

File Formats

© 1998 Sony Computer Entertainment

Publication date: March 1998

Sony Computer Entertainment America
919 E. Hillsdale Blvd., 2nd Flr
Foster City, CA 94404

Sony Computer Entertainment Europe
Waverley House
7-12 Noel Street
London W1V 4HH, England

The *File Formats* manual is supplied pursuant to and subject to the terms of the Sony Computer Entertainment PlayStation™ License and Development Tools Agreements or the Developer Agreement.

The *File Formats* manual is intended for distribution to and use by only Sony Computer Entertainment licensed Developers and Publishers in accordance with the PlayStation™ License and Development Tools Agreements or the Developer Agreement.

Unauthorized reproduction, distribution, lending, rental or disclosure to any third party, in whole or in part, of this book is expressly prohibited by law and by the terms of the Sony Computer Entertainment PlayStation™ License and Development Tools Agreements or the Developer Agreement.

Ownership of the physical property of the book is retained by and reserved by Sony Computer Entertainment. Alteration to or deletion, in whole or in part, of the book, its presentation, or its contents is prohibited.

The information in the *File Formats* manual is subject to change without notice. The content of this book is Confidential Information of Sony Computer Entertainment.

PlayStation and PlayStation logos are trademarks of Sony Computer Entertainment. All other trademarks are property of their respective owners and/or their licensors.

Table of Contents

About this Manual

Changes Since Last Release	xi
Related Documentation	xi
Typographic Conventions	xi
Order Information	xi

Chapter 1: Streaming Audio and Video Data

STR: Streaming (Movie) Data	1-3
BS: MDEC Bitstream Data	1-8
XA: CD-ROM Voice Data	1-28

Chapter 2: 3D Graphics

RSD: 3D Model Data	2-3
TMD: Modeling Data for OS Library	2-24
PMD: High-Speed Modeling Data	2-35
TOD: Animation Data	2-40
HMD: Hierarchical 3D Model, Animation and Other Data	2-49

Chapter 3: 2D Graphics

TIM: Screen Image Data	3-3
SDF: Sprite Editor Project File	3-8
PXL: Pixel Image Data	3-11
CLT: Palette Data	3-14
ANM: Animation Information	3-16
TSQ: Animation Time Sequence	3-22
CEL: Cell Data	3-23
BGD: BG Map Data	3-27

Chapter 4: Sound

SEQ: PS Sequence Data	4-3
SEP: PS Multi-Track Sequence Data	4-3
VAG: PS Single Waveform Data	4-5
VAB: PS Sound Source Data	4-5
DA: CD-DA Data	4-7

List of Figures

Figure 1-1: Streaming data	1-3
Figure 1-2: Frame data elements	1-4
Figure 1-3: Sector header	1-4
Figure 1-4: Streaming data with header frame	1-5
Figure 1-5: MDEC animation sector header	1-6
Figure 1-6: Arrangement of data on CD-ROM for XA-ADPCM audio	1-7
Figure 1-7: Streaming data with audio	1-7
Figure 1-8: Data sector in streaming data with audio	1-7
Figure 1-9: XA-ADPCM audio sectors for streaming data with audio	1-8
Figure 1-10: Original MDEC Image Data	1-8
Figure 1-11: Macroblock Arrangement	1-11
Figure 1-12: Runlevel Code Format	1-18
Figure 1-13: Quantization Step Placement	1-18
Figure 1-14: Runlevel Pairs	1-18
Figure 1-15: Header Word Format	1-19
Figure 1-16: BS Header Format	1-21
Figure 1-17: BS Blocks	1-22
Figure 2-1: Overall structure of an RSD file	2-4
Figure 2-2: Sample RSD file	2-5
Figure 2-3: Overall structure of a PLY file	2-6
Figure 2-4: Data length record in a PLY file	2-6
Figure 2-5: VERTEX descriptor in a PLY file	2-6
Figure 2-6: NORMAL descriptor in a PLY file	2-6
Figure 2-7: POLYGON descriptor in a PLY file	2-7
Figure 2-8: POLYGON descriptor for triangular/quadrangular polygons	2-7
Figure 2-9: POLYGON descriptor for straight lines	2-7
Figure 2-10: Polygon descriptor for sprites	2-7
Figure 2-11: Sample PLY file	2-8
Figure 2-12: Overall structure of a MAT file	2-8
Figure 2-13: MATERIAL descriptor for a MAT file	2-9
Figure 2-14: No texture (colored polygons/lines)	2-10
Figure 2-15: No texture (Gouraud-colored polygons/lines)	2-10
Figure 2-16: Textured polygons/sprites	2-10
Figure 2-17: Polygons with colored textures	2-11
Figure 2-18: Polygons with gradation texture	2-11
Figure 2-19: Polygons/sprites with repeating textures	2-11
Figure 2-20: Polygons with repeating colored textures	2-12
Figure 2-21: Polygons with repeating gradation textures	2-12
Figure 2-22: Sample MAT file	2-12
Figure 2-23: Overall structure of a GRP file	2-13
Figure 2-24: GROUP descriptor for GRP file	2-13
Figure 2-25: Following Lines	2-13
Figure 2-26: Sample GRP file	2-14
Figure 2-27: Overall structure of a MSH file	2-14
Figure 2-28: Sample MSH file	2-15
Figure 2-29: Overall structure of a PVT file	2-15
Figure 2-30: Sample PVT file	2-15
Figure 2-31: Overall structure of a COD file	2-16
Figure 2-32: Sample COD files	2-17
Figure 2-33: Overall structure of an OGP file	2-18
Figure 2-34: Sample OGP file	2-19
Figure 2-35: Overall structure of a MOT file	2-19
Figure 2-36: Sample MOT file	2-22
Figure 2-37: TMD File Format	2-24

Figure 2-38: Structure of Header	2-25
Figure 2-39: OBJ TABLE structure	2-25
Figure 2-40: Drawing Packet General Structure	2-26
Figure 2-41: Mode	2-26
Figure 2-42: Flag	2-26
Figure 2-43: Vertex Structure	2-27
Figure 2-44: Normal Structure	2-27
Figure 2-45: Fixed-Point Format	2-27
Figure 2-46: TSB	2-28
Figure 2-47: CBA	2-28
Figure 2-48: Mode Value of 3 Vertex Polygon with Light Source Calculation	2-29
Figure 2-49: Packet Configuration of 3 Vertex Polygon with Light Source Calculation	2-30
Figure 2-50: Mode Value of 4 Vertex Polygon with Light Source Calculation	2-30
Figure 2-51: Packet Configuration for 4 Vertex Polygon with Light Source Calculation	2-31
Figure 2-52: Mode Value of 3 Vertex Polygon with No Light Source Calculation	2-32
Figure 2-53: Packet configuration for 3 Vertex Polygon with No Light Source Calculation	2-32
Figure 2-54: Mode Value of 4 Vertex Polygon with No Light Source Calculation	2-32
Figure 2-55: Packet Configuration for 4 Vertex Polygon with No Light Source Calculation	2-33
Figure 2-56: Mode Value of Straight Line	2-33
Figure 2-57: Packet Configuration for "Straight Line"	2-34
Figure 2-58: Mode Value of 3D Sprite	2-34
Figure 2-59: Packet Configuration for Sprites	2-34
Figure 2-60: Overall structure of PMD files	2-35
Figure 2-61: OBJECT Structure	2-36
Figure 2-62: Packet Gp structure	2-37
Figure 2-63: VERTEX structure	2-39
Figure 2-64: TOD file format	2-40
Figure 2-65: Frame	2-41
Figure 2-66: PACKET	2-42
Figure 2-67: packet type values and packet data contents	2-42
Figure 2-68: Packet Data Configuration when Attribute	2-43
Figure 2-69: Packet data when forced light-source calculation is switched ON	2-44
Figure 2-70: Flag when Coordinate (RST)	2-44
Figure 2-71: Packet Data Configuration when Coordinate (RST)	2-45
Figure 2-72: Packet Data Configuration when TMD Data ID	2-46
Figure 2-73: Packet Data Configuration when Parent Object	2-46
Figure 2-74: Packet Data Configuration when Matrix Value	2-46
Figure 2-75: Flag when Light Source Packet	2-46
Figure 2-76: Packet Data when Light Source Packet	2-47
Figure 2-77: Flag for Camera	2-47
Figure 2-78: Composition of packet data with camera (part 1)	2-48
Figure 2-79: Composition of packet data with camera (part 2)	2-48
Figure 2-80: The meanings and values of the flag when object control is set	2-48
Figure 2-81: HMD Structure	2-50
Figure 2-82: HMD Header section	2-50
Figure 2-83: One primitive block which has been primitive chained	2-51
Figure 2-84: Primitive Structure	2-51
Figure 2-85: Primitive Header	2-52
Figure 2-86: Primitive Structure	2-53
Figure 2-87: HMD Header Section	2-57
Figure 2-88: Variable transfer area transferred to the primitive driver	2-58
Figure 2-89: One Primitive	2-59
Figure 2-90: Type Field	2-60
Figure 2-91: Polygon Primitive Driver	2-61
Figure 2-92: Primitive Type of Polygon Primitive	2-62
Figure 2-93: Number and Size of Polygons	2-62

Figure 2-94: Shared Primitive Driver	2-91
Figure 2-95: Primitive Type of Shared Primitive	2-91
Figure 2-96: Shared Polygon Processing Flow	2-92
Figure 2-97: Parameter Memory Area of Image Primitive Driver	2-95
Figure 2-98: Image Primitive Type Field	2-95
Figure 2-99: Animation Structure	2-97
Figure 2-100: Diagram Showing Correlation of All Animation Sections	2-98
Figure 2-101: Animation Primitive Type Field	2-99
Figure 2-102: Type Field when TGT=0	2-100
Figure 2-103: Type Field when TGT=1	2-101
Figure 2-104: LENGTH=16 bit, WRITE=0x1	2-101
Figure 2-105: LENGTH=16bit, WRITE=0x7	2-101
Figure 2-106: LENGTH=8bit, WRITE=0x1	2-101
Figure 2-107: LENGTH=8bit, WRITE=0x7	2-101
Figure 2-108: Sequence Header	2-102
Figure 2-109: Sequence Pointer	2-102
Figure 2-110: Setting Update Location	2-103
Figure 2-111: Sequence Count and Size	2-103
Figure 2-112: Fixed Point Format Used in SPEED Specification	2-104
Figure 2-113: Sequence Management Data	2-105
Figure 2-114: Sequence Descriptor (Normal)	2-105
Figure 2-115: Sequence Descriptor (Jump)	2-106
Figure 2-116: Sequence Descriptor (Control)	2-106
Figure 2-117: Format of Parameters in the Argument Area	2-108
Figure 2-118: Bezier Interpolation	2-110
Figure 2-119: BSPLINE Work Area	2-110
Figure 2-120: BSPLINE Interpolation	2-111
Figure 2-121: Primitive Type Field	2-120
Figure 2-122: MIMe Primitive DRIVER	2-120
Figure 2-123: Primitive Type of MIMe Primitive	2-120
Figure 2-124: dflags Example	2-122
Figure 2-125: Ground Primitive Type Field	2-126
Figure 2-126: Sequence Management Construction	2-136
Figure 2-127: Sequence With No Jumps	2-138
Figure 2-128: Sequence With Jumps	2-138
Figure 2-129: Loop Sequence	2-138
Figure 2-130: Immediate Sequence Switching 1	2-139
Figure 2-131: Immediate Sequence Switching 2	2-139
Figure 3-1: TIM File Format	3-3
Figure 3-2: The order of bytes in a file	3-3
Figure 3-3: Structure of TIM File Header	3-4
Figure 3-4: Flag Word	3-4
Figure 3-5: CLUT	3-5
Figure 3-6: A CLUT entry	3-5
Figure 3-7: Pixel data	3-6
Figure 3-8: Frame buffer data (pixel data)	3-6
Figure 3-9: SDF File Structure	3-8
Figure 3-10: PXL File Structure	3-11
Figure 3-11: Byte Order in File	3-11
Figure 3-12: Structure of PXL File Header	3-11
Figure 3-13: FLAG Bit Configuration	3-12
Figure 3-14: Configuration of Pixel Data Section	3-12
Figure 3-15: VRAM Data (Pixel Data)	3-13
Figure 3-16: CLT File Structure	3-14
Figure 3-17: Byte Order in File	3-14
Figure 3-18: Structure of CLT File Header	3-14
Figure 3-19: FLAG Bit Configuration	3-15

Figure 3-20: Structure of CLUT Section	3-15
Figure 3-21: CLUT Entry	3-16
Figure 3-22: ANM file format	3-16
Figure 3-23: File Header	3-16
Figure 3-24: FLAG	3-17
Figure 3-25: Sequence	3-17
Figure 3-26: SPRITEGp	3-18
Figure 3-27: FLAG	3-19
Figure 3-28: CBA	3-19
Figure 3-29: FLAG2	3-20
Figure 3-30: CLUTGp	3-21
Figure 3-31: SEQ Data Structure	3-22
Figure 3-32: HEADER	3-22
Figure 3-33: SEQUENCE	3-22
Figure 3-34: CEL file format	3-23
Figure 3-35: HEADER	3-23
Figure 3-36: FLAG	3-23
Figure 3-37: CELL Data Section	3-24
Figure 3-38: CBA	3-24
Figure 3-39: TSB	3-25
Figure 3-40: FLAG	3-26
Figure 3-41: ATTR Format (8 Bit)	3-26
Figure 3-42: ATTR Format (16 Bit)	3-26
Figure 3-43: BG file format	3-27
Figure 3-44: HEADER	3-27
Figure 3-45: FLAG	3-27
Figure 3-46: Cell Arrangement in MAP (when 8 x 8)	3-28
Figure 3-47: MAP	3-28
Figure 3-48: ATTR (8 bit)	3-28
Figure 3-49: ATTR (16 bit)	3-28
Figure 4-1: SEQ Format	4-3
Figure 4-2: SEP Format	4-4
Figure 4-3: VAG Format	4-5
Figure 4-4: VAB Format	4-6
Figure 4-5: DA Format	4-7

List of Tables

Table 1-1: XA-ADPCM audio data types supported by PlayStation	1-6
Table 1-2: Data/gap ratios	1-7
Table 1-3: BS Format Versions	1-23
Table 1-4: DC Codebook	1-24
Table 1-5: Decompression Speed	1-24
Table 1-6: Transfer Speed	1-24
Table 2-1: Polygon Numbers	2-9
Table 2-2: TYPE bit layout	2-37
Table 2-3: Packet data bit-by-bit breakdown	2-43
Table 2-4 : Primitive Pointer Table	2-57
Table 3-1: STP Bit Function in Combination with R, G, B Data	3-6
Table 3-2: Display	3-10
Table 3-3: Color	3-10
Table 3-4: Role of STP Bit	3-16
Table 4-1: VAG Offset Table	4-7

About this Manual

This manual is a release of the PlayStation file format specifications as of Run-time Library release 4.2. The purpose of this manual is to provide a detailed reference to PlayStation file formats. Other documents in the Developer Reference Series provide overview level information regarding these formats (see Related Documentation below).

Changes Since Last Release

Substantial changes were made to the HMD section.

Minor changes were made to the ANM and CEL sections.

Related Documentation

The following volumes in the Developer Reference Series also contain file format information:

PlayStation Operating System
Run-time Library Overview
3D Graphics Tools
Sprite Editor
Sound Artist Tool

Typographic Conventions

Certain Typographic Conventions are used through out this manual to clarify the meaning of the text. The following details the specific conventions used to represent literals, arguments, keywords, etc.

The following conventions apply to all narrative text outside of the structure and function descriptions.

<i>Convention</i>	<i>Meaning</i>
	A revision bar. Indicates that information to the left or right of the bar has been changed or added since the last release.
<code>courier</code>	Indicates literal program code.
Bold	Indicates a document, chapter or section title.

The following conventions apply within structure and function descriptions only:

<i>Convention</i>	<i>Meaning</i>
Medium Bold	Denotes structure or function types and names.
<i>Italic</i>	Denotes function arguments and structure members.
{ }	Denotes the start and end of the member list in a structure declaration.

Order Information

Additional copies of this documentation can be ordered by contacting:

Sony Computer Entertainment America
919 East Hillsdale Blvd., 2nd floor
Foster City, CA 94404
Tel (650) 655-8000

Chapter 1:

Streaming Audio and Video Data

STR: Streaming (Movie) Data

"Streaming" refers to a processing format for successive reading and processing of data from a CD-ROM. STR format is a CD-ROM data format defined to enable streaming for the PlayStation.

Although streaming is generally used to successively read and play back animation or audio data, it is not limited to such applications. Streaming can also be used for various other kinds of time-series data processing that involves continuous changes.

Continuously reading data from a CD-ROM is called "streaming". The streaming library is used separately from other CD-ROM functions. The bitstream used for animation and movie playback is obtained via this streaming mechanism. Image size and other such supplemental information are not included in the bitstream. For this reason, the supplemental data format (STR format) separately defines information required for animation playback in the header.

Streaming data

As shown in Figure 1-1, streaming data in an STR file is represented as a continuous array of frame data elements. Frame data is used to represent a location for streaming. The frame contents differ depending on what kind of data is used for streaming. For example, in animation, each element of frame data may contain one of the still images that makes up the animation sequence. If the application is animation of 3D modeling data, each element of frame data will contain one unit of the modeling data that makes up the animation sequence.

The type of data within each frame is not specified by the STR format, so it can be freely defined in various ways as needed. The data length and type is specified independently for each frame, so it is possible to mix various types and sizes of frame data within an STR file as needed. Continuously reading data from a CD-ROM is called "streaming". The streaming library is used separately from other CD-ROM functions.

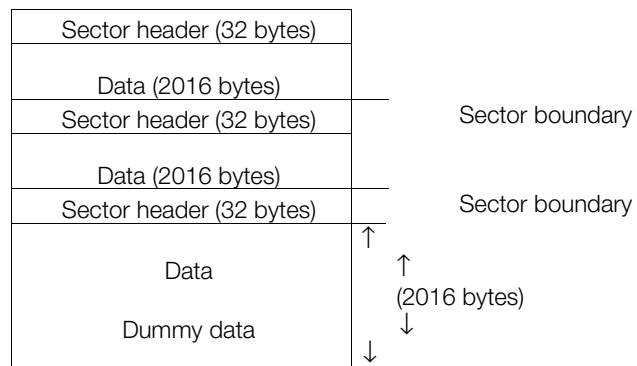
Figure 1-1: Streaming data

frame data
frame data
frame data
frame data

Frame data

Frame data in an STR file is divided into 2048-byte sectors, corresponding to CD-ROM data sectors. Each sector starts with a 32-byte header with information about the sector data, followed by 2016 bytes of data. Each frame must begin on a sector boundary. When necessary, the last sector of data for a frame should be padded with filler to reach the 2048-byte boundary, as shown in Figure 1-2.

Figure 1-2: Frame data elements



Sector headers

A sector header, located at the start of each sector, is divided into fields as shown in Figure 1-3.

StSTATUS

The StSTATUS is the STR format identifier and version information.

StTYPE

StTYPE indicates the frame data's data type: if the MSB is a 1, the data type is a system-defined format; if the MSB is 0 it is a user-defined format.

After setting the MSB to 0, the user can set the other 15 bits as desired. This enables other frame formats to be incorporated into the STR format.

StSECTOR_OFFSET

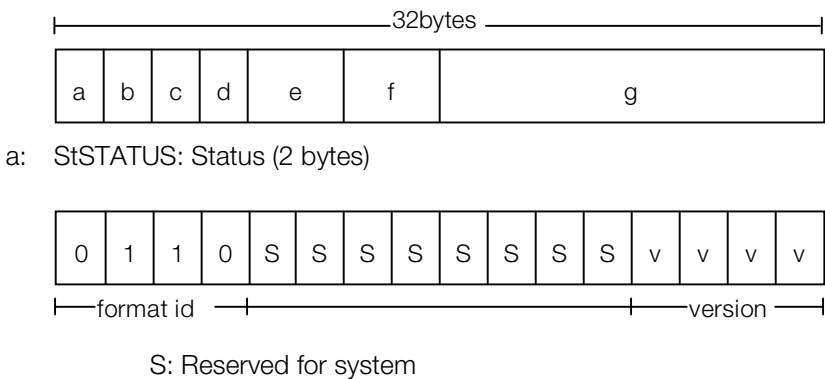
StSECTOR_OFFSET is the sector number in the frame, StSECTOR_SIZE is the number of sectors in the frame, and StFRAME_NO is the frame number of the streaming data.

These values are used to ensure that the streaming library reads the frame data consecutively without missing any sectors.

StUSER

StUSER is the user-defined field, which can be used as needed for various data types.

Figure 1-3: Sector header



b: StTYPE: data type (2 bytes)



s = 1 indicates system-defined format
0 indicates user-defined format

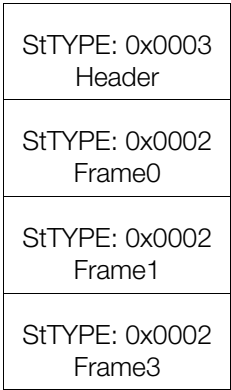
- c: StSECTOR_OFFSET: sector number in frame (2 bytes)
- d: StSECTOR_SIZE: number of sectors in the frame (2 bytes)
- e: StFRAME_NO: Frame number of streaming data (4 bytes, starting from 1)
- f: StFRAME_SIZE: Frame size (in long words, 4 bytes)
- g: StUSER: User-defined area (16 bytes)

Note: All are little endian

User-defined frame data

The user can set the MSB of the sector header's StTYPE field to 0 to enable streaming of user-defined data. If the entire run of streaming data uses the same attribute values, it is possible to use just one header (header frame) for the entire run instead of for each element of frame data, thereby reducing the total amount of data. In streaming data that has a header frame, the StTYPE field in the header frame may be changed to a Data field (see Figure 1-4).

Figure 1-4: Streaming data with header frame



In the example shown in the figure, the LSB in the StTYPE field is used to distinguish frame data in a header frame from other non-header frame data.

When reading data from a CD-ROM, the StTYPE field in the frame data read by the application can be freely accessed. This allows the interpretation of the data in the frame to change according to the type of frame.

Examples of the types of data saved in the header frame are: CLUT data that is not saved in each frame (for animation), a table of jumps to certain frames, and so on.

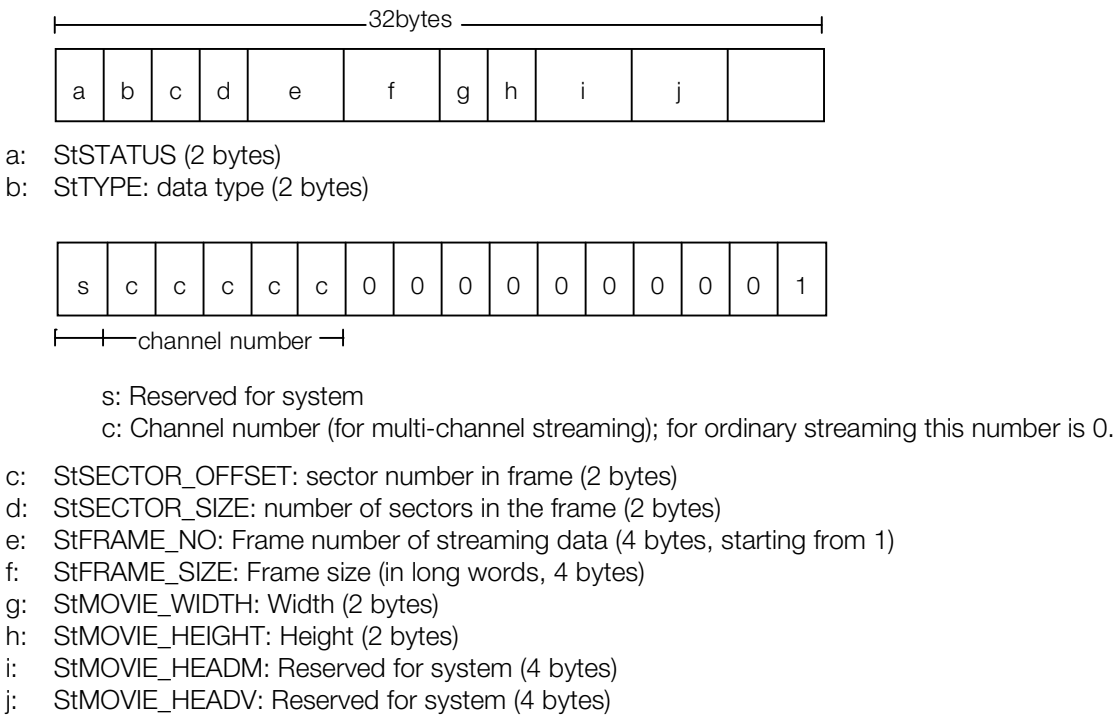
System-defined frame data

The following is currently provided as system-defined frame data.

MDEC animation (Type: 0x8001)

Figure 1-5 shows an MDEC animation sector header.

Figure 1-5: MDEC animation sector header



Streaming data with audio

Successive audio playback

The PlayStation plays back ADPCM (Adaptive Differential Pulse Code Modulation) audio data as specified in the CD-ROM XA (eXtended Audio) standard (this is hereafter abbreviated as XA-ADPCM audio).

The PlayStation supports the following four types of XA-ADPCM audio:

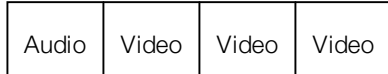
Table 1-1: XA-ADPCM audio data types supported by PlayStation

Sampling frequency	Stereo/monaural
37.8 Khz	Stereo
37.8 Khz	Monaural
18.9 Khz	Stereo
18.9 Khz	Monaural

XA-ADPCM audio is sent directly from the CD-ROM decoder to the SPU without using main memory.

The streaming format described above is used to read CD-ROM data into main memory. Consequently, it cannot be used for playing back XA-ADPCM audio.

When playing back XA-ADPCM audio, the data sectors on the CD-ROM must be arranged (interleaved) as shown in Figure 1-6. Figure 1-7 shows an example of this structure for streaming data with audio.

Figure 1-6: Arrangement of data on CD-ROM for XA-ADPCM audio

The ratio of data sector size to gap size depends on the type of XA-ADPCM audio data and the CD-ROM's playback speed, as shown below.

Table 1-2: Data/gap ratios

CD-ROM playback speed	Type	Data/gap ratio
Double speed	37.8 kHz, stereo	1 sector/7 sectors
Double speed	37.8 kHz, monaural	1 sector/15 sectors
Double speed	18.9 kHz, stereo	1 sector/15 sectors
Double speed	18.9 kHz, monaural	1 sector/31 sectors
Standard speed	37.8 kHz, stereo	1 sector/3 sectors
Standard speed	37.8 kHz, monaural	1 sector/7 sectors
Standard speed	18.9 kHz, stereo	1 sector/7 sectors
Standard speed	18.9 kHz, monaural	1 sector/15 sectors

Streaming data with audio

Streaming data with audio means ordinary (non-audio) streaming data is interleaved with XA-ADPCM audio.

This kind of interleaved XA-ADPCM audio data must use the structure shown in Figure 1-6 above.

Accordingly, the structure of streaming data with audio has streaming data inserted in the "gap" sections shown in Figure 1-6. Figure 1-7 shows an example of this structure for streaming data with audio.

Figure 1-7: Streaming data with audio

Frame 1 Frame 2 Frame 3

Streaming data: 30 fps (5 sectors/frame)

XA-ADPCM audio: 37.8 KHz, stereo

CD-ROM playback speed: Double speed

Note that the size of a sector of streaming data with audio is different from that of ordinary streaming data.

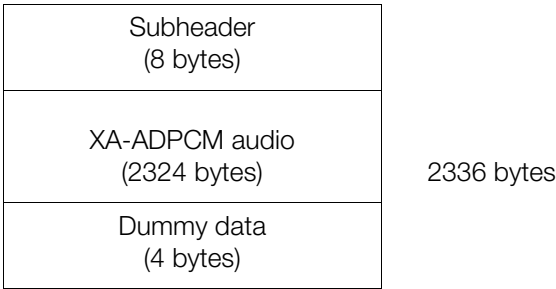
Figure 1-8: Data sector in streaming data with audio

Subheader (8 bytes)	2336 bytes
Sector header (32 bytes)	
Data (2016 bytes)	
Dummy data (280 bytes)	

The sector size is 2048 bytes for ordinary (non-audio) streaming data, and 2336 bytes for streaming data with audio.

Figure 1-8 and Figure 1-9 show sectors used for streaming data with audio.

Figure 1-9: XA-ADPCM audio sectors for streaming data with audio



Note that the sector header and data areas in Figure 1-8 are exactly identical to their counterparts in Figure 1-2.

The subheader size is specified by the CD-ROM XA standard. This subheader contains information such as flags for distinguishing data sectors from audio sectors.

A dummy data section is added to the end of the sector to make the sector size for XA-ADPCM audio sectors the same as for non-XADPCM audio sectors.

BS: MDEC Bitstream Data

This section discusses the image data format used in MDEC and libpress, as well as how to create data in this format.

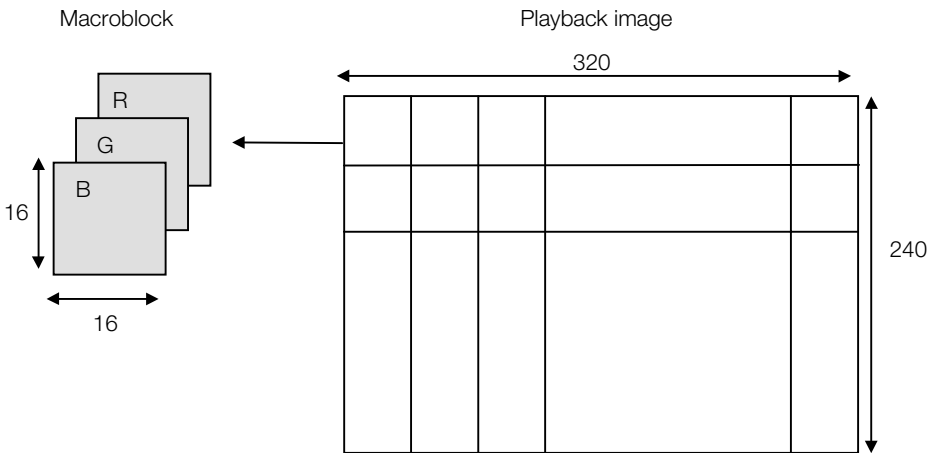
Original Image Data

Original MDEC image data consists of a series of 24-bit RGB images arranged over time, where each image is a multiple of 16 pixels wide by a multiple of 16 pixels tall. Each of these images is called a frame. Because correlations between frames are not used in MDEC, each frame can be processed independently.

MDEC treats each frame as a collection of small regions of 16x16 pixels. These small regions are called *macroblocks*. MDEC takes the decoded results and rewrites these to main memory as macroblocks. The reconstruction of these into a single frame is performed by the CPU and the GPU.

For example, data for a 320x240 image is split up into a number of 16x16 macroblocks, and each macroblock is then compressed.

Figure 1-10: Original MDEC Image Data



The vertical and horizontal frame sizes should both be multiples of 16. If this is not the case, some additional processing is necessary.

Splitting into Macroblocks

The pixels in a frame are generally ordered from top left to bottom right. When encoding, the pixels are re-ordered so that they can be unified in a macroblock.

```
#define WIDTH      320
#define HEIGHT     160

typedef struct
{
    u_char r, g, b, pad;
} PIXEL

make_macro_block(frame, macroblock)
PIXEL frame[HEIGHT][WIDTH];
PIXEL macroblk[][16][16];
{
    int ox, oy, x, y, i = 0;

    for (ox = 0; ox < WIDTH; ox += 16)
    {
        for (oy = 0; oy < HEIGHT; oy += 16, i++)
        {
            for (y = 0; y < 16; y++)
            {
                for (x = 0; x < 16; x++)
                {
                    macroblock[i][y][x] =
                        frame[oy+y][ox+x];
                }
            }
        }
    }
}
```

In this case, the macroblock is ordered vertically from the top left. This makes it possible to reduce the number of times the frame buffer transfer command (LoadImage) is executed.

If the macroblocks are to be ordered from left to right:

```
for (y = 0; y < HEIGHT; y += 16)
{
    for (x = 0; x < WIDTH; x += 16)
    {
        setRECT(&rect, x, y, 16, 16);
        LoadImage(&rect, p);
        p += 16 * 16;
    }
}
```

If the macroblocks are to be ordered from top to bottom:

```
for (y = 0; y < HEIGHT; y += 16)
{
    setRECT(&rect, x, 0, 16, HEIGHT);
    LoadImage(&rect, p);
    p += 16 * HEIGHT;
}
```

RGB-YCbCr Conversion

MDEC performs its internal processing using the YCbCr color system. Macroblocks in the RGB color system (RGB macroblocks) must be converted to the YCbCr color system. This process is called CSC (Color Space Conversion). The luminance values of the pixels can be expressed as a point in the three-dimensional space formed by the R,G,B components. CSC can be understood as a coordinate transformation for this coordinate system (color space).

In MDEC, the conversion formula below is used to convert the luminance of a pixel to the RGB color system.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.0 & 0 & 1.402 \\ 1.0 & -0.3437 & -0.7143 \\ 1.0 & 1.772 & 0 \end{bmatrix} \times \begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix}$$

The inverse of this matrix is generally used to convert from the RGB system to the YCbCr system. The MOVCONV encoder uses this matrix.

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.16871 & -0.33130 & -0.5 \\ 0.5 & -0.4187 & -0.081 \end{bmatrix} \times \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Physically, the Y signal is the luminance signal, and Cb, Cr are the color-difference signals. A black-and-white TV set uses only the Y signal. For a black-and-white screen, the Cb and Cr components are all 0.

The luminance values, recorded in the frame buffers according to the RGB system, are converted in the PlayStation to the YCbCr system and are output from the video terminal (the S terminal). The receiver (TV monitor) receives this video signal and converts it back into RGB. Voltages corresponding to the separate RGB components are sent to the electron beam, which lights up the phosphors arranged on the picture tube. This creates the final image seen by the user.

Because numerous color space conversions are performed, the color shade of the final output can sometimes differ from the expected color. These variations can be due to shifts caused by gamma correction coefficients, or they can be due to shifts caused by the conversion matrices.

Electron beam voltages and luminance values are generally not directly proportional, so some correction is made to luminance ahead of time. This is known as gamma correction. The luminance B of the phosphors on the monitor and the input voltage E have the exponential relationship shown below.

$$B = \text{pow}(aE, \text{gamma});$$

Thus, the signal source raises the RGB signals to the power of $(1/\gamma)$ beforehand. The value of γ varies slightly from monitor to monitor.

For images composed with computer graphics (CG), gamma correction may be set for a high-definition computer monitor, or the file on a hard disk may have no correction at all. If gamma correction is improper or not present, proper correction must be performed at this encoding stage. If the luminance of the final CG image is darker (or brighter) than expected, improper gamma correction could be the problem.

The characteristics of the phosphors used in a monitor can also vary from monitor to monitor. Characteristics may also vary from country to country based on the users' tastes.

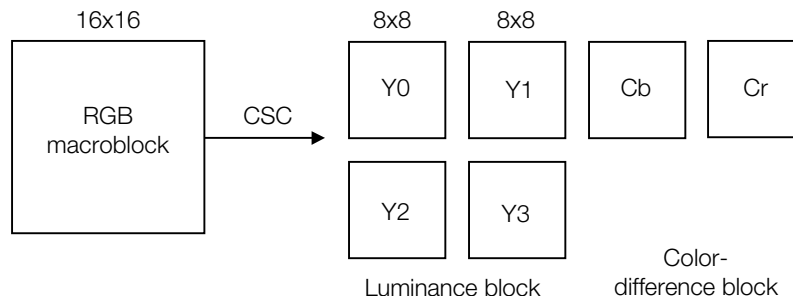
These factors in a video monitor's handling of video signals can be evaluated to some extent by comparing the image from the PlayStation's RGB analog output to the image from the video output.

Differences in color shade can be due to the image source, but if this difference is noticeable enough, some sort of counter-correction needs to be applied at the encoding stage. Differences in the displayed color system and the color system used during encoding can result in mishandling of errors (noise) during the encoding. This could be the problem if the image quality changes only for image sources having a specific color shade. A minor discrepancy in the color space used in encoding could come out as a significant color difference on the display.

Creation of Macroblocks

Since the luminance signal (Y) generally does not require as high a resolution as the color-difference signals, its data is reduced by $1/4$ ($1/2$ in the x direction and $1/2$ in the y direction) relative to the Cb and Cr elements. At the same time, the Y element is split into four 8×8 blocks. Thus, a 16×16 macroblock is split into two color-difference blocks (Cb,Cr) and four luminance blocks (Y0,Y1,Y2,Y3). This collection of small 8×8 units is called a block. A macroblock is arranged in the following order, beginning with a color-difference block.

Figure 1-11: Macroblock Arrangement



The following methods are possible ways to reduce color-difference blocks by $1/4$:

- Make the block using only even-numbered x,y points from the original 16×16 blocks.
- Use the average of four adjacent points from the original block as one point in the block.

In a) elements at or over $1/4$ of the sampling frequency (f_s) are carried over as noise (aliasing noise).

In b) the processing used to determine the average acts as a sort of two-dimensional lowpass filter (LPF), and so provides better results than a). This method is used in MOVCONV. This method is still unable to completely eliminate elements over $1/4 f_s$, however, so some aliasing remains. To improve on this a filter having more dimensions and better cut-off properties would be needed, but this would require greater encoding time. Use a filter having a number of dimensions suited to the application.

Block Offset

A color space expressed in 8-bit (0-255) RGB values would have the following range in the YCbCr color space:

Y 0 ~ 255
Cb -128 ~ +127
Cr -128 ~ +127

In order to unify the ranges for the luminance blocks and the color-difference blocks, -128 is added to all the luminance values in the luminance block. This allows all internal processing to be performed using unsigned chars. During decoding in MDEC, a block is checked to see if it is a luminance block or a color-difference block. If it is a luminance block, a value of +128 is added, which returns the value to its original mode.

DCT

DCT (Discrete Cosine Transformation) is applied to the blocks making up the macroblock. DCT is generally a type of similarity transformation called an orthogonal transformation. Taking an 8x8 matrix X, where the elements are the luminance values in a block, the transformation defined by

$$Y = P \cdot X \cdot P_i$$

is called a similarity transformation (P is a matrix having an inverse matrix, and P_i is the inverse of P). When P_t is the transpose of matrix P ($P_t(x,y)=P(y,x)$), and

$$P_i = P_t$$

then this matrix is called an orthogonal matrix, and this transformation is called an orthogonal transformation. Using orthogonal matrix P, the orthogonal transformation can be written as

$$Y = P \cdot X \cdot P_t$$

DCT is this orthogonal transformation, where P has the values shown below.

$$P = \begin{bmatrix} 4096 & 4096 & 4096 & 4096 & 4096 & 4096 & 4096 & 4096 \\ 5681 & 4816 & 3218 & 1130 & -1130 & -3218 & -4816 & -5681 \\ 5532 & 2217 & -2217 & -5532 & -5532 & -2217 & 2217 & 5352 \\ 4816 & -1120 & -5681 & -3218 & 3218 & 5681 & 1130 & -4816 \\ 4096 & -4096 & -4096 & 4096 & 4096 & -4096 & -4096 & 4096 \\ 3218 & -5681 & 1120 & 4816 & -4816 & -1130 & 5681 & -3218 \\ 2217 & -5352 & 5352 & -2217 & -2217 & 5352 & -5352 & 2217 \\ 1130 & -3218 & 4816 & -5681 & 5681 & -4816 & 3218 & -1130 \end{bmatrix} \times 1/64$$

In this case, P_t is as follows, so that $P \cdot P_t = E$ (E: unit matrix).

$$P_t = \begin{bmatrix} 4096 & 5681 & 5352 & 4816 & 4096 & 3218 & 2217 & 1130 \\ 4096 & 4816 & 2217 & -1130 & -4096 & -5681 & -5352 & -3218 \\ 4096 & 3218 & -2217 & -5681 & -4096 & 1130 & 5352 & 4816 \\ 4096 & 1130 & -5352 & -3218 & 4096 & 4816 & -2217 & -5681 \\ 4096 & -1130 & -5352 & 3218 & 4096 & -4816 & -2217 & 5681 \\ 4096 & -3218 & -2217 & 5681 & -4096 & -1130 & 5352 & -4816 \\ 4096 & -4816 & 2217 & 1130 & -4096 & 5681 & -5352 & 3218 \\ 4096 & -5681 & 5352 & -4816 & 4096 & -3218 & 2217 & -1130 \end{bmatrix} \times 1/64$$

Based on $P \cdot P_t = E$, the inverse transformation of DCT can be expressed as

$$X = P_t \cdot Y \cdot P$$

Thus, it can be seen that IDCT is simply DCT with matrix P replaced by the transpose matrix of P.

Physical Significance of DCT

Physically, DCT signifies a frequency transformation. The upper left element (element (0,0)) of the 8x8 matrix obtained from a DCT transformation expresses the DC (direct current) element of the original image block X, and is equivalent to the average of all the elements in image block X. The other elements express the AC (alternating current) elements, and the frequency elements increase to the right and down in the matrix.

In natural images, the frequency elements are generally concentrated in the lower regions. Thus, performing a DCT transformation results in smaller values toward the bottom right. Compression using DCT takes advantage of this tendency in the elements.

Quantization

After DCT transformation, each element in a block is quantized according to different resolutions. A quantization table (Q table) is used to indicate the quantization widths (steps) for each element.

MDEC uses the quantization table shown below. The same table is currently used for both the luminance blocks and the color-difference blocks.

Luminance block:

$$Q_{\text{tab}} = \begin{bmatrix} 2 & 16 & 19 & 22 & 26 & 27 & 29 & 34 \\ 16 & 16 & 22 & 24 & 27 & 29 & 34 & 37 \\ 19 & 22 & 26 & 27 & 29 & 34 & 34 & 38 \\ 22 & 22 & 26 & 27 & 29 & 34 & 37 & 40 \\ 22 & 26 & 27 & 29 & 32 & 35 & 40 & 48 \\ 26 & 27 & 29 & 32 & 35 & 40 & 48 & 58 \\ 26 & 27 & 29 & 34 & 38 & 46 & 56 & 69 \\ 27 & 29 & 35 & 38 & 46 & 56 & 69 & 83 \end{bmatrix} \times 1/16$$

Color-difference block:

$$Q_{\text{tab}} = \begin{bmatrix} 2 & 16 & 19 & 22 & 26 & 27 & 29 & 34 \\ 16 & 16 & 22 & 24 & 27 & 29 & 34 & 37 \\ 19 & 22 & 26 & 27 & 29 & 34 & 34 & 38 \\ 22 & 22 & 26 & 27 & 29 & 34 & 37 & 40 \\ 22 & 26 & 27 & 29 & 32 & 35 & 40 & 48 \\ 26 & 27 & 29 & 32 & 35 & 40 & 48 & 58 \\ 26 & 27 & 29 & 34 & 38 & 46 & 56 & 69 \\ 27 & 29 & 35 & 38 & 46 & 56 & 69 & 83 \end{bmatrix} \times 1/16$$

The actual quantization is performed for each element by dividing it by the product of the corresponding Q table value and QUANT, which determines the quantization step for the entire block. DC elements are not affected by QUANT.

```
y[0] = x[0] • 16/(qtab[0] • 8);
```

```
for (i = 1; i < 64; i++)
```

```
    y[i] = x[i]/(QUANT • Qtable[i]);
```

Q table values increase toward the bottom right of the matrix. This is because the higher frequency elements of the image do not need as much accuracy as the lower frequency elements.

Making the QUANT (the overall quantization step) value large increases the amount of lost data, thus decreasing image quality after decoding. However, since the number of 0 elements in the block is increased, the size of the data used in the run-level transformation is decreased.

Zig-zag Transformation

The quantized block is numbered one-dimensionally in a type of ordering called zig-zag ordering. Quantization and zig-zag transformation are performed together in the following manner.

```
static int zscan[] = {
    0 ,1 ,8 ,16,9 ,2 ,3 ,10,
    17,24,32,25,18,11,4 ,5 ,
    12,19,26,33,40,48,41,34,
    27,20,13,6 ,7 ,14,21,28,
    35,42,49,56,57,50,43,36,
    29,22,15,23,30,37,44,51,
    58,59,52,45,38,31,39,46,
    53,60,61,54,47,55,62,63,
};
static block_t iqtabs[] = {
    2,16,19,22,26,27,29,34,
    16,16,22,24,27,29,34,37,
    19,22,26,27,29,34,34,38,
    22,22,26,27,29,34,37,40,
    22,26,27,29,32,35,40,48,
    26,27,29,32,35,40,48,58,
    26,27,29,34,38,46,56,69,
    27,29,35,38,46,56,69,83,}
blk_zig[0] = blk_dct[0]/iqtabs[0];
for (i = 1; i < 64; i++) {
    j = zscan[i];
    blk_zig[i] = blk_dct[j]*16/(iqtabs[j]*q_scale);
}
```

By arranging the blocks in zig-zag order, the elements (coefficients) in the block are arranged starting from the elements corresponding to lower frequency elements. There are more 0 elements further back in the series to make run-level compression easier.

The example below shows a macroblock going from DCT transformation to zig-zag transformation. These transformations are for a 16x16 black-and-white block. Because the data is black and white, the elements in the color-difference block are all zero.

(QUANT = 8)

Cb

src:	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
dct:	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
zig:	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0

Cr

src:	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
dct:	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
zig:	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0

Y0

src:							
-128	-128	-128	-128	-128	-128	-128	-128
-128	-128	-128	-128	-128	-68	-51	-39
-128	-128	-128	-83	-59	-42	-28	-17
-128	-128	-83	-56	-38	-23	-11	0
-128	-128	-59	-38	-22	-8	5	14
-128	-68	-42	-23	-8	6	18	28
-128	-51	-28	-11	5	18	30	40
-128	-39	-17	0	14	28	40	51
dct:							
-229	-155	-23	-21	-10	-4	-1	-4
-155	42	34	23	12	12	7	0
-23	34	7	-6	-13	-10	-7	-5
-21	23	-6	-5	-2	1	3	3
-10	12	-13	-2	5	4	5	6
-4	12	-10	1	4	-4	-4	1
-1	7	-7	3	5	-4	-6	-1
-4	0	-5	3	6	1	-1	2
zig:							
-229	-19	-19	-2	5	-2	-2	3
3	-2	-1	2	1	2	-1	0
1	0	0	1	0	0	1	-1
0	-1	1	0	0	0	-1	0
0	-1	1	0	0	-1	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Y1

src:							
-128	-128	-128	-128	-128	-128	-128	-128
-29	-21	-16	-15	-128	-128	-128	-128
-7	1	7	11	12	5	-128	-128
10	18	25	30	33	32	23	-128
25	33	40	45	49	50	46	-128
38	46	53	59	63	65	64	56
50	58	65	71	74	78	78	73
60	69	75	82	86	89	89	85
dct:							
-1	55	-57	20	-23	16	-5	-4
-263	57	-8	-7	2	2	6	-9
-57	-34	48	-40	26	-14	10	-8
-38	-35	22	-5	-2	7	-9	8
-22	-18	-10	27	-20	14	-16	14
-3	-30	-2	14	2	-11	3	5
1	-19	-6	9	12	-23	13	-1
-12	8	-19	17	-1	-9	6	0
zig:							
-1	7	-33	-6	7	-6	2	-1
-3	-3	-2	-3	4	-1	-2	1
0	-3	2	-1	0	0	-2	-1
0	2	0	0	0	0	-1	0
2	0	-1	-1	1	0	1	-1
0	1	0	0	0	1	0	1
-1	1	1	-1	-1	0	1	0
-1	0	0	0	0	0	0	0

Y2

src:							
	-128	-29	-7	10	25	38	50
	-128	-21	1	18	33	46	58
	-128	-16	7	25	40	53	65
	-128	-15	11	30	45	59	71
	-128	-128	12	33	49	63	74
	-128	-128	5	32	50	65	78
	-128	-128	-128	23	46	64	78
	-128	-128	-128	-128	-128	56	73
dct:							
	-1	-263	-57	-38	-22	-3	1
	55	57	-34	-35	-18	-30	-19
	-57	-8	48	22	-10	-2	-6
	20	-7	-40	-5	27	14	9
	-23	2	26	-2	-20	2	12
	16	2	-14	7	14	-11	-23
	-5	6	10	-9	-16	3	13
	-4	-9	-8	8	14	5	-1
zig:							
	-1	-33	7	-6	7	-6	-3
	-1	2	-2	-1	4	-3	-2
	-1	2	-3	0	1	0	0
	0	-1	-2	0	-1	-1	0
	0	-1	0	0	-1	1	0
	1	0	0	-1	0	0	1
	0	0	-1	-1	1	1	0
	0	1	0	0	0	0	0

Y3

src:							
	70	78	85	91	96	99	99
	78	86	94	99	104	106	106
	85	94	101	106	110	112	112
	91	99	106	112	115	118	117
	96	104	110	115	120	120	118
	99	106	112	118	120	120	112
	99	106	112	117	118	112	-128
	94	103	107	109	-128	-128	-128
dct:							
	300	109	-84	28	-33	26	-6
	109	-150	53	-7	14	-18	-5
	-84	53	23	-52	28	-11	21
	28	-7	-52	65	-32	8	-14
	-33	14	28	-32	-1	22	-12
	26	-18	-11	8	22	-39	27
	-6	-5	21	-14	-12	27	-21
	-10	20	-26	19	-3	-8	7
zig:							
	300	14	14	-9	-19	-9	3
	5	3	-3	-1	2	-1	-3
	1	-4	-4	1	2	0	-1
	5	2	-1	0	-1	0	-1
	-2	-1	0	-1	1	1	0
	0	1	1	-1	-1	1	1
	-1	1	-1	-2	-1	1	0
	1	0	0	-1	0	0	0

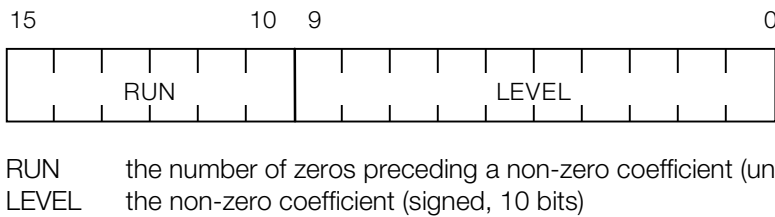
Runlevel Conversion

Because there tend to be sequences of zeros in zigzag ordered blocks, data is compressed by combining two or more continuous zeros. This is called runlevel conversion. Runlevel codes consist of the following two-dimensional data:

(run,level) = (the number of zeros preceding level, value of the element)

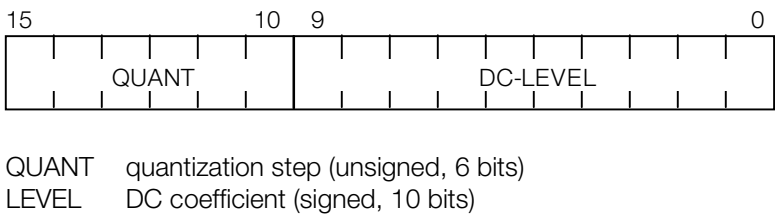
An actual runlevel code has 16 bits and is in the following format:

Figure 1-12: Runlevel Code Format



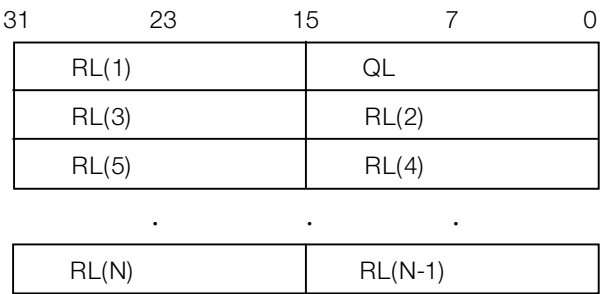
A runlevel code at the beginning of a block always has a run field of 0. (The runlevel is (0,0) even if the DC element is 0.) Therefore, the quantization step (QUANT) is always placed in the run field of the first runlevel code in a block.

