinate i to the corresponding screen space x.

The mappings are:

$$x = i\ N_x + O_x$$

$$y = j\ N_y + O_y$$

where N_x and N_y are the numbers of processors in the x and y directions, respectively, and O_x and O_y are the x and y offsets into the processor array, respectively.

NOTES

PMxat is implemented as a macro.

SEE ALSO

PMmyx(3X)
PMmyy(3X)
PMyat(3X)

NAME

PMyat - map subscreen coordinates to screen space

SYNOPSIS

```
#include <pxm.h>

float PMyat(scrn, j)
PMsubscrn *scrn;
short j;
```

DESCRIPTION

PMyat maps the subscreen coordinate j to the corresponding screen space y.

The mappings are:

$$x = i\ N_x + O_x$$

$$y = j\ N_y + O_y$$

where N_x and N_y are the numbers of processors in the x and y directions, respectively, and O_x and O_y are the x and y offsets into the processor array, respectively.

NOTES

PMyat is implemented as a macro.

SEE ALSO

PMmyx(3X)
PMmyy(3X)
PMxat(3X)

NAME

PMzaddr – generate a ZRAM pointer to a row

SYNOPSIS

#include <pxm.h>

float *PMzaddr(i, j)
int i, j;

DESCRIPTION

PMzaddr() loads a page register with an appropriate descriptor, and then constructs a valid pointer that references that page register.

PMzaddr returns a pointer to the z value on the given row (*i,j*). Because the page register is loaded in fixed row addressing mode, the pointer can be used directly up to the end of the given row. To generate a column mode address use **PMzaddrcol()**.

i and *j* are coordinates in the range [0, 255]. Values outside this range will generate unpredictable results.

NOTES

Refer to **PMzbrk(3X)** for page register use.

PMzaddr() does not consider the PMsubscrn structure.

The pointer returned by this function can be cast to allow access to *char*, *int*, or other types of data stored in Z memory.

SEE ALSO

PMgetzbuf(3X)
PMzaddrcol(3X)
PMqzput(3X)
PMqzget(3X)
PMv0get(3X)

NAME

PMzaddrcol – generate a ZRAM pointer to a column

SYNOPSIS

```
#include <pxm.h>

float *PMzaddrcol(i, j)
int i, j;
```

DESCRIPTION

PMzaddrcol() loads a page register with an appropriate descriptor, and then constructs a valid pointer that references that page register.

PMzaddrcol() returns a pointer to the z value on the given row (*i,j*). Because the page register is loaded in fixed column addressing mode, the pointer can be used directly up to the end of the given column. To generate a row mode address use **PMzaddr()**.

i and *j* are coordinates in the range [0, 255]. Values outside this range will generate unpredictable results.

NOTES

Refer to **PMzbrk(3X)** for page register use.

PMzaddrcol() does not consider the PMsubscrn structure.

The pointer returned by this function can be cast to allow access to *char*, *int*, or other types of data stored in Z memory.

SEE ALSO

PMgetzbuf(3X)
PMzaddr(3X)
PMqzput(3X)
PMqzget(3X)
PMv0get(3X)

NAME

PMzbrk, PMblock_reg, PMavail_reg, PMset_lowreg, PMset_hireg - reserve DRAM and page registers for dynamic allocation

SYNOPSIS

```
#include <pxm.h>

PMzdesc PMzbrk(numblocks)
int numblocks;


#include <pageregs.h>

PMblock_reg(n)
int n;

PMavail_reg(n)
int n;

PMset_lowreg(n)
int n;

PMset_hireg(n)
int n;
```

DESCRIPTION

PMzbrk is the initialization call to create a list of memory resources for DRAM (also called ZRAM) that are used in subsequent calls to **PMgetzaddr(), PMgetzdesc()** and **PMfreezaddr()**. *numblocks* is the number of kilobytes (or rows) of DRAM to reserve and is in the range of 1 to 256 inclusive. The memory is reserved from the end of DRAM. For example, **PMzbrk(2)** reserves the last 2 rows of DRAM, rows 254 and 255.

