Hardware Control Library for the IRIS/COSMOS/HOLLY

KAMUI

User's Manual

Version 1.34

Approved by	Checked by	Created by

CONTENTS

1. OVERVIEW

KAMUI is a driver maintenance layer used by drivers and some other software in a system in which the second-generation PowerVR chip is used. These drivers and software are expected to run at a high speed in this system.

KAMUI supports the following environments:

a) IRIS (Tiling accelerator for PC) + ARC1 evaluation board

b) COSMOS (Tiling accelerator for SH4) + ARC1 evaluation board

c) SH4 HOLLY (CLX1): Incorporates a tiling accelerator.

d) SH4 HOLLY (CLX2): Incorporates a tiling accelerator.

The basic input/output parameters for KAMUI remain the same in any of the above environments, except that some functions are unavailable in items a) and b) because the ARC1 does not have functions such as Bump Mapping and Trilinear Filter, but the HOLLY does.

KAMUI supports the following functions:

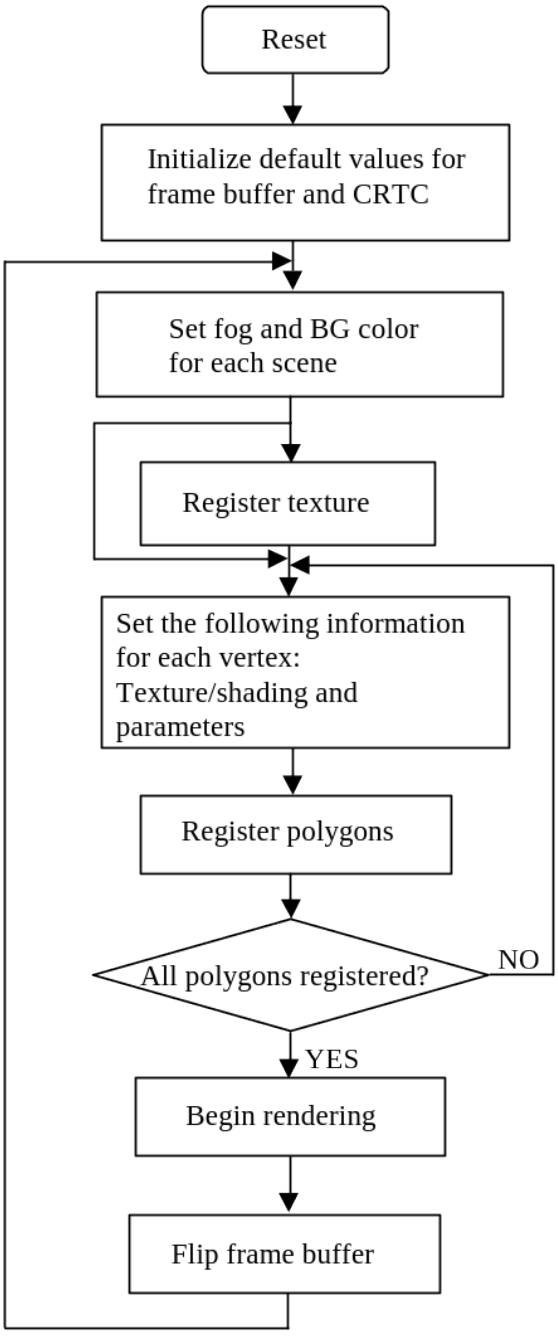
- Registering trigonal polygon strips and tetragonal polygons with scenes.
- Setting rendering conditions (context) for each polygon (strip) separately.
- Handling the frame buffer
- Handling the texture
- Setting various special effects (such as fog and modifier volume)
- Other service routines

KAMUI performs only minimum error checks. If invalid parameters (those not stipulated, for example) are given to KAMUI, its normal operation is not guaranteed.

2. PROCESSING FLOW OF KAMUI

2.1 BASIC PROCESSING FLOW

The following flowchart shows the basic processing flow of KAMUI.



After a reset occurs, an application must first set the frame buffer and display controller. The frame buffer is allocated two areas: Primary surface for ordinary display, and off-screen surface that is a rendering target of a nondisplay area. The frame buffer is configured in a double buffer system, in which the Flip command causes the two areas to be exchanged with each other.

After initializing the frame buffer, the application sets parameters (such as fog table data and background color) related to an entire scene.

Now, the application registers texture. This is done by simply transferring texture data from an area allocated in system memory to texture memory. (Texture memory will not receive texture data directly from file I/O units.) In the previous example, texture is registered after parameters related to the entire scene are registered. Texture registration can be done at any point in the above flowchart after an initialization. (Do not rewrite texture data during the course of rendering, however.)

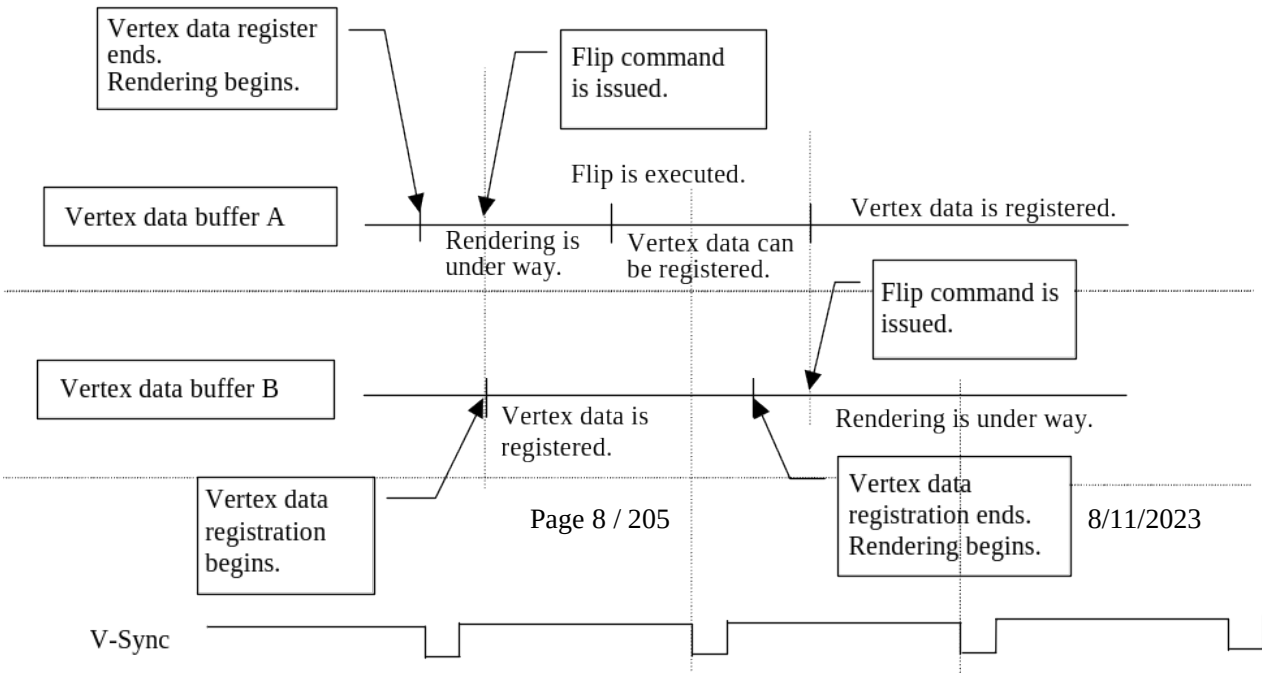
Next, the application sets rendering conditions (context) for each polygon to be registered (such as data format, texture, and texture filter). Registering all rendering conditions every time would be extremely inefficient in some cases. So, the application only has to set changes from the previous setting. After setting is completed, polygon vertex data is registered. **All vertex data is registered in the Strip format**. If you want to specify a single polygon, register it as a polygon with Strip = 1.

Upon completion of polygon registration, a rendering start command is issued. Processing to be performed at the end of rendering can be added using a callback function to be executed at the end of rendering.

The frame buffer can be flipped after the rendering start command is issued. When the frame buffer Flip command is issued, it is stacked in a queue. **If rendering is completed before the first V-Sync after the Flip command is executed, the primary surface and off-screen surface are exchanged automatically**. This way, continuous display becomes possible.

The Flip command is only enqueued. It does not wait for V-Sync to be detected. After executing the Flip command, a program utilizing KAMUI can start registering vertex data for the next scene immediately, as long as vertex data registration is possible (if the registration area is not currently being used for rendering). In KAMUI, vertex data for rendering is submitted to double buffering, similarly to the frame buffer. This is because vertex data being currently rendered and parameters being registered are necessary. Programmers need not be aware of that, however.

The following chart shows the timing when vertex data is registered, rendering is started, and the Flip command is issued.



The processing shown above is performed as pipelining. If the following conditions occur, wait states are inserted:

- An attempt is made to issue a rendering start command for the next scene when hardware rendering is under way (this can occur if processing ends within a short time because only a small amount of vertex data is registered after the Flip command is issued). Alternatively, an attempt is made to issue a rendering start command for the next scene when hardware rendering is under way (this can occur if hardware rendering takes much time).

Environment mapping requires that rendering be applied to rectangular texture. So, special frame buffer control is necessary. See the description of the frame buffer handling function for details.

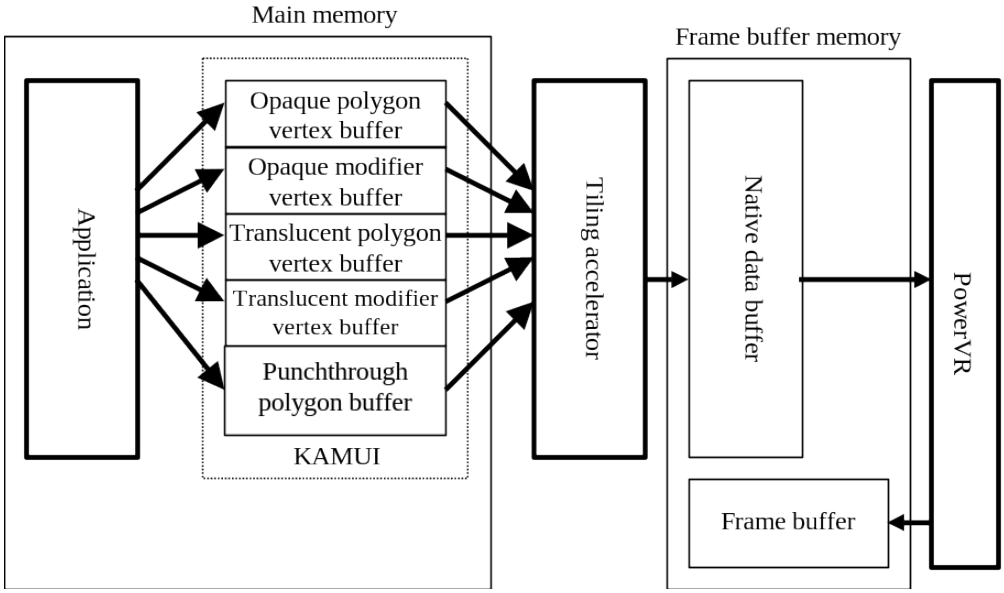
2.2 VERTEX DATA BUFFERS AND INTERNAL BUFFERS

To have KAMUI draw, the application program allocates a **vertex data buffer area** (for holding vertex data) in system memory, enters vertex data in the vertex data buffer area, and issues a rendering command. The vertex data buffer area is divided into the following five buffers:

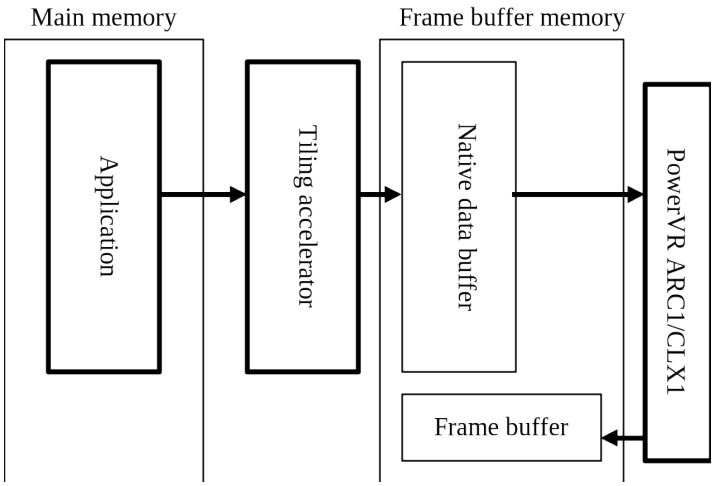
- Buffer for opaque polygon (opaque polygon)
- Buffer for opaque modifier volume (opaque modifier)
- Buffer for translucent/transparent polygon (translucent polygon)
- Buffer for translucent/transparent modifier volume (translucent modifier)
- Buffer for punchthrough polygon (punchthrough polygon)

Vertex data is held in any of the above buffers.

The application program must also allocate a **native data buffer** for temporarily holding PowerVR's native-format polygons. The native data buffer is allocated in frame buffer memory. These buffers and the related polygon data flow are shown below.



Vertex data can be written directly to the hardware (tiling accelerator) without allocating any of the five vertex data buffers. This method is referred to as the **direct mode**. The method explained before is known as the **buffer mode**. In the direct mode, the native data buffer is allocated in frame buffer memory. The related buffer and polygon data flow are shown below.



This method uses a smaller memory area than the other method. However, the application program must simultaneously supply each type of vertex data (opaque polygon, opaque modifier, translucent polygon, translucent modifier, and punchthrough polygon) for the same scene to the hardware (tiling accelerator).

See Section 3.6 for how to register vertex data.

2.3 LATENCY MODEL

KAMUI can select a model that matches specific rendering conditions by switching between latency models (using a changeover function). Two different latency models, 3V and 2V, can be selected. These latency models have different features and merits.

- 3V latency model
  - Enables the best performance to be achieved.

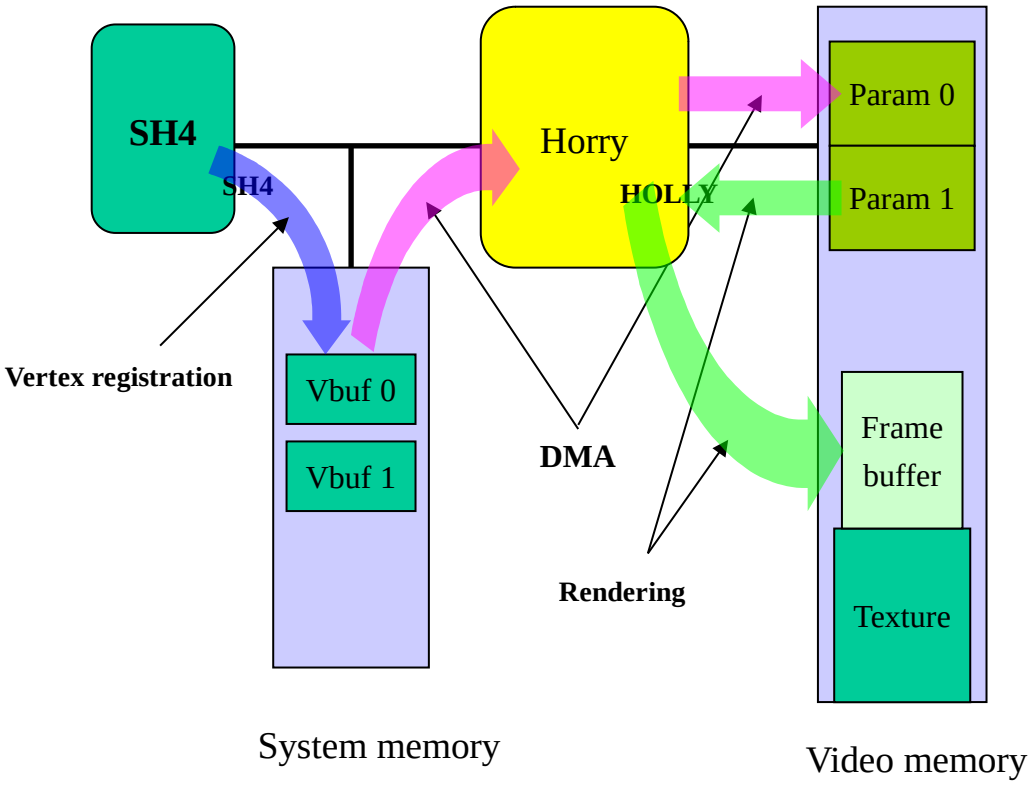
- 2V latency model
  - Eliminates the need to use a vertex buffer for polygons (for example, opaque polygons) of the type used most frequently **(to save system memory)**.
  - Assures a quick key entry response.

The type of vertex data registration function that can be used varies with the latency model currently being used. If a vertex data registration function that does not match the current latency model is used, KAMUI may behave incorrectly. Avoid using such a function.
Each latency model is described below.

2.4 3V LATENCY MODEL

2.4.1 3V Latency Model

The 3V latency model is a programming model intended to enable a KATANA system to achieve the best possible performance and maintain a constant latency ratio. One V period is assigned to vertex data registration, DMA (transfer to the TA), and rendering separately. The following shows how the programming pipeline behaves.



Programming pipeline

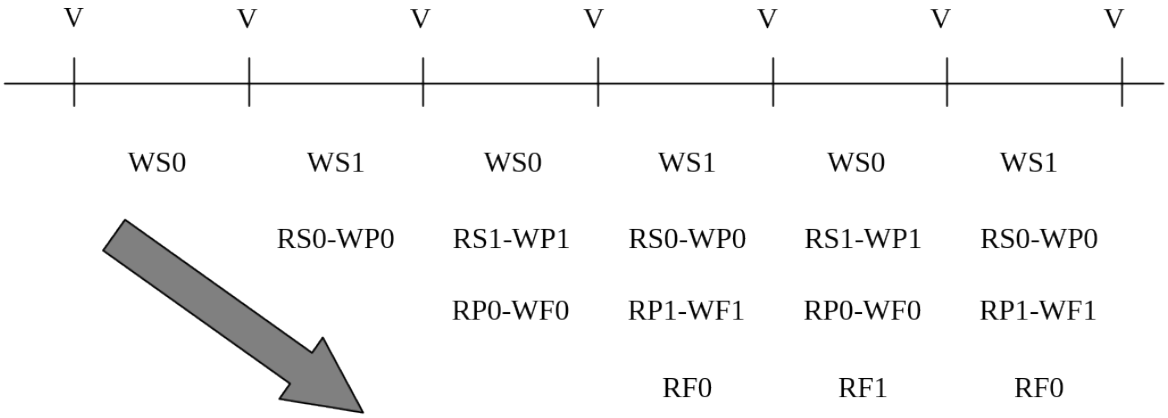
Vertex registration:	Represents the registration of the vertex data that constitutes a scene. It includes a game sequence, geometry, and the creation of data to be sent to the TA.
DMA:	Created data is sent to ARC1 system memory via the TA using a DMA mode.
Render:	Rendering is carried out as directed by parameters developed by the TA.
Disp:	The results of rendering are displayed.

Frame buffer double buffering requires the following hardware resources:

1. SH4 system memory side
   Two vertex data buffers (one buffer for one scene)
2. CLX (ARC1) side memory
   Two native data buffers that match CLX's internal parameters (one buffer for one scene)
   Two display frame buffers
   Texture memory

The pipeline mentioned above uses memory as follows:

WS?: Writing to system memory ? (0 or 1) by SH4
RS?: Reading from system memory ? (0 or 1) by DMA
WP?: Writing to CLX parameter memory ? (0 or 1) by DMA
RP?: Reading from CLX parameter memory ? (0 or 1) by renderer
WF?: Writing to CLX frame buffer ? (0 or 1) by renderer
RF?: Reading from CLX frame buffer ? (0 or 1) for display by renderer



Use of the pipeline does not cause contention for hardware resources and can minimize arbitration for 60 fps.

2.4.2 Usable Functions and How to Use Them

The following vertex data registration functions are used in the 3V latency model. **No direct-mode function can be used** at all because of the characteristics of the programming model.