Figure 1-13: Quantization Step Placement



A runlevel pair RL is transferred as a two-set 32-bit pack.

Figure 1-14: Runlevel Pairs



The number of run-level pairs varies from block to block, so a (run,level)=(63,128)=0xfe00 is inserted as a termination code after the final run-level pair in a block.

For example,

```

-229 -19 -19 -2 5 -2 -2 3 3 -2 -1 2 1 2 -1 0 1 0 0
1 0 0 1 -1 0 -1 1 0 0 0 -1 0 0 -1 1 0 0 -1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

```

A matrix having this data would be organized as follows:

```

(0,-229)(0,-19)(0,-19)(0,-2)(0,5)(0,-2)(0,3)(0,3)(0,-2)(0,-1)
(0,2)(0,1)(0,2)(0,-1)(1,1)(2,1)(2,1)(0,-1)(1,-1)(0,1)(3,-1)(2,-1)
(0,1)(2,-1)(NOP)

```

A NOP: (run,level)=(63,128)=0xfe00 is inserted at the end of the block as a delimiter. This indicates the end of the block and also indicates that the subsequent block elements will be a string of zeros. For details, see the next section.

The previous block would be converted as follows (QUANT=8):

```

0x03ED231B,0x03FE03ED,0x03FE0005,0x000303FE,0x03FE0003,0x000203FF,0x00020001
,0x040103FF,0x08010801,0x07FF03FF,0x0FFF0001,0x00010BFF,0xFE000000,

```

The RL Data Format

RL data combines all the runlevel transformed block data required for decoding. RL data includes a one-word (32-bit) header and a footer for maintaining 32-word boundaries.

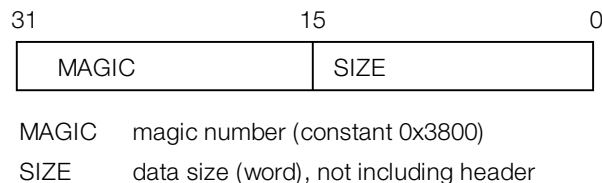
Each runlevel making up a block is delimited with a NOP (0xfe00). NOPs indicate the following:

1. When a NOP appears at a position other than between blocks: That block is ended and the subsequent block elements are filled with 0.
2. When a NOP appears between blocks: The NOP is skipped.
3. When two NOPs appear in the middle of a block, the first NOP functions as a delimiter and the second NOP is simply skipped.

The blocks are assumed to be in the following order: Cb, Cr, Y0, Y1, Y2, Y3. Six blocks make a single macroblock. The number of blocks in an RL must be a multiple of a macroblock unit (6).

A header word has the following format:

Figure 1-15: Header Word Format



The data size must be a multiple of 32. Consequently, a footer containing an appropriate number of NOPs is added as padding to make the data a fixed length.

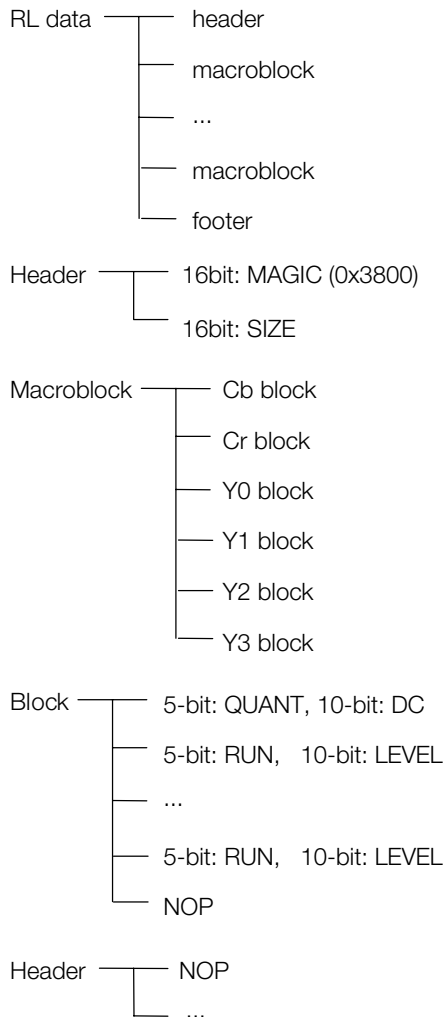
The results of a runlevel transformation of the previous macroblock is shown below. Because all the elements in the Cb and Cr blocks are 0, it can be seen that both blocks are contained in one word that includes the delimiter.

```

0x38000060, 0xFE002000, 0xFE002000, 0x03ED231B, 0x03FE03ED, 0x03FE0005,
0x000303FE, 0x03FE0003, 0x000203FF, 0x00020001, 0x040103FF,
0x08010801, 0x07FF03FF, 0xFFF0001, 0x00010BFF, 0xFE000BFF, 0x000723FF,
0x03FA03DF, 0x03FA0007, 0x03FF0002, 0x03FD03FD, 0x03FD03FE, 0x03FF0004,
0x000103FE, 0x000207FD, 0x0BFE03FF, 0x040203FF, 0x040213FF, 0x03FF07FF,
0x04010001, 0x040103FF, 0x04010C01, 0x000103FF, 0x03FF0001, 0x040103FF,
0xFE0007FF, 0x03DF23FF, 0x03FA0007, 0x03FA0007, 0x03FD03FD, 0x000203FF,
0x03FF03FE, 0x03FD0004, 0x07FF03FE, 0x03FD0002, 0x08020401, 0x03FE07FF,
0x03FF07FF, 0x07FF0402, 0x00010BFF, 0x000107FF, 0x08010BFF, 0xBFF03FF,
0x000103FF, 0x07FF0001, 0xFE000401, 0x000E212C, 0x03F7000E, 0x03F703ED,
0x00050003, 0x00030005, 0x03FF03FD, 0x03FF0002, 0x000203FD, 0x03FC0001,
0x000103FC, 0x07FF0002, 0x00050002, 0x03FF0002, 0x07FF07FF, 0x03FE03FE,
0x07FF03FF, 0x00010001, 0x00010C01, 0x03FF03FF, 0x00010001, 0x03FF03FF,
0x03FF0001, 0x03FF03FE, 0x04010001, 0xBFF0001, 0xFE00FE00, 0xFE00FE00,
0xFE00FE00, 0xFE00FE00, 0xFE00FE00, 0xFE00FE00, 0xFE00FE00, 0xFE00FE00,
0xFE00FE00, 0xFE00FE00, 0xFE00FE00, 0xFE00FE00, 0xFE00FE00, 0xFE00FE00,
0xFE00FE00, 0xFE00FE00,

```

The syntax for RL data is as follows:



VLC

VLC (Variable Length Coding) is based on Huffman coding theory with reversible data compression using a fixed dictionary (code book). Huffman coding takes advantage of the probabilities of occurrence of data (known as a *word*, in this case, a runlevel code). Data is compressed by assigning the shorter codes to words having higher probabilities of occurrence.

For example, the word (run,level) = (0, 1) has a high frequency of occurrence, and so is assigned the code 01 (2 bits). On the other hand, (run,level) = (23,1) has a low frequency of occurrence and so it is assigned a code of 00000000111101 (14 bits).

The look-up table for words and codes is called a dictionary (code book). Huffman coding assumes that the probabilities of occurrence of data is known ahead of time, and that these probabilities do not change over time (non-memory data source). In this case, the code book is established in the beginning, and this code book is called a static dictionary.

Code books vary according to the characteristics of the image source. For this reason, to achieve optimum compression ratios, code books should actually be prepared for each image according to the specific characteristics of that image. However, to simplify things, libpress prepares a single standard code book that is applied to standard images. This code book is based on a runlevel code distribution in a typical natural image. Different code books should be used for applications such as animation, where the probability distribution in the runlevel codes is different from that of natural images.

The contents of the standard code book are appended at the end of this text. The bit sequences in the left-hand column correspond to the runlevel codes in the right-hand column. EOB is an abbreviation of EndOfBlock and represents an NOP (0xfe00).

Word (32-bit) boundaries are not significant in the coded data. Because byte ordering in the PlayStation is little-endian, coded bit sequences are ordered starting with low-order bits. Code sequences packed in this way are called *bitstreams*.

If a runlevel code that is not in the code book appears, an escape code 000001 followed by an FLC (fixed-length code) is used. The (run, level) of an FLC are described at the end of this section.

VLC transformation during decoding is performed with software. Decoding operations for the rest of the runlevel, encoded data are performed with hardware by MDEC. Thus, VLC decoding and runlevel decoding are generally performed with pipelines in order to decrease processing time. In the library, the DecDCTvlc() function performs VLC decoding, and the DecDCTin()/DecDCTout() functions perform DCT processing for runlevel and after.

BS Format

RL format data on which VLC transformation has been performed is called BS data. The BS format is made up of bitstreams in which headers and blocks are coded.

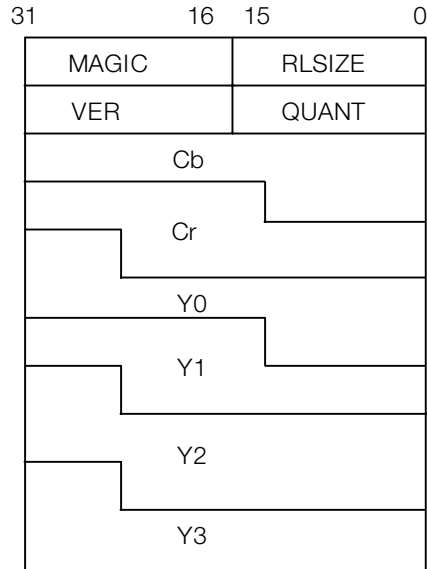
The header is made up of two words and has the format shown below. In the current BS format, it is assumed that all the blocks contained have identical quantization steps (QUANT). Consequently, QUANT is stored only once in the header.

Figure 1-16: BS Header Format

31	15	0
MAGIC	RLSIZE	
VER	QUANT	

MAGIC magic number (constant 0x3800)
 RLSIZE data size (word) of runlevel, not including header
 VER version number (constant 2)
 QUANT quantization step

A block contains a 10-bit DC element followed by a bitstream converted from a runlevel representing the AC element. The block boundaries are not necessarily at the word boundaries.

Figure 1-17: BS Blocks

The RL data footer padding (0xfe00) is not included in BS data. So when BS decoding is performed, DecDCTvlc() automatically adds footers to maintain 32-word boundaries.

The BS format does not contain data for the frame size or frame rate to be used in the final playback. These settings need to be indicated to the program in other ways.

The VLC transformation of the runlevel from the previous example is shown below.

```
0x38000060, 0x00020008, 0x02C60020, 0x8039C01C, 0x29284931, 0x6476A4F4,
0xE3D752BB, 0xE050977F, 0x860500BC, 0xE5960868, 0x3A7192C3,
0x278C2D1C, 0xFCDD3463, 0xB7EC8E6F, 0x005EF7FE, 0x60500508, 0x7D098659,
0xA5D0E185, 0x3A5F5B04, 0xE7CB8C75, 0xD9DA575F, 0x30004B00, 0x200EC006,
0x51304062, 0x7D1C9851, 0xD0DCB460, 0x311D8741, 0xEFB1DD29, 0xBFDDDBFD,
0x5CFF3F36, 0x00008000,
```

Actual VLC Decoding

After the header section, the block data which makes up each macro block is Huffman coded and stored. An outline of VLC code decoding is described here.

It is assumed that the block start word is the decoding of the

0x061FF0D1

block. The DC (direct current) component is registered at the start of the block data. Generally, the value of the element at the upper left corner of the block (0,0) is called the DC component and other elements are called AC (alternating current) components.

Since MPEG bit streams are stipulated by short word big-endian, they are converted to:

```
0x061FF0D1 = 1111 0000 1101 0001 0000 0110 0001 1111
                F    0    D    1    0    6    1    F
```

DC is registered as fixed code, so the head 10bit (1111 0000 11) becomes the DC value. In such a case, since MSB is 1, note that DC will be a negative value and

1111 0000 11 = -60

DC will have a 10bit code.

When the block is a color difference block (Cb/Cr), it will become the DC value as is. However, when the block is a luminance block (Y0, Y1, Y2, Y3), a 128 offset will be added to the DC component. In this example, the DC component is:

$$-60 + 128 = 68$$

This is the actual start block DC level. With a luminance block, it can be seen that the average luminance value of the encoded image (upper right 8x8 area) start block is 68. The purpose of this offset is to set the dynamic range between the luminance (Y) block and the color difference block (Cb, Cr) to (-128, 128). Note that the offset is not added in the color difference block (Cb, Cr).

The run level format AC component is coded and then follows after the DC component.

In 0x061FF0D1 = 1111 0000 1101 0001 0000 0110 0001 1111

the 01000 bitstream following 1111000011 indicates

$$(\text{run}, \text{level}) = (0, +2).$$

This can be decoded by referring to the bit stream entered last and the run level compatibility (codebook).

The decoding block run level from the above is:

$$(\text{run}, \text{level}) = (\text{QUANT}, -60), (0, 2) \dots$$

The QUANT value BS header section is used as is.

BS Format Version

Versions 2 and 3 of the current bs format have different DC component decoding methods.

Table 1-3: BS Format Versions

version	Block DC component
2	FLC (Fixed Length Code)
3	VLC (Variable Length Code)

As described above, the DC section is coded as is (fixed code format) and no variable coding is performed. The forecasted DC coding and Huffman coding are packaged in version 3.

DC forecasts in version 3 are carried out as follows:

- (1) Codes the difference from the DC component in the previous block.
- (2) VLC encodes the differential value bit width.
- (3) Registers the bit width of the VLC encoded differential value + the differential value

The luminance block (Y0, Y1, Y2, Y3) and color difference block (Cb, Cr) use a different VLC table (Huffman table) than that used in (2).

The table is:

Table 1-4: DC Codebook

Differential value bit width	Luminance Block	Color Difference Block
0	100	00
1	00	01
2	01	10
3	101	110
4	110	1110
5	1110	11110
6	11110	111110
7	111110	1111110
8	1111110	11111110

In other words, with Y block, when 111110 comes to the header, the DC value which follows next will have coding and will be an 8bit width. This is actually shifted 1 bit to the left. For example,

111110 11111111

indicates -255.

In version 3 encoding, the Huffman coding of the DC component improves the compression ratio, but this also increases the processing time. Version 2 should be used in applications where CPU processing time can be a bottleneck.

Decoding Speed

The maximum MDEC decompression speed is 9000 macroblocks/second. This is equivalent to a rate of 30 frames of 320x240 images decompressed in a second. This decompression speed has no relation to the compression ratio. Of course, the resolution of the images is in inverse proportion to the playback frame rate. In other words, 320x240 images can be played back at 30 frames/second, and 640x240 images can be played back at 15 frames/second.

Table 1-5: Decompression Speed

Resolution	160x240	320x240	640x240	640x480
Frames/second	60	30	15	7.5

In practice, the CPU performs VLC decoding through DCT decompression in the background. If the CPU is performing other compute-intensive processing, VLC decoding can become a bottleneck.

The transfer rate for the CD-ROM can be selected as either 150KB/sec (standard speed) or 300KB/sec (double speed). With double-speed playback, data can be read from the CD-ROM at a rate of 30 frames/second if the bitstream for one frame is compressed to 10KB (=300KB/30) or less.

Table 1-6: Transfer Speed

Data size	5KB	10KB	20KB	40KB
Frames/second	60	30	15	7.5

The playback rate for animation is determined by these two factors. For example, double-speed playback can be performed if the bitstream for one frame (320x240) recorded on a CD-ROM is compressed to 10KB or less. As long as this condition is met, the frame rate, the image resolution and the compression ratio can be set as desired..

Improving Image Quality

The following methods are possible ways to improve the image quality in MDEC animations.

Preprocessing

Image noise at the initial stage, i.e. the image input stage, is the biggest factor in image quality during playback. For this reason, very high quality images should be used. Image sources need to be at least 24-bit images.

The signal to noise (S/N) ratio for image sources can be decreased by horizontal filtering as well as filtering over time.

If a high-resolution image capturing system can be used, sampling should be done at a high resolution (high sampling rate), and then a lowpass filter should be applied for down-conversion. If 2x oversampling can be performed, then random noise (white noise) can be reduced by up to 1/2.

If possible, a 60 frames/second image source should be used in creating images for 15 frames/second playback, and one frame should be created from four sequential frames. Simply creating new frames from the average of four frames can reduce noise by up to 1/4.

When macroblocks are to be created by compressing 16x16 pixel Cb, Cr (color data) blocks to 8x8 pixels, raising the filtering coefficient to a higher dimension can eliminate aliasing. Block noise (noise that appears in a 16x16 pixel unit) during decoding is due to the Cb, Cr blocks. Block noise in an image source can be reduced by taking advantage of the correlation between adjacent blocks during Cb, Cr block creation.

Block noise is visually conspicuous when the encoded noise takes on different values between frames. This can occur when there are large changes in the quantization steps between adjacent frames. When BS data is recorded by interleaving with audio sectors, the audio sectors and the movie sectors are asynchronous, and the data sizes assigned to each frame vary. This leads to variations in the quantization steps and appears as noise. Noise caused by variation in quantization can often be seen in cases where the image source has a lot of static areas (animation backgrounds). In these cases, encoding must always be performed with fixed sectors.

Evaluation during Encoding

The displays of some home televisions and monitors give more emphasis than necessary to the color-difference signal. In addition, there are cases where gamma correction is not performed correctly. This means that the error in the color-difference component during calculations is evaluated as being lower than it actually is, depending on the situation. There are also cases where the color-difference signal does not have a resolution of 1/4 to start with, and this can cause the playback image to appear deteriorated on specific TV sets. In these cases, the evaluation functions used in encoding needs to be modified.

Making changes in the code book

The current code book is based on natural images. The compression ratio is lower for images such as animation which have a probability distribution different from natural images. In these cases, a dedicated code book should be prepared. The current version of libpress does not allow switching of code books, but this feature is planned for future versions. Also, since VLC encoding is performed with software, it is possible for an application to perform its own coding.

Correlation between Frames

In encoding that takes advantage of frame correlations, the benefits are outweighed by the penalties, such as the need for processing dropped frames and the CPU load during decoding. However, for image sources that obviously have a lot of fixed scenes, it would be possible to take solely 0-degree correlations. In such cases, it would be necessary to have data indicating correlation in addition to the BS format data.

Processing during Playback

In general, better results are obtained by playing back in 24-bit mode compared to playing back in 16-bit mode. This is because a standard encoder truncates the low-order bits when going from 24 bits to 16 bits, and quantization errors are not taken into consideration in the evaluation functions. When playing back at 16 bits, the input image source should be rounded off to 16 bits in the manner shown below, rather than simply truncating to 16 bits.

$$\text{pix16} = (\text{pix24} + 0 \times 10) \& 0\text{xfe}$$

Codes

The VLC Code Book

The VLC code book is shown below. This code book is compatible with the one used in MPEG microlayers.

I.			
=====bit pattern		(run,level)-----	
---10	(EOB)110	0 +1111	0 -10110
1 +10111	1 -101000	0 +201001	0 -201010
2 +101011	2 -1001010	0 +3001011	0 -3001110
3 +1001111	3 -1001100	4 +1001101	4 -10001100
1 +20001101	1 -20001110	5 +10001111	5 -10001010
6 +10001011	6 -10001000	7 +10001001	7 -1000001
(ESCAPE)00001100	0 +400001101	0 -400001000	2 +200001001
2 -200001110	8 +100001111	8 -100001010	9 +100001011
9 -1001001100	0 +5001001101	0 -5001000010	0 +6001000011
0 -6001001010	1 +3001001011	1 -3001001000	3 +2001001001
3 -2001001110	10 +1001001111	10 -1001000110	11 +1001000111
11 -1001000100	12 +1001000101	12 -1001000000	13 +1001000001
13 -100000010100	0 +700000010101	0 -700000011000	1 +400000011001
1 -400000010110	2 +300000010111	2 -300000011110	4 +200000011111
4 -200000010010	5 +200000010011	5 -200000011100	14 +100000011101
14 -100000011010	15 +1-----		
II.			
=====bit pattern		(run,level)-----	
00000011011	15 -100000010000	16 +100000010001	16 -1000000111010
0 +80000000111011	0 -80000000110000	0 +90000000110001	0 -90000000100110
0 +100000000100111	0 -100000000100000	0 +110000000100001	0 -
110000000110110	1 +50000000110111	1 -50000000101000	2 +40000000101001
2 -40000000111000	3 +30000000111001	3 -30000000100100	4 +30000000100101
4 -30000000111100	6 +20000000111101	6 -20000000101010	7 +20000000101011
7 -20000000100010	8 +20000000100011	8 -20000000111110	17 +10000000111111
17 -10000000110100	18 +10000000110101	18 -10000000110010	19
+10000000110011	19 -10000000101110	20 +10000000101111	20 -10000000101100
21 +10000000101101	21 -100000000110100	0 +1200000000110101	0 -
1200000000110010	0 +1300000000110011	0 -1300000000110000	0 +1400000000110001
0 -1400000000101110	0 +1500000000101111	0 -1500000000101100	1
+600000000101101	1 -600000000101010	1 +700000000101011	1 -700000000101000
2 +500000000101001	2 -500000000100110	3 +400000000100111	3 -400000000100100
5 +300000000100101	5 -300000000100010	9 +200000000100011	9 -200000000100000
10 +200000000100001	10 -200000000111110	22 +100000000111111	22 -
100000000111100	23 +100000000111101	23 -1	

III.

```

=====bit pattern (run,level)-----
00000000111010 24 +100000000111011 24 -100000000111000 25 +100000000111001
25 -100000000110110 26 +100000000110111 26 -100000000111110 0
+16000000000111111 0 -16000000000111100 0 +17000000000111101 0 -
17000000000111010 0 +18000000000111011 0 -18000000000111000 0
+19000000000111001 0 -19000000000110110 0 +20000000000110111 0 -
20000000000110100 0 +21000000000110101 0 -21000000000110010 0
+22000000000110011 0 -22000000000110000 0 +23000000000110001 0 -
23000000000101110 0 +24000000000101111 0 -24000000000101100 0
+25000000000101101 0 -25000000000101010 0 +26000000000101011 0 -
26000000000101000 0 +27000000000101001 0 -27000000000100110 0
+28000000000100111 0 -28000000000100100 0 +29000000000100101 0 -
29000000000100010 0 +30000000000100011 0 -30000000000100000 0
+31000000000100001 0 -31000000000110000 0 +32000000000110001 0 -
32000000000101110 0 +33000000000101111 0 -33000000000101100 0
+34000000000101101 0 -34000000000101010 0 +35000000000101011 0 -
35000000000101000 0 +36000000000101001 0 -36000000000100110 0
+37000000000100111 0 -37000000000100100 0 +38000000000100101 0 -
38000000000100010 0 +39000000000100011 0 -39000000000100000 0
+40000000000100001 0 -40000000000111110 1 +80000000000111111 1 -
80000000000111100 1 +90000000000111101 1 -90000000000111010 1 +10
-----

```

IV.

```

=====bit pattern (run,level)-----
000000000111011 1 -10000000000111000 1 +11000000000111001 1 -11000000000110110
1 +12000000000110111 1 -12000000000110100 1 +13000000000110101 1 -
13000000000110010 1 +14000000000110011 1 -14000000000100110 1
+15000000000100111 1 -15000000000100100 1 +16000000000100101 1 -
16000000000100010 1 +17000000000100011 1 -17000000000100000 1
+18000000000100001 1 -18000000000101000 6 +30000000000101001 6 -
30000000000110100 11 +20000000000110101 11 -20000000000110010 12
+20000000000110011 12 -20000000000110000 13 +20000000000110001 13 -
20000000000101110 14 +20000000000101111 14 -20000000000101100 15
+20000000000101101 15 -20000000000101010 16 +20000000000101011 16 -
20000000000111110 27 +10000000000111111 27 -10000000000111100 28
+10000000000111101 28 -10000000000111010 29 +10000000000111011 29 -
10000000000111000 30 +10000000000111001 30 -10000000000110110 31
+10000000000101111 31 -1-----

```

EOB: End Of Block

ESC: Escape

FLC Code

ESC is followed by a FLC(fixed-length code). The (run, level) of an FLC is defined as follows:

```

=====Fixed-length Code run-----
000000 000001 1000010 2
.....
111111 63-----

=====Fixed-length Code level-----
-1000 0000 0000 0001 -2561000 0000 0000 0010 -2551000 0000 0000 0011 -254
.....
1000 0000 0111 1111 -1291000 0000 1000 0000 -128-----
1000 0001 -1271000 0010 -126 .....
1111 1110 -21111 1111 -10000 0001 10000 0010
2 .....
0111 1111 127-----0000 0000 1000 0000
1280000 0000 1000 0001 129 .....
0000 0000 1111 1111 255-----

```

XA: CD-ROM Voice Data

XA is the PlayStation CD-ROM XA voice data format. The typical extension in DOS is “.XA”.

The XA format is based on the following specifications. The XA file output by RAW2XA has a sub-header.

CD-ROM XA

SYSTEM DESCRIPTION CD-ROM XA

Copyright May 1991

Chapter 2:

3D Graphics

RSD: 3D Model Data

Overview

The RSD format is a data format that is used to represent 3D models. The PlayStation artist tools are designed to work with RSD-formatted models.

RSD-formatted models are represented as four separate files. These files are sometimes referred to collectively as "RSD" or "RSD data".

The four files comprising an RSD-formatted model are:

- RSD File
The RSD file describes relationships between PLY/MAT/GRP files, texture files and extended files.
- PLY File
The PLY file contains positional information about the vertices of polygons.
- MAT File
The MAT file contains material information on polygons.
- GRP File
The GRP file contains grouping information on polygons.

Starting with this version, four types of extended files and one type of sub-extended file have been added to the RSD format. These files were added to provide support for HMD.

The new extended file types are:

- MSH File
The MSH file contains information on how polygons are linked.
- PVT File
The PVT file contains information on offsets for centers of rotation.
- COD File
The COD file contains information on COORDINATE for VERTEX.
- MOT File
The MOT file contains animation information.

The sub-extended file type is:

- OGP File
The OGP file contains grouping information for VERTEX.

The RSD file contains information describing the relationships among all of the other files. Thus, the collection of files that describe the structure of an object can be determined from the RSD file. (Since an OGP file is a sub-extended file, it can only be specified from within a COD file.)

When all of the RSD files are located in a single directory, they can be specified by their filenames alone. If the files are in separate directories they must be referenced using their relative (or absolute) pathnames.

(Note: This is the same convention as a TIM file.)

All files in RSD are text files with individual lines delimited by newline characters (either LF or CR/LF). Lines beginning with '#' are treated as comments.

Each of the files described in this manual is based on the following versions.

- RSD
Version 3.0
- PLY
Version 3.0
- MAT
Version 3.0
- GRP
Version 3.0
- MSH
Version 1.0
- PVT
Version 1.0
- COD
Version 1.0
- OGP
Version 1.0
- MOT
Version 1.0

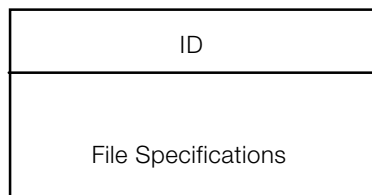
RSD File

The RSD file contains information on how the PLY/MAT/GRP files, texture files and extended files are combined for a given object.

Beginning with this version, extended files can be used to represent multiple objects with a single set of RSD files, allowing data to be managed in character units.

Overall structure

Figure 2-1: Overall structure of an RSD file



ID

The ID is a string of the form "@RSDnnnnnn" (where nnnnnn is a number) indicating the RSD file format version number. For example, version 3.0 would be specified with the string "@RSD970401".

File Specifications

- PLY File specification
PLY=PLY filename
- MAT File specification
MAT=MAT filename
- GRP File specification
GRP=GRP filename

- MSH File specification
MSH=MSH filename
- PVT File specification
PVT=PVT filename
- COD File specification
COD=COD filename
- MOT File specification
NMOT= number of MOT files
MOT[n]= filename of nth MOT file
:
:
- TIM(texture) File specification
NTEX= number of TIM (texture) files
TEX[n]= filename of nth TIM (texture) file
:
:

Sample file

The following is a simple example of an RSD file

Figure 2-2: Sample RSD file

```
@RSD970401
PLY=sample.ply
MAT=sample.mat
GRP=sample.grp
MSH=sample.msh
PVT=sample.pvt
COD=sample.cod
NMOT=1
MOT[0]=anim.mot
NTEX=3
TEX[0]=texture.tim
TEX[1]=texture2.tim
TEX[2]=texture3.tim
```

PLY File

The PLY file contains positional information about the vertices of polygons and related objects.

If extended files are not used, the coordinate system of a PLY file is the same as the coordinate system of the extended PlayStation library (libgs). In other words, the X axis is oriented (increases) in the horizontal direction towards the right side of the screen, the Y axis is oriented in the vertical direction towards the bottom of the screen, and the Z axis is oriented into the screen.

When extended files are used during HMD conversion, each vertex contained in the COD file is converted into the coordinate system of the extended PlayStation library (libgs), taking into account the rotations and translations specified in the PVT file.

The direction (obverse or reverse) of a single-faced polygon is determined by the order in which the vertices are described in a POLYGON descriptor. The obverse of the polygon is defined as the plane for which the vertices of a polygon are described clockwise (CW).

Overall structure**Figure 2-3: Overall structure of a PLY file**

ID
Data length record
VERTEX descriptor :
NORMAL descriptor :
POLYGON descriptor :

ID

The ID is a string of the form "@PLYnnnnnn" (where nnnnnn is a number) indicating the PLY file format version number. For example, version 3.0 would be specified with the string "@PLY970401".

Data length record

The data length record describes the number of data lines for each of the data blocks which follow. The items on each line are delimited by a space or tab.

Figure 2-4: Data length record in a PLY file

VERTEX count	NORMAL count	POLYGON count
-----------------	-----------------	------------------

VERTEX descriptor

The VERTEX descriptor consists of three floating point values which represent the coordinates of a vertex. There is one vertex per line.

Figure 2-5: VERTEX descriptor in a PLY file

X coord	Y coord	Z coord
---------	---------	---------

NORMAL descriptor

The NORMAL descriptor consists of three floating point values which represent the elements of a normal vector.

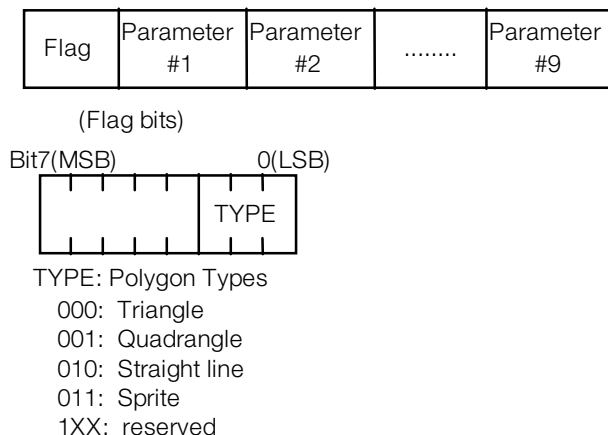
Figure 2-6: NORMAL descriptor in a PLY file

X element	Y element	Z element
-----------	-----------	-----------

POLYGON descriptor

The POLYGON descriptor consists of a flag that indicates the type of polygon together with nine parameters that describe the polygon. The meaning of the parameters varies according to the value of the TYPE field in the flag.

Figure 2-7: POLYGON descriptor in a PLY file



The flag is a hexadecimal integer that indicates the type of polygon. The '0x' prefix is not used.

[Triangular and Quadrangular Polygons]

The parameter section describes the vertices (VERTEX) and normals (NORMAL) of a polygon. Vertices and normals are represented as integers with values in the range from 0 to 3, where the value indicates the location of the data within the group (0 represents the start of the group).

When flat shading is to be applied to a polygon, the normals for all the vertices are identical, and the value of the first vertex is used. When Gouraud shading is to be applied, the normals have different values.

For triangles, the data corresponding to the fourth vertex (vertex 3 and normal 3) is set to 0.

For quadrangular polygons, vertices 1, 2 and 3 form one triangle, and vertices 2, 3 and 4 form a second triangle.

Figure 2-8: POLYGON descriptor for triangular/quadrangular polygons

Flag	Vertex 0	Vertex 1	Vertex 2	Vertex 3	Normal 0	Normal 1	Normal 2	Normal 3
------	-------------	-------------	-------------	-------------	-------------	-------------	-------------	-------------

[Straight Lines]

The parameter section contains the VERTEX numbers of the two endpoints of the line.

Figure 2-9: POLYGON descriptor for straight lines

Flag	Vertex 0	Vertex 1	0	0	0	0	0	0
------	-------------	-------------	---	---	---	---	---	---

[Sprites]

In modeling data, sprites are rectangular images located in 3D space. They can be viewed as textured polygons that are always oriented toward the viewpoint.

The parameter section contains VERTEX data which represents the sprite position along with the width and height of the sprite image (also known as the sprite pattern).

Figure 2-10: Polygon descriptor for sprites

Flag	Vertex 0	Width	Height	0	0	0	0	0
------	-------------	-------	--------	---	---	---	---	---

Example

The following is a simple example of a PLY file.

Figure 2-11: Sample PLY file

```
@PLY970401
# Number of Items
8 12 12
# Vertex
0 0 0
0 0 100
0 100 0
0 100 100
100 0 0
100 0 100
100 100 0
100 100 100
# Normal
0.000000E+00 0.000000E+00 -1.000000E+00
0.000000E+00 0.000000E+00 -1.000000E+00
1.000000E+00 0.000000E+00 -0.000000E+00
1.000000E+00 0.000000E+00 0.000000E+00
0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00 0.000000E+00 1.000000E+00
-1.000000E+00 -0.000000E+00 -0.000000E+00
-1.000000E+00 0.000000E+00 0.000000E+00
-0.000000E+00 1.000000E+00 0.000000E+00
0.000000E+00 1.000000E+00 0.000000E+00
0.000000E+00 -1.000000E+00 0.000000E+00
0.000000E+00 -1.000000E+00 0.000000E+00
# Polygon
0 6 2 0 0 0 0 0 0
0 6 0 4 0 1 1 1 0
0 7 6 4 0 2 2 2 0
0 7 4 5 0 3 3 3 0
0 3 7 5 0 4 4 4 0
0 3 5 1 0 5 5 5 0
0 2 3 1 0 6 6 6 0
0 2 1 0 0 7 7 7 0
0 7 3 2 0 8 8 8 0
0 7 2 6 0 9 9 9 0
0 4 0 1 0 10 10 10 0
0 4 1 5 0 11 11 11 0
```

MAT File**Overall structure**

Figure 2-12: Overall structure of a MAT file

ID
MATERIAL descriptor count
MATERIAL descriptor : :

ID

The ID is a string of the form "@MATnnnnnn" (where nnnnnn is a number) indicating the MAT file format version number. For example, version 3.0 would be specified with the string "@MAT970401".

MATERIAL descriptor count

The MATERIAL descriptor count contains the number of MATERIAL descriptors which follow (i.e. number of lines).

MATERIAL descriptor

The MATERIAL descriptor contains the material information for a specific polygon.

Figure 2-13: MATERIAL descriptor for a MAT file

Polygon No.	Flag	Shading	Material Info...
-------------	------	---------	------------------

[Polygon No.]

A polygon number is an index into a POLYGON group in the PLY file. The polygon number is used to represent a particular polygon. Multiple polygons can be included by specifying a range on a single line.

Table 2-1: Polygon Numbers

Values	Specified polygons
1	1 only
0-5	0 1 2 3 4 5
2,4,6	2 4 6

[Flag]

The flag is a hexadecimal integer which represents the material attributes of a polygon. The '0x' prefix is not used. The meaning of the bits in the flag are as follows.

Bit0: Light-source calculation mode
 0: Light-source calculations performed
 1: Fixed color

When light-source calculations are performed, the rendering color is determined by the orientation of the light source relative to the polygon. When a fixed color is used, the color is constant regardless of orientation.

Bit 1: Back face
 0: Single-faced polygon
 1: Double-faced polygon

Bit 2: Semitransparency flag
 0: Semitransparency OFF
 1: Semitransparency ON

When semitransparency is ON, untextured polygons are always made to be semitransparent. Polygons with semitransparent textures are made semitransparent/opaque/transparent depending on the STP bit of the texture data.

Bits 3-5: Semitransparency rate
 000:50% back + 50% polygon
 001:100% back + 100% polygon
 010:100% back - 100% polygon
 011:100% back + 25% polygon
 1XX:reserved

The current library does not permit semitransparency rates to be changed for individual polygons.

bit6 Reserved (must be 0)

bit7 Preset HMD generation switch (used only for generating HMD data)

0:OFF

1:ON

[Shading]

A single character indicating the shading mode.

"F": Flat

"G": Gouraud (smooth)

For flat shading, shading is based on the normal of the first vertex specified in the PLY file.

[Material information]

The format of this section depends on the material type, such as whether there are textures, and so on.

Figure 2-14: No texture (colored polygons/lines)

TYPE	R	G	B
------	---	---	---

TYPE: Material type, value is "C"

R, G, B: Polygon color, RGB component (0-255)

Figure 2-15: No texture (Gouraud-colored polygons/lines)

TYPE	R0	G0	B0	R1	G1	B1	...	R3	G3	B3
------	----	----	----	----	----	----	-----	----	----	----

TYPE: Material type, value is "G"

Rn, Gn, Bn: RGB component of the nth vertex (n = 0-3)
(4th vertex is 0,0,0 for triangles)

Figure 2-16: Textured polygons/sprites

TYPE	TNO	U0	V0	U1	V1	U2	V2	U3	V3
------	-----	----	----	----	----	----	----	----	----

TYPE: Material type, value is "T"

TNO: Specifies the TIM data file to use
(The texture number of the descriptor in the RSD file)

Un, Vn: Location of the texture space for vertex n.
The values of the 4th vertex (U3, V3) are (0,0) for a triangle.

Figure 2-17: Polygons with colored textures

TYPE	TNO	U0	V0	U1	V1	U2	V2	U3	V3	R	G	B
------	-----	----	----	----	----	----	----	----	----	---	---	---

TYPE: Material type, value is "D"
 TNO: Specifies the TIM data file to use
 (The texture number of the descriptor in the RSD file)
 Un, Vn: Location of the texture space for vertex n.
 The values of the 4th vertex (U3, V3) are (0,0) for a triangle.
 R, G, B: Polygon color, RGB component (0-255)

* Polygons with colored textures are used to apply brightness to individual polygon textures, without the use of light-source calculations. This allows a textured polygon to be rendered three-dimensionally without performing any light-source calculation. The colored-texture material type is valid only when the light-source calculation mode is set to fixed color.

Figure 2-18: Polygons with gradation texture

TYPE	TNO	U0	V0	U1	V1	U2	V2	U3	V3
------	-----	----	----	----	----	----	----	----	----

R0	G0	B0	R1	G1	B1	...	R3	G3	B3
----	----	----	----	----	----	-----	----	----	----

TYPE: Material type, value is "H"
 TNO: Specifies the TIM data file to use
 (The texture number of the descriptor in the RSD file)
 Un, Vn: Location of the texture space for vertex n.
 The values of the 4th vertex (U3, V3) are (0,0) for a triangle.
 Rn, Gn, Bn: RGB component of vertex n (n = 0-3)
 For triangles, the RGB value of the 4th vertex is 0,0,0.

* Polygons with gradation textures provide the same effect as textured Gouraud-shading but without using light-source calculations. The gradation-texture material type is valid only when the light-source calculation mode is set to fixed color.

Figure 2-19: Polygons/sprites with repeating textures

TYPE	TNO	TUM	TVM	TUA	TVA	U0	V0	U1	V1	U2	V2	U3	V3
------	-----	-----	-----	-----	-----	----	----	----	----	----	----	----	----

TYPE: Material type, value is "W"
 TNO: Specifies the TIM data file to use
 (The texture number of the descriptor in the RSD file)
 TUM,TVM: UV coordinate of the repeating mask of the texture pattern.
 TUA,TVA: UV upper address of the repeating texture pattern.
 Un, Vn: Location of the texture space for vertex n.
 The values of the 4th vertex (U3, V3) are (0,0) for a triangle.

Figure 2-20: Polygons with repeating colored textures

TYPE	TNO	TUM	TVM	TUA	TVA	U0	V0	U1	V1	U2	V2	U3	V3
R	G	B											

TYPE: Material type, value is "S"
 TNO: Specifies the TIM data file to use
 (The texture number of the descriptor in the RSD file)
 TUM,TVM: UV coordinate of the repeating mask of the texture pattern.
 TUA,TVA: UV upper address of the repeating texture pattern.
 Un, Vn: Location of the texture space for vertex n.
 The values of the 4th vertex (U3,V3) are (0,0) for a triangle.
 R, G, B: Polygon color, RGB component (0-255)

* Polygons with repeating colored textures are used to apply brightness to individual polygon textures without using light-source calculations. This allows a textured polygon to be rendered three-dimensionally without performing any light-source calculation. The repeating-colored-texture material type is valid only when the light-source calculation mode is set to fixed color.

Figure 2-21: Polygons with repeating gradation textures

TYPE	TNO	TUM	TVM	TUA	TVA	U0	V0	U1	V1	U2	V2	U3	V3
R0	G0	B0	R1	G1	B1	...	R3	G3	B3				

TYPE: Material type, value is "N"
 TNO: Specifies the TIM data file to use
 (The texture number of the descriptor in the RSD file)
 TUM,TVM: UV coordinate of the repeating mask of the texture pattern.
 TUA,TVA: UV upper address of the repeating texture pattern.
 Un, Vn: Location of the texture space for vertex n.
 The values of the 4th vertex (U3, V3) are (0,0) for a triangle.
 R, G, B: Polygon color, RGB component (0-255)

* Polygons with gradation textures provide the same effect as textured Gouraud shading but without using light-source calculations. The repeating-gradation texture material type is valid only when the light-source calculation mode is set to fixed color.

Sample file

The following is an example of a simple MAT file.

Figure 2-22: Sample MAT file

```

@MAT970401
# Number of Items
10
# Materials
0-5      0 F C 255 255 255
6        0 G T 1 10 0 25 71 40 25 0 0
7        0 G T 1 10 30 20 75 40 25 0 0
8        0 G T 1 18 73 30 79 40 25 0 0
9        0 G T 1 12 23 29 77 40 25 0 0
10       0 F T 1 18 13 75 72 40 25 0 0
11       0 F T 0 22 10 24 74 40 25 0 0
12       0 F T 0 30 39 41 79 40 25 0 0
13       1 F D 0 116 47 118 77 69 46 69 77 30 187 187
14       1 F H 0 69 46 69 77 17 45 15 77 101 210 138 52 211 188 101 210
  
```

GRP File

The GRP file allows a name to be assigned to all of the polygons in a PLY file. With a group name, multiple polygons can be easily selected with the material editor.

Overall structure

Figure 2-23: Overall structure of a GRP file

ID
GROUP descriptor count
GROUP descriptor : :

ID

The ID is a string of the form "@GRPnnnnnn" (where nnnnnn is a number) indicating the GRP file format version number. For example, version 3.0 would be specified with the string "@GRP970401".

GROUP descriptor count

The number of GROUP descriptors which follow is specified in this field.

GROUP descriptor

A GROUP descriptor defines the structure of a group. A GROUP descriptor consists of two or more lines as indicated below.

[First line]

Figure 2-24: GROUP descriptor for GRP file

Group name	Polygon No. line count	No. of polygons
------------	------------------------	-----------------

Group name : Name of the associated group.
 Polygon No. line count : Number of following lines that contain the
 descriptor associated with this polygon no.
 No. of polygons : Number of polygons belonging to this group.

[Following lines (Polygon Nos.)]

Specifies the polygon numbers that belong to the group. The values represent the polygon index number within the PLY file. Multiple polygons can be included on a single line if a range is specified.

Figure 2-25: Following Lines

Values	Specified Polygons
1	1 only
3-7	3 4 5 6 7
2,4,6	2 4 6

Sample file

A simple example of a GRP file is shown below.

Figure 2-26: Sample GRP file

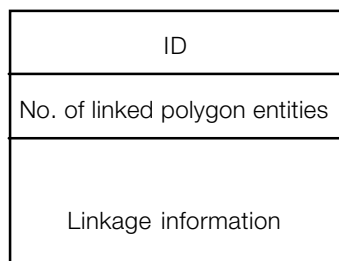
```
@GRP970401
# Number of Group
2
# Group list
upper_part 2 5
10-13
25
lower_part 3 3
3
5
7
```

MSH File

The MSH file is an extended RSD file that contains linkage information. The MSH file permits HMD data with linkage information to be generated from RSD-formatted data.

Overall structure

Figure 2-27: Overall structure of a MSH file

**ID**

The ID is a string of the form "@MSHnnnnnn" (where nnnnnn is a number) indicating the MSH file format version number. For example, version 1.0 would be specified with the string "@MSH970401".

Number of linked polygon entities

The number of linked polygon entities which follow (number of mesh groups) is specified in this field.

Linkage information

This field defines the number and sequence of links in a linked polygon entity. The link sequence is defined by the order of the polygon indexes specified in the PLY file. Line breaks are optional.

No. of links	Link sequence (order of polygon indexes)...
--------------	---

Sample file

A simple example of a MSH file is shown below.

Figure 2-28: Sample MSH file

```
#ID
@MSH970401
#Number of Mesh Grupe
5
#Mesh Information
5      0 1 2 3 4
10     5 6 7 8 9 10 11 12 13 14
5      15 16 17 18 19
1      20
1      21
```

PVT File

The PVT file is an extended RSD file that contains offsets for the centers of rotation. This information is used when VERTEX values are rewritten using the coordinate system of the extended PlayStation library (libgs).

Overall structure

Figure 2-29: Overall structure of a PVT file

ID
Number of elements
COORDINATE INDEX & offset

ID

The ID is a string of the form "@PVTnnnnnn" (where nnnnnn is a number) indicating the PVT file format version number. For example, version 1.0 would be specified with the string "@PVT970401".

Number of elements

This field specifies the number of COORDINATE INDEXes and offsets which follow.

COORDINATE INDEX & offset

This field contains the offset value of the specified COORDINATE index.

COORDINATE INDEX	X	Y	Z
------------------	---	---	---

Sample file

A simple example of a PVT file is shown below.

Figure 2-30: Sample PVT file

```
#ID
@PVT970401
#Number of Items
5
#Pivot Information
0      100 -100 100
1      50  50  50
3      0   0 -100
4      200 1000 -1000
5      100 100 100
```

COD File

The COD file is an extended RSD file that allows COORDINATE attributes to be applied to VERTEXes in a PLY file. The COD file permits HMD data containing multiple MATRIXes to be generated from RSD-formatted data.

Overall structure

Figure 2-31: Overall structure of a COD file

ID
OGP Filename
No. of COORDINATEs
COORDINATE
Object group or VERTEX which belongs to the COORDINATE

ID

The ID is a string of the form "@CODnnnnnn" (where nnnnnn is a number) indicating the COD file format version number. For example, version 1.0 would be specified with the string "@COD970401".

OGP Filename

This field specifies the filename of an OGP file. The format is OGP=OGP filename. Specifying OGP=NULL means that there is no OGP file.

Number of COORDINATEs

This field specifies the number of COORDINATEs which follow.

COORDINATE

This field describes the COORDINATE structure. The COORDINATE structure is defined across multiple lines in the file as shown below.

