Library Reference

Library Reference
Software Development Tool

- This product is sold on a membership agreement basis to Members of Net Yaroze, which is operated by Sony Computer Entertainment Inc.
- The  symbol, 'PlayStation' and 'Net Yaroze' are trademarks of Sony Computer Entertainment Inc.
- Company and product names recorded in/on this product are generally trademarks of each company. Note that in/on this product the symbols '® 'and 'TM' are not used explicitly.

Published February 1997

Written and produced by :
Sony Computer Entertainment Inc.
Akasaka Oji Building
8-1-22 Akasaka, Minato-ku, Tokyo, Japan 107
Enquiries to: Network Business Project
E-mail:ny-info@scei.co.jp
TEL:+81 (0) 3-3475-1711

Sony Computer Entertainment Europe
Waverley House
7-12 Noel Street
London W1V 4HH, England
Inquiries to: The Yaroze Team
E-mail: yaroze-info@scee.sony.co.uk
TEL:+44 (0) 171 447 1616 / +44 (0) 7000 YAROZE

Sony Computer Entertainment America
919 E. Hillsdale Blvd., 2nd Floor
Foster City, CA 94404, USA
Inquiries to: The Yaroze Team
E-mail: yaroze@interactive.sony.com
TEL:+1-415-655-3600

Table of Contents

About Net Yaroze

In order to get started with Net Yaroze, you should have ***C programming*** experience at some level of competence and you should be familiar with ***2D graphic creation/editing tools***. You should also have a basic understanding of ***3D modelling packages*** and ***sound creation/editing tools.*** Together these will help you get the most out of your Net Yaroze System.

The Net Yaroze Manual Set

There are three books in the Net Yaroze manual set.

1. *Start-Up Guide*

   An introductory booklet explaining the contents and requirements of the Net Yaroze Starter Kit. The *Start-Up Guide* gives step-by-step instructions on setting up the Net Yaroze software on your PC. It also explains how to run software on the Net Yaroze system.

2. *User's Guide*

   A reference manual explaining how to create software for the Net Yaroze system.

3. *Library Reference* (this document)

   A manual listing and describing the functions and structures in the Net Yaroze libraries.

1

Graphics Library

CVECTOR

8-bit 3D (color) vector

Structure

```
typedef struct {
                    unsigned char  r, g, b, cd;
} CVECTOR;
```

Members

r, *g*, *b*	Vector elements
cd	System reserved

Explanation

CVECTOR is used to define the structure of an 8-bit 3D (color) vector.

DISPENV

Display environment

Structure

```
typedef struct {
        RECT  disp;
        RECT  screen;
        unsigned char  isinter;
        unsigned char  isrgb24;
        unsigned char  pad0,pad1;
} DISPENV;
```

Members

disp Frame buffer display area.

disp defines the display area within the frame buffer.

The width of *disp* can be set to 256, 320, 360, 512 or 640.

The height of *disp* can be set to 240 or 480.

screen Output screen display area.

screen is calculated on the basis of a standard monitor screen and is independent of the value of *disp*. The standard monitor screen uses (0,0) for the coordinate of the upper left-hand corner and (256, 240) for the coordinate of the lower right-hand corner.

isinter Interlaced mode flag

- 0: Non-interlaced mode
- 1: Interlaced mode

isrgb24 24-bit mode flag

0: 16-bit mode

1: 24 bit mode

pad0, pad1 System reserved

Explanation

DISPENV is used to specify parameters for the display environment such as screen display mode and frame buffer display position.

DRAWENV

Drawing environment

Structure

```
typedef struct {
        RECT  clip;
        short  ofs[2];
        RECT  tw;
        unsigned short  tpage;
        unsigned char  dtd;
        unsigned char  dfe;
        unsigned char  isbg;
        unsigned char  r0, g0, b0;
        DR_ENV  dr_env;
} DRAWENV;
```

Members

clip Drawing area.

clip specifies the rectangular area used for drawing. Drawing will not be performed outside the area specified by *clip*.

ofs Offset.

The values (ofs[0], ofs[1]) are used as offsets when calculating the address that is used for drawing in the frame buffer. The offsets are added to the values of all coordinates to determine the address used by drawing commands when drawing in the frame buffer.

tw Texture window.

tw specifies the rectangular area within the texture page that is used repeatedly for applying textures.

tpage Texture page initial value.

dtd Dither treatment flag

0: Dithering OFF.

1: Dithering ON.

Dfe Display area drawing flag

0: Drawing to the display area is disabled.

1: Drawing to the display area is enabled.

isbg Clear drawing area flag

0: The drawing area is not cleared when the drawing environment is established.

1: The entire clipped area is painted with the brightness values (*r0*,*g0*,*b0*) when the drawing environment is established.

r0,*g0*,*b0* Background color. Valid only when *isbg* = 1.

dr_env System reserved.

Explanation

DRAWENV sets basic parameters related to drawing such as the drawing offset and the drawing clip area.

Notes

Drawing can be performed in the region (0, 0)-(1023, 511) within the drawing space.

Offsets and addresses to which the offset has been added are wrapped around at (-1024, -1024)-(1023, 1023).

Values that can be specified for the texture window are limited to the combinations shown in the following table.

tw.w	0(=256)	16	32	64	128
tw.x	0	multiple of 16	multiple of 32	multiple of 64	multiple of 128
tw.h	0(=256)	16	32	64	128
tw.y	0	multiple of 16	multiple of 32	multiple of 64	multiple of 128

GsBG

BG (background picture) handler

Structure

```
struct GsBG {
        unsigned long  attribute;
        short  x,y;
        short  w,h;
        short  scrollx,scrolly;
        unsigned char  r,g,b;
        GsMAP  *map;
        short  mx,my;
        short  scalex,scaley;
        long  rotate;
};
```

Members

attribute	32-bit attribute (see the description in GsSPRITE)
x, *y*	Display position of the upper left-hand point
w, *h*	BG display size (in pixels)
scrollx, *scrolly*	x,y scroll values
r, *g*, *b*	Display brightness is set in r, g, b. (Normal brightness is 128.)
map	Pointer to map data
mx, *my*	Rotation and enlargement central point coordinates
scalex, *scaley*	Scale values in x and y directions

rotate Rotation angle (4096 = 1 degree)

Explanation

GsBG is a structure used to describe a background. A background (BG) consists of a large rectangle drawn from GsMAP data based upon a combination of small rectangles defined by GsCELL data. Every background has an associated GsBG structure. The background may be manipulated via this structure.

To enter a GsBG object in an ordering table, use GsSortBg().

x, y specify the screen display position.

w, h specify the BG display size in pixels, and are not dependent on cell size or map size.

If the display area is larger than the map, the content of the map is repeatedly displayed. (this is referred to as tiling)

scrollx, scrolly specify the offset from the map display position in dots.

r, g, b specify the brightness values for red, green, and blue. The allowed range is 0 to 255. 128 is the brightness of the original pattern; 255 represents a two-fold increase in brightness.

map is a pointer to the start of the map data. Map data is represented by the GsMAP structure.

mx, *my* specify the center of rotation and scaling as relative coordinates. The upper left-hand point of the BG is the point of origin. For example, if rotation is around the center of the BG, these values should be specified as w/2 and h/2, respectively.

scalex, *scaley* specify enlargement/reduction values in the x and y directions. These values represent units of 4096 (which is equivalent to a fixed point value of 1.0). *scalex* and *scaley* can be set up to 8 times the original size.

rotate specifies the amount of rotation around the Z-axis.

GsBOXF

Rectangle handler

Structure

```
struct GsBOXF {
        unsigned long  attribute;
        short  x,y;
        unsigned short  w,h;
        unsigned char  r,g,b;
};
```

Members

attribute	Attribute bits as shown below. (Bits not defined below are reserved by the system)
	28-29: Semi-transparency rate
	0: 0.5 x Back + 0.5 x Forward
	1: 1.0 x Back + 1.0 x Forward
	2: 1.0 x Back - 1.0 x Forward
	3: 1.0 x Back + 0.25 x Forward
	30: Semi-transparency mode
	0: Semi-transparency OFF
	1: Semi-transparency ON
	31: Display mode
	0: Enable display 1: Display display
x, y	Display position (upper left-hand point)
w, h	Size of rectangle (width, height)

r, g, b	Drawing color

Explanation

GsBOXF is a structure used to represent a rectangle drawn in a single color. The GsSortBoxFill() function can be used to enter a GsBOXF object in the ordering table.

GsCELL

Cells which define a BG

Structure

```
struct GsCELL {
                    unsigned char  u, v;
                    unsigned short  cba;
                    unsigned short  flag;
                    unsigned short  tpage;
};
```

Members

u	Offset (X-direction) within the page
v	Offset (Y-direction) within the page
cba	CLUT ID
flag	Inversion control flag
tpage	Texture page number

Explanation

A rectangular array of GsCell structures is used to describe individual cells that fit together to create a background (BG). Each individual GsCell structure defines a rectangular portion of the overall BG.

cba displays the position within the frame buffer of a CLUT corresponding to the cell, and has the following meaning.

Bit	Value
Bit 0~5	X position of CLUT/16
Bit 6~15	Y position of CLUT

tpage is a page number that indicates the position of a sprite pattern within the frame buffer.

The *u* and *v* parameters specify the offset position for the sprite pattern within the texture page defined by *tpage*.

flag controls whether the image is flipped during drawing. The meaning of each bit is as shown below.

Bit	Value
Bit 0	Vertical flip (0 = no flip, 1 = flip)
Bit 1	Horizontal flip (0 = no flip, 1 = flip)
Bit 2~15	Reserved

GsCOORDINATE2

Matrix-type coordinate system

Structure

```
struct GsCOORDINATE2 {
                    unsigned long  flg;
                    MATRIX coord;
                    MATRIX workm;
                    GsCOORD2PARM  *param;
                    GsCOORDINATE2  *super;
};
```

Members

flg	Flag indicating whether coord was rewritten
coord	The matrix
workm	Result of multiplying this coordinate system with the WORLD coordinate system
param	Pointer for scaling, rotation and translation parameters
super	Pointer to the parent coordinate system

Explanation

GsCOORDINATE2 is defined by the matrix *coord* and a pointer to the parent coordinate system.

workm retains the result of matrix multiplication in each node of GsCOORDINATE2 from WORLD coordinates and performed by the GsGetLw() and GsGetLs() functions. However, *workm* does not store the result of matrix multiplication for a coordinate system that is directly coupled to the WORLD coordinate system.

flg is used during GsGetLw() and GsGetLs() to omit calculations for a node when calculations were already performed. When *flg* is set to 1, calculations are omitted. When *flg* is set to 0, calculations are performed. The programmer must clear this flag whenever *coord* is changed otherwise the GsGetLw() and GsGetLs() functions will not execute properly.

GsDOBJ2

Object structure definition used by the GsCOORDINATE2 3D object handler

Structure

```
struct GsDOBJ2 {
        unsigned long  attribute;
        GsCOORDINATE2  *coord2;
        unsigned long  *tmd;
        unsigned long  id;
};
```

Members

attribute	Object attribute (32-bits)
coord2	Pointer to a local coordinate system
tmd	Pointer to model data
id	System reserved

Explanation

GsDOBJ2 structures can be used to manipulate 3D models. Each object in a 3D model has an associated GsDOBJ2 structure.

The GsDOBJ2 structure can be linked to TMD file model data using GsLinkObject4().The GsDOBJ2 structure can be registered in the ordering table using GsSortObject4().

The *coord2* parameter is a pointer to a GsCOORDINATE2 structure defining the object's coordinate system. The location, inclination, and size of the object are defined in a matrix in this structure.

tmd contains the starting address of TMD model data stored in memory. *tmd* is calculated and set using GsLinkObject4().

attribute is 32 bits and is used to store various display attributes as shown below. All other bits not defined below are reserved by the system.

(a) Bit 3: Light source calculation mode

This bit controls whether light source calculation will be performed using FOG.

0: Normal mode without FOG. This is the fastest mode requiring the least amount of calculation time.

1: FOG mode. FOG attributes are set in the GsFOGPARAM structure and updated using the GsSetFogParam() function.

(b) Bit 5: Light source calculation mode control bit

This bit controls whether light source calculation mode is determined by bit 3 of the attribute field or by the lighting mode set by GsSetLightMode().

0: Lighting mode is set by GsSetLightMode(). Bit 3 of the attribute field is ignored.

1: Lighting mode is set according to bit 3 of the attribute field.

(b) Bit 6: Light source calculation ON/OFF switch

This bit is used to switch light source calculation on and off.

When light source calculation is turned off, a texture-mapped polygon is displayed in the original texture color. An unmapped polygon is displayed in the model data color.

(c) Bit 30: Semi-transparency ON/OFF

This bit is used to switch semi-transparency on and off.

This bit must be used with the uppermost bit (STP bit) of the texture color field to enable semi-transparency. (The texture color field is the texture pattern when direct mode is set and the CLUT color field when indexed mode is set). The semi-transparency and non-transparency of each pixel may also be controlled using the STP bit.

(d) Bit 31: Display ON/OFF

This bit is used to switch the display on and off.

(e) Bits 9-11: Automatic partitioning (polygon subdivision)

0: No automatic partitioning

1: 2x2 partitioning

2: 4x4 partitioning

3: 8x8 partitioning

4: 16x16 partitioning

5: 32x32 partitioning

These bits specify the number of subdivisions for the automatic partitioning function. Automatic partitioning, or polygon subdivision, causes all the polygons contained within an object to subdivide. It is used for reducing texture distortion and preventing fragmentation of adjacent polygons. Note that division should be kept to a minimum in order to prevent the number of polygons from increasing exponentially.

GsFOGPARAM

Fog (depth cue) information

Structure

```
struct GsFOGPARAM {
                    short  dqa;
                    long  dqb;
                    unsigned char  rfc, gfc, bfc;
};
```

Members

dqa	Parameter specifying the degree of blending with respect to depth
dqb	Parameter specifying the degree of blending with respect to depth
rfc, gfc, bfc	Background colors

Explanation

GsFOGPARAM describes fog (depth cue) information

dqa and *dqb* are background color attenuation coefficients. They can be calculated using the following formulas:

$$dqa = -df * 4096/64/h$$

$$dqb = 1.25 * 4096 * 4096$$

df is the distance where the attenuation coefficient is 1; that is, the distance from the viewpoint to the point where the background colors are completely blended.

h is the distance from the viewpoint to the screen and is also known as the projection distance.

GsF_LIGHT

Parallel light source

Structure

```
struct GsF_LIGHT {
                    long  vx, vy, vz;
                    unsigned char  r, g, b;
};
```

Members

vx, *vy*, *vz*	Light source directional vectors
r, *g*, *b*	Light colors

Explanation

GsF_LIGHT describes parallel light source information. It is set by the GsSetFlatLight() function.

Up to three parallel light sources can be set simultaneously.

The light source directional vector is specified by *vx*, *vy*, *vz*. It is unnecessary for the programmer to perform normalization as this is done by the GsSetFlatLight() function.

A polygon whose normal vectors are opposite to these directional vectors is exposed to the strongest light.

Light source colors are specified in the 8-bit values *r*, *g*, *b*.

GsGLINE

Straight line handler with gradation

Structure

```
struct GsGLINE {
        unsigned long  attribute;
        short  x0,y0;
        short  x1,y1;
        unsigned char  r0,g0,b0;
        unsigned char  r1,g1,b1;
};
```

Members

attribute Attribute bits as shown below. (Bits not defined below are reserved by the system)

28-29: Semi-transparency rate

0:	0.5 x Back + 0.5 x Forward
1:	1.0 x Back + 1.0 x Forward
2:	1.0 x Back - 1.0 x Forward
3:	1.0 x Back + 0.25 x Forward

30: Semi-transparency mode

0:	Semi-transparency OFF
1:	Semi-transparency ON

31: Display mode

0:	Enable display
1:	Disable display

x0,y0 Drawing starting point position

x1, *y1*	Drawing ending point position
r0, *g0*, *b0*	Drawing color of the starting point
r1, *g1*, *b1*	Drawing color of the ending point

Explanation

GsGLINE is a structure used to represent straight lines with gradation. It is the same as GsLINE except that drawing colors for the starting point and ending point may be specified separately.

GsIMAGE

Image data structure information

Structure

```
struct GsIMAGE {
        short  pmode;
        short  px, py;
        unsigned short  pw, ph;
        unsigned long  *pixel;
        short  cx, cy;
        unsigned short  cw, ch;
        unsigned long  *clut;
}
```

Members

pmode	Pixel mode	
	0:	4-bit CLUT
	1:	8-bit CLUT
	2:	16-bit Direct
	3:	24-bit Direct
	4:	Multiple mode combinations
px, *py*	Pixel data storage location within the frame buffer	
pw, *ph*	Pixel data width and height	
pixel	Pointer to pixel data	
cx, *cy*	CLUT data storage location within the frame buffer	
cw, *ch*	CLUT data width and height	

clut Pointer to CLUT data

Explanation

GsIMAGE is loaded with TIM format data information using the GsGetTimInfo() function.

For information on the TIM file format, please refer to the Net Yaroze Members' Web site.

GsLINE

Straight line handler

Structure

```
struct GsLINE {
        unsigned long  attribute;
        short  x0,y0;
        short  x1,y1;
        unsigned char  r,g,b;
};
```

Members

attribute Attribute bits as shown below. (Bits not defined below are reserved by the system)

28-29: Semi-transparency rate

- 0: 0.5 x Back + 0.5 x Forward
- 1: 1.0 x Back + 1.0 x Forward
- 2: 1.0 x Back - 1.0 x Forward
- 3: 1.0 x Back + 0.25 x Forward

30: Semi-transparency mode

- 0: Semi-transparency OFF
- 1: Semi-transparency ON

31: Display mode

- 0: Enable display
- 1: Disable display

x0,y0 Drawing starting point position

x1,y1 Drawing ending point position

r, *g*, *b*	Drawing color

Explanation

GsLINE is a structure that represents straight lines. Use GsSortLine() to register a GsLINE in the ordering table.

attribute is 32 bits, and sets various attributes for display as shown below.

(a) Semi-transparency rate (bit 28-29)

If semi-transparency is turned on using bit 30, bits 28 and 29 are used to set the pixel blending method. Normal semi-transparency processing is performed when these bits are set to 0, pixel addition when set to 1, pixel subtraction when set to 2, and 25% addition when set to 3.

(b) Semi-transparency mode (bit 30)

Turns semi-transparency ON/OFF

(c) Display mode (bit 31)

Turns the display ON/OFF

GsMAP

BG composition MAP

Structure

```
struct GsMAP {
        unsigned char  cellw, cellh;
        unsigned short  ncellw, ncellh;
        GsCELL  *base;
        unsigned short  *index;
};
```

Members

cellw, *cellh*	Cell size (0 is the same as 256)
ncellw, *ncellh*	Size of the background (BG) in cells
base	Pointer to the GsCELL structure array
index	Pointer to cell information

Explanation

The GsMAP structure contains map data that is used to create a background from GsCELL data. Map data is managed by cell index array information.

cellw, *cellh* specify the size of one cell in pixels. Note that one BG is made up of cells of the same size.

ncellw and *ncellh* specify the size of the BG map in units of cells.

base specifies the starting address of the GsCELL array.

index specifies the starting address of the cell data table. Cell data is a list of index values whose size is equivalent to (*ncellw* * *ncellh*) for the array specified by *base*. If a cell value is 0xFFFF it indicates a NULL (transparent) cell.

GsOT

Ordering table header

Structure

```
struct GsOT {
                    unsigned short  length;
                    GsOT_TAG  *org;
                    unsigned short  offset;
                    unsigned short  point;
                    GsOT_TAG  *tag;
};
```

Members

length	Bit length of the ordering table (OT)
org	Pointer to start address of GsOT_TAG table
offset	OT screen coordinate system Z-axis offset
point	OT screen coordinate system Z-axis typical value
tag	Pointer to current GsOT_TAG

Explanation

The GsOT structure describes the header of the ordering table. This header has pointers to the actual ordering table array, specified by the *org* and *tag* members. These members are initialized using the GsClearOT() function.

The *org* member always points to the start of the ordering table. The *tag* field points to the element within the ordering table at which drawing will take place.