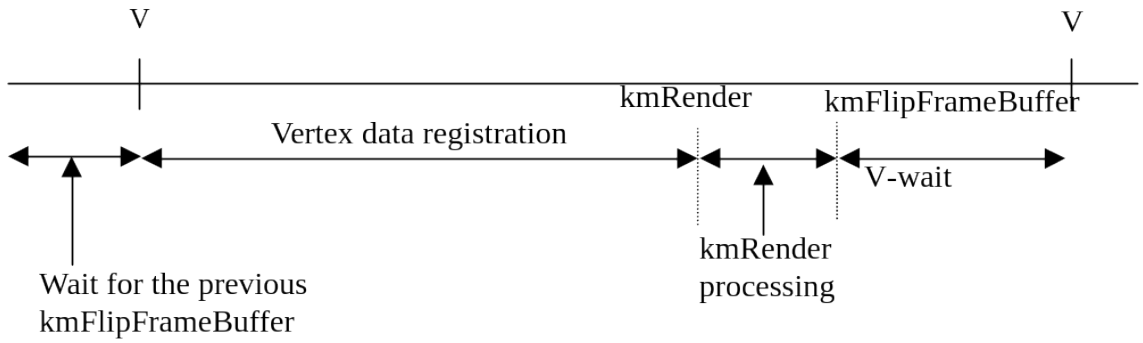
- kmProcessVertexRenderState
- kmSetVertexRenderState
- kmStartVertexStrip
- kmSetVertex
- kmRender
- kmFlipFrameBuffer

Processing flow example

```
kmChangeLatencyMode(3V_LATENCY, NULL);
/* Represents 3V latency */
……..
kmCreate VertexBuffer
kmProcessVertexRenderState
while(1){
    kmSetVertexRenderState
    kmStartVertexStrip
    kmSetVertex
    kmSetVertex
    …….
    kmRender
                                                    kmFlipFrameBuffer
}
```

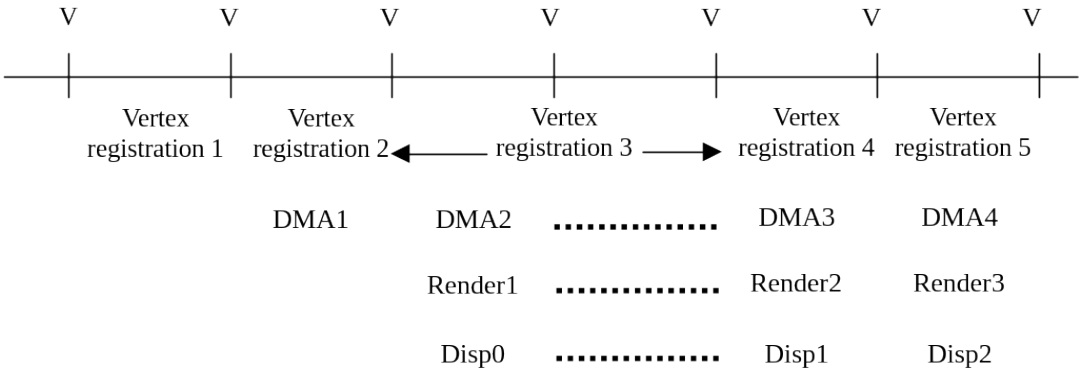
Vertex data registration ends with `kmRender`. `KmFlipFrameBuffer` directs to wait for the next Vsync.

`kmChangeLatencyMode` must precede `CreateVertexBuffer`.

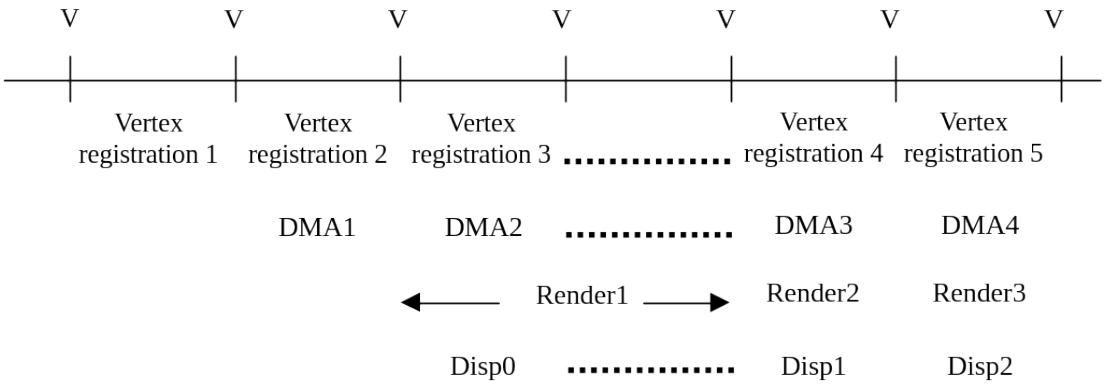


2.4.3 Processing Overflow

If a pipeline process does not fit within one Int (that is, if processing overflows), pipelining is continued by allowing the process to be prolonged.
If rendering does not end within one Int, it is possible to reset the renderer to start rendering another scene (except for COSMOS+ARC1). If rendering is discontinued forcibly during a scene, the scene cannot be completed, resulting in an invalid scene.
Shown below is an example in which a process is prolonged to perform wait control.



In this example, a process called vertex registration 3 is prolonged, thus deferring DMA 3, render 2, and display 1.

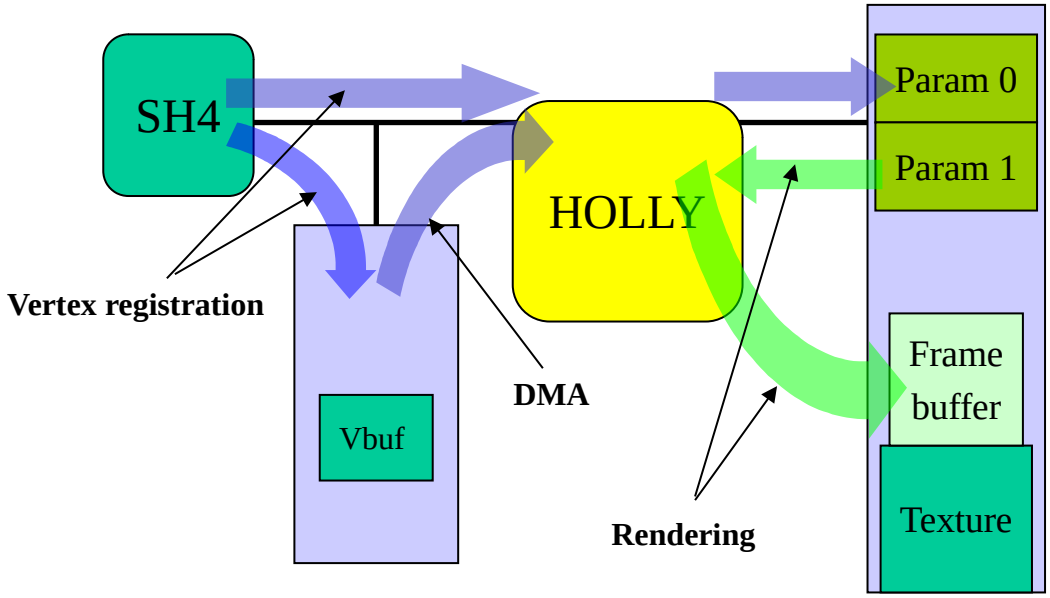


In the example shown above, render 1 takes time, causing other processes to wait.

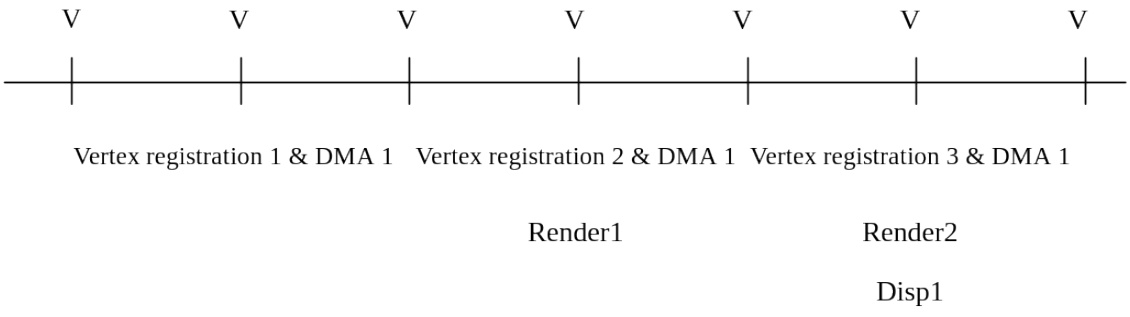
2.5 2V LATENCY MODEL

2.5.1 2V Latency Model

The 2V latency model is used to make key entry response quicker or to reduce the required vertex buffer capacity.
Unlike pipelining used in the 3V latency model, pipelining in the 2V latency model compresses vertex data registration and DMA processing within one V period, as shown below.



Vertex data registration in the 2V latency programming model can be performed using the three methods explained below.
The first method supplies a list of specific vertex types directly to the TA. Assuming that the data type most frequently used throughout a scene is the opaque polygon, the opaque data is sent directly to the TA. Data registered in other lists (opaque modifier, translucent, translucent modifier) is temporarily stored in memory. When kmRender is executed, data in these lists is transferred by DMA. If a DMA transfer does not end within a frame where it begins, a prolonged process occurs.



2.5.2 Usable Functions and How to Use Them

The following functions can be used with the 2V latency model. The function group to use is determined according to the vertex data registration method being used. As stated in the previous section, either of two methods can be used to register vertex data.

a) Vertex data registration that uses the same functions as the 3V latency model

- `kmProcessVertexRenderState`
- `kmSetVertexRenderState`
- `kmStartVertexStrip`
- `kmSetVertex`
- `kmRender`
- `kmFlipFrameBuffer`

The 2V latency model uses the above functions (the same functions as those of the 3V latency model). For `kmCreateVertexBuffer`, however, it is necessary to specify a buffer from which the data is to be sent directly to the TA.

Example:

```
 kmChangeLatencyMode(2V_LATENCY, OPAQUE_LIST);
 /* Specifies that the 2V latency is to transfer the OPAQUE */
 /* list directly to the TA. */
…… (Frame buffer creation)
 /* List size is set to 0, because the Opaque data is sent */
 /* directly. */
 kmCreateVertexBuffer(0, 0x60, 0x2000, 0x60);
    kmProcessVertexRenderState
 while(1){
       kmSetVertexRenderState
       kmStartVertexStrip
       kmSetVertex
       kmSetVertex
       …….
       kmRender
                                                  kmFlipFrameBuffer
 }
```

b) Using `kmFlushVertexBuffer`

This method flushes a vertex data buffer for a specific list type. Only one list type buffer can be flushed for one frame. Flushing the buffer triggers the DMA transfer of the accumulated data of the list type for which `kmFlushVertexBuffer` has been issued, to the TA. **Note that once a list of opaque vertex data in a frame has been flushed, it is impossible to flush a list of translucent vertex data in the same frame.**

Example:

```
kmChangeLatencyMode(2V_LATENCY, OPAQUE_LIST);
 /* Specifies that the 2V latency is to transfer the OPAQUE */
 /* list directly to the TA. */
…… (Frame buffer creation)
kmCreateVertexBuffer(0x3000, 0x60, 0x2000, 0x60);
kmProcessVertexRenderState
while(1){
      kmSetVertexRenderState
      kmStartVertexStrip
      kmSetVertex
      kmSetVertex
      …….
      kmFlushVertexBuffer (KM_OPAQUE_POLYGON)

      kmSetVertex
      …….
      kmFlushVertexBuffer (KM_OPAQUE_POLYGON)
      …….
      kmRender
                                                kmFlipFrameBuffer
}
```

c) Vertex data registration using direct-mode functions

This method uses KAMUI's direct-mode vertex data registration function (such as `kmSetVertexDirect`) to transfer all lists directly to the TA under the user's control. In this case, however, the TA specification prohibits registration with an opaque or translucent list within the frame in which opaque list registration has already been performed.

- `kmCreateTABuffer` or `kmCreateNativeDataBuffer`
- `kmProcessVertexRenderState`
- `kmSetVertexRenderState`
- `kmStartVertexStripDirect`
- `kmSetVertexDirect`
- `kmSetEndOfListDirect`
- `kmRenderDirect`
- `kmFlipFrameBuffer`

Example:

```
kmChangeLatencyMode(2V_LATENCY, VERTEX_DIRECT);
/* Specifies that the 2V latency model is to use */
/* VERTEX_DIRECT. */
…… (Frame buffer creation)
kmCreateNativeDataBuffer
kmProcessVertexRenderState
while(1){
      kmSetVertexRenderState
      /* Opaque vertex definition */
      kmStartVertexStripDirect
      kmSetVertexDirect
      …….
      KmSetEndOfListDirect
      /* Translucent vertex definition */
      kmStartVertexStripDirect
      kmSetVertexDirect
      …….
      KmSetEndOfListDirect
      …….
      kmRenderDirect
                                                kmFlipFrameBuffer
}
```

3. KAMUI FUNCTIONS

3.1 NAMING RULES

The following naming rules apply to the KAMUI functions and structures.

Function	`kmXxxx`
ENUM value, #define	`KM_XXXX`
Structure and variable	`KMXXXX`
Structure pointer	`PKMXXXX`

An attempt to call a function that has not been implemented will be responded with `KMSTATUS_NOT_IMPLEMENTED`.

3.2 INITIALIZATION FUNCTIONS

This function group initializes the rendering hardware and KAMUI.

3.2.1 Initializing the Rendering Chip

```
KMSTATUS kmInitDevice (KMVIDEOMODE nVideoMode)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function initializes the rendering hardware. It outputs video signals to produce a blank screen having the default background color (black). First call this function when using KAMUI.
For the ARC1, `KM_VGA` is assumed no matter what mode is specified.

Argument:
`nVideoMode` (input)
This argument specifies a video mode by selecting one from:

`KM_NTSC` ……. NTSC bloc (North America and Japan)
`KM_PAL` ……. PAL bloc (European countries)
`KM_VGA` ……. VGA (ARC1)

Return values:

`KMSTATUS_SUCCESS`	Initialized successfully
`KMSTATUS_INVALID_VIDEOMODE`	Invalid video mode specified
`KMSTATUS_HARDWARE_NOT_PRESENTED`	The hardware is unusable.

Example:

```
kmInitDevice(KM_NTSC);
```

3.2.2 Setting the Display Mode

```
KMSTATUS kmSetDisplayMode(KMDISPLAYMODE nDisplayMode
                      KMBPPMODE nBpp,
                      KMBOOLEAN bDither,
                      KMBOOLEAN bAntiAlias)

KMSTATUS kmChangeDisplayFilterMode(KMBOOLEAN bDither,
                      KMBOOLEAN bAntiAlias)
```

IRIS+ARC1	COSMOS+ARC1	Holly
◆	◆	♥

Note The ARC1 can be used only in VGA mode.

Explanation:
This function sets the display mode of the frame buffer. `kmChangeFilterMode` is used to turn `on/off` the dithering and antialiasing filters later.

Arguments:
`nDisplayMode` (input)
This argument specifies a display mode, which can be selected from the following:

`KM_DSPMODE_VGA` …….…..	VGA (640 x 480) 60 Hz
`KM_DSPMODE_NTSCNI320x240`…	NTSC 320 x 240 60 Hz Noninterlace
`KM_DSPMODE_NTSCI320x240`…..	NTSC 320 x 240 30 Hz Interlace(*1)
`KM_DSPMODE_NTSCNI640x240`….	NTSC 640 x 240 60 Hz Noninterlace
`KM_DSPMODE_NTSCI640x240`…..	NTSC 640 x 240 30 Hz Interlace
`KM_DSPMODE_NTSCNI320x480`...	NTSC 320 x 480 60 Hz Pseudo-noninterlace(*2)
`KM_DSPMODE_NTSCI320x480`.....	NTSC 320 x 480 30 Hz Interlace(*2)
`KM_DSPMODE_NTSCNI640x480FF`	NTSC 640 x 480 60 Hz Noninterlace FF(*3)
`KM_DSPMODE_NTSCNI640x480`…	NTSC 640 x 480 60 Hz Pseudo-noninterlace(*2)
`KM_DSPMODE_NTSCI640x480`……	NTCS 640 x 480 30 Hz Interlace(*2)
`KM_DSPMODE_PALNI320x240`…..	PAL 320 x 240 50 Hz Noninterlace
`KM_DSPMODE_PALI320x240`……..	PAL 320 x 240 25 Hz Interlace(*1)
`KM_DSPMODE_PALNI640x240`…..	PAL 640 x 240 50 Hz Noninterlace
`KM_DSPMODE_PALI640x240`…….	PAL 640 x 240 25 Hz Interlace
`KM_DSPMODE_PALNI320x480`…….	PAL 320 x 480 50 Hz Pseudo-noninterlace(*2)
`KM_DSPMODE_PALI320x480`…..	PAL 320 x 480 25 Hz Interlace(*2)
`KM_DSPMODE_PALNI640x480FF`	PAL 640 x 480 50 Hz Noninterlace FF(*3)
`KM_DSPMODE_PALNI640x480`…..	PAL 640 x 480 50 Hz Pseudo-noninterlace(*2)
`KM_DSPMODE_PALI640x480`…….	PAL 640 x 480 25 Hz Interlace(*2)

*1 In interlace mode, the same picture is drawn in both odd and even frames.
*2 This is a high-resolution mode, in which different pictures are drawn in odd and even fields.
*3 This is a noninterlace flicker free mode, in which vertical filtering is enabled to perform rendering for odd and even frames separately. This reduces flicker. In this case, rendering for each frame must be completed within 16.66 ms.

nBpp (input)

This argument specifies a frame buffer color mode, using a predefined constant listed below.

KM_DSPBPP_RGB565 …….	RGB565 format
KM_DSPBPP_RGB555 …….	RGB555 format
KM_DSPBPP_ARGB1555 …..	ARGB1555 format
KM_DSPBPP_RGB888 …….	RGB888 format
KM_DSPBPP_ARGB8888 ……	ARGB8888 format

bDither (input)

This argument determines whether to enable dithering for rendering results written by PowerVR to a 16-bit frame buffer. If the destination frame buffer is RGB888 or ARGB8888, this flag is ignored.

TRUE …. Dithering is used.

FALSE … Dithering is not used.

bAntiAlias (input)

This argument determines whether to use an antialiasing filter. Use of the antialiasing filter may reduce the operation speed.

TRUE …. The antialiasing filter is used.

FALSE ... No antialiasing filter is used.

Return values:

KMSTATUS_SUCCESS	Set successfully
KMSTATUS_INVALID_DISPLAY_MODE	Invalid display mode A display mode that does not match that specified during initialization was specified.

3.2.3 Specifying a System Configuration

```
KMSTATUS kmSetSystemConfiguration(
              PKMSYSTEMCONFIGSTRUCT pSystemConfigStruct)
```

IRIS+ARC1	COSMOS+ARC1	Holly
√	√	♥

Explanation:
This function sets the KAMUI system configuration according to the parameters specified in the `KMSYSTEMCONFIGSTRUCT` type structure. A native data buffer (double buffer) and display frame buffer are allocated in frame buffer memory. The capacity of the native data buffer area is obtained as follows:

Native data buffer capacity = whole frame buffer memory capacity - (size of the specified maximum texture + display frame buffer capacity)

Caution A texture surface shall be allocated after this function is executed.

This function substitutes for the following functions of KAMUI version 1.28 or earlier.