[Lines 1 - 3]

These lines contain the matrix coefficients. Values are represented as decimal numbers.

```
m[0][0]  m[0][1]  m[0][2]
m[1][0]  m[1][1]  m[1][2]
m[2][0]  m[2][1]  m[2][2]
```

[Line 4]

This line specifies the amount of translation. Values are represented as decimal numbers.

```
t[0]  t[1]  t[2]
```

[Line 5]

This line specifies the elements of the rotation vector that are used to generate the matrix coefficients. Values are represented as decimal numbers.

```
vx  vy  vz
```


[Line 6]

This line specifies an index for the parent COORDINATE. Indexes are entered in order beginning with 0. If the parent is the world coordinate system, the index refers to itself.

super

Object group or VERTEX which belongs to the COORDINATE

When an OGP file is specified in the object name section, an object group which belongs to the COORDINATE is specified here.

When there is no OGP file specified in the object name section, a VERTEX which belongs to the COORDINATE is specified here.

When object groups are specified

Object groups are represented as object group names specified within the OGP file. Multiple object groups can be represented in a list by specifying the number of object groups. Line breaks are optional.

COORDINATE index	No. of object groups	Object group name...
------------------	----------------------	----------------------

When VERTEXes are specified

VERTEXes are represented as index numbers in the PLY file. Multiple VERTEXes can be represented in a list by specifying the number of VERTEXes. Line breaks are optional.

COORDINATE index	No. of VERTEXes	Index of VERTEX...
------------------	-----------------	--------------------

Note: The two methods described above cannot be mixed in a single COD file. If an OGP file is specified, only object groups can be included. Otherwise, only VERTEXes can be included.

Sample files

Simple examples of COD files are shown below.

[Object file specified]

Figure 2-32: Sample COD files

```
#ID
@COD970401
#OGP File
OGP=test.ogp
#Number of COORDINATE
2
#COORDINATE
4096 0 0
0 4096 0
0 0 4096
0 0 0
0 0 0
0
-4096 0 0
0 -4096 0
0 0 -4096
0 0 0
0 0 0
0
#COORDINATE of Object Grupe
0 1    body
1 2    hand head
```

[Object file not specified]

```
#ID
@COD970401
#OGP File
OGP=NULL
#Number of COORDINATE
2
#COORDINATE
4096 0 0
0 4096 0
0 0 4096
0 0 0
0 0 0
0
-4096 0 0
0 -4096 0
0 0 -4096
0 0 0
0 0 0
0
#COORDINATE of Object Grupe
0 5      0 1 2 3 4
1 5      5-9
```

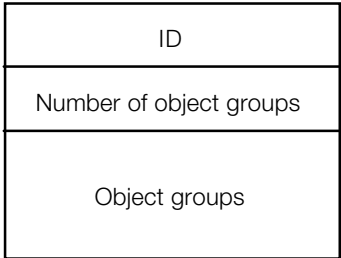
Spaces or '-' are used as delimiters.

OGP File

The OGP file is a sub-extended RSD file that allows a group of VERTEXes to be defined as a single object group. The OGP file makes it easier to manage attributes for an entire object group.

Overall structure

Figure 2-33: Overall structure of an OGP file



ID

The ID is a string of the form "@OGPnnnnnn" (where nnnnnn is a number) indicating the OGP file format version number. For example, version 1.0 would be specified with the string "@OGP970401".

Number of object groups

This field specifies the number of object groups which follow.

Object groups

This field describes the object group structures. A VERTEX which belongs to an object group is represented by its index which is specified in the PLY file. Multiple VERTEXes can be included in a list by specifying the number of VERTEXes. Line breaks are optional.

Object group name	No.of VERTEXes	Index of VERTEX...
-------------------	----------------	--------------------

Sample file

A simple example of an OGP file is shown below.

Figure 2-34: Sample OGP file

```
#ID
@OGP970401
#Number of Object Grupe
2
#Object Grupe
head 10    0 1 2 3 4 5 6 7 8 9
body 10    10-19
```

Spaces or '-' are used as delimiters.

MOT File

A MOT file is an extended RSD file that contains animation information.

Overall structure

Figure 2-35: Overall structure of a MOT file

ID
Animation type
INDEX count
INDEX & sequence count & WORK TOP
Sequence header information
Sequence control information
Parameter information

ID

The ID is a string of the form "@MOTnnnnnn" (where nnnnnn is a number) indicating the MOT file format version number. For example, version 1.0 would be specified with the string "@MOT970401".

Animation type

This field indicates the type of animation. Currently, the only supported type is COORDINATE transformation.

(When COORDINATE is used)

COORDINATE

INDEX count

This field specifies the number of sets of INDEX, sequence count, and WORK TOP (described below). Each set together with the sequence header information is repeated INDEX count times.

INDEX count

INDEX & sequence count & WORK TOP

This information specifies the index of the original data which is to be transformed, the number of sequence groups, and the name of the TOP point for the WORK sequence.

For example, if the animation type is COORDINATE, then the original data index refers to the index of COORDINATE in the COD file and the sequence group count refers to the number of animation patterns (described below). If a WORK sequence does not exist, a string that has not been used for a sequence point name is assigned to WORK TOP.

INDEX	Sequence count	WORK TOP
-------	----------------	----------

Sequence header information

This field contains header information for an animation pattern (described below).

The following is a description of the reserved words.

START	Start point name	Stream ID
-------	------------------	-----------

START

Reserved word indicating the start point for an animation sequence.

Start point name

Describes the start point name indicated in the sequence control information section. The start point name is a string with a maximum length of 256.

Stream ID

Represented as a hexadecimal value in 0x?? (7-bit) notation. The stream ID is a bit pattern that is used when comparing with another sequence factor.

KEY2	SEQ TOP1	SEQ TOP2
------	----------	----------

KEY2

Reserved word indicating the top point of a key frame. Up to two KEY2s can be specified.

SEQ TOP1,2

Indicates the top point of the key frame in the sequence control information section. TOP1 is used in the lower 16 bits in the sequence header data when converting to HMD. TOP2 is used in the upper 16 bits. If a specified point does not exist, a string that has not been used as a point name is assigned. Point names are expressed as strings of up to 256 characters.

Sequence control information

This field describes the actual animation motion. The contents depend on the type of reserved word.

A start point of an arbitrary string of 256 characters or less can be inserted as positional information. The start point indicates the animation position beginning with the next line.

The following is a description of the reserved words.

SEQ	Frame count	Animation packet type	Animation packet type index
-----	-------------	-----------------------	-----------------------------

SEQ

Reserved word which represents a sequence.

Frame count

Number of frames up to this key frame.

Animation packet type

Type of interpolation used in animation. (A detailed description can be found in the section on parameter data)

Animation packet type index

Index of the animation packet in the animation packet section.

GOTO	Jump point name	Stream ID (after jump)	Stream ID (before jump)
------	-----------------	---------------------------	----------------------------

GOTO

Reserved word which allows branching within a sequence flow.

Jump point name

Target sequence position to which to jump to, represented by a string of 256 characters or less.

Stream ID

Represented as a hexadecimal value in 0x?? (7-bit) notation. The stream ID is a bit pattern that is used when comparing with another sequence factor.

END	Stream ID
-----	-----------

END

Reserved word which ends a sequence flow.

Stream ID

Represented as a hexadecimal value in 0x?? (7-bit) notation. The stream ID is a bit pattern that is used when comparing with another sequence factor.

Parameter information

This information describes the value of parameters such as the translation amount. The packet type for the parameter is the same as that defined for HMD. The packet type is given following the TYPE keyword. A sample packet type is shown below.

TYPE ps0r0t0

p Indicates parameter. This is the only type currently available.

s?r?t? The packet type for the parameter varies according to the '?'. If 0, that parameter is assumed to be absent.

12 packet types are currently supported. Individual packet types are shown below.

Scale Ratio: 1 = 4096

Rotation Unit: Degree, 360 degrees = 4096

(ps0r0t0) Dummy Matrix				Dx	Dy			
(ps0r0t1) Translation Linear								
			Tx	Ty	Tz			
(ps0r0t9) Translation(short) Linear								
			Tx	Ty	Tz			
(ps9r0t9) Scale(one) Translation(short) Linear								
			Tx	Ty	Tz	Scale		
(ps1r0t0) Scale Linear								
			Sx	Sy	Sz			
(ps0r1t0) Rotation linear								
			Rx	Ry	Rz			
(ps9r1t0) Scale(one) Rotation Linear								
			Rx	Ry	Rz	Scale		
(ps0r1t1) Rotation Translation Linear								
			Tx	Ty	Tz	Rx	Ry	Rz
(ps9r1t1) Scale(one) Rotation Translation Linear								
			Tx	Ty	Tz	Rx	Ry	Rz
								Scale
(ps1r1t1) Scale Rotation Translation Linear								
			Tx	Ty	Tz	Rx	Ry	Rz
						Sx	Sy	Sz
(ps0r1t9) Rotation Translation(short) Linear								
			Tx	Ty	Tz	Rx	Ry	Rz
(ps1r1t9) Scale Rotation Translation(short) Linear								
			Tx	Ty	Tz	Rx	Ry	Rz
						Sx	Sy	Sz

Sample file

A simple example of an MOT file is shown below.

Figure 2-36: Sample MOT file

```
#ID
@MOT970401
#Animation Type
COORDINATE
#Number of Index
2
#Index, Number of Sequence
0 4 WORK
```

```

#Sequence Header
START TOP11 0x01
KEY2 TOP11A TOP11B
START TOP12 0x01
START TOP13 0x01
#Index, Number of Sequence
0 3 DUMMY
#Sequence Header
START TOP21 0x01
START TOP22 0x01
START TOP23 0x01
#Sequence
WORK
SEQ 0 ps0r0t0 0
SEQ 0 ps0r0t0 1
TOP11
SEQ 250 ps0r0t1 0
TOP11A
SEQ 250 ps0r0t1 1
TOP11B
GOTO TOP11 0x02 0x01
END 0x02
TOP12
SEQ 250 ps0r0t1 2
SEQ 250 ps0r0t1 3
GOTO TOP12 0x02 0x01
END 0x02
TOP13
SEQ 250 ps0r0t1 4
SEQ 250 ps0r0t1 5
GOTO TOP13 0x02 0x01
END 0x02
TOP21
SEQ 250 ps0r1t0 0
SEQ 250 ps0r1t0 1
GOTO TOP21 0x02 0x01
END 0x02
TOP22
SEQ 250 ps0r1t0 2
SEQ 250 ps0r1t0 3
GOTO TOP22 0x02 0x01
END 0x02
TOP23
SEQ 250 ps0r1t0 4
SEQ 250 ps0r1t0 5
GOTO TOP23 0x02 0x01
END 0x02
#Animation Packet Type
TYPE ps0r0t0
0 0
0 0
TYPE ps0r0t1
-400 -400 2000
400 400 2000
-50 -50 1500
50 50 1500
100 100 1000
-100 -100 1000
TYPE ps0r1t0
0 0 0
4096 0 0
0 0 0
0 4096 0
0 0 0
0 0 4096

```

TMD: Modeling Data for OS Library

The TMD format contains 3D modeling data which is compatible with the PlayStation expanded graphics library (libgs). TMD data is downloaded to memory and may be passed as an argument to functions provided by LIBGS. TMD files are created using the RSDLINK utility, which reads an RSD file created by the SCE 3D Graphics Tool or a comparable program.

The data in a TMD file is a set of graphics primitives—polygons, lines, etc.—that make up a 3D object. A single TMD file can contain data for one or more 3D objects.

Coordinate Values

Coordinate values in the TMD file follow the 3D coordinate space handled by the 3D graphics library. The positive direction of the X axis represents the right, the Y axis the bottom, and the Z axis the depth. The spatial coordinate value of each object is a signed 16-bit integer value ranging from -32768 to +32767.

In the 3D object design phase and within the RSD format, the vertex information is stored as a floating point value. Conversion from RSD into TMD involves converting and scaling vertex values as needed. The scale used is reflected in the object structure, described later, as the reference value. This value can provide an index for mapping from object to world coordinates. The current version of LIBGS ignores the scale value.

File Format

TMD files are configured by 4 blocks. They have 3 dimensional object tables, and 3 types of data entities—PRIMITIVE, VERTEX, and NORMAL—which configure these.

Figure 2-37: TMD File Format

HEADER
OBJ TABLE
PRIMITIVE
VERTEX
NORMAL

Header

The header section is composed of three word (12 bytes) data carrying information on data structure.

Figure 2-38: Structure of Header

ID
FLAGS
NOBJ

ID: Data having 32 bits (one word). Indicates the version of a TMD file. The current version is 0x00000041.

FLAGS: Data having 32 bits (one word). Carries information on TIM data configuration. The least significant bit is FIXP. The other bits are reserved and their values are all zero. The FIXP bit indicates whether the pointer value of the OBJECT structure described later is a real address. A value of one means a real address. A value of zero indicates the offset from the start.

NOBJ: Integral value indicating the number of objects

Obj Table

The OBJ TABLE block is a table of structures holding pointer information indicating where the substance of each object is stored. Its structure is as shown below.

Figure 2-39: OBJ TABLE structure

OBJECT #1
OBJECT #2
:
:

The object structure has the following configuration:

```
struct object
{
    u_long *vert_top;
    u_long n_vert;
    u_long *normal_top;
    u_long n_normal;
    u_long *primitive_top;
    u_long n_primitive;
    long scale;
}
```

(Explanation of members)

vert_top: Start address of a vertex
n_vert: Number of vertices
normal_top: Start address of a normal
n_normal: Number of normals
primitive_top: Start address of a primitive
n_primitive: Number of primitives

Among the members of the structure, the meanings of the pointer values (*vert_top*, *normal_top*, *primitive_top*) change according to the value of the FIXP bit in the HEADER section. If the FIXP bit is 1, they indicate the actual address, and if the FIXP bit is 0, they indicate a relative address taking the top of the OBJECT block as the 0 address.

The type of the scaling factor is "signed long", and its value raised to the second power is the scale value. That is to say, if the scaling factor is 0, the scale value is an equimultiple; if the scaling factor is 2, the scale value is 4; if the scaling factor is -1, the scale value is 1/2. Using this value, it is possible to return to the scale value at the time of design.

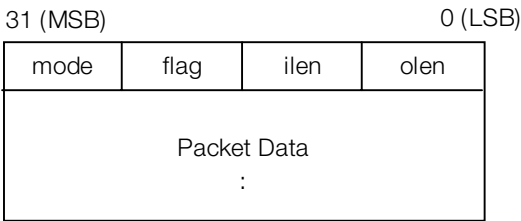
Primitive

The PRIMITIVE section is an arrangement of the drawing packets of the structural elements (primitives) of the object. One packet stands for one primitive (see Figure below).

The primitives defined in TMD are different from the drawing primitives handled by libgpu. A TMD primitive is converted to a drawing primitive by undergoing perspective transformation processing performed by the libgs functions.

Each packet is of variable length, and its size and structure vary according to the primitive type.

Figure 2-40: Drawing Packet General Structure

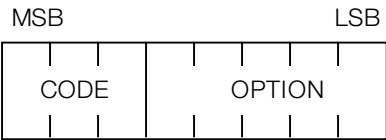


Each item in the figure above is as follows:

Mode (8 bit)

Mode indicates the type of primitive and added attributes. They have the following bit structure:

Figure 2-41: Mode



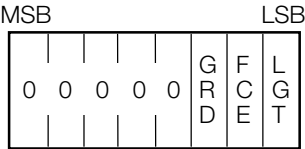
CODE: 3 bit code expressing entities
001 = Polygon (triangle, quadrilateral)
010 = Straight line
011 = Sprite

OPTION: Varies with the option, bit and CODE values
(Listed with the list of packet data configurations described later)

Flag (8 bit)

Flag indicates option information when rendering and has the following bit configuration:

Figure 2-42: Flag



GRD: Valid only for the polygon not textured, subjected to light source calculation
1: Gradation polygon
0: Single-color polygon

- FCE: 1: Double-faced polygon
 0: Single-faced polygon
 (Valid, only when the CODE value refers to a polygon.)
- LGT: 1: Light source calculation not carried out
 0: Light source calculation carried out

llen (8 bit)

Indicates the length, in words, of the packet data section.

Olen (8 bit)

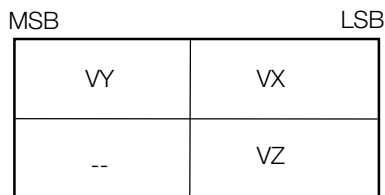
Indicates the word length of the 2D drawing primitives that are generated by intermediate processing.

Packet Data

Parameters for vertices and normals. Content varies depending on type of primitive. Please refer to “Packet data configuration” which will be discussed later.

Vertex

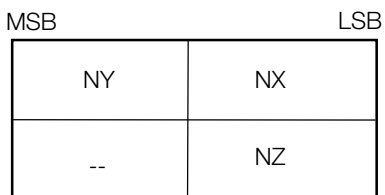
The vertex section is composed of a set of structures representing vertices. The following gives the format of one structure.

Figure 2-43: Vertex Structure

VX, VY, XZ: x, y and z values of vertex coordinates (16-bit integer)

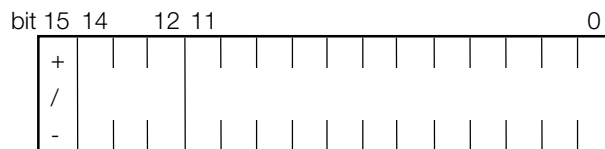
Normal

The normal section is composed of a set of structures representing normals. The following gives the format of one structure.

Figure 2-44: Normal Structure

NX, NY, NZ: x, y and z components of a normal (16-bit fixed-point value)

NX, NY and NZ values are signed 16-bit fixed-point values where 4096 is considered to be 1.0.

Figure 2-45: Fixed-Point Format

Sign: 1 bit

Integral part: 3 bits

Decimal part: 12 bits

Packet Data Composition Table

This section lists packet data configurations for each primitive type.
The following parameters are contained in the packet data section:

Vertex(n)
Index value of 16-bit length pointing to a vertex. Indicates the position of the element from the start of the vertex section for an object covering the polygon.

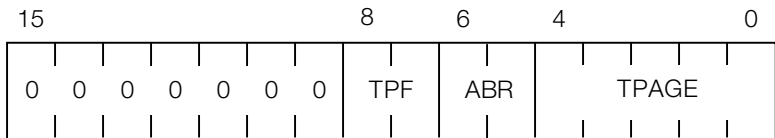
Normal(n)
Index value of 16-bit length pointing to a normal. Same as Vertex.

Un, Vn
X and Y coordinate values on the texture source space for each vertex

Rn, Gn, Bn
RGB value representing polygon color being an unsigned 8-bit integer. Without light source calculation, the predetermined brightness value must be entered.

TSB
Carries information on a texture/sprite pattern.

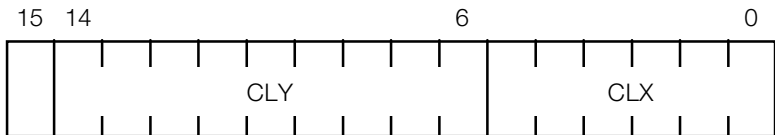
Figure 2-46: TSB



- TPAGE: Texture page number (0 to 31)
- ABR: Semi-transparency rate (Mixture rate).
Valid, only when ABE is 1.
00 50%back + 50%polygon
01 100%back + 100%polygon
10 100%back - 100%polygon
11 100%back + 25%polygon
- TPF: Color mode
00 4 bit
01 8 bit
10 15 bit

CBA
Indicates the position where CLUT is stored in the VRAM.

Figure 2-47: CBA



- CLX: Upper six bits of 10 bits of X coordinate value for CLUT on the VRAM
- CLY: Nine bits of Y coordinate value for CLUT on the VRAM

3 Vertex Polygon with Light Source Calculation

Bit Configuration of Mode Value

The primitive section mode value bit configuration is shown below. For the value of each bit please refer to “3 vertex polygon with light source calculation.”

Figure 2-48: Mode Value of 3 Vertex Polygon with Light Source Calculation

MSB				LSB			
0	0	1	IIP	0	TME	ABE	TGE

- IIP: Shading mode
 0: Flat shading
 1: Gouraud shading
- TME: Texture specification
 0: Off
 1: On
- ABE: Translucency processing
 0: Off
 1: On
- TGE: Brightness calculation at time of texture mapping
 0: On
 1: Off (Draws texture as is)

Packet Configuration**Figure 2-49: Packet Configuration of 3 Vertex Polygon with Light Source Calculation**

Flat, No-Texture (solid)				Gouraud, No-Texture (solid)			
0x20	0x00	0x03	0x04	0x30	0x00	0x04	0x06
0x20*	B	G	R	0x30*	B	G	R
Vertex 0		Normal 0		Vertex 0		Normal 0	
Vertex 2		Vertex 1		Vertex 1		Normal 1	
				Vertex 2		Normal 2	

Flat, No-Texture (gradation)				Gouraud, No-Texture (gradation)			
0x20	0x04	0x05	0x06	0x30	0x04	0x06	0x06
0x20*	B0	G0	R0	0x30*	B0	G0	R0
—	B1	G1	R1	—	B1	G1	R1
—	B2	G2	R2	—	B2	G2	R2
Vertex 0		Normal 0		Vertex 0		Normal 0	
Vertex 2		Vertex 1		Vertex 1		Normal 1	
				Vertex 2		Normal 2	

Flat, Texture				Gouraud, Texture			
0x24	0x00	0x05	0x07	0x34	0x00	0x06	0x09
CBA		V0	U0	CBA		V0	U0
TSB		V1	U1	TSB		V1	U1
—	—	V2	U2	—	—	V2	U2
Vertex 0		Normal 0		Vertex 0		Normal 0	
Vertex 2		Vertex 1		Vertex 1		Normal 1	
				Vertex 2		Normal 2	

* same value as mode

In the above example, the values of mode and flag indicate a single-faced polygon and semi-transparency processing not carried out.

4 Vertex polygon with Light Source Calculation**Bit Configuration of Mode Value**

The primitive section mode value bit configuration is shown below. For the value of each bit please refer to “3 vertex polygon with light source calculation.”

Figure 2-50: Mode Value of 4 Vertex Polygon with Light Source Calculation

MSB				LSB			
0	0	1	11D	1	TME	ABE	TGE

(bit 3 is set to designate a 4-vertex primitive)

Packet Configuration**Figure 2-51: Packet Configuration for 4 Vertex Polygon with Light Source Calculation**

Flat, No-Texture (solid)				Gouraud, No-Texture (solid)			
0x28	0x00	0x04	0x05	0x38	0x00	0x05	0x08
0x28*	B	G	R	0x38*	B	G	R
Vertex 0		Normal 0		Vertex 0		Normal 0	
Vertex 2		Vertex 1		Vertex 1		Normal 1	
—		Vertex 3		Vertex 2		Normal 2	
				Vertex 3		Normal 3	

Flat, No-Texture (gradation)				Gouraud, No-Texture (gradation)			
0x28	0x04	0x07	0x08	0x38	0x04	0x08	0x08
0x28*	B0	G0	R0	0x38*	B0	G0	R0
—	B1	G1	R1	—	B1	G1	R1
—	B2	G2	R2	—	B2	G2	R2
—	B3	G3	R3	—	B3	G3	R3
Vertex 0		Normal 0		Vertex 0		Normal 0	
Vertex 2		Vertex 1		Vertex 1		Normal 1	
—		Vertex 3		Vertex 2		Normal 2	
				Vertex 3		Normal 3	

Flat, Texture				Gouraud, Texture			
0x2c	0x00	0x07	0x09	0x3c	0x00	0x08	0x0c
CBA		V0	U0	CBA		V0	U0
TSB		V1	U1	TSB		V1	U1
—	—	V2	U2	—	—	V2	U2
—	—	V3	U3	—	—	V3	U3
Vertex 0		Normal 0		Vertex 0		Normal 0	
Vertex 2		Vertex 1		Vertex 1		Normal 1	
—		Vertex 3		Vertex 2		Normal 2	
				Vertex 3		Normal 3	

* same value as mode

3 Vertex Polygon with No Light Source Calculation**Bit Configuration of Mode Value**

The primitive section mode value bit configuration is shown below. For the value of each bit please refer to “3 vertex polygon with light source calculation.”

Figure 2-52: Mode Value of 3 Vertex Polygon with No Light Source Calculation

MSB				LSB			
0	0	1	IIP	0	TME	ABE	TGE

(bit 3 is set to designate a 4-vertex primitive)

Packet Configuration

Figure 2-53: Packet configuration for 3 Vertex Polygon with No Light Source Calculation

Flat, No-Texture				Gradation, No-Texture			
0x21	0x01	0x03	0x04	0x31	0x01	0x05	0x06
0x21*	B	G	R	0x31*	B0	G0	R0
Vertex 1		Vertex 0		—	B1	G1	R1
—		Vertex 2		—	B2	G2	R2
Vertex 1		Vertex 0		Vertex 1		Vertex 0	
—		Vertex 2		—		Vertex 2	

Flat, Texture				Gradation, Texture			
0x25	0x01	0x06	0x07	0x35	0x01	0x08	0x09
CBA		V0	U0	CBA		V0	U0
TSB		V1	U1	TSB		V1	U1
—	—	V2	U2	—	—	V2	U2
—	B	G	R	—	B0	G0	R0
Vertex 1		Vertex 0		—	B1	G1	R1
—		Vertex 2		—	B2	G2	R2
Vertex 1		Vertex 0		Vertex 1		Vertex 0	
—		Vertex 2		—		Vertex 2	

* same value as mode

4 Vertex Polygon with No Light Source Calculation

Bit Configuration of Mode Value

The primitive section mode value bit configuration is shown below. For the value of each bit please refer to “3 vertex polygon with light source calculation.”

Figure 2-54: Mode Value of 4 Vertex Polygon with No Light Source Calculation

MSB				LSB			
0	0	1	IIP	1	TME	ABE	TGE

(bit 3 is set to designate a 4-vertex primitive)

Packet Configuration**Figure 2-55: Packet Configuration for 4 Vertex Polygon with No Light Source Calculation**

Flat, No-Texture				Gradation, No-Texture			
0x29	0x01	0x03	0x05	0x39	0x01	0x06	0x08
0x29*	B	G	R	0x39*	B0	G0	R0
Vertex 1		Vertex 0		—	B1	G1	R1
Vertex 3		Vertex 2		—	B2	G2	R2
				—	B3	G3	R3
				Vertex 1		Vertex 0	
				Vertex 3		Vertex 2	

Flat, Texture				Gradation, Texture			
0x2d	0x01	0x07	0x09	0x3d	0x01	0x0a	0x0c
CBA		V0	U0	CBA		V0	U0
TSB		V1	U1	TSB		V1	U1
—	—	V2	U2	—	—	V2	U2
—	—	V3	U3	—	—	V3	U3
—	B	G	R	—	B0	G0	R0
Vertex 1		Vertex 0		—	B1	G1	R1
Vertex 3		Vertex 2		—	B2	G2	R2
				—	B3	G3	R3
				Vertex 1		Vertex 0	
				Vertex 3		Vertex 2	

* same value as mode

Straight Line**Bit Configuration of Mode Value**

The primitive section mode value bit configuration is as follows:

Figure 2-56: Mode Value of Straight Line

MSB				LSB			
0	1	0	IIP	0	0	ABE	0

IIP: With or without gradation
 0: Gradation off (Monochrome)
 1: Gradation on

ABE: Translucency processing on/off
 0: off
 1: on

Packet Configuration

Figure 2-57: Packet Configuration for “Straight Line”

Gradation OFF				Gradation ON			
0x40	0x01	0x02	0x03	0x50	0x01	0x03	0x04
0x40*	B	G	R	0x50*	B0	G0	R0
Vertex 1		Vertex 0		—	B1	G1	R1
Vertex 1		Vertex 0		Vertex 1		Vertex 0	

* same value as mode

3 Dimensional Sprite

A 3 dimensional sprite is a sprite with 3-D coordinates and the drawing content is the same as a normal sprite.

Bit Configuration of Mode Value

Figure 2-58: Mode Value of 3D Sprite

MSB				LSB			
0	1	1	SIZ	1	ABE	0	

- SIZ: Sprite size
00: Free size (Specified by W, H)
01: 1 x 1
10: 8 x 8
11: 16 x 16
- ABE: Translucency processing
0: Off
1: On

Packet Data Configuration

Figure 2-59: Packet Configuration for Sprites

Free size

0x64	0x01	0x03	0x05
TSB		Vertex 0	
CBA		V0	U0
H		W	

8x8

0x74	0x01	0x02	0x04
TSB		Vertex 0	
CBA		V0	U0

1X1

0x6c	0x01	0x02	0x04
TSB		Vertex 0	
CBA		V0	U0

16X16

0x7c	0x01	0x02	0x04
TSB		Vertex 0	
CBA		V0	U0

PMD: High-Speed Modeling Data

The PMD format is used for modeling data supported by the extended graphics library (libgs). The PMD format has a narrower range of functions than the TMD format, but this smaller scope enables faster processing.

PMD format handles the following kinds of objects.

- Triangular and rectangular polygons only
- Packet creation areas contained in the data
- Groups of polygons having the same attributes

The PMD file format consists of a table of 3D objects along with their PRIMITIVE and VERTEX descriptions.

Figure 2-60: Overall structure of PMD files

ID
PRIM POINT
VERT POINT
OBJ TABLE
:
PRIMITIVE Gp.
:
VERTEX Gp.
:

ID:	32-bit word containing the version of the PMD file. For the current version, this is 0x00000042.
PRIM POINT:	A 32-bit integer indicating the offset from the start of the PRIMITIVE Gp section of the file.
VERT POINT:	A 32-bit integer indicating the offset from the start of the VERTEX Gp section of the file. Enter "0" for an independent vertex.
OBJ TABLE:	The array of objects.
PRIMITIVE Gp:	A collection (primitive group) of polygons having the same attributes.
VERTEX Gp:	An array of vertex coordinates. VERTEX groups exist only in the case of shared vertices.

Coordinate Values

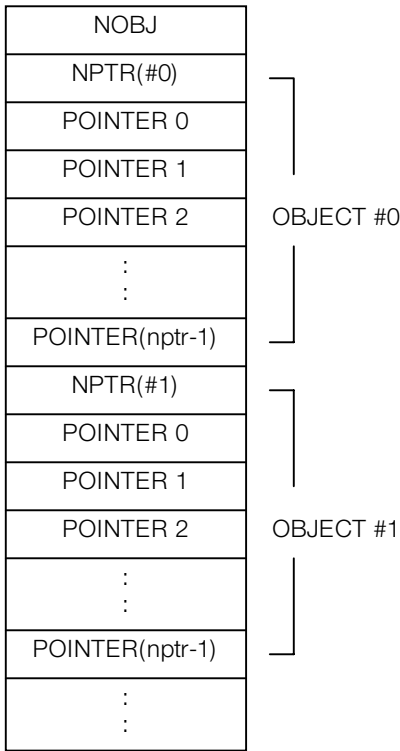
Coordinate values in the PMD file follow the 3D coordinate space handled by the 3D graphics library. The positive direction of the X axis represents the right, the Y axis the bottom, and the Z axis the depth. The spatial coordinate value of each object is a signed 16-bit integer value ranging from -32768 to +32767.

In the 3D object design phase and within the RSD format, the vertex information is stored as a floating point value. Conversion from RSD into PMD involves converting and scaling vertex values as needed. The scale used is reflected in the object structure, described later, as the reference value. This value can provide an index for mapping from object to world coordinates. The current version of LIBGS ignores the scale value.

OBJ TABLE

OBJ TABLE is a table that contains pointer information regarding the PRIMITIVE Gp for a particular object. A single object is composed of primitive groups.

Figure 2-61: OBJECT Structure



- NOBJ: Number of objects in OBJ TABLE
- NPTR: Number of pointers in a single object
- POINTER: Pointer to a primitive group

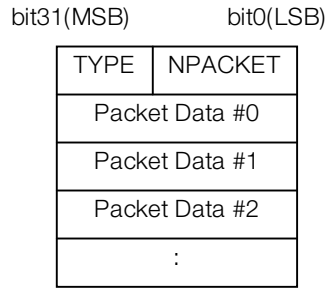
PRIMITIVE Gp

A PRIMITIVE Gp is a group of object structural element (primitive) graphics packets; a single packet contains one primitive.

Primitives defined by PMD are different from drawing primitives handled by libgpu. When PMD primitives undergo perspective transformation by libgs functions, they are converted to drawing primitives.

Each PRIMITIVE Gp has the following structure.

Figure 2-62: Packet Gp structure



TYPE: Packet type (see Table 2-2)

NPACKET: Number of packets

Table 2-2: TYPE bit layout

Bit no.	When 0	When 1
16	Triangle	Quadrilateral
17	Flat	Gouraud
18	Texture-On	Texture-Off
19	Independent vertex	Shared vertex
20	Light source calculation Off	Light source calculation On
21	Back clip	No back clip
22-31	(Reserved for system)	

Packet Data structures change with the value of TYPE. Packet Data structure are broken down by type.

The POLY_... primitive group structure comes in a set of two which corresponds to a double buffer. The contents of both of these must be initialized in advance. Bits 20 and 21 have no effect on packet data structure.

The pkt in each structure indicates a corresponding drawing primitive packet, the vertex coordinate value of v1~v4, and the values of vp1~vp4 offset from the start of the shared vertex row.

TYPE=00 (Triangle/Flat/Texture-On/Independent vertex)

```

struct _poly_ft3 {
    POLY_FT3 pkt[2];
    SVECTOR v1, v2, v3;
}

```

TYPE=01 (Quadrangle/Flat/Texture-On/Independent vertex)

```

struct _poly_ft4 {
    POLY_FT4 pkt[2];
    SVECTOR v1, v2, v3, v4;
}

```

TYPE=02 (Triangle/Gouraud/Texture-On/Independent vertex)

```

struct _poly_gt3 {
    POLY_GT3 pkt[2];
    SVECTOR v1, v2, v3;
}

```

TYPE=03 (Quadrangle/Gouraud/Texture-On/Independent vertex)

```

struct _poly_gt4 {
    POLY_GT4 pkt[2];
    SVECTOR v1, v2, v3, v4;
}

```

TYPE=04 (Triangle/Flat/Texture-Off/Independent vertex)

```

struct _poly_f3 {
    POLY_F3 pkt[2];
    SVECTOR v1, v2, v3;
}

```

TYPE=05 (Quadrangle/Flat/Texture-Off/Independent vertex)

```

struct _poly_f4 {
    POLY_F4 pkt[2];
    SVECTOR v1, v2, v3, v4;
}

```

TYPE=06 (Triangle/Gouraud/Texture-Off/Independent vertex)

```

struct _poly_g3 {
    POLY_G3 pkt[2];
    SVECTOR v1, v2, v3;
}

```

TYPE=07 (Quadrangle/Gouraud/Texture-Off/Independent vertex)

```

struct _poly_g4 {
    POLY_G4 pkt[2];
    SVECTOR v1, v2, v3, v4;
}

```

TYPE=08 (Triangle/Flat/Texture-On/Shared vertex)

```

struct _poly_ft3c {
    POLY_FT3 pkt[2];
    long vp1, vp2, vp3;
}

```

TYPE=09 (Quadrangle/Flat/Texture-On/Shared vertex)

```

struct _poly_ft4c {
    POLY_FT4 pkt[2];
    long vp1, vp2, vp3, vp4;
}

```

TYPE=0a (Triangle/Gouraud/Texture-On/Shared vertex)

```
struct _poly_gt3c {
    POLY_GT3 pkt[2];
    long vp1, vp2, vp3;
}
```

TYPE=0b (Quadrangle/Gouraud/Texture-On/Shared vertex)

```
struct _poly_gt4c {
    POLY_GT4 pkt[2];
    long vp1, vp2, vp3, vp4;
}
```

TYPE=0c (Triangle/Flat/Texture-Off/Shared vertex)

```
struct _poly_f3c {
    POLY_F3 pkt[2];
    long vp1, vp2, vp3;
}
```

TYPE=0d (Quadrangle/Flat/Texture-Off/Shared vertex)

```
struct _poly_f4c {
    POLY_F4 pkt[2];
    long vp1, vp2, vp3, vp4;
}
```

TYPE=0e (Triangle/Gouraud/Texture-Off/Shared vertex)

```
struct _poly_g3c {
    POLY_G3 pkt[2];
    long vp1, vp2, vp3;
}
```

TYPE=0f (Quadrangle/Gouraud/Texture-Off/Shared vertex)

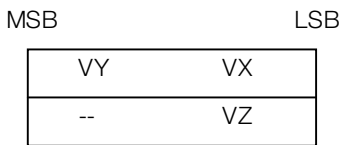
```
struct _poly_g4c {
    POLY_G4 pkt[2];
    long vp1, vp2, vp3, vp4;
}
```

pkt[] indicates the corresponding rendering primitive packet.
v1 to v4 indicates coordinate values of vertices.
vp1 to vp4 indicate offsets from the start of a row of shared vertices.

VERTEX

The VERTEX group is an SVECTOR structure array with shared vertices. The format of one of these structures is shown below.

Figure 2-63: VERTEX structure



VX, VY, VZ: The X, Y, and Z values of the vertex coordinates (16 bit integers)

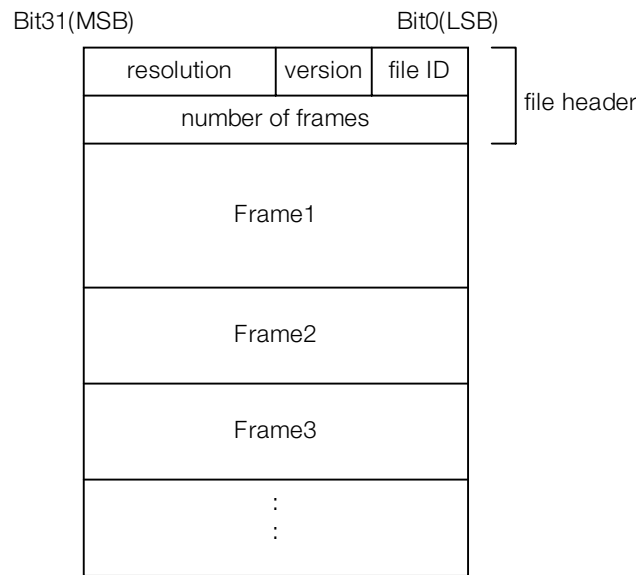
TOD: Animation Data

TOD format is used for specifying information along the flow of time, relative to a 3-dimensional object. It corresponds to the extended graphics library (libgs).

To be more precise, for each frame in a 3-dimensional animation (or frame sequence), the TOD file describes the required data relating to the 3-dimensional objects to be created, modified, or erased, and arranges the data for each frame along the flow of time.

A TOD file, as shown below, consists of a file header followed by frame data.

Figure 2-64: TOD file format



Header

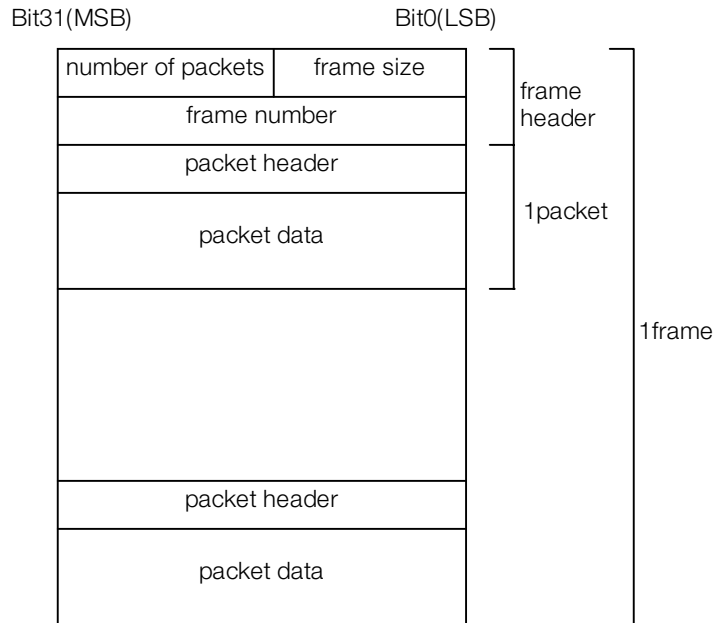
At the top of the TOD file, there is a 2-word (64-bit) HEADER, in which the following four kinds of information are described.

- (a) File ID (8 bits)
This identifies the file as an animation file. Its value is 0x50.
- (b) Version (8 bits)
Animation version. Its value starts at 0x00.
- (c) Resolution (16 bits)
This is the time in which 1 frame is displayed (in units of ticks (1 tick = 1/60 seconds)).
- (d) Number of frames (32 bits)
This is the number of frames described in the file.

Frame

Following the header the frame is described. Frames are arranged chronologically. Each FRAME consists of a frame header followed by a PACKET, as shown below.

Figure 2-65: Frame



Frame Header

There is a 2 word frame header at the beginning of each frame. The following information is described in a frame header.

- Frame size (16 bits)
Frame length (including header) in words.
- Number of packets (16 bits)
Number of packets.
- Frame numbers (32 bits)
Frame number.

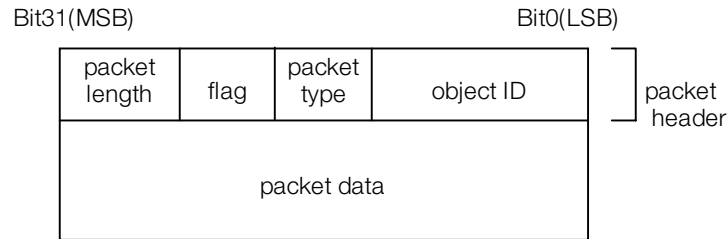
PACKET

After the frame header come the PACKETS. Each PACKET consists of a one-word packet header at the top, followed by the packet data (see Figure 2-66). There are several different kinds of PACKETS.

The size of the packet data in each PACKET will of course be different if the PACKETS are of different kinds; even if the PACKETS are of the same kind, the size of the packet data may be different.

A PACKET consists of a packet header and packet data, as shown below.

Figure 2-66: PACKET



Packet Header

The PACKET header contains the following information.

- Object ID (16 bits)
The identification of the object to be handled.
- Packet type (4 bits)
The type of packet data.
- Flag (4 bits)
The meaning of the flag varies from packet to packet.
- Packet length (8 bits)
This is the size of the packet (including the header) in units of words (4 bytes).

Object refers to a 3-dimensional object (a GsDOBJ2 structure) handled by libgs (the extended graphics library) which is to be made to reflect the packet data.

Packet type contains the classification of the data stored in the packet data. The significance of the flag varies according to the packet type.

Packet length indicates the length of the packet in units of words (4 bytes).

Packet Data

Several kinds of data, such as the GsCOORDINATE2 structure RST value and the TMD data ID (the modeling data ID), are stored in the packet data.

The packet type slot in the header indicates which type the PACKET is. The relationship between the packet type value and the type of data is as follows:

Figure 2-67: packet type values and packet data contents

0000	Attribute
0001	Coordinate (RST)
0010	TMD data ID
0011	Parent object ID
0100	Matrix value
0101	TMD data
0110	Light source
0111	Camera
1000	Object control
1001 – 1101	User defined
1110	System reserved
1111	Special commands

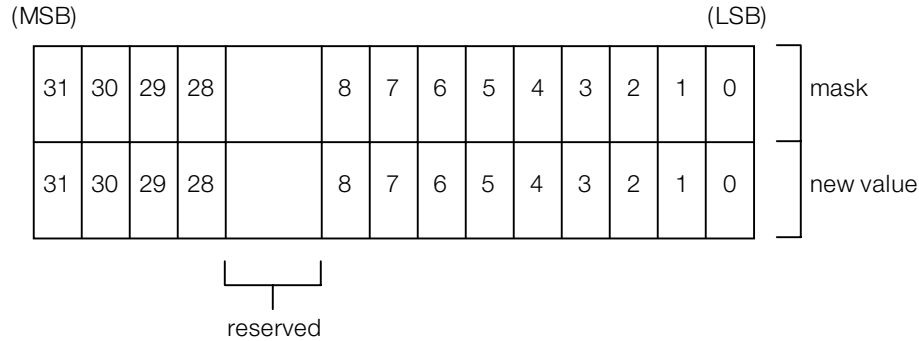
The different kinds of data are explained below.

Packet Data - Attribute

When packet type is 0000, the data that designates attribute of the GsDOBJ structure in the packet data is stored. In this case a flag is not used.

Packet data is composed of 2 words as follows:

Figure 2-68: Packet Data Configuration when Attribute



The first word is a mask which indicates the section which changes value and the section which does not change value. 0 is set in the bit which corresponds to the item which will change and 1 is set in the bit for the value which will not change.

In the second word, new data is input to the bits corresponding to items which are going to change, and the other bits are set to 0.

Note that the first and second words differ in the following respect: in the first word, the default value for the bits which are not going to be changed is 1, while in the second word, this default value is 0.

The breakdown of the bits of the second word packet data shown in Figure 2-68 is described below.

Table 2-3: Packet data bit-by-bit breakdown

Bit (0) - bit (2)	Material damping 00: Material damping 0 01: Material damping 1 02: Material damping 2 03: Material damping 3
Bit (3)	Lighting mode, part 1 0: Fog off (no fog) 1: Fog on (fog)
Bit (4)	Lighting mode, part 2 0: Material on (material) 1: Material off (no material)
Bit (5)	Lighting mode, part 3 0: Use lighting mode 1: Use default lighting mode
Bit (6)	Light source 0: Light-source calculation off 1: Forced light-source calculation on
Bit (7)	NearZ overflow handling 0: z overflow clip 1: z overflow not clip

Bit (8)	Back clipping status 0: Valid 1: Invalid
Bit (9) - bit (27)	Reserved (initialized at 0)
Bit (28) - bit (29)	Semi-transparency rate 00: 50% back + 50% polygon 01: 100% back + 100% polygon 10: 100% back - 100% polygon 11: 50% back + 25% polygon
Bit (30)	Semi-transparency rate 0: Off 1: On
Bit (31)	Display 0: Display 1: No display

For example, to switch forced light-source calculation ON, the packet data bits should be set as shown in Figure 2-69.

Bit (6) of the first word is given the value 0, showing that the light source is to be changed. The other bits are given the value 1, showing that they are not going to be changed. Accordingly, the first word is 0xffbf.

Bit (6) of the second word is given the value 1 to indicate that forced light-source calculation is ON, and the other bits, which correspond to items which are not going to be changed, are given the default value 0. The second word is therefore 0x0040.

Figure 2-69: Packet data when forced light-source calculation is switched ON

MSB					LSB										mask
1	1	1	1		1	1	0	1	1	1	1	1	1	1	
0	0	0	0		0	0	1	0	0	0	0	0	0	0	new value

Packet Data - Coordinate (RST)

When packet type is 0001, data that sets the coordinates of the GsDOBJ structure is stored in packet data.

In this case the flag will have the following meaning.

Figure 2-70: Flag when Coordinate (RST)

translation	scaling	rotation	matrix type
-------------	---------	----------	-------------

Matrix type: RST matrix type
0: Absolute value
1: Differential matrix from preceding frame

Rotation: Rotation (R) flag
0: None
1: Has

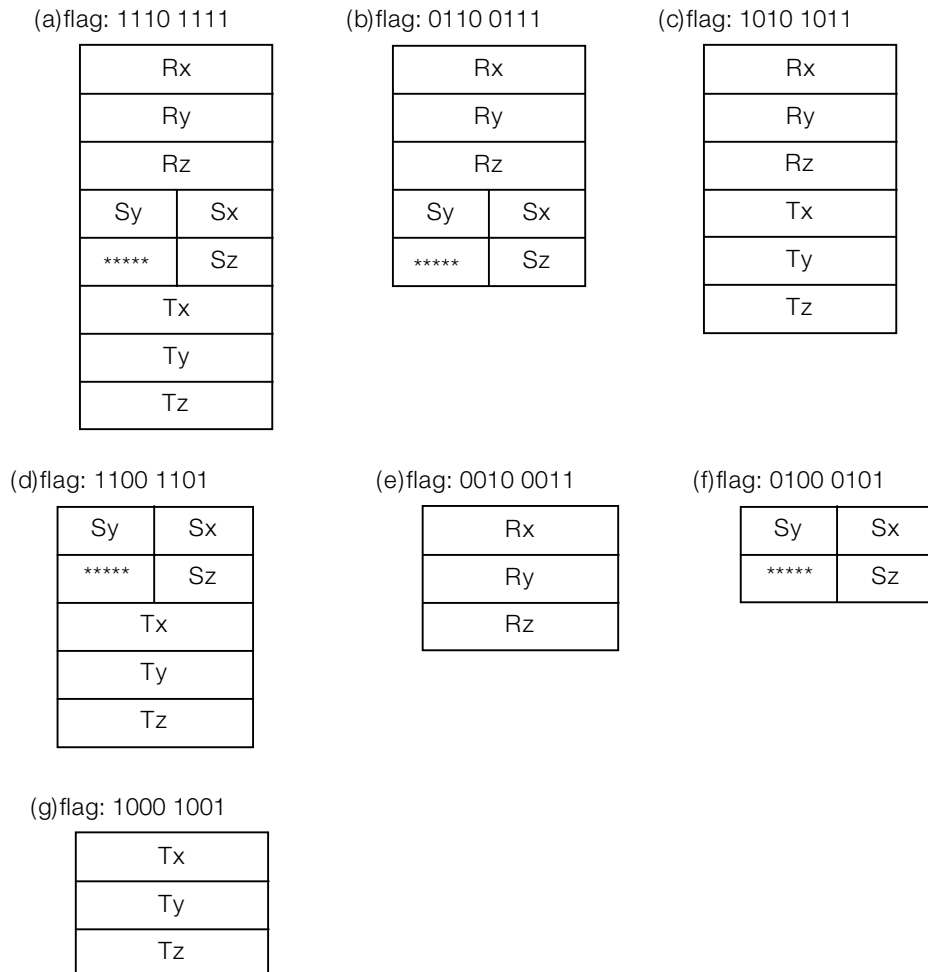
Scaling: Screening (S) flag
0: None
1: Has

Translation Parallel movement (T) flag
 0: None
 1: Has

The configuration of packet data will differ according to the values of the flag rotation bit, the scaling bit, and the translation bit as per Figure 2-70.