The *length* field indicates the size of the ordering table. It is a value from 1-14 where the actual ordering table size is 2**length (i.e. a value of 14 indicates an array of 16384 GsOT_TAG items, while a value of 8 indicates an array of 256 GsOT_TAG items).

org points to a GsOT_TAG array running from 0-1. If the value "14" is specified, *org* points to a GsOT_TAG array running from 0-16384.

The GsClearOt() function initializes memory from *org* up through the size indicated by *length*. Note that memory will be destroyed if the size of the GsOT_TAG array pointed to by *org* is greater than that specified by *length*.

point is used by the GsSortOt() function in the sorting of ordering tables.

The ordering table Z-axis offset is specified by *offset*. For example, if *offset* = 256, the start of the ordering table is Z = 256. (Not yet supported.)

Notes

The *length* and *org* values should be set first. The other members are set by the GsClearOt() function.

See also

GsClearOt(), GsDrawOt(), GsSortOt()`

GsOT_TAG

Ordering table element

Structure

```
struct GsOT_TAG {
                        unsigned  p : 24;
                        unsigned char  num : 8;
};
```

Members

p	Pointer to next item in ordering table list
num	Number of words in current GPU packet (i.e. primitive)

Explanation

An ordering table is a linked list of GsOT_TAG structures and various types of GPU primitive structures.

The *p* field of a GsOT_TAG structures indicates the least significant 24-bits of a pointer to the next item in the list. A value of 0xFFFFFF indicates the end of the list.

The GsOT structure is used to manage an array of GsOT_TAG structures. Memory should be allocated for the GsOT_TAG structures after the GsOT structure is initialized.

GsRVIEW2

Viewpoint position (REFERENCE-type)

Structure

```
struct GsRVIEW2 {
                    long  vpx, vpy, vpz;
                    long  vpx, vpy, vpz;
                    long  rz;
                    GsCOORDINATE2  *super
};
```

Members

vpx, *vpy*, *vpz*	Viewpoint coordinates
vrx, *vry*, *vrz*	Reference point coordinates
rz	Viewpoint twist
super	Pointer to the coordinate system that sets the viewpoint (GsCOORDINATE2 type)

Explanation

GsVIEW2 holds information about the viewpoint. It is set by the GsSetRefView2() function.

vpx, *vpy*, *vpz* specify the viewpoint coordinates in the coordinate system specified by *super*.

vrx, *vry*, *vrz* specify the reference coordinates in the coordinate system specified by *super*.

When the Z-axis is a vector from the viewpoint to the reference point, *rz* specifies the viewpoint twist, or screen inclination against the Z-axis. It is represented as an integer, with 4096 equivalent to one degree.

The viewpoint and reference point coordinate systems are set in *super*.

As an example of using this function, an airplane cockpit view can be realized simply by setting *super* to the airplane coordinate system.

GsSPRITE

Sprite handler

Structure

```
struct GsSPRITE {
        unsigned long  attribute;
        short  x,y;
        unsigned short  w,h;
        unsigned short  tpage;
        unsigned char  u,v;
        short  cx,cy;
        unsigned char  r,g,b;
        short  mx,my;
        short  scalex,scaley;
        long  rotate;
};
```

Members

attribute	32-bit attribute field (explanation is given below)
x, *y*	Display position of the upper left-hand point
w, *h*	Width and height of the sprite (Not displayed if w or h is 0.)
tpage	Sprite pattern texture page number
u, *v*	Sprite pattern offset within the page
cx, *cy*	Sprite CLUT address*r*,
g, *b*	Display brightness for r, g, and b (normal brightness is 128)
mx, *my*	Rotation and enlargement central point coordinates

scalex, *scaley*	Scale values in x and y directions
rotate	Rotation angle (4096 = 1 degree)
attribute bits	(Bits not defined below are reserved by the system)

6: Brightness adjustment switch

0:	ON
1:	OFF

24-25: Sprite pattern color mode

0:	4-bit CLUT
1:	8-bit CLUT
2:	15-bit Direct

27: Rotation & scaling functions

0:	ON
1:	OFF

28-29: Semi-transparency rate

0:	0.5 x Back + 0.5 x Forward
1:	1.0 x Back + 1.0 x Forward
2:	1.0 x Back - 1.0 x Forward
3:	1.0 x Back + 0.25 x Forward

30: Semi-transparency mode

0:	Semi-transparency OFF
1:	Semi-transparency ON

31: Sprite display mode

0:	Displayed
1:	Not displayed

Explanation

GsSPRITE is a structure used to display a sprite. This structure makes it possible to manipulate each sprite via its parameters.

To enter a GsSPRITE in an ordering table, use GsSortSprite() or GsSortFastSprite().

x, *y* specify the screen display position. *mx*, *my* specify the point in the sprite pattern used as the display position in GsSortSprite(); in GsSortFastSprite(), the point at the upper left-hand corner of the sprite is used as the display position.

w, *h* specify the width and height of the sprite in pixels.

tpage specifies the texture page number (0-31) of the sprite pattern.

u, *v* specify the offset within the page from the upper left-hand point of the sprite pattern. The range that may be specified is (0, 0) - (255, 255).

cx, *cy* specify the starting position of the CLUT (color palette) as a VRAM address. (Valid for 4-bit/8-bit mode only.)

r, *g*, *b* specify the brightness values for red, green, and blue. The allowed range is 0 to 255. 128 is the brightness of the original pattern; 255 represents a two-fold increase in brightness.

mx, *my* specify the coordinates for the center of rotation and scaling. The upper left-hand point of the sprite is the point of origin. For example, if rotation is around the center of the sprite, *mx* and *my* should be specified as w/2 and h/2, respectively.

scalex, *scaley* specify enlargement/reduction values in the x and y directions. These values represent units of 4096 (which is equivalent to a fixed point value of 1.0). *scalex* and *scaley* can be set up to 8 times the original size.

rotate specifies the amount of rotation around the Z-axis and is represented as an integer, where 4096 is equivalent to 1 degree.

attribute is 32 bits, and sets various attributes for display. An explanation of each bit follows. Bits not defined below are reserved by the system.

(a) Brightness adjustment switch (bit 6)

This bit sets whether or not the sprite pattern pixel colors are to be drawn with brightness adjusted according to the *(r,g,b)* values. When this bit is 1, brightness is not adjusted and the *(r,g,b)* values are ignored.

(b) Sprite pattern color mode (bit 24-25)

A sprite pattern can use either 4-bit color mode or 8-bit color mode, both of which use the color table, and 15-bit mode, which directly displays colors. These bits are used to set one of these color modes. Please see above for bit definitions.

(c) Rotation & scaling functions (bit 27)

This bit turns on or off the sprite scaling function. If rotation or scaling of the sprite is not needed, this bit should be set to OFF for high speed processing.

GsSortFastSprite() ignores this bit and always set the scaling function to OFF.

(d) Semi-transparency rate (bit 28-29)

These bits set the method of pixel blending together with bit 30, when semi-transparency is turned on. Normal semi-transparent processing is performed when these bits are set to 0, pixel addition when set to 1, pixel subtraction when set to 2, and 25% addition when set to 3.

(e) Semi-transparency mode (bit 30)

This bit turns semi-transparency on or off. It must be used with the uppermost bit (STP bit) of the texture color field to turn semi-transparency on (texture pattern when in direct mode and CLUT color field when in indexed mode). The semi-transparency and non-transparency of each pixel may also be controlled using this STP bit.

(f) Sprite display mode (bit 31)

This bit turns the display mode on and off.

GsVIEW2

Viewpoint position (Matrix-type)

Structure

```
struct GsVIEW2 {
                    MATRIX view;
                    GsCOORDINATE2 *super
};
```

Members

view	Matrix for conversion from parent coordinates to viewpoint coordinates
super	Pointer to the coordinate system that sets the viewpoint

Explanation

GsVIEW2 sets the viewpoint coordinate system. It directly specifies the matrix for converting from the parent coordinate system to the viewpoint coordinate system in *view*.

The function that sets GsVIEW2 is GsSetView2().

MATRIX

3 x 3 matrix

Structure

```
typedef struct  {
                        short  m[3][3];
                        long   t[3];
} MATRIX;
```

Members

m	Element values of the 3 x 3 matrix
t	Amount of translation

Explanation

MATRIX defines a 3 x 3 matrix. The value of each element is specified by *m[i][j]*.

t[i] specifies the amount of translation after conversion.

RECT

Frame buffer rectangular area

Structure

```
typedef struct {
        short x, y;
        short w, h;
} RECT;
```

Members

x, *y*	Coordinates for the upper left-hand corner of the rectangular area
w, *h*	Width and height of the rectangular area

Explanation

RECT is used to specify a rectangular area in the frame buffer. Negative values or values that exceed the size of the frame buffer (1024 x 512) cannot be used.

SVECTOR

16-bit 3D vector

Structure

```
typedef struct {
                        short  vx, vy;
                        short  vz, pad;
} SVECTOR;
```

Members

vx, *vy*, *vz*	Vector elements
pad	System reserved

Explanation

SVECTOR is used to define the structure of a 16-bit 3D vector.

VECTOR

32-bit 3D vector

Structure

```
typedef struct {
                    long  vx, vy;
                    long  vz, pad;
} VECTOR;
```

Members

vx, *vy*, *vz*	Vector elements
pad	System reserved

Explanation

VECTOR is used to define the structure of a 32-bit 3D vector.

ApplyMatrix

Multiplies a vector by a matrix

Syntax

```
VECTOR* ApplyMatrix (
MATRIX *m,
SVECTOR *v0,
VECTOR *v1
)
```

Arguments

m	Pointer to input matrix
v0	Pointer to input short vector
v1	Pointer to result vector

Explanation

The short vector *v0* is multiplied by the matrix *m* beginning with the rightmost end.

The product is stored in vector *v1*.

The argument format is as follows:

m->*m*[i][j] : (1,3,12)

v0->vx,vy,vz :(1,15,0)

v1->vx,vy,vz :(1,31,0)

Return Value

The vector product *v1* is returned.

Notes

This function destroys the rotation matrix.

ApplyMatrixLV

Multiplies a vector by a matrix

Syntax

VECTOR* **ApplyMatrixLV** (
MATRIX **m*,
VECTOR **v0*,
VECTOR **v1*
)

Arguments

m	Pointer to input matrix
v0	Pointer to input vector
v1	Pointer to result vector

Explanation

The vector *v0* is multiplied by the matrix *m* beginning with the rightmost end. The product is stored in vector *v1*.

The argument format is as follows:

m->*m*[i][j] : (1,3,12)
v0->vx,vy,vz :(1,31,0)
v1->vx,vy,vz :(1,31,0)

Return Value

The vector product *v1* is returned.

Notes

This function destroys the rotation matrix.

ApplyMatrixSV

Multiplies a vector by a matrix

Syntax

SVECTOR* **ApplyMatrixSV** (
MATRIX **m*,
SVECTOR **v0*,
SVECTOR **v1*
)

Arguments

m	Pointer to input matrix
v0	Pointer to input short vector
v1	Pointer to result short vector

Explanation

The short vector *v0* is multiplied by the matrix *m* beginning with the rightmost end.

The product is stored in short vector *v1*.

The argument format is as follows:

m->*m*[i][j] : (1,3,12)

v0->vx,vy,vz :(1,15,0)

v1->vx,vy,vz :(1,15,0)

Return Value

The vector product *v1* is returned.

Notes

This function destroys the rotation matrix.

ClearImage

Fills the frame buffer with a specified pixel value

Syntax

int **ClearImage** (
RECT **recp*,
u_char *r*,
u_char *g*,
u_char *b*
)

Arguments

recp	Pointer to the rectangular area to be cleared in the frame buffer
r, *g*, *b*	Pixel value to be used for clearing

Explanation

The rectangular area of the frame buffer specified by *recp* is cleared with the pixel value specified by (*r,g,b*).

Return Value

The slot number of the GPU's draw/command queue where the command resides is returned.

Notes

MoveImage() is a non-blocking function, therefore DrawSync() must be used to determine the completion of transfer.
The transfer is not affected by clipping or offsetting that may be in effect for the drawing environment.

CompMatrix

Performs composite coordinate transformation

Syntax

MATRIX***CompMatrix** (

MATRIX**m0*,

MATRIX**m1*,

MATRIX**m2*

)

Arguments

m0	Pointer to input matrix
m1	Pointer to input matrix
m2	Pointer to result matrix

Explanation

The composite coordinate transformation of matrix *m0* and matrix *m1* (including translation) is performed. The result is placed in matrix *m2*.

The following algorithm is used.

[*m2*->m] = [*m0*->m] * [*m1*->m]

(*m2*->t) = [*m0*->m] * (*m1*->t) + (*m0*->t)

However the values of the *m1*->*t* elements must be within the range of $\left[-2^{15}, 2^{15}\right)$.

The argument format is as follows::

m0->*m*[i][j] : (1,3,12)

m0->t[i]: (1,31,0)

m1->m[i][j] : (1,3,12)

m1->t[i]: (1,15,0)

m2->m[i][j] : (1,3,12)

m2->t[i]: (1,31,0)

Return Value

The result matrix *m2* is returned.

Notes

This function destroys the rotation matrix.

DrawSync

Waits for completion of all drawing

Syntax

int **DrawSync** (
int *mode*
)

Arguments

mode	0:	Waits for completion of all non-blocking functions registered in the GPU's command/execution queue.
	1:	The current slot number of the GPU's command/execution queue is returned.

Explanation

DrawSync waits until all drawing has completed.

Return Value

The current slot number of the GPU's command/execution queue is returned.

FntFlush

Draws the contents of a print stream

Syntax

u_long ***FntFlush**(
int *id*
)

Arguments

id	Print Stream ID

Explanation

The contents of the specified print stream are drawn in the frame buffer. FntFlush() initializes and draws a sprite primitive list corresponding to the characters in the specified print stream.

Return Value

A pointer to the start of the OT used for drawing is returned.

Notes

After drawing is completed, the contents of the print stream are flushed.

FntLoad

Loads the system font patterns into the frame buffer

Syntax

void **FntLoad**(
int *tx*,
int *ty*
)

Arguments

tx, ty	Upper left-hand coordinate where the font patterns will be
loaded	into the frame buffer

Explanation

The system font patterns used for debugging are loaded into the frame buffer beginning at the coordinates specified by (*tx, ty*). All print streams are initialized. The system font patterns are 4-bit textures and require an area of size 256 x 128.

Return Value

None

Notes

FntLoad() must always be executed before FntOpen() and FntFlush().
The font area must not overlap with the frame buffer area used by the application.

FntOpen

Opens a print stream

Syntax

int **FntOpen**(
int *x*,
int *y*,
int *w*,
int *h*,
int *isbg*,
int *n*
)

Arguments

x, y	Display start position
w, h	Display area
isbg	Background auto-clear flag 0: Clear the background to (0,0,0) when displaying. 1: Do not clear the background.
n	Maximum number of characters that can be drawn

Explanation

The stream used for on-screen printing is opened. Subsequently, character strings up to *n* characters long can be drawn in the rectangular area of the frame buffer specified by (*x,y*)-(*x+w, y+h*) using the FntPrint() function.

If *isbg* has a value of "1", the background is cleared when a character string is drawn.

Return Value

The ID of the print stream that was opened is returned.

Notes

Up to 8 streams can be opened simultaneously.

Opened streams cannot be closed until the next FntLoad().

FntPrint

Sends output to a print stream

Syntax

int **FntPrint**(
int *id*,
char **format,*
...
)

Arguments

id	Print stream ID
format	Print format string

Explanation

The character string specified as an argument is formatted and sent as output to the specified print stream. Formatting is performed according to *format* and is identical to the specification of the format string for the C library fprintf() function.

Return Value

The number of characters in the stream is returned.

Notes

The character string is not displayed until FntFlush() is executed.

GetClut

Calculates value of member clut in a primitive

Syntax

u_short **GetClut** (
int *x*,
int *y*
)

Arguments

x, y	Frame buffer address of the CLUT

Explanation

The texture CLUT ID is calculated and its value is returned.

Return Value

The calculated CLUT ID is returned.

Notes

The CLUT address is limited to a multiple of 16 in the x direction.

GetTPage

Calculates value of member tpage in a primitive

Syntax

u_short **GetTPage** (

int *tp*,

int *abr*,

int *x*,

int *y*

)

Arguments

tp	Texture mode	
	0:	4-bit CLUT
	1:	8-bit CLUT
	2:	16-bit Direct
abr	Semi-transparency rate	
	0:	0.5 x Back + 0.5 x Forward
	1:	1.0 x Back + 1.0 x Forward
	2:	1.0 x Back - 1.0 x Forward
	3:	1.0 x Back + 0.25 x Forward
x, y	Texture page address	

Explanation

The texture page ID of the texture page address specified by (*x, y*) is calculated and its value is returned.

Return Value

The calculated texture page ID is returned.

Notes

The semi-transparency rate is also valid for polygons for which texture mapping is not performed.

The texture page address is limited to a multiple of 64 in the x direction and a multiple of 256 in the y direction.

GsClearOt

Initializes an OT

Syntax

void **GsClearOt** (
unsigned short *offset*,
unsigned short *point*,
GsOT **otp*
)

Arguments

offset	Ordering table offset value
point	Ordering table typical Z value
otp	Pointer to an ordering table

Explanation

The ordering table pointed to by *otp is* initialized.

offset specifies the Z value used for the start of the ordering table. *point* specifies a typical Z value for the entire ordering table. *point* is used to set depth priority when multiple ordering tables are linked together.

The length field of the GsOT structure must be properly set before this function is called so that the size of the ordering table to be cleared can be determined.

Return Value

None

See Also

GsOT, GsDrawOt()

GsDefDispBuff

Defines the double buffers used for drawing

Syntax

void **GsDefDispBuff (**

int *x0,*

int *y0,*

int *x1,*

int *y1,*

)

Arguments

x0, y0	Buffer 0 origin (upper left-hand) coordinates
x1, y1	Buffer 1 origin (upper left-hand) coordinates

Explanation

The display areas in the frame buffer used for double buffering are established.

The *x0*, *y0* parameters specify the coordinates within the frame buffer for buffer #0. The *x1*, *y1* parameters specify the coordinates within the frame buffer for buffer #1. If *x0, y0* and *x1, y1* are specified as the same coordinates in non-interlaced mode, the double buffers are released. However, double-buffer swapping of even-numbered and odd-numbered fields is performed automatically when *x0, y0* and *x1, y1* are specified as the same coordinates in interlaced mode.

Normally, buffer #0 is located at (0,0) and buffer #1 is located at (0, *yres*), where *yres* is the vertical resolution specified using the GsInitGraph() function.

The GsSwapDispBuffer() function is used to swap the double buffers. Double buffering is implemented using the GPU/GTE offset which can be set using GsInitGraph(). When using the GPU to manage the offset, the double buffer offset will be added at the time of drawing, not at the time of packet creation.

Return Value

None

See Also

GsInitGraph(), GsSwapDispBuff()

GsDrawOt

Execute drawing commands associated with an OT

Syntax

void **GsDrawOt** (
GsOT **otp*
)

Arguments

otp	Pointer to an ordering table

Explanation

The drawing commands that are entered in the ordering table pointed to by *otp* are executed. Drawing is performed by the GPU.
GsDrawOt() is a non-blocking function and returns immediately. The actual drawing is performed in the background.

Notes

This function does not execute correctly if it is called when GPU drawing is in progress. ResetGraph(1) should be used to terminate any current drawing process or DrawSync(0) should be used to wait until drawing is completed before this function is called.

Return Value

None

See Also

GsOT, GsClearOt()

GsGetActiveBuff

Gets the drawing buffer index

Syntax

int **GsGetActiveBuff** (void)

Arguments

None

Explanation

The double buffer index (PSDIDX) is obtained and its value is returned.
The value of the index is either 0 or 1.
By entering the index in the external variables PSDBASEX[] and PSDBASEY[], the two-dimensional address of the double buffer origin (upper left-hand coordinate) in the frame buffer can be obtained.

Return Value

The double buffer index (0 for buffer 0; 1 for buffer 1) is returned.