```
kmCreateFrameBufferSurface
kmCreateVertexBuffer
kmCreateTABuffer
kmActivateFrameBuffer
```

If `kmSetSystemConfiguration` is used, do not call any of the above functions.

Arguments:
`pSystemConfigStruct` (input)
This argument is a pointer to the `KMSYSTEMCONFIGSTRUCT` type structure, which is defined as follows:

[KMSYSTEMCONFIGSTRUCT]

```
typedef struct   _tagKMSYSTEMCONFIGSTRUCT {
        /* Size of KMSYSTEMCONFIGSTRUCT */
        KMDWORD           dwSize;
        /* system configuration flags */
        KMDWORD           flags;
        /* for Frame buffer */
        PPKMSURFACEDESC   ppSurfaceDescArray;       /* Array of SurfaceDesc */
        KMINT32           nNumOfFrameBuffer;        /* Number Of Frame Buffer */
        KMINT32           nWidth;                   /* Width of Frame buffer */
        KMINT32           nHeight;                  /* Height of Frame buffer */
        KMBPPMODE         nBpp;                     /* Bpp for Frame buffer */
        /* for Texture Memory */
        KMINT32           nTextureMemorySize;       /* Texture Memory size */
        /* for Vertex buffer */
        PKMVERTEXBUFFDESC pBufferDesc;     /* pointer to VERTEXBUFFDESC */
        PKMDWORD          pVertexBuffer;    /* pointer to Vertex buffer */
        KMINT32           nVertexBufferSize;        /* size of Vertex buffer */
        KMFLOAT           fBufferSize[5];           /* buffer size in percent */
        KMVERTEXBUFMODEL  VbufModel;                /* Vertex Buffer model */
        /* reserve area */
        KMDWORD           reserved01;               /* reserved for future use */
        KMDWORD           reserved02;               /* reserved for future use */
        KMDWORD           reserved03;               /* reserved for future use*/
        KMDWORD           reserved04;               /* reserved for future use*/
        KMDWORD           reserved05;               /* reserved for future use*/
        KMDWORD           reserved06;               /* reserved for future use*/
        KMDWORD           reserved07;               /* reserved for future use*/
} KMSYSTEMCONFIGSTRUCT, *PKMSYSTEMCONFIGSTRUCT;
```

The setting of each member is explained below:

[System configuration specification flag]

`dwSize` (input)

This argument sets the size of the `KMSYSTEMCONFIGSTRUCT` structure in `Sizeof(KMSYSTEMCONFIGSTRUCT)`.

`flags` (input)

This argument specifies the types of data related to the system configuration. It is the result of ORing the following flags. If no flag is to be specified, the argument shall be reset to zero.

`KM_CONFIGFLAG_ENABLE_CLEAR_FRAMEBUFFER`

This argument causes a frame buffer to be cleared when it is allocated.

`KM_CONFIGFLAG_ENABLE_STRIPBUFFER`

This argument causes a frame buffer to be created in `StripBuffer` format. The `nWidth` and `nHeight` members are enabled. (`ARC1` does not have a `strip buffer` function. If this argument is specified for `ARC1`, it may not operate normally.)

`KM_CONFIGFLAG_ENABLE_2V_LATENCY`

This argument causes KAMUI to operate in the 2V latency mode. If this argument is not entered, KAMUI operates in the 3V latency mode.

`KM_CONFIGFLAG_NOWAITVSYNC`

This argument causes a frame buffer surface to be displayed before a `V-sync` interrupt occurs. If this argument is not entered, the frame buffer surface is displayed after a `V-sync` interrupt has occurred.

`KM_CONFIGFLAG_USEDIRECTMODE`

This argument specifies that the direct mode is to be used. The members related to the vertex buffer (`pBufferDesc`, `pVertexBuffer`, `nVertexBufferSize`, and `nBufferSize[5]`) are ignored. If this argument is not entered, the buffer mode is used.

`KM_CONFIGFLAG_NOWAIT_FINISH_TEXTUREDMA`

This argument causes the texture load function to be terminated before DMA transfer of a texture to frame buffer memory, started by the function, ends. If this argument is entered, the `kmQueryFinishLastTextureDMA` function can be used to check whether DMA transfer has been completed. Avoid accessing memory from which a DMA transfer is under way.

[Parameters related to the frame buffer]

`ppSurfaceDescArray` (output)
This argument specifies an array of pointers to the `KMSURFACEDESC` type structure for each frame buffer (front and back). If `KMSTATUS_NOT_ENOUGH_MEMORY` is returned, the contents of this frame buffer structure will be undefined. The application program must prepare an area for each of the two frame buffer structures and an array of pointers to the structures.

`nNumOfFrameBuffer` (input)
This argument specifies the number of frame buffer surfaces to be generated. Currently, "2" (front and back) should be specified.

`ppSurfaceDescArray` should be specified using `nNumOfFrameBuffer`, as follows:

```
KMSURFACEDESC Surface1;
KMSURFACEDESC Surface2;
PKMSURFACEDESC ppSurfaceArray[nNumOfFrameBuffer];
ppSurfaceArray[0] = &Surface1;
ppSurfaceArray[1] = &Surface2;
ppSurfaceDescArray = ppSurfaceArray;
```

`nWidth` and `nHeight` (input)
These arguments specify the horizontal and vertical sizes of the frame buffer surface. If the frame buffer is in `StripBuffer` format, the sizes must be an integer multiple of 32. (`ARC1` has no `strip buffer` function.)

`nBpp` (input)
This argument specifies the color mode for a frame buffer to be reconfigured, using the following predefined constants:

`KM_DSPBPP_RGB565` …….	RGB565 format
`KM_DSPBPP_RGB555` …….	RGB555 format
`KM_DSPBPP_ARGB1555` …..	ARGB1555 format
`KM_DSPBPP_RGB888` …….	RGB888 format
`KM_DSPBPP_ARGB8888` ……	ARGB8888 format

[Parameters related to the texture area]

`nTextureMemorySize` (input)
This argument specifies the maximum required texture size. It is used to determine the native data buffer area capacity. It must represent the maximum capacity of textures to be used simultaneously, in `DWORD` units (doublewords, or the number of bytes divided by 4). This size must be a multiple of 32 bytes.

Example Specify `1048576 (0x100000)` if up to 4 Mbytes of textures are to be used simultaneously.

[Parameters related to the vertex buffer]

`pBufferDesc` (input)
This argument inputs a pointer to a vertex data buffer descriptor of `KMVERTEXBUFFDESC` type. The application program must prepare an area for this structure. It is referenced by the `kmStartVertexStrip` and `kmSetVertex` macros. See Section 3.6 for the `KMVERTEXBUFFDESC` type structure.

`pVertexBuffer` (input)
This argument sets a pointer to the vertex buffer allocated by the application program in system memory. KAMUI uses this pointer as the base address of the vertex buffer. (To avoid `malloc` in KAMUI, the application program must prepare the vertex buffer.) This address must be on a 32-byte boundary.

`nVertexBufferSize` (input)
This argument specifies the size of the vertex data buffer allocated in system memory by the application program, in `DWORD` units (the number of bytes divided by 4). This size must be a multiple of 32 bytes. See "How to obtain the size (`nVertexBufferSize`) of a vertex data buffer (VertexBuffer)" for an explanation of how to determine the size.

`nBufferSize[5]` (input)
This argument specifies, as a percentage the number of polygons used in one scene for each of five list types.
These five list types are specified with a floating point value of between `0.0f` and `100.0f`. The total of the specified values must be `100.0f`. If it exceeds `100.0f`, KAMUI may not operate normally. KAMUI uses these values to assign a vertex data buffer to each polygon type.

`nBufferSize[0]`: Specifies, as a percentage, the number of opaque polygons to be used.
`nBufferSize[1]`: Specifies, as a percentage, the number of opaque modifier volumes to be used.
`nBufferSize[2]`: Specifies, as a percentage, the number of translucent/transparent polygons to be used.
`nBufferSize[3]`: Specifies, as a percentage, the number of translucent/transparent modifier volumes to be used.
`nBufferSize[4]`: Specifies, as a percentage, the number of punchthrough polygons to be used.

(`nBufferSize[4]` can be specified only for `CLX2`. It must always be 0 for any version other than `CLX2`. Otherwise, KAMUI may not operate normally.)

`VbufModel` (input)

This argument specifies the way the `VertexBuffer` is to be used by selecting one of the following:

`KM_VERTEXBUFMODEL_NORMAL`
: This value must always be specified for the `3V latency` mode.

`KM_VERTEXBUFMODEL_FLUSH_OPAQUE`
: This value must be specified to `flush` a list of `opaque` polygons in the `2V latency` mode.

`KM_VERTEXBUFMODEL_FLUSH_OPAQUE_MODIFIER`
: This value must be specified to `flush` a list of `opaque modifiers` in the `2V latency` mode.

`KM_VERTEXBUFMODEL_FLUSH_OPAQUE_TRANS`
: This value must be specified to `flush` a list of `translucent` polygons in the `2V latency` mode.

`KM_VERTEXBUFMODEL_FLUSH_OPAQUE_TRANS_MODIFIER`
: This value must be specified to `flush` a list of `translucent modifiers` in the `2V latency` mode.

`KM_VERTEXBUFMODEL_FLUSH_PUNCH_THROUGH`
: This value must be specified to `flush` a list of `punchthrough` polygons in the `2V latency` mode.

`KM_VERTEXBUFMODEL_NOBUF_OPAQUE`
: This value must be specified to send a list of `opaque` polygons directly to the `TA` in the `2V latency` mode.

`KM_VERTEXBUFMODEL_NOBUF_OPAQUE_MODIFIER`
: This value must be specified to send a list of `opaque modifiers` directly to the `TA` in the `2V latency` mode.

`KM_VERTEXBUFMODEL_NOBUF_TRANS`
: This value must be specified to send a list of `translucent` polygons directly to the `TA` in the `2V latency` mode.

`KM_VERTEXBUFMODEL_NOBUF_TRANS_MODIFIER`
: This value must be specified to send a list of `translucent modifiers` directly to the `TA` in the `2V latency` mode.

`KM_VERTEXBUFMODEL_NOBUF_PUNCH_THROUGH`
: This value must be specified to send a list of `punchthrough` polygons directly to the `TA` in the `2V latency` mode.

`reserved01` to `reserved07`

These are reserved for future expansion. Their contents are undefined.

Return values:

`KMSTATUS_SUCCESS`	System configuration set up successfully.
`KMSTATUS_NOT_ENOUGH_MEMORY`	Insufficient memory capacity for native data and frame buffers.

How to obtain the size (`nVertexBufferSize`) of a vertex data buffer (VertexBuffer)

The vertex data buffer size is determined by summing the sizes of lists of the five data types (`OpaquePolygon`, `OpaqueModifier`, `TransPolygon`, `TransModifier`, and `PunchthroughPolygon`). The actually required size is double the size obtained this way, because KAMUI uses double buffers, that is:

VertexBufferSize = (OpaquePolygon ListSize
+ OpaqueModifier ListSize
+ TransPolygon ListSize
+ TransModifier ListSize
+ PunchThroughPolygon ListSize
) × 2

The size of each list consists of the following four elements. So, the size of each list is obtained (in doubleword units) as follows:

ListSize = (VertexParameterSize **<1>**
+ GlobalParameterSize **<2>**
+ ControlParameterSize **<3>**
+ EndOfListSize **<4>**
)

<1> VertexParameterSize

VertexParameterSize = maximum number of vertices used in one scene
× size of vertex data to be used (VertexParameter)

The size of vertex data depends on a vertex type to be used. When multiple vertex types are used in one scene, calculate the "maximum number of vertices x size of vertex data" for each vertex type and add them.

The following table shows the relation between vertex types and their data sizes.

Type	Size (DWORD)	Type	Size (DWORD)	Type	Size (DWORD)
Vertex0	8	Vertex6	16	Vertex12	16
Vertex1	8	Vertex7	8	Vertex13	16
Vertex2	8	Vertex8	8	Vertex14	16
Vertex3	8	Vertex9	8	Vertex15	16
Vertex4	8	Vertex10	8	Vertex16	16
Vertex5	16	Vertex11	16	Vertex17	16

<2> GlobalParameterSize

KAMUI adds data which indicates the beginning of each vertex data strip to the strip. This data is called a global parameter (GlobalParameter). The size of the global parameter is obtained as follows:

GlobalParameterSize = maximum number of kmStartVertexStrip used in one scene
× global parameter size

The global parameter size is usually 8 (doublewords). In the following five cases, however, the global parameter size is 16 (doublewords).

- The vertex type is 7, ColorType is Intensity, and the offset color is used.
- The vertex type is 8, ColorType is Intensity, and the offset color is used.
- The vertex type is 10 and ColorType is Intensity.
- The vertex type is 13 and ColorType is Intensity.
- The vertex type is 14 and ColorType is Intensity.

If more than one global parameter size is used, it is necessary to sum all the sizes.

<3> ControlParameterSize

KAMUI specifies UserClipping for kmSetUserClipping by writing it, in control parameter (ControlParameter) format, into a vertex data buffer in much the same way as vertex data. The size of the control parameter is obtained as follows:

ControlParameterSize =
maximum number of kmSetUserClipping used in one scene
× control parameter size

The control parameter size is always 8 (doublewords).

<4> EndOfListSize

KAMUI writes data indicating the end of each list. This data, EndOfListSize, is always 8 (doublewords). The actual EndOfListSize must be set to 16 (or 48 for ARC1) doublewords, because internal data uses an additional 8 (or 40 for ARC1) doublewords.

If a list (`OpaquePolygon`, `OpaqueModifier`, `TransPolygon`, `TransModifier`, or `PunchthroughPolygon`) is not used, its size is assumed to be zero. It is unnecessary to add the EndOfListSize.

Note that in the `2V latency` mode, `VertexParameter` is held in a single buffer, and no buffer is required for `OpaquePolygon`.

3.2.4 Switching the Latency Mode

```
KMSTATUS kmChangeLatencyModel(KMLATENCYMODEL LatencyMode,
                              KMVERTEXBUFMODEL VbufModel)
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	♥

Explanation:
This function changes the current latency model.

Arguments:
`LatencyMode` (input)
This argument specifies a latency model by selecting it from the following:

`KM_LATENCYMODEL_3V`	Specifies the 3V latency model.
`KM_LATENCYMODEL_2V`	Specifies the 2V latency model.

`VbufModel` (input)
This argument specifies the way in which the vertex buffer is to be used by selecting one of the following:

`KM_VERTEXBUFMODEL_NORMAL`
: This value must always be specified for the 3V latency mode.

`KM_VERTEXBUFMODEL_FLUSH_OPAQUE`
: This value must be specified to flush a list of opaque polygons in the 2V latency mode.

`KM_VERTEXBUFMODEL_FLUSH_OPAQUE_MODIFIER`
: This value must be specified to flush a list of opaque modifiers in the 2V latency mode.

`KM_VERTEXBUFMODEL_FLUSH_TRANS`
: This value must be specified to flush a list of translucent polygons in the 2V latency mode.

`KM_VERTEXBUFMODEL_FLUSH_TRANS_MODIFIER`
: This value must be specified to flush a list of translucent modifiers in the 2V latency mode.

`KM_VERTEXBUFMODEL_FLUSH_PUNCH_THROUGH`
: This value must be specified to flush a list of punchthrough polygons in the 2V latency mode.

`KM_VERTEXBUFMODEL_NOBUF_OPAQUE`
: This value must be specified to send a list of opaque polygons directly to the TA in the 2V latency mode.

`KM_VERTEXBUFMODEL_NOBUF_OPAQUE_MODIFIER`
: This value must be specified to send a list of opaque modifiers directly to the TA in the 2V latency mode.

KM_VERTEXBUFMODEL_NOBUF_TRANS

This value must be specified to send a list of translucent polygons directly to the TA in the 2V latency mode.

KM_VERTEXBUFMODEL_NOBUF_TRANS_MODIFIER

This value must be specified to send a list of translucent modifiers directly to the TA in the 2V latency mode.

KM_VERTEXBUFMODEL_NOBUF_PUNCH_THROUGH

This value must be specified to send a list of punchthrough polygons directly to the TA in the 2V latency mode.

Caution `kmChangeLatencyMode` must precede `kmCreateVertexBuffer`.

Return value:

KMSTATUS_SUCCESS	Function executed successfully.

3.3 SURFACE HANDLING FUNCTIONS

These functions generate, erase, or flip the frame buffer surface and texture surface. They also manage the states of the surfaces.

3.3.1 Creating the Primary Surface and Off-Screen Surface (ARC1)

```
KMSTATUS kmCreateFrameBufferSurface (
                            PKMSURFACEDESC pSurfaceDesc,
                            KMINT32 nWidth,
                            KMINT32 nHeight,
                            KMBOOLEAN bStripBuffer,
                            KMBOOLEAN bBufferClear)
```

IRIS+ARC1	COSMOS+ARC1	Holly
√	√	◆

Explanation:
This function allocates a frame buffer surface in frame buffer memory and initializes it. Before this function is called, `kmSetDisplayMode` must be executed.

Holly allocates an actual frame buffer and stores information into `pSurfaceDesc` asynchronously with this function, because this is requested by the frame buffer memory management. So, **`kmSetSystemConfiguration` must be used for Holly**.

Arguments:

`pSurfaceDesc` (output)
This argument is a pointer to a surface information structure. Surface information is returned using the pointer. It becomes undefined if `KMSTATUS` is responded with `KMSTATUS_NOT_ENOUGH_MEMORY`.

`nWidth` and `nHeight` (input)
These arguments specify the horizontal and vertical sizes of the frame buffer surface, if the frame buffer to be used is of StripBuffer format. The sizes must be a multiple of 32. If the frame buffer is not in StripBuffer format, this parameter is ignored, and the screen size is determined by `nDisplayMode` specified in `kmSetDisplayMode`.
The ARC1 has no strip buffer function. So, both arguments must always be 0 for the ARC1.

`bStripBuffer` (input)
If this argument is TRUE, a frame buffer in StripBuffer format is created. No strip buffer function is available to the ARC1. So, this argument must always be FALSE for the ARC1.

`bBufferClear` (input)

`TRUE` … When a surface is created, it is cleared to 0.

`FALSE` .. When a surface is created, it is not cleared to 0.

Caution For Holly, this function cannot be used together with `kmSetSystemConfiguration`. If this function is issued after `kmSetSystemConfiguration`, Holly may not operate normally.

Return values:

`KMSTATUS_SUCCESS`	Function successful
`KMSTATUS_NOT_ENOUGH_MEMORY`	Failed because of insufficient memory

Examples:

```
KMSTATUS status;
SURFACEDESC SurfDesc;
status = kmCreateFrameBufferSurface(&SurfDesc, 0, 0, FALSE, TRUE);
```

3.3.2 Creating the Texture Surface

```
KMSTATUS kmCreateTextureSurface(
                    PKMSURFACEDESC pSurfaceDesc,
                    KMINT32 nWidth,
                    KMINT32 nHeight,
                    KMTEXTURETYPE nTextureType)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function secures a texture surface in texture memory. This API can allocate texture surfaces in all formats.

Whenever possible, KAMUI aligns the texture address with a 2 KB boundary so as to quicken memory access. Therefore, no texture surface may be created even if the total of the free frame buffer capacities is larger than the capacity to be allocated to the texture. In this case, it is necessary to perform garbage collection, using `kmGarbageCollectTexture`.

ARC1 interleaves a VQ/Twiddled mipmap format texture. Therefore, a texture surface may not be created even if the total amount of free space in the frame buffer exceeds the size of the texture to be secured.
ARC1 searches for the smallest texture pair that is larger than the specified size if the texture surface of a VQ/Twiddled mipmap is specified, and creates a texture surface in a single bank of that pair if it is vacant.

KAMUI aligns the first texture surface address and size with a 32-byte boundary.

Caution This function must be executed after `kmCreateFrameBufferSurface`, `kmCreateVertexBuffer`, and `kmSetSystemConfiguration` are called.

Efficient use of texture memory

(Common to ARC1 and CLX1/2)
- Secure/release as many texture surfaces as possible at the same time.
- Call the creation of a frame buffer area or native data area (`kmSetSystemConfiguration`, `kmCreateFrameBufferSurface`, `kmCreateVertexBuffer`, `kmCreateTABuffer`) before the creation of a texture surface, and avoid releasing and re-creating these until AP ends.

(ARC1-specific)
- Execute `kmCreateVertexBuffer` before `kmCreateFrameBufferSurface`.
- Allocate mipmap and VQ texture surface first.
- Avoid alternately allocating texture surfaces of different sizes. Whenever possible, simultaneously allocate surfaces of the same size and format.

Arguments:

`pSurfaceDesc` (output)

This argument is a pointer to a surface information structure. Surface information is returned to the structure using the pointer. It becomes undefined if `KMSTATUS` is responded with `KMSTATUS_NOT_ENOUGH_MEMORY`.

`nWidth` and `nHeight` (input)

These arguments specify the horizontal and vertical texture sizes. If MIPMAP is used, the top-level texture size must be specified. The value specified for `nWidth` or `nHeight` must be 8, 16, 32, 64, 128, 256, 512, or 1,024.

`nTextureType` (input)

This argument specifies a texture format. The texture format is specified by ORing a category code and pixel format code selected from those listed below.

- Category codes

```
KM_TEXTURE_TWIDDLED       // Twiddled format
KM_TEXTURE_TWIDDLED_RECTANGLE
                          // Rectangular Twiddled format
                            (Cannot be used with ARC1.)
KM_TEXTURE_TWIDDLED_MM    // Twiddled format with a mipmap
KM_TEXTURE_VQ             // VQ compression format
KM_TEXTURE_VQ_MM          // VQ compression format with a mipmap
KM_TEXTURE_SMALLVQ        // Small VQ compression format
KM_TEXTURE_SMALLVQ_MM     // Small VQ compression format with a
                            mipmap
KM_TEXTURE_PALETTIZE4     // 4-bpp palette format
                            (Cannot be used with ARC1.)
KM_TEXTURE_PALETTIZE4_MM  // 4-bpp palette format with a mipmap
                            (Cannot be used with ARC1.)
KM_TEXTURE_PALETTIZE8     // 8-bpp palette format
                            (Cannot be used with ARC1.)
KM_TEXTURE_PALETTIZE8_MM  // 8-bpp palette format with a mipmap
                            (Cannot be used with ARC1.)
KM_TEXTURE_RECTANGLE      // Rectangle
KM_TEXTURE_STRIDE         // Rectangle (stride specification)
```

- Pixel format codes

```
KM_TEXTURE_ARGB1555
KM_TEXTURE_RGB565
KM_TEXTURE_ARGB4444
KM_TEXTURE_YUV422          (Cannot be used with ARC1.)
KM_TEXTURE_BUMP            (Cannot be used with ARC1.)
```

Note With the palette format texture (`KM_TEXTURE_PALETTIZED4`, `KM_TEXTURE_PALETTIZED4_MM`, `KM_TEXTURE_PALETTIZED8`, `KM_TEXTURE_PALETTIZED8_MM`), the pixel format cannot be specified. Specify the pixel format of the palette format texture by setting a palette (`kmSetPaletteMode`).

Return values:

`KMSTATUS_SUCCESS`	Secured successfully
`KMSTATUS_INVALID_TEXTURE_TYPE`	Invalid texture type specified
`KMSTATUS_NOT_ENOUGH_MEMORY`	Insufficient memory

3.3.3 Creating the Texture Surface for Mipmap/VQ Texture

```
KMSTATUS kmCreateCombinedTextureSurface(
                    PKMSURFACEDESC pSurfaceDesc1,
                    PKMSURFACEDESC pSurfaceDesc2,
                    KMINT32 nWidth,
                    KMINT32 nHeight,
                    KMTEXTURETYPE nTextureType)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function secures a texture surface in texture memory. It secures two texture surfaces of the same size and format.

This API explicitly secures two surfaces interleaved by the ARC1 (Twiddled-mipmap or VQ (mipmap)) in pairs. Like `kmCreateTextureSurface`, however, this API can allocate texture surfaces in all formats.

KAMUI aligns the first texture surface address and size with a 32-byte boundary.

Caution This function must be executed after `kmCreateFrameBufferSurface`, `kmCreateVertexBuffer`, and `kmSetSystemConfiguration` are called.