In Figure 2-71, Rx, Ry and Rz indicate one degree as 4096, with a fixed point decimal value (1, 19, 12) that indicate the X axis component, the Y axis component, and the Z axis component of the angle of rotation. In the same way, Sx, Sy and Sz indicate the X axis component, the Y axis component, and the Z axis component of the scaling as a fixed point decimal (1, 3, 12), while Tx, Ty and Tz respectively indicate the X axis component, the Y axis component, and the Z axis component of the translation as an integer (1, 31, 0) that signals 32 bits.

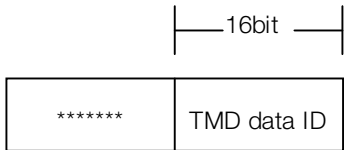
Figure 2-71: Packet Data Configuration when Coordinate (RST)



Packet Data - TMD Data ID

When packet type is 0010, the modeling data ID (TMD data) of the real object is stored in the packet data (See Figure 2-72). The TMD data ID is composed of 2 bytes. In this case no flag is used.

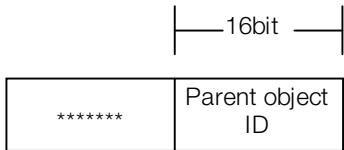
Figure 2-72: Packet Data Configuration when TMD Data ID



Packet Data - Parent Object ID

When packet type is 0011, the parent object ID of the object specified is stored in packet data (see Figure 2-73). The parent object ID is composed of 2 bytes. In this case no flag is used.

Figure 2-73: Packet Data Configuration when Parent Object



Packet Data - MATRIX Value

When the packet type is 0100, the data which designates coord members of the GsCOORDINATE2 structure to which GsDOBJ2 structure points is stored in packet data. In this case a flag is not used.

Figure 2-74: Packet Data Configuration when Matrix Value

R01	R00
R10	R02
R12	R11
R21	R20
*****	R22
Tx	
Ty	
Tz	

Packet Data - TMD Data Body

When packet type is 0101, TMD data is stored. This is not presently supported.

Packet Data - Light Source

When packet type is 0110, the data that designates light source is stored in packet data. When this is the case, the object ID is separate from the normal object ID and becomes the light source ID. Flags have the following meanings:

Figure 2-75: Flag when Light Source Packet

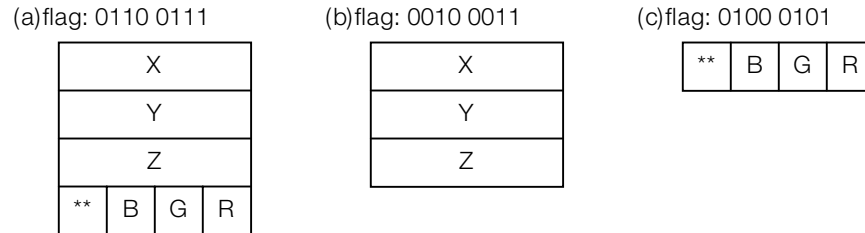
*****	Color	Direction	Data type
-------	-------	-----------	-----------

- Data type: Data type
 0: Absolute value
 1: Difference from preceding frame
- Direction: Direction flag
 0: None
 1: Has

Color: Color flag
 0: None
 1: Has

The configuration of packet data will differ according to the value of the flag direction bit and the color bit.

Figure 2-76: Packet Data when Light Source Packet



Packet Data-Camera

When packet type is 0111, data which designates viewpoint location information is stored in the packet. When this is the case, the object ID is separate from the normal object ID and becomes the camera ID. Flags have the meaning indicated in Figure 2-77. Please be careful to note that the meaning of other bits will change depending on the type bit.

Figure 2-77: Flag for Camera

(1) camera type: 0

z angle	position & reference	data type	camera type = 0
---------	----------------------	-----------	--------------------

(2) camera type: 1

translation	rotation	data type	camera type = 1
-------------	----------	-----------	--------------------

When camera type bit is 0 other bits are:

Data type: Data type
 0: Absolute value
 1: Difference from preceding frame

Position & reference Position and reference position flag
 0: None
 1: Has

z angle Reference angle flag from level
 0: None
 1: Has

When camera type bit is 1 other bits are:

Data type: Data type
 0: Absolute value
 1: Difference from preceding frame

Rotation: Rotation (R) flag
 0: None
 1: Has

Translation: Horizontal movement (T) flag
 0: None
 1: Has

The structure of packet data differs according to the flag content, as shown in Figure 2-78 and Figure 2-79.

Figure 2-78: Composition of packet data with camera (part 1)

(a) flag: 1100 or 1110	(b) flag: 0100 or 0110
Tx	Tx
Ty	Ty
Tz	Tz
TRx	TRx
TRy	TRy
TRz	TRz
Z	

(c) flag: 1000 or 1010
Z

Tx, Ty, Tz: camera position

TRx, TRy, TRz: camera close-up position

Figure 2-79: Composition of packet data with camera (part 2)

(a) flag: 1101 or 1111	(b) flag: 0101 or 0111
Rx	Rx
Ry	Ry
Rz	Rz
Tx	
Ty	
Tz	

(c) flag: 1001 or 1011
Tx
Ty
Tz

Rx, Ry, Rz: Rotation

Tx, Ty, Tz: Translation

Packet Data-Object Control

If the packet type is 1000, object control is not set. In this case, there is no packet data. The flag has the meanings shown below.

Figure 2-80: The meanings and values of the flag when object control is set

0	create
1	kill
0010-1111	system reserved

Packet Data-Extended Commands

If the packet type is 1110, it shows the extended commands.

Packet Data-Special Commands

If the packet type is 1111, animation control is performed. Details of these special commands have not yet been finalized.

HMD: Hierarchical 3D Model, Animation and Other Data

Some of the descriptions in this section use HMD Assembler (labp) format. Refer to the labp section of the Data Conversion Manual.

HMD is a generic graphics format that allows model data, texture data, and animation data to be handled all within an integrated framework.

HMD can be easily extended to handle additional kinds of data with a unique identification code known as a type.

HMD data can be easily played back on the PlayStation using libgs. A program that is used to playback HMD-formatted data is referred to as a primitive driver. Primitive drivers are linked to HMD data through their type.

Sony Computer Entertainment has created a set of standard primitive drivers for libgs. These primitive drivers have standardized interfaces or APIs, so end users and middleware companies can also build their own primitive drivers.

The HMD format is supported by Library Version 4.0 and later.

Abstract of the HMD (for All categories)

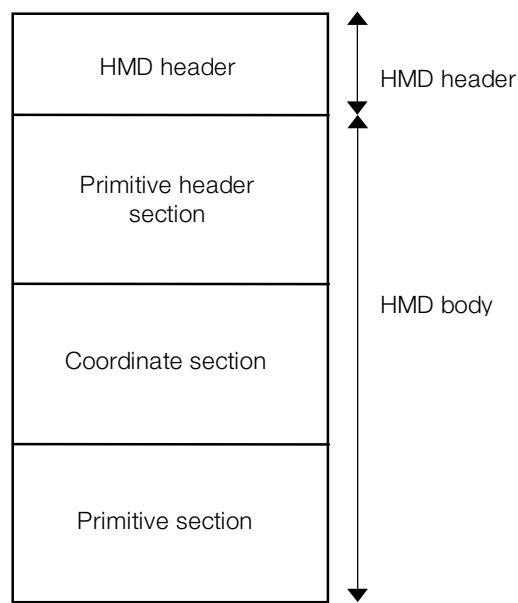
The HMD format supports several categories of data. Examples of categories include model data and image data. Each category can have its own individual data format. This chapter describes the HMD structures that are the same across all categories of data.

HMD data is divided into the following two main parts.

1. The HMD header
2. The HMD body

The HMD body is made up of areas known as sections. Two sections, one called the primitive section and the other known as the primitive header section, are always required. Other sections are included only if required by the specific type.

Figure 2-81: HMD Structure



In the example shown above, a coordinate section is included in addition to the required sections. The following is a detailed description of the HMD format.

Notations

In this discussion, pointer values, which represent addresses, are converted at runtime into real addresses in memory. The process of converting pointer values to real addresses is known as mapping and is performed by the `GsMapUnit()` function.

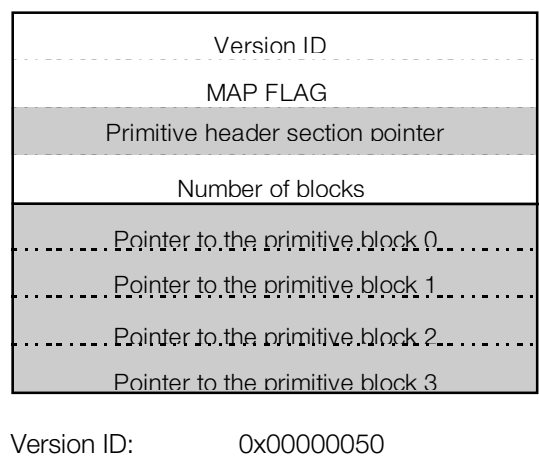
HMD data can be used only after addresses are mapped.

Pointers are shown highlighted in the figures. The initial value of a pointer is the number of words from the top of HMD data, where one word is equal to 32 bits.

HMD Header

The HMD header contains the version ID, MAP FLAG, the starting address of the primitive header section, and the number of primitive blocks. Primitive blocks and the primitive header section will be described in more detail later. The HMD header also contains a list of pointers to the primitive blocks.

Figure 2-82: HMD Header section



MAP FLAG: A flag that is used to indicate if the GsMapUnit() function has been called or not. The GsMapUnit() references this field and changes it. This value is 1 if mapped, otherwise 0.

Number of blocks: The number of primitive blocks pointers.

HMD Data

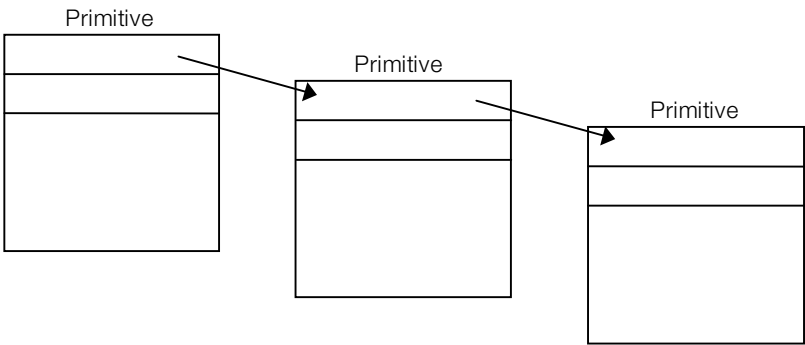
Primitive Section

A primitive section is defined to be a collection of primitive blocks.

Primitive Block

A primitive block is defined to be a chain of one or more primitives linked together by pointers. HMD data consists of one or more primitive blocks.

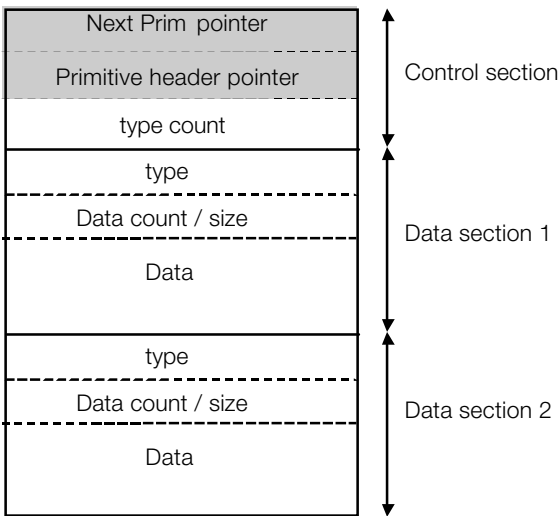
Figure 2-83: One primitive block which has been primitive chained



The Structure of a Primitive

Primitives consist of a control section and one or more data sections.

Figure 2-84: Primitive Structure



Control Section

Next Prim pointer: Pointer to the header of the next primitive, thus forming the primitives chain. A value of 0xFFFFFFFF indicates that this is the last primitive in the chain.

Primitive header pointer: Pointer to the primitive header. Primitive headers will be described later.

Number of types: Number of data sections. The MSB serves as a flag indicating whether the NextPrim pointer and the primitive header pointer have been mapped. The MSB of the type count is 1 if UNMAPped, and 0 if MAPped.

Data Section

type: Identifier of the data. Type is overwritten during a SCAN operation with the starting address of the driver used to process the data. Each type is unique within HMD. If the value of type is changed, the contents of the data and its driver can also be changed. SCAN and type will each be described in more detail later.

Number of data / Size: The upper 16 bits of this field contain the data count for a single data section. A single type generally processes multiple sets of data. The data count indicates how many sets of data there are to process. In other words, how many times the data process will be repeated. The lower 16 bits contain the size of one data section in words.

Data: The actual data is placed here. The data format depends on the value of the type field.

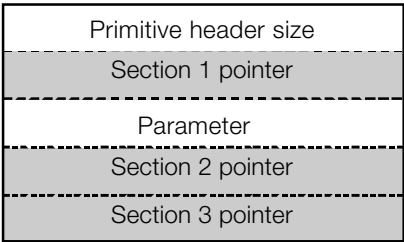
Primitive header

Primitive headers are grouped together and placed in the primitive header section. A pointer to the primitive header section is saved in the HMD header.

The first word of the primitive header structure is the size of the primitive header in words. Pointers to each of the sections follow. There is one primitive header for each primitive block. Within the primitive header are pointers to the sections referred to by the primitive block.

Setting the MSB of its data word to 1 identifies a pointer to a section. When the MSB is 0, the data is interpreted as a numeric value rather than as an address pointer. These unmapped values can be used as parameters for a primitive driver.

Figure 2-85: Primitive Header

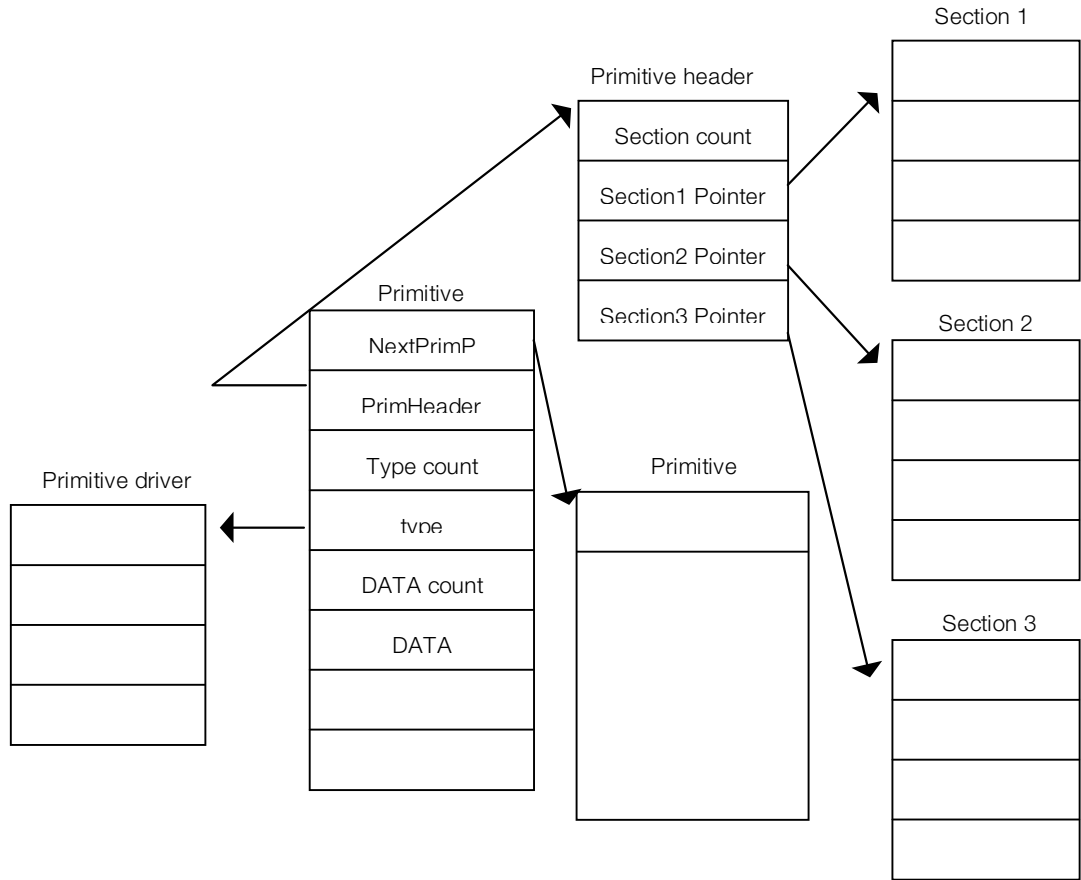


Basic structure of a primitive

A primitive is made up of three components: the primitive header, the primitive driver, and its data. The figure below shows the relationship between these components. The primitive header contains pointers to the beginning of the corresponding data sections. The data sections, which the pointers refer to depend on the type of primitive.

The primitive data and its corresponding sections are evaluated together by the primitive driver. The primitive driver is identified by the type field, which is overwritten, with the starting address of the driver. This process is known as a SCAN. SCAN uses the GsScanUnit() function to extract the address of the type field and its value for each primitive. Then it can be replaced with the starting address of the primitive driver into the type field for each primitive.

Figure 2-86: Primitive Structure



The following sections give examples of each of the data categories for use with HMD.

HMD Model Data (Category 0)

The model data must contain the following sections:

1. HMD header section
2. Primitive header section
3. Coordinate section
4. Primitive section
5. Polygon section
6. Vertex section

The following sections can also be included:

1. Normal section
2. Image section

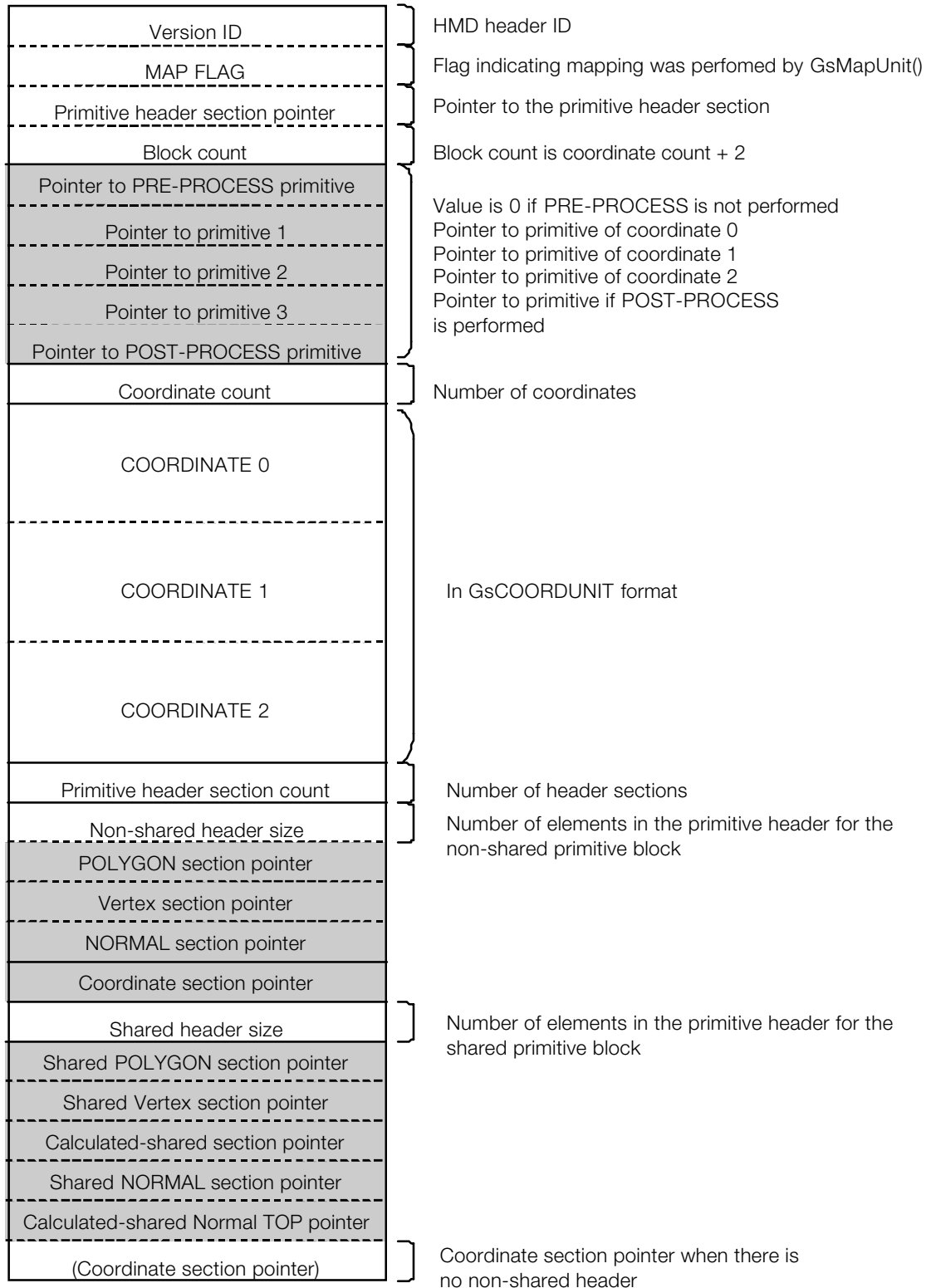
In HMD, model data consists of multiple coordinate systems and each coordinate system is assigned to a separate primitive block.

The HMD primitive header section contains a primitive header for each primitive block.

The following description is an example of the HMD format for model data.

Overall Structure

The structure of shared vertices HMD



NEXT Prim pointer	}	Pointer to the next primitive. The calculation process for the shared primitive's VERTEX and NORMAL is defined by the next chain of the non-shared primitive.
Non-shared header pointer		
type count	}	Pointer to primitive header of non-shared vertex
type	}	Number of types
Polygon count / size	}	TYPE
POLYGON IDX	}	The number of polygons and size for this type
type	}	Index into the primitive type's polygon section
Polygon count / size		
POLYGON IDX		
TERMINATE	}	TERMINATE indicates that this is the last primitive
Shared header pointer		
type count	}	MSB of the type count is a flag indicating map completed
type	}	Shared primitive type
Polygon count / size		
Shared VERTEX count		
Shared VERTEX offset (src)		
Shared VERTEX offset (dst)		
Shared NORMAL count		
Shared NORMAL offset (src)		
Shared NORMAL offset (dst)		
TERMINATE		
Shared header pointer		
type count		
type		
Polygon count / size		
Shared POLYGON IDX		

Pointer to the next primitive.

The calculation process for the shared primitive's VERTEX and NORMAL is defined by the next chain of the non-shared primitive.

Pointer to primitive header of non-shared vertex

Number of types

TYPE

The number of polygons and size for this type

Index into the primitive type's polygon section

TERMINATE indicates that this is the last primitive

MSB of the type count is a flag indicating map completed

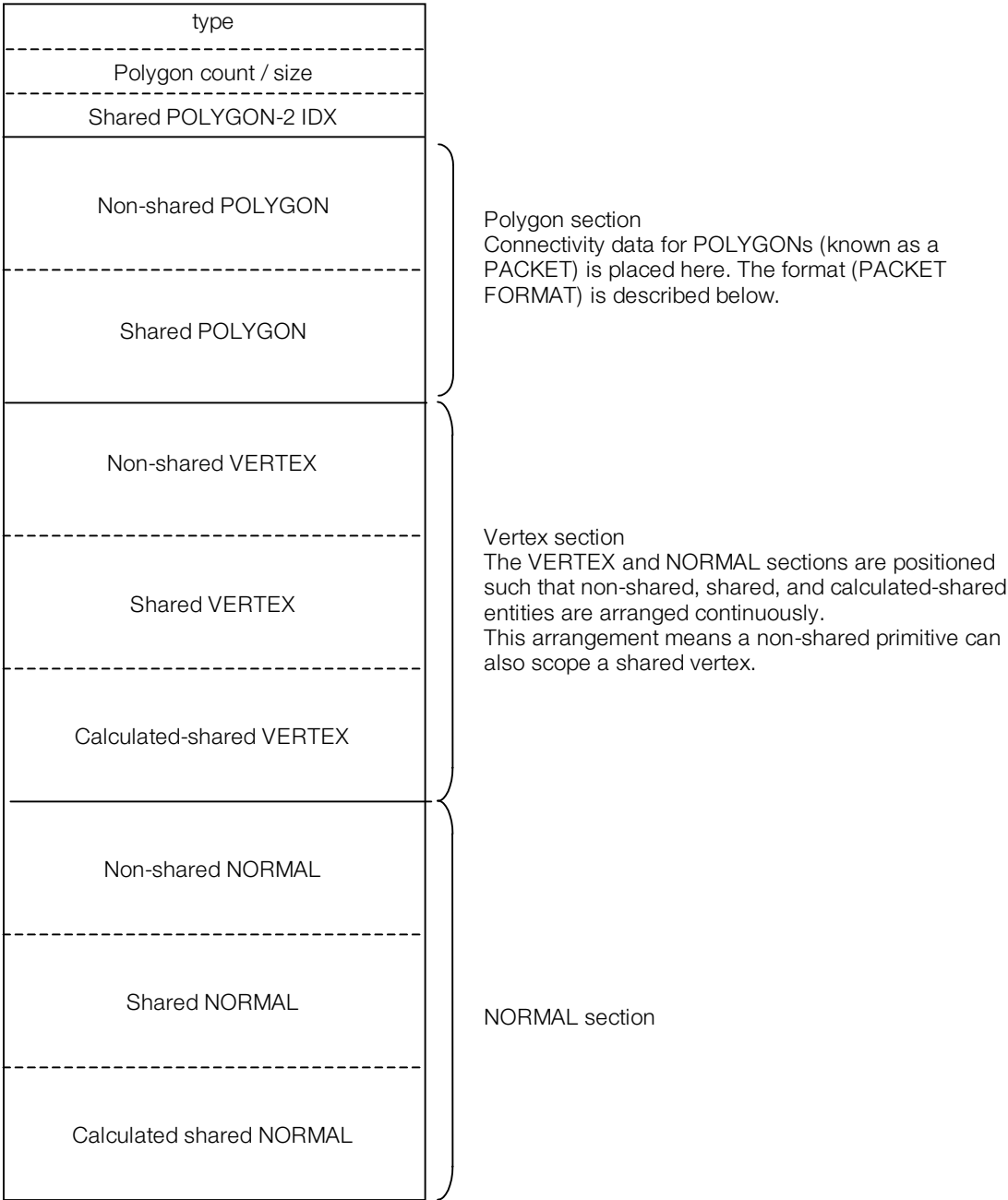
Shared primitive type

Offset specifies the number of words from the start of the shared VERTEX.

Two buffers for input and output are independently defined allowing shared primitives to be reused.

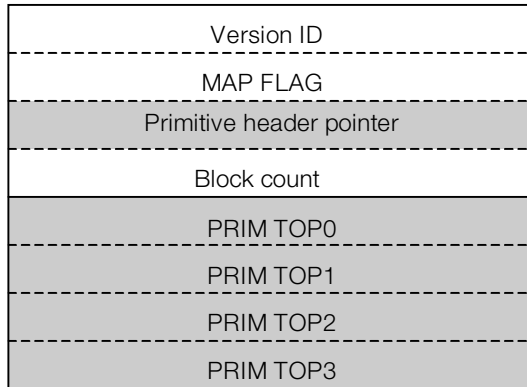
This shared primitive is defined as the POST-PROCESS primitive.

The header is the same as that for the shared primitive.



HMD Header Section

Figure 2-87: HMD Header Section



Version ID:	Version number of the HMD format. Currently 0x00000050.
MAP FLAG:	Flag indicating whether mapping was performed. This flag is accessed and updated by GsMapUnit(). This value is 0x00000000 if MAPped, and 0x00000001 if UNMAPped.
Primitive header top:	Pointer to primitive header section (offset value from top, in words) MSB is 1 when data in the primitive header section has been mapped using GsMapUnit().
Block count:	Number of blocks. There is 1 block per coordinate as well as a PRE-PROCESS block and a POST-PROCESS block. Therefore the block count is equal to the number of coordinates + 2.
Primitive pointer table:	Contains a pointer to the primitive in each block. The first block is used for PRE-PROCESS and does not have a coordinate. The next blocks correspond to indexes from the coordinate tops. The last block is used for POST-PROCESS and does not have a coordinate.

Table 2-4 : Primitive Pointer Table

Block	Coordinate	Primitive	Process
BLOCK 0		PRIM 0	PRE-PROCESS
BLOCK 1	COORDINATE0	PRIM 1	Block 1 process
BLOCK 2	COORDINATE1	PRIM 2	Block 2 process
BLOCK 3	COORDINATE2	PRIM 3	Block 3 process
BLOCK N	COORDINATE N-1	PRIM N	Block N process
BLOCK N+1		PRIM N+1	POST-PROCESS

COORDINATE Section

The coordinate section contains coordinate system data for each block.

The first word of the coordinate section indicates the number of coordinates.

Coordinates are represented in GsCOORDUNIT format as shown below.

```
GsCOORDUNIT {
  unsigned long flg;
  MATRIX matrix;
  MATRIX workm;
  SVECTOR rot;
  struct GsCOORDUNIT *super;
}
```

The consistency between rot and matrix must be maintained during construction of HMD data.

Primitive Header Section

The primitive header section contains a collection of primitive headers and global data for the primitive block. When a primitive driver is called, GsSortUnit() copies the data shown below to a variable transfer area. The size of the copied data is saved in the header size.

This process enables the primitive driver to access data in the primitive header. Since the primitive header contains pointers to normal and vertex section headers, the driver is able to access data in these sections.

The MSB of the data denotes whether or not the value will be mapped. If the value will not be mapped (MSB = 0), it is considered to be an ordinary number. If it will be mapped (MSB = 1), the value is treated as a pointer.

Figure 2-88: Variable transfer area transferred to the primitive driver

primtop
Tag(OT)
Shift(OT)
Offset(OT)
OUTP(packet area)
Primitive Header
:
:

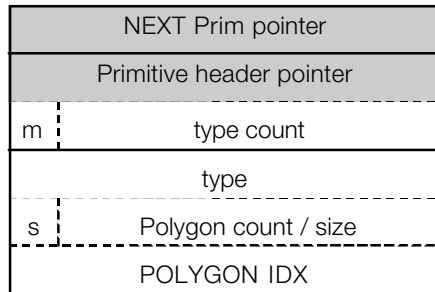
Primitive Section

The primitive section contains one or more primitive blocks. Each block corresponds to one coordinate. For model data, the first primitive is used for non-shared vertex data and the next block is a primitive that is used for shared vertex data. If non-shared vertex primitives and shared vertex primitives are not present in the model data, this section can be omitted.

Primitives

As shown below, primitives consist of several types of data.

Figure 2-89: One Primitive



The first three words in a primitive specify the control section. This section is made up of a NextPrim pointer, a primitive header pointer, and a type count. The MSB of the type count(*m*) serves as a flag indicating whether or not the NextPrim pointer and the primitive header pointer have been mapped. If *m*=1, the pointers have not been mapped. Conversely, if *m*=0, the pointers have been mapped.

The data section of a primitive is organized in three-word units. Each unit is made up of a type field, which serves as an identifying code, the number of polygons in the data for this type, and POLYGON IDX, which is a pointer to the actual polygon data. These three words are repeated according to the number of types in the control section.

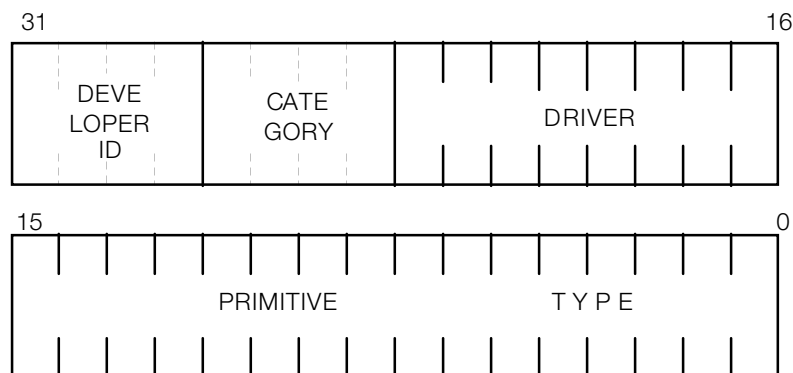
The MSB of the polygon count is a flag indicating whether a SCAN operation has been performed. The lower 16 bits indicate the size of the data contained in the type.

The upper 8 bits (*n*) of POLYGON IDX can be used as a parameter for the primitive driver. In the current implementation, DIV and ADV in the polygon data DRIVER bits (category 0) are used to control the number of polygon divisions. DIV stores the actual number of divisions (fixed divisions), while ADV stores the maximum number of divisions (automatic division). The allowed values of DIV and ADV are defined in `libgs.h` as `GsUNIT_DIV1` - `GsUNIT_DIV5`.

type

The type field consists of 32 bits arranged in four sections. The upper 8 bits contain data that is common to all categories.

Figure 2-90: Type Field

**Common to all Categories**

DEVELOPER ID: Contains the ID of the developer who created this format. If ID is unique, the developer may use the other bits freely. A total of 16 IDs are available. The value zero is assigned to Sony Computer Entertainment.

All 16 ID
 0x0: SCE
 0xf: User defined

CATEGORY: This identifies the category of data such as polygon data or image data. 16 categories are available.

0: Polygon data
 1: Shared polygon data
 2: Image data
 3: Animation data
 4: MIMe data
 5: Ground data

Polygon Data (Category 0)**DRIVER**

These bits can be used to change the behavior of the primitive driver for a given type of primitive data. For example, polygon subdivision can be enabled. 8 bits are available.

Figure 2-91: Polygon Primitive Driver

I	C	S	B	A	L	F	D
N	L	T	O	D	G	O	I
I	P	P	T	V	T	G	V

- DIV** 0: Disable subdivision
1: Perform subdivision
- FOG** 0: Turn FOG OFF
1: Turn FOG ON
- LGT** 0: Perform light-source calculation
1: Disable light-source calculation (forcibly mask off light-source calculation during execution)
- ADV** 0: Do not perform automatic division
1: Perform automatic division
- BOT** 0: Single-sided polygon
1: Double-sided polygon
- STP** 0: (Make semi-transparent if already semi-transparent. Make opaque if already opaque)
1: make all polygons semi-transparent
- CLP** (not implemented)
0: Disable clipping
1: Clip polygons that are outside the screen at beyond the upper and left
- INI** 0: Do not initialize
1: Initialize

When initialization is specified, an initialization function is called to set up the environment before SCAN is performed. In some cases, this bit is set when a type is first used.

PRIMITIVE TYPE

The value of these bits depends on the type of primitive.

Figure 2-92: Primitive Type of Polygon Primitive

						T I L E	P S T	M I P	L M D	CODE	I I P	C O L	T M E
--	--	--	--	--	--	------------------	-------------	-------------	-------------	------	-------------	-------------	-------------

- TME**
0: Disable texture mapping
1: Perform texture mapping
- COL**
0: Use one material color for identical polygons
1: Use a separate color for each vertex
- IIP**
0: Flat-shaded polygon
1: Gouraud-shaded polygon
- CODE**
Describes the shape of the polygon
0: Reserved by the system
1: Triangle
2: Quadrangle
3: Strip mesh
4-7: Reserved by the system
- LMD**
0: Has normal
1: Does not have normal
- MIP**
(not implemented)
0: Disable MIP-mapping
1: Perform MIP-mapping
- PST**
0: No presets
1: Preset packet available
- TILE**
0: No information for tiled textures
1: Information available for tiled textures
- MIMe**
0: Normal polygon
1: MIMe polygon (not implemented)

Number of polygons / Size

Figure 2-93: Number and Size of Polygons

f l g	Polygon count	Data size (in words)
-------------	---------------	----------------------

- fig**
flag indicating whether or not SCAN was performed
0: SCAN was performed
1: SCAN was not performed
- Number of polygons**
Number of polygons in type.
- Data size_@**
Size of data in type (in words)

Polygon Section

The polygon section contains polygon connection information. PACKETS are used to represent this information and are classified according to type.

A PACKET has NORMAL and VERTEX fields that are referenced by an index, and an RGB field that contains actual values.

The polygon type can be one of the following shapes:

1. Triangle
2. Quadrangle
3. MESH

For MESH, the first num field specifies the number of connections.

A list of PACKETS by type is shown below.

The type of polygon is shown at the upper left, and the value of the type field is shown at the upper right. The contents of the PACKET are drawn as a series of rows, with each row representing one word (32 bits). The meaning of the symbols shown is basically the same as that for TMD.

Polygon Types

With Light-source Calculation

Flat No-Texture Triangle

```
0x00000008; DRV(0)|PRIM_TYPE(TRI); GsUF3
```

```
B(r); B(g); B(b); B(0x20);
```

```
H(norm0); H(vert0);
```

```
H(vert1); H(vert2);
```

Gouraud No-Texture Triangle

```
0x0000000c; DRV(0)|PRIM_TYPE(TRI|IIP); GsUG3
```

```
B(r); B(g); B(b); B(0x30);
```

```
H(norm0); H(vert0);
```

```
H(norm1); H(vert1);
```

```
H(norm2); H(vert2);
```

Colored Flat No-Texture Triangle

```
0x0000000a; DRV(0)|PRIM_TYPE(TRI|COL); GsUCF3
```

```
B(r0); B(g0); B(b0); B(0x30);
```

```
B(r1); B(g1); B(b1); B(0x30);
```

```
B(r2); B(g2); B(b2); B(0x30);
```

```
H(norm0); H(vert0);
```

```
H(vert1); H(vert2);
```

Colored Gouraud No-Texture Triangle

0x0000000e; DRV(0)|PRIM_TYPE(TRI|IIP|COL); GsUCG3

B(r0); B(g0); B(b0); B(0x30);

B(r1); B(g1); B(b1); B(0x30);

B(r2); B(g2); B(b2); B(0x30);

H(norm0); H(vert0);

H(norm1); H(vert1);

H(norm2); H(vert2);

Flat Texture Triangle

0x00000009; DRV(0)|PRIM_TYPE(TRI|TME); GsUFT3

B(u0); B(v0); H(cba);

B(u1); B(v1); H(tsb);

B(u2); B(v2); H(0);

H(norm0); H(vert0);

H(vert1); H(vert2);

Gouraud Texture Triangle

0x0000000d; DRV(0)|PRIM_TYPE(TRI|IIP|TME); GsUGT3

B(u0); B(v0); H(cba);

B(u1); B(v1); H(tsb);

B(u2); B(v2); H(0);

H(norm0); H(vert0);

H(norm1); H(vert1);

H(norm2); H(vert2);

Colored Flat Texture Triangle

0x0000000b; DRV(0)|PRIM_TYPE(TRI|COL|TME); GsUCFT3

B(r0); B(g0); B(b0); B(0x34);

B(r1); B(g1); B(b1); B(0x34);

B(r2); B(g2); B(b2); B(0x34);

B(u0); B(v0); H(cba);

B(u1); B(v1); H(tsb);

B(u2); B(v2); H(0);

H(norm0); H(vert0);

H(vert1); H(vert2);

Colored Gouraud Texture Triangle

```
0x0000000f; DRV(0)|PRIM_TYPE(TRI|IIP|COL|TME); GsUCGT3
```

```
B(r0); B(g0); B(b0); B(0x34);
```

```
B(r1); B(g1); B(b1); B(0x34);
```

```
B(r2); B(g2); B(b2); B(0x34);
```

```
B(u0); B(v0); H(cba);
```

```
B(u1); B(v1); H(tsb);
```

```
B(u2); B(v2); H(0);
```

```
H(norm0); H(vert0);
```

```
H(norm1); H(vert1);
```

```
H(norm2); H(vert2);
```

Flat No-Texture Quad

```
0x00000010; DRV(0)|PRIM_TYPE(QUAD); GsUF4
```

```
B(r); B(g); B(b); B(0x28);
```

```
H(norm0); H(vert0);
```

```
H(vert1); H(vert2);
```

```
H(vert3); H(0);
```

Gouraud No-Texture Quad

```
0x00000014; DRV(0)|PRIM_TYPE(QUAD|IIP); GsUG4
```

```
B(r); B(g); B(b); B(0x38);
```

```
H(norm0); H(vert0);
```

```
H(norm1); H(vert1);
```

```
H(norm2); H(vert2);
```

```
H(norm3); H(vert3);
```

Colored Flat No-Texture Quad

```
0x00000012; DRV(0)|PRIM_TYPE(QUAD|COL); GsUCF4
```

```
B(r0); B(g0); B(b0); B(0x38);
```

```
B(r1); B(g1); B(b1); B(0x38);
```

```
B(r2); B(g2); B(b2); B(0x38);
```

```
B(r3); B(g3); B(b3); B(0x38);
```

```
H(norm0); H(vert0);
```

```
H(vert1); H(vert2);
```

```
H(vert3); H(0);
```

Colored Gouraud No-Texture Quad

```
0x00000016; DRV(0)|PRIM_TYPE(QUAD|IIP|COL); GsUCG4
```

```
B(r0); B(g0); B(b0); B(0x38);
```

```
B(r1); B(g1); B(b1); B(0x38);
```

```
B(r2); B(g2); B(b2); B(0x38);
```

```
B(r3); B(g3); B(b3); B(0x38);
```

```
H(norm0); H(vert0);
```

```
H(norm1); H(vert1);
```

```
H(norm2); H(vert2);
```

```
H(norm3); H(vert3);
```

Flat Texture Quad

```
0x00000011; DRV(0)|PRIM_TYPE(QUAD|TME); GsUFT4
```

```
B(u0); B(v0); H(cba);
```

```
B(u1); B(v1); H(tsb);
```

```
B(u2); B(v2); H(0);
```

```
B(u3); B(v3); H(norm0);
```

```
H(vert0); H(vert1);
```

```
H(vert2); H(vert3);
```

Gouraud Texture Quad

```
0x00000015; DRV(0)|PRIM_TYPE(QUAD|IIP|TME); GsUGT4
```

```
B(u0); B(v0); H(cba);
```

```
B(u1); B(v1); H(tsb);
```

```
B(u2); B(v2); H(norm0);
```

```
B(u3); B(v3); H(vert0);
```

```
H(norm1); H(vert1);
```

```
H(norm2); H(vert2);
```

```
H(norm3); H(vert3);
```

Colored Flat Texture Quad

```
0x00000013; DRV(0)|PRIM_TYPE(QUAD|COL|TME); GsUCFT4
```

```
B(r0); B(g0); B(b0); B(0x3c);
```

```
B(r1); B(g1); B(b1); B(0x3c);
```

```
B(r2); B(g2); B(b2); B(0x3c);
```

```
B(r3); B(g3); B(b3); B(0x3c);
```

```
B(u0); B(v0); H(cba);
```

```
B(u1); B(v1); H(tsb);
```

```
B(u2); B(v2); H(0);
```

```
B(u3); B(v3); H(norm0);
```

```
H(vert0); H(vert1);
```

```
H(vert2); H(vert3);
```

Colored Gouraud Texture Quad

```
0x00000017; DRV(0)|PRIM_TYPE(QUAD|IIP|COL|TME); GsUCGT4
```

```
B(r0); B(g0); B(b0); B(0x3c);
B(r1); B(g1); B(b1); B(0x3c);
B(r2); B(g2); B(b2); B(0x3c);
B(r3); B(g3); B(b3); B(0x3c);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(norm0);
B(u2); B(v2); H(vert0);
H(norm1); H(vert1);
H(norm2); H(vert2);
H(norm3); H(vert3);
```

Flat No-Texture Mesh

```
0x00000018; DRV(0)|PRIM_TYPE(MESH); GsUMF3
```

```
H(num); H(0);
B(r); B(g); B(b); B(0x20);
H(norm2); H(vert0);
H(vert1); H(vert2);
/*-----*/
B(r); B(g); B(b); B(0x20);
H(norm3); H(vert3);
```

Gouraud No-Texture Mesh

```
0x0000001c; DRV(0)|PRIM_TYPE(MESH|IIP); GsUMG3
```

```
H(num); H(0);
B(r2); B(g2); B(b2); B(0x30);
H(norm0); H(vert0);
H(norm1); H(vert1);
H(norm2); H(vert2);
/*-----*/
B(r3); B(g3); B(b3); B(0x30);
H(norm1); H(norm2);
H(norm3); H(vert3);
```

Colored Flat No-Texture Mesh**0x0000001a; DRV(0)|PRIM_TYPE(MESH|COL)**

```

H(num); H(0);

B(r0); B(g0); B(b0); B(0x30);
B(r1); B(g1); B(b1); B(0x30);
B(r2); B(g2); B(b2); B(0x30);

H(norm2); H(vert0);
H(vert1); H(vert2);

/*-----*/

B(r3); B(g3); B(b3); B(0x30);
H(norm3); H(vert3);

```

Colored Gouraud No-Texture Mesh**0x0000001e; DRV(0)|PRIM_TYPE(MESH|IIP|COL)**

```

H(num); H(0);

B(r0); B(g0); B(b0); B(0x30);
B(r1); B(g1); B(b1); B(0x30);
B(r2); B(g2); B(b2); B(0x30);

H(norm0); H(vert0);
H(norm1); H(vert1);
H(norm2); H(vert2);

/*-----*/

B(r3); B(g3); B(b3); B(0x30);
H(norm1); H(norm2);
H(norm3); H(vert3);

```

Flat Texture Mesh**0x00000019; DRV(0)|PRIM_TYPE(MESH|TME); GsUMFT3**

```

H(num); H(0);

B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(0);

H(norm2); H(vert0);
H(vert1); H(vert2);

/*-----*/

B(u1a); B(v1a); H(cba);
B(u2a); B(v2a); H(tsb);
B(u3); B(v3); H(0);
H(norm3); H(vert3);

```

Gouraud Texture Mesh

```

0x0000001d; DRV(0)|PRIM_TYPE(MESH|IIP|TME); GsUMGT3
H(num); H(0);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(0);
H(norm0); H(vert0);
H(norm1); H(vert1);
H(norm2); H(vert2);
/*-----*/
B(u1a); B(v1a); H(cba);
B(u2a); B(v2a); H(tsb);
B(u3); B(v3); H(0);
H(norm1); H(norm2);
H(norm3); H(vert3);

```

Colored Flat Texture Mesh

```

0x0000001b; DRV(0)|PRIM_TYPE(MESH|COL|TME)
H(num); H(0);
B(r0); B(g0); B(b0); B(0x34);
B(r1); B(g1); B(b1); B(0x34);
B(r2); B(g2); B(b2); B(0x34);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(0);
H(norm2); H(vert0);
H(vert1); H(vert2);
/*-----*/
B(r1); B(g1); B(b1); B(0x34);
B(r2); B(g2); B(b2); B(0x34);
B(r3); B(g3); B(b3); B(0x34);
B(u1a); B(v1a); H(cba);
B(u2a); B(v2a); H(tsb);
B(u3); B(v3); H(0);
H(norm3); H(vert3);

```

Colored Gouraud Texture Mesh

```
0x0000001f; DRV(0)|PRIM_TYPE(MESH|IIP|COL|TME)
```

```
H(num); H(0);
```

```
B(r0); B(g0); B(b0); B(0x34);
```

```
B(r1); B(g1); B(b1); B(0x34);
```

```
B(r2); B(g2); B(b2); B(0x34);
```

```
B(u0); B(v0); H(cba);
```

```
B(u1); B(v1); H(tsb);
```

```
B(u2); B(v2); H(0);
```

```
H(norm0); H(vert0);
```

```
H(norm1); H(vert1);
```

```
H(norm2); H(vert2);
```

```
/*-----*/
```

```
B(r1); B(g1); B(b1); B(0x34);
```

```
B(r2); B(g2); B(b2); B(0x34);
```

```
B(r3); B(g3); B(b3); B(0x34);
```

```
B(u1a); B(v1a); H(cba);
```

```
B(u2a); B(v2a); H(tsb);
```

```
B(u3); B(v3); H(0);
```

```
H(norm1); H(norm2);
```

```
H(norm3); H(vert3);
```

Without Light-source Calculation (Model Data without Normals)**Flat No-Texture Triangle**

```

0x00040048; DRV(LGT)|PRIM_TYPE(LMD|TRI); GsUNF3
B(r); B(g); B(b); B(0x20);
H(vert0); H(vert1);
H(vert2); H(0);

```

Gouraud No-Texture Triangle

```

0x0004004c; DRV(LGT)|PRIM_TYPE(LMD|TRI|IIP); GsUNG3
B(r0); B(g0); B(b0); B(0x30);
B(r1); B(g1); B(b1); B(0x30);
B(r2); B(g2); B(b2); B(0x30);
H(vert0); H(vert1);
H(vert2); H(0);

```

Flat Texture Triangle

```

0x00040049; DRV(LGT)|PRIM_TYPE(LMD|TRI|TME); GsUNFT3
B(r); B(g); B(b); B(0x24);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(vert0);
H(vert1); H(vert2);

```

Gouraud Texture Triangle

```

0x0004004d; DRV(LGT)|PRIM_TYPE(LMD|TRI|IIP|TME); GsUNGT3
B(r0); B(g0); B(b0); B(0x34);
B(r1); B(g1); B(b1); B(0x34);
B(r2); B(g2); B(b2); B(0x34);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(vert0);
H(vert1); H(vert2);

```

Flat No-Texture Quad

```

0x00040050; DRV(LGT)|PRIM_TYPE(LMD|QUAD); GsUNF4
B(r); B(g); B(b); B(0x28);
H(vert0); H(vert1);
H(vert2); H(vert3);

```

Gouraud No-Texture Quad

```

0x00040054; DRV(LGT)|PRIM_TYPE(LMD|QUAD|IIP); GsUNG4
B(r0); B(g0); B(b0); B(0x38);
B(r1); B(g1); B(b1); B(0x38);
B(r2); B(g2); B(b2); B(0x38);
B(r3); B(g3); B(b3); B(0x38);
H(vert0); H(vert1);
H(vert2); H(vert3);

```

Flat Texture Quad

```

0x00040051; DRV(LGT)|PRIM_TYPE(LMD|QUAD|TME); GsUNFT4
B(r); B(g); B(b); B(0x2c);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(vert0);
B(u3); B(v3); H(vert1);
H(vert2); H(vert3);

```

Gouraud Texture Quad

```

0x00040055; DRV(LGT)|PRIM_TYPE(LMD|QUAD|IIP|TME); GsUNGT4
B(r0); B(g0); B(b0); B(0x3c);
B(r1); B(g1); B(b1); B(0x3c);
B(r2); B(g2); B(b2); B(0x3c);
B(r3); B(g3); B(b3); B(0x3c);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(vert0);
B(u3); B(v3); H(vert1);
H(vert2); H(vert3);

```

Flat No-Texture Mesh

```

0x00040058; DRV(LGT)|PRIM_TYPE(LMD|MESH); GsUMNF3
H(num); H(0);
B(r2); B(g2); B(b2); B(0x20);
H(vert0); H(vert1);
H(vert2); H(0);
/*-----*/
B(r3); B(g3); B(b3); B(0x20);
H(vert3); H(0);

```


Gouraud No-Texture Mesh

```
0x0004005c; DRV(LGT)|PRIM_TYPE(LMD|MESH|IIP); GsUMNG3
```

```
H(num); H(0);
B(r0); B(g0); B(b0); B(0x30);
B(r1); B(g1); B(b1); B(0x30);
B(r2); B(g2); B(b2); B(0x30);
H(vert0); H(vert1);
H(vert2); H(0);
/*-----*/
B(r1); B(g1); B(b1); B(0x30);
B(r2); B(g2); B(b2); B(0x30);
B(r3); B(g3); B(b3); B(0x30);
H(vert3); H(0);
```

Flat Texture Mesh

```
0x00040059; DRV(LGT)|PRIM_TYPE(LMD|MESH|TME); GsUMNFT3
```

```
H(num); H(0);
B(r0); B(g0); B(b0); B(0x24);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(vert0);
H(vert1); H(vert2);
/*-----*/
B(r3); B(g3); B(b3); B(0x24);
B(u1a); B(v1a); H(cba);
B(u2a); B(v2a); H(tsb);
B(u3); B(v3); H(vert3);
```

Gouraud Texture Mesh

```
0x0004005d; DRV(LGT)|PRIM_TYPE(LMD|MESH|IIP|TME); GsUMNGT3
```

```
H(num); H(0);
B(r0); B(g0); B(b0); B(0x34);
B(r1); B(g1); B(b1); B(0x34);
B(r2); B(g2); B(b2); B(0x34);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(vert0);
H(vert1); H(vert2);
/*-----*/
B(r1); B(g1); B(b1); B(0x34);
B(r2); B(g3); B(b3); B(0x34);
B(r3); B(g3); B(b3); B(0x34);
B(u1a); B(v1a); H(cba);
B(u2a); B(v2a); H(tsb);
B(u3); B(v3); H(vert3);
```

Tiled Textures

Tiled Textures with Light-source Calculation

TUM: Tiling mask for the U coordinate of the texture pattern (5 bits)

TVM: Tiling mask for the V coordinate of the texture pattern (5 bits)

TUA: Upper address of U for tiling the texture pattern (5 bits)

TVA: Upper address of V for tiling the texture pattern (5 bits)

A packet that is used for tiled textures contains a repetition parameter at the beginning of the packet, and a reset parameter at the end of the packet. This allows tiled and non-tiled textures to coexist.

tum, tvn, tua, tva serve as parameters for calculating UV' from given UV values (u,v) using the following equation.

$$UV' = ((\sim(tum \ll 3) \& u) | (tua \ll 3), (\sim(tvm \ll 3) \& v) | (tva \ll 3));$$

In the following example, a texture window for tiling is set up in the texture page, with (x, y) representing the upper left corner, and (w, h) representing the width and height:

```
tum = (~ (w - 1) & 0x0ff) >> 3;
```

```
tvm = (~ (h - 1) & 0x0ff) >> 3;
```

```
tua = (x & 0x0ff) >> 3;
```

```
tva = (y & 0x0ff) >> 3;
```

At reset, all four parameters are set to zero.