See Also

PSDIDX

GsGetLs

Calculates a local-to-screen transformation matrix

Syntax

void **GsGetLs** (
GsCOORDINATE2 **coord*,
MATRIX **m*
)

Arguments

coord	Pointer to a local coordinate system
m	Pointer to result matrix

Explanation

The local-to-screen transformation matrix of the coordinate system pointed to by *coord* is calculated and the result is stored in the matrix pointed to by *m*.
The GsCOORDINATE2 structure pointed to by *coord* describes a node in the hierarchical coordinate system. When the transformation matrix is calculated, calculation is performed for each node in the hierarchy and results are stored in the *workm* member within each node of the coordinate system. To improve performance, calculation is only performed for nodes which have changed, as determined by the value of the *flag* member, even when GsGetLw() is called repeatedly. It should be noted that when a parent node is changed, all subordinate nodes will be recalculated automatically even if their *flag* members are set.
It is the responsibility of the programmer to clear *flag* within each node if the member *coord* has changed, otherwise the transformation matrix will not be calculated correctly.

Return Value

None

See Also

GsSetLsMatrix()

GsGetLw

Calculates a local-to-world transformation matrix

Syntax

void **GsGetLw** (
GsCOORDINATE2 **coord*,
MATRIX **m*
)

Arguments

coord	Pointer to a local coordinate system
m	Pointer to the result matrix

Explanation

The local-to-world transformation matrix of the coordinate system pointed to by *coord* is calculated and the result is stored in the matrix pointed to by *m*.
The GsCOORDINATE2 structure pointed to by *coord* describes a node in the hierarchical coordinate system. When the transformation matrix is calculated, calculation is performed for each node in the hierarchy and results are stored in the *workm* member within each node of the coordinate system. To improve performance, calculation is only performed for nodes which have changed, as determined by the value of the *flag* member, even when GsGetLw() is called repeatedly. However, when a parent node is changed, all subordinate nodes will be recalculated automatically even if their *flag* members are set.
It is the responsibility of the programmer to clear *flag* within each node if the member *coord* has changed, otherwise the transformation matrix will not be calculated correctly.

Return Value

None

See Also

GsGetLws(), GsSetLightMatrix()

GsGetLws

Calculates both local-to-world and local-to-screen transformation matrices

Syntax

void **GsGetLws** (
GsCOORDINATE2 **coord2*
MATRIX **lw*,
MATRIX **ls*
)

Arguments

coord2	Pointer to local coordinate system
lw	Pointer to local-to-world coordinate system result matrix
ls	Pointer to local-to-screen coordinate system result matrix

Explanation

The local-to-world and the local-to-screen transformation matrices for the coordinate system pointed to by *coord2* are calculated and the results are stored in the matrices pointed to by *lw* and *ls*, respectively.
The local-to-world matrix must be specified when light source calculation is performed. GsGetLws() is a fast method for obtaining this matrix.
GsGetLws() is equivalent to calling GsGetLw() followed by GsGetLs(), only with faster performance.

Return Value

None

See Also

GsGetLs(), GsGetWs()

GsGetTimInfo

Gets TIM data header

Syntax

void **GsGetTimInfo** (
unsigned long **tim*,
GsIMAGE **im*
)

Arguments

tim	Pointer to TIM data header
im	Pointer to result image structure

Explanation

Information obtained from the TIM data header pointed to by *tim* is placed in the GsIMAGE structure pointed to by *im*.
TIM data begins at the address after the ID, which is 4 bytes after the start of the TIM file.
For a description of the TIM file format, please refer to the Net Yaroze Members' Web site.

Return Value

None

See Also

GsIMAGE

GsGetWorkBase

Gets the base address of the drawing command area

Syntax

PACKET ***GsGetWorkBase** (void)

Arguments

None

Explanation

The address of the first packet in the drawing command work area is obtained and its value is returned.

Note that the type PACKET is defined as an unsigned char in libps.h.

Return Value

The address of the first packet in the drawing command work area is returned.

See Also

GsSetWorkBase(), GsOUT_PACKET_P

GsIncFrame

Increments the frame ID

Syntax

GsIncFrame()

Arguments

None

Explanation

The global variable PSDCNT is incremented by 1.
PSDCNT is used by GsGetLw(), GsGetLs() and GsGetLws() to determine the validity of the matrix cache when the double buffer is swapped. PSDCNT is 32 bits in length, and restarts at 1 rather than 0 when it overflows.
GsIncFrame() is automatically called from within GsSwapDispBuff(). If the double buffer is swapped without using GsSwapDispBuff(), but the functions GsGetLw(), GsGetLs() and GsGetLws() are used, GsIncFrame() needs to be called every time the double buffer is swapped.

See Also

PSDCNT, GsGetLw(), GsGetLs(), GsGetLws(), GsSwapDispBuff()

Table: Graphics External Variables

Global	Type	Description
CLIP2	RECT	2-dimensional clipping area
PSDOFSX [2]	unsigned short	Double buffer base point (X coordinate) Set by GsDefDispbuff()
PSDOFSY [2]	unsigned short	Double buffer base point (Y coordinate) Set by GsDefDispbuff()
PSDIDX	unsigned short	Double buffer index
PSDCNT	unsigned long	Number incremented by frame buffer switch
POSITION	_GsPOSITION	2-dimensional offset
GsDRAWENV	DRAWENV	Drawing Environment
GsDISPENV	DISPENV	Display Environment
GsLSMATRIX	MATRIX	Local screen matrix Set by GsSetLs()
GsWSMATRIX	MATRIX	World screen matrix Set by GsSetRefView(), etc.
GsLIGHT_MODE	int	Default light mode
GsLIGHTWSMATRIX	MATRIX	Light matrix Set by GsSetFlatLight()
GsIDMATRIX	MATRIX	Unit matrix
GsIDMATRIX2	MATRIX	Unit matrix (including aspect conversion)
GsOUT_PACKET_P	unsigned long	Pointer to top of packet area Set by GsSetWorkBase()
GsLMODE	unsigned long	Attribute decoding result (light mode)
GsLIGNR	unsigned long	Attribute decoding result (light ignored)
GsLIOFF	unsigned long	Attribute decoding result (without shading)
GsNDIV	unsigned long	Attribute decoding result (division number)
GsTON	unsigned long	Attribute decoding result (semi-transparency)
GsDISPON	unsigned long	Attribute decoding result (display / no display

GsInit3D

Initializes the 3D graphics system

Syntax

void **GsInit3D** (void)

Arguments

None

Explanation

The 3D graphics system within the library is initialized.

The 3D graphics system needs to be initialized by this function first, so that 3D processing functions such as GsSetRefView(), GsInitCoordinate2() and GsSortObject4() can be used.

The following are the steps in the initialization process.

(1) The screen origin is reset to the center of the screen.

(2) The default light source is set to LIGHT_NORMAL.

Return Value

None

Notes

It is necessary to initialize the graphics system with GsInitGraph() before calling this function.

See Also

GsInitGraph(), GsSetRefView(), GsInitCoordinate2(), GsSortObject4()

GsInitCoordinate2

Initialize a local coordinate system

Syntax

void **GsInitCoordinate2** (
GsCOORDINATE2 **super,*
GsCOORDINATE2 **base*
)

Arguments

super	Pointer to the parent coordinate system
base	Pointer to the local coordinate system to be initialized

Explanation

The local coordinate system pointed to by *base* and its parent coordinate system pointed to by *super* are initialized.
The base coordinate system is initialized using *base*->coord whereas the parent coordinate system is initialized using *super*; i.e. the *base*->super member is ignored

Return Value

None

See Also

GsInitFixBg16

Initializes high-speed BG work area

Syntax

void **GsInitFixBg16** (

GsBG **bg*,

unsigned long **work*

)

Arguments

bg	Pointer to a background descriptor
work	Pointer to the work area (primitive area)

Explanation

The work area used by the GsSortFixBg16() function is initialized.

The size of the work area varies with the screen resolution according to the following formula:

Work area size (in long units) =

(((ScreenW/CellW+1)*(ScreenH/CellH+1+1)*6+4)*2+2)

where

ScreenH = screen height in pixels (240/480)

ScreenW = screen width in pixels (256/320/384/512/640)

CellH = cell height (in pixels)

CellW = cell width (in pixels)

GsInitFixBg16() should only be executed once. It does not need to be executed every frame.

Return Value

None

See Also

GsSortFixBg16()

GsInitGraph

Initializes the graphics system

Syntax

```
void GsInitGraph (
int x_res,
int y_res,
int intl,
int dither,
int vram
)
```

Arguments

x_res	Horizontal resolution (256/320/384/512/640)
y_res	Vertical resolution (240/480)
intl	Interlace display flag (bit 0) 0: Non-interlaced 1: Interlaced Double Buffer Offset Mode (bit 2) 0: GTE offset 1: GPU offset
dither	Dither processing flag 0: OFF 1: ON
vram	Frame buffer mode 0: 16-bit

1: 24-bit

Explanation

The graphics system is initialized.

The operational settings of the graphics system are reflected in the global variables GsDISPENV and GsDRAWENV. The program can use these global variables to determine the current settings of the graphics system.

The Double Buffer Offset Mode controls the offset that is added to the frame buffer when drawing is performed. The Mode determines whether the offset is managed by GTE or the GPU.

When the Mode is specified as GTE, the offset is added to the drawing address when the drawing packet is created. When the Mode is specified as GPU, the offset is added when drawing is performed. Having the GPU manage the offset is usually preferred because the offset is not included in the drawing packet.

In 24-bit mode, the frame buffer can only operate in image display mode and polygons cannot be drawn.

GsInitGraph() must be called to initialize the graphics system. GsInitGraph() also initializes the internal structures GsIDMATRIX and GsIDMATRIX2. These structures must be initialized before any of the Gs*** functions can be used.

Return Value

None

GsLinkObject4

Links an object to TMD data

Syntax

void **GsLinkObject4** (
unsigned long **tmd,*
GsDOBJ2 **obj_base,*
unsigned long *n*
)

Arguments

tmd	Pointer to TMD data
obj_base	Pointer to the linking object structure
n	Index of the object to be linked

Explanation

The object pointed to by *obj_base* is linked to the *n*-th object in the TMD data pointed to by *tmd*. This allows TMD 3D objects to be handled using GsDOBJ2.

Return Value

None

Notes

Objects linked by GsLinkObject4() can be registered in the ordering table by GsSortObject4().

See Also

GsSortObject4(), GsDOBJ2

GsMapModelingData

Maps TMD data to real addresses

Syntax

void **GsMapModelingData** (
unsigned long **p*
)

Arguments

p	Pointer to starting address of TMD data

Explanation

Offsets within the TMD data pointed to by *p* are converted into real addresses. The converted addresses are stored back in the TMD data.

TMD data includes various fields which contain the memory addresses of certain pieces of data. During the preparation of TMD data, however, the memory address where the data will be loaded is not yet known. Therefore, address fields in the TMD data are stored as offsets from the start of the data. The GsMapModelingData() function changes these offsets into real addresses after the TMD data has been loaded into memory. This must be done before the TMD data can be used.

TMD data begins at the address after the ID which is 4 bytes after the start of the TMD file.

For a description of the TMD file format, please refer to the Net Yaroze Members' Web site.

Return Value

None

Notes

A flag is set in the TMD data to indicate that the offset addresses have been converted into real addresses. So, no side effects occur even if GsMapModelingData() is called repeatedly.

GsScaleScreen

Scales the screen coordinate system

Syntax

void **GsScaleScreen** (
SVECTOR **scale*
)

Arguments

scale	Pointer to the scale vector (each element is a 12-bit integer)

Explanation

The screen coordinate system is scaled relative to the world coordinate system by an amount specified in the vector pointed to by *scale*.

Each element of *scale* is a 12-bit integer. Scaling is performed relative to the original screen coordinate system set by GsSetView2() and GsSetRefView2(). If each element of *scale* is set to the value ONE, the screen coordinate system is reset to its original values. Note that ONE is defined to be the value 4096.

World coordinates are represented in 32 bits, whereas screen coordinates are represented in 16 bits. The different representations would normally cause problems such as Far Clip proximity. GsScaleScreen() eliminates these problems by scaling the screen coordinate system within the wider area of the world coordinate system.

For example, if the elements of *scale* are each set to ONE/2, the screen coordinate system will be expanded to the equivalent of 17 bits after scaling. Since the precision of screen coordinate values is only 16 bits, the low-order bit will be ignored.

Note that when scaling is performed, objects that exist in screen coordinate systems having different scaling factors cannot be entered in the same ordering table. For example, in order to enter an object that was calculated with the normal scale screen coordinate system in an ordering table which already contains an object with a 1/2 scale screen coordinate system, it is necessary to right-shift the excess bit before entering the object in the ordering table. This can be done by specifying the shift amount when calling GsSortObject4().

Return Value

None

GsSetAmbient

Set ambient color

Syntax

void **GsSetAmbient** (
unsigned short *r,*
unsigned short *g,*
unsigned short *b*
)

Arguments

r, g, b	RGB values of the ambient color (0~4095)

Explanation

The ambient color is set to the RGB value specified by (*r, g, b*).
The *r, g, b* arguments can pass values up to but not including 4096.
A value of 4096 would represent a fixed point value of 1.0 and correspond to normal ambient brightness. A value of 0 corresponds to minimum brightness. A value of 1/8 of normal brightness would be specified with *r, g, b* values of 4096/8.

Return Value

None

See also

GsSetFlatLight()

GsSetClip

Sets the clipping area for drawing

Syntax

void **GsSetClip** (
RECT **clip*
)

Arguments

clip	Pointer to RECT structure defining the clipping area

Explanation

The clipping area defined by *clip* is set as the clipping area for drawing.
This function is different from GsSetDrawBuffClip() in that its argument can be used to specify a clipping area. Note that this clipping value is a relative one within the double buffer, and thus the clipping position will not change even when double buffers are swapped.
The clipping area set by GsSetClip is valid only within the current drawing buffer and becomes invalid when the buffers are swapped using GsSwapDispBuff(). This is because GsSwapDispBuff() calls GsSetDrawBuffOffset() which sets the clipping area from the clip member in the global variable GsDRAWENV. This value is obtained from the global variable CLIP2.

Return Value

None

Notes

This function does not execute correctly if it is called when GPU drawing is in progress. ResetGraph(1) should be used to terminate any current drawing process or DrawSync(0) should be used to wait until drawing is completed before this function is called.

See Also

GsSetDrawBuffClip()

GsSetClip2D

Sets a 2-dimensional clipping area

Syntax

void **GsSetClip2D** (

RECT **rectp*

)

Arguments

rectp	Pointer to a 2D clipping area

Explanation

The 2D area pointed to by *rectp* is set as the clipping area.

The clipping area is not affected by double buffer swapping, so the same area is automatically clipped regardless of which buffer is active.

GsSetDrawBuffClip() must be called in order for the clip area to take effect immediately. Otherwise, the setting becomes valid with the next frame.

Return Value

None

GsSetDrawBuffClip

Sets the clipping area for drawing

Syntax

void **GsSetDrawBuffClip** (void)

Arguments

None

Explanation

The clipping area used in drawing is updated with the value that was set using GsClip2D().

The clipping value is relative within the double buffers. In other words, the clipping position does not change when the double buffers are swapped.

Return Value

None

Notes

This function does not execute correctly if it is called when GPU drawing is in progress. ResetGraph(1) should be used to terminate any current drawing process or DrawSync(0) should be used to wait until drawing is completed before this function is called.

See Also

GsSetClip2D(), GsSetClip()

GsSetDrawBuffOffset

Sets the drawing offset

Syntax

void **GsSetDrawBuffOffset** (void)

Arguments

None

Explanation

The drawing offset is updated using the value defined in the global variable POSITION. The drawing offset is used for double buffer swapping and can be managed by the GTE or the GPU.

The drawing offset is a relative offset within the double buffer. The offset value is preserved even when double buffers are swapped.

The third argument of GsInitGraph() determines whether the offset is managed by the GTE or the GPU.

Return Value

None

Notes

This function does not execute correctly if it is called when GPU drawing is in progress. ResetGraph(1) should be used to terminate any current drawing process or DrawSync(0) should be used to wait until drawing is completed before this function is called.

See Also

GsSetOrign(), GsSetOffset(), POSITION

GsSetFlatLight

Sets a parallel light source

Syntax

void **GsSetFlatLight** (
unsigned short *id,*
GsF_LIGHT **lt*
)

Arguments

id	Light source number (0,1,2)
lt	Pointer to light source information

Explanation

The parallel light source specified by *id* is set according to the GsF_LIGHT information pointed to by *lt*. Up to three separate light sources can be set (*id* = 0, 1, 2).

Return Value

None

Notes

When the contents of the GsF_LIGHT structure are overwritten, the new values will not be reflected in the light source until GsSetFlatLight() is called.

See Also

GsF_LIGHT, GsSetAmbient()

GsSetFogParam

Sets the fog parameter

Syntax

void **GsSetFogParam** (
GsFOGPARAM **fogparam*
)

Arguments

fogparam	Pointer to a fog parameter structure

Explanation

The fog parameter is set according to the GsFOGPARAM structure pointed to by *fogparam*. Fog is only effective if the light source mode is 1 (see GsSetLightMode()).

Return Value

None

See Also

GsFOGPARAM, GsSetLightMode()

GsSetLightMatrix

Sets the light matrix

Syntax

void **GsSetLightMatrix** (
MATRIX **mp*
)

Arguments

mp	Pointer to a local screen light matrix

Explanation

The local screen light matrix pointed to by *mp* is multiplied by the matrix of three light vectors and the result is set as the light matrix in GTE.
The light matrix needs to be set in GTE prior to performing light source calculations. Depending on the type of model data, the GsSortObject4() function may perform light source calculation at the time of execution. In this case, the light matrix needs to be preset using GsSetLightMatrix().
The matrix *mp* specified as the argument to GsSetLightMatrix() is normally a local world matrix.

Return Value

None

See Also

GsSortObject4(), GsGetLw()

GsSetLightMode

Sets the default light source mode

Syntax

void **GsSetLightMode** (
unsigned short *mode*
)

Arguments

mode	Light source mode (0~1)
	0: normal lighting
	1: normal lighting with fog ON

Explanation

The default light source mode is set to the value specified by *mode*.
The light source mode can also be set independently for each object using the *attribute* member of GsDOBJ2. Bit 5 of *attribute* determines whether the light source mode is set by the GsDOBJ2 attribute field or by the mode set with GsSetLightMode(). See the description of GsDOBJ2 for more information.

Return Value

None

GsSetLsMatrix

Sets the local screen matrix

Syntax

void **GsSetLsMatrix** (
MATRIX **mp*
)

Arguments

mp	Pointer to a local screen matrix

Explanation

The local screen matrix pointed to by *mp* is set as the local screen matrix in GTE. The local screen matrix in GTE needs to be set prior to performing perspective transformation using a function such as GsSortObject4().

Return Value

None

See Also

GsSortObject4(), GsGetLs()

GsSetOffset

Sets the drawing offset

Syntax

void **GsSetOffset** (
int *offx,*
int *offy*
)

Arguments

offx	Drawing offset X coordinate
offy	Drawing offset Y coordinate

Explanation

The value of the drawing offset is set to *(offx, offy)*.

GsSetOffset() differs from GsSetDrawBuffOffset() in that GsSetDrawBuffOffset() sets the offset to the value specified by the global variable POSITION, whereas GsSetOffset() sets the offset to the value specified by the argument.

The value set by GsSetOffset() is temporary in that it does not update the global variable POSITION and is only valid within the current draw buffer. This is because when the double buffer is swapped, GsSwapDispBuff() calls GsSetDrawBuffOffset() which sets the offset from the global variable POSITION. This value remains valid until it is changed by GsSetOrigin().

The offset specified as an argument is a relative offset inside the double buffer. In other words, the offset is added to the double buffer base to calculate the drawing address.

The third argument of GsInitGraph() determines whether the offset is managed by the GTE or the GPU.

Return Value

None

Notes

This function does not execute correctly if it is called when GPU drawing is in progress. ResetGraph(1) should be used to terminate any current drawing process or DrawSync(0) should be used to wait until drawing is completed before this function is called.