The macros **PMblock_reg(), PMavail_reg(), PMset_lowreg()** and **PMset_hireg(),** defined in *pageregs.h,* are provided as a way of manipulating the list of page registers that are made available to access DRAM through calls to **PMgetzaddr()**. By default **PMzbrk** makes the page registers in the range 0 to 13 inclusive, available. These macros only have an affect when called after **PMzbrk.**

PMblock_reg() and **PMavail_reg()** are used to specify individual page registers to be excluded or included, respectively, from use by **PMgetzaddr.**

Another way to specify the page registers is to provide a range with calls to **PMset_lowreg()** and **PMset_hireg()**. The range is inclusive. For example, the calls

```
PMset_lowreg( 10 )
PMset_hireg( 13 )
```

indicate that the page registers 10, 11, 12 and 13 can be used by **PMgetzaddr()**. The low and high registers can be set in the range from 0 to 13, inclusive. Page registers 14 and 15 are reserved for use by the host.

It is necessary to block certain page registers to avoid conflicts when using other DEVtools functions such as **PMgetpix,** that use page registers internally. If a page register is no longer needed by a specific routine later on, it could be made available to **PMgetzaddr** with a call to **PMavail_reg()**.

Register assignments are given in the following table.

Page Register Assignments	
Page Register	Function
0	**PMgetscan(), PMputscan(), PMclear(), PMgetcol(), PMputcol(), PMgetrow(), PMputrow()**
1	**PMgetscan(), PMputscan(), PMclear(), PMgetcol(), PMputcol(), PMgetrow(), PMputrow()**
2	**PMv0get(), PMgetpix(), PMgetcol(), PMputcol(), PMgetrow(), PMputrow()**
3	**PMv0get(), PMgetpix(), PMgetcol(), PMputcol(), PMgetrow(), PMputrow()**
4	**PMv0put(), PMputpix()**
5	**PMv0put(), PMputpix()**
6	**PMv1get()**
7	**PMv1get()**
8	**PMv1put()**
9	**PMv1put()**
10	**PMpixaddr()**
11	**PMpixaddr()**
12	**PMzget(), PMgetzbuf(), PMzaddr()**
13	**PMzput(), PMputzbuf(), PMzaddrcol()**
14	Reserved for host use
15	Reserved for host use

NOTES

Requesting a number of blocks greater than 256 can cause **PMgetzdesc()** to fail in unpredictable ways.

SEE ALSO

PMgetzaddr(3X)
PMgetzdesc(3X)
PMfreezaddr(3X)

NAME

PMzget – read a float from the z buffer

SYNOPSIS

```
#include <pxm.h>

float *PMzget(i, j, zptr)
short i, j;
float *zptr;
```

DESCRIPTION

PMzget() reads a single z value from the Z buffer. Unlike **PMgetzbuf()**, the coordinate system used allows full access to z buffer memory.

i and *j* are coordinates in the range [0, 255]. Values outside this range will generate unpredictable results.

zptr is a pointer to a floating point variable that will be written with the z value contained at (*i*,*j*) in the z buffer.

PMzget() returns a pointer to the next z value on the given row (*i*+1,*j*). This pointer can be used by **PMqzget()** for more efficient frame buffer access. For even faster access, the pointer returned can be used directly (unlike the pointer returned from **PMv0get()**) because z buffer memory is fully mapped.

NOTES

Refer to **PMzbrk(3X)** for page register use.

PMzget() does not consider subscreens.

The pointer returned by this function can be cast to allow access to *char*, *int*, or other types of data stored in Z memory.