Flat Texture Triangle

```
0x00000209; DRV(0) | PRIM_TYPE(TILE | TRI | TME); GsUTFT3
```

```
TUM(tum) | TVM(tvm) | TUA(tua) | TVA(tva) | 0xe2000000;
```

```
B(u0); B(v0); H(cba);
```

```
B(u1); B(v1); H(tsb);
```

```
B(u2); B(v2); H(0);
```

```
H(norm0); H(vert0);
```

```
H(vert1); H(vert2);
```

Gouraud Texture Triangle

```
0x0000020d; DRV(0) | PRIM_TYPE(TILE | TRI | IIP | TME); GsUTGT3
```

```
TUM(tum) | TVM(tvm) | TUA(tua) | TVA(tva) | 0xe2000000;
```

```
B(u0); B(v0); H(cba);
```

```
B(u1); B(v1); H(tsb);
```

```
B(u2); B(v2); H(0);
```

```
H(norm0); H(vert0);
```

```
H(norm1); H(vert1);
```

```
H(norm2); H(vert2);
```

Colored Flat Texture Triangle

```

0x0000020b; DRV(0)|PRIM_TYPE(TILE|TRI|COL|TME)
TUM(tum)|TVM(tvm)|TUA(tua)|TVA(tva)|0xe2000000;
B(r0); B(g0); B(b0); B(0x34);
B(r1); B(g1); B(b1); B(0x34);
B(r2); B(g2); B(b2); B(0x34);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(0);
H(norm0); H(vert0);
H(vert1); H(vert2);

```

Colored Gouraud Texture Triangle

```

0x0000020f; DRV(0)|PRIM_TYPE(TILE|TRI|IIP|COL|TME)
TUM(tum)|TVM(tvm)|TUA(tua)|TVA(tva)|0xe2000000;
B(r0); B(g0); B(b0); B(0x34);
B(r1); B(g1); B(b1); B(0x34);
B(r2); B(g2); B(b2); B(0x34);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(0);
H(norm0); H(vert0);
H(norm1); H(vert1);
H(norm2); H(vert2);

```

Flat Texture Quad

```

0x00000211; DRV(0)|PRIM_TYPE(TILE|QUAD|TME); GsUTFT4
TUM(tum)|TVM(tvm)|TUA(tua)|TVA(tva)|0xe2000000;
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(0);
B(u3); B(v3); H(norm0);
H(vert0); H(vert1);
H(vert2); H(vert3);

```

Gouraud Texture Quad

```

0x00000215; DRV(0)|PRIM_TYPE(TILE|QUAD|IIP|TME); GsUTGT4
TUM(tum)|TVM(tvm)|TUA(tua)|TVA(tva)|0xe2000000;
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(norm0);
B(u3); B(v3); H(vert0);
H(norm1); H(vert1);
H(norm2); H(vert2);
H(norm3); H(vert3);

```

Colored Flat Texture Quad

```

0x00000213; DRV(0)|PRIM_TYPE(TILE|QUAD|COL|TME)
TUM(tum)|TVM(tvm)|TUA(tua)|TVA(tva)|0xe2000000;
B(r0); B(g0); B(b0); B(0x3c);
B(r1); B(g1); B(b1); B(0x3c);
B(r2); B(g2); B(b2); B(0x3c);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(0);
B(u3); B(v3); H(norm0);
H(vert0); H(vert1);
H(vert2); H(vert3);

```

Colored Gouraud Texture Quad

```

0x00000217; DRV(0)|PRIM_TYPE(TILE|QUAD|IIP|COL|TME)
TUM(tum)|TVM(tvm)|TUA(tua)|TVA(tva)|0xe2000000;
B(r0); B(g0); B(b0); B(0x3c);
B(r1); B(g1); B(b1); B(0x3c);
B(r2); B(g2); B(b2); B(0x3c);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(norm0);
B(u3); B(v3); H(vert0);
H(norm1); H(vert1);
H(norm2); H(vert2);
H(norm3); H(vert3);

```

Flat Texture Mesh

```

0x00000219; DRV(0) | PRIM_TYPE(TILE | MESH | TME)
H(num); H(0);
TUM(tum) | TVM(tvm) | TUA(tua) | TVA(tva) | 0xe2000000;
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(0);
H(norm2); H(vert0);
H(vert1); H(vert2);
/*-----*/
TUM(tum) | TVM(tvm) | TUA(tua) | TVA(tva) | 0xe2000000;
B(u1a); B(v1a); H(cba);
B(u2a); B(v2a); H(tsb);
B(u3); B(v3); H(0);
H(norm3); H(vert3);

```

Gouraud Texture Mesh

```

0x0000021d; DRV(0) | PRIM_TYPE(TILE | MESH | IIP | TME)
H(num); H(0);
TUM(tum) | TVM(tvm) | TUA(tua) | TVA(tva) | 0xe2000000;
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(0);
H(norm0); H(vert0);
H(norm1); H(vert1);
H(norm2); H(vert2);
/*-----*/
TUM(tum) | TVM(tvm) | TUA(tua) | TVA(tva) | 0xe2000000;
B(u1a); B(v1a); H(cba);
B(u2a); B(v2a); H(tsb);
B(u3); B(v3); H(0);
H(norm1); H(norm2);
H(norm3); H(vert3);

```

Colored Flat Texture Mesh

```

0x0000021b; DRV(0)|PRIM_TYPE(TILE|MESH|COL|TME)
H(num); H(0);
TUM(tum)|TVM(tvm)|TUA(tua)|TVA(tva)|0xe2000000;
B(r0); B(g0); B(b0); B(0x34);
B(r1); B(g1); B(b1); B(0x34);
B(r2); B(g2); B(b2); B(0x34);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(0);
H(norm2); H(vert0);
H(vert1); H(vert2);
/*-----*/
TUM(tum)|TVM(tvm)|TUA(tua)|TVA(tva)|0xe2000000;
B(r1); B(g1); B(b1); B(0x34);
B(r2); B(g2); B(b2); B(0x34);
B(r3); B(g3); B(b3); B(0x34);
B(u1a); B(v1a); H(cba);
B(u2a); B(v2a); H(tsb);
B(u3); B(v3); H(0);
H(norm3); H(vert3);

```

Colored Gouraud Texture Mesh

```

0x0000021f; DRV(0)|PRIM_TYPE(TILE|MESH|IIP|COL|TME)
H(num); H(0);
TUM(tum)|TVM(tvm)|TUA(tua)|TVA(tva)|0xe2000000;
B(r0); B(g0); B(b0); B(0x34);
B(r1); B(g1); B(b1); B(0x34);
B(r2); B(g2); B(b2); B(0x34);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(0);
H(norm0); H(vert0);
H(norm1); H(vert1);
H(norm2); H(vert2);
/*-----*/
TUM(tum)|TVM(tvm)|TUA(tua)|TVA(tva)|0xe2000000;
B(r1); B(g1); B(b1); B(0x34);
B(r2); B(g2); B(b2); B(0x34);
B(r3); B(g3); B(b3); B(0x34);
B(u1a); B(v1a); H(cba);
B(u2a); B(v2a); H(tsb);
B(u3); B(v3); H(0);
H(norm1); H(norm2);
H(norm3); H(vert3);

```

Tiled Textures without Light-source Calculation**Flat Texture Triangle**

```

0x00040249; DRV(LGT)|PRIM_TYPE(TILE|LMD|TRI|TME)
TUM(tum)|TVM(tvm)|TUA(tua)|TVA(tva)|0xe2000000;
B(r); B(g); B(b); B(0x24);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(vert0);
H(vert1); H(vert2);

```

Gouraud Texture Triangle

```

0x0004024d; DRV(LGT)|PRIM_TYPE(TILE|LMD|TRI|IIP|TME)
TUM(tum)|TVM(tvm)|TUA(tua)|TVA(tva)|0xe2000000;
B(r0); B(g0); B(b0); B(0x34);
B(r1); B(g1); B(b1); B(0x34);
B(r2); B(g2); B(b2); B(0x34);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(vert0);
H(vert1); H(vert2);

```

Flat Texture Quad

```

0x00040251; DRV(LGT)|PRIM_TYPE(TILE|LMD|QUAD|TME)
TUM(tum)|TVM(tvm)|TUA(tua)|TVA(tva)|0xe2000000;
B(r); B(g); B(b); B(0x2c);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(vert0);
B(u3); B(v3); H(vert1);
H(vert2); H(vert3);

```

Gouraud Texture Quad

```

0x00040255; DRV(LGT)|PRIM_TYPE(TILE|LMD|QUAD|IIP|TME)
TUM(tum)|TVM(tvm)|TUA(tua)|TVA(tva)|0xe2000000;
B(r0); B(g0); B(b0); B(0x3c);
B(r1); B(g1); B(b1); B(0x3c);
B(r2); B(g2); B(b2); B(0x3c);
B(r3); B(g3); B(b3); B(0x3c);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(vert0);
B(u3); B(v3); H(vert1);
H(vert2); H(vert3);

```

Flat Texture Mesh

```

0x00040259; DRV(LGT)|PRIM_TYPE(TILE|LMD|MESH|TME)
H(num); H(0);
TUM(tum)|TVM(tvm)|TUA(tua)|TVA(tva)|0xe2000000;
B(r0); B(g0); B(b0); B(0x24);
B(u0); B(v0); H(cba);
B(u1); B(v1); H(tsb);
B(u2); B(v2); H(vert0);
H(vert1); H(vert2);
/*-----*/
TUM(tum)|TVM(tvm)|TUA(tua)|TVA(tva)|0xe2000000;
B(r3); B(g3); B(b3); B(0x24);
B(u1a); B(v1a); H(cba);
B(u2a); B(v2a); H(tsb);
B(u3); B(v3); H(vert3);

```


Gouraud Texture Mesh

```
0x0004025d; DRV(LGT)|PRIM_TYPE(TILE|LMD|MESH|IIP|TME)
```

```
H(num); H(0);
```

```
TUM(tum)|TVM(tvm)|TUA(tua)|TVA(tva)|0xe2000000;
```

```
B(r0); B(g0); B(b0); B(0x35);
```

```
B(r1); B(g1); B(b1); B(0x35);
```

```
B(r2); B(g2); B(b2); B(0x35);
```

```
B(u0); B(v0); H(cba);
```

```
B(u1); B(v1); H(tsb);
```

```
B(u2); B(v2); H(vert0);
```

```
H(vert1); H(vert2);
```

```
/*-----*/
```

```
TUM(tum)|TVM(tvm)|TUA(tua)|TVA(tva)|0xe2000000;
```

```
B(r1); B(g1); B(b1); B(0x35);
```

```
B(r2); B(g2); B(b2); B(0x35);
```

```
B(r3); B(g3); B(b3); B(0x35);
```

```
B(u1a); B(v1a); H(cba);
```

```
B(u2a); B(v2a); H(tsb);
```

```
B(u3); B(v3); H(vert3);
```

Preset model data**Flat No-Texture Triangle**

```
0x00040148; DRV(LGT)|PRIM_TYPE(PST|LMD|TRI); GsUPNF3
```

```
DMAtag;
```

```
B(r); B(g); B(b); B(0x20);
```

```
H(x0); H(y0);
```

```
H(x1); H(y1);
```

```
H(x2); H(y2);
```

```
DMAtag;
```

```
B(r); B(g); B(b); B(0x20);
```

```
H(x0); H(y0);
```

```
H(x1); H(y1);
```

```
H(x2); H(y2);
```

```
H(vert0); H(vert1);
```

```
H(vert2); H(0);
```

Gouraud No-Texture Triangle

```
0x0004014c; DRV(LGT)|PRIM_TYPE(PST|LMD|TRI|IIP); GsUPNG3
```

```
DMAtag;
```

```
B(r0); B(g0); B(b0); B(0x30);
```

```
H(x0); H(y0);
```

```
B(r1); B(g1); B(b1); B(0);
```

```
H(x1); H(y1);
```

```
B(r2); B(g2); B(b2); B(0);
```

```
H(x2); H(y2);
```

```
DMAtag;
```

```
B(r0); B(g0); B(b0); B(0x30);
```

```
H(x0); H(y0);
```

```
B(r1); B(g1); B(b1); B(0);
```

```
H(x1); H(y1);
```

```
B(r2); B(g2); B(b2); B(0);
```

```
H(x2); H(y2);
```

```
H(vert0); H(vert1);
```

```
H(vert2); H(0);
```

Flat Texture Triangle

0x00040149; DRV(LGT)|PRIM_TYPE(PST|LMD|TRI|TME); GsUPNFT3

DMAtag;

B(r); B(g); B(b); B(0x24);

H(x0); H(y0);

B(u0); B(v0); H(cba);

H(x1); H(y1);

B(u1); B(v1); H(tsb);

H(x2); H(y2);

B(u2); B(v2); H(0);

DMAtag;

B(r); B(g); B(b); B(0x24);

H(x0); H(y0);

B(u0); B(v0); H(cba);

H(x1); H(y1);

B(u1); B(v1); H(tsb);

H(x2); H(y2);

B(u2); B(v2); H(vert0);

H(vert1); H(vert2);

Gouraud Texture Triangle

0x0004014d; DRV(LGT)|PRIM_TYPE(PST|LMD|TRI|IIP|TME); GsUNGT3

DMAtag;

B(r0); B(g0); B(b0); B(0x34);

H(x0); H(y0);

B(u0); B(v0); H(cba);

B(r1); B(g1); B(b1); B(0);

H(x1); H(y1);

B(u1); B(v1); H(tsb);

B(r2); B(g2); B(b2); B(0);

H(x2); H(y2);

B(u2); B(v2); H(0);

DMAtag;

B(r0); B(g0); B(b0); B(0x34);

H(x0); H(y0);

B(u0); B(v0); H(cba);

B(r1); B(g1); B(b1); B(0);

H(x1); H(y1);

B(u1); B(v1); H(tsb);

B(r2); B(g2); B(b2); B(0);

H(x2); H(y2);

B(u2); B(v2); H(vert0);

H(vert1); H(vert2);

Flat No-Texture Quad

0x00040150; DRV(LGT)|PRIM_TYPE(PST|LMD|QUAD); GsUPNF4

DMAtag;

B(r); B(g); B(b); B(0x28);

H(x0); H(y0);

H(x1); H(y1);

H(x2); H(y2);

H(x3); H(y3);

DMAtag;

B(r); B(g); B(b); B(0x28);

H(x0); H(y0);

H(x1); H(y1);

H(x2); H(y2);

H(x3); H(y3);

H(vert0); H(vert1);

H(vert2); H(vert3);

Gouraud No-Texture Quad

0x00040154; DRV(LGT)|PRIM_TYPE(PST|LMD|QUAD|IIP); GsUPNG4

DMAtag;

B(r0); B(g0); B(b0); B(0x38);

H(x0); H(y0);

B(r1); B(g1); B(b1); B(0);

H(x1); H(y1);

B(r2); B(g2); B(b2); B(0);

H(x2); H(y2);

B(r3); B(g3); B(b3); B(0);

H(x3); H(y3);

DMAtag;

B(r0); B(g0); B(b0); B(0x38);

H(x0); H(y0);

B(r1); B(g1); B(b1); B(0);

H(x1); H(y1);

B(r2); B(g2); B(b2); B(0);

H(x2); H(y2);

B(r3); B(g3); B(b3); B(0);

H(x3); H(y3);

H(vert0); H(vert1);

H(vert2); H(vert3);

Flat Texture Quad

```
0x00040151; DRV(LGT)|PRIM_TYPE(PST|LMD|QUAD|TME); GsUPNFT4
```

```
DMAtag;
```

```
B(r); B(g); B(b); B(0x2c);
```

```
H(x0); H(y0);
```

```
B(u0); B(v0); H(cba);
```

```
H(x1); H(y1);
```

```
B(u1); B(v1); H(tsb);
```

```
H(x2); H(y2);
```

```
B(u2); B(v2); H(0);
```

```
H(x3); H(y3);
```

```
B(u3); B(v3); H(0);
```

```
DMAtag;
```

```
B(r); B(g); B(b); B(0x2c);
```

```
H(x0); H(y0);
```

```
B(u0); B(v0); H(cba);
```

```
H(x1); H(y1);
```

```
B(u1); B(v1); H(tsb);
```

```
H(x2); H(y2);
```

```
B(u2); B(v2); H(0);
```

```
H(x3); H(y3);
```

```
B(u3); B(v3); H(0);
```

```
H(vert0); H(vert1);
```

```
H(vert2); H(vert3);
```

Gouraud Texture Quad

```
0x00040155; DRV(LGT)|PRIM_TYPE(PST|LMD|QUAD|IIP|TME); GsUPNGT4
```

```
DMAtag;
```

```
B(r0); B(g0); B(b0); B(0x3c);
```

```
H(x0); H(y0);
```

```
B(u0); B(v0); H(cba);
```

```
B(r1); B(g1); B(b1); B(0);
```

```
H(x1); H(y1);
```

```
B(u1); B(v1); H(tsb);
```

```
B(r2); B(g2); B(b2); B(0);
```

```
H(x2); H(y2);
```

```
B(u2); B(v2); H(0);
```

```
B(r3); B(g3); B(b3); B(0);
```

```
H(x3); H(y3);
```

```
B(u3); B(v3); H(0)
```

```
DMAtag;
```

```
B(r0); B(g0); B(b0); B(0x3c);
```

```
H(x0); H(y0);
```

```
B(u0); B(v0); H(cba);
```

```
B(r1); B(g1); B(b1); B(0);
```

```
H(x1); H(y1);
```

```
B(u1); B(v1); H(tsb);
```

```
B(r2); B(g2); B(b2); B(0);
```

```
H(x2); H(y2);
```

```
B(u2); B(v2); H(0);
```

```
B(r3); B(g3); B(b3); B(0);
```

```
H(x3); H(y3);
```

```
B(u3); B(v3); H(0)
```

```
H(vert0); H(vert1);
```

```
H(vert2); H(vert3);
```

Flat No-Texture Mesh

```
0x00040158; DRV(LGT) | PRIM_TYPE(PST | LMD | MESH)
```

```
H(num); H(0);
```

```
DMAtag;
```

```
B(r2); B(g2); B(b2); B(0x20);
```

```
H(x0); H(y0);
```

```
H(x1); H(y1);
```

```
H(x2); H(y2);
```

```
DMAtag;
```

```
B(r2); B(g2); B(b2); B(0x20);
```

```
H(x0); H(y0);
```

```
H(x1); H(y1);
```

```
H(x2); H(y2);
```

```
H(vert0); H(vert1);
```

```
H(vert2); H(0);
```

```
/*-----*/
```

```
DMAtag;
```

```
B(r3); B(g3); B(b3); B(0x20);
```

```
H(x1); H(y1);
```

```
H(x2); H(y2);
```

```
H(x3); H(y3);
```

```
DMAtag;
```

```
B(r3); B(g3); B(b3); B(0x20);
```

```
H(x1); H(y1);
```

```
H(x2); H(y2);
```

```
H(x3); H(y3);
```

```
H(vert3); H(0);
```

Gouraud No-Texture Mesh

```
0x0004015c; DRV(LGT)|PRIM_TYPE(PST|LMD|MESH|IIP)
```

```
H(num); H(0);
```

```
DMAtag;
```

```
B(r0); B(g0); B(b0); B(0x30);
```

```
H(x0); H(y0);
```

```
B(r1); B(g1); B(b1); B(0);
```

```
H(x1); H(y1);
```

```
B(r2); B(g2); B(b2); B(0);
```

```
H(x2); H(y2);
```

```
DMAtag;
```

```
B(r0); B(g0); B(b0); B(0x30);
```

```
H(x0); H(y0);
```

```
B(r1); B(g1); B(b1); B(0);
```

```
H(x1); H(y1);
```

```
B(r2); B(g2); B(b2); B(0);
```

```
H(x2); H(y2);
```

```
H(vert0); H(vert1);
```

```
H(vert2); H(0);
```

```
/*-----*/
```

```
DMAtag;
```

```
B(r1); B(g1); B(b1); B(0x30);
```

```
H(x1); H(y1);
```

```
B(r2); B(g2); B(b2); B(0);
```

```
H(x2); H(y2);
```

```
B(r3); B(g3); B(b3); B(0);
```

```
H(x3); H(y3);
```

```
DMAtag;
```

```
B(r1); B(g1); B(b1); B(0x30);
```

```
H(x1); H(y1);
```

```
B(r2); B(g2); B(b2); B(0);
```

```
H(x2); H(y2);
```

```
B(r3); B(g3); B(b3); B(0);
```

```
H(x3); H(y3);
```

```
H(vert3); H(0);
```


Flat Texture Mesh

```
0x00040159; DRV(LGT) | PRIM_TYPE(PST | LMD | MESH | TME)
```

```
H(num); H(0);
```

```
DMAtag;
```

```
B(r0); B(g0); B(b0); B(0x24);
```

```
H(x0); H(y0);
```

```
B(u0); B(v0); H(cba);
```

```
H(x1); H(y1);
```

```
B(u1); B(v1); H(tsb);
```

```
H(x2); H(y2);
```

```
B(u2); B(v2); H(0);
```

```
DMAtag;
```

```
B(r0); B(g0); B(b0); B(0x24);
```

```
H(x0); H(y0);
```

```
B(u0); B(v0); H(cba);
```

```
H(x1); H(y1);
```

```
B(u1); B(v1); H(tsb);
```

```
H(x2); H(y2);
```

```
B(u2); B(v2); H(vert0);
```

```
H(vert1); H(vert2);
```

```
/*-----*/
```

```
DMAtag;
```

```
B(r0); B(g0); B(b0); B(0x24);
```

```
H(x1); H(y1);
```

```
B(u0); B(v0); H(cba);
```

```
H(x2); H(y2);
```

```
B(u1); B(v1); H(tsb);
```

```
H(x3); H(y3);
```

```
B(u2); B(v2); H(0);
```

```
DMAtag;
```

```
B(r0); B(g0); B(b0); B(0x24);
```

```
H(x1); H(y1);
```

```
B(u0); B(v0); H(cba);
```

```
H(x2); H(y2);
```

```
B(u1); B(v1); H(tsb);
```

```
H(x3); H(y3);
```

```
B(u2); B(v2); H(vert3);
```

Gouraud Texture Mesh

0x0004015d; DRV(LGT)|PRIM_TYPE(PST|LMD|MESH|IIP|TME)

```

H(num); H(0);
DMAtag;
B(r0); B(g0); B(b0); B(0x34);
H(x0); H(y0);
B(u0); B(v0); H(cba);
B(r1); B(g1); B(b1); B(0);
H(x1); H(y1);
B(u1); B(v1); H(tsb);
B(r2); B(g2); B(b2); B(0);
H(x2); H(y2);
B(u2); B(v2); H(0);
DMAtag;
B(r0); B(g0); B(b0); B(0x34);
H(x0); H(y0);
B(u0); B(v0); H(cba);
B(r1); B(g1); B(b1); B(0);
H(x1); H(y1);
B(u1); B(v1); H(tsb);
B(r2); B(g2); B(b2); B(0);
H(x2); H(y2);
B(u2); B(v2); H(vert0);
H(vert1); H(vert2);
/*-----*/
DMAtag;
B(r0); B(g0); B(b0); B(0x34);
H(x1); H(y1);
B(u0); B(v0); H(cba);
B(r1); B(g1); B(b1); B(0);
H(x2); H(y2);
B(u1); B(v1); H(tsb);
B(r2); B(g2); B(b2); B(0);
H(x3); H(y3);
B(u2); B(v2); H(0);
DMAtag;
B(r0); B(g0); B(b0); B(0x34);
H(x1); H(y1);
B(u0); B(v0); H(cba);
B(r1); B(g1); B(b1); B(0);
H(x2); H(y2);
B(u1); B(v1); H(tsb);
B(r2); B(g2); B(b2); B(0);
H(x3); H(y3);
B(u2); B(v2); H(vert3);

```

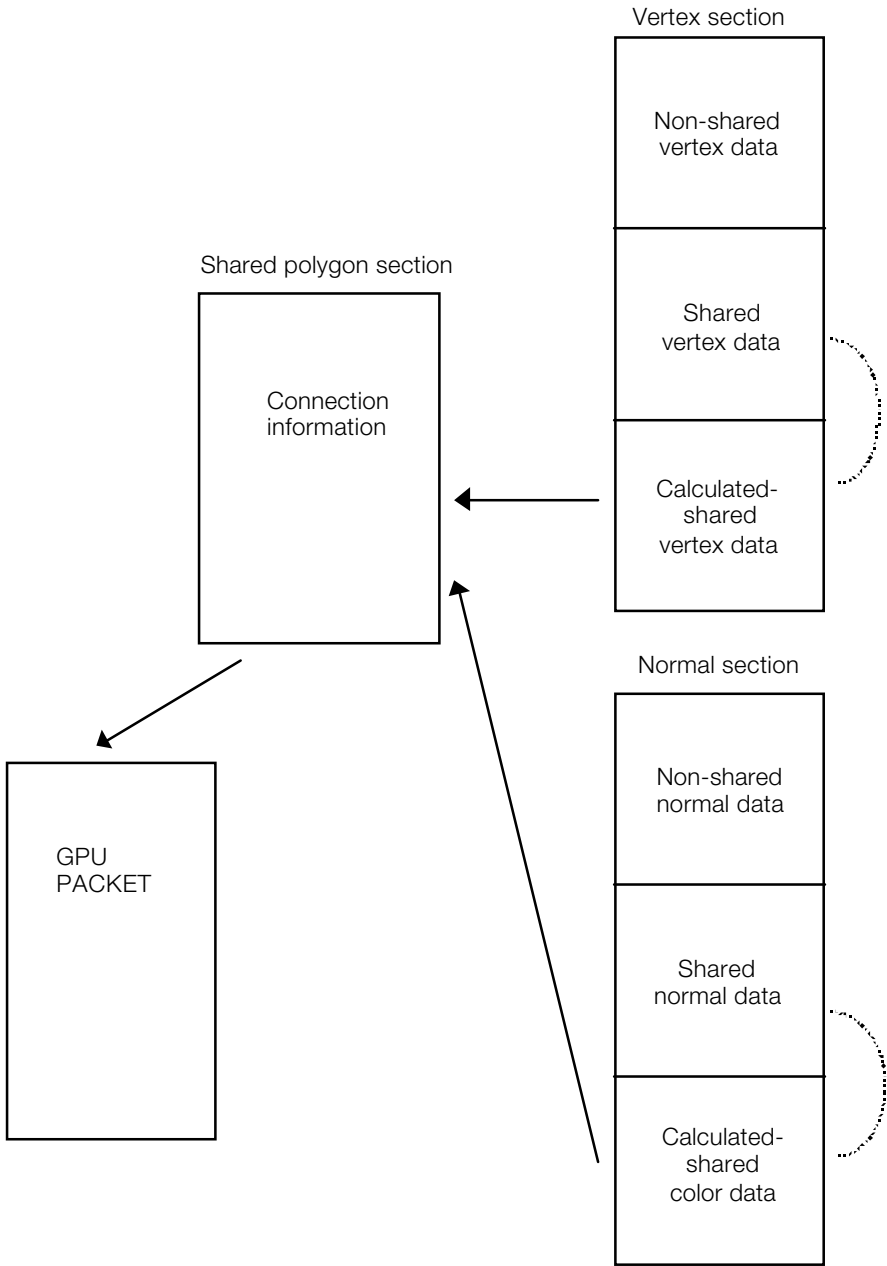

The format of the connection data, which a shared driver refers to, is the same as the format for a non-shared polygon PACKET. The format of the calculated area, which a shared driver refers to, is shown below:

VERTEX
H(vx); H(vy);
H(otz); H(p);

NORMAL
H(r); H(g);
H(b); H(0);

Processing Flow for Shared Polygons

Figure 2-96: Shared Polygon Processing Flow



The arrows with the dotted line indicate the processing flow of the PRE-CALCULATION driver. Vertex and normal calculations are performed for each coordinate.

The arrows with the solid line indicate the processing flow of the shared driver. Pre-calculated vertex data and color data are used to create a GPU PACKET. The format of the connection data for the shared driver is the same as that for an independent PACKET and is identified by the type field.

Image Primitive Section (Category 2)

The HMD format is able to represent image data as a primitive. This allows HMD to provide integrated management of modeling data, image data, and animation data.

Of course, image data can be set up separately without including it in HMD data. For example, TIM can be used to represent image data. Conversely, HMD data can be created which contains only image data as well.

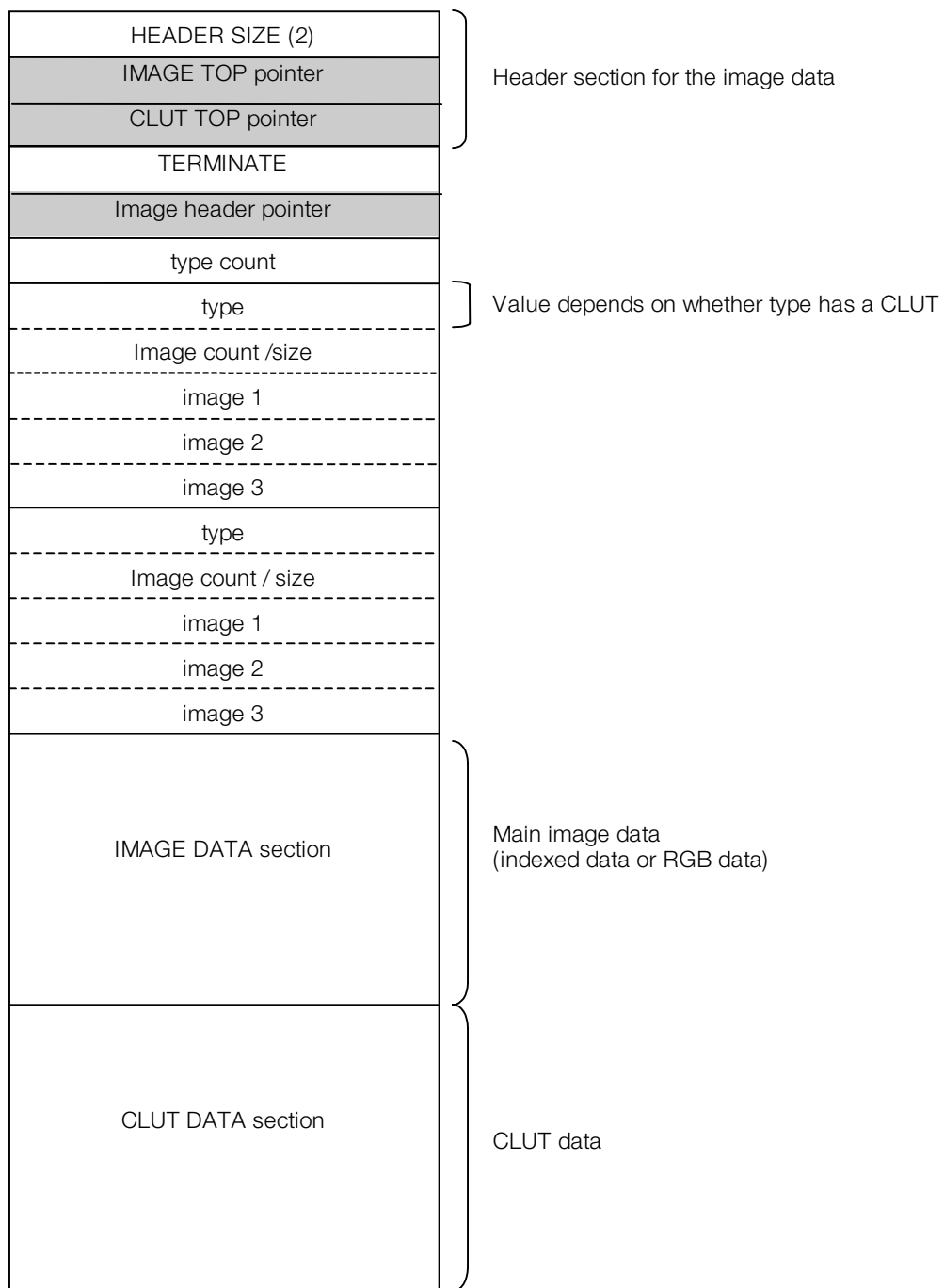


Figure 2-97: Parameter Memory Area of Image Primitive Driver

prmtop
tag(OT)
shift(OT)
offset(OT)
OUTP(packet area)
Image header pointer
CLUT top pointer

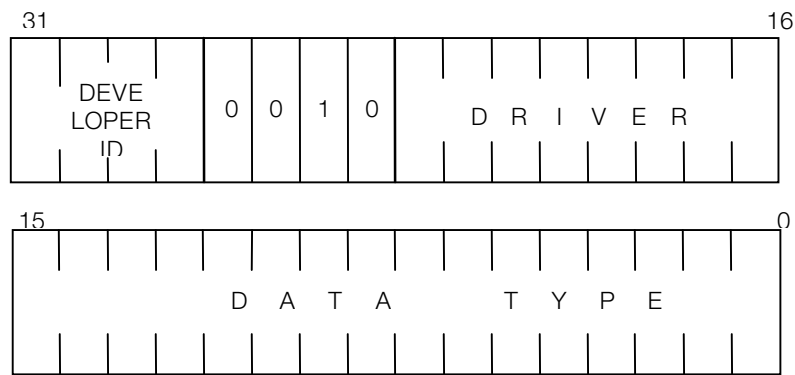
Parameter Settings

Behavior of the image primitive driver

Image primitives are linked to the PRE-PROCESS at the beginning of HMD's coordinate section. A VRAM transfer function is called during the SCAN operation. A NULL driver (type=0x00000000, a primitive driver that does not do anything) can be set in the type field once the transfer is complete so that the transfer to VRAM will be performed only once.

Image Type

Figure 2-98: Image Primitive Type Field



DRIVER: Currently all 0's

DATA TYPE: Indicates type of data
 0: No CLUT
 1: CLUT

Non-CLUT Primitive

```
0x02000000; DEV_ID(SCE) | CTG(CTG_IMAGE) | DRV(0) | PRIM_TYPE(NOCLUT); GsUIMG0
H(dx); H(dy);
H(w); H(h);
image_idx;
```

Primitive with CLUT

```
0x02000001; DEV_ID(SCE) | CTG(CTG_IMAGE) | DRV(0) | PRIM_TYPE(WITHCLUT); GsUIMG1
H(dx); H(dy);
H(w); H(h);
image_idx;
H(dx); H(dy);
H(w); H(h);
clut_idx;
```

Run-time Environment for Image Primitive Driver

The image primitive driver is called with the following environment.

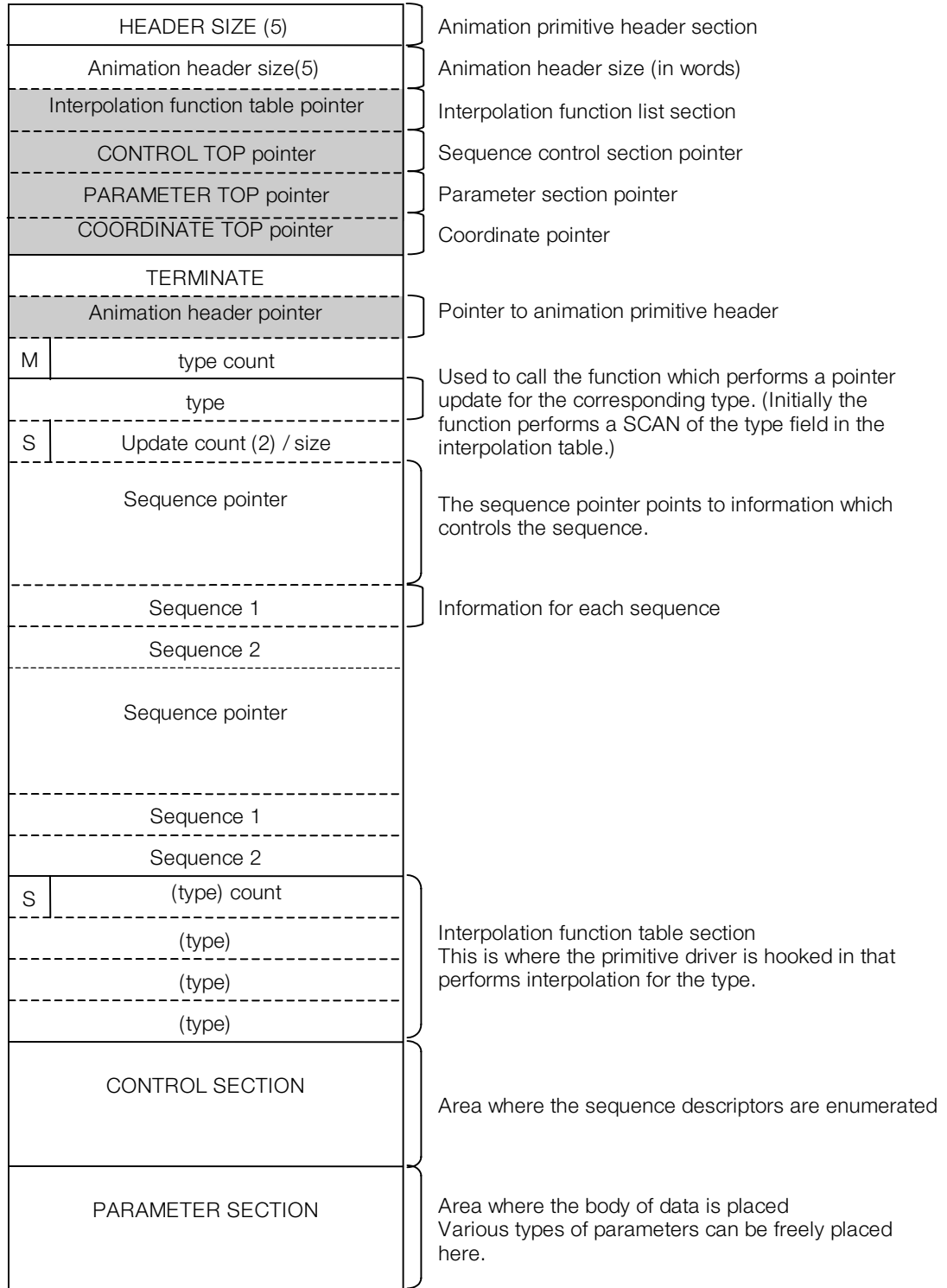
The following variables are copied to the parameter memory area.

Animation Primitive Section (Category 3)

An animation primitive section can be divided into the following five subsections:

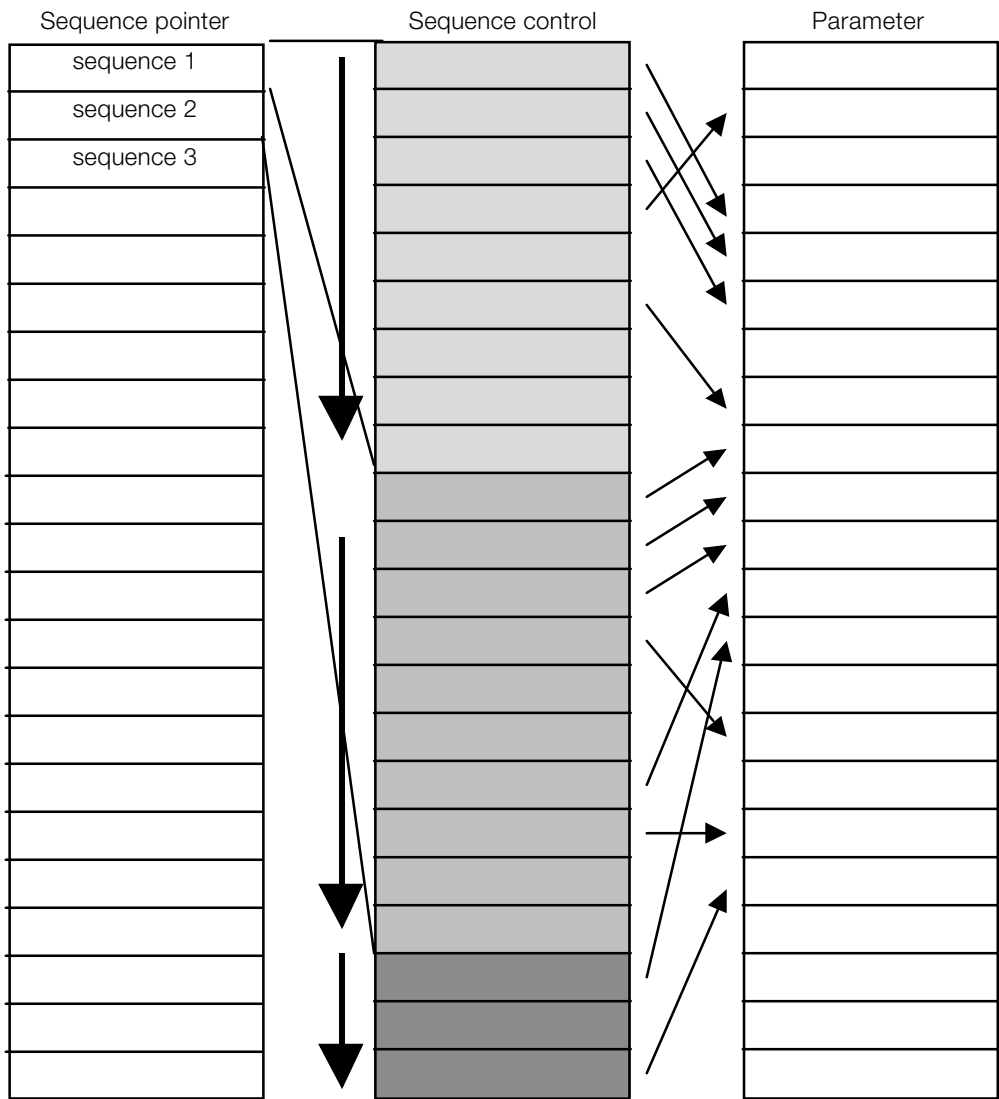
1. Animation primitive header section
2. Sequence pointer section
3. Interpolation function table section
4. Sequence control section
5. Parameter section

Figure 2-99: Animation Structure



Relationships Between Sections in Animation Data

Figure 2-100: Diagram Showing Correlation of All Animation Sections



Animation Header Section

The animation header section must contain a pointer to the interpolation function table, a sequence control section, and a pointer to the start of the parameter section.

Pointers to the sections, which need to be updated, are placed in the corresponding low-order address. For example, when a COORDINATE is to be rewritten, COORDINATE TOP is saved. If a vertex is to be rewritten, VERTEX TOP is saved.

Sequence Pointer Section

The sequence pointer section contains the sequence pointer and sequence information for each sequence. The update index contains separate information for the upper 8 bits and the lower 24 bits.

Interpolation Function Table Section

The interpolation function table section contains the type fields for the interpolation functions referred to by the sequence descriptors. The type fields are stored in an array and the interpolation method to be used is determined from the index of this array. The GsScanAnim() function must first be used to extract the type field and perform a SCAN to obtain the starting address of the actual primitive driver.

Sequence Control Section

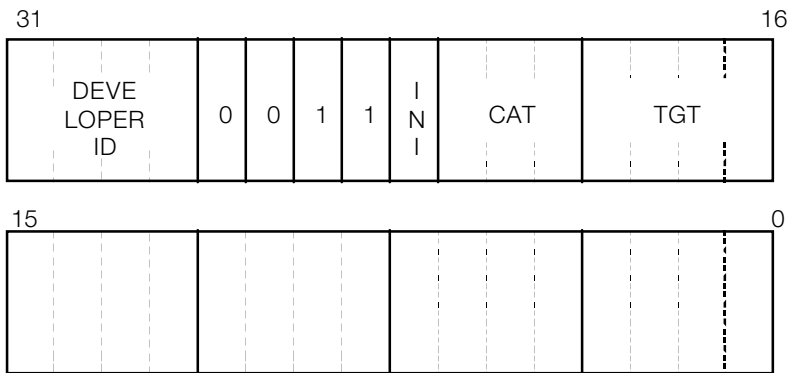
A sequence is represented as an array of sequence descriptors in the sequence control section. A sequence descriptor accesses the interpolation function table section and the parameter section using an index in order to specify the interpolation method that will be used between a key frame and the parameter of a key frame.

Parameter Section

A sequence descriptor accesses an interpolation function and an interpolation parameter using an index. The parameter section contains an array of interpolation parameters for various formats and interpolation functions.