See Also

GsSetDrawBuffOffset()

GsSetOrign

Sets the drawing offset of the screen origin

Syntax

```
void GsSetOrign (
int x,
int y
)
```

Arguments

x	Screen origin position X
y	Screen origin position Y

Explanation

The screen origin specified by (*x, y*) is set as the drawing offset in the global variable POSITION.

GsSetOrign() sets the drawing offset in the global variable POSITION but it does not set it in the GPU. In other words, the new offset will not take effect until a function such as GsSetDrawBuffOffset() that sets the GPU drawing offset is called.

The offset value in the global variable POSITION is valid until GsSetOrgn() is called again.

The offset specified by (*x, y*) is relative within the double buffer. In other words, the values (*x, y*) are added to the double buffer base. In reality, the offset is set by the *offx* and *offy* members of the global variable POSITION.

Notes

The third argument of GsInitGraph() determines whether the offset is executed by GTE or by GPU and is specified using GsOFSGTE or GsOFSGPU, respectively.

Return Value

None

GsSetProjection

Sets the projection distance

Syntax

void **GsSetProjection** (
unsigned short *h*
)

Arguments

h	Projection distance.

Explanation

The projection distance is set to *h*. The projection distance is the distance between the viewpoint and the projection plane.
The size of the projection plane is specified by the (*xres, yres*) arguments to GsInitGraph(). This size is constant for a given resolution. This means that the field of view decreases as the projection distance is increased, and conversely, the field of view increases as the projection distance is decreased.
As shown in the table below, the aspect ratio is a function of the screen resolution and may not always be 1 to 1. In particular, when the resolution is 640 x 240, the X coordinate should be scaled by 1/2 to compensate.

Resolution	640x480	640x240	320x240
Aspect ratio	1:1	2:1	1:1

Return Value

None

GsSetRefView2

Sets the viewpoint position

Syntax

int **GsSetRefView2** (
GsRVIEW2 **pv*
)

Arguments

pv	Pointer to viewpoint position information (reference type)

Explanation

The World Screen Matrix (WSMATRIX) is calculated using the viewpoint information pointed to by *pv*.
Since WSMATRIX does not change unless the viewpoint is moved, this function need not be called for each frame. However, if the viewpoint is moved, GsSetRefView2() must be called for changes to be reflected in each frame.
GsSetRefView2() should be called each frame if the GsRVIEW2 member *super* is set to anything other than WORLD. In this case, the viewpoint will move if the parameters of the parent coordinate system change even if the other viewpoint parameters have not changed.

Return Value

0 is returned if the viewpoint was successfully set, otherwise, 1 is returned.

See Also

GsRVIEW2,GsWSMATRIX, GsSetView2()

GsSetView2

Sets the viewpoint position

Syntax

int **GsSetView2** (
GsVIEW2 **pv*
)

Arguments

pv	Pointer to viewpoint position information (matrix type)

Explanation

The World Screen (WS) matrix is directly set from the viewpoint information pointed to by *pv*.
When GsSetRefView2() is used to determine the WS matrix from the viewpoint, errors may occur when the viewpoint is moved due to insufficient precision in the operation. For this reason it is usually better to use GsSetView2().
GsSetView2() should be called each frame if the GsRVIEW2 member *super* is set to anything other than WORLD. In this case, the viewpoint will move if the parameters of the parent coordinate system change even if the other viewpoint parameters have not changed.
If GsIDMATRIX2 is used as the base matrix, then the aspect ratio of the screen will be adjusted automatically.

Return Value

0 is returned if the viewpoint was successfully set, otherwise, 1 is returned.

See Also

GsVIEW2, GsWSMATRIX, GsSetRefView2()

GsSetWorkBase

Sets the base address of the drawing command area

Syntax

void **GsSetWorkBase** (
PACKET **base_addr*
)

Arguments

base_addr	pointer to the first PACKET of the drawing command area

Explanation

The address of the packet at the start of the drawing command work area is set to the address specified by *base_addr*. The drawing command work area is used by functions such as GsSortObject4() and GsSortSprite().
This function must be called at the start of processing for each frame. *Base_addr* should be set to the starting address of the packet area allocated by the user.
Note that the type PACKET is defined as an unsigned char in libps.h.

Return Value

None

See Also

GsSortObject4(), GsSortSprite(), GsSortFastSprite(), GsOUT_PACKET_P

GsSortBoxFill

Enters a rectangle in an OT

Syntax

void **GsSortBoxFill** (
GsBOXF **bp*,
GsOT **otp*,
unsigned short *pri*,
)

Arguments

bp	Pointer to a rectangle descriptor
otp	Pointer to an ordering table
pri	Position in the ordering table

Explanation

The rectangle pointed to by *bp* is entered into the ordering table pointed to by *otp* at position *pri*.

Return Value

None

GsSortClear

Enters a Clear Screen command in an OT

Syntax

void **GsSortClear** (
unsigned char *r*,
unsigned char *g*,
unsigned char *b*,
GsOT **otp*
)

Arguments

r, g, b	Background color RGB values
otp	Pointer to an ordering table

Explanation

A Clear Screen command is entered at the beginning of the ordering table pointed to by *otp*.

Return Value

None

Notes

GsSortClear() merely enters the Clear Screen command in the ordering table. The screen will not actually be cleared until GsDrawOT() is called.

GsSortFastSprite

Enters a sprite in an OT

Syntax

void **GsSortFastSprite**(
GsSPRITE **sp*,
GsOT **otp*,
unsigned short *pri*
)

Arguments

sp	Pointer to a sprite
otp	Pointer to an ordering table
pri	Position in the ordering table

Explanation

The sprite pointed to by *sp* is entered into the ordering table pointed to by *otp*.
The parameters of the sprite, such as its display position, are provided by the members of GsSPRITE.
pri is the priority (i.e. position) of the sprite in the ordering table. The highest priority is obtained by setting *pri* to 0. The lowest priority is determined by the size of the ordering table. If the value of *pri* is greater than the size of the ordering table, it is clipped to the maximum size.
GsSortFastSprite() provides higher speed processing than GsSortSprite(), however, scaling and rotation cannot be used. The GsSPRITE members *mx*, *my*, *scalex*, *scaley*, and *rotate* are ignored.

Return Value

None

See Also

GsSortSprite(), GsSPRITE

GsSortFixBg16

Enters a background descriptor in an OT at high speed

Syntax

void **GsSortFixBg16** (
GsBG **bg*,
unsigned long **work*,
GsOT **otp*,
unsigned short *pri*
)

Arguments

bg	Pointer to a background descriptor
work	Pointer to the work area (primitive area) initialized with GsInitFixBg16()
otp	Pointer to an ordering table
pri	Position in the ordering table

Explanation

The background image pointed to by *bg* is entered into the ordering table pointed to by *otp* at position *pri*.

GsSortFixBg16() has the following features and restrictions:

- BG rotation and scaling cannot be performed.
- Cell size is fixed (16x16).
- Only texture pattern color modes 4-bit and 8-bit are supported.
- Map size is optional.
- Scrolling is permitted in 1 pixel units.

· Only full screen mode is supported.

This function uses the work area pointed to by *work* to store drawing primitives. The work area must be initialized in advance by GsInitFixBg16().The packet area set with GsSetWorkBase() is not used.

Return Value

None

See Also

GsInitFixBg16()

GsSortGLine

Enters a graded straight line in an OT

Syntax

void **GsSortGLine** (
GsGLINE **lp*,
GsOT **otp*,
unsigned short *pri*
)

Arguments

lp	Pointer to a graded straight line descriptor
otp	Pointer to an ordering table
pri	Position in the ordering table

Explanation

The graded straight line pointed to by *lp* is entered into the ordering table pointed to by *otp* at position *pri*.
GsSortGLine() handles straight lines and GsSortLine() handles single-color straight lines.

Return Value

None

See Also

GsSortLine()

GsSortLine

Enters a single-color straight line in an OT

Syntax

```
void GsSortLine (
GsLINE *lp,
GsOT *otp,
unsigned short pri
)
```

Arguments

lp	Pointer to a single-color straight line descriptor
otp	Pointer to an ordering table
pri	Position in the ordering table

Explanation

The single-color straight line pointed to by *lp* is entered into the ordering table pointed to by *otp* at position *pri*.

GsSortLine() handles single-color straight lines and GsSortGLine() handles straight lines with gradation.

Return Value

None

See Also

GsSortGLine()

GsSortObject4

Inserts an object in an OT

Syntax

void **GsSortObject4** (
GsDOBJ2 **objp*,
GsOT **otp*,
long *shift*,
u_long **scratch*
)

Arguments

objp	Pointer to a GsDOBJ2 object
otp	Pointer to an ordering table
shift	Number of bits the Z value will be right-shifted when the
object	is entered in the ordering table.
scratch	Pointer to a scratchpad buffer

Explanation

Performs perspective transformation and light source calculation for the 3D object pointed to by *objp*. Generates drawing commands for the object and stores those commands at the packet area work base address, previously specified with GsSetWorkBase(). Z-sorts the drawing commands and enters them in the ordering table pointed to by *otp*, shifting the Z values by an amount specified by *shift*.

The precision of Z may be adjusted using *shift*. The maximum size of the ordering table (i.e. its resolution) is 14 bits and is specified in the GsOT structure. The value of *shift* must be set so that the area allocated to the ordering table will not be exceeded when the object is entered. For example, if the size of the ordering table is 12 bits, the shift value must be set to 14-12, or 2.

scratch is a pointer to the Scratchpad buffer which is used when automatic polygon subdivision is being performed on the object. The size of the Scratchpad buffer is 1024 bytes.

The Scratchpad allows access to the Data Cache on the R3000. Using the Cache provides much faster access than main memory RAM for time-critical operations such as polygon subdivision. The Scratchpad is memory-mapped from addresses 0x1F800000 to 0x1F8003FF.

Automatic polygon subdivision is enabled by setting the appropriate bits in the *attribute* member of GsDOBJ2. Subdivision is enabled by ORing the *attribute* member with one of the macros 'GSDIV1' through 'GsDIV5' which are defined in libps.h. For example, if GsDIV1 is specified, a single polygon will be divided into 4 segments of 2 x 2. If GsDIV5 is specified, a single polygon will be divided into 1024 segments of 32 x 32.

Return Value

None

See Also

GsDOBJ2, GsSetWorkBase()

GsSortOt

Inserts a source OT into a destination OT

Syntax

```
GsOT *GsSortOt (
GsOT *ot_src,
GsOT *ot_dest
)
```

Arguments

ot_src	Pointer to source ordering table
ot_dest	Pointer to destination ordering table

Explanation

The source ordering table pointed to by *ot_src* is inserted into the destination ordering table pointed to by *ot_dest*. The typical Z value of the source ordering table used for the insertion operation is given by the *point* member of *ot_src*.

Return Value

ot_dest, the pointer to the combined ordering table is returned.

See Also

GsOT

GsSortSprite

Enters a sprite in an OT

Syntax

void **GsSortSprite** (
GsSPRITE **sp*,
GsOT **otp*,
unsigned short *pri*
)

Arguments

sp	Pointer to a sprite
otp	Pointer to an ordering table
pri	Position in the ordering table

Explanation

The sprite pointed to by *sp* is entered into the ordering table pointed to by *otp*.
The parameters of the sprite, such as its display position, are provided by the members of GsSPRITE.
pri is the priority (i.e. position) of the sprite in the ordering table. The highest priority is obtained by setting *pri* to 0. The lowest priority is determined by the size of the ordering table. If the value of *pri* is greater than the size of the ordering table, it is clipped to the maximum size.

Return Value

None

See Also

GsOT, GsSPRITE, GsSortFastSprite()

GsSwapDispBuff

Swaps double buffers

Syntax

void **GsSwapDispBuff** (void)

Arguments

None

Explanation

The display buffer and drawing buffer that were defined using GsDefDispBuff() are swapped.

Swapping is usually performed immediately after the start of the vertical blanking interval.

.

GsSwapDispBuff() performs the following functions:

1. Sets display starting address
2. Cancels blanking
3. Sets double buffer index
4. Switches two-dimensional clipping
5. Sets GTE or GPU offset
6. Increments PSDCNT

Note: Double buffering is implemented using an offset. The third argument of GsInitGraph() determines whether the offset is managed by the GTE or the GPU.

Return Value

None

Notes

This function does not execute correctly if it is called when GPU drawing is in progress. ResetGraph(1) should be used to terminate any current drawing process or DrawSync(0) should be used to wait until drawing is completed before this function is called.

See Also

GsDefDispBuff()

gteMIMefunc

Adds a vertex data vector to a differential data vector multiplied by a control coefficient

Syntax

void **gteMIMefunc** (
SVECTOR **otp,*
SVECTOR **dfp,*
long *n,*
long *p*
)

Arguments

otp	Pointer to vertex data vector (input/output)
dfp	Pointer to differential data vector (input)
n	Number of vertices (differentials)
p	MIMe weight (control) coefficient

Explanation

The vertex data vector *otp* is added to the product of the differential data vector *dfp* and the MIMe control coefficient *p*, and the result is stored back in the vertex data vector *otp*.

p is considered to be a 12-bit integer.This following program fragment illustrates the algorithm that is used.

```
void gteMIMefunc(SVECTOR *otp, SVECTOR *dfp, long n, long p)
{
                        int i;
```

```
        for( i = 0; i < n; i++){
            (otp+i)->x += ( (int)((dfp+i)->x) * p )>>12;
            (otp+i)->y += ( (int)((dfp+i)->y) * p )>>12;
            (otp+i)->z += ( (int)((dfp+i)->z) * p )>>12;
        }
}
```

The argument format is as follows::

p : (1,19,12)

otp, dfp arbitrary

Return Value

None

KanjiFntClose

Closes a print stream

Syntax

int **KanjiFntClose**(void)

Arguments

None

Explanation

All streams currently open and used by KanjiFntPrint() are initialized and cleared.

Return Value

None

Notes

Since KanjiFntClose() only initializes the internal state, this function completes even when there is no stream open.

KanjiFntFlush

Draws the contents of a print stream

Syntax

u_long ***KanjiFntFlush (**

int *id*

)

Arguments

id	Print Stream ID

Explanation

The contents of the specified print stream are drawn in the frame buffer. KanjiFntFlush() initializes and draws a sprite primitive list corresponding to the characters in the specified print stream.

Return Value

A pointer to the start of the ordering table used for drawing is returned.

Notes

After drawing is completed, the contents of the print stream are flushed.

KanjiFntOpen

Opens a print stream

Syntax

```
int KanjiFntOpen (
int x,
int y,
int w,
int h,
int dx,
int dy,
int cx,
int cy,
int isbg,
int n
)
```

Arguments

x, y	Display start positions
w, h	Display area
dx,dy	Kanji font pattern frame buffer address
cx,cy	Kanji CLUT frame buffer address
isbg	Background auto-clear flag 0: Clear the background to (0,0,0) when displaying. 1: Do not clear the background.
n	Maximum number of characters that can be drawn

Explanation

The stream used for on-screen printing is opened. Subsequently, character strings up to *n* characters long can be drawn in the rectangular area of the frame buffer specified by (*x,y*)-(*x+w, y+h*) using the KanjiFntPrint() function.

If *isbg* has a value of "1", the background is cleared when a character string is drawn.

Return Value

The ID of the print stream that was opened is returned.

Notes

Up to 8 streams can be opened simultaneously.

Opened streams cannot be closed until the next KanjiFntLoad().

The Kanji font area must not overlap with the frame buffer area used by the application.

KanjiFntPrint

Sends output to a print stream

Syntax

int **KanjiFntPrint**(
int *id*,
char **format,*
...
)

Arguments

id	Print stream ID
format	Print format

Explanation

The SHIFT-JIS full-width character string specified as an argument is formatted and sent as output to the specified print stream. Formatting is performed according to the *format* string and is identical to the specification of the format string for the C library fprintf() function.

Return Value

The number of characters in the stream is returned.

Notes

The Kanji code must be SHIFT-JIS. Full-width and half-width characters can be mixed in the character string, but they are all changed to full-width at the time of display. Half-width kana are not supported. The character string is not displayed until KanjiFntFlush() is executed.

Krom2Tim

Converts a SHIFT-JIS character string to 4-bit CLUT TIM data

Syntax

```
int Krom2Tim(
u_char *sjis,
u_long *taddr,
int dx,
int dy,
int cx,
int cy,
u_int fg,
u_int bg
)
```

Arguments

sjis	SHIFT-JIS Character String
taddr	Pointer to TIM data result storage area
dx, *dy*	x,y coordinates of pixel data in VRAM
cx, *cy*	x,y coordinates of CLUT data in VRAM
fg, *bg*	Foreground and background colors

Explanation

The SHIFT-JIS character string specified by *sjis* is converted to 4-bit CLUT TIM data. The result is placed in the buffer pointed to by *taddr*.

Return Value

0 is returned when conversion is successful. -1 is returned when the code specified by *sjis* is invalid.

Notes

The Kanji code must be SHIFT-JIS encoded. Full-width and half-width characters can be mixed in the character string, but they are all changed to full-width at the time the characters are displayed. Half-width kana are not supported.

For the area specified by *taddr*, the size shown in the following formula must be allocated in advance.

128 x (length of character string specified by *sjis*) + 84(bytes)

Krom2Tim2

Converts a SHIFT-JIS character string to 4-bit CLUT TIM data

Syntax

int **Krom2Tim2**(
u_char **sjis*,
u_long **taddr*,
int *dx*,
int *dy*,
int *cdx*,
int *cdy*,
u_int *fg*,
u_int *bg*
)

Arguments

sjis	SHIFT-JIS Character String
taddr	Pointer to TIM data result storage area
dx, *dy*	x,y coordinates of pixel data in VRAM
cx, *cy*	x,y coordinates of CLUT data in VRAM
fg, *bg*	Foreground and background colors

Explanation

The SHIFT-JIS character string specified by *sjis* is converted to 4-bit CLUT TIM data. The result is placed in the buffer pointed to by *taddr*. Krom2Tim2() is the user-defined character version of Krom2Tim().

Return Value

0 is returned when conversion is successful. -1 is returned when the code specified by *sjis* is invalid.

Notes

The Kanji code must be SHIFT-JIS encoded. Full-width and half-width characters can be mixed in the character string, but they are all changed to full-width at the time the characters are displayed. Half-width kana are not supported.

For the area specified by *taddr*, the size shown in the following formula must be allocated in advance, where *num* is the length of the string specified by *sjis*.

If (*num*<16)
(32 * *num* + 16) * 4 (bytes)
else
(32 * 16* (((*num*-1)/16 + 1) +16) * 4 (bytes)

LoadImage

Transfers image data from memory to the frame buffer

Syntax

int **LoadImage** (
RECT **recp*,
u_long **p*
)

Arguments

recp	Pointer to the destination rectangular area in the frame buffer
p	Pointer to the source area in main memory

Explanation

Data from the source area in main memory pointed to by *p* is transferred to the rectangular area of the frame buffer pointed to by *recp.*

Return Value

The slot number of the GPU's draw/command queue where the command resides is returned.

Notes

LoadImage() is a non-blocking function, therefore DrawSync() must be used to determine the completion of transfer.
The transfer is not affected by clipping or offsetting that may be in effect for the drawing environment.
The transfer area must be located within a drawable area: (0,0) - (1023,511).

MoveImage

Transfers data within the frame buffer

Syntax

```
int MoveImage (
RECT *recp,
int x,
int y
)
```

Arguments

recp	Pointer to source rectangle in the frame buffer
x,y	Pointer to upper left-hand corner of the destination rectangle in the frame buffer

Explanation

The rectangular area in the frame buffer specified by *recp* is transferred to a rectangular area of the same size starting at location *x,y* in the frame buffer.

Return Value

The slot number of the GPU's draw/command queue where the command resides is returned.

Notes

MoveImage() is a non-blocking function, therefore DrawSync() must be used to determine the completion of transfer.
The transfer is not affected by clipping or offsetting that may be in effect for the drawing environment.
The transfer area must be located within a drawable area: (0,0) - (1023,511).

The contents of the source are unaffected by the transfer. If the source and destination areas are the same, the results are unpredictable.

MulMatrix0

Multiplies two matrices

Syntax

MATRIX* **MulMatrix0** (

MATRIX **m0*,

MATRIX **m1*,

MATRIX **m2*

)

Arguments

m0, *m1*	Pointers to input matrices
m2	Pointer to result matrix

Explanation

Matrix *m0* is multiplied by matrix *m1* and the result is stored in matrix *m2*.