SEE ALSO

PMgetzbuf(3X)
PMqzget(3X)
PMv0get(3X)

NAME

PMzput – write a float to the Z-buffer

SYNOPSIS

```
#include <pxm.h>

float *PMzput(i, j, zval)
short i, j;
float zval;
```

DESCRIPTION

PMzput writes a single Z value to the Z buffer. Unlike **PMputzbuf**, the coordinate system used allows full access to Z buffer memory.

i and *j* are coordinates in the range [0, 255]. Values outside this range will generate unpredictable results.

zval is the floating point value to be written at (*i*,*j*) in the Z buffer.

PMzput returns a pointer to the next pixel on the given row (*i*+1,*j*). This pointer may be used by **PMqzput** for more efficient frame buffer access.

For even faster access the pointer returned can be used directly (unlike the pointer returned from **PMv0put**), since Z buffer memory is fully mapped.

NOTES

Refer to **PMzbrk(3X)** for page register use.

PMzput does not consider subscreens.

The pointer returned by this function can be cast to allow access to *char*, *int*, or other types of data stored in Z memory.

SEE ALSO

PMputpix(3X)
PMqzput(3X)
PMv0put(3X)

NAME

printf – formatted output conversion on host

SYNOPSIS

void printf(format [, arg] ...)
char *format;

DESCRIPTION

printf does formatted output in the same way as the UNIX system library **printf**. The full range of flags, widths, precisions, and format specifiers of the UNIX system **printf** is allowed. The actual printing is done by a host program calling **DEVpoll_nodes,** for example **devprint**, and is displayed on whatever device that process was invoked on.

It is possible to print a floating point number stored in host (IEEE) format with the new modifier, *i*, that is used in the same way as the *l* modifier of the UNIX system **printf**. However, because the DSP C compiler generates DSP DA move instructions that can destroy bits when passing an IEEE float to **printf**, the compiler must be fooled into thinking the float is a different data type of the appropriate size that will be passed with a bitwise copy. For example, to print a single host float:

```
printf("%if", *(long *)&host_float);
```

This technique must also be used whenever a float variable that is going to be passed to a function contains data that is not in DSP float format.

A new format specifier **%b** is also allowed. It formats the argument as an unsigned binary short integer. If **%lb** is specified, a 32-bit argument is assumed.

printf sends a *system message* and its arguments to the host and then returns. Some time later the host processes the format string and reads any pointer data from the nodes via DMA. The node program must therefore be careful not to modify any of the data or page registers associated with pointers in the **printf** argument list. To accomplish this **PMwaitsem** can be called right after **printf** to cause the node to wait until after the host has completely finished its **printf** processing.

NOTES

Up to 10 arguments of any scalar type may be given to **printf**. Using more than 10 arguments causes undefined behavior.

Because ints are 16-bits on the DSP32 and 32-bits on the host, the **l** modifier must be used when a 32-bit integer quantity is to be printed; for example, to print a float in hex format:

```
printf("%#lX", f);
```

SEE ALSO

devprint(1)
PMwaitsem(3N)
DEVpoll_nodes(3H)
printf() on host system

NAME

DEVimage_header - format of a DEVtools image file.

SYNOPSIS

```
#include <devimage.h>

typedef struct
{
        unsigned long   magic;                  /* magic number to indicate format */
        unsigned long   optional_header_size;   /* size of optional header */
        unsigned long   image_format;           /* how the pixels are stored */
        unsigned long   pixel_size;             /* number of bytes per pixel */
        unsigned long   storage_mode;           /* order of pixels in the file */
        unsigned long   pixels_per_line;        /* number of pixels per scan line */
        unsigned long   number_of_lines;        /* number of scan lines */
        unsigned long   x_offset;               /* initial X value */
        unsigned long   y_offset;               /* initial Y value */
} DEVimage_header;
```

DESCRIPTION

The **DEVimage_header** structure precedes all data in an image file and specifies information necessary to correctly display the image. **DEVimage_header** contains only a minimum amount of information about the image. It is assumed that the optional header that follows **DEVimage_header** will contain more specific information on the file's contents if necessary.