Animation Type

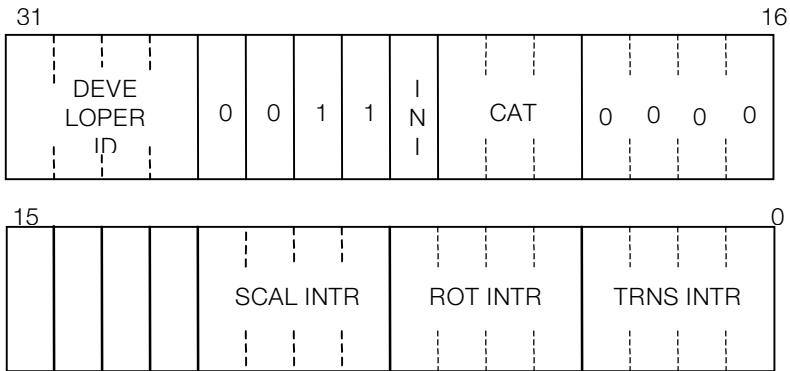
Figure 2-101: Animation Primitive Type Field



- INI: Determines whether a SCAN of the interpolation table section will be performed
0: Do not perform SCAN (SCAN already performed)
1: Perform SCAN for interpolation function table
- CAT: Indicates category of the frame update driver
0: Standard frame update driver (performs frame updates and calls interpolation function)
- TGT: Update target
0: Update the COORDINATE section
1: General update type

When TGT=0 (Update COORDINATE)

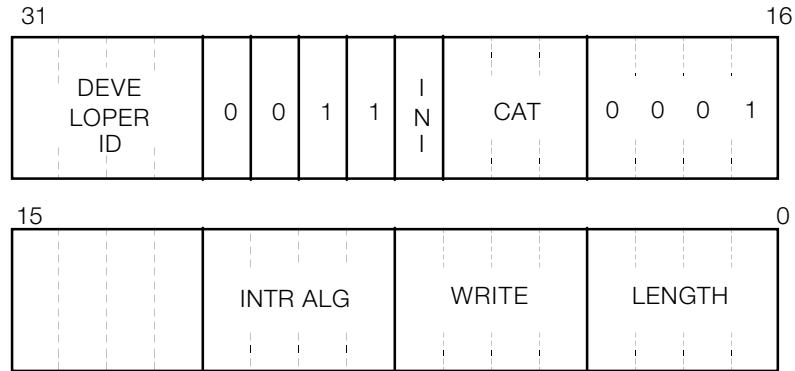
Figure 2-102: Type Field when TGT=0



- SCAL INTR Specifies the interpolation method when scaling
- 0: Do not interpolate
 - 1: LINEAR
 - 2: BEZIER
 - 3: B-SPRINE
 - 4: beta-SPRINE
 - 9: LINEAR (one parameter)
 - A: BEZIER (one parameter)
 - B: B-SPRINE (one parameter)
- ROT INTR Specifies the interpolation method when rotating
- 0: Do not interpolate
 - 1: LINEAR
 - 2: BEZIER
 - 3: B-SPRINE
- TRNS INTR Specifies the interpolation method when translating
- 0: Do not interpolate
 - 1: LINEAR
 - 2: BEZIER
 - 3: B-SPRINE
 - 9: LINEAR(short)
 - A: BEZIER(short)
 - B: B-SPRINE(short)

When TGT=1 (General Purpose Update)

Figure 2-103: Type Field when TGT=1



LENGTH: 0: 32bit
 1: 16bit
 2: 8 bit

WRITE: Specify areas to update.
 This field has 4bits, therefore, up to 4 units are allowed to update.
 In the examples below, areas to update are colored with gray.

Figure 2-104: LENGTH=16 bit, WRITE=0x1

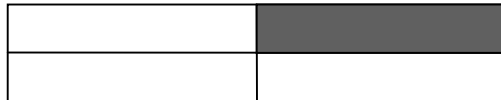


Figure 2-105: LENGTH=16bit, WRITE=0x7



Figure 2-106: LENGTH=8bit, WRITE=0x1

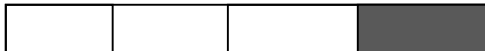


Figure 2-107: LENGTH=8bit, WRITE=0x7



INTR ALG: Interpolation algorithm

1. Linear
2. Bezier
3. B-Spline

Sequence Header

The sequence header contains information that is used to manage the various sequences.

Figure 2-108: Sequence Header

Sequence pointer		
TRAVELING	STREAM ID	Start IDX
-	STREAM ID	Start IDX

Sequence Pointer

The sequence pointer holds sequence information during playback. When multiple sequences are set up to be played back simultaneously, a sequence pointer is assigned to each playback sequence. The programmer uses the sequence pointers to control the real time playback of sequences. The members of the sequence pointers are continuously referenced by the interpolation primitive driver, which provides instantaneous response.

The figure below shows the data format for a sequence pointer. The areas, which have been written with HMD data, are highlighted. The areas without highlighting are work areas used by the program for replacing values and controlling the sequence.

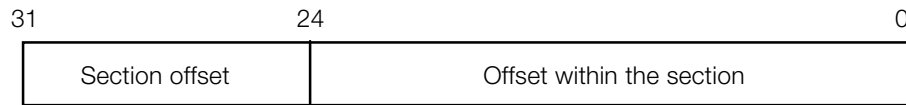
Figure 2-109: Sequence Pointer

Update index		
Sequence count / size		
A FRAME	INTR IDX	
SRC INTR IDX	SPEED	STREAM ID
TFRAME	RFRAME	
TCTR IDX	CTR IDX	
TRAVELING	START SID	START IDX

Update Index

The update index contains the target address to be updated by the sequence. The upper 8 bits hold the section offset, and the lower 24 bits hold the offset within the section.

Figure 2-110: Setting Update Location



The primitive header contains a list of starting addresses, and the "section offset" is an index into that list specifying which section will be updated. For example, if the index is 0 the interpolation function table section will be used, and if the index is 1 the CONTROL section will be used.

The "offset within the section" is an index which points to the position within the section specified by the "section offset" that will be updated. The offset is specified in words. For example, if the second coordinate is to be rewritten, the offset would be $\text{sizeof}(\text{GsCOORDUNIT}) / 4 + 1$. The +1 is included because the word at the beginning of the coordinate section is included in the coordinate count.

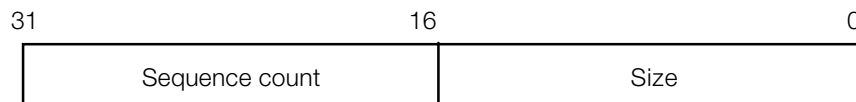
With some types of animation, vertices or normals may be updated instead of coordinates. In such cases, a pointer to the start of the section to be updated is added to the animation header, and the pointer is specified from the section offset of the update index. The position of the data to be updated can then be specified with the offset within the section. The type of data to be updated is identified with the type field.

Sequence Count/Size

The upper 16 bits hold the sequence count and the lower 16 bits hold the size. Sequence count is the number of sequences that are managed by the sequence pointer.

Size is the number of words remaining until the next sequence pointer.

Figure 2-111: Sequence Count and Size



INTR IDX: The value in this field is an index into the key frame containing parameters to be used after interpolation of the current frame. The application can change this value if the sequence is to be dynamically switched.

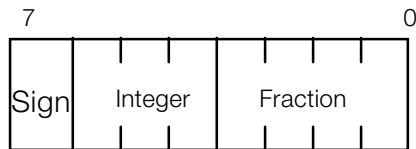
If this field contains the value 0xffff, updates will not be performed.

A FRAME: The total frame count of the sequence. Setting AFRAME to 0 can stop the sequence. If ENDBit is detected in the sequence control descriptor, AFRAME will be automatically set to 0. If the value of AFRAME is set to 0xffff, the total frame count will be infinity and the value will not be decremented.

SRC INTRIDX: Holds the work area to be assigned to INTR IDX.

SPEED: Specifies the update speed for the sequence pointer.

Figure 2-112: Fixed Point Format Used in SPEED Specification



SPEED is a two's complement fixed-point number, with 1 bit for the sign, 3 bits for the integer part, and 4 bits for the fraction. If the value of SPEED is negative, the sequence pointer is decremented when it is updated, and animations will be played back in reverse.

- If the sign bit is 1, the sequence pointer is decremented, resulting in the animation being played back in reverse.
- The integer part has three bits, so animation playback can be sped up by a factor of 7.
- The fractional part has four bits, so animation playback can be slowed down to 1/15.
- If all 8 bits are 0, the update speed of the sequence pointer is set to the previous update rate. Note that operation may be unpredictable if 0 is specified as the initial value.

TFRAME: The time between key frames for the data currently playing. This value is specified as a frame count and is updated automatically when the key frame is switched. TFRAME is represented as a fixed-point decimal integer, where the value 0x10 represents one frame.

RFRAME: The time between the motion currently playing and the original key frame. This value is specified as a frame count and is re-read when the key frame is switched. RFRAME is represented as a fixed-point decimal integer, where the value 0x10 represents one frame.

STREAM ID: Used for multiply-defined sequences. Sequence jumps take place only when STREAM IDs match. The STREAM ID can be changed dynamically during execution. This allows the efficient use of memory during interactive animation.

The STREAM ID has 7 valid bits, ranging from 0 to 127.

STREAM ID 0 has special meaning. This value matches to any SID. We do not recommend to use STREAM ID 127 as a condition of JUMP sequence. Then, STREAM ID 127 is possible to use with opposite meaning to STREAM ID 0.

TCTR IDX: Holds the index of the target key frame (among the two key frames used for interpolation). The target key frame is the key frame that is in the direction of convergence. The index is automatically updated when the key frame is switched. To specify the start of a sequence, the index of the starting sequence descriptor should be placed in TCTR IDX and RFRAME should be set to 0.

CTR IDX: Holds the index to the original key frame (among the two key frames used for interpolation). The original key frame is the key frame that has already passed. The index is automatically updated when the key frame is switched.

START IDX: Holds the starting index for a sequence. When it is desired to start a sequence, START IDX should be placed in TCTR IDX, START SID should be placed in SID, and RFRAME should be set to 0.

START IDX is also can be used as index to refer to control descriptor for sequence specific parameters. In this case, START IDX must not identical to starting index of sequence, the next sequence management data is allowed to use.

START SID: Holds the stream ID of the sequence to be started.

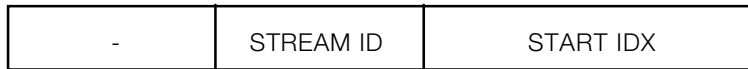
TRAVELING: Cleared to 0 when the key frame is switched. The programmer can use this variable freely. For example, to determine if the current interpolation is finished, a non-zero value can be entered in this field during key-frame interpolation. When the current interpolation completes, this field will be cleared to 0.

Sequence Management Data

A single sequence pointer can be used to define multiple sequences, and the sequences can be played back selectively. In these cases, the selected sequence data is added after the last sequence pointer.

This information is referred to as sequence management data. It consists of the final word of the sequence pointer with TRAVELING omitted.

Figure 2-113: Sequence Management Data



Sequence Index

This field holds the index of the sequence control descriptor at the starting point of the sequence. The application can start a sequence by copying the sequence index into the sequence pointer's TCTR IDX and setting RFRAME to 0.

STREAM ID

Holds the STREAM ID for the starting sequence. The application can start a sequence by copying this value into the STREAM ID of the sequence pointer.

Interpolation Functions Table Section

The interpolation method for key frames can be varied even within a single sequence. The interpolation method is specified with an index into a type array. All sequence descriptors except jumps have this index, which can be used to specify the interpolation method.

The interpolation function table section is an area that contains this type array.

The entry in the type array of the interpolation function table is converted beforehand to the starting address of the primitive driver for that type. This operation is performed by the SCAN function GsScanAnim(). When a SCAN is required, the INI bit of TYPE should be set to 1.

The SCAN function for the interpolation function table is called when the SCAN operation for the HMD data is performed. After the SCAN completes, the type is updated with the starting address of the frame update driver function and the INI bit is set to 0.

The first word of the interpolation function table section contains the number of types. The uppermost bit is used as a flag indicating whether a SCAN operation (GsScanAnim()) was performed. If the flag is set to 1, a SCAN has not been performed. 0 indicates that SCAN has been performed.

Sequence Control Section

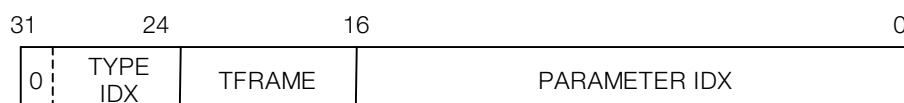
The actual sequence is represented in the sequence control section as a list. One element of the list is defined as the sequence descriptor. Sequence descriptors can be classified as one of two types. One type is the descriptor for a sequential sequence. The other type is the descriptor for a branching sequence. The uppermost bit of the sequence descriptor determines the type.

MSB: bit31 Identifier that indicates whether or not the sequence control descriptor points to a normal key frame.

0: PARAMETER IDX

1: SEQUENCE IDX

Figure 2-114: Sequence Descriptor (Normal)



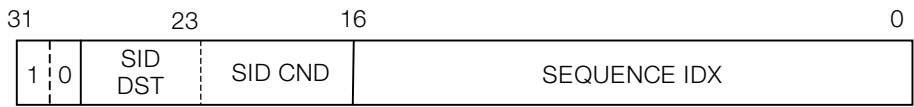
- TYPE IDX:

This field is an index into the interpolation function table, which specifies the interpolation function to be used. Since seven bits are available. Up to 128 interpolation functions can be accessed.
- TFRAME:

The frame number of the next sequence descriptor (in integer format). When this value is placed in the TFRAME member of the sequence pointer, it must be converted to fixed-point decimal format (with base 0x10).
- PARAMETER IDX:

Index to parameter data for the key frame referred to by the sequence descriptor.

Figure 2-115: Sequence Descriptor (Jump)



STREAM ID @bit16-29 The STREAM ID can be used to define multiple sequence links in a single sequence. STREAM IDs are divided into a SID DST (upper 7 bits) and a SID CND (lower 7 bits). SID DST specifies the STREAM ID for the destination of the jump while SID CND determines whether a jump will be performed when the STREAM IDs matches.

SID CND 0 matches to any current stream ID. In this case, SID DST will not be updated.

SID 127 is reserved to use as an ID that never matches to any stream ID except 0. The Stream ID is updated according to the following rules.

DST = 0 and CND = 0: Unconditional jump. The Stream ID is not updated.

DST = 0 and CND != 0: Jump if the current SID matches CND.
The Stream ID is set to 0.

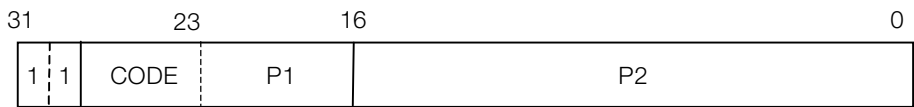
DST != 0 and CND = 0: Unconditional jump. The Stream ID is set to DST.

DST != 0 and CND != 0: Jump if the current SID matches CND.
The stream ID is set to DST.

SID 127 is defined to not match any non-zero stream ID.

SEQUENCE IDX: Contains the index of the control descriptor for the destination of the jump.

Figure 2-116: Sequence Descriptor (Control)



The parameters P1 and P2 can take on different values depending on CODE.

CODE: 0x01: END

If P1 matches the current STREAM ID, the sequence is halted.

CODE: 0x02: WORK

This indicates work area for each sequence pointer that is required by BSPLINE interpolation.

P1=127 Fixed

P2: Offset in parameter section indicates work area (in words).

Notes Regarding Switching of Interpolation Functions During a Sequence

A single interpolation function can be defined for each sequence control descriptor so that the interpolation function can be switched for each key frame. However, the parameters of the interpolation function must have the same format. Thus, if the interpolation function is switched, the program must ensure that the parameter format for the SRC FRAME and the DST FRAME match.

Example:

KEY 0 (parameter format A) TFRAME = 0
 KEY 1 (parameter format B) TFRAME = 30

Interpolation cannot be performed here since the SRC FRAME and the DST FRAME has different parameter formats. In this case, a sequence control descriptor is added to unify the formats.

KEY 0 (parameter format A) TFRAME = 0
 KEY 00 (parameter format B) TFRAME = 0
 KEY 1 (parameter format B) TFRAME = 30

KEY00 performs parameter format conversion from A to B. The TFRAME of the descriptor must be 0 to perform this conversion. Note that the sequence will jump if there is a discontinuity between KEY0 and KEY00.

Behavior of Interpolation Driver When TFRAME is 0

Even if TFRAME is 0, interpolation driver is called. Thus, any interpolation driver should return without interpolation if TFRAME is 0. It is possible to use TFRAME=0 to change internal status of interpolation driver. For example, first 3 control points for spline function are written as key frames with TFRAME=0.

While TFRAME is 0 or return value of interpolation driver is 1, interpolation driver is called continuously, and RFRAME is not updated.

Parameter Section

The parameter section contains the actual parameters and that is referenced by an index in the sequence control section. The parameters in this section can take on various forms (for example, VECTORS and MATRIXes). The code, which accesses these parameters, is responsible for their management.

Run-time Environment of the Animation Primitive Driver

The animation frame update primitive driver and the interpolation primitive driver are called with the following environment.

Figure 2-117: Format of Parameters in the Argument Area

prmtop
tag(OT)
shift(OT)
offset(OT)
OUTP(packet area)
Animation header size
Interpolation function table pointer
CONTROL TOP pointer
PARAMETER TOP pointer
COORDINATE TOP pointer
???
base
src
dst
intr

The colored areas must always be set. The other areas are copied from the primitive header, so these areas will be updated if the header format changes.

The animation header size specifies the number of elements after the interpolation function table pointer exclusive of the last four elements. In the example above, the header size would have a value of "???+4" with the "???" determined from the element count. The header size is used by the interpolation function to locate the start of the interpolation function's parameter section (described next).

The last four parameters are the arguments area for the interpolation function.

base: starting address of the sequence pointer

src: starting address of the source key frame to interpolate

dst: starting address of the destination key frame to interpolate

intr: address where parameters will be saved after interpolation (if this value is 0, the parameters will not be saved)

Behavior of the Primitive Driver

Primitive drivers can be divided into the following two types:

1. Frame update drivers
2. Interpolation drivers

Primitive drivers are called each time GsSortUnit() is called.

Animation primitives are linked in the PRE-PROCESS area at the beginning of HMD's coordinate section. The animation primitive driver is initialized in the following manner.

1. When HMD initialization is performed with GsScanUnit(), GsScanAnim() should be called to perform a SCAN operation.
2. The starting address of the frame update driver should be entered in the HMD type field. This ensures that the frame update driver will be called each time GsSortUnit() is called. The frame update driver will call the interpolation driver.

The frame update driver specifies the calling interface for the interpolation driver. Thus, the program must be aware of the relationship between the interpolation driver and the frame update driver. The three bits in the type field that identify the frame update driver must be the same for the corresponding interpolation function.

The calling interface used by the standard frame update driver to call an interpolation function is described below.

FUNC(sp)

sp is a pointer to the start of the parameter area.

As described above, the parameter area pointed to by sp contains the base, src, dst, and intr parameters.

base: starting address of the sequence pointer corresponding to an update area which begins at the update index

src: address of the interpolation source

dst: address of the interpolation destination

intr: address for holding interpolated data. Data is not saved if this value is 0. To make interpolation parameter, intr is allowed to use to indicate destination key frame created previously.

The frame update driver provided by Sony Computer Entertainment Inc., has type set to 0x03000000. The corresponding interpolation primitive driver needs to have the parameter format described above.

Interpolation Algorithms

The following 3 algorithms are available for interpolation driver.

1. LINEAR
2. BEZIER
3. BSPLINE

LINEAR

This interpolates linear between SRC KEYFRAME and DST KEYFRAME parameters.

$$T = (TFRAME - RFRAME) / TRFRAME$$

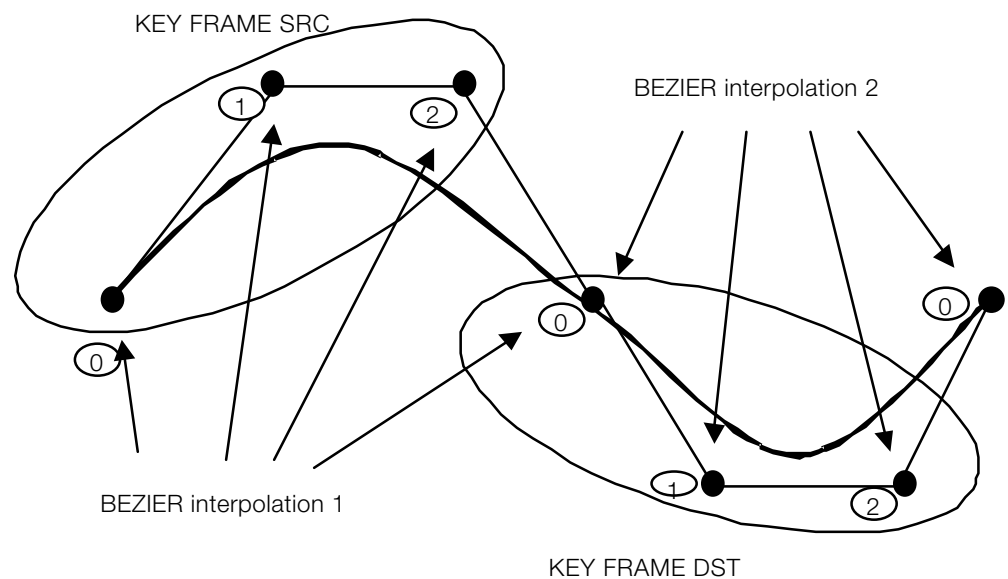
$$(1 - T) * SRC_KEYFRAME + T * DST_KEYFRAME$$

BEZIER

BEZIER type KEY FRAME has 3 control points.

Interpolation is performed with control point 0, 1 and 2 of SRC KEY FRAME, and control point 0 of DST KEY FRAME.

Figure 2-118: Bezier Interpolation



BSPLINE

BSPLINE type KEY FRAME has a control point as same as LINEAR type.

BSPLINE interpolation is performed between SRC-2, SRC-1, SRC and DST KEYFRAME.

The beginning of sequence has no history of previous key frames, thus, 3 key frames are required to enumerated with TFRAME=0.

To make a history of key frames, 4 words in key frame area of parameter section are required. Sequence descriptor (control: work) that indexed by START IDX in sequence pointer, indicates this area.

Figure 2-119: BSPLINE Work Area

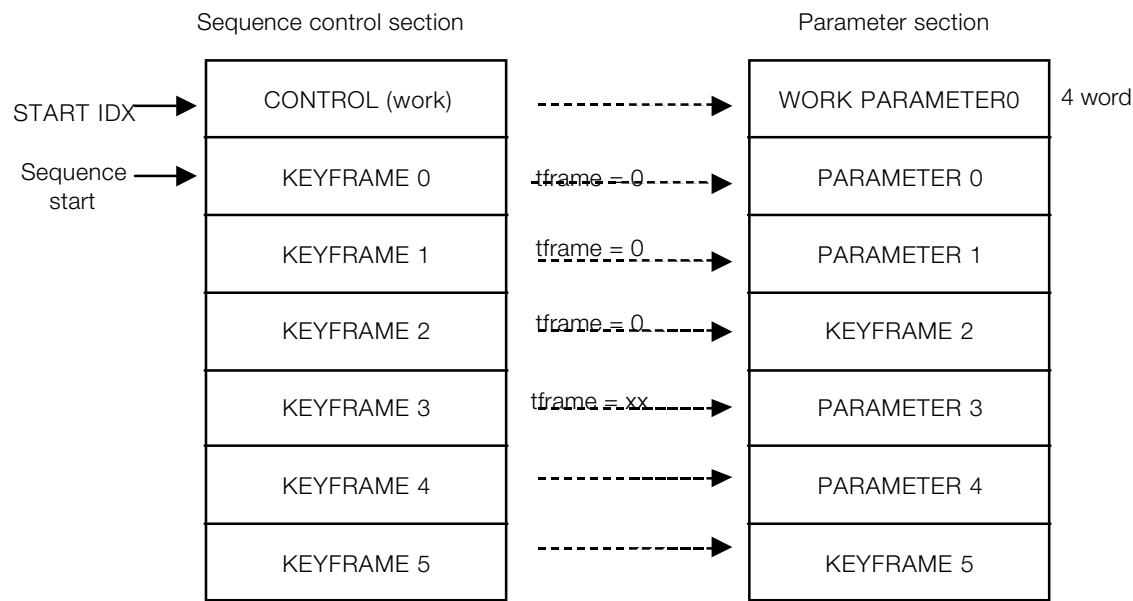
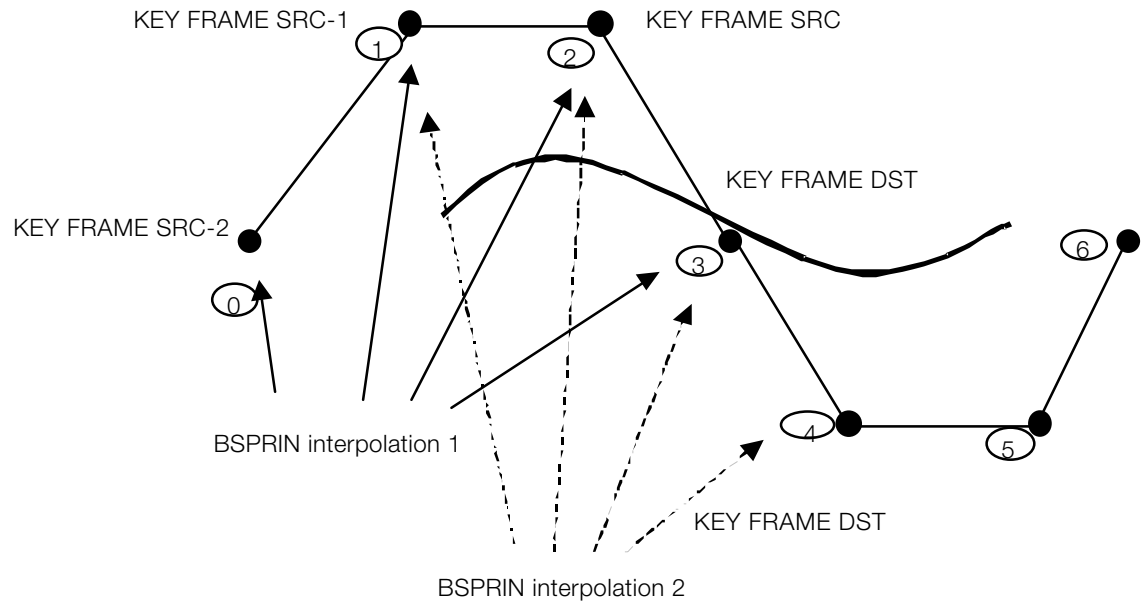


Figure 2-120: BSPLINE Interpolation



Animation Packets (COORDINATE)

DEV_ID(SCE)|CTG(CTG_ANIM)|DRV(CAT_STD|TGT_COORD)|PRIM_TYPE(x)

PARAMETER

0x03000010; SI_NONE|RI_LINEAR|TI_NONE; ps0r1t0

H(rx); H(ry); H(rz); H(0);

0x03000910; SI_LINEAR_1|RI_LINEAR|TI_NONE; ps9r1t0

H(rx); H(ry); H(rz);

H(scale);

0x03000030; SI_NONE|RI_BSPLINE|TI_NONE; ps0r3t0

H(rx); H(ry); H(rz); H(0);

0x03000001; SI_NONE|RI_NONE|TI_LINEAR; ps0r0t1

tx; ty; tz;

0x03000901; SI_LINEAR_1|RI_NONE|TI_LINEAR; ps9r0t1

tx; ty; tz;

H(scale); H(0);

0x03000011; SI_NONE|RI_LINEAR|TI_LINEAR; ps0r1t1

tx; ty; tz;

H(rx); H(ry); H(rz); H(0);

0x03000111; SI_LINEAR|RI_LINEAR|TI_LINEAR; ps1r1t1

tx; ty; tz;

H(rx); H(ry); H(rz);

H(sx); H(sy); H(sz);

0x03000911; SI_LINEAR_1|RI_LINEAR|TI_LINEAR; ps9r1t1

tx; ty; tz;

H(rx); H(ry); H(rz);

H(scale);

0x03000031; SI_NONE|RI_BSPLINE|TI_LINEAR; ps0r3t1

tx; ty; tz;

H(rx); H(ry); H(rz); H(0);

0x03000002; SI_NONE|RI_NONE|TI_BEZIER; ps0r0t2

tx0; ty0; tz0;

tx1; ty1; tz1;

tx2; ty2; tz2;


```

0x03000902; SI_LINEAR_1|RI_NONE|TI_BEZIER; ps9r0t2
tx0; ty0; tz0;
tx1; ty1; tz1;
tx2; ty2; tz2;
H(scale); H(0);

0x03000012; SI_NONE|RI_LINEAR|TI_BEZIER; ps9r0t2
tx0; ty0; tz0;
tx1; ty1; tz1;
tx2; ty2; tz2;
H(rx); H(ry); H(rz); H(0);

0x03000112; SI_LINEAR|RI_LINEAR|TI_BEZIER; ps1r1t2
tx0; ty0; tz0;
tx1; ty1; tz1;
tx2; ty2; tz2;
H(rx); H(ry); H(rz);
H(sx); H(sy); H(sz);

0x03000912; SI_LINEAR_1|RI_LINEAR|TI_BEZIER; ps9r1t2
tx0; ty0; tz0;
tx1; ty1; tz1;
tx2; ty2; tz2;
H(rx); H(ry); H(rz);
H(scale);

0x03000032; SI_NONE|RI_BSPLINE|TI_BEZIER; ps0r3t2
tx0; ty0; tz0;
tx1; ty1; tz1;
tx2; ty2; tz2;
H(rx); H(ry); H(rz); H(0);

0x03000003; SI_NONE|RI_NONE|TI_BSPLINE; ps0r0t3
tx; ty; tz;

0x03000013; SI_NONE|RI_LINEAR|TI_BSPLINE; ps0r1t3
tx; ty; tz;
H(rx); H(ry); H(rz); H(0);

0x03000033; SI_NONE|RI_BSPLINE|TI_BSPLINE; ps0r3t3
tx; ty; tz;
H(rx); H(ry); H(rz); H(0);

```

```

0x03000009; SI_NONE|RI_NONE|TI_LINEAR_S; ps0r0t9
H(tx); H(ty); H(tz); H(0);

0x03000909; SI_LINEAR_1|RI_NONE|TI_LINEAR_S; ps9r0t9
H(tx); H(ty); H(tz);
H(scale);

0x03000019; SI_NONE|RI_LINEAR|TI_LINEAR_S; ps0r1t9
H(tx); H(ty); H(tz);
H(rx); H(ry); H(rz);

0x03000119; SI_LINEAR|RI_LINEAR|TI_LINEAR_S; ps1r1t9
H(tx); H(ty); H(tz);
H(rx); H(ry); H(rz);
H(sx); H(sy); H(sz); H(0);

0x03000919; SI_LINEAR_1|RI_LINEAR|TI_LINEAR_S; ps9r1t9
H(tx); H(ty); H(tz);
H(rx); H(ry); H(rz);
H(scale); H(0);

0x03000039; SI_NONE|RI_BSPLINE|TI_LINEAR_S; ps0r3t9
H(tx); H(ty); H(tz);
H(rx); H(ry); H(rz);

0x0300000a; SI_NONE|RI_NONE|TI_BEZIER_S; ps0r0tA
H(tx0); H(ty0); H(tz0);
H(tx1); H(ty1); H(tz1);
H(tx2); H(ty2); H(tz2); H(0);

0x0300090a; SI_LINEAR_1|RI_NONE|TI_BEZIER_S; ps9r0tA
H(tx0); H(ty0); H(tz0);
H(tx1); H(ty1); H(tz1);
H(tx2); H(ty2); H(tz2);
H(scale);

0x0300001a; SI_NONE|RI_LINEAR|TI_BEZIER_S; ps0r1tA
H(tx0); H(ty0); H(tz0);
H(tx1); H(ty1); H(tz1);
H(tx2); H(ty2); H(tz2);
H(rx); H(ry); H(rz);

```

0x0300011a; SI_LINEAR|RI_LINEAR|TI_BEZIER_S; ps1r1tA

H(tx0); H(ty0); H(tz0);
 H(tx1); H(ty1); H(tz1);
 H(tx2); H(ty2); H(tz2);
 H(rx); H(ry); H(rz);
 H(sx); H(sy); H(sz); H(0);

0x0300091a; SI_LINEAR_1|RI_LINEAR|TI_BEZIER_S; ps9r1tA

H(tx0); H(ty0); H(tz0);
 H(tx1); H(ty1); H(tz1);
 H(tx2); H(ty2); H(tz2);
 H(rx); H(ry); H(rz);
 H(scale); H(0);

0x0300003a; SI_NONE|RI_BSPLINE|TI_BEZIER_S; ps0r3tA

H(tx0); H(ty0); H(tz0);
 H(tx1); H(ty1); H(tz1);
 H(tx2); H(ty2); H(tz2);
 H(rx); H(ry); H(rz);

0x0300000b; SI_NONE|RI_NONE|TI_BSPLINE_S; ps0r0tB

H(tx); H(ty); H(tz); H(0);

0x0300001b; SI_NONE|RI_LINEAR|TI_BSPLINE_S; ps0r1tB

H(tx); H(ty); H(tz);
 H(rx); H(ry); H(rz);

0x0300003b; SI_NONE|RI_BSPLINE|TI_BSPLINE_S; ps0r3tB

H(tx); H(ty); H(tz);
 H(rx); H(ry); H(rz);

0x03000020; SI_NONE|RI_BEZIER|TI_NONE; ps0r2t0

H(rx0); H(ry0); H(rz0);
 H(rx1); H(ry1); H(rz1);
 H(rx2); H(ry2); H(rz2); H(0);

0x03000021; SI_NONE|RI_BEZIER|TI_LINEAR; ps0r2t1

tx; ty; tz;
 H(rx0); H(ry0); H(rz0);
 H(rx1); H(ry1); H(rz1);
 H(rx2); H(ry2); H(rz2); H(0);

0x03000022; SI_NONE|RI_BEZIER|TI_BEZIER; ps0r2t2

tx0; ty0; tz0;

tx1; ty1; tz1;

tx2; ty2; tz2;

H(rx0); H(ry0); H(rz0);

H(rx1); H(ry1); H(rz1);

H(rx2); H(ry2); H(rz2); H(0);

0x03000023; SI_NONE|RI_BEZIER|TI_BSPLINE; ps0r2t2

tx; ty; tz;

H(rx0); H(ry0); H(rz0);

H(rx1); H(ry1); H(rz1);

H(rx2); H(ry2); H(rz2); H(0);

0x03000029; SI_NONE|RI_BEZIER|TI_LINEAR_S; ps0r2t9

H(tx); H(ty); H(tz);

H(rx0); H(ry0); H(rz0);

H(rx1); H(ry1); H(rz1);

H(rx2); H(ry2); H(rz2);

0x0300002a; SI_NONE|RI_BEZIER|TI_BEZIER_S; ps0r2tA

H(tx0); H(ty0); H(tz0);

H(tx1); H(ty1); H(tz1);

H(tx2); H(ty2); H(tz2);

H(rx0); H(ry0); H(rz0);

H(rx1); H(ry1); H(rz1);

H(rx2); H(ry2); H(rz2);

0x0300002b; SI_NONE|RI_BEZIER|TI_BSPLINE_S; ps0r2tB

H(tx); H(ty); H(tz);

H(rx0); H(ry0); H(rz0);

H(rx1); H(ry1); H(rz1);

H(rx2); H(ry2); H(rz2);

Animation Packets (General)

DEV_ID(SCE) | CTG(CTG_ANIM) | DRV(CAT_STD | TGT_GENERAL) | PRIM_TYPE(x)

LINEAR

General Single Linear(32bit)

0x03010110; GI_LINEAR | GI_WR(0x1) | GI_32
p0;

General Single Linear(32bit)

0x03010111; GI_LINEAR | GI_WR(0x1) | GI_16
0x03010121; GI_LINEAR | GI_WR(0x2) | GI_16
0x03010141; GI_LINEAR | GI_WR(0x4) | GI_16
H(p0); H(0);

General vector Linear(16bit)

0x03010171; GI_LINEAR | GI_WR(0x7) | GI_16
H(p0); H(p1); H(p2); H(0);

General Single Linear(8bit)

0x03010112; GI_LINEAR | GI_WR(0x1) | GI_8
0x03010122; GI_LINEAR | GI_WR(0x2) | GI_8
0x03010142; GI_LINEAR | GI_WR(0x4) | GI_8
B(p0); B(0); B(0); B(0);

General vector Linear(8bit)

0x03010172; GI_LINEAR | GI_WR(0x7) | GI_8
B(p0); B(p1); B(p2); B(0);

BEZIER**General single Bezier(32bit)**

0x03010210; GI_BEZIER | GI_WR(0x1) | GI_32
p00; p10; p20;

General single Bezier(16bit)

0x03010211; GI_BEZIER | GI_WR(0x1) | GI_16
0x03010221; GI_BEZIER | GI_WR(0x2) | GI_16
0x03010241; GI_BEZIER | GI_WR(0x4) | GI_16
H(p00); H(p10); H(p20); H(0);

General vector Bezier(16bit)

0x03010271; GI_BEZIER | GI_WR(0x7) | GI_16
H(p00); H(p01); H(p02);
H(p10); H(p11); H(p12);
H(p20); H(p21); H(p22); H(0);

General single Bezier(8bit)

```
0x03010212; GI_BEZIER|GI_WR(0x1)|GI_8
0x03010222; GI_BEZIER|GI_WR(0x1)|GI_8
0x03010242; GI_BEZIER|GI_WR(0x1)|GI_8
B(p00); B(p10); B(p20); B(0);
```

General vector Bezier(8bit)

```
0x03010272; GI_BEZIER|GI_WR(0x7)|GI_8
B(p00); B(p01); B(p02); B(0);
B(p10); B(p11); B(p12); B(0);
B(p20); B(p21); B(p22); B(0);
```

BSPLINE**General Single Bspline(32bit)**

```
0x03010310; GI_BSPLINE|GI_WR(0x1)|GI_32
p0;
```

General Single Bspline(16bit)

```
0x03010311; GI_BSPLINE|GI_WR(0x1)|GI_16
0x03010321; GI_BSPLINE|GI_WR(0x2)|GI_16
0x03010341; GI_BSPLINE|GI_WR(0x4)|GI_16
H(p0); H(0);
```

General vector Bspline(16bit)

```
0x03010371; GI_BSPLINE|GI_WR(0x7)|GI_16
H(p0); H(p1); H(p2); H(0);
```

General single Bspline(8bit)

```
0x03010312; GI_BSPLINE|GI_WR(0x1)|GI_8
0x03010322; GI_BSPLINE|GI_WR(0x2)|GI_8
0x03010342; GI_BSPLINE|GI_WR(0x4)|GI_8
B(p0); B(0); B(0); B(0);
```

General vector Bspline(8bit)

```
0x03010372; GI_BSPLINE|GI_WR(0x7)|GI_8
B(p0); B(p1); B(p2); B(0);
```

MIMe Primitive (Category 4)

Please refer to the following documents for more information on the MIMe primitive.

- libgs reference, section on the GsARGUNIT_JntMIMe structure
- libgs reference, section on the GsARGUNIT_RstJntMIMe structure
- libgs reference, section on the GsARGUNIT_VNMIMe structure
- libgs reference, section on the GsARGUNIT_RstVNMIMe structure
- libgs reference, section on the GsInitRstVtxMIMe, GsInitRstNrmMIMe function
- libgs reference, section on the GsU_04# function

The following symbols are used to indicate the MIMe type in a MIMe primitive.

- **JntMIMe** Joint MIMe (common to the following two types)
JntAxesMIMe: Joint-axes MIMe (Joint MIMe using rotation-axes interpolation)
JntRPYMIMe: Joint row-pitch-yaw MIMe (Joint MIMe using RPY interpolation)
- **RstJntMIMe** (common to the following two types)
RstJntAxesMIMe: Reset MIMe based on rotation-axes interpolation
RstJntRPYMIMe: Reset MIMe based on RPY interpolation
- **VNMIMe** Vertex / normal MIMe (common to the following two types)
VtxMIMe: Vertex MIMe
NrmMIMe: Normal MIMe
- **RstVNMIMe** Reset vertex / normal MIMe (common to the following two types)
RstVtxMIMe: Reset vertex MIMe
RstNrmMIMe: Reset normal MIMe

Areas needed specifically for MIMe primitives

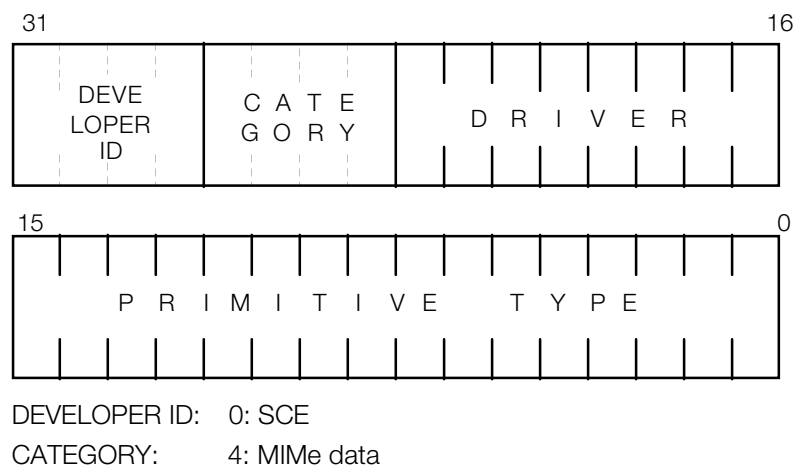
- MIMe DIFF section
- ORGSVN section (for VtxMIMe, NrmMIMe)
- MIMEPR area (when HMD contains MIMEPR)

Notes on Formats

- Up to 32 MIMe differences can be used for a single primitive.
- The JntMIMe function uses the same primitive block as the corresponding reset function (RstJntMIMe). However, VNMIMe and RstVNMIMe do not share this block and use their own primitive.
- When two or more JntMIMe primitives are used for a single joint, the corresponding reset functions (RstJntMIMe) must be called in reverse order otherwise, the state will not be correct).

type

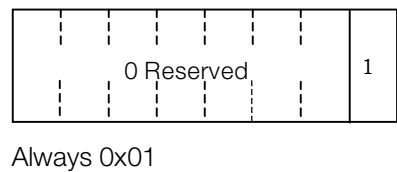
Figure 2-121: Primitive Type Field



DRIVER

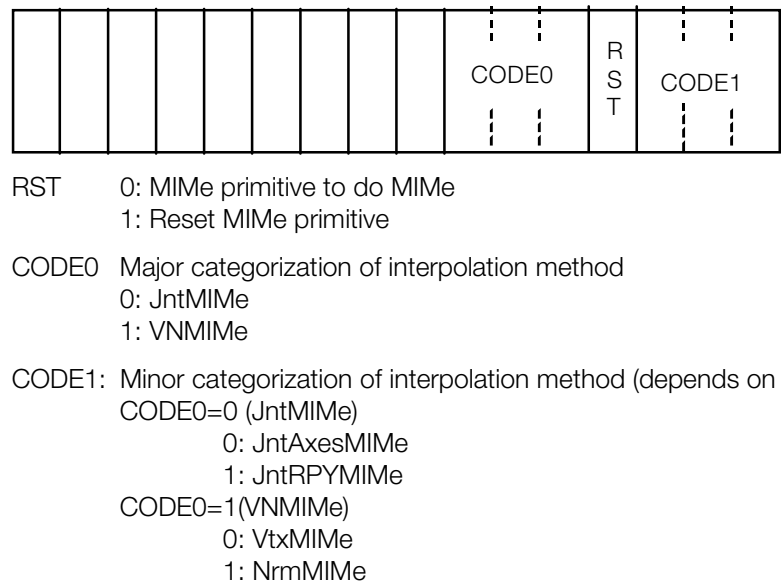
In MIMe category, DRIVER bits are defined as below.

Figure 2-122: MIMe Primitive DRIVER



PRIMITIVE TYPE

Figure 2-123: Primitive Type of MIMe Primitive



Format

Header for MIMe Primitive Block

HEADLEN:	Length of primitive header. This value will be changed by GsMap...MIMe(), GsMapRst....MIMe() functions.
COORD TOP:	Starting address of COORDINATE section (the number of long words from start of HMD)
MIMEPR PTR:	If HMD contains MIMEPR, the number of long words from start of HMD. If MIMEPR is outside of HMD, the value is 0.
MIMENUM:	The number of the MIMe keys. reserved(16bit): reserved (0)
MIMEID(16bit):	ID of the primitive (this area can be used freely by user and modeler)
MIMe DIFF TOP:	starting address of MIME DIFF section (number of long words from start of HMD)
ORGSVN TOP:	starting address of ORGSVN section (number of long words from start of HMD)
VERTEX TOP:	starting address of VERTEX section (number of long words from start of HMD)
NORMAL TOP:	starting address of NORMAL section (number of long words from start of HMD)

MIMeHeader(JntMIMe)

```
5;      /* header size */
M(CoordSect / 4);
M(MIMePr_ptr / 4);
MIMe_num;
H(MIMEID); H(0 /* reserved */);
M(MIMeDiffSect / 4);
```

MIMeHeader(RstJntMIMe)

```
3;      /* header size */
M(CoordSect / 4);
H(MIMEID); H(0 /* reserved */);
M(MIMeDiffSect / 4);
```

MIMeHeader(VNMIMe)

```
7;      /* header size */
M(MIMePr_ptr / 4);
MIMe_num;
H(MIMEID); H(0 /* reserved */);
M(MIMeDiffSect / 4);
M(MIMeOrgsVNSect / 4);
M(VertSect / 4);
M(NormSect / 4);
```

```

MIMEHeader(RstVNMIME)
5;      /* header size */
H(MIMEID); H(0 /* reserved */);
M(MIMeDiffSect / 4);
M(MIMeOrgsVNSect / 4);
M(VertSect / 4);
M(NormSect / 4);

```

MIMe Primitive

TYPE: type of the primitive.
 m(1bit): Initial value is 1 (changes to 0 during execution when TYPE is scanned and the function pointer is embedded)
 Num of DIFFs: MIMe DIFF IDX count
 MIMe DIFF IDX: starting address of MIMe DIFF (number of long words from MIMe DIFF TOP)

```

MIMe primitive
DEV_ID(SCE) | CTG(CTG_MIME) | DRV(MIMe_PRIM) | PRIM_TYPE(x)
H(size); M(H(num_diffs)); /* size = num_diffs + 1 */
(MIMeDiff0 - MIMeDiffSect) / 4;
:
(MIMeDiffN - MIMeDiffSect) / 4; /* N = num_diffs - 1 */

```

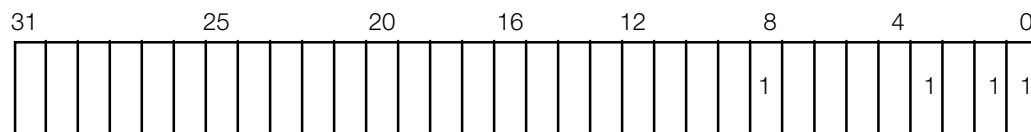
MIMe DIFF

Data in the MIMe DIFF section related to differences.

DIFFS NUM: number of DIFFS (DIFFS for Rst are not counted)
 COORDID: COORDINATE ID (the joint to apply MIMe)
 ONUM: Number of RstVNMIME blocks that correspond to VNMIME.
 dflags: bits with differences (DIFFS) are set to 1, 0 otherwise.

Example: When MIMe-key #0, #1, #3, #8 have differences

Figure 2-124: dflags Example



->In this case, dflags=0x0000010B

VNMIME Changed:

The changed address within RstVNMIMEDiffData of the corresponding RstVNMIME

MIMEDiffData:

For Rst, original data for resets.

Otherwise, actual difference values for each key. The DIFFS must be ordered in the same sequence as the dflags bits.

Details of the formats are shown below.

JntMIMEDiff/RstJntMIMEDiff

JntMIME and RstJntMIME are paired and use the same MIMEDIFF.

H(coord_ID); H(diffs_num);

dflags;

JntMIMEDiffData0:

: /* Jnt???MIMEDiffData format */

JntMIMEDiffDataN:

: /* N = diffs_num - 1 */

RstJntMIMEDiffData:

: /* RstJnt???MIMEDiffData format */

VNMIMEDiff

VNMIMEDiff:

H(onum); H(diffs_num);

dflags;

(VNMIMEDiffData0 - VNMIMEDiff) / 4;

:

(VNMIMEDiffDataN - VNMIMEDiff) / 4; /* N = diffs_num - 1 */

(VNMIMEChanged0 - MIMEDiffSect) / 4;

:

(VNMIMEChangedM - MIMEDiffSect) / 4; /* M = onum - 1 */

VNMIMEDiffData0:

: /* VNMIMEDiffData format */

VNMIMEDiffDataN:

: /* N = diffs_num - 1 */

RstVNMIMEDiff

H(0); H(diffs_num);

RstVNMIMEDiffData0:

: /* RstVNMIMEDiffData format */

RstVNMIMEDiffDataN:

: /* N = diffs_num - 1 */

MimeDiffData

Actual difference values for each key.

The format and contents vary according to the interpolation method.

Difference Value Data

ntp:

Bit 0 is 0 when the rotation values (dvx-dvz and m) are all 0. Otherwise, Bit 0 is 1.

Bit 1 is 0 when the translation values (dtx-dtz) are all 0. Otherwise, Bit 1 is 1.

JntRPYMeDiffData

```
H(dvx); H(dvy); H(dvz); H(ntp); /* rot difference value */
dtx; dty; dtz; /* t[0-2] difference value */
```

JntAxesMeDiffData

```
H(dvx); H(dvy); H(dvz); H(ntp); /* rot difference value rotation vector */
dtx; dty; dtz; /* t[0-2] difference value */
```

VNMeDiffData

```
vstart; /* number of first different vertex */
H(0 /* reserved */); H(vnum); /* number of difference vertices */
H(dvx0); H(dvy0); H(dvz0); H(0);
:
H(dvxN); H(dvyN); H(dvzN); H(0); /* N = vnum - 1 */
```

Original reset data**RstVNMeDiffData**

```
vstart; /* Number of the first vertex/normal which is */
/* different */
ostart; /* Number of ORGSVN area start which is used */
VNMeChanged: /* Referred from VNMeDiff */
H(changed); /* Initial value 0 */
/* At runtime, this value will be changed to 1 */
/* when the vertices or normal vectors in this */
/* region are changed to 0 when RstMe is reset */
H(vnum); /* Number of different vertices/normals */
```

RstJntRPYMeDiffData

```
H(dvx); H(dvy); H(dvz); /* Initial value is undefined */
/* The original coordinate's rot value will */
/* be saved here during execution */
H(changed); /* Initial value is 0; flag indicating data was */
/* saved */
dtx; dty; dtz; /* Initial value is undefined. The */
/* original coordinate's t[0-2] value */
/* will be saved here during execution */
```

RstJntAxesMIMeDiffData

```

H(m00); H(m01); H(m02);    /* Initial value is undefined */
                           /* The original coordinate's m[0-2] */
                           /* [0-2] value will be saved here during */
                           /* execution*/

H(m10); H(m11); H(m12);
H(m20); H(m21); H(m22);
H(changed); /* Initial value is 0; flag indicating data was */
            /* saved */

dtx; dty; dtz;             /* Initial value is undefined. The */
                           /* original coordinate's t[0-2] value */
                           /* will be saved here during execution */

```

MIMeOrgsVN Section

Initial values are not defined. These values are used in the following manner during execution. dx-z is the original vertex/normal data that had been saved.