The argument format is as follows::

m0,*m1*,*m2*->m[i][j] : (1,3,12)

Return Value

The matrix product *m2* is returned.

Notes

This function destroys the rotation matrix.

PopMatrix

Pops the rotation matrix off the stack

Syntax

void **PopMatrix** (void)

Arguments

None

Explanation

The rotation matrix is popped off the stack.

Return Value

None

PushMatrix

Pushes the rotation matrix onto the stack

Syntax

void **PushMatrix** (void)

Arguments

None

Explanation

The rotation matrix is pushed onto the stack. The stack has a maximum of 20 slots.

Return Value

None

PutDispEnv

Sets the display environment

Syntax

DISPENV ***PutDispEnv** (
DISPENV **env*
)

Arguments

env	Pointer to the display environment

Explanation

The basic parameters of the display environment are established according to the display environment pointed to by *env*. The display environment becomes effective immediately when the function is called.

Return Value

A pointer to the new display environment is returned if the function succeeded. “0” is returned if PutDispEnv failed to set the new display environment.

Notes

The drawing environment specified by PutDispEnv() is valid until the next call to PutDispEnv() or GsSwapDispBuff().

See Also

GsSwapDispBuff(), DISPENV

PutDrawEnv

Sets the drawing environment

Syntax

DRAWENV ***PutDrawEnv** (
DRAWENV **env*
)

Arguments

env	Pointer to the drawing environment

Explanation

The basic parameters of the drawing environment (e.g. drawing offset and drawing clip area) are established according to the drawing environment pointed to by *env*.

Return Value

A pointer to the new drawing environment is returned if the function succeeded. "0" is returned if PutDrawEnv() failed to set the new drawing environment.

Notes

The drawing environment specified by PutDrawEnv() is valid until the next call to PutDrawEnv() or GsSwapDispBuff().

See Also

GsSwapDispBuff(), DRAWENV

ResetGraph

Initializes the graphics system

Syntax

int **ResetGraph** (
int *mode*
)

Arguments

mode	Specifies the reset mode as indicated below.	
	0:	Full reset. The drawing environment and display environment are initialized.
	1:	The current drawing is cancelled and the command queue is flushed.

Explanation

The graphics system is reset with the mode specified by *mode*.

Return Value

None

RotMatrix

Calculates a rotation matrix from a rotation vector

Syntax

MATRIX* **RotMatrix** (
MATRIX **m*
SVECTOR **r*
)

Arguments

m	Pointer to output rotation matrix
r	Pointer to input rotation vector

Explanation

The rotation matrix *m* is calculated from the rotation vector (*r*->vx,*r*->vy,*r*->vz). In the rotation vector, 360° is represented as 4096, and 4096 is equivalent to 1.0 in each element.

The matrix is an expansion of the following product.

Using GTE coordinate transformation, the vectors are multiplied from the right. Thus the matrix rotates around the Z, Y and X axes in that order.

Angle value

c0=cos(*r*->vx), s0=sin(*r*->vx)

c1=cos(*r*->vy), s1=sin(*r*->vy)

c2=cos(*r*->vz), s2=sin(*r*->vz)

The argument format is as follows::

m->*m*[i][j] : (1,3,12)

r->vx,vy,vz :(1,3,12) (however 360° is 1.0)

Return Value

The rotation matrix *m* is returned.

RotMatrixX

Rotates a matrix around the X-Axis

Syntax

MATRIX* **RotMatrixX** (

long *r*,

MATRIX **m*

)

Arguments

r	Input rotation amount
m	Pointer to the matrix (used for both input and result)

Explanation

The matrix *m* is rotated around the X-axis by an amount specified by *r*. In the rotation amount, 360° is represented as 4096, and 4096 is equivalent to 1.0.

The matrix is calculated as follows::

c=cos(*r*), s=sin(*r*)

The argument format is as follows::

m->*m*[i][j] : (1,3,12)

r:(1,3,12) (however 360° is equivalent to 1.0)

Return Value

The rotated matrix *m* is returned.

RotMatrixY

Rotates a matrix around the Y-Axis

Syntax

MATRIX* **RotMatrixY** (

long *r*,

MATRIX **m*

)

Arguments

r	Input rotation amount
m	Pointer to the matrix (used for both input and result)

Explanation

The matrix *m* is rotated around the Y-axis by an amount specified by *r*. In the rotation amount, 360° is represented as 4096, and 4096 is equivalent to 1.0.

The matrix is calculated as follows::

c=cos(*r*), s=sin(*r*)

The argument format is as follows::

m->*m*[i][j] : (1,3,12)

r:(1,3,12) (however 360° is 1.0)

Return Value

The rotated matrix *m* is returned.

RotMatrixZ

Rotates a matrix around the Z-Axis

Syntax

MATRIX* **RotMatrixZ** (

long *r*,

MATRIX **m*

)

Arguments

r	Input rotation amount
m	Pointer to the matrix (used for both input and result)

Explanation

The matrix *m* is rotated around the Z-axis by the amount specified by *r*. In the rotation amount, 360° is represented as 4096, and 4096 is equivalent to 1.0.

The matrix is calculated as follows::

c=cos(*r*), s=sin(*r*)

The argument format is as follows::

m->*m*[i][j] : (1,3,12)

r:(1,3,12) (however 360° is 1.0)

Return Value

The rotated matrix *m* is returned.

ScaleMatrix

Scales a matrix

Syntax

MATRIX* **ScaleMatrix** (
MATRIX**m*,
VECTOR**v*
)

Arguments

m	Pointer to matrix (both input and result)
v	Pointer to scale vector

Explanation

The matrix *m* is scaled by an amount specified by the scale vector *v*. Each element of *v* is an integer with 4096 equivalent to 1.0.

If *m* = ,*v* =

then *m*=

The argument format is as follows::

m->m[i][j] : (1,3,12)
v->vx,vy,vz : (1,19,12)

Return Value

The scaled matrix *m* is returned.

ScaleMatrixL

Scales a matrix

Syntax

MATRIX* **ScaleMatrixL** (
MATRIX**m*,
VECTOR**v*
)

Arguments

m	Pointer to matrix (both input and result)
v	Pointer to scale vector

Explanation

The matrix *m* is scaled by an amount specified by the scale vector *v*. Each element of *v* is an integer with 4096 equivalent to 1.0.

If *m* = ,*v* =

then *m*=

The argument format is as follows::

m->*m*[i][j] : (1,3,12)
v->vx,vy,vz : (1,19,12)

Return Value

The scaled matrix *m* is returned.

SetDispMask

Sets the display mask

Syntax

void **SetDispMask**(

int *mask*

)

Arguments

mask	0:	Inhibits screen display.
	1:	Enables screen display.

Explanation

The display is turned on or off according to the value of *mask*.

Return Value

None

StoreImage

Transfers image data from the frame buffer to memory

Syntax

```
int StoreImage (
RECT *recp,
u_long *p
)
```

Arguments

recp	Pointer to the source rectangular area in the frame buffer
p	Pointer to the upper left-hand corner of the destination rectangular area in main memory

Explanation

Data from the rectangular area pointed to by *recp* is transferred to the destination rectangular area in main memory pointed to by *p*.

Return Value

The slot number of the GPU's draw/command queue where the command resides is returned.

Notes

StoreImage() is a non-blocking function, therefore DrawSync() must be used to determine the completion of transfer.
The transfer is not affected by clipping or offsetting that may be in effect for the drawing environment.
The transfer area must be located within a drawable area: (0,0) - (1023,511).

TransMatrix

Translates a matrix

Syntax

MATRIX* **TransMatrix** (

MATRIX**m*,

VECTOR**v*

)

Arguments

m	Pointer to the matrix (used for both input and result)
v	Pointer to the translation vector

Explanation

The matrix *m* is translated by an amount specified by the translation vector *v*.

The argument format is as follows::

m->*m*[i][j] : (1,3,12)

m->t[i]: (1,31,0)

v->vx,vy,vz : (1,31,0)

Return Value

The translated matrix *m* is returned.

TransposeMatrix

Transposes a matrix

Syntax

MATRIX* **TransposeMatrix** (

MATRIX**m0*

MATRIX**m1*

)

Arguments

m0	Pointer to input matrix
m1	Pointer to result matrix

Explanation

The transpose of matrix *m0* is calculated and the result is placed in matrix *m1*.

The argument format is as follows::

m0->m[i][j] : (1,3,12)

m1->m[i][j] : (1,3,12)

Return Value

The transposed matrix *m1* is returned.

VSync

Waits for next vertical synchronization period

Syntax

int **VSync**(
int *mode*
)

Arguments

mode	0:	Blocks until vertical synchronization occurs.
	1:	Returns the elapsed time in HSync units from the last call to VSync().
	n:	(n>1) Blocks for n VSync periods from the last call to VSync().
	n:	(n<0) Returns absolute time from program start in VSync units.

Explanation

Vsync() waits for the start of the next vertical synchronization interval.

Return Value

mode>=0	The elapsed time since the last call to VSync() is returned in HSync units.
mode<0	The absolute time since program start is returned in VSync units.

VSyncCallback

Defines a function to be executed during each VSync period

Syntax

int **VSyncCallback**(

void (***func*)()

)

Arguments

func	Pointer to the callback function (function to be executed).

Explanation

Specifies that the function *func* will be called at the start of the vertical synchronization period.

When *func* is “0”, any function previously registered as a callback function is removed.

Return Value

None

Notes

Subsequent interrupts will be masked within *func*. Therefore, it is necessary to return quickly after *func* is called.

2

Sound Library

SndVolume

Volume

Structure

```
struct SndVolume {
        unsigned short left;
        unsigned short right;
};
```

Members

left	Left channel volume value
right	Right channel volume value

SsGetMute

Gets mute state

Syntax

char **SsGetMute** (void)

Explanation

The current mute state is obtained and returned.

Return Value

The mute state is returned as indicated below.

SS_MUTE_ON	Mute on
SS_MUTE_OFF	Mute off

See Also

SsSetMute()

SsGetMVol

Gets the main volume

Syntax

void **SsGetMVol** (
SndVolume **m_vol*
)

Arguments

m_vol	Pointer to SndVolume structure for storing result

Explanation

The value of the main volume is obtained and placed in the SndVolume structure pointed to by *m_vol*.

Return Value

None

See Also

SsSetMVol()

SsGetSerialAttr

Gets the value of a CD audio attribute

Syntax

char **SsGetSerialAttr** (

char *s_num*,

char *attr*

)

Arguments

s_num	Must be set to SS_CD	
attr	Attribute to be obtained	
	attr = SS_MIX	Mixing
	attr = SS_REV	Reverberation

Explanation

The value of the CD audio attribute specified by *attr* is obtained and returned.

Return Value

1 is returned if the specified attribute is on and 0 is returned if off.

See Also

SsSetSerialAttr()

SsGetSerialVol

Gets CD audio volume

Syntax

```
void SsGetSerialVol (
char s_num,
SndVolume *s_vol
)
```

Arguments

s_num	Must be set to SS_CD
s_vol	Pointer to SndVolume structure for storing result

Explanation

The current CD audio volume value is placed in the structure pointed to by *s_vol*.

Return Value

None

See Also

SsSetSerialVol()

SsIsEos

Returns the playback state of a song

Syntax

short **SsIsEos** (
short *access_num*,
short *seq_num*
)

Arguments

access_num	SEQ access number
seq_num	Must be set to 0

Explanation

The playback state of the song specified by *access_num* is obtained and its value is returned.

Return Value

1 is returned if the song is being played, otherwise 0 is returned.

SsPlayBack

Plays back SEQ data

Syntax

void **SsPlayBack** (
short *access_num*,
short *seq_num*,
short *l_count*
)

Arguments

access_num	SEQ access number
seq_num	Must be set to 0
l_count	Song repetition count

Explanation

The song which is currently playing is interrupted and the song which is specified by *access_num* is played.
The song repetition count is specified in *l_count*. If *l_count* is set to SSPLAY_INFINITY, the song will play continuously.

Return Value

None

See Also

SsSeqPlay()

SsSeqClose

Closes SEQ data

Syntax

void **SsSeqClose** (

short *seq_access_num*

)

Arguments

seq_access_num SEQ access number

Explanation

The SEQ data specified by *seq_access_num* is closed.

Return Value

None

See Also

SsSeqOpen()

SsSeqGetVol

Gets SEQ volume

Syntax

void **SsSeqGetVol** (
short *access_num*,
short *seq_num,*
short **voll*,
short **volr*
)

Arguments

access_num	SEQ access number
seq_num	Must be set to 0
voll	Pointer to SEQ left volume result value
volr	Pointer to SEQ right volume result value

Explanation

The current left and right volume values of the song specified by *access_num* are obtained and placed at the storage locations pointed to by *voll* and *volr,* respectively.

Return Value

None

See Also

SsSeqSetVol()

SsSeqOpen

Opens SEQ data

Syntax

short **SsSeqOpen** (
unsigned long* *addr*,
short *vab_id*
)

Arguments

addr	Pointer to start of SEQ data in main memory
vab_id	VAB ID

Explanation

The SEQ data beginning at the area in memory pointed to by *addr* and having the specified *vab_id* is opened and its SEQ access number is returned.
If an attempt is made to open more than 32 SEQ data groups, the function fails and a value of -1 is returned.

Return Value

The SEQ access number is returned if the SEQ data was successfully opened. -1 is returned if an error occurred.

See Also

SsSeqClose()

SsSeqPause

Pauses the playback of SEQ data

Syntax

void **SsSeqPause** (

short *seq_access_num*

)

Arguments

seq_access_num	SEQ access number

Explanation

The playback of the song specified by *seq_access_num* is paused.

Return Value

None

See Also

SsSeqPlay(), SsSeqReplay()

SsSeqPlay

Plays SEQ data

Syntax

void **SsSeqPlay** (
short *seq_access_num*,
char *play_mode*,
short *l_count*
)

Arguments

seq_access_num	SEQ access number
play_mode	Playback mode
	SSPLAY_PAUSE Pause immediately before playing
	SSPLAY_PLAY Begin playing immediately
l_count	Song repetition count

Explanation

The song specified by *seq_access_num* is played according to the specified playback mode.
The song repetition count is specified by *l_count*. If *l_count* is set to SSPLAY_INFINITY, the song will loop continuously.
If *play_mode* is set to SSPLAY_PAUSE, the song will be held in pause state until it is released. If play_mode is set to SSPLAY_PLAY, the song will begin playing immediately.

Return Value

None

See Also

SsSeqPause(), SsPlayBack(), SsSeqStop()

SsSeqReplay

Resumes playback of SEQ data

Syntax

void **SsSeqReplay** (
short *seq_access_num*
)

Arguments

seq_access_num SEQ access number

Explanation

The playback of the song specified by *seq_access_num,* which is currently paused, is resumed.
SsSeqReplay() can be used to continue playing a song that was previously paused with SsSeqPause() or SsSeqPlay().

Return Value

None

See Also

SsSeqPlay(), SsSeqPause()

SsSeqSetAccelerando

Accelerates the tempo

Syntax

void **SsSeqSetAccelerando** (
short *seq_access_num*,
long *tempo*,
long *v_time*
)

Arguments

seq_access_num	SEQ access number
tempo	Tempo
v_time	Time (in ticks)

Explanation

The tempo of the song specified by *seq_access_num* is incrementally increased to the value specified by *tempo* in the amount of time specified by *v_time*.
v_time is specified in number of ticks where 1 tick = 1/60 sec. SsSeqSetAccelerando() gradually increases the tempo to the specified *tempo*. *v_time* indicates how long it will take to reach that tempo.

Return Value

None

See Also

SsSeqSetRitardando()

SsSeqSetNext

Links SEQ data for playback

Syntax

void **SsSeqSetNext** (
short *seq_access_num1*,
short *seq_access_num2*
)

Arguments

seq_access_num1	SEQ access number
seq_access_num2	SEQ access number

Explanation

The SEQ data specified by *seq_access_num2* is linked to the SEQ data specified by *seq_access_num1*. This will cause the song specified by *seq_access_num2* to be played immediately after the song specified by *seq_access_num1*.

Return Value

None

SsSeqSetRitardando

Slows the tempo

Syntax

void **SsSeqSetRitardando** (
short *seq_access_num*,
long *tempo*,
long *v_time*
)

Arguments

seq_access_num	SEQ access number
tempo	Tempo
v_time	Time (in ticks)

Explanation

The tempo of the song specified by *seq_access_num* is incrementally decreased to the value specified by *tempo* in the amount of time specified by *v_time*.
v_time is specified in number of ticks where 1 tick = 1/60 sec. SsSeqSetRitardando() gradually decreases the tempo to the specified *tempo*. *v_time* indicates how long it will take to reach that tempo.

Return Value

None

See Also

SsSeqSetAccelerando()

SsSeqSetVol

Sets SEQ volume

Syntax

void **SsSeqSetVol** (
short *seq_access_num*,
short *voll*,
short *volr*
)

Arguments

seq_access_num	SEQ access number
voll	Left channel main volume
volr	Right channel main volume

Explanation

The main volume of the song specified by *seq_access_num* is set to the values specified by *voll* (left) and *volr* (right). The volume values can range from 0 to 127.

Return Value

None

See Also

SsSeqGetVol()

SsSeqStop

Ends playback of SEQ data

Syntax

void **SsSeqStop** (

short *seq_access_num*

)

Arguments

seq_access_num SEQ access number

Explanation

The playback of the song specified by *seq_access_num* is ended.

Return Value

None

See Also

SsSeqPlay()

SsSetMute

Sets or clears the mute state

Syntax

void **SsSetMute** (

char *mode*

)

Arguments

mode	Mute mode	
	SS_MUTE_ON	Mute on
	SS_MUTE_OFF	Mute off

Explanation

The mute state is set to the mode specified by *mode*.

Return Value

None

See Also

SsGetMute()

SsSetMVol

Sets the main volume

Syntax

void **SsSetMVol** (
short *voll*,
short *volr*
)

Arguments

voll	Left channel volume
volr	Right channel volume

Explanation

The main volume is set to the values specified by *voll* (left) and *volr* (right). Volume values can range from 0 to 127.

The main volume must be set before playing a song.

Return Value

None

See Also

SsGetMVol()

SsSetSerialAttr

Sets CD audio attribute

Syntax

void **SsSetSerialAttr** (

char *s_num*,

char *attr*,

char *mode*

)

Arguments

s_num	Must be set to SS_CD		
attr	Attribute		
		attr = SS_MIX	Mixing
		attr = SS_REV	Reverberation
mode	Desired mode		
		mode = SS_SON	Set attribute on
		mode = SS_SOFF	Set attribute off

Explanation

The CD audio attribute specified by *attr* is set according to the value of *mode*.

Return Value

None

See Also

SsGetSerialAttr()

SsSetSerialVol

Set CD audio volume

Syntax

void **SsSetSerialVol** (

short *s_num*,

short *voll*,

short *volr*

)

Arguments

s_num	Must be set to SS_CD
voll	Left channel volume
volr	Right channel volume

Explanation

The CD audio volume is set to the values specified by *voll* (left) and *volr* (right).

Volume values can range from 0 to 127.

Return Value

None

See Also

SsGetSerialVol()

SsSetTempo

Sets the tempo

Syntax

void **SsSetTempo** (
short *access_num*,
short *seq_num*,
short *tempo*
)

Arguments

access_num	SEQ access number
seq_num	Must be set to 0
tempo	Song tempo

Explanation

The tempo of the song specified by *access_num* is set to the value specified by *tempo*. SsSetTempo() can be used to change the tempo set by SsSeqPlay(). The new tempo will take effect immediately.

Return Value

None

SsUtAllKeyOff

Sets all voices to key off state

Syntax

void **SsUtAllKeyOff** (

short *mode*

)

Arguments

mode Must be 0

Explanation

Forcibly keys off all voices.