Arguments:
`pSurfaceDesc1` (output)
This argument is a pointer (No. 1) to surface structure information. Surface information is returned to the structure using the pointer. It becomes undefined if, for `KMSTATUS`, `KMSTATUS_NOT_ENOUGH_MEMORY` is returned.

`pSurfaceDesc2` (output)
This argument is a pointer (No. 2) to surface structure information. Surface information is returned to the structure using the pointer. It becomes undefined if, for `KMSTATUS`, `KMSTATUS_NOT_ENOUGH_MEMORY` is returned.

`nWidth` and `nHeight` (input)
These arguments specify the horizontal and vertical texture sizes. If MIPMAP is used, the top-level texture size must be specified. The value specified for `nWidth` or `nHeight` must be 8, 16, 32, 64, 128, 256, 512, or 1,024.

`nTextureType` (input)
This argument specifies a texture format. See `kmCreateTextureSurface`.

Return values:

`KMSTATUS_SUCCESS`	Secured successfully
`KMSTATUS_INVALID_TEXTURE_TYPE`	Invalid texture type specified
`KMSTATUS_NOT_ENOUGH_MEMORY`	Insufficient memory

3.3.4 Creating the Texture Surface in Contiguous Address Areas

```
KMSTATUS kmCreateContiguousTextureSurface(
                    PPKMSURFACEDESC     ppSurfaceDesc,
                    KMINT32             nTexture,
                    KMINT32             nWidth,
                    KMINT32             nHeight,
                    KMTEXTURETYPE       nTextureType)
```

IRIS+ARC1	COSMOS+ARC1	Holly
◆	◆	♥

Explanation:
This function simultaneously secures two or more texture surfaces at contiguous addresses in the frame buffer. It is used to read two or more textures of YUV422 type in succession, by using the YUV converter of tiling accelerator of the CLX1/2 (see `kmLoadYUVTexture`).

This API, however, can also allocate texture surfaces in all formats, excluding "small VQ compression format" and "small VQ compression format with a mipmap."
However, the ARC1 cannot secure surfaces interleaved by this function (VQ, VQ-mipmap, Twiddled-mipmap).

Caution This function must be executed after `kmSetSystemConfiguration`, `kmCreateFrameBufferSurface`, and `kmCreateVertexBuffer` are called.

Arguments:
`ppSurfaceDesc` (output)
This argument is a pointer to a `KMSURFACEDESC` structure. Texture surface information is returned to the structure. It becomes undefined if, for `KMSTATUS`, `KMSTATUS_NOT_ENOUGH_MEMORY` is returned.

`nTexture` (input)
This argument specifies the number of texture surfaces to be secured in succession.

`nWidth` and `nHeight` (input)
These arguments specify the horizontal size and vertical size of the texture. The value specified for `nWidth` or `nHeight` must be 8, 16, 32, 64, 128, 256, 512, or 1,024.

`nTextureType` (input)
This argument specifies a texture format. The texture format is specified by ORing a category code and pixel format code selected from those listed below.

- Category codes

```
KM_TEXTURE_TWIDDLED       // Twiddled format
KM_TEXTURE_TWIDDLED_MM    // Twiddled format with a mipmap
                             (Cannot be specified with ARC1)
KM_TEXTURE_TWIDDLED_RECTANGLE
                          // Twiddled-Rectangle
                             (Cannot be used with ARC1)
KM_TEXTURE_VQ             // VQ compression format
                             (Cannot be specified with ARC1)
KM_TEXTURE_VQ_MM          // VQ compression format with a mipmap
                             (Cannot be specified with ARC1)
KM_TEXTURE_PALETTIZE4     // 4-bpp palette format
                             (Cannot be used with ARC1)
KM_TEXTURE_PALETTIZE4_MM  // 4-bpp palette format with a mipmap
                             (Cannot be used with ARC1)
KM_TEXTURE_PALETTIZE8     // 8-bpp palette format
                             (Cannot be used with ARC1)
KM_TEXTURE_PALETTIZE8_MM  // 8-bpp palette format with a mipmap
                             (Cannot be used with ARC1)
KM_TEXTURE_RECTANGLE      // Rectangle
KM_TEXTURE_STRIDE         // Rectangle
                             (with stride specification)
```

- Pixel format codes

```
KM_TEXTURE_ARGB1555
KM_TEXTURE_RGB565
KM_TEXTURE_ARGB4444
KM_TEXTURE_YUV422          // (Cannot be used with ARC1)
KM_TEXTURE_BUMP            // (Cannot be used with ARC1)
```

Return values:

`KMSTATUS_SUCCESS`	Texture memory secured successfully
`KMSTATUS_INVALID_TEXTURE_TYPE`	Invalid texture type specified
`KMSTATUS_NOT_ENOUGH_MEMORY`	Insufficient memory

3.3.5 Using Frame Buffer as Texture Surface

This API has been deleted, because the CLX hardware specification does not allow it to be implemented. (Ver 1.30)

3.3.6 Setting the Alpha Threshold Value

```
KMSTATUS kmSetAlphaThreshold(KMINT32 nThreshold)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function sets a threshold value for determining an α bit to be used when ARGB1555 is specified as the bit depth of a rendering frame buffer surface. If the rendering result is greater than or equal to the specified threshold value, the α bit is set to 1. This bit is used for chroma-key composition by RAMDAC.

Argument:
`nThreshold` (input)
This argument specifies a threshold value from 0 to 255. If a value less than 0 or greater than 255 is specified, 0 or 255 is assumed, respectively.

Return value:
`KMSTATUS_SUCCESS` Set successfully

3.3.7 Determining the Display Screen

```
KMSTATUS kmActivateFrameBuffer(
                    PKMSURFACEDESC pPrimarySurfaceDesc,
                    PKMSURFACEDESC pBackBufferSufaceDesc,
                    KMBOOLEAN bStripBuffer,
                    KMBOOLEAN bWaitVSync)
```

IRIS+ARC1	COSMOS+ARC1	Holly
√	√	◆

Explanation:
This function notifies KAMUI of the frame buffer surface to be used for display purposes.

Arguments:
`pPrimarySurfaceDesc` (input)
This argument is the surface structure of a frame buffer surface to be used for displaying. The surface structure must be one obtained by securing a surface using `kmCreateFrameBufferSurface`.

`pBackBufferSurfaceDesc` (input)
This argument is the surface structure of a frame buffer surface to be submitted to rendering.

`bStripBuffer` (input)
This argument must be `TRUE` if the strip buffer is used. This function cannot be used with ARC1.

`bWaitVSync` (input)
This argument specifies whether to defer displaying a frame buffer surface till the timing of Vsync. If the argument is `TRUE`, this function defers displaying till the timing of Vsync.

Caution This function must not be issued if Holly uses `kmSetSystemConfiguration`. If this function is issued after `kmSetSystemConfiguration`, normal operation is not guaranteed.

Return value:

`KMSTATUS_SUCCESS`	Display switched successfully

3.3.8 Flipping the Display Screen

```
KMSTATUS kmFlipFrameBuffer(VOID)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
If the frame buffer is configured for double buffering[1], this function exchanges the display primary surface with the off-screen surface (rendering target). The `kmFlipFrameBuffer` command issued by the application program is enqueued within KAMUI. KAMUI flips the display at the Vsync timing after the command is issued, if rendering to an off-screen surface has been completed.
If the strip buffer is used, software-based Flip is not needed.

Argument:
None

Return values:

`KMSTATUS_SUCCESS`	Flip command issued successfully
`KMSTATUS_CANT_FLIP_SURFACE`	Failure in issuing Flip command

[1] Unnecessary if the strip buffer is used.

3.4 SETTING PARAMETERS FOR EACH SCENE SEPARATELY

This section explains the functions for setting the parameters that can be specified separately for each scene.

3.4.1 Setting the Culling Parameter

```
KMSTATUS kmSetCullingRegister(KMFLOAT fCullVal)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function specifies a threshold value for culling small polygons.

Argument:
`fCullVal` (input)
This argument sets a determinant value for a plane parameter.

Return value:
`KMSTATUS_SUCCESS` Set successfully

3.4.2 Setting the Color Clamp Value

```
KMSTATUS kmSetColorClampValue(
                          KMPACKEDARGB MaxVal,
                          KMPACKEDARGB MinVal)
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	♥

Explanation:
This function specifies the color clamp value. Color clamping is applied ahead of fogging. If you want to change the clamp color when rendering, do so within a callback function for rendering termination. If an attempt is made to change the clamp color at any other timing, a screen image may become invalid.

Arguments:
`MaxVal` (input)
This argument specifies a maximum value for color clamping. It is a packed 32-bit color. If you want to specify the RGB color with a brightness of 128, enter 0x00808080.

`MinVal` (input)
This argument specifies a minimum value for color clamping. It is a packed 32-bit color. If you want to specify the RGB color with a brightness of 20, enter 0x00141414.

Return value:
`KMSTATUS_SUCCESS` Set successfully

3.4.3 Setting the Fog Color

```
KMSTATUS kmSetFogTableColor(KMPACKEDARGB FogTableColor)
KMSTATUS kmSetFogVertexColor(KMPACKEDARGB FogVertexColor)
```

IRIS+ARC1	COSMOS+ARC1	Holly
√	√	♥

Explanation:
This function specifies a fog color. For the HOLLY, different fog colors can be specified in FogVertex and FogTable. For the ARC1, however, the most recently specified value is valid even if different fog colors are specified using FogVertex and FogTable. If you want to change the fog color when rendering, do so within a callback function for rendering termination. If an attempt is made to change the fog color at any other timing, a screen image may become invalid.

Arguments:
`FogTableColor` and `FogVertexColor` (input)
These arguments specify the packed 32-bit color to be used in FogTable and FogVertex.

Return value:
`KMSTATUS_SUCCESS` Set successfully

3.4.4 Specifying a Fog Density

```
KMSTATUS kmSetFogDensity(KMDWORD FogDensity)
```

IRIS+ARC1	COSMOS+ARC1	Holly
√	√	♥

Explanation:
This function assigns a coefficient (scale factor) to TSP fog.
FogDensity consists of two bytes. The higher byte indicates the mantissa and the lower byte indicates the exponent (the nth power of 2).

Example) 0x0100 = 0.0000001(b) = 0.015625
0x8000 = 1.0(b) = 1.0
0xFF00 = 1.1111111(b) = 1.984375
0xFF01 = 11.111111(b) = 3.96875
0xFF06 = 1111111.1(b) = 128.5
0xFF07 = 11111111.0(b) = 255
0xFF08 = 111111110.0(b) = 510
0xFF09 = 1111111100.0(b) = 1,020
0xFF0A = 2040
0xFF0B = 4080
0xFF0C = 8160

If a low value is specified for FogDensity, the effect of fog appears from the polygon with the higher 1/w (Fog density increases).
If a high value is specified for FogDensity, the effect of fog can be seen only on the polygon with a low 1/w (Fog density decreases).
For details, see the description of `kmSetFogTable`.

Argument:
`fogDensity` (input)
This argument is a coefficient of TSP fog (scale factor).
Specify this argument as `"kmSetFogDensity (0xFF09)"`.

Return value:
`KMSTATUS_SUCCESS` Set successfully

3.4.5 Setting the Fog Table

```
KMSTATUS kmSetFogTable(PKMFLOAT pfFogTable)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function registers the fog table. A pointer to an array holding 128 different float values is passed via the argument. The fog table takes effect on the polygon for which FogTable is specified by VERTEXCONTEXT of 1/w = 0.0 (infinite point) to 1.0 (depth = 1.0).
An element of a fog table that is 0.0 has an attenuation rate 0 and an element that is 1.0 has the maximum attenuation rate.
A fog table consists of 128 elements with indexes 0 to 127.
The element of index of a fog table specifies fog density of the following position with a depth of (1/w value):

Depth = (pow (2.0, Index >> 4) * (float) ((Index & 0x0F) + 16)/16.0f)/FogDensity

Therefore, specify the density starting from the most distant point, in sequence.
Specify FogDensity in this equation with `kmSetFogDensity`.

Argument:
`pfFogTable` (input)
This argument specifies a pointer to an array of fog table values. The array consists of 128 different parameters.

Return value:
`KMSTATUS_SUCCESS` Set successfully

3.4.6 Setting the On-Chip Palette Mode

```
KMSTATUS kmSetPaletteMode(KMPALETTEMODE Palettemode)
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	♥

Explanation:
This function specifies a mode of the on-chip palette used by the palettized texture. A palette has 1,024 entries. For details of how to set a palette, see the description of `kmSetPaletteData`.

Caution Palette data setting (kmSetPaletteData/kmSetPaletteBankData) cannot precede palette mode setting (kmSetPaletteMode). If the palette mode type does not match the palette data type, invalid data will be set in the palette register.

Argument:
`PaletteMode` (input)

`KM_PALETTE_16BPP_ARGB1555`	16-BPP mode, ARGB1555 format
`KM_PALETTE_16BPP_RGB565`	16-BPP mode, RGB565 format
`KM_PALETTE_16BPP_ARGB4444`	16-BPP mode, ARGB4444 format
`KM_PALETTE_32BPP_ARGB8888`	32-BPP mode, ARGB8888 format

Return value:

`KMSTATUS_SUCCESS`	Set successfully

3.4.7 Setting the On-Chip Palette Data

```
KMSTATUS kmSetPaletteData(PKMPALETTEDATA pPaletteTable)
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	♥

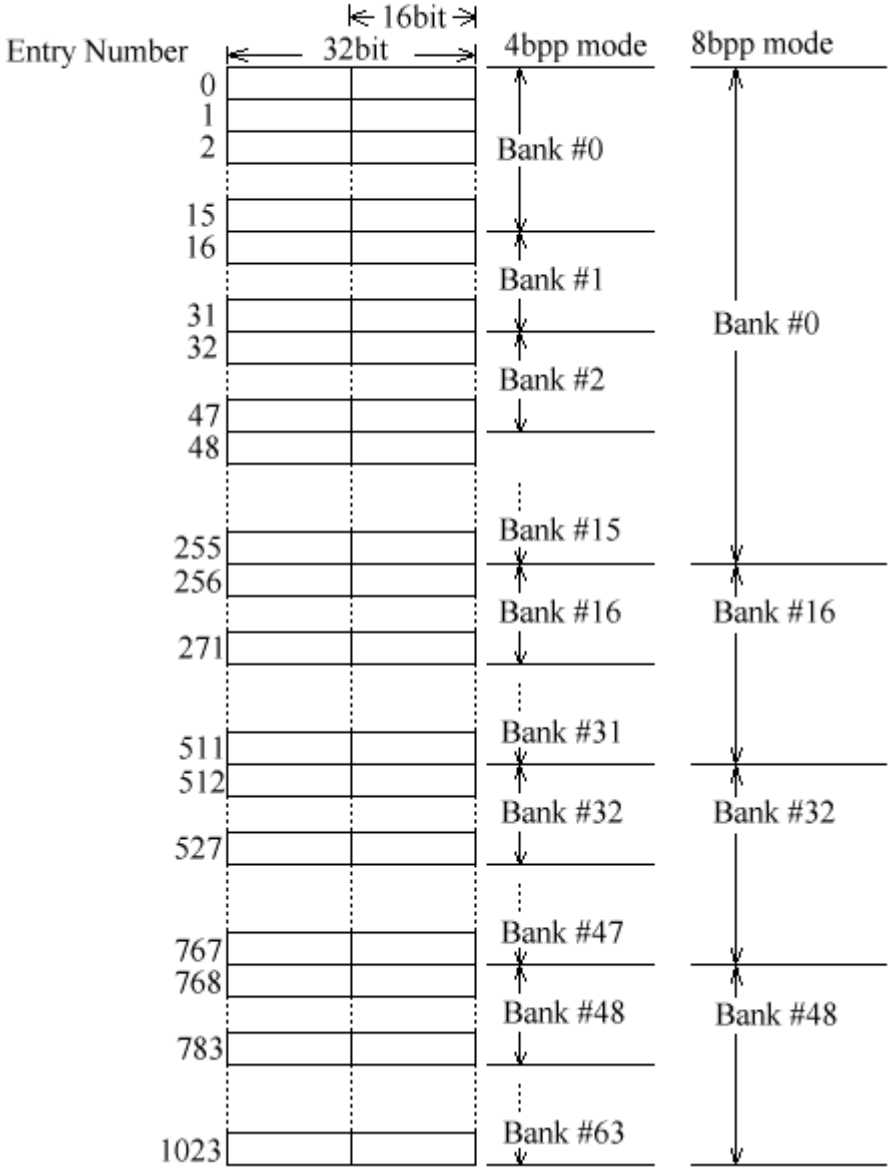
Explanation:
This function sets the on-chip palette used by the palettized texture. A palette has a total of 1,024 entries. The number of entries is the same regardless of whether the screen mode is 16 bpp or 32 bpp. Because a palette can be read at 1 clock/pixel in 16-bpp screen mode, the speed is higher in 32-bpp mode (2 clocks/pixel).
With Palettized-4bpp, the 1,024 entries are divided into 64 banks (1,024 entries/16 colors = 64 banks). With Palettized-8bpp, the 1,024 entries are divided into four banks (1,024 entries/256 colors = 4 banks). The banks are not separated physically, each bank being created by calculating pointers to the 1,024 entries.
The 4-bpp palette texture and 8-bpp palette texture can exist together in one scene, but the overlapping portion of the 1,024 entries is shared. Changing the contents of a palette, therefore, affects both the 4-bpp and 8-bpp textures.
The bank of a palette can be specified in units of VERTEX (polygon). Specify a bank number by using the `PaletteBank` member of `KMVERTEXCONTEXT`. The entry that is actually used is selected as follows, depending on the palette bank number (`PaletteBank`) and index value of each pixel of the texture (`index_data`).

```
if (PixelFormat == 8BPP)
{
    palette_entry = (PaletteBank << 4) & 0x300 + index_data;
}

if (PixelFormat == 4BPP)
{
    palette_entry = (PaletteBank << 4) + index_data;
}
```

A value of 0 to 63 can be specified for `PaletteBank` in 4-bpp mode.
Also, 0 to 63 can be specified in 8-bpp mode, but only four types of values, 0 (0 to 15), 16 (16 to 31), 32 (32 to 47), and 48 (48 to 63), can be used in this mode because only the higher two bits of the six are valid for a `PaletteBank` value.

Palette Register Structure



Caution Palette data setting (kmSetPaletteData/kmSetPaletteBankData) cannot precede palette mode setting (kmSetPaletteMode). If the palette mode type does not match the palette data type, invalid data will be set in the palette register.

Argument:

pPaletteTable (input)

This argument specifies a pointer to a palette setting array. The array is defined as follows:

```
KMPALETTEDATA   PaletteTable;
```

Example) The following coding is for setting 16-bpp data in the first 256 entries of a palette.

```
j = 0;
for(i = 0; i < 512; i+=2) {
        PaletteTable.dwPaletteData[j++]
                = (KMDWORD)((pdata[i+1]*256) + pdata[i]);
}
kmSetPaletteData(&PaletteTable);
```

The number of elements constituting the palette data must be 1,024. If there are no 1,024 elements, KAMUI may not operate normally.

Return value:

KMSTATUS_SUCCESS	Set successfully

3.4.8 Rewriting Part of the On-Chip Palette Data

```
KMSTATUS kmSetPaletteBankData(
              KMINT32             PaletteEntry,
              KMINT32             DataSize,
              PKMPALETTEDATA      pPaletteTable
              )
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	♥

Explanation:
This function rewrites a specified portion of the on-chip palette used by the palettized texture. See the descriptions of `kmSetPaletteData` for the structure of the palette.
The values that can be specified by `PaletteEntry` are 0 to 1,023 for both 4- and 8-bpp palette modes. The values need not be aligned with a bank boundary. They can start at any entry.
Data items in an area pointed to by `pPaletteTable` are sent to the palette between entries `PaletteEntry` and `PaletteEntry + DataSize` sequentially, starting at the beginning of the area.
If `PaletteEnrty + DataSize` > 1,024, data for palette numbers greater than 1,023 is ignored. Put another way, data transfer ends at palette number 1,023.

Caution Palette data setting (kmSetPaletteData/kmSetPaletteBankData) cannot precede palette mode setting (kmSetPaletteMode). If the palette mode type does not match the palette data type, invalid data will be set in the palette register.

Arguments:
`PaletteEntry` (input)
This argument specifies the first entry number of a palette where data is to be written, using a number between 0 and 1,023. A palette portion that begins with the specified entry number will be rewritten.

`DataSize` (input)
This argument specifies the size of the data to be written (number of entries), using a number between 1 and 1,024.

`pPaletteTable` (input)
This argument is a pointer to a palette setting array. The array is defined as follows:

```
KMPALETTEDATA       PaletteTable;
```

The number of elements constituting the palette data must be greater than or equal to the value specified in **`DataSize`**. If the number of elements is less than that value, normal operation is not guaranteed.

Return value:
`KMSTATUS_SUCCESS` Set successively

Example:
```
kmSetPaletteBankData( 32, 64, pPaletteTable);
```
This example coding rewrites 64 entries in the palette, starting at entry 32.