For portability reasons, each member of the structure is stored in the image file as an array of 8 decimal ACSII characters. The two routines **DEVget_image_header** and **DEVput_image_header** should be used to read/write and convert the image header from/to ASCII. Each of the members of the structure are explained in detail below.

The *magic* member of the structure contains a "magic number" indicating whether this file is in DEVtools image format or not. A value of DEV_IMAGE_MAGIC indicates that the file is in DEVtools image format, other values indicate that the format is *not* DEVtools image format.

The *optional_header_size* member gives the size of the optional header in bytes. The optional header is placed directly after the image header in the file. If the optional header is not present, this field is 0.

The *image_format* field tells how the pixel information is stored in the image file. Valid formats are:

```
#define DEV_USER_DEFINED            /* user defined image type */
#define DEV_RGBA_PACKED_PIXELS          /* RGBA order, 4 bytes per pixel */
#define DEV_RGB_PACKED_PIXELS               /* RGB order, 3 bytes per pixel */
#define DEV_MONO_PIXELS             /* one byte per pixel */
#define DEV_MONO_R_PIXELS           /* one red byte per pixel */
#define DEV_MONO_G_PIXELS           /* one green byte per pixel */
#define DEV_MONO_B_PIXELS           /* one blue byte per pixel */
#define DEV_MONO_A_PIXELS           /* one alpha byte per pixel */
#define DEV_MONO_16_PIXELS          /* 16 bit pixels */
#define DEV_DSP_FLOAT_PIXELS        /* 32 bit DSP floating point pixels */
#define DEV_IEEE_FLOAT_PIXELS       /* 32 bit IEEE floating point pixels */
#define DEV_RGB_PACKED_PIXELS       /* unpacked (16 bit components) RGB pixels */
#define DEV_RGBA_PACKED_PIXELS      /* unpacked (16 bit components) RGBA pixels */
#define DEV_RGB_PACKED_ENCODED_PIXELS       /* run-length encoded RBG pixels */
#define DEV_ABGR_PACKED_PIXELS      /* packed ABGR pixels */
```

```
#define DEV_RGB_ENCODED_PIXELS /* unpacked, run-length encoded RGB pixels */
```

The *pixel_size* field contains the number of bytes that make up a single pixel.

The *storage_mode* indicates the order in which the pixels are stored in the image. Valid values for *storage_mode* are:

DEV_ROW_MAJOR - pixels are stored by rows, that is in the order (0,0), (1,0), (2,0),...,(0,1), (1,1), ...

DEV_COLUMN_MAJOR - pixels are stored by columns, that is in the order (0,0), (0,1), (0,2),...,(1,0), (1,1), ...

The *pixels_per_line* member indicates the number of pixels per scan line (width) for this image.

The *number_of_lines* field indicates how many scan lines (height) are contained in this image.

The *x_offset* field stores the X value of the initial pixel.

The *y_offset* field stores the Y value of the initial pixel.

SEE ALSO

DEVget_image_header(3S)
DEVput_image_header(3S)
devsave(1)
devdisp(1)
picsave(1)
picdisp(1)
raydisp(1)
raysave(1)

NAME

PMcommand – data structure used for FIFO commands

SYNOPSIS

```
#include <pxm.h>

typedef struct {
        short   opcode;
        short   count;
        float   *data_ptr;
} PMcmdtype;

extern PMcmdtype PMcommand;
```

DESCRIPTION

Host programs usually operate on the Pixel Machine by sending data packets to the pipe nodes through the FIFOs. The pipe nodes may modify, delete, or pass on the command packets unmodified, or they may also generate new packets. The format of these data packets (called **commands**) is:

$$\text{OPCODE COUNT PARAM}_1 \ldots \text{PARAM}_{\text{count}}$$

where OPCODE and COUNT are 16–bit values, and each of the parameters in

$$\text{PARAM}_1 \ldots \text{PARAM}_{\text{count}}$$

is a 32–bit value.