Return Value

None

See Also

SsUtKeyOn(), SsUtKeyOff(), SsSeqPlay()

SsUtChangePitch

Changes the pitch

Syntax

short **SsUtChangePitch** (
short *voice*,
short *vabId*,
short *prog*,
short *old_note*,
short *old_fine*,
short *new_note*,
short *new_fine*
)

Arguments

voice	Voice number
vabId	VAB ID
prog	Program number
old_note	Note number passed to SsUtKeyOn()
old_fine	Fine pitch passed to SsUtKeyOn() (note number)
new_note	Updated note number
new_fine	Updated pitch

Explanation

The pitch of the voice keyed on by SsUtKeyOn() is changed according to the values specified by *new_note* and *new_fine*.

Return Value

0 is returned if the pitch was successfully changed, otherwise -1 is returned.

See Also

SsUtPitchBend()

SsUtGetReverbType

Gets the reverberation type

Syntax

short **SsUtGetReverbType** (void)

Arguments

None

Explanation

The value of the current reverberation type is obtained and returned.

Return Value

The value of the current reverberation type is returned.

See Also

SsUtSetReverbType()

SsUtGetVVol

Gets voice volume

Syntax

short **SsUtGetVVol** (
short *vc*,
short **voll*,
short **volr*
)

Arguments

vc	Voice number
voll	Pointer to memory area for storing result volume (left)
volr	Pointer to memory area for storing result volume (right)

Explanation

The values of the volume specified by *vc* are obtained and placed in the storage locations specified by *voll* (left) and *volr* (right).

Return Value

0 is returned if the values were successfully obtained, otherwise -1 is returned.

See Also

SsUtSetVVol()

SsUtKeyOff

Keys off voice

Syntax

short **SsUtKeyOff** (
short *voice*,
short *vabId*,
short *prog*,
short *tone*,
short *note*
)

Arguments

voice	Voice number
vabId	VAB ID
prog	Program number
tone	Tone number
note	Pitch specification (note number) in half tones

Explanation

The voice that was keyed on by SsUtKeyOn() is keyed off.

Return Value

0 is returned if the voice was successfully keyed off, otherwise -1 is returned.

See Also

SsUtKeyOn(), SsUtAllKeyOff()

SsUtKeyOn

Keys on voice

Syntax

short **SsUtKeyOn** (
short *vabId*,
short *prog*,
short *tone*,
short *note*,
short *fine*,
short *voll*,
short *volr*
)

Arguments

vabId	VAB ID
prog	Program number
tone	Tone number
note	Pitch specification (note number) in half tones
fine	Fine pitch (100/127 cents)
voll	Volume (left)
volr	Volume (right)

Explanation

The voice specified by the *vabid*, *prog*, and other arguments is keyed on at the specified pitch and volume. The allocated voice number is returned.
note specifies the actual key that will be played and represents one of the 128 keys on the piano. *fine* specifies the pitch of that note.

Return Value

If the function is successful, the allocated voice number (0 to 23) is returned.

-1 is returned if the key on failed to complete.

See Also

SsUtKeyOff(), SsUtAllKeyOff()

SsUtPitchBend

Applies a pitch bend

Syntax

short **SsUtPitchBend** (
short *voice*,
short *vabId*,
short *prog*,
short *note*,
short *pbend*
)

Arguments

voice	Voice number
vabId	VAB ID
prog	Program number
note	Pitch specification (note number) in half tones.
pbend	Pitch bend value

Explanation

The pitch bend specified by *pbend* is applied to the voice keyed on by SsUtKeyOn().

Return Value

0 is returned if the pitch bend was successfully applied, otherwise -1 is returned.

See Also

SsUtChangePitch()

SsUtReverbOff

Turns off reverberation

Syntax

void **SsUtReverbOff** (void)

Arguments

None

Explanation

Reverberation is turned off.

Return Value

None

See Also

SsUtReverbOn()

SsUtReverbOn

Turns reverberation on

Syntax

void **SsUtReverbOn** (void)

Arguments

None

Explanation

Reverberation is turned on using the type and depth previously set with SsUtSetReverbType() and SsUtSetReverbDepth().

Return Value

None

See Also

SsUtReverbOff()

SsUtSetReverbDelay

Sets the amount of reverberation delay

Syntax

void **SsUtSetReverbDelay** (

short *delay*

)

Arguments

delay Reverberation delay amount (0~127)

Explanation

The amount of reverberation delay is set to the value specified by *delay*.

Reverberation delay applies only to echo-type and delay-type reverberation.

Return Value

None

See Also

SsUtSetReverbType()

SsUtSetReverbDepth

Sets the reverberation depth

Syntax

void **SsUtSetReverbDepth** (
short *ldepth*,
short *rdepth*
)

Arguments

ldepth	Left reverberation depth value (0~127)
rdepth	Right reverberation depth value (0~127)

Explanation

The reverberation depth is set to the values specified by *ldepth* (left) and *rdepth* (right).

Return Value

None

See Also

SsUtSetReverbType()

SsUtSetReverbFeedback

Sets the amount of reverberation feedback

Syntax

void **SsUtSetReverbFeedback** (

short *feedback*

)

Arguments

feedback	Amount of reverberation feedback (0~127)

Explanation

The amount of reverberation feedback is set to the value specified by *feedback*.

Reverberation feedback applies only to echo-type reverberation.

Return Value

None

See Also

SsUtSetReverbType()

SsUtSetReverbType

Sets the reverberation type

Syntax

short **SsUtSetReverbType** (
short *type*
)

Arguments

type Reverberation type

Type	Mode	Delay time *	Feedback*
SS_REV_TYPE_OFF	Off	X	X
SS_REV_TYPE_ROOM	Room	X	X
SS_REV_TYPE_STUDIO_A	Studio (small)	X	X
SS_REV_TYPE_STUDIO_B	Studio (medium)	X	X
SS_REV_TYPE_STUDIO_C	Studio (large)	X	X
SS_REV_TYPE_HALL	Hall	X	X
SS_REV_TYPE_SPACE	Space echo	X	X
SS_REV_TYPE_ECHO	Echo	O	O
SS_REV_TYPE_DELAY	Delay	O	O
SS_REV_TYPE_PIPE	Pipe echo	X	X

* Delay time and feedback amount can also be set

Explanation

The reverberation type specified by *type* is set. When the reverberation type is set, the reverberation depth is automatically set to 0.

When data is left in the reverberation work area, noise will appear when the reverberation depth is set. The following procedure can be used to circumvent this problem.

```
SsUtSetReverbType(SS_REV...);
SsUtReverbOn();
        :
Wait several seconds
        :
SsUtSetReverbDepth(64,64);

Number and type are as shown in the table above
```

Return Value

Type is returned if the reverberation type was successfully set, otherwise -1 is returned.

See Also

SsUtGetReverbType(), SsUtSetReverbDepth(), SsUtSetReverbFeedback(), SsUtSetReverbDelay()

SsUtSetVVol

Sets voice volume

Syntax

short **SsUtSetVVol** (
short *vc*,
short *voll*,
short *volr*
)

Arguments

vc	Voice number
voll	Volume (left)
volr	Volume (right)

Explanation

The volume of the voice specified by *vc* is set to the values specified by *voll* (left) and *volr* (right).

Return Value

0 is returned if the volume was successfully set, otherwise -1 is returned.

See Also

SsUtGetVVol()

SsVabClose

Closes VAB data

Syntax

void **SsVabClose**(

short *vab_id*

)

Arguments

vab_id	VAB ID

Explanation

The VAB data specified by *vab_id* is closed.

Return Value

None

See Also

SsVabTransfer()

SsVabTransfer

Accesses and transfers VAB waveform data

Syntax

short **SsVabTransfer** (
unsigned char **vh_addr*,
unsigned char **vb_addr,*
short *vabid,*
short *i_flag*
)

Arguments

vh_addr	Pointer to start of VH data
vb_addr	Pointer to start of VB data
vabid	VAB ID
i_flag	Must be set to 1

Explanation

The sound source header list (VH data) pointed to by *vh_addr* is accessed and the corresponding waveform data (VB data) pointed to by *vb_addr* is transferred to the SPU sound buffer. The VAB identification number is specified in *vabid.* When *vabid* is -1, SsVabTransfer() allocates and returns an available VAB ID number (0-15).

Return Value

The VAB ID number is returned if the VAB data was successfully transferred to the SPU sound buffer. If the function fails to complete, one of the following values is returned:

-1	VAB ID cannot be assigned or invalid VH
-2	Invalid VB
-3 or greater	Other errors

See Also

SsVabClose()

3

Standard C Library

abs

Absolute value

Syntax

```
#include <stdlib.h>
long abs (
long i
)
```

Arguments

i	Integer argument

Explanation

The absolute value of *i* is calculated and its value is returned.

Since int and long are declared as the same type in the R3000, this function is equivalent to labs().

Return Value

The absolute value of *i* is returned.

See Also

labs()

atoi

Converts a string to an integer

Syntax

```
#include <stdlib.h>
long atoi (
const char *s
)
```

Arguments

s	Pointer to a character string

Explanation

The string pointed to by *s* is converted to its integer equivalent and its value is returned.

atoi(s) is equivalent to (long)strtol(*s*,(char**)NULL). Since int and long are declared as the same type in the R3000, atoi() is also equivalent to atol().

Return Value

The integer value of the converted character string is returned.

See Also

atol(), strtol()

atol

Converts a string to an integer

Syntax

```
#include <stdlib.h>
long atol(
const char *s
)
```

Arguments

s	Pointer to a character string

Explanation

The string pointed to by *s* is converted to its integer equivalent and its value is returned.

atol(s) is equivalent to (long)strtol(*s*,(char**)NULL). Since int and long are declared as the same type in the R3000, atol() is also equivalent to atoi().

Return Value

The integer value of the converted character string is returned.

See Also

atoi(), strtol()

bcmp

Compares two memory blocks

Syntax

```
#include <memory.h>
long bcmp(
char *b1,
char *b2,
long n
)
```

Arguments

b1	Pointer to memory block 1
b2	Pointer to memory block 2
n	Number of bytes to compare

Explanation

The first *n* bytes of the memory blocks pointed to by *b1* and *b2* are compared and the result is indicated in the return value.

Return Value

The result of the comparison is indicated in the return value as shown below.

Result	Return Value
b1<b2	<0
b1=b2	=0
b1>b2	>0

See Also

memcmp()

bcopy

Copies memory

Syntax

```
#include <memory.h>
void bcopy(
char *src,
char *dest,
long n
)
```

Arguments

src	Pointer to source data area
dest	Pointer to destination data area
n	Number of bytes to copy

Explanation

The first *n* bytes of the data area pointed to by *src* are copied to the data area pointed to by *dest*.

Return Value

None

See Also

memcpy()

bsearch

Binary search

Syntax

```
#include <stdlib.h>
void *bsearch (
const void *key,
const void *base,
size_t n,
size_t w,
long(*fcmp)(const void *, const void *)
)
```

Arguments

key	Pointer to data which is the retrieval key
base	Pointer to destination area to be searched
n	Number of elements in the data area
w	Size of 1 element
fcmp	Pointer to comparison function

Explanation

A binary search for the retrieval key pointed to by *key* is made in the destination area pointed to by *base* using the comparison function *fcmp*. A pointer to the found item is returned.

base is considered to be an array of *n* items with size *w*. The search is performed looking for the first item that matches the retrieval key. *fcmp* must return a value representing the result of the comparison as in bcmp().

Return Value

The address of the first item matching the retrieval key is returned. 0 is returned if a matching item was not found.

bzero

Clears memory

Syntax

```
#include <memory.h>
void *bzero (
unsigned char *p,
int n
}
```

Arguments

p	Pointer to a memory block
n	Number of bytes to clear

Explanation

The first *n* bytes of the memory block pointed to by *p* are cleared.

Return Value

p, a pointer to the start of the memory block is returned.

See Also

bcopy(), bcmp()

calloc

Allocates memory

Syntax

```
#include <stdlib.h>
void *calloc (
size_t n,
size_t s
)
```

Arguments

n	Number of blocks to allocate
s	Block size

Explanation

An *n* x *s* byte block is allocated from the heap. The allocated memory is cleared.

Return Value

A pointer to the allocated memory block is returned. NULL is returned if the allocation failed.

See Also

malloc(), realloc(), free()

free

Frees allocated memory blocks

Syntax

```
#include <stdlib.h>
void free (
void*block
)
```

Arguments

block	Pointer to allocated memory to be freed

Explanation

The memory area pointed to by *block* and previously allocated by calloc(), malloc() or realloc() is freed.

Return Value

None

See Also

calloc(), malloc(), realloc()

getc

Gets one character from an input stream

Syntax

```
#include <stdio.h>
char getc (
FILE *stream
)
```

Arguments

stream	Pointer to an input stream

Explanation

The next character is read from the input stream pointed to by *stream.* The character is returned by the function.

Return Value

The next character read from the input stream is returned if the function was successful. Otherwise, EOF is returned in case of end-of-file or error.

See Also

getchar(), gets()

getchar

Gets one character from the standard input stream

Syntax

#include <stdio.h>

char **getchar**(void)

Arguments

None

Explanation

The next character is read from the standard input stream and returned.

getchar() is equivalent to getc(stdin).

Return Value

The next character read from the standard input stream is returned if the function was successful. Otherwise, EOF is returned in case of end-of-file or error.

See Also

getc(), gets()

gets

Reads a line from the standard input stream

Syntax

```
#include <stdio.h>
char *gets (
char *s
)
```

Arguments

s	Pointer to destination buffer

Explanation

A line is read from the standard input stream (stdin) and placed in the buffer pointed to by *s*. The end of line is indicated by a newline character. The newline is discarded and replaced by a null character at the end of the string pointed to by *s*.

Return Value

A pointer to the character string *s* is returned if the function was successful. NULL is returned in case of end-of-file or error.

See Also

getc(), getchar()

isXXX

Performs character testing

Syntax

```
#include <ctype.h>
long isXXX(
long c
)
```

Arguments

c	Input character

Explanation

isXXX defines a general class of macros. The character *c* is tested according to one of the conditions shown in the following table. The specified macro returns a non-zero value if the test succeeded and 0 if the condition was not met.

Function Name	Condition
isalnum	isalpha(c) || isdigit(c)
isalpha	isupper(c) || islower(c)
isascii	ASCII characters
iscntrl	Control characters
isdigit	Decimal characters
isgraph	Printable characters except spaces
islower	Lower case characters
isprint	Printable characters including spaces
ispunct	Printable characters except spaces, English letters and numbers
isspace	Spaces, page breaks, line feeds, carriage returns, tabs
isupper	Upper case letters
isxdigit	Hexadecimal characters

Return Value

A non-zero value is returned if the input character *c* satisfied the condition. 0 is returned if the condition was not satisfied.

labs

Absolute value

Syntax

```
#include <stdlib.h>
long labs (
long i
)
```

Arguments

i	Integer argument

Explanation

The absolute value of *i* is calculated and its value is returned.

Since int and long are declared as the same type in the R3000, this function is equivalent to abs().

Return Value

The absolute value of *i* is returned.

See Also

abs()

longjmp

Non-local jump

Syntax

```
#include <setjmp.h>
void longjmp (
jmp_buf p,
int val
)
```

Arguments

p	Environment storage variable
val	Specified return value of setjmp()

Explanation

The program jumps to the location specified in the environment storage area pointed to by *p* which was set by a previous call to setjmp().

Return Value

After a longjmp() is completed, program execution continues as if the corresponding invocation of setjmp() had just returned the value specified by *val*. longjmp() cannot cause setjmp() to return the value 0; if *val* is 0, setjmp() returns the value 1.

See Also

setjmp()

malloc

Allocates memory

Syntax

```
#include <stdlib.h>
void *malloc (
size_t s
)
```

Arguments

s	Number of bytes to allocate

Explanation

A block of *s* bytes is allocated from the heap.

Return Value

A pointer to the allocated memory block is returned. NULL is returned if the allocation failed.

* At the time of user program activation heap memory is defined as follows:

Bottom address	Highest address of module + 4
Top address	Highest available memory - 64KB

See Also

calloc(), realloc(), free()

memchr

Searches a memory block for a specified character

Syntax

```
#include <memory.h>
void *memchr (
const void *s,
long c,
size_t n
)
```

Arguments

s	Pointer to a memory block
c	Desired character
n	Maximum number of bytes to search

Explanation

The first occurrence of the character *c* in the memory block pointed to by *s* is located and a pointer to the character is returned. At most *n* bytes of the memory block will be searched.

Return Value

A pointer to the located character *c* is returned. NULL is returned if *c* cannot be found in the memory block.

memcmp

Compares two memory blocks

Syntax

```
#include <memory.h>
long memcmp (
const void *s1,
const void *s2,
size_t n
)
```

Arguments

s1	Pointer to memory block 1
s2	Pointer to memory block 2
n	Maximum number of bytes to compare

Explanation

The first *n* bytes of the memory blocks pointed to by *s1* and *s2* are compared and the result is indicated in the return value.

Return Value

The result of the comparison is indicated in the return value as shown below.

Result	Return Value
s1<s2	<0
s1=s2	=0
s1>s2	>0

See Also

bcmp()

memcpy

Copies a memory block

Syntax

```
#include <memory.h>
void *memcpy (
void *dest,
const void *src,
size_t n
)
```

Arguments

dest	Pointer to destination memory block
src	Pointer to source memory block
n	Number of bytes to copy

Explanation

The first *n* bytes of the memory block pointed to by *src* are copied to the memory block pointed to by *dest*.

Return Value

dest, a pointer to the destination memory block is returned.

See Also

bcopy()

memmove

Copies a memory block

Syntax

```
#include <memory.h>
void  *memmove (
void *dest,
const void *src,
size_t n
)
```

Arguments

dest	Pointer to destination memory block
src	Pointer to source memory block
n	Number of bytes to copy

Explanation

The first *n* bytes of the memory block pointed to by *src* are copied to the memory block pointed to by *dest*.

The block is copied correctly even if *src* and *dest* overlap.

Return Value

dest, a pointer to the destination memory block is returned.

memset

Writes a specified character to a memory block

Syntax

```
#include <memory.h>
void *memset (
const void *s,
long c,
size_t n
)
```

Arguments

s	Pointer to destination memory block
c	Fill character
n	Number of characters

Explanation

The first *n* bytes of the memory block pointed to by *s* are filled with the character *c*.

Return Value

s, a pointer to the destination memory block is returned.

printf

Writes formatted output to the standard output stream

Syntax

```
#include <stdio.h>
long printf (
const char *fmt[,argument ...]
)
```

Arguments

fmt	Input format conversion string

Explanation

Writes formatted output to stdout, the standard output stream.
Please refer to the C language reference for a detailed explanation of the input format conversion string. The format specifiers “f”, “e”, “E”, “g” and “G” cannot be used. printf2() of the mathematical function service is used to display floating-point numbers.

Return Value

The length of the output character string is returned. NULL is returned if an error occurred.

See Also

sprintf(), printf2()

putc

Writes one character to an output stream

Syntax

```
#include <stdio.h>
void putc (
long c,
FILE *stream
)
```

Arguments

c	Output character
stream	Pointer to output stream

Explanation

The character *c* is written to the output stream pointed to by *stream*.

Return Value

None

See Also

putchar(), puts()

putchar

Writes one character to the standard output stream

Syntax

```
#include <stdio.h>
long putchar(
char c,
)
```

Arguments

c	Output character

Explanation

The character *c* is written to the standard output stream.

putchar() is equivalent to putc(stdout).

Return Value

None

See Also

putc(), puts()

puts

Writes a character string to the standard output stream

Syntax

```
#include <stdio.h>
void puts (
const char *s
)
```

Arguments

s	Output character string

Explanation

The character string *s* is written to the standard output stream (stdout). A newline character is appended to the end of the string.

Return Value

None

See Also

putc(), putchar()

qsort

Quick sort

Syntax

```
#include <stdlib.h>
void qsort (
void *base,
size_t n,
size_t w,
long (*fcmp)(const void *, const void *)
)
```

Arguments

base	Pointer to array in memory to be sorted
n	Number of elements
w	Size of each element
fcmp	Pointer to comparison function

Explanation

The array pointed to by *base* is sorted using a quicksort algorithm.

n specifies the number of elements in the array and *w* is the size of each element.

fcmp is the comparison function and must return a value representing the result of the comparison as in bcmp().

qsort() calls malloc() internally so there must be sufficient space on the heap to perform the sort.