3.4.9 Setting the Border Color

```
KMSTATUS kmSetBorderColor(KMPACKEDARGB BorderColor)
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	♥

Explanation:
This function sets the color for borders (for portions outside the display screen).

Argument:
`BorderColor` (input)
This argument specifies a packed ARGB color.

Return value:
`KMSTATUS_SUCCESS` Set successfully

3.4.10 Registering the Rendering Parameter of the Background Plane

```
KMSTATUS        kmSetBackGroundRenderState (
                PKMVERTEXCONTEXT pVertexContext)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function registers information in a KMVERTEXCONTEXT structure set by kmProcessVertexRenderState to the system as the rendering parameters of the background plane. When subsequently setting the background plane (kmSetBackGroundPlane), KMVERTEXCONTEXT specified here becomes valid.
In VERTEXCONTEXT of Background, DSTBlendingMode must be zero.
To change the background plane, it is necessary to execute kmProcessVertexRenderState, kmSetBackGroundRenderState, then kmSetBackGroundPlane.

Argument:
pVertexContext (input)
This argument sets a pointer to context.

Return value:
KMSTATUS_SUCCESS Registering rendering parameters successful

3.4.11 Setting the Background Plane

```
KMSTATUS kmSetBackGroundPlane(PVOID pVertex[3],
                              KMVERTEXTYPE VertexType,
                              KMINT32 StructSize)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function registers a background plane. Before using this function, it is necessary to call `kmSetBackGroundRenderState`. If the Z value of BackgroundPlane is greater than the object to be displayed, the object may not be displayed (keep the value of 1/w less than the object to be displayed. The default 1/w value of a background plane is 0.001).
To change the background plane, it is necessary to execute `kmProcessVertexRenderState`, `kmSetBackGroundRenderState`, and `kmSetBackGroundPlane` in this order.

Arguments:
`pVertex[3]` (input)
This argument is a pointer to a vertex data structure that indicates coordinates on a background plane. For details, see the description of `kmSetVertex`.

`VertexType` (input)
This argument indicates the data type of vertex data. For details, see the description of `kmSetVertex`.

`StructSize` (input)
This argument indicates the data type size of vertex data. Specify it like `sizeof (KMVERTEX_01)` in accordance with the type used for the vertex data. For details, see the description of `kmSetVertex`.

Return value:
`KMSTATUS_SUCCESS` Registering background plane successful

3.4.12 Setting Autosort Mode

```
KMSTATUS kmSetAutoSortMode(KMBOOLEAN bEnable)
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	♥

Explanation:
This function turns on/off autosort mode of translucent polygon.

There are two translucent polygon drawing modes, auto-sort mode and pre-sort mode. This function can select either mode for each scene separately.

Auto-sort mode
The hardware automatically sorts translucent polygons in pixel units in ascending order of Z-coordinates (inner pixels first) rather than the order in which they were registered in KAMUI before being drawn. In this case, α blending is performed correctly even if translucent polygons cross one another. If there are many overlapping translucent polygons, their processing speed is reduced, however.
The `DepthMode` of `VERTEXCONTEXT` is ignored in the auto-sort mode. The Z-coordinates of the pixels are compared, always using `KM_GREATEREQUAL`. If two pixels are at the same Z-coordinate, they are drawn in the order in which they were registered with KAMUI.

Pre-sort mode
Polygons are drawn in the order in which they were registered with KAMUI. So, the application program has to sort them into order of Z-coordinates. If translucent polygons cross one another, α blending cannot be performed correctly. If it is easy for the application program to sort polygons into the order of Z-coordinates (as with 2D sprites), the pre-sort mode should be used, as it is faster than the auto-sort mode.

Argument:
`bEnable` (input)
If it is TRUE, this argument specifies autosort mode for translucent planes. If it is FALSE, it emulates software-based sorting, which is the conventional sorting type.

Return value:
`KMSTATUS_SUCCESS` Set successfully

3.4.13 Specifying Pixel-Unit Clipping

```
KMSTATUS kmSetPixelClipping( KMINT32      Xmin,
                             KMINT32      Ymin,
                             KMINT32      Xmax,
                             KMINT32      Ymax)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function sets pixel-unit clipping for rendering output to the frame buffer.

Arguments:
`Xmin, Ymin, Xmax, Ymax` (input)
These arguments specify the coordinates of the upper-left and lower-right corners of a clipping area in pixel units. "`(Xmin,Ymin) - (Xmax,Ymax)`" cannot be larger than the screen size. If screen mode is 24 bpp, coordinates specified for a clipping area must be even numbers; in other words, the clipping area can be specified only in two-pixel units. If they are not even, values that are 1 greater than specified are assumed for `(Xmin,Ymin)`, and values that are 1 less than specified are assumed for `(Xmax,Ymax)`.

Return values:

`KMSTATUS_SUCCESS`	Set successfully
`KMSTATUS_INVALID_PARAMETER`	Invalid parameter

3.4.14 Specifying the Stride Size

```
KMSTATUS kmSetStrideWidth(KMINT32  nWidth)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function sets the stride size when the stride texture is used. The stride size must be a multiple of 32. The value that can be set is a multiple of 32 in the range of 32 to 992.

Argument:
`nWidth` (input)
This argument sets the stride size.

Return values:

`KMSTATUS_SUCCESS`	Set successfully
`KMSTATUS_INVALID_PARAMETER`	Invalid parameter

3.4.15 Setting the Cheap Shadow Mode

```
KMSTATUS kmSetCheapShadowMode(KMINT32 nIntensity)
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	♥

Explanation:
This function selects the cheap (simple) shadow mode. The cheap shadow mode is intended to represent the shadow of polygons by lowering their luminance when they approach the modifier volume.

After cheap shadow mode has been set by this function, all the modifier volume are set in cheap shadow mode. Coexistence with two-parameter polygons in a scene is not allowed. To terminate cheap shadow mode, enter a negative number as the argument, then call this function.

The cheap shadow mode is turned on and off by issuing this function before `kmProcessVertexRenderState` for the `VERTEXCONTEXT` of polygons to be influenced by the mode.
Once the cheap shadow mode is turned on, `kmProcessVertexRenderState` need not be executed if only the intensity of the shadow is to be changed.

Similarly to the two-parameter volume, `KM_MODIFIER_A` is set in the `SelectModifier` member of the `VERTEXCONTEXT` for the polygons to be influenced by the cheap shadow mode. The vertex data used consists of regular one-parameter polygons.

Argument:
`nIntensity` (input)
This argument sets the luminance of a polygon in the modifier volume, using a value from 0 to 255. The hardware multiplies the base color and offset color for the polygon by the specified value after it is divided by 256. If the argument specifies 128, the multiplier is 0.5 (= 128/256). If a negative value is input, the setting of cheap shadow mode is completed, and the normal 2-parameter polygon becomes valid from the scene.

Return values:

`KMSTATUS_SUCCESS`	Set successfully
`KMSTATUS_INVALID_PARAMETER`	Invalid parameter

3.4.16 Specifying the Number of VsyncWait States

```
KMSTATUS kmSetWaitVsyncCount (KMINT32 VWaitNum)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function specifies how many times Vsync has been passed before the display is flipped. This function is used to obtain a constant frame rate.
If VwaitNum is set to 2 for NTSC signals, for example, if KAMUI flips the display at the second Vsync, rather than the first Vsync, after the hardware has finished rendering, even if the rendering was completed within 16.67 ms. If the rendering takes two or more Vsync periods even when VwaitNum is 2, however, KAMUI flips the display at the first Vsync after the rendering has been completed.
The default value that is assumed if this function is not called is 1. Therefore, Flip is executed at the next Vsync after rendering has been completed.

Argument:
`VwaitNum` (input)
This argument specifies the number of `Wait` states of `Vsync`. Set a value of 1 or greater.
If the argument is 0, normal operation is not guaranteed.
A negative value is ignored even if specified.

Return value:
`KMSTATUS_SUCCESS` Set successfully

3.4.17 Forced Reset of Renderer

```
KMSTATUS kmResetRenderer(VOID)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function resets the rendering pipeline through software. It is used for forced reset if data in a strip cannot be fully drawn when a strip buffer is used.

Argument:
None

Return value:

`KMSTATUS_SUCCESS`	Reset successfully

3.4.18 Setting the Split Screen Mode

```
KMSTATUS kmSetUserClipLevelAdjust(KMADJASTTYPE   Adjust,
                                  PKMINT32      pLine  )
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	♥

Explanation:
This function specifies the horizontal screen display mode. With this function, it is possible to position the horizontal boundary of the user clipping area almost at the center of the screen. Horizontally split screens are used, for example, for fighting games.

Usually, the screen is split using `kmSetUserClipping/kmSetUserClippingDirect`, which can specify the coordinates of a clipping area only in 32-pixel units because of hardware constraints, however. So it is impossible to split the clipping area at the center of the screen.

To split the screen horizontally on KAMUI, use this function to specify `KM_LEVEL_ADJUST_HALF` before setting the Y-coordinates of the user clipping areas as listed below. The horizontal boundary of the clipping areas will be placed as close as possible to the center of the screen.

Vertical screen resolution	User clipping area Y-coordinate
240	0 to 119 and 120 and 239
480	0 to 255 and 256 to 479

In this case, several lines in the upper section of the screen disappear. The number of these lines can be obtained from `pLine`.

Arguments:
`Adjust` (input)
This argument specifies the horizontal screen display mode by selecting one from the following:

KMADJUSTTYPE	
`KM_LEVEL_ADJUST_NORMAL`	The ordinary display mode is selected.
`KM_LEVEL_ADJUST_HALF`	The split screen mode is selected.

`pLine` (output)
If the `Adjust` argument specifies `KM_LEVEL_ADJUST_HALF`, KAMUI returns the amount of vertical screen shift in pixel units to this variable. As many lines as the value of `pLine`, counted from the top of the screen, will disappear. When registering vertexes, add this value to the Y-coordinate of each vertex. `NULL` can be specified in `pLine`. In this case, however, the user cannot obtain the number of lines to be added to the Y-coordinate of a vertex.

If the `Adjust` argument specifies `KM_LEVEL_ADJUST_NORMAL`, the contents of the area specified by `pLine` will not be changed.

Return value:
`KMSTATUS_SUCCESS` Set successfully

3.4.19 Screen Blanking

```
KMSTATUS kmBlankScreen(KMBOOLEAN bBlanking)
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	♥

Explanation:
This function switch the screen to blanking/non-blanking. Mainly this function will be used for Getting Gun-Position. If you want to fill the screen as white, set border color to white and call this function.

Argument:
bBlanking(input)
TRUE: Stop displaying Frame Buffer and screen color becomes border color.
FALSE: Replace the blanking.

Return value:
`KMSTATUS_SUCCESS` blanking/non-blanking successfully

3.4.19 Getting Gun Position

```
KMSTATUS kmGetGunTriggerPos(PKMDWORD pHPos, PKMDWORD pVPOS)
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	♥

Explanation:
This function could get Gun Position. This function should be used with kmBlankScreen.

Argument:
pHPos: pointer to the position valuable from H-blank OUT.
pVPos: pointer to the position valuable from V-blank OUT

Return value:
`KMSTATUS_SUCCESS` getting position successfully

Note) If you use kmSetUserClipLevelAdjust, actual position and the kmGetGunTriggerPos value is different. Please adjust the pLine which is the return value of kmSetUserClipLevelAdjust.

3.5 SETTING PARAMETERS OF EACH VERTEX

3.5.1 VERTEXCONTEXT

KAMUI has all parameters that can be set for each vertex (strip) in a KMVERTEXCONTEXT structure. An application can have and selectively use two or more KMVERTEXCONTEXT structures.
To do this, the application first allocates the KMVERTEXCONTEXT structure and define member values. Next, KMVERTEXCONTEXT is completed by `kmProcessVertexRenderState`. Then, the structure is registered in the system by `kmSetVertexRenderState`. The finished structures can be switched just by executing `kmSetVertexRenderState`. To modify some members of a finished KMVERTEXCONTEXT structure, `kmProcessVertexRenderState` and `kmSetVertexRenderState` must be executed again.

A polygon (or two-parameter polygon) influenced by a modifier volume requires two KMVERTEXCONTEXT structures inside and outside the modifier volume. The application allocates two KMVERTEXCONTEXT structures and specifies the parameters inside the modifier volume in one structure and the parameters outside the modifier volume in the other structure. The parameters outside the modifier volume are registered in the system by `kmProcessVertexRenderState` and `kmSetVertexRenderState`. The parameters inside the modifier volume are registered in the system by `kmProcessVertexRenderState` and `kmSetModifierRenderState`.

When using VERTEXCONTEXT by `kmProcessVertexRenderState` for the first time, all the members must be specified. Set all the flags for `RenderState` and define all the parameters. The operation is not guaranteed if any bit is undefined.

<KMVERTEXCONTEXT>
Structure holding parameters that can be specified for each vertex (strip). This structure has the following members.

```
typedef struct tagKMVERTEXCONTEXT
{
KMDWORD                 RenderState;            // Render Context

/* for Global Parameter */
KMPARAMTYPE             ParamType               // Parameter Type
KMLISTTYPE              ListType                // List Type
KMCOLORTYPE             ColorType               // Color Type
KMUVFORMAT              UVFormat                // UV format
```

```
/* for ISP/TSP Instruction Word */
KMDEPTHMODE               DepthMode;              // Specified DepthMode
KMCULLINGMODE             CullingMode;            // Culling Mode
KMSCREENCOORDINATION      ScreenCoordination;     // Screen Coordination
                                                  (ignored by Holly)
KMSHADINGMODE             ShadingMode;            // Shading Mode
KMMODIFIER                SelectModifier;         // Modifier Volume Valiant
KMBOOLEAN                 bZWriteDisable;         // Z Write Disable

/* for TSP Control Word */
KMBLENDINGMODE            SRCBlendingMode;        // Source Blending Mode
KMBLENDINGMODE            DSTBlendingMode;        // Desitination Blending Mode
KMBOOLEAN                 bSRCSel;                // Source Select
KMBOOLEAN                 bDSTSel;                // Distination Select
KMFOGMODE                 FogMode;                // Fogging
KMBOOLEAN                 bUseSpecular;           // Specular Highlight
KMBOOLEAN                 bUseAlpha;              // Alpha
KMBOOLEAN                 bIgnoreTextureAlpha;    // Ignore Texture Alpha
KMFLIPMODE                FlipUV;                 // Flip U,V
KMCLAMPMODE               ClampUV;                // Clamp U,V
KMFILTERMODE              FilterMode;             // Texture Filter
KMBOOLEAN                 bSuperSample;           // Anisotoropic Filter
KMDWORD                   MipMapAdjust;           // Mipmap D Adjust
KMTEXTURESHADINGMODE      TextureShadingMode;     // Texture Shading Mode
KMBOOLEAN                 bColorClamp;            // Color Clamp
KMDWORD                   PaletteBank;            // Palette Bank

/* for Texture Control Bits/Address */
PKMSURFACEDESC            pTextureSurfaceDesc;    // Texture DESC Pointer

/* Intensity FaceColor Setting */
KMFLOAT                   fFaceColorAlpha;        // Face Color Alpha
KMFLOAT                   fFaceColorR;            // Face Color Red
KMFLOAT                   fFaceColorG;            // Face Color Green
KMFLOAT                   fFaceColorB;            // Face Color Blue
```

```
/* Intensity Specular Highlight Setting */
KMFLOAT                  fOffsetColorAlpha;       // Specular Alpha
KMFLOAT                  fOffsetColorR;           // Specular Red
KMFLOAT                  fOffsetColorG;           // Specular Green
KMFLOAT                  fOffsetColorB;           // Specular Blue

/* Variables Internally Used */
KMDWORD                  GLOBALPARAMBUFFER;       // Global Parameter Buffer
KMDWORD                  ISPPARAMBUFFER;          // ISP Parameter Buffer
KMDWORD                  TSPPARAMBUFFER;          // TSP Parameter Buffer
KMDWORD                  TexturePARAMBUFFER;      // TextureParameter Buffer

/* for ModifierInstruction */
KMDWORD   ModifierInstruction;         /* ModifierInstruction*/
KMFLOAT   fBoundingBoxXmin;            /* BoundingBoxXmin(ShadowVolume)*/
KMFLOAT   fBoundingBoxYmin;            /* BoundingBoxYmin(ShadowVolume)*/
KMFLOAT   fBoundingBoxXmax;            /* BoundingBoxXmax(ShadowVolume)*/
KMFLOAT   fBoundingBoxYmax;            /* BoundingBoxYmax(ShadowVolume)*/

/* Added on Ver.1.30 */
KMBOOLEAN                bDCalcExact;             // D-param calc
KMSTRIPLENGTH            StripLength              // Strip Length
KMUSERCLIPMODE           UserClipMode             // UserClip Mode

} KMVERTEXCONTEXT, *PKMVERTEXCONTEXT;
```

To modify some or all members, specify the type of the member to be modified in a `RenderState` structure using the following status flag. At the same time, specify a value for the corresponding structure member. Then, execute `kmProcessVertexRenderState`.

The following status flags can be specified in **`RenderState`**.

```
KM_PARAMTYPE             0x00100000, /* Parameter Type */
KM_LISTTYPE              0x00200000, /* List Type */
KM_STRIPLENGTH           0x08000000, /* Divided Strip Length */
KM_USERCLIPMODE          0x10000000, /* User Clip Mode */
KM_COLORTYPE             0x00400000, /* Specified Color Format */
KM_UVFORMAT              0x00800000, /* Specified UV Format */
KM_DEPTHMODE             0x00000001, /* Z-value Comparison Mode Setting */
KM_CULLINGMODE           0x00000002, /* Culling Mode Setting */
KM_SCREENCOORDINATION    0x00000004, /* Coordinate System Setting */
KM_SHADINGMODE           0x00000008,
        /* Texture Gouraud, Texture Flat, Non-Texture Gouraud */
KM_MODIFIER              0x00000010, /* Modifier Volume Valiant No or A */
KM_ZWRITEDISABLE         0x00000020, /* Z Write Disable or not */
```

```
KM_SRCBLENDINGMODE        0x00000040, /* Blending Mode */
KM_DSTBLENDINGMODE        0x00000080, /* Blending Mode */
KM_SRCSELECT              0x01000000, /* SRC Blending Select */
KM_DSTSELECT              0x02000000, /* DST Blending Select */
KM_FOGMODE                0x00000100, /* Fog Non or Vertex or Table */
KM_USESPECULAR            0x00000200, /* Specular Highlighted or not */
KM_USEALPHA               0x00000400, /* Alpha Blended or not */
KM_IGNORETEXTUREALPHA     0x00000800, /* Ignore Texture Alpha */
KM_FLIPUV                 0x00002000, /* Texture Flipping */
KM_CLAMPUV                0x00001000, /* Texture Clamping */
KM_FILTERMODE             0x00004000, /* Point-sample or Bilinear or Trilinear */
KM_SUPERSAMPLE            0x00008000, /* Anisotoropic Filter */
KM_MIPMAPDADJUST          0x00010000, /* MipMap D Adjust */
KM_TEXTURESHADINGMODE     0x00020000, /* Modulate, Decal Alpha, Modulate Alpha */
KM_COLORCLAMP             0x00040000 /* Color Clamping */
KM_PALETTEBANK            0x00080000 /* Palette Bank */
KM_DCALCEXACT             0x04000000 /* DCALC Exact */
```

The following values can be specified in the members:

ParamType

Specify a vertex data type.
One of the following values can be specified:
KMPARAMTYPE

```
KM_POLYGON           = 0     // Normal trigonal polygon
KM_MODIFIERVOLUME    = 1     // Modifier volume (shadow/light)
KM_SPRITE            = 2     // Sprite (tetragonal polygon)
```

ListType

Specify the type of a list which stores vertex data.
One of the following values can be specified:
KMLISTTYPE

```
KM_OPAQUE_POLYGON        = 0  // Opaque polygon
KM_OPAQUE_MODIFIER       = 1  // Opaque modifier volume
KM_TRANS_POLYGON         = 2  // Translucent/transparent polygon
KM_TRANS_MODIFIER        = 3  // Translucent/transparent
                                 modifier volume
KM_PUNCHTHROUGH_POLYGON  = 4  // Punchthrough polygon
```

(This value cannot be specified for ARC1 or CLX1. If it is specified, it is assumed to be KM_TRANS_POLYGON.)

StripLength

Specify the size (in polygons) of a strip into which the hardware is to disassemble the input vertex strip data.
If you want to override a value (initially set to KM_STRIP_06) specified in KmSetStripLength, set this member to a desired value. If kmProcessVertexRenderState is executed without specifying this member, the strip size becomes as many polygons as the value specified in kmSetStripLength.

Any of the following values can be selected:
KMSTRIPLENGTH

```
KM_STRIP_01   = 0   //  Divided into strips of strip length 1
KM_STRIP_02   = 1   //  Divided into strips of strip length 2
KM_STRIP_04   = 2   //  Divided into strips of strip length 4
KM_STRIP_06   = 3   //  Divided into strips of strip length 6
```

UserClipMode

Specify whether to influence the clipping area specified in kmSetUserClipping, as follows:
KMUSERCLIPMODE

```
KM_USERCLIP_DISABLE  = 0     // Disables user clipping.
KM_USERCLIP_RESERVE  = 1     // Should not be specified.
KM_USERCLIP_INSIDE   = 2     // Enables user clipping for the
                                part inside the specified
                                clipping area.
KM_USERCLIP_OUTSIDE  = 3     // Enables user clipping for the
                                part outside the specified
                                clipping area.
```

ColorType

Specify a vertex color format.
One of the following values can be specified:
KMCOLORTYPE

```
KM_PACKEDCOLOR       = 0     // 32bit ARGB packed color format
KM_FLOATINGCOLOR     = 1     // 32bit x 4 floating color format
KM_INTENSITY         = 2        // Intensity format
KM_INTENSITY_PREV_FACE_COL = 3  // Intensity format (See below.)
```

KM_INTENSITY_PREV_FACE_COL
The previously registered face color is used. If this polygon type is to be used, a polygon of KM_INTENSITY type must have been used previously at least once in the same scene. (It cannot be used for ARC1, however.)