The global data structure, PMcommand, defined in both the pipe and pixel nodes, reflects this packet structure. The members of this structure contain the following:

PMcommand.opcode: contains the opcode

PMcommand.count: contains the *negated* count of the number of bytes in the parameter list

PMcommand.data_ptr: points to a static buffer containing the parameters. It may be changed to point to a user-defined buffer.

Pipe node programs read a command from the input FIFO in two steps:

call **PMgetop** to load an opcode and count from the input FIFO

if parameter count is nonzero, call **PMgetdata** to load parameters from the input FIFO.

Pixel nodes read a command by calling **PMgetcmd**, which loads all three components of the command.

Pipe node programs may write a command to the output FIFO in two ways. First, by calling **PMputop** followed (if **count** is nonzero) by a call to **PMputdata**. Secondly, by calling **PMputcmd**, which combines the functionality of **PMputop** and **PMputdata**.

By changing members of the PMcommand structure, a pipe node program may modify the command stream as needed.

Pixel node programs read commands from the last pipe node but cannot write commands.

SEE ALSO

DEVwrite(3H)
PMgetcmd(3X)
PMgetdata(3P)

PMgetop(3P)
PMputcmd(3P)
PMputdata(3P)
PMputop(3P)

NAME

DEVpipe_read – reads a block of memory from a pipe DSP

SYNOPSIS

```
#include <host/devtools.h>

int DEVpipe_read(pixel_system, node, addr, buffer, nbytes)
DEVpixel_system *pixel_system;
int node;
DEVushort addr;
DEVbyte *buffer;
int nbytes;
```

DESCRIPTION

DEVpipe_read reads a block of memory from a pipe DSP. The data is retrieved from DSP memory using parallel DMA.

pixel_system is a pointer to the system descriptor, *node* is the number of the pipe node from which the data is to be read. *addr* is the location in the DSP address space that contains the data is be read. *buffer* points to the location into which the data is to be read. *nbytes* is the number of bytes of data to be read. *nbytes* should always be an even number. If *nbytes* is odd, *nbytes*+1 bytes of data will be read.

No byte order translation is performed. The data read will be in the same byte order as it is in the DSP memory.

DEVpipe_read uses parallel DMA I/O to transfer the data. As a result, the parallel control register is updated by this routine. The parallel communications modes are altered to:

enable DMA

set PAR to be autoincremented on DMA

set the interrupt vector to 16-bit mode

DEVpipe_read should always return zero.

NOTES

If *nbytes* is odd, **DEVpipe_read** will read *nbytes*+1 bytes of data and return -1 as its return value. The return value should be the number of bytes written, not zero.

NAME

DEVieee_dsp – convert from the host's floating–point format to the DSP32 floating point format

SYNOPSIS

#include <host/devtools.h>

DEVulong
DEVieee_dsp(f)
double f;

DESCRIPTION

The host and the DSP32 use different formats for floating point numbers. **DEVieee_dsp** converts a single floating point number in the IEEE format used by the host to a 32 bit floating point number in DSP32 format. The number to be converted is stored in *f*.

The value returned by **DEVieee_dsp** must be converted to the correct Pixel Machine byte order. This is done implicitly when the value is written to the pipe, but it must de done explicitly using **DEVbswapl()** or **DEVswap_long()** if the value is sent to the Pixel Machine in some other way (e.g., via DMA).

RETURNS

DEVieee_dsp returns a 32 bit number in the DSP32 floating point format.

NOTES

DSP floating point values should always be treated as unsigned long values on the host to prevent the compiler from performing undesired type–casting; for example, promotion to double when used as a function argument.

SEE ALSO

DEVdsp_ieee(3S)
DEVbswapl(3S)
DEVswap_long(3S)
DEVswap_short(3S)