Return Value

None

rand

Generates a random number

Syntax

#include <stdlib.h>

long **rand** (void)

Arguments

None

Explanation

A pseudo-random number is generated between 0 and RAND_MAX(0x7FFF=32767) and its value is returned.

Return Value

The generated pseudo random number is returned.

See Also

srand()

realloc

Reallocates memory

Syntax

```
#include <stdlib.h>
void *realloc (
void *block,
size_t s
)
```

Arguments

block	Pointer to area to be reallocated
s	Size of new area

Explanation

The previously allocated area of storage pointed to by *block* is reallocated to the size specified by *s*. realloc() is identical to malloc() if *block* is NULL.

Return Value

A pointer to the reallocated block is returned. The reallocated block may be located at a different address than the original block. NULL is returned if the allocation failed in which case the original block is not released.

See Also

calloc(), malloc(), free()

setjmp

Defines destination of longjmp()

Syntax

```
#include <setjmp.h>
int setjmp (
jmp_buf p
)
```

Arguments

p	Environment storage variable

Explanation

Saves environmental information (such as the stack pointer) in the memory area pointed to by *p* for later use by longjmp().

Return Value

When setjmp() is directly invoked, 0 is returned. If the return is from a call to longjmp(), *val*, the argument to longjmp() is returned.

See Also

longjmp()

sprintf

Writes formatted output to a character string

Syntax

```
#include <stdio.h>
long sprintf(
char *s,
const char *fmt[,argument...]
)
```

Arguments

s	Pointer to storage area holding converted character string
fmt	Input format conversion string

Explanation

Writes formatted output to a character string and places the result in the storage area pointed to by *s*.

Please refer to the C language reference for a detailed explanation of the input format conversion string. The format specifiers “f”, “e”, “E”, “g” and “G” cannot be used. sprintf2() of the mathematical function service is used to display floating-point numbers.

Return Value

The length of the output character string is returned. NULL is returned if an error occurred.

See Also

printf(), sprintf2()

srand

Initializes the random number generator

Syntax

```
#include <stdlib.h>
void srand (
unsigned int seed
)
```

Arguments

seed	Random number seed

Explanation

Sets the new seed for the random number generator to the value specified by *seed*. The random number generator is reinitialized to its initial starting point by specifying a seed of 1.

Return Value

None

See Also

rand()

strcat

Concatenates two character strings

Syntax

```
#include <strings.h>
char *strcat (
char *dest,
const char *src
)
```

Arguments

dest	Pointer to destination character string
src	Pointer to source character string

Explanation

The string pointed to by *src* is concatenated to the end of the string pointed to by *dest*.

A pointer to the concatenated string is returned.

Return Value

dest, a pointer to the concatenated string is returned.

See Also

strncat()

strchr

Searches for the first occurrence of a specified character in a character string

Syntax

```
#include <strings.h>
char *strchr (
const char *s,
long c
)
```

Arguments

s	Pointer to a character string
c	Target character

Explanation

The first occurrence of the character c is located in the string pointed to by s. A pointer to the located character is returned.

Return Value

A pointer to the first occurrence of the character *c* within the string *s* is returned.
NULL is returned if *c* cannot be found within *s*.

strcmp

Compares two character strings

Syntax

```
#include <strings.h>
long  strcmp (
const char *s1,
const char *s2
)
```

Arguments

s1	Pointer to string 1
s2	Pointer to string 2

Explanation

The string pointed to by *s1* is compared with the string pointed to by *s2*, character-by-character as unsigned chars, and the result is indicated in the return value.

Return Value

The result of the comparison is indicated in the return value as shown below.

Result	Return Value
s1<s2	<0
s1=s2	=0
s1>s2	>0

strcpy

Copies a character string

Syntax

```
#include <strings.h>
char *strcpy (
char *dest,
const char *src
)
```

Arguments

dest	Pointer to destination memory area
src	Pointer to source character string

Explanation

The string pointed to by *src* is copied to the memory area pointed to by *dest*.

Return Value

dest, a pointer to the destination string is returned.

See Also

strncpy()

strcspn

Searches for a character string comprising only characters not included in a specified character set

Syntax

```
#include <strings.h>
size_t strcspn (
const char *s1,
const char *s2
)
```

Arguments

s1	Pointer to target character string to be searched
s2	Pointer to string which contains the character set

Explanation

The character string starting from the beginning of the string pointed to by *s1* that does not contain any characters that are contained in the character set pointed to by *s2* is located and its length is returned.

Return Value

The length of the segment within the string pointed to by *s1* that does not contain any characters from the string pointed to by *s2* is returned.

strlen

Finds the length of a character string

Syntax

```
#include <strings.h>
long strlen (
const char *s
)
```

Arguments

s	Pointer to a character string

Explanation

The length of the string pointed to by *s* (i.e. the number of characters in the string) is obtained and its value is returned.

Return Value

The length of the string pointed to by *s* is returned.

strncat

Concatenates two character strings

Syntax

```
#include <strings.h>
char  *strncat (
char *dest,
const char *src,
size_t n
)
```

Arguments

dest	Pointer to destination character string
src	Pointer to source character string
n	Number of characters from source string to concatenate

Explanation

n characters from the string pointed to by *src* are concatenated to the end of the string pointed to by *dest*. The concatenated string is returned.

Return Value

dest, a pointer to the concatenated string is returned.

strncmp

Compares two character strings

Syntax

```
#include <strings.h>
long stncmp (
const char *s1,
const char *s2,
size_t n
)
```

Arguments

s1	Pointer to string 1
s2	Pointer to string 2
n	Number of characters to compare

Explanation

The first *n* characters of the string pointed to by *s1* are compared to the corresponding characters of the string pointed to by *s2*, character-by-character as unsigned chars, and the result is indicated in the return value.

Return Value

The result of the comparison is indicated in the return value as shown below.

Result	Return
s1<s2	<0
s1=s2	=0
s1>s2	>0

strncpy

Copies a character string

Syntax

```
#include <strings.h>
char  *strncpy (
char *dest,
const char *src,
size_t n
)
```

Arguments

dest	Pointer to destination memory area
src	Pointer to source character string
n	Number of bytes to copy

Explanation

n bytes of the string pointed to by *src* are copied to the memory area pointed to by *dest*.

Return Value

dest, a pointer to the destination string is returned.

strpbrk

Searches for the first occurrence of a specified character in a character set

Syntax

```
#include <strings.h>
char *strpbrk (
const char *s1,
const char *s2
)
```

Arguments

s1	Pointer to string to be searched
s2	Pointer to string containing the character set

Explanation

The character string pointed to by *s1* is searched for the first occurrence of any character contained in the string pointed to by *s2*. A pointer to the character within *s2* is returned.

Return Value

A pointer to the character within *s2* is returned. NULL is returned if none of the characters in the string pointed to by *s2* are contained within the character set.

strrchr

Searches for the last occurrence of a specified character in a character string

Syntax

```
#include <strings.h>
char *strrchr (
const char *s,
long c
)
```

Arguments

s	Pointer to a character string
c	Target character

Explanation

The last occurrence of the character *c* is located in the character string pointed to by *s*. A pointer to the located character is returned.

Return Value

A pointer to the last occurrence of the character *c* within the string pointed to by *s* is returned. NULL is returned if *c* cannot be found within the string.

strspn

Searches for a character string comprising only characters in a specified character set

Syntax

```
#include <strings.h>
size_t strspn (
const char *s1,
const char *s2
)
```

Arguments

s1	Pointer to target character string to be searched
s2	Pointer to string containing the character set

Explanation

The character string starting from the beginning of the string pointed to by *s1* that contains only characters from the string pointed to by *s2* is located and its length is returned.

Return Value

The length of the segment within the string pointed to by *s1* that contains only characters from the string pointed to by *s2* is returned.

strstr

Searches for the occurrence of a specified partial character string

Syntax

```
#include <strings.h>
char *strstr (
const char *s1,
const char *s2
)
```

Arguments

s1	Pointer to target character string to be searched
s2	Pointer to character string to be located

Explanation

The first occurrence of the string pointed to by *s2* is located within the string pointed to by *s1*. A pointer to the position of *s2* within the string pointed to by *s1* is returned.

Return Value

A pointer to the position of the located string pointed to by *s2* within the string pointed to by *s1* is returned. NULL is returned if the string pointed to by *s2* is not contained within the string pointed to by *s1*.

strtok

Searches for a character string bounded by characters in a specified set of delimiters

Syntax

```
#include <strings.h>
char *strtok (
char *s1,
const char *s2
)
```

Arguments

s1	Pointer to target character string to search
s2	Pointer to character set containing token delimiters

Explanation

A sequence of calls to strtok() breaks the string pointed to by *s1* into a sequence of tokens, each of which is delimited by a character from the string pointed to by *s2*. The first call in the sequence has *s1* as its first argument, and is followed by calls with a NULL pointer as their first argument.

The first call in the sequence searches the string pointed to by *s1* for the first character that is not contained in the current separator string pointed to by *s2*. If no such character is found, then there are no tokens in the string pointed to by *s1* and the function returns a NULL pointer. If such a character is found, it is the start of the first token.

strtok() then searches from there for a character that is contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by *s1*, and subsequent searches for a token will return a NULL pointer. If such a character is found, it is overwritten by a NULL character, which terminates the current token. strtok() saves a pointer to the following character, from which the next search for a token will start.

Each subsequent call, with a NULL pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

Return Value

A pointer to the first character of a token is returned, or NULL is returned if there is no token.

strtol

Converts a character string to an integer

Syntax

```
#include <stdlib.h>
long strtol (
const char *s,
char **endp
)
```

Arguments

s	Pointer to a character string
endp	Pointer to a storage area that will contain a pointer to the character that stopped the conversion

Explanation

The character string pointed to by *s* is converted to an integer and its value is returned. Note that long and int are declared as the same type in the R3000.

The string pointed to by *s* must be in one of the following formats.

[ws][sn][ddd]

[ws]	White space (can be omitted)
[sn]	Sign (can be omitted)
[ddd]	Number string (can be omitted)

strtol() stops conversion when a character is encountered that cannot be converted.

endp A pointer to the character that terminated the conversion is placed at the storage location specified by *endp* unless *endp* is NULL.

Return Value

The result of converting the character string pointed to by *s* to an integer is returned if the conversion was successful. 0 is returned if an error occurred.

See Also

strtoul()

strtoul

Converts a character string to an unsigned integer

Syntax

```
#include <stdlib.h>
unsigned long strtoul (
const char *s,
char **endp
)
```

Arguments

s	Pointer to a character string
endp	Pointer to a storage area that will contain a pointer to the character that stopped the conversion

Explanation

The character string pointed to by *s* is converted to an unsigned integer and its value is returned. Note that long and int are declared as the same type in the R3000.

The string pointed to by *s* must be in one of the following formats.

[ws][sn][ddd]

[ws]	White space (can be omitted)
[sn]	Sign (can be omitted)
[ddd]	Number string (can be omitted)

strtol() stops conversion when a character is encountered that cannot be converted. A pointer to the character that terminated the conversion is placed at the storage location specified by *endp* unless *endp* is NULL.

Return Value

The result of converting the character string pointed to by *s* to an unsigned integer is returned if the conversion was successful. 0 is returned if an error occurred.

See Also

strtol()

toascii

Masks off bit 7 of a specified character

Syntax

```
#include <ctype.h>
long toascii (
long c
)
```

Arguments

c	Input character

Explanation

Bit 7 of the input character *c* is masked off.

Return Value

The result of the masking operation is returned.

tolower

Converts a character to lower case

Syntax

```
#include <ctype.h>
long tolower (
long c
)
```

Arguments

c	Input character

Explanation

The input character *c* is converted to lower case.

Return Value

The lower case character corresponding to the input character *c* is returned.

toupper

Converts a character to upper case

Syntax

```
#include <ctype.h>
long toupper (
long c
)
```

Arguments

c	Input character

Explanation

The input character *c* is converted to upper case.

Return Value

The upper case character corresponding to the input character *c* is returned.

4

Math Library

acos

Arccosine

Structure

double acos (

double *x*

)

Arguments

x	Argument. Range is [-1 to 1].

Explanation

The arccosine of *x* is calculated and its value is returned.

Return value

The arccosine of *x* (acos(*x*)) is returned. The result range is [0, pi].

Error processing is shown in the table below.

Condition	Return value	Error
fabs(*x*)>1	0	Domain error

See Also

cos(), asin(), atan()

asin

Arcsine

Structure

double asin (

double *x*

)

Arguments

x	Argument. Range is [-1 to 1].

Explanation

The arcsine of *x* is calculated and its value is returned.

Return value

The arcsine of *x* (asin(*x*)) is returned. The result range is [-pi/2, pi/2].

Error processing is shown in the table below.

Condition	Return value	Error
fabs(*x*)>1	0	Domain error

See Also

sin(), acos(), atan()

atan

Arctangent

Structure

double atan (

double *x*

)

Arguments

x	Argument.

Explanation

The arctangent of *x* is calculated and its value is returned.

Return value

The arctangent of *x* (atan(*x*)) is returned. The result range is [-pi/2, pi/2].

See Also

tan(), asin(), acos(), atan2()

atan2

Arctangent

Structure

double atan2 (

double *x*,

double *y*

)

Arguments

x	Dividend
y	Divisor

Explanation

The arctangent of *x*/*y* is calculated and its value is returned.

Return value

The arctangent of *x*/*y* (atan(*x*/*y*)) is returned. The result range is [-pi, pi].

Error processing is shown in the table below.

Condition	Return value	Error
x==0 && *y*==0	0	Domain error

See Also

atan()

atof

Converts a character string to a floating-point number

Syntax

double **atof**(
const char *s
)

Arguments

s	Pointer to a character string

Explanation

The character string pointed to by *s* which represents a floating point number, is converted to a double-precision floating point value.

Return Value

The result of converting the input string pointed to by *s* to a double-precision value is returned. If the value exceeds the range that can be expressed, either +HUGE_VAL(1.797693134862316e+308) or -HUGE_VAL is returned according to the sign. 0 is returned if an underflow occurs.

Notes

Error processing is shown in the following table.

Condition	Return Value	Error
Outside the range that can be expressed	+/- HUGE_VAL	Domain error
Underflow	0	Domain error

See Also

strtod()

ceil

Finds the smallest integer not less than x (ceiling function)

Structure

double ceil (
double *x*
)

Arguments

x	Argument

Explanation

The smallest integer greater than or equal to *x* is obtained and its corresponding double-precision floating point value is returned.

Return value

The smallest integer (in double-precision floating point format) that is not less than *x* is returned.

See Also

floor()

cos

Cosine

Structure

double cos (

double *x*

)

Arguments

x	Argument in radians

Explanation

The cosine of *x* is calculated and its value is returned.

Return value

The cosine of *x* (cos(*x*)) is returned.

See Also

sin(), tan(), acos()

cosh

Hyperbolic cosine

Structure

```
double cosh (
double x
)
```

Arguments

x	Argument in radians

Explanation

The hyperbolic cosine of *x* is calculated and its value is returned.

Return value

The hyperbolic cosine of *x* (cosh(*x*)) is returned.

See Also

sinh(), tanh()

exp

Exponent

Syntax

double **exp** (

double *x*

)

Arguments

x Argument

Explanation

The exponential of *x* (e to the *x*-th power) is calculated and the result is returned.

Return Value

e to the *x*-th power (e^x) is returned.

See Also

pow(), log()

fabs

Absolute value (macro)

Syntax

fabs (

double *x*

)

Arguments

x	Floating-point argument

Explanation

The absolute value of the argument *x* is calculated and its value is returned.

Return Value

The absolute value of *x* is returned.

Notes

fabs() is a macro.

floor

Finds the largest integer not greater than x (base function)

Structure

double **floor** (
double *x*
)

Arguments

x	Argument

Explanation

The largest integer less than or equal to *x* is obtained and its corresponding double-precision floating point value is returned.

Return value

The largest integer (in double-precision floating point format) that is not greater than *x* is returned.

See Also

ceil()

fmod

Calculates floating point remainder of x/y

Structure

```
double fmod (
double x,
double y
)
```

Arguments

x	Floating point dividend
y	Floating point divisor

Explanation

The remainder of x/y is calculated and its value is returned as a double-precision floating point number.

Return value

The double precision floating point number remainder of *x*/*y* is returned.

Notes

The sign of the return value will be the same as the sign of *x*. 0 is returned if *y* is 0.

frexp

Divides a floating point number into a normalized mantissa and an exponent ( )

Structure

```
double frexp (
double x,
int *n
)
```

Arguments

x	Argument
n	Pointer to a storage location that stores the exponent

Explanation

The floating point number *x* is separated into a normalized mantissa [1/2,1) and an exponent ( ). The exponent is placed at the storage location pointed to by *n*.
The normalized mantissa is returned by the function.

Return value

The normalized mantissa [1/2, 1) is returned.

Notes

[a,b) indicates a value in the range: a ≤ value ≤ b.

hypot

Absolute value of a complex number

Structure

```
double hypot (
double x,
double y
)
```

Arguments

x	Real part
y	Imaginary part

Explanation

The absolute value of the complex number (x+iy) is calculated and its value is returned.

Return value

The square root of the sum of x^2 and y^2 is returned.

ldexp

Calculates a real number from a mantissa and an exponent

( )

Structure

```
double ldexp (
double x,
long n
)
```

Arguments

x	Floating point mantissa
n	Integer exponent

Explanation

The real number is calculated and its value is returned.

Return value

The value $x \times 2^n$ is returned.

log

Natural logarithm

Syntax

double **log** (

double *x*

)

Arguments

x Argument

Explanation

The natural logarithm of *x* is calculated and the result is returned.

Return Value

The natural logarithm of *x* (ln(*x*)) is returned.

Notes

x must be greater than 0, otherwise an error will occur as shown below.

Condition	Return Value	Error
x<0	0	Domain error
x==0	1	Range error

See Also

exp(), log10()

log10

Common logarithm

Syntax

double **log10** (

double *x*

)

Arguments

x Argument

Explanation

The common logarithm of *x* is calculated and the result is returned.

Return Value

The common logarithm of *x* (log10(*x*)) is returned.

Notes

x must be greater than 0, otherwise an error will occur as shown below.

Condition	Return Value	Error
x<0	0	Domain error
x==0	1	Range error

See Also

log()

modf

Separates a floating point number into integer and fractional parts

Structure

```
double modf (
double x,
double *y
)
```

Arguments

x	Floating point number
y	Pointer to a buffer for storing the integer part

Explanation

x is separated into an integer part and a fractional part. The integer part is placed at the storage location pointed to by *y*. The fractional part is returned by the function.

Return value

The fractional part of *x* is returned.

Notes

The sign for both integer parts and fractional parts will be the same as the sign of *x*.

pow

x to the y-th power

Syntax

double **pow** (

double *x*,

double *y*

)

Arguments

x	Base number
y	Power

Explanation

x to the *y*-th power is calculated and the result is returned.

Return Value

x to the *y*-th power (x^y) is returned.

Notes

Error processing is shown in the following table.

Condition	Return Value	Error
x==0 && y>0	0	
x==0 && y<=0	1	Domain error
x<0 && “y is not an integer”	0	Domain error

See Also

exp()

printf2

Writes formatted output to the standard output stream (supports float and double type)

Structure

long printf2(
const char **fmt*, [argument...]
)

Arguments

fmt	Input format conversion string

Explanation

Writes formatted output to stdout, the standard output stream.
Please refer to the C language reference for a detailed explanation of the input format conversion string. The format specifiers “f”, “e”, “E”, “g” and “G” can be used.
printf2() requires more stack space than print().

Return Value

The length of the output character string is returned. NULL is returned if an error occurred.

See Also

sprintf2()

sin

Sine

Structure

double sin (

double *x*

)

Arguments

x	Argument in radians

Explanation

The sine of *x* is calculated and its value is returned.

Return value

The sine of *x* (sin(*x*)) is returned.

See Also

cos(), tan(), asin()

sinh

Hyperbolic sine

Structure

```
double sinh (
double x
)
```

Arguments

x	Argument in radians

Explanation

The hyperbolic sine of *x* is calculated and its value is returned.