MIMeOrgsVN

```

H(dvx0); H(dvy0); H(dvz0); H(0);
:
H(dvxN); H(dvyN); H(dvzN); H(0);

```

Ground Primitives (Category 5)

Ground primitive is allowed to use as one of HMD primitive. This primitive generates packets at run time based on width and height of a grid, and count of grids. Thus, data amount can be reduced in HMD data.

Primitive Header Section

Primitive header section

Primitive header format depends on texture is used or not.

(1) Non-textured

```
4;      /* header size */  
  
M(GndPolySect / 4); /* Polygon section */  
  
M(GndGridSect / 4); /* Grid section */  
  
M(GndVertSect / 4); /* Vertex section */  
  
M(GndNormSect / 4); /* Normal section */
```

(2) Textured

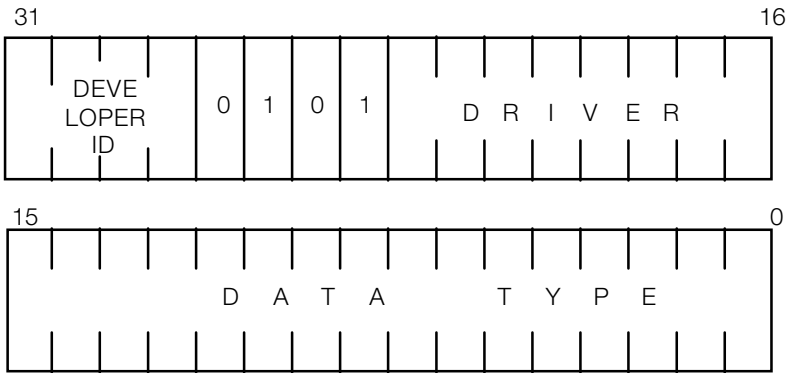
```
5;      /* header size */  
  
M(GndPolySect / 4); /* Polygon section */  
  
M(GndGridSect / 4); /* Grid section */  
  
M(GndVertSect / 4); /* Vertex section */  
  
M(GndNormSect / 4); /* Normal section */  
  
M(GndUVSect / 4);   /* UV section */
```

Type

Type of ground primitive is defined as below.

Ground TYPE

Figure 2-125: Ground Primitive Type Field



DRIVER: All 0 in this version
DATA TYPE: Defines type of data
0: Flat
1: Flat texture

Primitive Section

Primitive section is common for non-textured and textured type.

Ground primitive

```
DEV_ID(SCE) | CTG(CTG_GND) | DRV(x) | PRIM_TYPE(y)
H(size); H(0);
(GndPoly - GndPolySect) / 4;
(GndGrid - GndGridSect) / 4;
(GndVert - GndVertSect) / 4;
```

Polygon Section

Required information to generate actual polygons is saved in polygon section.

Polygon section is common for non-textured and textured type.

```
H(x0); H(y0);          /* Start point X coordinate; start point Y */
                        /* coordinate */
H(w); H(h);            /* 1 grid width; 1 grid height */
H(m); H(n);            /* Vertices count (horizontal); vertices */
                        /* count (vertical) */
H(size); H(base);      /* Size; base vertex */
H(v0); H(c0);          /* Start vertex number 0; grids count 0 */
:
H(vN); H(cN);          /* Start vertex number N; Grids count N; N; N = */
                        /* size - 1 */
```

Grid Section

Grid section has information for each grid, for example, indexes to normal vectors, RGB value and UV.

Grid section format depends on non-textured or textured type.

(1) Non-textured

```
B(r); B(g); B(b); B(0);
H(norm_idx); H(0);
:
B(r); B(g); B(b); B(0);
H(norm_idx); H(0);
```

(2) Textured

```
H(norm_idx); H(UV_idx);
:
H(norm_idx); H(UV_idx);
```

Vertex Section

Vertex section has information for each vertex, for example, Z value.

```
H(z0); H(z1);
:
H(zN-1); H(zN);
```

UV section

UV section has actual texture UV values that are referred from grid section.

```
H(uv0); H(cba);
H(uv1); H(tsb);
H(uv2); H(uv3);
:
H(uv0); H(cba);
H(uv1); H(tsb);
H(uv2); H(uv3);
```


HMD Library Primitive Types

The following is a list of primitive types implemented in the current HMD library. The list is shown in HMD assembler (labb) format. The following notation is used:

```
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(BOT)|PRIM_TYPE(TRI); /* 00100008; 4.2 */
```

In this example, the developer ID is "SCE" (0; standard primitive driver), the category is "CTG_POLY" (polygon primitive), the driver bit is "BOT" (double-sided flag ON), and the primitive type is "TRI" (triangle). The actual bit pattern is "00100008" in hexadecimal. A primitive driver function name can be obtained by adding "GsU_" to the actual bit pattern value. Primitive types that have been newly implemented in library 4.2 will also have "; 4.2" added after the bit pattern value. If there is no designation, the primitive type was implemented in version 4.1 or earlier.

Library 4.2 provides a beta release of a pseudo-environment map driver. These are expressed using the following notation.

```
DEV_ID(SCE)|CTG(6)|DRV(0x00 /* ??? */)|PRIM_TYPE(0x0100 /* ??? */);/* 06000100; 4.2 */
```

Since the pseudo-environment map driver is a beta release, symbol definitions are not included in the "hmd.def" HMD assembler definition file. Also, symbolic output is not supported in the "xhmd" HMD disassembler. This document, "hmd.doc", does not describe pseudo-environment mapping. A brief description is provided in the sample data directory at "psx\sample\graphics\hmd\data\sce\envmap\".

```
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(0); /* 00000000 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(TRI); /* 00000008 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(TRI|TME); /* 00000009 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(TRI|COL); /* 0000000a */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(TRI|COL|TME); /* 0000000b */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(TRI|IIP); /* 0000000c */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(TRI|IIP|TME); /* 0000000d */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(TRI|IIP|COL); /* 0000000e */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(TRI|IIP|COL|TME); /* 0000000f */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(QUAD); /* 00000010 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(QUAD|TME); /* 00000011 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(QUAD|COL); /* 00000012 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(QUAD|COL|TME); /* 00000013 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(QUAD|IIP); /* 00000014 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(QUAD|IIP|TME); /* 00000015 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(QUAD|IIP|COL); /* 00000016 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(QUAD|IIP|COL|TME); /* 00000017 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(MESH); /* 00000018 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(MESH|TME); /* 00000019 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(MESH|IIP); /* 0000001c */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(MESH|IIP|TME); /* 0000001d */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(TILE|TRI|TME); /* 00000209 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(TILE|TRI|IIP|TME); /* 0000020d */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(TILE|QUAD|TME); /* 00000211 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(0)|PRIM_TYPE(TILE|QUAD|IIP|TME); /* 00000215 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(DIV)|PRIM_TYPE(TRI); /* 00010008 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(DIV)|PRIM_TYPE(TRI|TME); /* 00010009 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(DIV)|PRIM_TYPE(TRI|IIP); /* 0001000c */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(DIV)|PRIM_TYPE(TRI|IIP|TME); /* 0001000d */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(DIV)|PRIM_TYPE(QUAD); /* 00010010 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(DIV)|PRIM_TYPE(QUAD|TME); /* 00010011 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(DIV)|PRIM_TYPE(QUAD|IIP); /* 00010014 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(DIV)|PRIM_TYPE(QUAD|IIP|TME); /* 00010015 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(FOG)|PRIM_TYPE(TRI); /* 00020008 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(FOG)|PRIM_TYPE(TRI|TME); /* 00020009 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(FOG)|PRIM_TYPE(TRI|IIP); /* 0002000c */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(FOG)|PRIM_TYPE(TRI|IIP|TME); /* 0002000d */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(FOG)|PRIM_TYPE(QUAD); /* 00020010 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(FOG)|PRIM_TYPE(QUAD|TME); /* 00020011 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(FOG)|PRIM_TYPE(QUAD|IIP); /* 00020014 */
DEV_ID(SCE)|CTG(CTG_POLY)|DRV(FOG)|PRIM_TYPE(QUAD|IIP|TME); /* 00020015 */
```

File Formats

```

DEV_ID(SCE) | CTG(CTG_POLY) | DRV(BOT) | PRIM_TYPE(TRI | IIP | COL | TME); /* 0010000f; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(BOT) | PRIM_TYPE(QUAD); /* 00100010; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(BOT) | PRIM_TYPE(QUAD | TME); /* 00100011; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(BOT) | PRIM_TYPE(QUAD | COL); /* 00100012; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(BOT) | PRIM_TYPE(QUAD | COL | TME); /* 00100013; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(BOT) | PRIM_TYPE(QUAD | IIP); /* 00100014; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(BOT) | PRIM_TYPE(QUAD | IIP | TME); /* 00100015; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(BOT) | PRIM_TYPE(QUAD | IIP | COL); /* 00100016; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(BOT) | PRIM_TYPE(QUAD | IIP | COL | TME); /* 00100017; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(BOT) | LGT | PRIM_TYPE(LMD | TRI); /* 00140048; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(BOT) | LGT | PRIM_TYPE(LMD | TRI | TME); /* 00140049; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(BOT) | LGT | PRIM_TYPE(LMD | TRI | IIP); /* 0014004c; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(BOT) | LGT | PRIM_TYPE(LMD | TRI | IIP | TME); /* 0014004d; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(BOT) | LGT | PRIM_TYPE(LMD | QUAD); /* 00140050; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(BOT) | LGT | PRIM_TYPE(LMD | QUAD | TME); /* 00140051; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(BOT) | LGT | PRIM_TYPE(LMD | QUAD | IIP); /* 00140054; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(BOT) | LGT | PRIM_TYPE(LMD | QUAD | IIP | TME); /* 00140055; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | PRIM_TYPE(TRI); /* 00200008 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | PRIM_TYPE(TRI | TME); /* 00200009 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | PRIM_TYPE(TRI | IIP); /* 0020000c */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | PRIM_TYPE(TRI | IIP | TME); /* 0020000d */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | PRIM_TYPE(QUAD); /* 00200010 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | PRIM_TYPE(QUAD | TME); /* 00200011 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | PRIM_TYPE(QUAD | IIP); /* 00200014 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | PRIM_TYPE(QUAD | IIP | TME); /* 00200015 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | LGT | PRIM_TYPE(LMD | TRI); /* 00240048; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | LGT | PRIM_TYPE(LMD | TRI | TME); /* 00240049; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | LGT | PRIM_TYPE(LMD | TRI | IIP); /* 0024004c; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | LGT | PRIM_TYPE(LMD | TRI | IIP | TME); /* 0024004d; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | LGT | PRIM_TYPE(LMD | QUAD); /* 00240050; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | LGT | PRIM_TYPE(LMD | QUAD | TME); /* 00240051; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | LGT | PRIM_TYPE(LMD | QUAD | IIP); /* 00240054; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | LGT | PRIM_TYPE(LMD | QUAD | IIP | TME); /* 00240055; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | BOT | PRIM_TYPE(TRI); /* 00300008; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | BOT | PRIM_TYPE(TRI | TME); /* 00300009; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | BOT | PRIM_TYPE(TRI | IIP); /* 0030000c; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | BOT | PRIM_TYPE(TRI | IIP | TME); /* 0030000d; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | BOT | PRIM_TYPE(QUAD); /* 00300010; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | BOT | PRIM_TYPE(QUAD | TME); /* 00300011; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | BOT | PRIM_TYPE(QUAD | IIP); /* 00300014; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | BOT | PRIM_TYPE(QUAD | IIP | TME); /* 00300015; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | BOT | LGT | PRIM_TYPE(LMD | TRI); /* 00340048; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | BOT | LGT | PRIM_TYPE(LMD | TRI | TME); /* 00340049; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | BOT | LGT | PRIM_TYPE(LMD | TRI | IIP); /* 0034004c; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | BOT | LGT | PRIM_TYPE(LMD | TRI | IIP | TME); /* 0034004d; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | BOT | LGT | PRIM_TYPE(LMD | QUAD); /* 00340050; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | BOT | LGT | PRIM_TYPE(LMD | QUAD | TME); /* 00340051; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | BOT | LGT | PRIM_TYPE(LMD | QUAD | IIP); /* 00340054; 4.2 */
DEV_ID(SCE) | CTG(CTG_POLY) | DRV(STP) | BOT | LGT | PRIM_TYPE(LMD | QUAD | IIP | TME); /* 00340055; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(0) | PRIM_TYPE(0); /* 01000000 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(0) | PRIM_TYPE(TRI | IIP); /* 0100000c */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(0) | PRIM_TYPE(TRI | IIP | TME); /* 0100000d */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(0) | PRIM_TYPE(QUAD | IIP); /* 01000014 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(0) | PRIM_TYPE(QUAD | IIP | TME); /* 01000015 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(0) | PRIM_TYPE(TILE | TRI | IIP | TME); /* 0100020d; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(0) | PRIM_TYPE(TILE | QUAD | IIP | TME); /* 01000215; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(FOG) | PRIM_TYPE(TRI | IIP); /* 0102000c */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(FOG) | PRIM_TYPE(TRI | IIP | TME); /* 0102000d */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(FOG) | PRIM_TYPE(QUAD | IIP); /* 01020014 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(FOG) | PRIM_TYPE(QUAD | IIP | TME); /* 01020015 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(LGT) | PRIM_TYPE(LMD | TRI); /* 01040048 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(LGT) | PRIM_TYPE(LMD | TRI | TME); /* 01040049 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(LGT) | PRIM_TYPE(LMD | TRI | IIP); /* 0104004c */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(LGT) | PRIM_TYPE(LMD | TRI | IIP | TME); /* 0104004d */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(LGT) | PRIM_TYPE(LMD | QUAD); /* 01040050 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(LGT) | PRIM_TYPE(LMD | QUAD | TME); /* 01040051 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(LGT) | PRIM_TYPE(LMD | QUAD | IIP); /* 01040054 */

```

```

DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(LGT) | PRIM_TYPE(LMD|QUAD|IIP|TME);          /* 01040055 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(BOT|LGT) | PRIM_TYPE(LMD|TRI);              /* 01140048; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(BOT|LGT) | PRIM_TYPE(LMD|TRI|TME);          /* 01140049; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(BOT|LGT) | PRIM_TYPE(LMD|TRI|IIP);          /* 0114004c; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(BOT|LGT) | PRIM_TYPE(LMD|TRI|IIP|TME);      /* 0114004d; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(BOT|LGT) | PRIM_TYPE(LMD|QUAD);             /* 01140050; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(BOT|LGT) | PRIM_TYPE(LMD|QUAD|TME);         /* 01140051; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(BOT|LGT) | PRIM_TYPE(LMD|QUAD|IIP);         /* 01140054; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(BOT|LGT) | PRIM_TYPE(LMD|QUAD|IIP|TME);     /* 01140055; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP) | PRIM_TYPE(TRI|IIP);                  /* 0120000c */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP) | PRIM_TYPE(TRI|IIP|TME);              /* 0120000d */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP) | PRIM_TYPE(QUAD|IIP);                 /* 01200014 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP) | PRIM_TYPE(QUAD|IIP|TME);             /* 01200015 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP|LGT) | PRIM_TYPE(LMD|TRI);              /* 01240048; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP|LGT) | PRIM_TYPE(LMD|TRI|TME);          /* 01240049; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP|LGT) | PRIM_TYPE(LMD|TRI|IIP);          /* 0124004c; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP|LGT) | PRIM_TYPE(LMD|TRI|IIP|TME);      /* 0124004d; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP|LGT) | PRIM_TYPE(LMD|QUAD);             /* 01240050; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP|LGT) | PRIM_TYPE(LMD|QUAD|TME);         /* 01240051; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP|LGT) | PRIM_TYPE(LMD|QUAD|IIP);         /* 01240054; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP|LGT) | PRIM_TYPE(LMD|QUAD|IIP|TME);     /* 01240055; 4.2 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP|BOT|LGT) | PRIM_TYPE(LMD|TRI);          /* 01340048 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP|BOT|LGT) | PRIM_TYPE(LMD|TRI|TME);      /* 01340049 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP|BOT|LGT) | PRIM_TYPE(LMD|TRI|IIP);      /* 0134004c */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP|BOT|LGT) | PRIM_TYPE(LMD|TRI|IIP|TME);  /* 0134004d */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP|BOT|LGT) | PRIM_TYPE(LMD|QUAD);         /* 01340050 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP|BOT|LGT) | PRIM_TYPE(LMD|QUAD|TME);     /* 01340051 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP|BOT|LGT) | PRIM_TYPE(LMD|QUAD|IIP);     /* 01340054 */
DEV_ID(SCE) | CTG(CTG_SHARED) | DRV(STP|BOT|LGT) | PRIM_TYPE(LMD|QUAD|IIP|TME); /* 01340055 */
DEV_ID(SCE) | CTG(CTG_IMAGE) | DRV(0) | PRIM_TYPE(NOCLUT);                     /* 02000000 */
DEV_ID(SCE) | CTG(CTG_IMAGE) | DRV(0) | PRIM_TYPE(WITHCLUT);                   /* 02000001 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_COORD) | PRIM_TYPE(SI_NONE|RI_NONE|TI_NONE);               /* 03000000 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_COORD) | PRIM_TYPE(SI_NONE|RI_NONE|TI_LINEAR);              /* 03000001 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_COORD) | PRIM_TYPE(SI_NONE|RI_NONE|TI_BEZIER);              /* 03000002 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_COORD) | PRIM_TYPE(SI_NONE|RI_NONE|TI_BSPLINE);             /* 03000003 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_COORD) | PRIM_TYPE(SI_NONE|RI_NONE);                       /* 03000004 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_COORD) | PRIM_TYPE(SI_NONE|RI_NONE|TI_LINEAR_S);             /* 03000009 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_COORD) | PRIM_TYPE(SI_NONE|RI_NONE|TI_BEZIER_S);            /* 0300000a */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_COORD) | PRIM_TYPE(SI_NONE|RI_NONE|TI_BSPLINE_S);           /* 0300000b */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_COORD) | PRIM_TYPE(SI_NONE|RI_LINEAR|TI_NONE);              /* 03000010 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_COORD) | PRIM_TYPE(SI_NONE|RI_LINEAR|TI_LINEAR);            /* 03000011 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_COORD) | PRIM_TYPE(SI_NONE|RI_LINEAR|TI_BEZIER);            /* 03000012 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_COORD) | PRIM_TYPE(SI_NONE|RI_LINEAR|TI_BSPLINE);           /* 03000013 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_COORD) | PRIM_TYPE(SI_NONE|RI_LINEAR|TI_LINEAR_S);          /* 03000019 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_COORD) | PRIM_TYPE(SI_NONE|RI_LINEAR|TI_BEZIER_S);          /* 0300001a */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_COORD) | PRIM_TYPE(SI_NONE|RI_LINEAR|TI_BSPLINE_S);         /* 0300001b */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_COORD) | PRIM_TYPE(SI_NONE|RI_BEZIER|TI_NONE);              /* 03000020 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_COORD) | PRIM_TYPE(SI_NONE|RI_BEZIER|TI_LINEAR);            /* 03000021 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_COORD) | PRIM_TYPE(SI_NONE|RI_BEZIER|TI_BEZIER);           /* 03000022 */

```

```

DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_NONE | RI_BEZIER | TI_BSPLINE); /* 03000023 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_NONE | RI_BEZIER | TI_LINEAR_S); /* 03000029 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_NONE | RI_BEZIER | TI_BEZIER_S); /* 0300002a */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_NONE | RI_BEZIER | TI_BSPLINE_S); /* 0300002b */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_NONE | RI_BSPLINE | TI_NONE); /* 03000030 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_NONE | RI_BSPLINE | TI_LINEAR); /* 03000031 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_NONE | RI_BSPLINE | TI_BEZIER); /* 03000032 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_NONE | RI_BSPLINE | TI_BSPLINE); /* 03000033 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_NONE | RI_BSPLINE | TI_LINEAR_S); /* 03000039 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_NONE | RI_BSPLINE | TI_BEZIER_S); /* 0300003a */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_NONE | RI_BSPLINE | TI_BSPLINE_S); /* 0300003b */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_LINEAR | RI_NONE | TI_NONE); /* 03000100 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_LINEAR | RI_LINEAR | TI_LINEAR); /* 03000111 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_LINEAR | RI_LINEAR | TI_BEZIER); /* 03000112 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_LINEAR | RI_LINEAR | TI_LINEAR_S); /* 03000119 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_LINEAR | RI_LINEAR | TI_BEZIER_S); /* 0300011a */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_LINEAR_1 | RI_NONE | TI_LINEAR); /* 03000901 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_LINEAR_1 | RI_NONE | TI_BEZIER); /* 03000902 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_LINEAR_1 | RI_NONE | TI_LINEAR_S); /* 03000909 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_LINEAR_1 | RI_NONE | TI_BEZIER_S); /* 0300090a */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_LINEAR_1 | RI_LINEAR | TI_NONE); /* 03000910 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_LINEAR_1 | RI_LINEAR | TI_LINEAR); /* 03000911 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_LINEAR_1 | RI_LINEAR | TI_BEZIER); /* 03000912 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_LINEAR_1 | RI_LINEAR | TI_LINEAR_S); /* 03000919 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_COORD) | PRIM_TYPE(SI_LINEAR_1 | RI_LINEAR | TI_BEZIER_S); /* 0300091a */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_GENERAL) | PRIM_TYPE(GI_LINEAR | GI_WR(0x1) | GI_32); /* 03010110 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_GENERAL) | PRIM_TYPE(GI_LINEAR | GI_WR(0x1) | GI_16); /* 03010111 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_GENERAL) | PRIM_TYPE(GI_LINEAR | GI_WR(0x1) | GI_8); /* 03010112 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_GENERAL) | PRIM_TYPE(GI_LINEAR | GI_WR(0x2) | GI_16); /* 03010121 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_GENERAL) | PRIM_TYPE(GI_LINEAR | GI_WR(0x2) | GI_8); /* 03010122 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_GENERAL) | PRIM_TYPE(GI_LINEAR | GI_WR(0x4) | GI_16); /* 03010141 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_GENERAL) | PRIM_TYPE(GI_LINEAR | GI_WR(0x4) | GI_8); /* 03010142 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD | TGT_GENERAL) | PRIM_TYPE(GI_LINEAR | GI_WR(0x7) | GI_16); /* 03010171 */
DEV_ID(SCE) | CTG(CTG_ANIM)

```

```

    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_LINEAR|GI_WR(0x7)|GI_8); /* 03010172 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_LINEAR|GI_WR(0x8)|GI_8); /* 03010182 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_BEZIER|GI_WR(0x1)|GI_32); /* 03010210 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_BEZIER|GI_WR(0x1)|GI_16); /* 03010211 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_BEZIER|GI_WR(0x1)|GI_8); /* 03010212 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_BEZIER|GI_WR(0x2)|GI_16); /* 03010221 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_BEZIER|GI_WR(0x2)|GI_8); /* 03010222 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_BEZIER|GI_WR(0x4)|GI_16); /* 03010241 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_BEZIER|GI_WR(0x4)|GI_8); /* 03010242 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_BEZIER|GI_WR(0x7)|GI_16); /* 03010271 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_BEZIER|GI_WR(0x7)|GI_8); /* 03010272 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_BSPLINE|GI_WR(0x1)|GI_32); /* 03010310 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_BSPLINE|GI_WR(0x1)|GI_16); /* 03010311 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_BSPLINE|GI_WR(0x1)|GI_8); /* 03010312 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_BSPLINE|GI_WR(0x2)|GI_16); /* 03010321 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_BSPLINE|GI_WR(0x2)|GI_8); /* 03010322 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_BSPLINE|GI_WR(0x4)|GI_16); /* 03010341 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_BSPLINE|GI_WR(0x4)|GI_8); /* 03010342 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_BSPLINE|GI_WR(0x7)|GI_16); /* 03010371 */
DEV_ID(SCE) | CTG(CTG_ANIM)
    | DRV(CAT_STD|TGT_GENERAL) | PRIM_TYPE(GI_BSPLINE|GI_WR(0x7)|GI_8); /* 03010372 */
DEV_ID(SCE) | CTG(CTG_MIME) | DRV(MIME_PRIM) | PRIM_TYPE(JNT_AXES); /* 04010010 */
DEV_ID(SCE) | CTG(CTG_MIME) | DRV(MIME_PRIM) | PRIM_TYPE(JNT_RPY); /* 04010011 */
DEV_ID(SCE) | CTG(CTG_MIME) | DRV(MIME_PRIM) | PRIM_TYPE(RST|JNT_AXES); /* 04010018 */
DEV_ID(SCE) | CTG(CTG_MIME) | DRV(MIME_PRIM) | PRIM_TYPE(RST|JNT_RPY); /* 04010019 */
DEV_ID(SCE) | CTG(CTG_MIME) | DRV(MIME_PRIM) | PRIM_TYPE(VTX); /* 04010020 */
DEV_ID(SCE) | CTG(CTG_MIME) | DRV(MIME_PRIM) | PRIM_TYPE(NRM); /* 04010021 */
DEV_ID(SCE) | CTG(CTG_MIME) | DRV(MIME_PRIM) | PRIM_TYPE(RST|VTX); /* 04010028 */
DEV_ID(SCE) | CTG(CTG_MIME) | DRV(MIME_PRIM) | PRIM_TYPE(RST|NRM); /* 04010029 */
DEV_ID(SCE) | CTG(CTG_GND) | DRV(0) | PRIM_TYPE(0); /* 05000000 */
DEV_ID(SCE) | CTG(CTG_GND) | DRV(0) | PRIM_TYPE(TME); /* 05000001 */
DEV_ID(SCE) | CTG(6) | DRV(0x00 /* ??? */) | PRIM_TYPE(0x0100 /* ??? */); /* 06000100; 4.2 */
DEV_ID(SCE) | CTG(6) | DRV(0x00 /* ??? */) | PRIM_TYPE(0x100c /* ??? */); /* 0600100c; 4.2 */
DEV_ID(SCE) | CTG(6) | DRV(0x00 /* ??? */) | PRIM_TYPE(0x1014 /* ??? */); /* 06001014; 4.2 */
DEV_ID(SCE) | CTG(6) | DRV(0x00 /* ??? */) | PRIM_TYPE(0x110c /* ??? */); /* 0600110c; 4.2 */
DEV_ID(SCE) | CTG(6) | DRV(0x00 /* ??? */) | PRIM_TYPE(0x1114 /* ??? */); /* 06001114; 4.2 */
DEV_ID(SCE) | CTG(6) | DRV(0x00 /* ??? */) | PRIM_TYPE(0x200c /* ??? */); /* 0600200c; 4.2 */
DEV_ID(SCE) | CTG(6) | DRV(0x00 /* ??? */) | PRIM_TYPE(0x2014 /* ??? */); /* 06002014; 4.2 */
DEV_ID(SCE) | CTG(6) | DRV(0x00 /* ??? */) | PRIM_TYPE(0x300c /* ??? */); /* 0600300c; 4.2 */
DEV_ID(SCE) | CTG(6) | DRV(0x00 /* ??? */) | PRIM_TYPE(0x3014 /* ??? */); /* 06003014; 4.2 */
DEV_ID(SCE) | CTG(6) | DRV(0x00 /* ??? */) | PRIM_TYPE(0x400c /* ??? */); /* 0600400c; 4.2 */
DEV_ID(SCE) | CTG(6) | DRV(0x00 /* ??? */) | PRIM_TYPE(0x4014 /* ??? */); /* 06004014; 4.2 */
DEV_ID(SCE) | CTG(6) | DRV(0x00 /* ??? */) | PRIM_TYPE(0x500c /* ??? */); /* 0600500c; 4.2 */
DEV_ID(SCE) | CTG(6) | DRV(0x00 /* ??? */) | PRIM_TYPE(0x5014 /* ??? */); /* 06005014; 4.2 */

```

HMD Animation

The HMD library also supports animation. Since HMD holds coordinate information, the motion of a hierarchical model can be described.

A special characteristic of HMD animation is the interactive control of animation sequences via the Realtime Motion Switch. This technique enables movement at arbitrary times between multiple pre-defined motion sequence patterns. This technique allows interactivity to be implemented — a feature which is indispensable in games. It also makes it possible to tune the authoring level (i.e. create apparent motion).

The amount of memory used for HMD animation data has been minimized by enabling sequences to be used. Entities are not represented in the data, as everything is referenced according to indexes and pointers.

Since the key frame interpolation method for HMD animation is managed by HMD Type, various interpolation methods can be used. A new interpolation method can also be defined by adding a Type.

A library for performing LINEAR, BEZIER and B-SPLINE coordinate rotation, translation and scaling with LINEAR, BEZIER and B-SPLINE interpolation is provided as a Primitive Driver. Also, the common interpolation functions for animating the optional data within HMD are provided by the LINEAR, BEZIER and B-SPLINE algorithms. In this way, animation of vertices, colors, etc. is possible.

Animation Definition

Sequence control descriptor

One animation sequence is defined by a list of 16-bit sequence control descriptors (SC). There are three kinds of SCs. One is a key frame descriptor (SCK), another is a jump descriptor (SCJ) and the third is for control (SCC).

A key frame descriptor (SCK) holds an index to the area in memory area that represents the key frame entity. A jump descriptor (SCJ) holds the index of an SCK jump destination. The SCK also holds the amount of time until the next key frame (TFRAME). It also maintains an index of interpolation functions (Type Idx). SCC displays control information such as sequence stoppage.

All sequences contain a 7-bit ID called a stream ID (SID). An SCJ holds both a source stream ID (SSID) and a destination stream ID (DSID). A jump due to an SCJ is performed only when the SSID matches the SID of the relevant sequence. If it does not match, the pointer moves directly to the next SC. The DSID determines the SID of the jump destination when a jump occurs. However, when the SID is equal to 0, it unconditionally matches all SIDs. As an operational rule, it is advisable that SID127 not be matched.

Sequence header

The sequence header, which unifies the management of individual pieces of sequence information, consists of the following two parts:

- 1) Sequence pointer
- 2) Sequence information

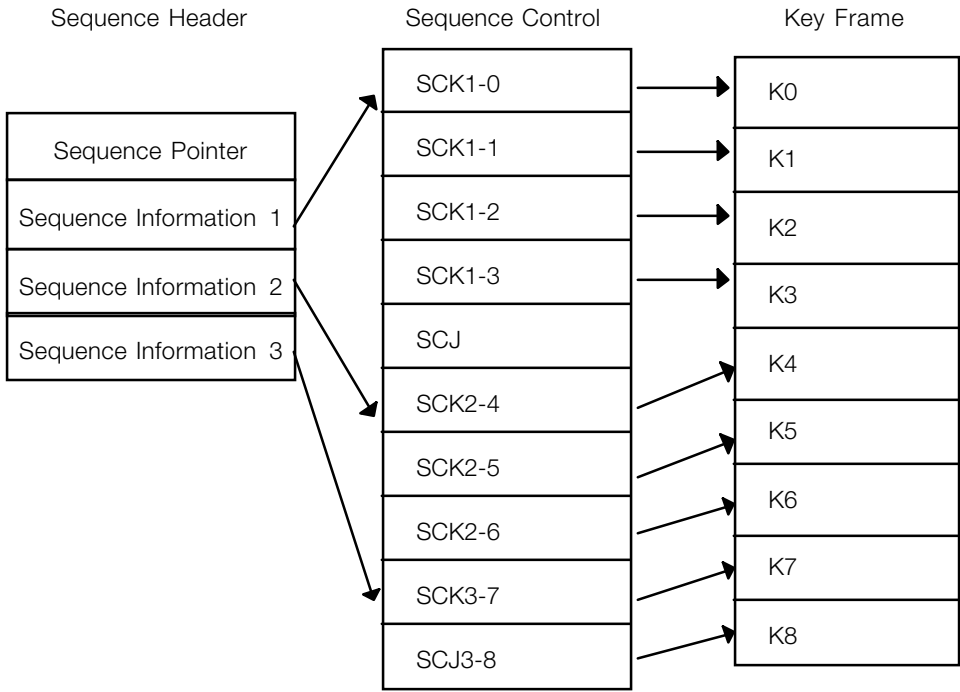
The sequence pointer directs animation playback control, which is described later.

The sequence information holds information on individual sequences. (Entries are listed for the number of sequences.)

The information for one sequence consists of the sequence starting index and the stream ID. The sequence starting index contains the index where that sequence begins of the area in which the SCs are listed. The stream ID indicates the SID at the time that that sequence is to be played back.

This information is referenced by all user programs. A user program controls a sequence by notifying the library via the sequence pointer.

Figure 2-126: Sequence Management Construction



Animation Playback

Frame update driver

A frame update driver interprets a sequence according to a time series and calls the appropriate interpolation function for performing the interpolation.

Frame update drivers are included according to the same framework as HMD primitive drivers. The frame update driver `GsU_03000000()`, which is provided with Version 4.0 of libgs, provides such features as the Realtime Motion Switch, forward and reverse playback, slow-motion playback up to 1/16 speed, and high-speed playback up to 8-times normal speed.

Interpolation driver

The interpolation driver is a function for performing key frame interpolation. Although the interpolation driver is identified according to Type in a similar manner as the frame update driver, it is not implemented by the HMD standard primitive driver framework. Instead, the special-purpose SCAN function `GsScanAnim()` is used, rather than the standard SCAN function `GsScanUnit()`.

When the SCAN function ends, the pointers to interpolation drivers are listed in a special-purpose area (interpolation function table section).

The SCK specifies the interpolation driver that should be called for each key frame according to Type Idx. This enables the key frame interpolation method to be switched within a single sequence.

Sequence pointer

The sequence pointer holds the playback point information of an animation. The playback of an animation can be controlled via this pointer.

The following elements are maintained in the sequence pointer:

- Rewrite IDX Specifies the areas that are to be updated by the animation.
- NUM Holds the number of sequences which can be substituted for that sequence pointer.
- INTR IDX SCK index indicating the area for holding the current parameters.
- AFRAME Manages the absolute frame numbers of the sequence.
- SRC INTRIDX Contains the area where parameters to be specified for INTR IDX are held.
- SPEED Playback speed.
- TFRAME Time interval between key frames.
- RFRAME Time interval from a key frame (decremented).
- Stream IDSequence ID number.
- TCTR IDX Index to the SC that holds the target key frame.
- CTR IDX Index to the SC that holds the source key frame.
- START IDX Holds the starting index of the sequence.
- START SID Holds the SID when the sequence starts.
- TRAVELING A variable that is reset to 0 at a key frame transition point. This can be freely used.

Some of these parameters can be set only by the programmer, and others can be updated by a frame update driver. For details, see the `GsSEQ` structure reference.

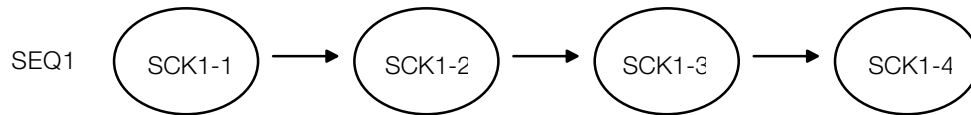
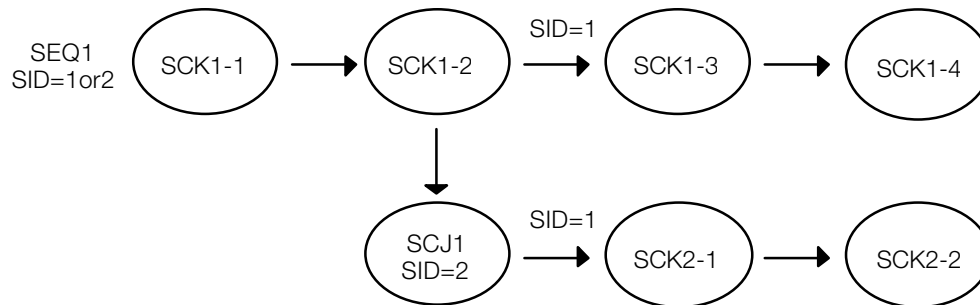
Realtime Motion Switch

This function makes interactive animation possible. It is implemented by the HMD frame update drivers and interpolation drivers.

The Realtime Motion Switch is divided into two functions. One function switches sequences in terms of key frame units according to the SID, and the other switches sequences immediately during interpolation.

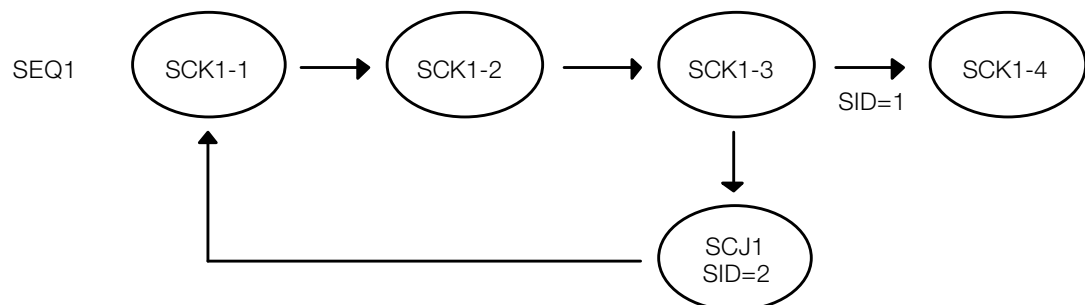
Sequence switching using the SID

Normal sequence

Figure 2-127: Sequence With No Jumps**Figure 2-128: Sequence With Jumps**

If the SCJ1 descriptor is written in advance and the Sid is 1, this kind of sequence branches to SCK2-1 after SCK1-2. If the information that the Sid is to be set to 0 after the jump is written for the SCJ1 descriptor, the Sid is set to 0 after the jump. Since SCJ descriptors can be arranged in multiple series, individual jump destinations can be specified for various Sids

Sequence branching can be controlled at execution time by changing (rewriting) the Sid from 0 to 1 before the sequence pointer passes the SCJ1 descriptor.

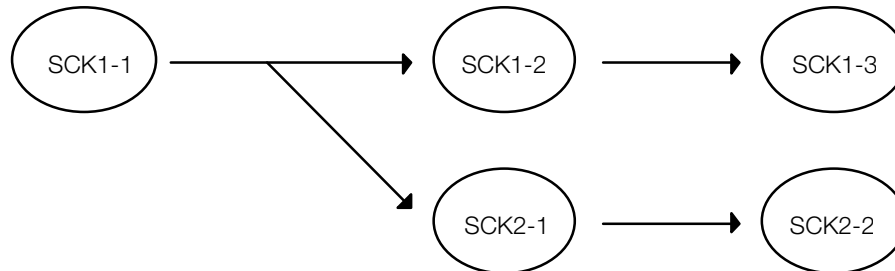
Loop sequence**Figure 2-129: Loop Sequence**

A loop sequence is realized by jumping forward according to a jump descriptor. Looping continues while the SID is 2, and control escapes the loop when the SID is set to 1. The loop can be controlled interactively by rewriting the SID at execution time.

Immediate sequence switching

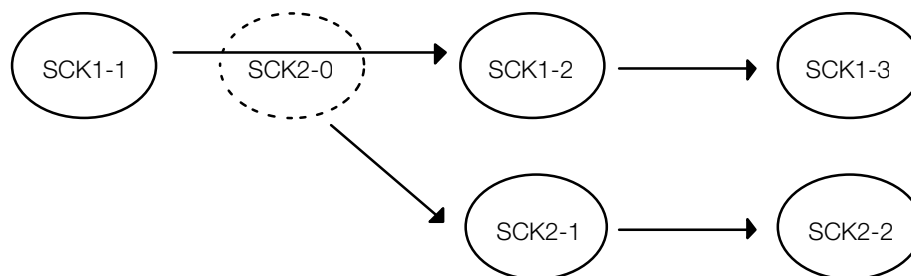
With sequence switching via the SID, a sequence is switched only when the key frame changes. This has the advantage that the switching is completely controllable because the sequence is switched only at points where an SID change can be issued and only at intended locations. However, since no response appears unless the key frame is reached, this method presents a problem from the standpoint of responsiveness. The figure below illustrates immediate sequence switching.

Figure 2-130: Immediate Sequence Switching 1



The sequence can be changed to key frame SCK2-1 at any time during interpolation between key frame SCK1-1 and key frame SCK1-2. In this case, a new virtual key frame SCK2-0 is defined at the branch point.

Figure 2-131: Immediate Sequence Switching 2



To implement this function, the sequence pointer is set as described below. An area for saving the current parameters is created in advance by entering a DUMMY key frame. This is defined as SCK2-0. Then, SCK2-0 is entered in INTR IDX at the frame at which the sequence was switched. 0xffff is entered in INTR IDX at the next frame. 0xffff prohibits parameter updating. This process enters the current location's parameters in the key frame entity pointed to by SCK2-0.

At the stage where SCK2-0 is captured, the SCK2-1 index is entered in TCTR IDX and the SCK2-0 index is entered in CTR IDX. Also, the time interval from SCK2-0 to SCK2-1 is entered in TFRAME and RFRAME. The next SID is entered in SID.

This implements immediate sequence switching.

Chapter 3:

2D Graphics

TIM: Screen Image Data

The TIM file covers standard images handled by the PlayStation unit, and can be transferred directly to its VRAM. It can be used commonly as sprite patterns and 3D texture mapping materials.

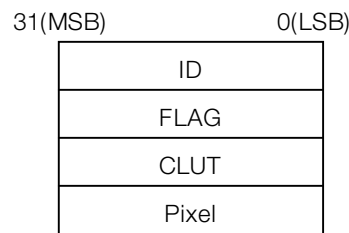
The following are the image data modes (color counts) handled by the PlayStation unit.

- 4-bit CLUT
- 8-bit CLUT
- 16-bit Direct color
- 24-bit Direct color

The VRAM supported by the PlayStation unit is based on 16 bits. Thus, only 16- and 24-bit data can be transferred directly to the frame buffer for display. Use as sprite pattern or polygon texture mapping data allows the selection of any of 4-bit, 8-bit and 16-bit modes.

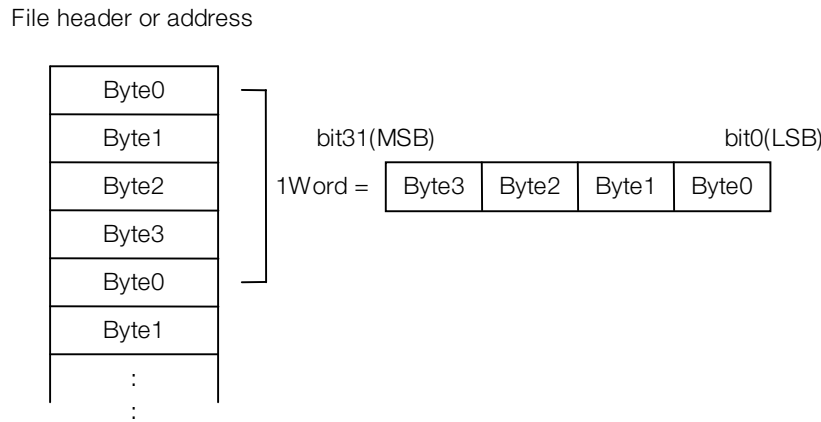
TIM files have a file header (ID) at the top and consist of several different blocks.

Figure 3-1: TIM File Format



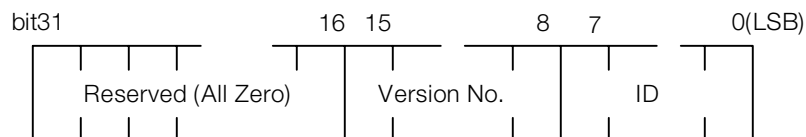
Each data item is a string of 32-bit binary data. The data is Little Endian, so in an item of data containing several bytes, the bottom byte comes first (holds the lowest address), as shown in Figure 3-2.

Figure 3-2: The order of bytes in a file



ID

The file ID is composed of one word, having the following bit configuration.

Figure 3-3: Structure of TIM File Header

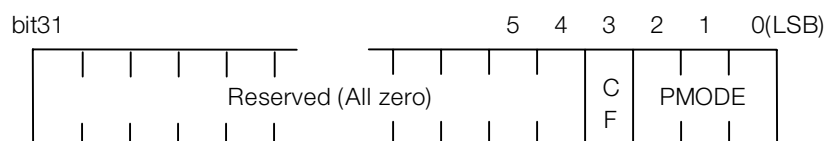
Bits 0 – 7: ID value is 0x10

Bits 8 – 15: Version number. Value is 0x00

Flag

Flags are 32-bit data containing information concerning the file structure. The bit configuration is as in Figure 3-4.

When a single TIM data file contains numerous sprites and texture data, the value of PMODE is 4 (mixed), since data of multiple types is intermingled.

Figure 3-4: Flag Word

Bits 0 -3 (PMODE): Pixel mode (Bit length)

0: 4-bit CLUT

1: 8-bit CLUT

2: 15-bit direct

3: 24-bit direct

4: Mixed

Bit 4 (CF): Whether there is a CLUT or not

0: No CLUT section

1: Has CLUT section

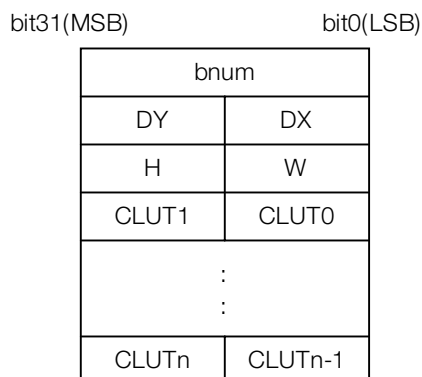
Other: Reserved

CLUT

The CF flag in the FLAG block specifies whether or not the TIM file has a CLUT block. A CLUT is a color palette, and is used by image data in 4-bit and 8-bit mode.

As shown in Figure 3-5, the number of bytes in the CLUT (bnum) is at the top of the CLUT block. This is followed by information on its location in the frame buffer, image size, and the substance of the data.

Figure 3-5: CLUT



bnum Data length of CLUT block. Units: bytes. Includes the 4 bytes of bnum

DX x coordinate in frame buffer

DY y coordinate in frame buffer

H Size of data in vertical direction

W Size of data in horizontal direction

CLUT 1~n CLUT entry (16 bits per entry)

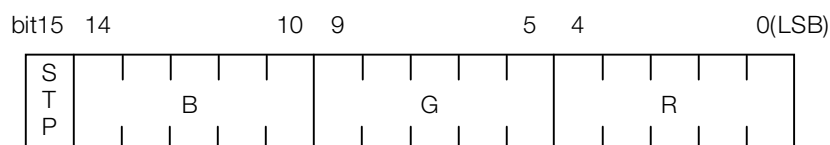
In 4-bit mode, one CLUT consists of 16 CLUT entries. In 8-bit mode, one CLUT consists of 256 CLUT entries.

In the PlayStation system, CLUTs are located in the frame buffer, so the CLUT block of a TIM file is handled as a rectangular frame buffer image. In other words, one CLUT entry is equivalent to one pixel in the frame buffer. In 4-bit mode, one CLUT is handled as an item of rectangular image data with a height of 1 and a width of 16; in 8-bit mode, it is handled as an item of rectangular image data with a height of 1 and a width of 256.

One TIM file can hold several CLUTs. In this case, the area in which several CLUTs are combined is placed in the CLUT block as a single item of image data.

The structure of a CLUT entry (= one color) is as follows:

Figure 3-6: A CLUT entry



STP Transparency control bit

R Red component (5 bits)

G Green component (5 bits)

B Blue component (5 bits)

The transparency control bit (STP) is valid when data is used as Sprite data or texture data. It controls whether or not the relevant pixel, in the Sprite or polygon to be drawn, is transparent. If STP is 1, the pixel is a semitransparent color, and if STP is other than 1, the pixel is a non-transparent color.

R, G and B bits control the color components. If they all have the value 0, and STP is also 0, the pixel will be a transparent color. If not, it will be a normal color (non-transparent).

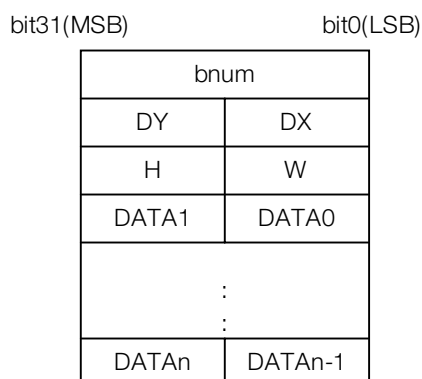
These relationships can be represented in a table as follows:

Table 3-1: STP Bit Function in Combination with R, G, B Data

STP/R,G,B	Translucent processing on	Translucent processing off
0/0,0,0	Transparent	Transparent
0/X,X,X	Not transparent	Not transparent
1/X,X,X	Semi-transparent	Not transparent
1/0,0,0	Non-transparent black	Non-transparent black

Pixel Data

Pixel data is the substance of the image data. The frame buffer of the PlayStation system has a 16-bit structure, so image data is broken up into 16-bit units. The structure of the pixel data block is as shown below.

Figure 3-7: Pixel data

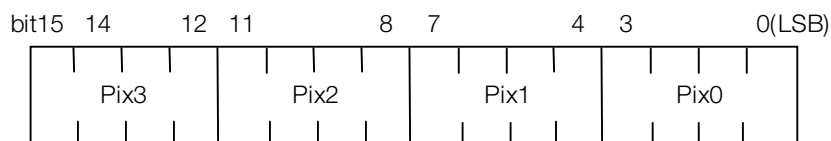
- bnum Data length of pixel data. Units: bytes. Includes the 4 bytes of bnum
- DX Frame buffer x coordinate
- DY Frame buffer y coordinate
- H Size of data in vertical direction
- W Size of data in horizontal direction (in 16-bit units)
- DATA 1~n Frame buffer data (16 bits)

The structure of one item of frame buffer data (16 bits) varies according to the image data mode. The structure for each mode is shown in Figure 3-8.