For sprite polygons, specify ColorType as KM_PACKEDCOLOR.

UVFormat

Specify the format of the U/V coordinate parameters included with the vertex data. The 32-bit UV expresses the U/V coordinates in a 32-bit Float format conforming to IEEE754. The 16-bit UV expresses a value with an accuracy of the 32-bit UV with the lower 16 bits of the mantissa deleted.
One of the following values can be specified:

```
KMUVFORMAT
KM_32BITUV          = 0     // 32bit KMFLOAT format
KM_16BITUV          = 1     // 16bit KMFLOAT format
```

For sprite polygons, specify `UVFormat` as `KM_16BITUV`.

DepthMode

Specify the Z-coordinate comparison mode as one of the following.

```
KMDEPTHMODE
KM_IGNORE           = 0
KM_LESS             = 1
KM_EQUAL            = 2
KM_LESSEQUAL        = 3
KM_GREATER          = 4
KM_NOTEQUAL         = 5
KM_GREATEREQUAL     = 6
KM_ALWAYS           = 7
```

CullingMode

No culling, clockwise culling, counterclockwise culling, or small polygon culling can be specified. One of the following values can be specified:

```
KMCULLINGMODE
KM_NOCULLING        = 0     // No culling
KM_CULLSMALL        = 1     // Small polygon culling
KM_CULLCCW          = 2     // Counterclockwise culling
KM_CULLCW           = 3     // Clockwise culling
```

When setting `KM_CULLSMALL`, it is necessary to execute `kmSetCullingRegister` for global setting. If `KM_CULLCCW` or `KM_CULLCW` is specified, small polygon culling is performed at the same time.

ScreenCoordination

Select the screen coordinate system (1/w) or projection coordinate system (w).

KMSCREENCOORDINATION

```
KM_SCREEN           = 0     // Screen coordinate system (1/w)
KM_PROJECTIVE       = 1     // Projection coordinate system (w)
```

(This member is valid only for ARC1. It is ignored by Holly (CLX1/2). Coordinates in the screen coordinate system (KM_SCREEN) must always be used for Holly (CLX1/2).

bDCalcExact

KMDCALCEXACT

If TRUE is set, the D parameter added by CLX1/2 is calculated accurately. With ARC1, a symptom where the selection of MipMap differs between the upper-right and lower-left of the screen is observed. This symptom can be eliminated by accurately calculating CLX1/2. However, the operation speed may drop because the calculation is time-consuming. If FALSE is set, the D parameter is calculated in the same manner as ARC1. This flag of ARC1 is meaningless.

ShadingMode

Specify one of four shading modes, which are combinations of presence or absence of texture and flat or gouraud shading.

KMSHADINGMODE

```
KM_NOTEXTUREFLAT[1]     = 0    // No texture, flat shading
KM_NOTEXTUREGOURAUD     = 1    // No texture, gouraud shading
KM_TEXTUREFLAT          = 2    // Texture, flat shading
KM_TEXTUREGOURAUD       = 3    // Texture, gouraud shading
```

If KM_TEXTUREFLAT is specified, the color data of the first and second vertices of the vertex strip becomes invalid.

For stripe polygons, specify ShadingMode as KM_NOTEXTUREFLAT or KM_TEXTUREFLAT.

SelectModifier

Specify whether the a polygon (two-parameter polygon or a polygon to which cheap shadow is to be applied) is influenced by a modifier volume.
If it is specified that the polygon is to be influenced by a modifier volume, use the vertex data structure for a two-parameter polygon. In the cheap shadow mode (when kmSetCheapShadowMode is executed), use a vertex data structure for one-parameter polygons.

[1] ARC1 does not have this mode. Never try to use this mode in the ARC1 environment.

If this value is specified for ARC1, KM_NOTEXTUREGOURAUD is assumed.

KMMODIFIER

```
KM_NOMODIFIER        = 0    // Not used
KM_MODIFIER_A        = 1    // Influenced by modifier volume A
```

ARC1 and HOLLY support modifier volume A only.

bZWriteDisable

When TRUE is specified, modification of the Z value is disabled.
ARC1 does not have this function.

SRCBlendingMode, DSTBlendingMode

Specify blending mode. These members correspond to the D3D blending mode.
One of the following values can be specified.

KMBLENDINGMODE

KM_BOTHINVSRCALPHA,

The source blending parameter is set as (1 - αs, 1 - αs, 1 - αs, 1 - αs). The destination blending parameter is set as (αs, αs, αs, αs). When this value is set for SRCBlendingMode, DSTBlendingMode is overridden. When this value is set for DSTBlendingMode, SRCBlendingMode is overridden.

KM_BOTHSRCALPHA,

The source blending parameter is set as (αs, αs, αs, αs). The destination blending parameter is set as (1 - αs, 1 - αs, 1 - αs, 1 - αs). When this value is set for SRCBlendingMode, DSTBlendingMode is overridden. When this value is set for DSTBlendingMode, SRCBlendingMode is overridden.

KM_DESTALPHA,

The blending parameter is set as (αd, αd, αd, αd).

KM_DESTCOLOR,

The blending parameter is set as (αd, Rd, Gd, Bd).

KM_INVDESTALPHA,

The blending parameter is set as (1 - αd, 1 - αd, 1 - αd, 1 - αd).

KM_INVDESTCOLOR,

The blending parameter is set as (1 - αd, 1 - Rd, 1 - Gd, 1 - Bd).

KM_INVSRCALPHA,

The blending parameter is set as (1 - αs, 1 - αs, 1 - αs, 1 - αs).

KM_INVSRCCOLOR,

The blending parameter is set as (1 - αs, 1 - Rs, 1 - Gs, 1 - Bs).

KM_SRCALPHA,

The blending parameter is set as (αs, αs, αs, αs).

KM_SRCCOLOR,

The blending parameter is set as (αs, Rs, Gs, Bs).

KM_ONE,

The blending parameter is set as (1, 1, 1, 1).

KM_ZERO

The blending parameter is set as (0, 0, 0, 0).

Note (αs, Rs, Gs, Bs) represents a source color, and (αd, Rd, Gd, Bd) represents a destination color.
Set `DSTBlendingMode` to zero for `VERTEXCONTEXT` for the background plane (see the description of `kmSetBackGroundRenderState`).

bSRCSel

The contents of the second accumulation buffer are used as the source color.

bDSTSel

The contents of the second accumulation buffer are used as the destination color.

FogMode

Specify a fog mode. In the ARC1 and HOLLY environments, two fog modes can be used: In FogTable mode, fogging is performed with reference to a table according to the depth value; In FogVertex mode, fog parameters are specified in the α channel section of `OffsetColor` of each vertex. One of the following values can be specified:

KMFOGMODE

`KM_FOGTABLE`	The `fog` α value is generated from the table data corresponding to a depth value by using linear interpolation.
`KM_FOGTABLE_2`	The polygon color is changed to the `fog` color, and the polygon α value is changed to the `fog` α value. (This value cannot be specified for `ARC1`.*)
`KM_FOGVERTEX`	The `OffsetColor` α value is used as the `fog` α value.
`KM_NOFOG`	No `fog` processing is performed.

* If `KM_FOGTABLE_2` is specified for `ARC1`, `KM_NOFOG` is assumed.

bUseSpecular

Specify whether specular highlight (offset color) is used. When `TRUE` is specified, an offset color is used. When using an offset color, use a vertex data structure that includes the offset color data.

When using bump mapping, specify `bUseSpecular` as `TRUE`.

bUseAlpha

Specify whether to enable the α bit in the shading color. Entering TRUE enables the α bit.

bIgnoreTextureAlpha

When TRUE is specified, the α bit in the texture data is ignored. Hardware ignores the transparency information included in the texture.

FlipUV

Specify whether to flip patterns when the texture is mapped repeatedly. One of the following values can be specified. If FlipUV and ClampUV are specified at the same time, the specification of FlipUV is ignored.

KMFLIPMODE

```
KM_NOFLIP      // Not flipped
KM_FLIP_V      // Flipped in the V-coordinate direction
KM_FLIP_U      // Flipped in the U-coordinate direction
KM_FLIP_UV     // Flipped in the U-coordinate and V-
                  coordinate directions
```

ClampUV

Specify a texture clamp.

One of the following values can be specified. If FlipUV and ClampUV are specified at the same time, the specification of FlipUV is ignored.

KMCLAMPMODE

```
KM_NOCLAMP     // Not clamped
KM_CLAMP_V     // Clamped in the V-coordinate direction
KM_CLAMP_U     // Clamped in the U-coordinate direction
KM_CLAMP_UV    // Clamped in the U-coordinate and V-
                  coordinate directions
```

FilterMode

Specify a texture filter mode. The point-sample, bilinear, or trilinear filter mode can be selected. However, bilinear mode is assumed if trilinear mode is specified with the ARC1. One of the following values can be specified:

KMFILTERMODE

```
KM_POINT_SAMPLE   // Point sampling
KM_BILINEAR       // Bilinear filter
KM_TRILINEAR_A    // Trilinear pass 1
KM_TRILINEAR_B    // Trilinear pass 2
```

bSuperSample

When TRUE is specified, the quad-speed super sampling filter (anisotropic filter) is used. This selection causes the quality of the texture mapping to be improved.

MipMapAdjust

The D parameter calculation for MIPMAP level selection is multiplied by a coefficient. By this multiplication, aliasing can be adjusted. The four low-order bits of the value are effective. The four bits form floating point data consisting of a two-bit integer portion and a two-bit fractional portion. Any of the following can be specified:

```
KM_MIPMAP_D_ADJUST_0_25    0x00000001     /* D=0.25 */
KM_MIPMAP_D_ADJUST_0_50    0x00000002     /* D=0.50 */
KM_MIPMAP_D_ADJUST_0_75    0x00000003     /* D=0.75 */
KM_MIPMAP_D_ADJUST_1_00    0x00000004     /* D=1.00 */
KM_MIPMAP_D_ADJUST_1_25    0x00000005     /* D=1.25 */
KM_MIPMAP_D_ADJUST_1_50    0x00000006     /* D=1.50 */
KM_MIPMAP_D_ADJUST_1_75    0x00000007     /* D=1.75 */
KM_MIPMAP_D_ADJUST_2_00    0x00000008     /* D=2.00 */
KM_MIPMAP_D_ADJUST_2_25    0x00000009     /* D=2.25 */
KM_MIPMAP_D_ADJUST_2_50    0x0000000A     /* D=2.50 */
KM_MIPMAP_D_ADJUST_2_75    0x0000000B     /* D=2.75 */
KM_MIPMAP_D_ADJUST_3_00    0x0000000C     /* D=3.00 */
KM_MIPMAP_D_ADJUST_3_25    0x0000000D     /* D=3.25 */
KM_MIPMAP_D_ADJUST_3_50    0x0000000E     /* D=3.50 */
KM_MIPMAP_D_ADJUST_3_75    0x0000000F     /* D=3.75 */
```

Usually, specify KM_MIPMAP_D_ADJUST_1_00 (1.0).

TextureShadingMode

Specify a D3D texture blending mode. One of the following values can be specified:

KMTEXTURESHADINGMODE

```
KM_DECAL              = 0
KM_MODULATE           = 1
KM_DECAL_ALPHA        = 2
KM_MODULATE_ALPHA     = 3
```

KM_DECAL

An offset value is added to the texture color.

The texture α value is still used.

```
Pixel Color = TextureRGB + OffsetRGB
Pixel α = Texture α
```

(This argument cannot be specified for ARC1. If it is specified for ARC1, KM_DECAL_ALPHA is assumed.)

KM_MODULATE

The texture color is multiplied by the color resulting from shading. Texture α is replaced by shading α.

Pixel Color = $Shading_{RGB} \times Texture_{RGB} + Offset_{RGB}$

Pixel α = Texture α

KM_DECAL_ALPHA

The texture color is blended with the shading color, according to texture α.

Pixel Color = $(Texture_{RGB} \times \text{Texture } \alpha) + \{Shading_{RGB} \times (1 - \text{Texture } \alpha)\} + Offset_{RGB}$

Pixel α = Shading α

KM_MODULATE_ALPHA

The texture color is multiplied by the shading color. Texture α is multiplied by shading α.

Pixel Color = $(Texture_{RGB} \times Shading_{RGB}) + Offset_{RGB}$

Pixel α = Shading $\alpha \times$ Texture α

bColorClamp

Specify whether a color is clamped. When `TRUE` is specified, a pixel color is clamped to the clamp value specified by `kmSetColorClampValue`. ARC1 does not support this function.

PaletteBank

Specify a palette bank number. This value is valid only when palettized texture is specified. The value that can be specified is 0 to 63 in the Palettized -4bpp mode. It is also 0 to 63 in the Palettized -8bpp mode, but only four types of values, 0 (0 to 15), 16 (16 to 31), 32 (32 to 47), and 48 (48 to 63), can be used in this mode because only the higher two bits of the six are valid (for details, see the description of `kmSetPaletteData`).

ARC1 does not support this function.

pTextureSurfaceDesc

Specify a pointer to the surface structure of the texture surface.

To change only the texture without changing the other parameters of `VERTEXCONTEXT`, only change the texture address of a `pTextureSurfaceDesc` member and call `kmProcessVertexRenderState` and `kmSetVertexRenderState`. If `ShadingMode` is specified as `KM_TEXTUREFLAT` or `KM_TEXTUREGOURAUD`, information of a `pTextureSurfaceDesc` member is loaded into the system each time `kmProcessVertexRenderState` is called.

If `NULL` is specified as a `pTextureSurfaceDesc` member, the texture address is not loaded even if `ShadingMode` is `KM_TEXTUREFLAT` or `KM_TEXTUREGOURAUD`. If the texture is not determined and if other parameters are to be set in advance, specify `NULL` as a `pTextureSurfaceDesc` member.

`ModifierInstruction`

Specify the type of the polygon data to be registered when a modifier volume is registered. Specify any of the following:

`KM_MODIFIER_INCLUDE_FIRST_POLY`
: Indicates the first polygon of the `Inclusion` modifier volume.

`KM_MODIFIER_EXCLUDE_FIRST_POLY`
: Indicates the last polygon of the `Exclusion` modifier volume.

`KM_MODIFIER_INCLUDE_LAST_POLY`
: Indicates the last polygon of the `Inclusion` modifier volume.

`KM_MODIFIER_EXCLUDE_LAST_POLY`
: Indicates the last polygon of the `Exclusion` modifier volume.

`KM_MODIFIER_NORMAL_POLY`
: Indicates a polygon other than those above.

This member does not have to be specified for anything other than a modifier volume.

`fBoundingBoxXmin`, `fBoundingBoxYmin` `fBoundingBoxXmax`, `fBoundingBoxYmax`

Specify an area on the screen in which the effect of a modifier volume is valid. The effect of a modifier volume becomes valid only in the range specified here. By setting this value as necessary, the rendering speed can be improved.
This member does not have to be specified for anything other than modifier volume.
This member is valid only with the ARC1. It need not be specified for CLX1/2.

3.5.2 Setting Rendering Parameters for Each Vertex

```
KMSTATUS kmProcessVertexRenderState(
                PKMVERTEXCONTEXT pVertexContext)
```

IRIS+ARC1	COSMOS+ARC1	Holly
◆	◆	♥

Note In the ARC1 environment, some functions are restricted.

Explanation:
This function sets rendering parameters that are used for each vertex (strip). The function performs preprocessing for registering to the system the rendering parameter specified by `pVertexContext`.
This function generates the following values from a value specified in `pVertexContext`:

Global Parameter
Combined ISP/TSP Instruction Word
TSP Control Word
Texture Control Bits

These values will be input to the tiling accelerator. They are held in the following members in `KMVERTEXCONTEXT`.

```
pVertexContext->GLOBALPARAMBUFFER
pVertexContext->ISPPARAMBUFFER
pVertexContext->TSPPARAMBUFFER
pVertexContext->TexturePARAMBUFFER
```

To register these members (saved by this function) with the system as parameters to be used for rendering, it is necessary to call `kmSetVertexRenderState`.

Caution When using VERTEXCONTEXT for the first time, the values of all the members of VERTEXCONTEXT must be specified (initialized). Set all the flags from `pVertexContext` to `RenderState` and define all the parameters. The operation is not guaranteed if any bit is undefined.

Argument:
`pVertexContext` (input)
Specify a pointer to the context.

Return values:

`KMSTATUS_SUCCESS`	Set successfully
`KMSTATUS_INVALID_SETTING`	Illegal mode setting

3.5.3 Registering Rendering Parameters for Each Vertex

```
KMSTATUS        kmSetVertexRenderState (
                PKMVERTEXCONTEXT pVertexContext)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function registers the following members (set in `pVertexContext` by `kmProcessVertexRenderState`) and the related values with the system as parameters to be used for rendering.

```
pVertexContext->GLOBALPARAMBUFFER
pVertexContext->ISPPARAMBUFFER
pVertexContext->TSPPARAMBUFFER
pVertexContext->TexturePARAMBUFFER
```

The parameters set by calling this function will be valid for vertex (strip) registration at or after `kmStartVertexStrip`, called later.

Argument:
`pVertexContext` (input)
Specify a pointer to the `KMVERTEXCONTEXT` structure.

Return value:

`KMSTATUS_SUCCESS`	Successful registration of rendering parameters

3.5.4 Registering Rendering Parameters of a Modifier Volume

```
KMSTATUS        kmSetModifierRenderState(
                PKMVERTEXCONTEXT pVertexContext)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function registers the following members (set in KMVERTEXCONTEXT by kmProcessVertexRenderState) with the system as the second rendering parameter for polygons (two-parameter polygons) to be influenced by the modifier volume.

```
pVertexContext->TSPPARAMBUFFER
pVertexContext->TexturePARAMBUFFER
```

When polygons (two-parameter polygons) to be influenced by the subsequent modifier volumes are registered, KMVERTEXCONTEXT (specified by this function) is used as the second rendering parameter.

The following members of the KMVERTEXCONTEXT structure are ignored.
(The settings by kmSetVertexRenderState are valid.)

```
/* for Global Parameter */
KMPARAMTYPE               ParamType               /* Parameter Type */
KMLISTTYPE                ListType                /* List Type */
KMSTRIPLENGTH             StripLength             /* Strip Length */
KMUSERCLIPMODE            UserClipMode            /* UserClip Mode */
KMCOLORTYPE               ColorType               /* Color Type */
KMUVFORMAT                UVFormat                /* UV format */

/* for ISP/TSP Instruction Word */
KMDEPTHMODE               DepthMode;              /* Specified DepthMode */
KMCULLINGMODE             CullingMode;            /* Culling Mode */
KMSCREENCORDINATION       ScreenCoordination;     /* Screen Coordination */
KMSHADINGMODE             ShadingMode;            /* Shading Mode */
KMMODIFIER                SelectModifier;         /* Modifier Volume Valiant */
KMBOOLEAN                 bZWriteDisable;         /* Z Write Disable */

/* for ModifierInstruction */
KMDWORD  ModifierInstruction;     /* ModifierInstruction*/
KMFLOAT  fBoundingBoxXmin;        /* BoundingBoxXmin(ShadowVolume)*/
KMFLOAT  fBoundingBoxYmin;        /* BoundingBoxYmin(ShadowVolume)*/
KMFLOAT  fBoundingBoxXmax;        /* BoundingBoxXmax(ShadowVolume)*/
KMFLOAT  fBoundingBoxYmax;        /* BoundingBoxYmax(ShadowVolume)*/
```

Argument:
`pVertexContext` (input)
Specify a pointer to the rendering context.

Return value:

`KMSTATUS_SUCCESS`	Successful registration of rendering parameters

3.5.5 Setting Global Clipping

```
KMSTATUS kmSetGlobalClipping(KMINT32 nWidth, KMINT32 nHeight)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function specifies a global clipping area. Rendering is performed only in the area determined by the 0,0 origin, `Width`, and `Height`.

Arguments:
`NWidth` and `nHeight` (input)
These arguments specify a global clipping area as a multiple of 32. To specify a 128 x 64 area, for example, set `Width` to 4 and `Height` to 2.

Return values:

`KMSTATUS_SUCCESS`	Set successfully
`KMSTATUS_INVALID_PARAMETER`	Setting ended in failure

3.5.6 Setting a Strip Length

```
KMSTATUS kmSetStripLength(KMSTRIPLENGTH StripLength)
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	♥

Explanation:
This function sets the default value for the strip length. It specifies the length of strip, which is the number of polygons. The hardware internally divides the input vertex strip data into strips of the specified length. The system's initial value is `KM_STRIP_06`.

If a value is set in the `StripLength` member of `VERTEXCONTEXT`, it overrides the value specified by this function.

The value specified by the function is sent to the hardware via a parameter (global parameter) for specifying the beginning of strips. Therefore, this function must be issued before kmProcessVertexRenderState is called for VERTEXCONTEXT of the subjected polygon. In subsequent vertex strip registrations, the strip length specified here will be used.
In the ARC1 environment, the hardware always divides the data into strips of strip length 1.