Return value

The hyperbolic sine of *x* (sinh(*x*)) is returned.

See Also

cosh(), tanh()

sprintf2

Writes formatted output to a character string (supports float and double type)

Structure

long sprintf2(
char **s*,
const char **fmt*, [argument...]
)

Arguments

s	Pointer to storage area holding converted character string
fmt	Input format conversion string

Explanation

Writes formatted output to a character string and places the result in the storage area pointed to by s.
Please refer to the C language reference for a detailed explanation of the input format conversion string. The format specifiers “f”, “e”, “E”, “g” and “G” can be used.
sprintf2() requires more stack space than sprintf().

Return Value

The length of the output character string is returned. NULL is returned if an error occurred.

See Also

printf2()

sqrt

Square root

Structure

double sqrt (

double *x*

)

Arguments

x	Argument (must be ≥ 0)

Explanation

The square root of *x* is calculated and its value is returned.

Return value

The square root of *x* (sqrt(*x*)) is returned.

Error processing is shown in the table below.

Condition	Return value	Error
x<0	0	Domain error

strtod

Converts a character string to a floating-point number

Syntax

double **strtod**(
const char **s*,
char ***endp*
)

Arguments

s	Pointer to a character string
endp	Pointer to a storage area that will contain a pointer to the character that stopped the conversion

Explanation

The character string pointed to by *s* is converted to a double-precision floating point number and its value is returned.

The string pointed to by *s* must be in one of the following formats.

[ws][sn][ddd]

[ws]	White space (can be omitted)
[sn]	Sign (can be omitted)
[ddd]	Number string (can be omitted)

strtod() stops conversion when a character is encountered that cannot be converted. A pointer to the character that terminated the conversion is placed at the storage location specified by *endp* unless *endp* is NULL.

Return Value

The result of converting the input string pointed to by *s* to a double-precision value is returned. If the value exceeds the range that can be expressed, either +HUGE_VAL(1.797693134862316e+308) or -HUGE_VAL is returned according to the sign. 0 is returned if an underflow occurs.

Notes

Error processing is shown in the following table.

Condition	Return Value	Error
Outside the range that can be expressed	+/- HUGE_VAL	Domain error
Underflow	0	Domain error

tan

Tangent

Structure

```
double tan (
double x
)
```

Arguments

x	Argument in radians

Explanation

The tangent of *x* is calculated and its value is returned.

Return value

The tangent of *x* (tan(*x*)) is returned.

See Also

sin(), cos(), atan()

tanh

Hyperbolic tangent

Structure

```
double tanh (
double x
)
```

Arguments

x	Argument in radians

Explanation

The hyperbolic tangent of *x* is calculated and its value is returned.

Return value

The hyperbolic tangent of *x* (tanh(*x*)) is returned.

See Also

sinh(), cosh()

5

Other Structures and Functions

CdlFILE

ISO-9660 file descriptor

Structure

```
typedef struct {
                    CdlLOC pos;
                    u_long size;
                    char name[16];
} CdlFILE;
```

Members

pos	File position
size	File size
name	Filename

Explanation

CdIFILE holds information that describes a file on the CD-ROM in the ISO-9660 file system.

CdlLOC

CD-ROM location

Structure

```
typedef struct {
                    u_char minute;
                    u_char second;
                    u_char sector;
                    u_char track;
} CdlLOC;
```

Members

minute	Minute
second	Second
sector	Sector
track	Track number

Explanation

CDILOC holds information that describes a physical location on the CD-ROM.

Notes

The *track* member is not currently supported.

CdPlay

Plays back CD-DA tracks

Structure

```
int CdPlay (
int mode,
int *tracks,
int offset
)
```

Arguments

mode	Mode (see below)
tracks	Pointer to array that specifies the track to be played. The last element must be 0.
offset	Index into the *tracks* array where playback will begin

Explanation

The sequence of track numbers starting at the array element specified by *tracks[offset]* is played one after another. When the end of the sequence is reached, playback is repeated or ended according to the setting of *mode* as shown in the table below.

Value	Description
0	Stop playback immediately.
1	The tracks specified by *tracks* are played consecutively, and playback is stopped when all the specified tracks have been played.
2	The tracks specified by *tracks* are played consecutively, and playback is returned to the start and repeated when all the specified tracks have been played.
3	The index of the *tracks* array for the track currently being played is returned. (No other action is taken.)

Return value

The array index of the track currently being played is returned (not the track number itself). -1 is returned on error or if there is no song playing.

Notes

All playback is in units of tracks. Therefore, it is not possible to start or stop or playing in the middle of a track.

CdReadExec

Loads executable files from the CD-ROM

Structure

struct EXEC *CdReadExec(
char **file*
)

Arguments

file	Executable filename

Explanation

The executable file specified by *file* is read from the CD-ROM and placed at the memory location specified in the header of the executable file.
Reading is performed in the background, so CdReadSync() should be used to determine when reading has completed.
The loaded file can be executed as a child process using Exec().

Return value

A pointer to the EXEC structure containing information about the executable files that has been loaded is returned.

Notes

The load address of the executable file should not overlap the area used by the parent process.

CdReadFile

Reads a data file from the CD-ROM

Structure

```
int CdReadFile(
char *file,
u_long *addr,
int nbyte
)
```

Arguments

file	Filename
addr	Pointer to result memory buffer
nbyte	Number of bytes to read

Explanation

nbyte bytes of the file specified by *file* are read from the CD-ROM and placed in the buffer pointed to by *addr*.

If *nbyte* is set to 0, the entire file is read. If *file* is set to NULL, the function starts reading at the last location read by the previous call.

Return value

The number of bytes read is returned if the function was successful. 0 is returned in case of an error.

Notes

The filename must be specified as an absolute path with lower case characters automatically converted to upper case.

Reading is performed in the background, so CdReadSync() should be used to determine when reading has completed.

CdReadSync

Waits for completion of CdRead

Structure

```
int CdReadSync (
int mode,
u_char *result
)
```

Arguments

mode	0: Wait for completion of read
	1: Obtain current status and return immediately
result	Pointer to buffer storing the result

Explanation

CdReadSync() waits for reading to complete that was started by CdReadFile() and CdReadExec().

Return value

The return value has the following meaning.

Return value	Meaning
Positive integer	Number of sectors left to read
0	Read completed
-1	Read error

CdSearchFile

Gets location and size of a CD-ROM file

Structure

```
CdlFILE *CdSearchFile (
CdlFILE *fp,
char *name
)
```

Arguments

fp	Pointer to CD-ROM file structure for storing the result
name	Filename

Explanation

The location of the file on the CD-ROM specified by *name* is obtained (in minutes, seconds, sectors) and the result is stored in the CD-ROM file structure pointed to by *fp*.

Return value

If the file was found on the CD-ROM, *fp*, the pointer to the CD-ROM file structure is returned. 0 is returned if the file was not found. -1 is returned if an error occurred.

Notes

The filename must be specified as an absolute path.
When the same directory is accessed repeatedly, the file location information is cached to improve performance.

close

Closes a file

Structure

int close (

int *fd*

)

Arguments

fd	File descriptor

Explanation

The file specified by the file descriptor, *fd*, is closed.

Return value

fd is returned if the file was successfully closed. -1 is returned in all other cases.

See Also

open()

delete

Deletes a file

Structure

int delete (

char ***name*

)

Arguments

name	Filename

Explanation

The file specified by *name* is deleted.

Return value

1 is returned if the file was successfully deleted. 0 is returned in all other cases.

DIRENTRY

Structure of a directory entry

Structure

```
struct DIRENTRY {
                    char name[20];
                    long attr;
                    long size;
                    struct DIRENTRY *next
                    long head;
                    char system[8];
}
```

Members

name	Filename
attr	Attribute (depends on file system)
size	File size (bytes)
next	Pointer to next file entry (for user)
head	Starting sector number
system	System reserved

Explanation

DIRENTRY stores information relating to files contained in the file system.

See Also

firstfile(), nextfile()

EnterCriticalSection and ExitCriticalSection

Inhibit and permit interrupts

Structure

void EnterCriticalSection(void)

void ExitCriticalSection(void)

Arguments

None

Explanation

EnterCriticalSection() inhibits interrupts while in a critical section.

ExitCriticalSection() enables interrupts after leaving a critical section.

Return value

None

EXEC

Structure of an executable file

Structure

```
struct EXEC {
        unsigned long pc0;
        unsigned long gp0;
        unsigned long t_addr;
        unsigned long t_size;
        unsigned long d_addr;
        unsigned long d_size;
        unsigned long s_addr;
        unsigned long s_size;
        unsigned long sp,fp,gp,base;
};
```

Members

pc0	Execution start address
gp0	gp register initial value
t_addr	Starting address of text section and initialized data section
t_size	Size of text section and initialized data section
d_addr	System reserved
d_size	System reserved
b_addr	Starting address of uninitialized data section
b_size	Size of uninitialized data section

s_addr	Stack start address (user-specified)
s_size	Stack size (user-specified)
sp,fp,gp,base	Register storage area

Explanation

The EXEC structure is located in the upper 2K bytes of an executable file (a file in PS-X EXE format). EXEC maintains information used to load and execute a program.
When a program is executed, stack information is added to the members of the EXEC structure and the structure is passed as an argument to the Exec() function.

See Also

Exec()

Exec

Execute an executable file

Structure

long Exec (
struct EXEC **exec*,
long *argc*,
char **argv*
)

Arguments

exec	Pointer to executable file descriptor
argc	Number of arguments
argv	Argument string

Explanation

An executable file that has been loaded into memory and which is described by the information contained in the EXEC structure pointed to by *exec* is executed. *argc* and *argv* are passed to the executable program.

If *exec*->s_addr is 0, neither the stack pointer nor the frame pointer is initialized.

The operation of Exec() is shown below.

(1) The uninitialized data section is cleared.

(2) sp, fp and gp are saved, then initialized (fp is set to the value of sp).

(3) The arguments of main() are set in the a0 and a1 registers.

(4) A jump is made to the execution start address.

(5) sp, fp and gp are restored after returning.

Return value

1 is returned if the function was successful. 0 is returned in all other cases.

Notes

Exec() must be executed in a critical section.

See Also

EXEC structure, Load()

firstfile

Retrieves information about a file

Structure

```
struct DIRENTRY *firstfile (
char *name,
struct DIRENTRY *dir
)
```

Arguments

name	Filename
dir	Pointer to DIRENTRY structure that stores information relating to the referenced file

Explanation

Information about the file corresponding to the specified filename is obtained and placed in the DIRENTRY structure pointed to by *dir*.

Return value

dir is returned if the function is successful. 0 is returned in all other cases.

Notes

The wildcard character '*' can be used in the filename. Characters after the '*' are ignored.

See Also

DIRENTRY structure, nextfile()

FlushCache

Flush the I-cache

Structure

void FlushCache (void)

Arguments

None

Explanation

The Instruction cache is flushed.

FlushCache() is usually called when a program modifies instruction code.

Return value

None

Notes

The contents of the instruction cache cannot be modified.

format

Initializes the file system

Structure

```
int format (
char *fs
)
```

Arguments

fs	File system name

Explanation

The file system specified by *fs* is initialized.

Return value

1 is returned if the initialization was successful. 0 is returned in all other cases.

Notes

format() is valid only for writeable file systems.

GetPadBuf

Gets controller buffers

Structure

```
void GetPadBuf (
volatile unsigned char **buf1,
volatile unsigned char **buf2
)
```

Arguments

buf1	Pointer to a storage location that will contain a pointer to the port1 controller.
buf2	Pointer to a storage location that will contain a pointer to the port2 controller.

Explanation

The storage locations of the system controller buffers are obtained and stored at the memory locations pointed to by *buf1* and *buf2*.
Communication with the controllers occurs every vertical synchronization interruption and information read from the controllers is stored in the system controller buffers. GetPadBuf() can be used to obtain pointers to these buffers.
There is a separate buffer for each controller. The table below shows the arrangement of data within each buffer.

Bytes	Content
0	0xff: No controller 0x00: Controller present
1	Upper 4 bits: Terminal type Lower 4 bits: Received data size (in 1/2 bytes)
2~	Received data (maximum 32 bytes)

The received data format varies according to the type of controller indicated by 'terminal type'. The terminal types supported by the library are shown in the following table.

Terminal Type	Device Name
0x1	Mouse
0x2	NeGCon
0x4	Standard controller
0x5	Joystick

Please refer to the "User's Guide" for a description of the contents of received data corresponding to each terminal type.

Return value

None

GetRCnt

Gets the value of a root counter

Structure

long GetRCnt (
unsigned long *spec*
)

Arguments

spec	The specified root counter

Explanation

The current value of the root counter specified by *spec* is returned.

Return value

If the function is successful, the 32-bit unsigned and expanded root counter value is returned. -1 is returned if the function failed to complete.

See Also

StartRCnt(), ResetRCnt()

GetVideoMode()

Obtains the currently active video signal mode

Structure

long GetVideoMode (void)

Arguments

None

Explanation

The currently active video signal mode is returned.

Return value

The currently active video signal mode is returned and can assume one of the following values.

MODE_NTSC:	NTSC video signal mode
MODE_PAL:	PAL video signal mode

Notes

The default mode before SetVideoMode () is called is MODE_NTSC regardless of machine type.

See Also

SetVideoMode()

InitHeap

Initialize the heap

Structure

void InitHeap (
void **head*,
long *size*
)

Arguments

head	Pointer to heap start address
size	Heap size (multiple of 4 bytes)

Explanation

The heap pointed to by *head* is initialized. Memory management functions are also initialized, so that malloc() and the other memory allocation functions can be used. The effective heap size is actually less than *size* because of overhead.

Return value

None

Notes

Multiple calls to InitHeap() are permitted only if they do not overlap, otherwise memory management functions will not operate properly.

See Also

malloc()

Load

Loads an executable file

Structure

```
long Load (
char *name,
struct EXEC *exec
)
```

Arguments

name	Filename
exec	Pointer to an executable file structure for storing the result

Explanation

The executable file specified by *name* is loaded into memory at the address specified in its header. Internal information about the file is written to the EXEC structure pointed to by *exec*.

Return value

1 is returned if the function was successful. 0 is returned in all other cases.

See Also

EXEC structure, Exec()

LoadTest

Test load of an executable file

Structure

```
long LoadTest (
char *name,
struct EXEC *exec
)
```

Arguments

name	Filename
exec	Pointer to an executable file structure for storing the result

Explanation

The information contained in the executable file specified by *name* is written to the structure pointed to by *exec*.

Return value

The execution start address is returned if the function is successful. 0 is returned in all other cases.

See Also

EXEC structure, Load()

lseek

Moves the file pointer

Structure

int lseek (

int *fd*,

unsigned int *offset*,

int *flag*

)

Arguments

fd	File descriptor
offset	Offset
flag	See below

Explanation

The file pointer of the device corresponding to the file descriptor specified by *fd* is moved according to the values of *offset* and *flag*.

offset specifies the number of bytes to move the pointer. *flag* has the meaning indicated below.

Flag	Macro function
SEEK_SET	Seek from the beginning of file. *offset* is relative to the beginning of the file.
SEEK_CUR	Seek from the current location. *offset* is relative from the current position of the file pointer.

lseek() cannot be used with character-type drivers.

Return value

The current file pointer is returned if the function succeeded. -1 is returned in all other cases.

See Also

open(), read(), write()

nextfile

Retrieves information about the next file

Structure

```
struct DIRENTRY *nextfile (
struct DIRENTRY *dir
)
```

Arguments

dir	Pointer to DIRENTRY structure that stores information relating to the referenced file

Explanation

nextfile() continues the lookup function started by firstfile(). Information on matching files is placed in the DIRENTRY structure pointed to by *dir*.

Return value

dir is returned if the function is successful. 0 is returned in all other cases.

Notes

If the shell cover of the CD-ROM drive is opened after the execution of firstfile(), nextfile() will fail and report that the file was not found.

See Also

DIRENTRY structure, firstfile()

open

Opens a file

Structure

int open (

char ***devname*,

int *flag*

)

Arguments

devname	Filename
flag	Open mode

Explanation

The file indicated by *devname* is opened and its file descriptor is returned.

flag can be set to one of the following macros.

Macro	Open mode
O_RDONLY	Read only
O_WRONLY	Write only
O_RDWR	Read and write
O_CREAT	Create file
O_NOBUF	Non-buffering mode
O_NOWAIT	Asynchronous mode

Return value

If the open succeeded, the file descriptor is returned. -1 is returned in the event of failure.

See Also

close()

read

Reads data from a file

Structure

```
int read (
int fd,
char *buf,
int n
)
```

Arguments

fd	File descriptor
buf	Pointer to read buffer
n	Number of bytes to read

Explanation

n bytes are read from the file specified by *fd* and placed in the buffer pointed to by *buf*.

Return value

The number of bytes read up to normal termination is returned. -1 is returned if an error occurred.

See Also

open()

rename

Renames a file

Structure

```
int rename (
char *src,
char *dest
)
```

Arguments

src	Source filename
dest	New filename

Explanation

The filename specified by *src* is renamed to the name specified by *dest*. The full path from the device name must be specified for both *src* and *dest*.

Return value

1 is returned if the rename was successful. 0 is returned in all other cases.

Notes

rename() is valid only for writeable file systems.

ResetRCnt

Resets a root counter

Structure

long ResetRCnt(
unsigned long *spec*
)

Arguments

spec	The specified root counter

Explanation

The root counter specified by *spec* is reset.

Return value

1 is returned if the root counter was successfully reset. 0 is returned in the event of failure.

See Also

GetRCnt(), StartRCnt()

SetVideoMode()

Sets the currently active video signal mode

Structure

long SetVideoMode (

long *mode*

)

Arguments

mode	Video signal mode		
		MODE_NTSC:	NTSC video signal mode
		MODE_PAL:	PAL video signal mode

Explanation

The currently active video signal mode is set to *mode* for use by the library functions.

Return value

The previously active video signal mode is returned.

Notes

SetVideoMode() should be called before any library functions.

See Also

GetVideoMode()

StartRCnt

Starts a root counter

Structure

long StartRCnt (
unsigned long *spec*
)

Arguments

spec	The specified root counter

Explanation

The root counter specified by *spec* is activated and enabled for interrupts.

Return value

1 is returned if the root counter was successfully activated. 0 is returned in the event of failure.

See Also

GetRCnt(), ResetRCnt()

TestCard

Test a memory card

Structure

long TestCard (
long *chan*
)

Arguments

chan	Slot number of memory card
	0: Slot 1
	1: Slot 2

Explanation

The memory card in the slot specified by *chan* is tested and the result of the test is returned.
Card initialization is performed from the memory card control screen of the PlayStation.
One to four vertical synchronization interruptions must be performed at the end of the operation (17 msec to 68 msec).

Return value

The test result is indicated in the return value as shown below.

0: Card missing
1: Card present
2: New card detected
3: Communication or card error detected
4: Uninitialized card detected

write

Writes data to a file

Structure

```
int write (
int fd,
char *buf,
int n
)
```

Arguments

fd	File descriptor
buf	Pointer to data write buffer
n	Number of bytes to write

Explanation

The first *n* bytes from the buffer pointed to by *buf* are written to the file specified by *fd*.

Return value

The number of bytes written up to normal termination is returned. -1 is returned if an error occurred.

See Also

open()

_get_errno

Obtains i/o error code

Structure

long _get_errno (void)

Arguments

None

Explanation

The cumulative error code from all file descriptors is obtained and its value is returned.

The error code is defined in sys/errno.h.

Return value

The error code is returned.

ioctl

Controls i/o devices

Syntax

long ioctl (*fd, com, arg*)
long *fd*;
long *com*;
long *arg;*

Arguments

fd	File descriptor returned by a preceding call to open()
com	Control operation to be performed (macro defined in sys\ioctl.h)
arg	Control command argument

Explanation

ioctl() executes control commands(i.e. set baud, reopen, etc.) on the PlayStation I/O devices(ie tty, bu, etc). The Control Command macros are defined in include\sys\ioctl.h. Please refer to the I/O Control Management section of the Kernel Management chapter for more information.

Return value

Returns the value “1” if it succeeds and the value “0” otherwise.

See also

open(), read(),write(), close()

6

Index

—T—

—V—

—W—

—_—