Care is needed when handling the size of the pixel data within the TIM data. The W value (horizontal width) in Figure 3-7 is in 16-pixel units, so in 4-bit or 8-bit mode it will be, respectively, 1/4 or 1/2 of the actual image size. Accordingly, the horizontal width of an image size in 4-bit mode has to be a multiple of 4, and an image size in 8-bit mode has to be an even number.

Figure 3-8: Frame buffer data (pixel data)

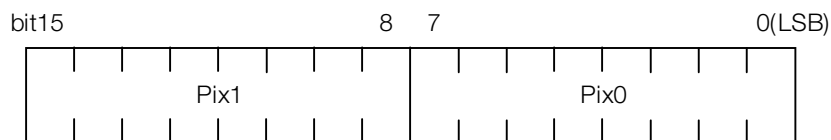
(a) In 4-bit mode



pix 0-3 pixel value (CLUT No.)

The order on the screen is pix0, 1, 2, 3, from the LSB side.

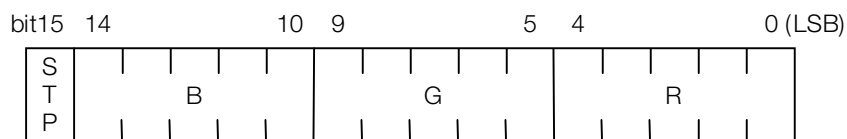
(b) In 8-bit mode



pix 0-1 pixel value (CLUT No.)

The order on the screen is pix0, 1, from the LSB side.

(c) In 16-bit mode



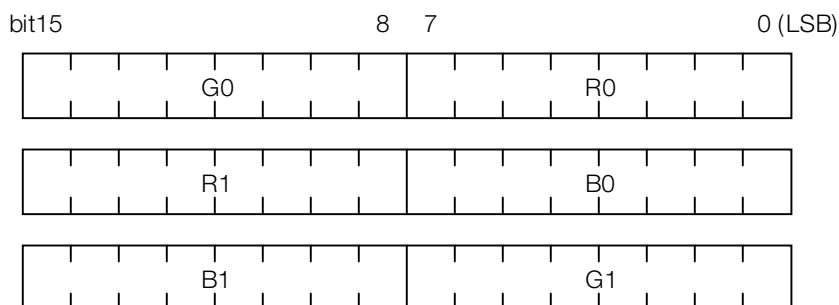
STP transparency control bit (see CLUT)

R Red component (5 bits)

G Green component (5 bits)

B Blue component (5 bits)

(d) In 24-bit mode:



R0, R1 Red component (8 bits)

G0, G1 Green component (8 bits)

B0, B1 Blue component (8 bits)

In 24-bit mode, 3 items of 16-bit data correspond to 2 pixels' worth of data. (R0, G0, B0) indicate the pixels on the left, and (R1, R2, B1) indicate the pixels on the right.

SDF: Sprite Editor Project File

The SDF file stores settings and file groups created and edited by the PlayStation sprite editor and enables all linked files to be loaded together.

The SDF file is an ASCII text file composed of seven blocks of information, as shown in Figure 3-9. Each block is designated by a unique keyword that begins each line within the block. For some blocks, a bank number, ranging from 0-3, is appended to the block keyword. Some blocks use only one bank of data while others use four. Following the keyword and bank number is a list of parameters. For those blocks with four banks of data, each bank must be specified, even if no parameters are given.

Figure 3-9: SDF File Structure

TIM .
CEL
MAP .
ANM
DISPLAY
COLOR
ADDR . .

The block is composed of lines assigned key word values.

Sample SDF File Contents

```
TIM0 file0.tim
TIM1 file1.pxl file1.clt
TIM2
TIM3
CEL0 file2.cel
MAP0 file3.bgd
MAP1 file4.bgd
MAP2
MAP3
ANM0 file5.anm
DISPLAY 1
COLOR0
ADDR0 768 0 0 480 16
ADDR1 768 256 0 496 16
ADDR2 512 0 256 480 16
ADDR3 512 256 256 496 16
```

TIM

The keyword of the TIM block is “TIM?” where “?” is a bank number from 0 to 3. All four banks must be specified. Following the keyword is the name of a TIM file, or the name of separate PXL and CLT files. If no data is required for a bank, the remainder of the line is left blank. For example:

```
TIM0 file0.tim
TIM1 file1.pxl file1.clt
TIM2
TIM3
```

Note: The key word of a bank not used must not be omitted, but assigned an item having no value.

CEL

The keyword of the CEL block is “CEL0”. There is only one bank of data. Following the keyword is the name of a CEL file. If no data is required for the CEL block, the remainder of the line following is left blank. For example:

```
CEL0 file0.tim
```

if no data is required:

```
CEL0
```

MAP

The keyword of the MAP block is “MAP?” where “?” is a bank number from 0 to 3. All four banks must be specified. Following the keyword is a filename. If no data is required for a bank, the remainder of the line is left blank. For example:

```
MAP0 file3.bgd
MAP1 file4.bgd
MAP2
MAP3
```

Note: The key word of a bank not used must not be omitted, but assigned an item having no value.

ANM

The keyword of the ANM block is “ANM0”. There is only one bank of data. Following the keyword is the name of a ANM file. If no data is required for the ANM block, the remainder of the line following is left blank. For example:

```
ANM0 file5.anm
```

if no data is required:

```
ANM0
```

DISPLAY

A DISPLAY block specifies and Artist Boards screen mode. The keyword for this block is DISPLAY. The argument value depends on the desired artist board screen mode as shown in Table 3-2.

Table 3-2: Display

Image mode	Value
256x240	0
320x240	1
512x240	2
640x240	3
256x480	4
320x480	5
512x480	6
640x480	7

For example:

```
DISPLAY 1
```

This would set the artist board to the 320x240 resolution mode.

COLOR

The COLOR block specifies the color mode. The keyword is COLOR. The value depends on the desired color mode, as shown in Table 3-3.

Table 3-3: Color

Color Mode	Value
16	0
256	1

For example:

```
COLOR 1
```

This would set the artist board to the 256 color mode.

ADDR

An ADDR block specifies the coordinates of images specified in the corresponding TIM bank, as well as the palette coordinates, and the number of color sets. The keyword of the ADDR block is "ADDR?" where "?" is a bank number from 0 to 3. All four banks must be specified. Following the keyword is a parameter list. The parameters for the ADDR block are as follows:

```
ADDR? X Y CX CY N
```

X: X coordinate of TIM image

Y: Y coordinate of TIM image

CX: X coordinate of palette

CY: Y coordinate of palette

N: Number of color sets

All parameters must be specified. For example:

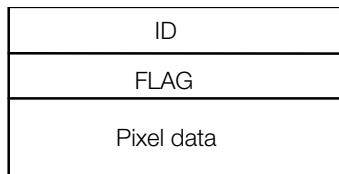
```
ADDR0 768 0 0 480 16
ADDR1 768 256 0 496 16
ADDR2 512 0 256 480 16
ADDR3 512 256 256 496 16
```

PXL: Pixel Image Data

The PXL format stores 4-bit or 8-bit indexed-color graphics images created and edited by the PlayStation sprite editor. Palette information is not included (this is contained within a CLT file that is used together with the PXL file).

The PXL format has a simple header containing an ID field and a FLAG field. After that comes raw pixel data. All values are stored in little-endian format. Bytes for 16-bit or 32-bit values are stored in ascending order (i.e. a 32-bit value would be stored byte 0, byte 1, byte 2, then byte 3).

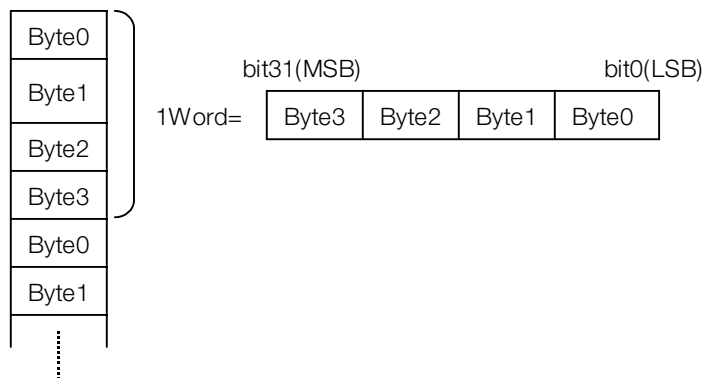
Figure 3-10: PXL File Structure



Data is of the 32-bit binary format. Because of LittleEndian, bytes are arranged in ascending order. (see Figure 3-11).

Figure 3-11: Byte Order in File

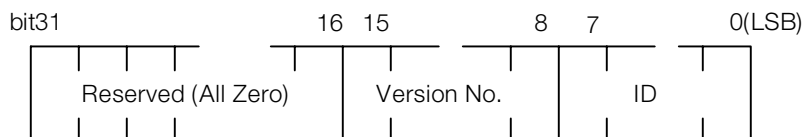
File header or address



ID

The ID field is a 32-bit value with the following bit definition:

Figure 3-12: Structure of PXL File Header

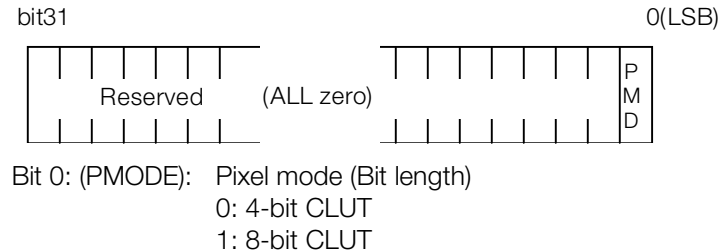


Bit 0 – 7: ID value is 0x11
 Bit 8 – 15: Version number. Value is 0x00

FLAG

The FLAG field is a 32-bit value containing information about the pixel data format. It has the following bit definition:

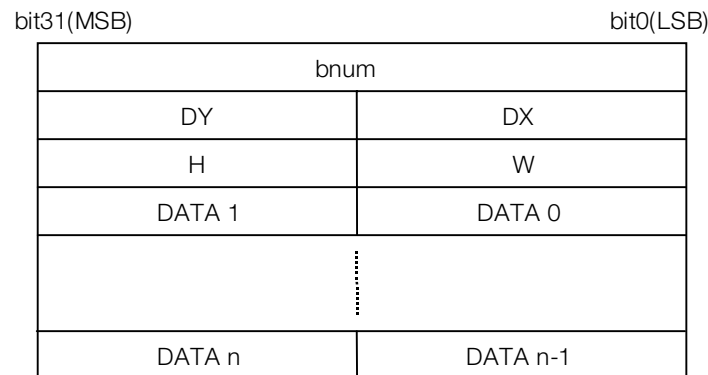
Figure 3-13: FLAG Bit Configuration



Pixel Data

The pixel data section contains the actual image information. It includes a short header as shown below, followed by the raw image data that is ready to be copied to the PlayStation's VRAM.

Figure 3-14: Configuration of Pixel Data Section

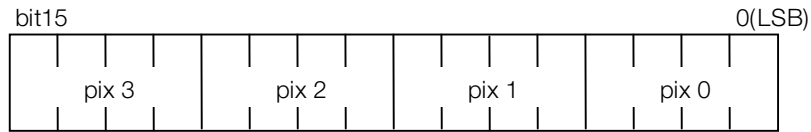


bnum Length of pixel data in bytes, including the 4 bytes of *bnum*
 DX Frame buffer x coordinate
 DY Frame buffer y coordinate
 H Size of data in vertical direction
 W Size of data in horizontal direction
 DATA VRAM (16 bits)

The configuration of a piece of VRAM data (16 bits) depends on the mode. The following gives the configuration in each mode.

Figure 3-15: VRAM Data (Pixel Data)

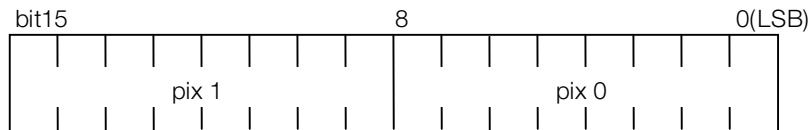
a) In 4-bit mode



pix 0-3 pixel value (CLUT No.)

The order on the screen is pix0, 1, 2, 3, starting from the left.

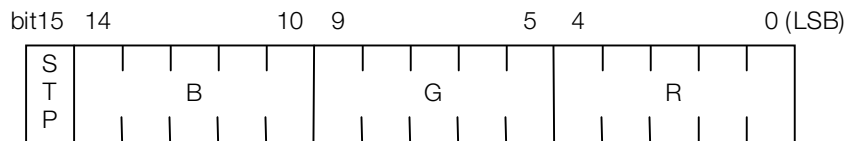
b) In 8-bit mode



pix 0-1 pixel value (CLUT No.)

The order on the screen is pix0, 1, starting from the left.

(c) In 16-bit mode



STP Transparency control bit (see CLUT)

R Red component (5 bits)

G Green component (5 bits)

B Blue component (5 bits)

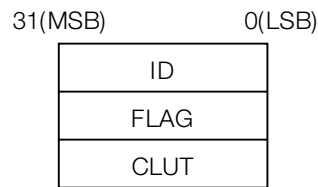
The coordinate system for the VRAM is based on 16 bits per pixel. Thus, note coordinate/size values in TIM data. In the X axis direction, a value of 2/4 is H in the 4-bit mode, and a value of 1/2 is H in the 8-bit mode. This means that, in the 4-bit mode, the image size must be a multiple of 4 and, in the 8-bit mode, the image size must be even.

CLT: Palette Data

The CLT file saves 8-bit or 4-bit palette data edited by the PlayStation sprite editor.

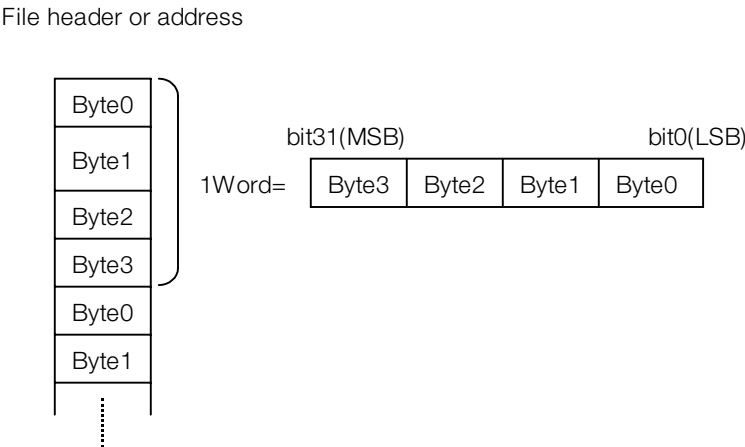
The CLT format has a simple header containing an ID field and a FLAG field. After that comes raw pixel data. All values are stored in little-endian format. Bytes for 16-bit or 32-bit values are stored in ascending order (i.e. a 32-bit value would be stored byte 0, byte 1, byte 2, then byte 3).

Figure 3-16: CLT File Structure



Data is of the 32-bit binary format. Because of LittleEndian, bytes are arranged in ascending order. (See Figure 3-17.)

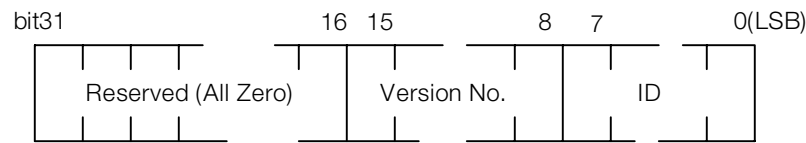
Figure 3-17: Byte Order in File



ID

The ID field is a 32-bit value with the following definition:

Figure 3-18: Structure of CLT File Header

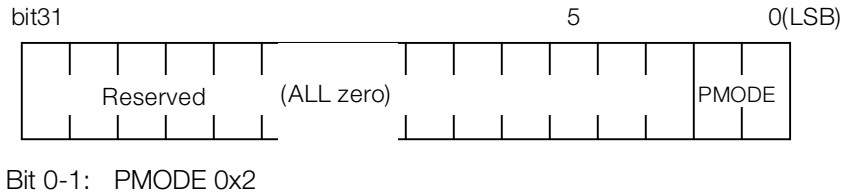


- Bit 0–7: ID value is 0x12
- Bit 8–15: Version number. Value is 0x00

FLAG

The FLAG field is a 32-bit value containing information about the pixel data format. It has the following bit definition:

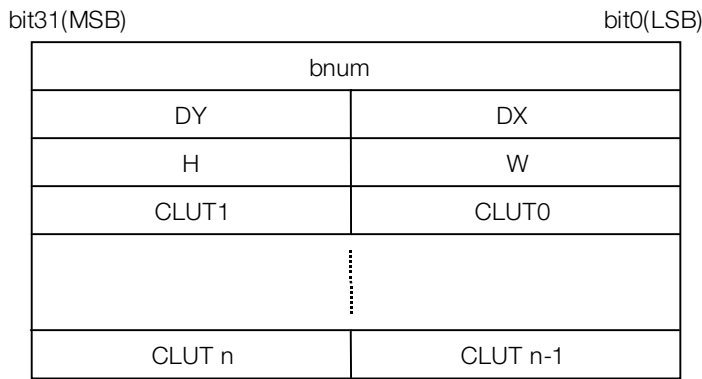
Figure 3-19: FLAG Bit Configuration



CLUT Section

The CLUT section begins with data on its byte count (bnum), followed by inner-VRAM positional information, index size and the main data body.

Figure 3-20: Structure of CLUT Section



- bnum Data length of CLUT block
- DX x coordinate in frame buffer
- DY y coordinate in frame buffer
- W Size of data in horizontal direction
- CLUT VRAM data (16 bits per entry)

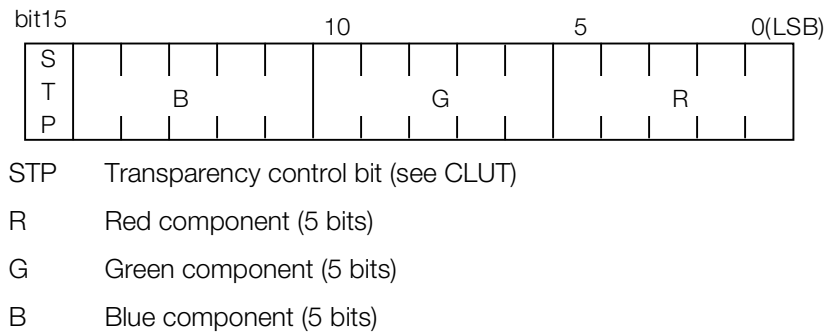
One CLUT set is composed of 16 CLUT entries in the 4-bit mode and of 256 CLUT entries in the 8-bit mode. (However, one file is composed of 16 sets of CLT data output from the sprite editor in the 4-bit mode.)

As CLUT is located on the VRAM, the PlayStation system handles the CLUT section in a TIM file as a rectangular VRAM image. This means that one CLUT entry is equivalent to one pixel in the VRAM. Thus, one CLUT set is handled as rectangular image data having a height of 1 and a width of 16 in the 4-bit mode and a height of 1 and a width of 256 in the 8-bit mode. (CLUT data output from the sprite editor is a rectangular image with a height of 16 and a width of 16 in the 4-bit mode.)

One TIM file can contain two or more CLUT sets. The area composed of two or more CLUT sets is considered to be a piece of image data and written in the CLUT section.

A CLUT entry, which expresses one color, has the following configuration.

Figure 3-21: CLUT Entry



The transparency control bit is applicable to sprite and texture data. If all of R, G, B and STP are zero, the color is regarded as being transparent. If not, the color is considered to be opaque.

For semitransparency processing, if the STP value is 1, the color is considered to be semitransparent. If not, the color is regarded as being opaque. (Only in all zeros, the color is considered to be transparent.)

Table 3-4: Role of STP Bit

STP/R,G,B	Semi-transparency processing on	Semi-transparency processing off
0/0,0,0	Transparent	Transparent
0/X,X,X	Not transparent	Not transparent
1/X,X,X	Semi-transparent	Not transparent

ANM: Animation Information

The ANM format contains information that specifies image data animation. An ANM file is typically used in conjunction with a TIM file, which contains the actual screen information.

The ANM file has a header at the top, and is divided into four blocks.

Figure 3-22: ANM file format

HEADER
SEQUENCE
SPRITEGp
CLUTGp

HEADER

This is the file header. Its structure is shown below.

Figure 3-23: File Header

31 (MSB)	16	8	0 (LSB)
FLAG	VERSION	ID	
NSEQUENCE	NSPRITEGp		

ID: 0x21

VERSION: 0x03

FLAG: See below

NSEQUENCE: Number of sequences

NSPRITEGp: Number of sprite groups

Figure 3-24: FLAG

(VERSION<2)



CLT Number of CLUTs for color animation

(VERSION>=2)



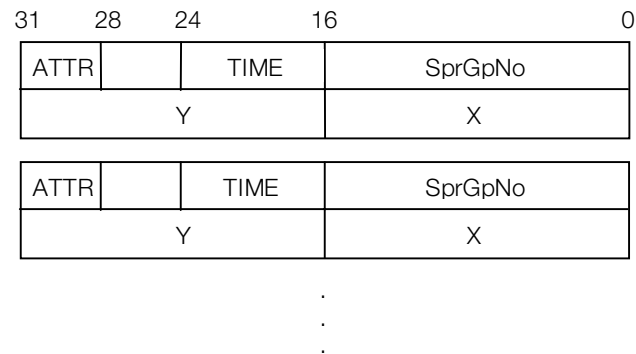
CLT Number of CLUTs for color animation

TPF Texture pattern pixel depth
 00:4-bit
 01:8-bit

SEQUENCE

Sequence data provides a set of coordinates of the hot spots in the frames, display time and sprite group numbers.

Figure 3-25: Sequence



TIME Display time (number of repetitions)

SprGpNo Number of Sprite group to be displayed, from among the sprit group data

X X coordinate of hot spot

Y Y coordinate of hot spot

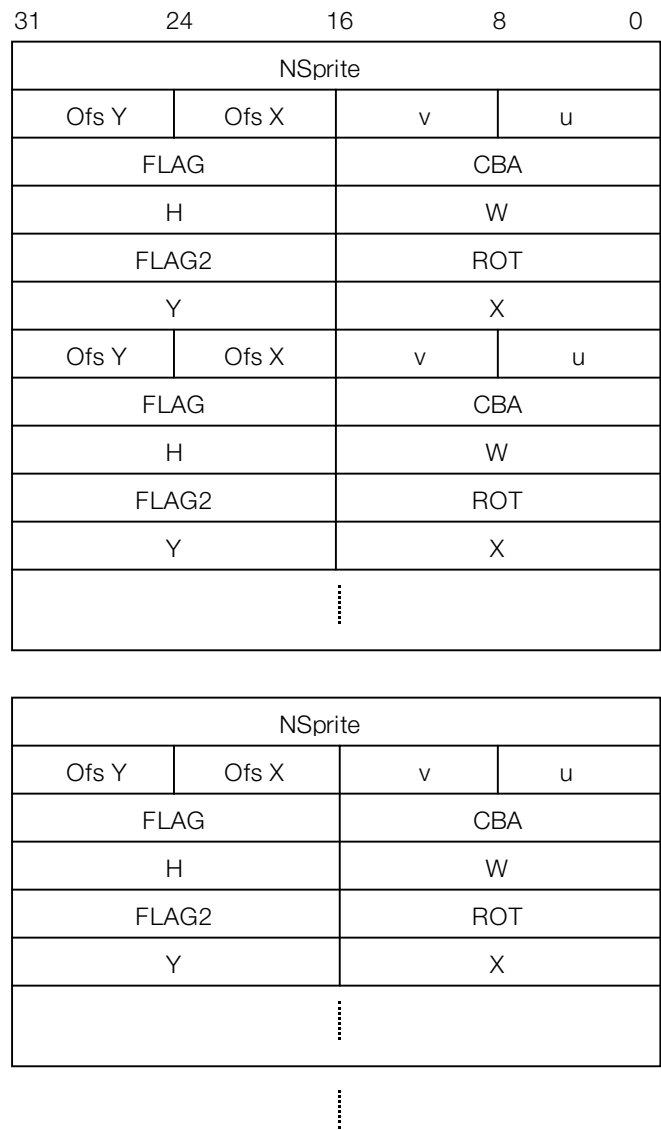
ATTR Attribute (user defined)

(VERSION<3): no ATTR

SPRITEGp

Sprite group data is a set of sprite groups, describing where a sprite is to be displayed in a frame.

Figure 3-26: SPRITEGp

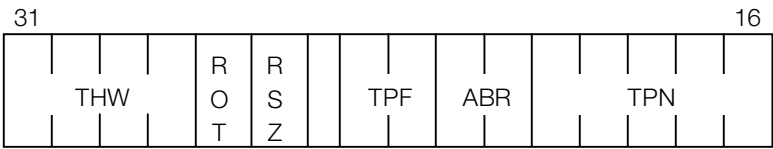


- NSprite Number of Sprites in one Sprite frame
- FLAG See Figure 3-27
- v Vertical offset from base address of texture page
- u Horizontal offset from base address of texture page
- Ofs Y Vertical offset from hotspot within frame
- Ofs X Horizontal offset from hotspot within frame
- CBA See Figure 3-28
- H Width of texture of optional size
- W Height of texture of optional size

- ROT Angle of rotation
- FLAG2 See Figure 3-29
- Y, X Scaling factor (specified as a fixed-point number)

FLAG has the following bit configuration.

Figure 3-27: FLAG

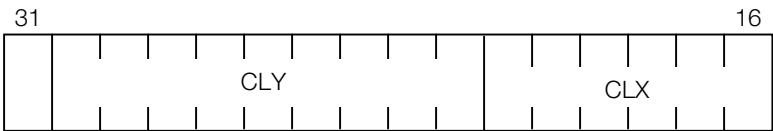


- THW The size of the rectangular area of the Sprite, divided by 8 (if it cannot be divided by 8, this bit is set to 0x0 and the actual size is specified using H and W, described earlier.)
- ROT Rotation status
0: Not rotated
1: Rotated
- RSZ Scaling status
0: Not scaled
1: Scaled
- TPF Pixel depth of texture pattern
00: 4-bit CLUT
01: 8-bit CLUT
10: 16-bit
- ABR Semi-transparency rate.
00: 0.50xF+0x50xB
01: 1.00xF+1.00xB
10: -1.00xF+1.00xB
11: 0.25xF+1.00xB
- TPN Texture page number (0-31)

CBA has the following bit configuration.

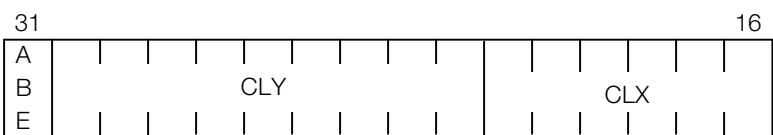
Figure 3-28: CBA

(VERSION<3)



- CLY Y coordinate of beginning of CLUT (9 bit)
- CLX X coordinate of beginning of CLUT (6 bit)

(VERSION>=3)



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

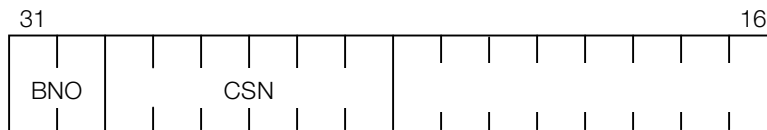
CLY Y coordinate of beginning of CLUT (9 bit)

CLX X coordinate of beginning of CLUT (6 bit)

FLAG2 has the following bit configuration.

Figure 3-29: FLAG2

(VERSION>=2)



BNO TIM bank number (Sprite editor)

CSN Color set number (Sprite editor)

Note:

In VERSION 0, TIM bank is:

In Bank 0, TPN=12

In Bank 1, TPN=28

In Bank 2, TPN=8

In Bank 3, TPN=24

In VERSION 1:

In CBA_x<255, CBA_y <=495, Bank is 0

In CBA_x<255, CBA_y >495, Bank is 1

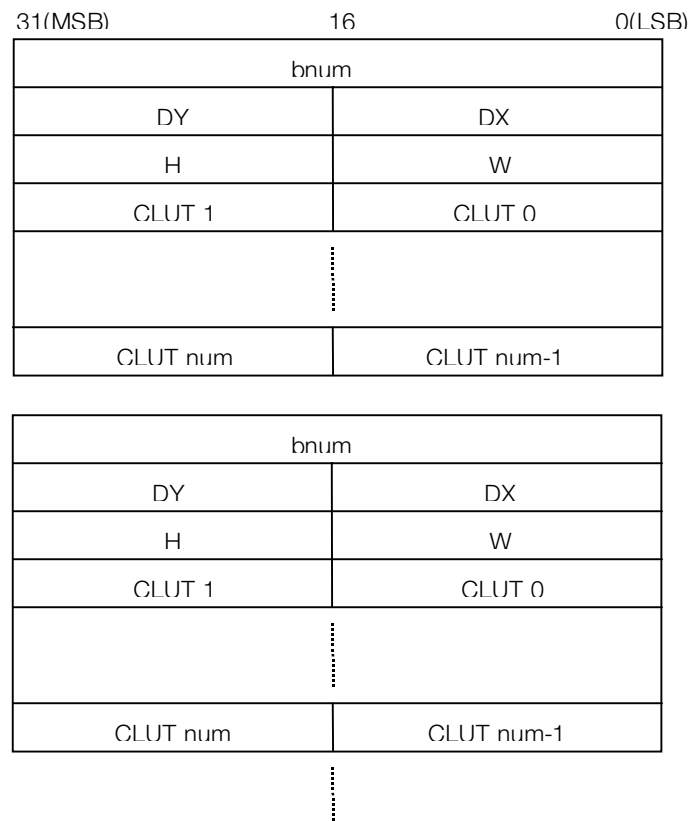
In CBA_x>=255, CBA_y <=495, Bank is 2

In CBA_x>=255, CBA_y >495, Bank is 3

CLUTGp

CLUT Gp is a group of CLUTs used for color animations. The number of CLUTs is specified by the CLT parameter of the FLAG in the HEADER block.

Figure 3-30: CLUTGp

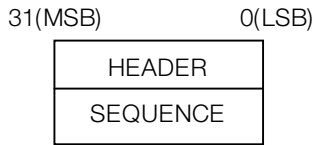


- bnum Data length of CLUT (in bytes)
- DX x coordinate in frame buffer
- DY y coordinate in frame buffer
- W Horizontal size of data
- H Vertical size of data
- CLUT 0~n CLUT entries (16 bits per entry)

TSQ: Animation Time Sequence

TSQ is a binary file that stores time sequence data for sprite animations created and edited by the PlayStation sprite editor. It has a short header composed of two blocks.

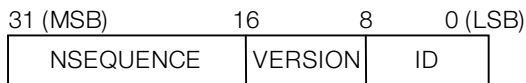
Figure 3-31: SEQ Data Structure



HEADER

The file header has the following configuration.

Figure 3-32: HEADER

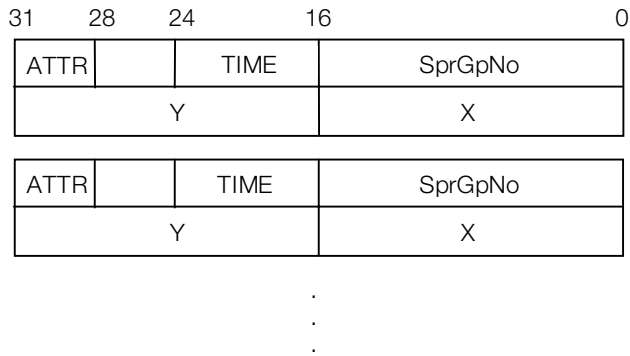


ID: 0x24
 VERSION: 0x01
 NSEQUENCE Sequence data count

SEQUENCE

Sequence data is a set of coordinates of the hot spots in the frames, display time, and sprite group numbers.

Figure 3-33: SEQUENCE



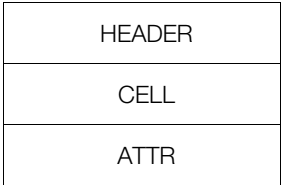
TIME Display time
 SprGpNo Number of Sprite group to be displayed
 X X coordinate of hot spot
 Y Y coordinate of hot spot
 ATTR Attribute (user defined)

(VERSION=0): no ATTR

CEL: Cell Data

CEL format stipulates the pointer table, in the VRAM, of the CELs forming constituents of the BG surface.
A CEL file has a header at the top, and is divided into three blocks.

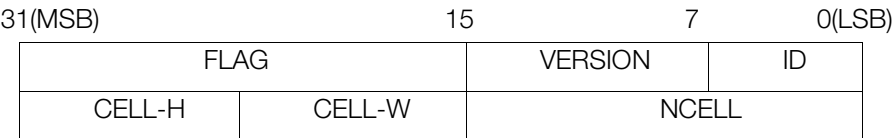
Figure 3-34: CEL file format



HEADER

The file header has the following configuration.

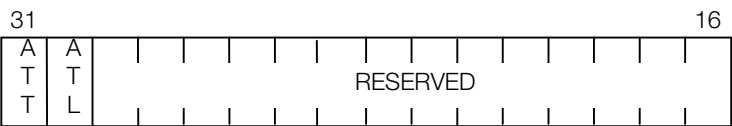
Figure 3-35: HEADER



- FLAG Described later
- ID 0x22
- VERSION 0x03
- NCELL Number of cell data items (units: cells)
- CELL-H Size of cell display window height (units: cells) (used locally in sprite editor)
- CELL-W Size of cell display window width (units: cells) (used locally in sprite editor)

FLAG has the following bit configuration.

Figure 3-36: FLAG



- ATT Indicates whether an ATTR block is included in this file
0: ATTR is not included
1: ATTR is included
- ATL Length of ATTR data
0: 8-bit
1: 16-bit

CELL

Cell data provides a table of VRAM pointers to cells constituting BG. Four bytes form one cell.

Figure 3-37: CELL Data Section

31(MSB)	16	8	0(LSB)
CBA	v	u	
TSB	FLAG		
CBA	v	u	
TSB	FLAG		
:			
:			

v Offset in vertical direction from base address of texture page (8 bits)

u Offset in horizontal direction from base address of texture page (8 bits)

CBA Described later

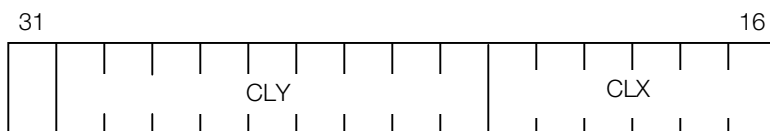
TSB Described later

FLAG Described later

CBA has the following bit configuration.

Figure 3-38: CBA

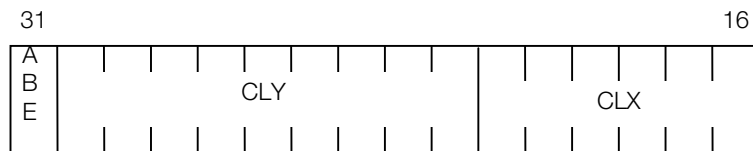
(VERSION<3)



CLY Y coordinate of beginning of CLUT (9 bits)

CLX X coordinate of beginning of CLUT (6 bits)

(VERSION>=3)



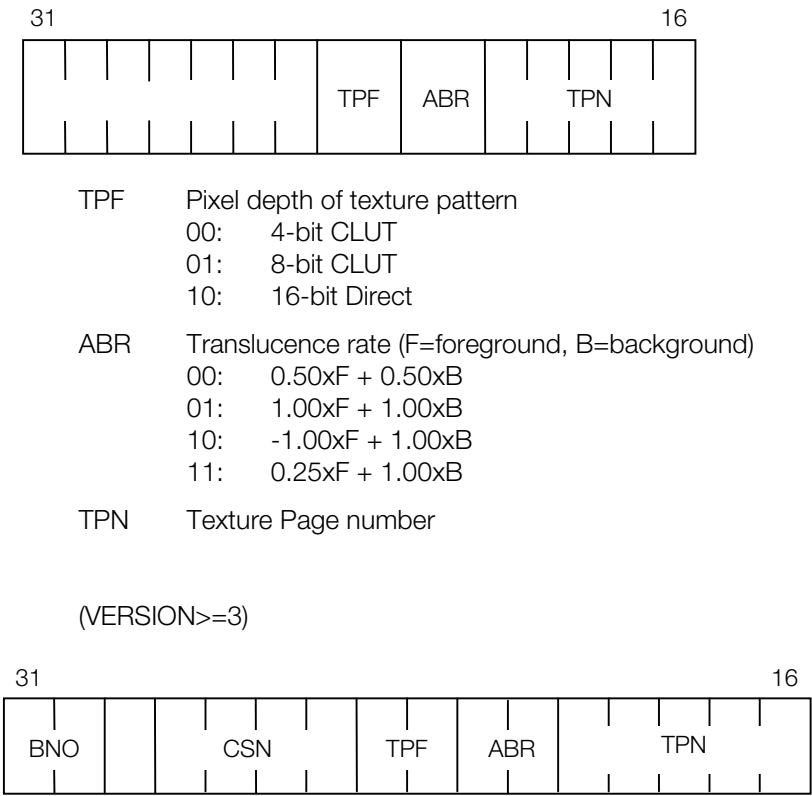
ABE 0 : Semi-transparency processing OFF
1 : Semi-transparency processing ON

CLY Y coordinate of beginning of CLUT (9 bits)

CLX X coordinate of beginning of CLUT (6 bits)

TSB has the following bit configuration.

Figure 3-39: TSB



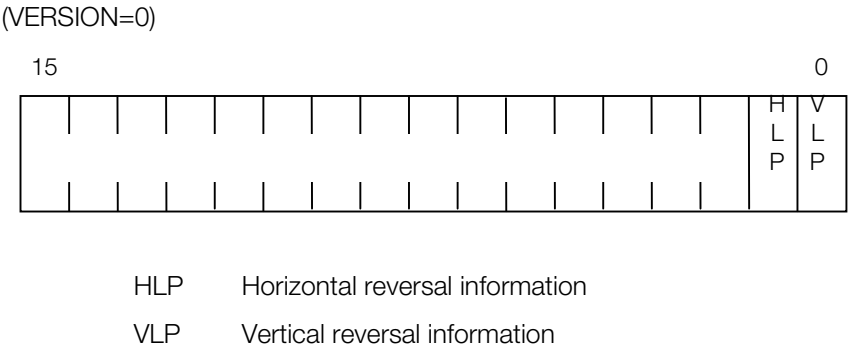
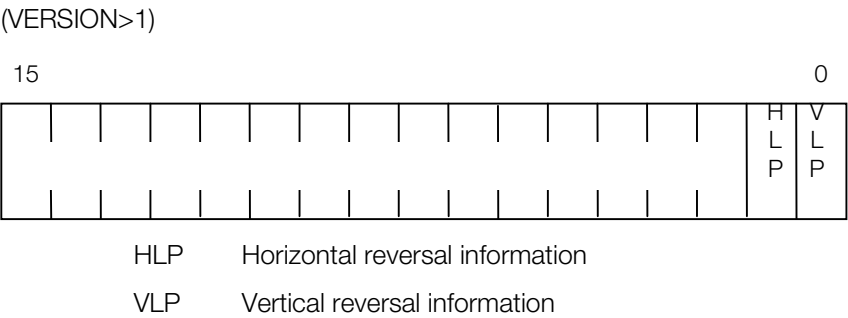
BNO: TIM bank (sprite editor) number
CSN: Color set (sprite editor) number

Note:
In VERSION 0, TIM bank is:
In Bank 0, TPN=12
In Bank 1, TPN=28
In Bank 2, TPN=8
In Bank 3, TPN=24

In VERSION 1 and 2:
In CBA_x<255, CBA_y <=495, Bank is 0
In CBA_x<255, CBA_y >495, Bank is 1
In CBA_x>=255, CBA_y <=495, Bank is 2
In CBA_x>=255, CBA_y >495, Bank is 3

FLAG has the following bit configuration.

Figure 3-40: FLAG



ATTR

Expresses attribute data. Attribute data is additional information concerning the cell and is arranged in the same order as CEL.

There are two types of attribute data, 8 bit length data and 16 bit length data and each is shown below. Data length is indicated in the Header section, in the ATL bit in the FLAG half word.

Figure 3-41: ATTR Format (8 Bit)

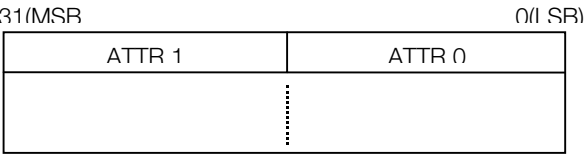
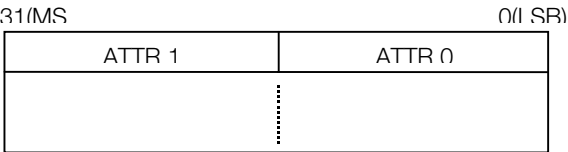


Figure 3-42: ATTR Format (16 Bit)



BGD: BG Map Data

The BGD file provides data constituting the BG plane used in the 2D system. BG refers to any row of rectangular pixel data. The BGD file is used along with the TIM and CEL files having the same name. Actual pixel images are carried by the TIM file.

The BGD file has a header at the top, and is divided into three blocks. The ATTR block may be omitted.

Figure 3-43: BG file format

HEADER
MAP
ATTR

HEADER

This is the file header. Its structure is as follows:

Figure 3-44: HEADER

31 (MSB)		0 (LSB)	
FLAG		VERSION	ID
CELLH	CELLW	MAPH	MAPW
ID	0x23		
VERSION	0X00		
FLAG	See Figure 3-45		
MAPH	Vertical size of BG map data (in cell units)		
MAPW	Horizontal size of cell BG map data (in cell units)		
CELLH	Vertical size of cell data (in pixel units)		
CELLW	Horizontal size of cell data (in pixel units)		

The structure of the FLAG in Figure 3-44 is as follows:

Figure 3-45: FLAG

31																16
A	A															
T	T															
T	L															
ATT	Indicates whether an ATTR block is included in this file 0: ATTR is not included 1: ATTR is included															
ATL	Length of ATTR data 0: 8-bit 1: 16-bit															

Chapter 4:

Sound

SEQ: PS Sequence Data

SEQ is the PlayStation sequence data format. The typical extension in DOS is “.SEQ”.

Figure 4-1: SEQ Format

	Byte count
ID (SEQp)	4
Version	4
Resolution of quarter note	2
Tempo	3
Rhythm	2
Score data	Any
End of SEQ	3

SEP: PS Multi-Track Sequence Data

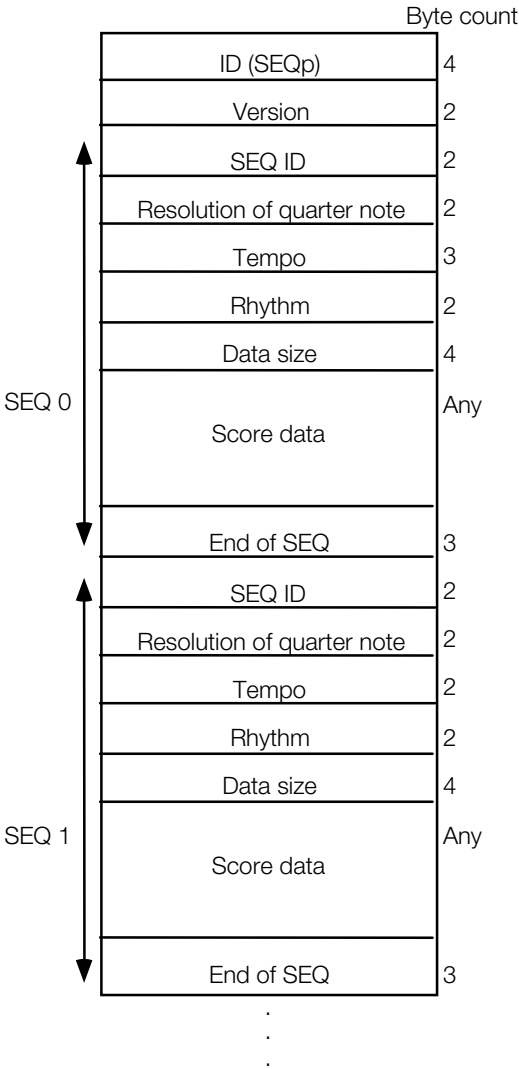
A SEP is a package containing multiple SEQ data files. SEPs enable multiple SEQ data files to be managed as one file.

SEPs can be accessed by specifying the ID number returned when the SEP is opened, along with the SEQ number of the SEQ data to be accessed.

For details of access-related functions, see the *Run-time Library Reference*.

The SEP data format is illustrated on the next page.

Figure 4-2: SEP Format



VAG: PS Single Waveform Data

VAG is the PlayStation single waveform data format for ADPCM-encoded data of sampled sounds, such as piano sounds, explosions, and music. The typical extension in DOS is “.VAG”.

Figure 4-3: VAG Format

	Byte count
ID (VAGp)	4
Version	4
Reserved	4
Data size (Bytes)	4
Sampling frequency	4
Reserved	12
Name	16
Waveform data	Any

VAB: PS Sound Source Data

The VAB file format is designed to manage multiple VAG files as a single group. It is a sound processing format that is handled as a single file at runtime.

A VAB file contains all of the sounds, sound effects, and other sound-related data actually used in a scene. Hierarchical management is used to support multitimbral (multisampling) functions.

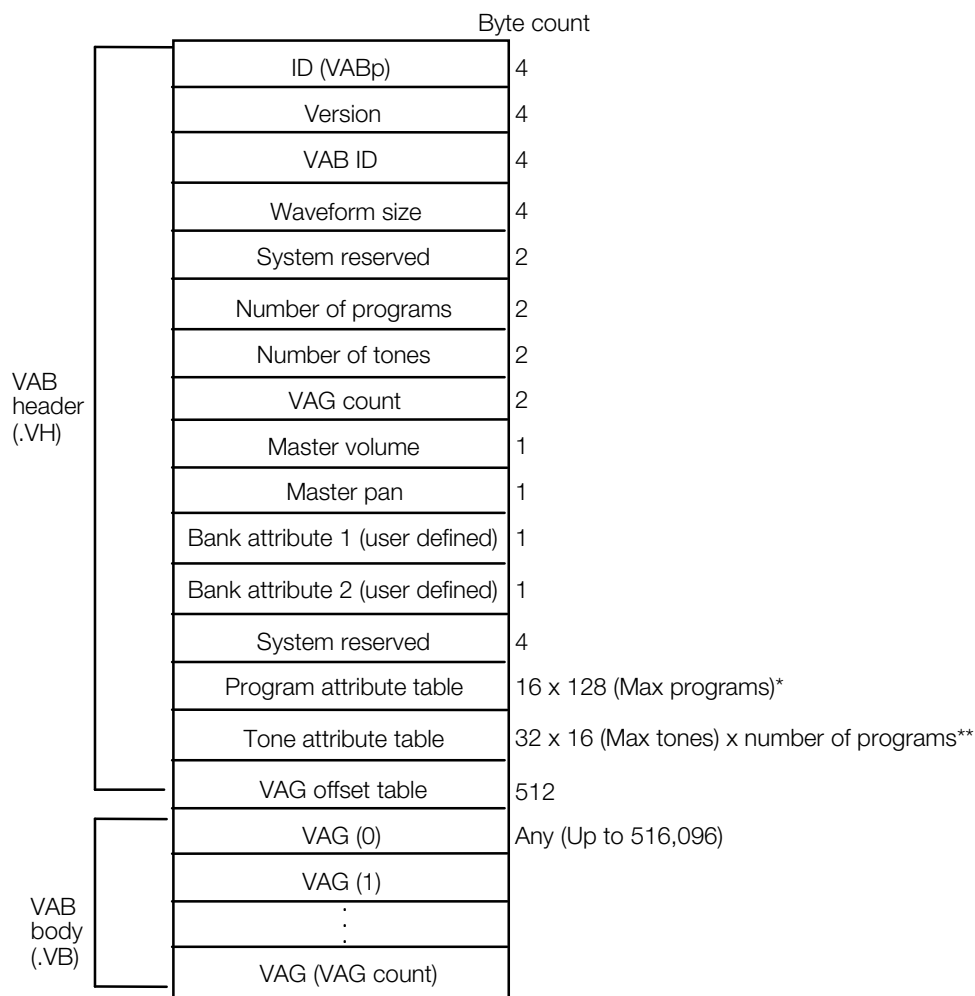
Each VAB file may contain up to 128 programs. Each of these programs can contain up to 16 tone lists. Also, each VAB file can contain up to 254 VAG files.

Since it is possible for multiple tone lists to reference the same waveform, users are able to set different playback parameters for the same waveform, thus giving the same waveform different sounds.

Organization

A VAB format file is organized as follows:

Figure 4-4: VAB Format



* See (b) in Structure

** See (c) in Structure

Structure

The structure of a VAB header is as follows. It is possible to set each attribute dynamically using this structure at the time of execution.

- (a) VabHdr structure is contained within the first 32 bytes (see libsnd in the Library Reference for details).
- (b) ProgAtr structure for 128 programs is contained in the program attribute table (see libsnd in the Library Reference for details).
- (c) VagAtr structure for each tone is contained in the tone attribute table (see libsnd in the Library Reference for details).
- (d) VAG offset table contains 3-bit right-shifted VAG data size stored in short (16 bit). For example:

Table 4-1: VAG Offset Table

VAG#	0	1	2	...
VAG offset table	0x1000	0x0800	0x0200	...
Actual size	0x8000	0x4000	0x1000	...
Offset	0x8000	0xc000	0xd000	...

DA: CD-DA Data

DA is the PlayStation CD-DA data format. The typical extension in DOS is “.DA”.

Figure 4-5: DA Format