Arguments:
`StripLength` (input)
One of the following values can be specified:

```
KMSTRIPLENGTH
       KM_STRIP_01    = 0   // Divided into strips of strip length 1
       KM_STRIP_02    = 1   // Divided into strips of strip length 2
       KM_STRIP_04    = 2   // Divided into strips of strip length 4
       KM_STRIP_06    = 3   // Divided into strips of strip length 6
```

Return value:
`KMSTATUS_SUCCESS` Set successfully

3.5.7 Direct Rewrite Mode for Rendering Status

Explanation:
This function changes part of the rendering status that was set in the system directly.

Usually, the rendering status is registered with the system by executing `kmSetVertexRenderState` after setting rendering conditions in `VERTEXCONTEXT` and executing `kmProcessVertexRenderState`. It may become necessary to modify part of the rendering status registered with the system. In this case, it is inefficient to re-execute the preprocess for `VERTEXCONTEXT`. To avoid this problem, the following function can be used to modify part of the rendering status registered with the system.

IRIS+ARC1	COSMOS+ARC1	Holly
◆	◆	♥

Note In the ARC1 environment, some functions are restricted.

Usage:

```
...(VERTEXCONTEXT setting)...
kmProcessVertexRenderState()
kmSetVertexRenderState()   // Sets VERTEXCONTEXT in the system.
kmStartVertexStrip()       // Begins polygon registration.
kmSetVertex()
......
kmChangeContext****()      // Modifies part of VERTEXCONTEXT.
kmStartVertexStrip()       // Begins to register new polygons.
kmSetVertex()
......
```

Arguments:
The arguments to be used for this function are the same as those specified for `VERTEXCONTEXT`. The function restriction in the ARC1 environment is also the same as those for `VERTEXCONTEXT`.

```
KMSTATUS kmChangeContextStripLength(KMSTRIPLENGTH StripLength)
```

Explanation: This function specifies the size (in polygons) of strips into which the tiling accelerator is to disassemble the input vertex strip data.

```
KMSTATUS kmChangeContextUserClipMode
                                (KMUSERCLIPMODE UserClipMode)
```

Explanation: This function specifies whether to influence the clipping area specified in `kmSetUserClipping`.

```
KMSTATUS kmChangeContextColorType(KMCOLORTYPE Color)
```

Explanation: This function modifies the vertex color format.

```
KMSTATUS kmChangeContextDepthMode(KMDEPTHMODE DepthMode)
```
Explanation: This function selects a Z-value comparison mode.

```
KMSTATUS kmChangeContextCullingMode(KMCULLINGMODE CullingMode)
```
Explanation: This function selects a culling mode.

```
KMSTATUS kmChangeContextZWriteDisable(
                                KMBOOLEAN bZwriteDisable)
```
Explanation: This function enables or disables the modification of the Z value.

```
KMSTATUS kmChangeContextSRCBlendMode(KMBLENDINGMODE SRCBlend)
```
Explanation: This function selects a source blending mode.

```
KMSTATUS kmChangeContextDSTBlendMode(KMBLENDINGMODE DSTBlend)
```
Explanation: This function selects a destination blending mode.

```
KMSTATUS kmChangeContextFogMode(KMFOGMODE FogMode)
```
Explanation: This function selects a fog mode.

```
KMSTATUS kmChangeContextFlipUV(KMFLIPMODE FlipMode)
```
Explanation: This function sets a texture `Flip`.

```
KMSTATUS kmChangeContextClampUV(KMCLAMPMODE ClampMode)
```
Explanation: This function sets a texture `Clamp`.

```
KMSTATUS kmChangeContextFilterMode(KMFILTERMODE FilterMode)
```
Explanation: This function sets a texture filter.

```
KMSTATUS kmChangeContextSuperSample(KMBOOLEAN bEnable)
```
Explanation: When the argument is `TRUE`, an `anisotropic filter` is used.

```
KMSTATUS kmChangeContextTextureShadingMode(
                            KMTEXTURESHADINGMODE TexShadingMode)
```
Explanation: This function specifies a texture blending mode.

```
KMSTATUS kmChangeContextColorClamp(KMBOOLEAN bColorClamp)
```
Explanation: When the argument is `TRUE`, the color is clamped according to the value of the global register.

```
KMSTATUS kmChangeContextPaletteBank(KMDWORD Bank)
```
Explanation: This function changes the current palette bank.

3.6 RECORDING VERTEX DATA

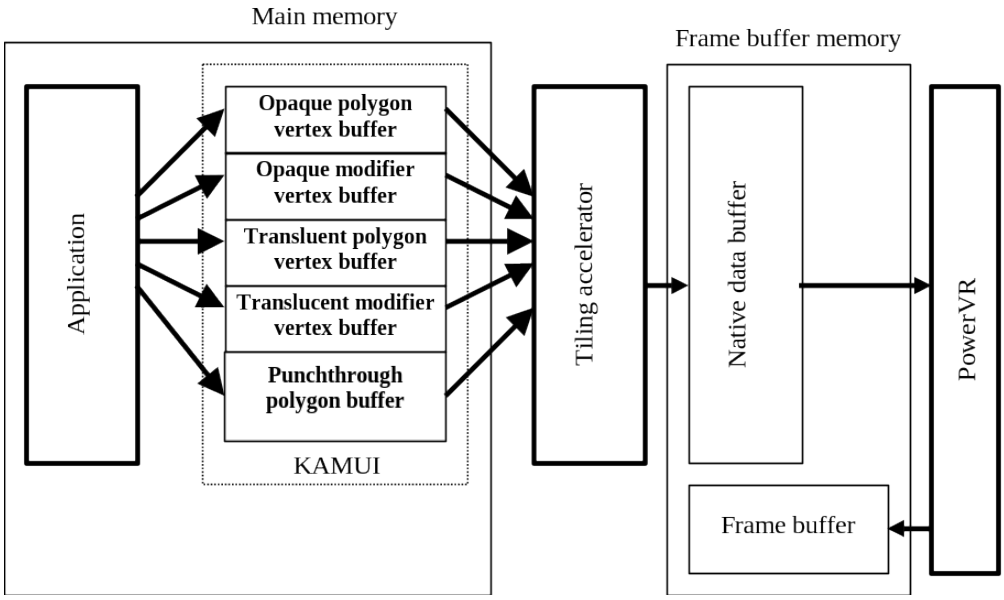
3.6.1 Recording Vertex Data

KAMUI allocates vertex data buffers. Drawing is performed by storing necessary vertex data in the buffers and issuing a rendering command. All vertex data is defined as strip-type data. When specifying a single polygon, specify strip length 1.

After initialization by KAMUI, the application program first allocates a vertex data buffer in system memory. It also executes `kmSetSystemConfiguration` (or `kmCreateFrameBufferSurface` and `kmCreateVertexBuffer`) to allocate a display frame buffer and native data buffer (for tiling accelerator output) in the frame buffer. The following five vertex data buffers are supported:

- Buffer for opaque polygon (opaque polygon)
- Buffer for opaque modifier volume (opaque modifier)
- Buffer for translucent/transparent polygon (translucent polygon)
- Buffer for translucent/transparent modifier volume (translucent modifier)
- Buffer for punch through polygon (punchthrough polygon)

All vertex data is stored in any of the five buffers.
Then, the parameters of common to scenes and the parameters common to vertices to be recorded are specified. (See Sections 3.4 and 3.5.)



Vertices are recorded by first issuing `kmStartVertexStrip`, then writing data (global parameters) indicating the beginning of vertex strips into the vertex data buffer. Then, execute `kmSetVertex` any number of times to record the vertex data. All vertex data defined by `kmSetVertex` is assumed to be a single strip. To record a new strip, issue `kmStartVertexStrip` again. At the end of a scene, issue `kmRender`.

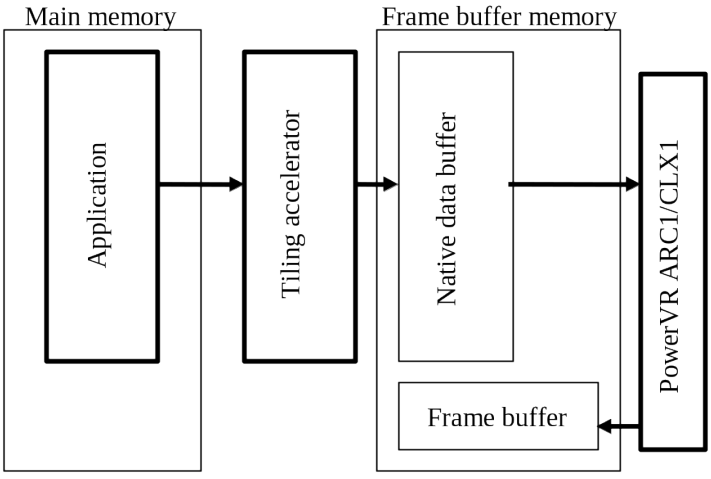
The vertex data of a single strip is defined as indicated below:

```
kmStartVertexStrip();        // Start of strip
do{
    ……
    kmSetVertex(pVertex, VertexType, sizeof(VertexType));
                             // Registration of one vertex
} while(…)
```

Caution `kmSetUserClipping` must not be issued between `kmStartVertexStrip` and `kmSetVertex`. If no `kmSetVertex` is issued after `kmStartVertexStrip`, the operation may be adversely affected.

The vertex data can also be directly written into the hardware (tiling accelerator) without allocating five vertex data buffers in main memory. This method is referred to as the **direct mode**. The former method (that allocates a vertex data buffer in main memory) is known as the **buffer mode**. The direct mode and buffer mode cannot be used simultaneously. The mode to be used is specified by `kmSetSystemConfiguration`.

In the direct mode, `kmSetSystemConfiguration` (or `kmCreateFrameBufferSurface` and `kmCreateTABuffer`) first allocates only a display frame buffer and native data buffer (for tiling accelerator output) in the frame buffer after KAMUI is initialized. Then, specify the general parameters of the scene and the parameters common to vertices to be recorded. (See Sections 3.4 and 3.5.)



Vertices are registered by first issuing `kmStartVertexStrip`**`Direct`**, then writing data (global parameters) indicating the beginning of vertex strips to the hardware. Then, execute `kmSetVertex`**`Direct`** any number of times to write the vertex data directly into the hardware. To record a new strip, issue `kmStartVertexStrip`**`Direct`** again. When the registration of data of a specific vertex type (opaque polygon, opaque modifier, translucent polygon, translucent modifier, punchthrough polygon) has been completed, `kmSetEndOfList`**`Direct`** is issued. At the end of a scene, issue `kmRender`**`Direct`**.

The vertex data of a single scene to be input to the hardware (tiling accelerator) must be classified by vertex type (opaque polygon, opaque modifier, translucent polygon, translucent modifier, punchthrough polygon). If the vertex data of a single type is input twice (the data of opaque polygon and translucent polygon is input, then the data of opaque polygon is input again, for example), the vertex data input earlier becomes invalid.

To specify user clipping, `kmSetUserClipping`**`Direct`** must be used instead of `kmSetUserClipping`.
`kmSetUserClipping`**`Direct`** must not be issued between `kmStartVertexStrip`**`Direct`** and `kmSetVertex`**`Direct`**. If no `kmSetVertex`**`Direct`** is issued after `kmStartVertexStrip`**`Direct`**, the operation may be adversely affected.

3.6.2 Vertex Data Structure

Vertex data consists of as many vertex data blocks (called strips) as the number of vertices, preceded by the hardware-specific data (global parameters) specified separately for individual strips. Several strips are combined to create a scene of one screen. To register a strip, set the global parameter with `kmStartVertexStrip`, as described earlier, then issue `kmSetVertex` for each vertex. At least three vertex parameters are necessary. The end of the strip is specified by Parameter Control Word at the beginning of the vertex parameter. Select either of the following Parameter Control Words:

```
KM_VERTEXPARAM_NORMAL       = 0xE0000000     // Normal vertex data
KM_VERTEXPARAM_ENDOFSTRIP   = 0xF0000000     // Vertex data at the end of the strip
```

If Parameter Control Word of the vertex data at the end of the strip is not `KM_VERTEXPARAM_ENDOFSTRIP`, the operation is not guaranteed.

If `KM_TEXTUREFLAT` is specified as `ShadingMode` of `VERTEXCONTEXT`, the color data of the first and second vertices of the vertex strip is invalid.

Sprite polygon

If sprite polygons are used, the Parameter Control Word for any polygon must always be `KM_VERTEXPARAM_ENDOFSTRIP`. To display multiple sprite polygons that all use `VERTEXCONTEXT`, it is possible to set vertex data for multiple sprite polygons consecutively, using `kmSetVertex` after issuing `kmStartVertexStrip`. In this case, however, the Parameter Control Word for each sprite polygon data item must always be `KM_VERTEXPARAM_ENDOFSTRIP`. `VERTEXCONTEXT` for sprite polygons must be set as follows:

```
ColorType       : KM_PACKEDCOLOR
UVFormat        : KM_16BITUV
ShadingMode     : KM_NOTEXTUREFLAT or KM_TEXTUREFLAT
```

The other settings are optional.

3.6.3 Vertex Parameters

Type 0 (Non-Textured, Packed Color) (IRIS+ARC1, COSMOS+ARC1, CLX1)

0x00	Parameter Control Word
0x04	X
0x08	Y
0x0C	Z
0x10	**Base Color**
0x14	Reserved
0x18	Reserved
0x1C	Reserved

Type 0 (Non-Textured, Packed Color) (CLX2)

0x00	Parameter Control Word
0x04	X
0x08	Y
0x0C	Z
0x10	Reserved
0x14	Reserved
0x18	**Base Color**
0x1C	Reserved

Type 1 (Non-Textured, Floating Color)

0x00	Parameter Control Word
0x04	X
0x08	Y
0x0C	Z
0x10	Base Color Alpha
0x14	Base Color R
0x18	Base Color G
0x1C	Base Color B

Type 2 (Non-Textured, Intensity) (IRIS+ARC1, COSMOS+ARC1, CLX1)

0x00	Parameter Control Word
0x04	X
0x08	Y
0x0C	Z
0x10	**Base Intensity**
0x14	Reserved
0x18	Reserved
0x1C	Reserved

Type 2 (Non-Textured, Intensity) (CLX2)

0x00	Parameter Control Word
0x04	X
0x08	Y
0x0C	Z
0x10	Reserved
0x14	Reserved
0x18	**Base Intensity**
0x1C	Reserved

Type 3 (Packed Color)

0x00	Parameter Control Word
0x04	X
0x08	Y
0x0C	Z
0x10	U
0x14	V
0x18	Base Color
0x1C	Offset Color

Type 4 (Packed Color, 16-bit UV)

0x00	Parameter Control Word
0x04	X
0x08	Y
0x0C	Z
0x10	U|V
0x14	Reserved
0x18	Base Color
0x1C	Offset Color

Type 5 (Floating Color)

0x00	Parameter Control Word
0x04	X
0x08	Y
0x0C	Z
0x10	U
0x14	V
0x18	Reserved
0x1C	Reserved
0x20	Base Color Alpha
0x24	Base Color R
0x28	Base Color G
0x2C	Base Color B
0x30	Offset Color Alpha
0x34	Offset Color R
0x38	Offset Color G
0x3C	Offset Color B

Type 6 (Floating Color, 16-bit UV)

0x00	Parameter Control Word
0x04	X
0x08	Y
0x0C	Z
0x10	U|V
0x14	Reserved
0x18	Reserved
0x1C	Reserved
0x20	Base Color Alpha
0x24	Base Color R
0x28	Base Color G
0x2C	Base Color B
0x30	Offset Color Alpha
0x34	Offset Color R
0x38	Offset Color G
0x3C	Offset Color B

Type 7 (Intensity)

0x00	Parameter Control Word
0x04	X
0x08	Y
0x0C	Z
0x10	U
0x14	V
0x18	Base Intensity
0x1C	Offset Intensity

Type 8 (Intensity, Compact UV)

0x00	Parameter Control Word
0x04	X
0x08	Y
0x0C	Z
0x10	U|V
0x14	Reserved
0x18	Base Intensity
0x1C	Offset Intensity

Type 9 (Non-Textured, Packed Color, with Two Volumes)

0x00	Parameter Control Word
0x04	X
0x08	Y
0x0C	Z
0x10	BaseColor 0
0x14	BaseColor 1
0x18	Reserved
0x1C	Reserved

Type 10 (Non-Textured, Intensity, with Two Volumes)

0x00	Parameter Control Word
0x04	X
0x08	Y
0x0C	Z
0x10	Base Intensity 0
0x14	Base Intensity 1
0x18	Reserved
0x1C	Reserved

Type 11 (Textured, Packed Color, with Two Volumes)

0x00	Parameter Control Word
0x04	X
0x08	Y
0x0C	Z
0x10	U0
0x14	V0
0x18	Base Color 0
0x1C	Offset Color 0
0x20	U1
0x24	V1
0x28	Base Color 1
0x2C	Offset Color 1
0x30	Reserved
0x34	Reserved
0x38	Reserved
0x3C	Reserved

Type 12 (Textured, Packed Color, 16-bit UV, with Two Volumes)

0x00	Parameter Control Word
0x04	X
0x08	Y
0x0C	Z
0x10	U0|V0
0x14	Reserved
0x18	Base Color 0
0x1C	Offset Color 0
0x20	U1|V1
0x24	Reserved
0x28	Base Color 1
0x2C	Offset Color 1
0x30	Reserved
0x34	Reserved
0x38	Reserved
0x3C	Reserved

Type 13 (Textured, Intensity, with Two Volumes)

0x00	Parameter Control Word
0x04	X
0x08	Y
0x0C	Z
0x10	U0
0x14	V0
0x18	Base Intensity 0
0x1C	Offset Intensity 0
0x20	U1
0x24	V1
0x28	Base Intensity 1
0x2C	Offset Intensity 1
0x30	Reserved
0x34	Reserved
0x38	Reserved
0x3C	Reserved

Type 14 (Textured, Intensity, 16-bit UV, with Two Volumes)

0x00	Parameter Control Word
0x04	X
0x08	Y
0x0C	Z
0x10	U0|V0
0x14	Reserved
0x18	Base Intensity 0
0x1C	Offset Intensity 0
0x20	U1|V1
0x24	Reserved
0x28	Base Intensity 1
0x2C	Offset Intensity 1
0x30	Reserved
0x34	Reserved
0x38	Reserved
0x3C	Reserved

Sprite type 0 (Type 15)

0x00	Parameter Control Word
0x04	AX
0x08	AY
0x0C	AZ
0x10	BX
0x14	BY
0x18	BZ
0x1C	CX
0x20	CY
0x24	CZ
0x28	DX
0x2C	DY
0x30	Reserved
0x34	Reserved
0x38	Reserved
0x3C	Reserved

Sprite type 1 (Type 16)

0x00	Parameter Control Word
0x04	AX
0x08	AY
0x0C	AZ
0x10	BX
0x14	BY
0x18	BZ
0x1C	CX
0x20	CY
0x24	CZ
0x28	DX
0x2C	DY
0x30	Reserved
0x34	AU|AV
0x38	BU|BV
0x3C	CU|CV

Shadow Volume (Type 17)

0x00	Parameter Control Word
0x04	AX
0x08	AY
0x0C	AZ
0x10	BX
0x14	BY
0x18	BZ
0x1C	CX
0x20	CY
0x24	CZ
0x28	Reserved
0x2C	Reserved
0x30	Reserved
0x34	Reserved
0x38	Reserved
0x3C	Reserved

The corresponding vertex data structures (18 types) are defined as follows:

```
typedef struct tagKMVERTEX_n (n=00…17)
{
       KMDWORD ParamControlWord;
       KMFLOAT fX;
       KMFLOAT fY;
       union{
          KMFLOAT fZ;
          KMFLOAT fInvW;
       } u;
       ……
} KMVERTEX_n, *PKMVERTEX_n;
```

The parameter types and names in a vertex data structure are listed below.
For the details of a vertex data structure, see the header file "kmvertex.h".

KMFLOAT fX, fY	……	Polygon vertex X- and Y-coordinates
KMFLOAT fZ, fInvW	……	Polygon vertex Z-coordinate
KMDWORD dwUV	……	Packed texture U- and V-coordinates
KMFLOAT fU, fV	……	Texture U- and V-coordinates
KMPACKEDARGB uBaseRGB	……	Packed base color RGB
KMFLOAT fBaseIntensity	……	Base intensity
KMFLOAT fBaseAlpha	……	Base color α
KMFLOAT fBaseRed	……	Base color red
KMFLOAT fBaseGreen	……	Base color green
KMFLOAT fBaseBlue	……	Base color blue
KMPACKEDARGB uOffsetRGB	……	Packed offset color RGB
KMFLOAT fOffsetIntensity	……	Offset intensity
KMFLOAT fOffsetAlpha	……	Offset color α
KMFLOAT fOffsetRed	……	Offset color red
KMFLOAT fOffsetGreen	……	Offset color green
KMFLOAT fOffsetBlue	……	Offset color blue
KMBUMPPARAM uBumpK1K2K3Q	……	Packed bump map parameter
KMFLOAT fU0m, fV0m	……	First texture U- and V-coordinates for two-parameter polygons
KMFLOAT fU1m, fV1m	……	Second texture U- and V-coordinates for two-parameter polygons
KMDWORD dwUV0m	……	First packed texture U- and V-coordinates for two-parameter polygons
KMDWORD dwUV1m	……	Second packed texture U- and V-coordinates for two-parameter polygons
KMPACKEDARGB uBaseRGB0m	……	First packed base color RGB for two-parameter polygons
KMPACKEDARGB uBaseRGB1m	……	Second packed base color RGB for two-parameter polygons
KMPACKEDARGB uOffsetRGB0m	……	First packed offset color RGB for two- parameter polygons
KMPACKEDARGB uOffsetRGB1m	……	Second packed offset color RGB for two-parameter polygons
KMFLOAT fBaseIntensity0m	……	First base intensity for two-parameter polygons
KMFLOAT fBaseIntensity1m	……	Second base intensity for two-parameter polygons
KMFLOAT fOffsetIntensity0m	……	First offset intensity for two-parameter polygons
KMFLOAT fOffsetIntensity1m	……	Second offset intensity for two-parameter polygons
KMFLOAT fAX, fAY, fAZ … fDZ	……	Sprite/modifier polygon vertex coordinates
KMDWORD dwUVA, dwUVB, dwUVC	……	Sprite packed texture U- and V-coordinates

3.6.4 Combining Parameters and Selecting Vertex Types

The type of a vertex to be used is determined according to the setting made for each VERTEXCONTEXT parameter. The following table lists the combinations of parameters corresponding to each vertex type.

ListType	ParamType	ColorType	UV Format	Shading Mode	Select Modifier	bUse Specular	VertexType	FaceColor / OffsetColor
KM_OPAQUE_POLYGON KM_TRANS_POLYGON KM_PUNCHTHROUGH_POLYGON	KM_POLYGON	KM_PACKEDCOLOR	Invalid	NO TEXTURE	KM_NOMODIFIER	FALSE	**0**	From Vertex Data
		KM_FLOATINGCOLOR	Invalid	NO TEXTURE	KM_NOMODIFIER	FALSE	**1**	From Vertex Data
		KM_INTENSITY	Invalid	NO TEXTURE	KM_NOMODIFIER	FALSE	**2**	From VERTEXCONTEXT
		KM_INTENSITY_PREV…	Invalid	NO TEXTURE	KM_NOMODIFIER	FALSE	**2**	Non
		KM_PACKEDCOLOR	Invalid	NO TEXTURE	KM_MODIFIER_A	FALSE	**9**	From Vertex Data
		KM_INTENSITY	Invalid	NO TEXTURE	KM_MODIFIER_A	FALSE	**10**	From VERTEXCONTEXT
		KM_INTENSITY_PREV…	Invalid	NO TEXTURE	KM_MODIFIER_A	FALSE	**10**	Non
		KM_PACKEDCOLOR	KM_32BITUV	USE TEXTURE	KM_NOMODIFIER	*	**3**	From Vertex Data
		KM_PACKEDCOLOR	KM_16BITUV	USE TEXTURE	KM_NOMODIFIER	*	**4**	From Vertex Data
		KM_FLOATINGCOLOR	KM_32BITUV	USE TEXTURE	KM_NOMODIFIER	*	**5**	From Vertex Data
		KM_FLOATINGCOLOR	KM_16BITUV	USE TEXTURE	KM_NOMODIFIER	*	**6**	From Vertex Data
		KM_INTENSITY	KM_32BITUV	USE TEXTURE	KM_NOMODIFIER	FALSE	**7**	From VERTEXCONTEXT
		KM_INTENSITY	KM_32BITUV	USE TEXTURE	KM_NOMODIFIER	TRUE	**7**	From VERTEXCONTEXT
		KM_INTENSITY	KM_16BITUV	USE TEXTURE	KM_NOMODIFIER	FALSE	**8**	From VERTEXCONTEXT
		KM_INTENSITY	KM_16BITUV	USE TEXTURE	KM_NOMODIFIER	TRUE	**8**	From VERTEXCONTEXT
		KM_INTENSITY_PREV…	KM_32BITUV	USE TEXTURE	KM_NOMODIFIER	*	**7**	Non
		KM_INTENSITY_PREV…	KM_16BITUV	USE TEXTURE	KM_NOMODIFIER	*	**8**	Non
		KM_PACKEDCOLOR	KM_32BITUV	USE TEXTURE	KM_MODIFIER_A	*	**11**	From Vertex Data
		KM_PACKEDCOLOR	KM_16BITUV	USE TEXTURE	KM_MODIFIER_A	*	**12**	From Vertex Data
		KM_INTENSITY	KM_32BITUV	USE TEXTURE	KM_MODIFIER_A	*	**13**	From VERTEXCONTEXT
		KM_INTENSITY	KM_16BITUV	USE TEXTURE	KM_MODIFIER_A	*	**14**	From VERTEXCONTEXT
		KM_INTENSITY_PREV…	KM_32BITUV	USE TEXTURE	KM_MODIFIER_A	*	**13**	Non
		KM_INTENSITY_PREV…	KM_16BITUV	USE TEXTURE	KM_MODIFIER_A	*	**14**	Non
	KM_SPRITE	KM_PACKEDCOLOR	Invalid	NO TEXTURE	KM_NOMODIFIER	FALSE	**15**	From VERTEXCONTEXT
		KM_PACKEDCOLOR	KM_16BITUV	USE TEXTURE	KM_NOMODIFIER	*	**16**	From VERTEXCONTEXT
KM_OPAQUE_MODIFIER KM_TRANS_MODIFIER	KM_MODIFIER VOLUME	Invalid	Invalid	Invalid	Invalid	Invalid	**17**	Non

* Each shading level groups the same parameter type so as to make it easy to read the table.
* "NO TEXTURE" in the ShadingMode column means that either KM_NOTEXTUREFLAT or KM_NOTEXTUREGOURAUD is selected. Similarly, "USE TEXTURE" means that either KM_TEXTUREFLAT or KM_TEXTUREGOURAUD is selected.
* The asterisk means that any value can be specified.
* The FaceColor/OffsetColor column indicates the data that determines the FaceColor/OffsetColor of the polygons. "From Vertex Data" means that the color data in the data for each vertex is to be used (it can be specified separately for individual vertices). "From VERTEXCONTEXT" means that the color data in VERTEXCONTEXT is to be used (it can specified separately for individual strips).

3.6.5 Setting a Modifier Volume

The **modifier volume** is polygon data for determining the area that is to be used for adding a shadow to ordinary polygons "three-dimensionally." The modifier volume is not actually drawn on the screen. A polygon in a scene can be divided into two sections, that is, the inside and outside of the modifier volume. A color and texture can be specified for each section separately. This makes effects such as a shadow and a spotlight.
A polygon to be divided into two, using the modifier volume, is called a two-parameter polygon. A two-parameter polygon has two texture/shading parameters. An object to be influenced by the modifier volume shall be registered with KAMUI using two-parameter polygons.

The use of two-parameter polygons results in an increased amount of data. This problem can be avoided by using the **cheap (simple) shadow mode**, which can still be used for representing shadows. This mode represents the shadow of a polygon by decreasing the luminance of a polygon portion approaching a modifier volume. This is set using the `kmSetCheapShadowMode` function. Once this function has been executed, all modifier volumes enter the cheap shadow mode. The cheap shadow mode cannot be used together with two-parameter polygons in one scene.

This section explains how to set a modifier volume.

Setting a two-parameter volume

(Polygon influenced by modifier volume)

- Setting of `VERTEXCONTEXT`
  Set `KM_MODIFIER_A` as the `SelectModifier` member. Set the two parameters (CONTEXT) for `kmSetVertexRenderState` for area 0 (explained below) and `kmSetModifierRenderState` for area 1.
- Format of vertex data
  Define the vertex data with one of `KMVERTEX_09` to `KMVERTEX_14`.
- When using cheap shadow mode
  The cheap shadow mode cannot be used simultaneously with two-parameter volumes in the same scene. The cheap shadow mode is turned on and off by issuing `kmSetCheapShadowMode` before `kmProcessVertexRenderState` for the `VERTEXCONTEXT` of polygons to be influenced by the mode. When the cheap shadow mode is to be used, therefore, `kmSetCheapShadowMode` must be executed at the beginning of the scene. After `kmSetCheapShadowMode` is executed, `kmProcessVertexRenderState` is executed by setting `KM_MODIFIER_A` in the `SelectModifier` member of `VERTEXCONTEXT` to be used for the polygon, in much the same way as a two-parameter volume. In this case, ordinary one-parameter polygons are used as vertex data. The cheap shadow mode does not operate if two-parameter polygons are used. Once the cheap shadow mode is set, `kmProcessVertexRenderState` need not be executed after `kmSetCheapShadowMode`, if only the intensity of a shadow is to be changed.

Setting of modifier volume

There are two types of modifier volumes. The first type is the opaque modifier volume that is effective only for opaque polygons. The second type is the translucent modifier volume that is effective only for translucent/transparent polygons. Each modifier type is further classified as either inclusion or exclusion volume. In the inclusion volume, a polygon portion inside the volume is defined as area 1, while in the exclusion volume, a polygon portion inside the volume is defined as area 0. The exclusion volume is used to generate area 0 inside the portion that was defined as area 1 in the inclusion volume.

- A modifier volume is registered to dedicated VertexBuffer.
  Secure buffer for an opaque modifier and translucent modifier by `kmCreateVertexBuffer`.
- Setting of VERTEXCONTEXT
  The polygons constituting a modifier volume must be classified into three types by using VERTEXCONTEXT: the first, the last, and others.
  Specify `KM_MODIFIER_INCLUDE_FIRST_POLY` or `KM_MODIFIER_EXCLUDE_FIRST_POLY` as the `ModifierInstruction` member for the first polygon. For the last polygon, specify `KM_MODIFIER_INCLUDE_LAST_POLY` or `KM_MODIFIER_EXCLUDE_LAST_POLY` as the `ModifierInstruction` member. Specify `KM_MODIFIER_NORMAL_POLY` as the `ModifierInstruction` member for the other polygons. In any case, specify `KM_MODIFIERVOLUME` as the `ParamType` member and `KM_OPAQUE_MODIFIER` or `KM_TRANS_MODIFIER` as `ListType`. ARC1 sets the screen coordinates at which the modifier volume is validated for each member of `fBoundingBoxXmin`, `fBoundingBoxYmin`, `fBoundingBoxXmax`, and `fBoundingBoxYmax` (this setting is not necessary with CLX1/2).
- Format of vertex data
  Define vertex data by `KMVERTEX_17`.

3.6.6 Pointer to Buffer for Registering Vertex Data

KMVERTEXBUFFDESC

KAMUI uses this structure to hold **pointers to KAMUI's internal areas** where different types of information related to the vertex data buffer are saved. It is prepared for reference by the `kmStartVertexStrip` and `kmSetVertex` macros in the buffer mode. If illegal data is written into an area indicated by any of these pointers, KAMUI cannot operate normally. The application program should avoid using the values in this structure as much as possible.

KAMUI sets values (pointers) in each member of the structure when `kmSetSystemConfiguration` (or `kmCreateVertexBuffer`, described later) is called. KAMUI updates the contents of the areas indicated by these pointers when `kmRender`, `kmRenderDirect`, or `kmRenderTexture` is executed.

```
typedef struct _tagKMVERTEXBUFFDESC
{
PKMDWORD                    *pCurrentPtr;
```

```
PKMDWORD                    pGlobalParam;
PKMCURRENTLISTSTATE         pCurrentListState;
PPKMVERTEXBUFFERPOINTER     ppBuffer;
KMDWORD                     reserved0;
KMDWORD                     reserved1;
KMDWORD                     reserved2;
KMDWORD                     reserved3;
} KMVERTEXBUFFDESC,*PKMVERTEXBUFFDESC;
```

pCurrentPtr

KAMUI sets this area with the start address of its internal area where the current vertex data buffer pointer is held. (See the description of the `CurrentVertex pointer array`, below.)

pGlobalParam

In this area, KAMUI sets the start address of the internal area used to generate hardware control parameters (global parameters) within KAMUI.

pCurrentListState

In this area, KAMUI sets the start address of the internal area where those types of information related to the vertex data buffer are held. (See the description of the `Current List status`, below.)

ppBuffer

In this area, KAMUI sets the start address of the internal pointer area that indicates a pointer to the first vertex data buffer. (See the description of the `Vertex buffer pointer structure`, below.)

reserved0 to reserved3

These areas are reserved for future use. They are used to align the structure with a 32-byte boundary.

[Reference]

The following functions can be used to update the contents of KMVERTEXBUFFDESC.

kmCreateVertexBuffer, kmSetSystemConfiguration, kmStartVertexStrip, kmSetVertex, kmSetUserClipping, kmRender, kmRenderTexture, kmChangeVertexOffset, kmChangeVertexPCW

CurrentVertex pointer array

This is KAMUI's internal variable in which it stores the current vertex data buffer pointer.

KAMUI divides the system memory area prepared by the application program into five sections (actually, ten sections because of double buffering). These areas are used as vertex data buffers, one for each vertex type. This pointer indicates the location in each buffer where KAMUI will store polygon data the next time KAMUI is called by `kmSetVertex`. The buffers are initialized by `kmCreateVertexBuffer` and `kmRender`. (Put another way, the pointer indicates the start address of each buffer in a new bank.) Each time the buffer is loaded with data by calling `kmSetVertex` or `kmSetUserClipping`, KAMUI increments the relevant pointer.

Current List status

This area is used for holding the information related to the vertex data buffer. It is a KAMUI internal variable. KAMUI defines it as follows:

```
typedef struct _tagKMCURRENTLISTSTATE
{
KMDWORD       ListType;
KMDWORD       GlobalParamSize;
KMDWORD       reserved0;
KMDWORD       VertexBank;
} KMCURRENTLISTSTATE,*PKMCURRENTLISTSTATE;
```

ListType

Into this area, KAMUI loads a polygon list type used the next time `kmSetVertex` is called. The `ListType` values have the following meanings:

```
0       : Opaque polygon
1       : Opaque modifier polygon
2       : Translucent polygon
3       : Translucent modifier polygon
4       : Punchthrough polygon
```

GlobalParamSize

Into this area, KAMUI loads the size of a hardware control parameter (global parameter) transferred the next time `kmStartVertexStrip` is called.

reserved0

This area is reserved for future use. Its contents are undefined.

VertexBank

Into this are, KAMUI loads a bank number for a vertex data buffer into which vertex data is to be stored the next time `kmStartVertexStrip` or `kmSetVertex` is called.

Vertex buffer pointer structure

This is an area for storing pointers to the beginning of each vertex data buffer. It is a KAMUI internal variable. KAMUI defines it as follows:

```
typedef struct _tagKMVERTEXBUFFERPOINTER
{
PKMDWORD       pOpaquePolygonBuffer;
PKMDWORD       pOpaqueModifierBuffer;
PKMDWORD       pTransPolygonBuffer;
PKMDWORD       pTransModifierBuffer;
PKMDWORD       pPunchThroughPolygonBuffer;
}
KMVERTEXBUFFERPOINTER,*PKMVERTEXBUFFERPOINTER,**PPKMVERTEXBUFFER
POINTER;
```

The pointers to two areas (for banks 0 and 1) are allocated consecutively, as follows:

```
PKMVERTEXBUFFERPOINTER VertexBuff[2];
```

KAMUI loads `ppBuffer` of `KMVERTEXBUFFDESC` with the start address (`VertexBuff`) of these consecutive pointer areas. Its contents are fixed when `kmCreateVertexBuffer` is called.

3.7 REGISTERING VERTEX DATA IN THE BUFFER MODE

In the buffer mode, the following functions are used to register vertex data and to direct the start of rendering.

The buffer mode cannot be used together with the direct mode. The mode to be used is specified using `kmSetSystemConfiguration`.

3.7.1 Allocating a Vertex Data Registration Buffer

```
KMSTATUS kmCreateVertexBuffer(PKMVERTEXBUFFDESC pBufferDesc,
                        KMINT32 OpaquePolygonBuffer,
                        KMINT32 OpaqueModifierBuffer,
                        KMINT32 TransPolygonBuffer,
                        KMINT32 TransModifierBuffer)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	◆

Explanation:
This function allocates vertex data and native data buffers as double buffers in the following locations:

Vertex data buffer	...	Main memory
Native data buffer	...	Frame buffer memory

The size of the native buffer is calculated from the size of each list specified using an argument.

In the buffer mode, `kmStartVertexStrip` or `kmSetVertex` is used to register vertex data. Rendering is started after `kmRender` registers data for one scene in each vertex data buffer and sends the data for all scene together to hardware. It is impossible to use `kmCreateVertexBuffer` and `kmCreateTAbuffer` simultaneously in one application program.

Caution This function is provided to maintain compatibility with the conventional specification (Ver l.28). Use `kmSetSystemConfiguration` whenever possible. This function cannot specify the capacity of the punchthrough polygon buffer, which will be supported by CLX2.
One application cannot use `kmCreateVertexBuffer` and `kmCreateTABuffer` simultaneously.

Arguments:
`pBufferDesc` (I/O)
Input a pointer to a vertex buffer descriptor of `PKMVERTEXBUFFDESC` type. The first addresses of the four buffers allocated here are set to the members of the vertex buffer descriptor.

`OpaquePolygonBuffer` (input)
Size of the vertex data buffer in which the data of opaque polygons is stored
Given as the number of `DWORD`s (number of bytes/4).

`OpaqueModifierBuffer` (input)
Size of the vertex data buffer in which the data of opaque modifier volumes is stored
Given as the number of `DWORD`s (number of bytes/4).

`TransPolygonBuffer` (input)
Size of the vertex data buffer in which the data of translucent/transparent polygons is stored
Given as the number of `DWORD`s (number of bytes/4).

`TransModifierBuffer` (input)
Size of the vertex data buffer in which the data of translucent/transparent modifier volumes is stored
Given as the number of `DWORD`s (number of bytes/4).

Return values:

`KMSTATUS_SUCCESS`	Vertex data buffers successfully allocated
`KMSTATUS_NOT_ENOUGH_MEMORY`	Insufficient memory

3.7.2 Releasing Vertex Data Registration Buffers

```
KMSTATUS kmDiscardVertexBuffer(PKMVERTEXBUFFDESC pBufferDesc)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function releases the four types of vertex data buffers (double buffer) for vertex data registration that have been allocated by `kmCreateVertexBuffer`. It also releases the native data buffer that was allocated in the frame buffer memory at the same time.

Argument:
`pBufferDesc` (I/O)
This argument inputs a pointer to a vertex data buffer descriptor of `PKMVERTEXBUFFDESC` type.

Return value:
`KMSTATUS_SUCCESS` Vertex data buffer released successfully

3.7.3 Starting Registering Vertex Data Strips

```
KMSTATUS kmStartVertexStrip( PKMVERTEXBUFFDESC pBufferDesc)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function directs KAMUI to begin registering vertex data strips. KAMUI writes `VERTEXCONTEXT` information set by `kmSetVertexRenderState` and `kmSetModifierRenderState` to one of the five different vertex data buffers. The vertex data buffer to be used is determined from the `ListType` member of `KMVERTEXCONTEXT`.

Arguments:
`pBufferDesc` (input)
This is a pointer to a vertex buffer descriptor of `PKMVERTEXBUFFDESC` type.

Return values:

`KMSTATUS_SUCCESS`	`VERTEXCONTEXT` information successfully written
`KMSTATUS_NOT_ENOUGH_MEMORY`	Vertex data buffer capacity insufficient

Caution If the `VERTEXCONTEXT` information written in this argument does not match the type of a vertex parameter written later by `kmSetVertex`, normal operation is not guaranteed.

<This function is required to run quickly, so it is defined as a macro to be developed in-line.>

To enable use of the macro, code the following:

```
#define _KM_USE_VERTEX_MACRO_
#include <kamui.h>
#include <kamuix.h>
```

3.7.4 Writing Vertex Data in a Buffer

```
KMSTATUS kmSetVertex(PKMVERTEXBUFFDESC pBufferDesc,
                     PVOID pVertex,
                     KMVERTEXTYPE VertexType,
                     KMINT32 StructSize)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function writes vertex data indicated by `pVertex` to a vertex data list specified by `ListType` of `KMVERTEXCONTEXT`, already registered with the system. If `Parameter Control Word` of the vertex data at the end of the strip is not `KM_VERTEXPARAM_ENDOFSTRIP`, the operation is not guaranteed.

Arguments:
`pBufferDesc` (input)
Pointer to a vertex data buffer descriptor of `PKMVERTEXBUFFDESC` type

`pVertex` (input)
Pointer to the vertex data structure

`VertexType` (input)
Vertex data type
Specify one of the following:

```
KM_VERTEXTYPE_00                    // Vertex data Type 0
KM_VERTEXTYPE_01                    // Vertex data Type 1
KM_VERTEXTYPE_02                    // Vertex data Type 2
KM_VERTEXTYPE_03                    // Vertex data Type 3
KM_VERTEXTYPE_04                    // Vertex data Type 4
KM_VERTEXTYPE_05                    // Vertex data Type 5
KM_VERTEXTYPE_06                    // Vertex data Type 6
KM_VERTEXTYPE_07                    // Vertex data Type 7
KM_VERTEXTYPE_08                    // Vertex data Type 8
KM_VERTEXTYPE_09                    // Vertex data Type 9
KM_VERTEXTYPE_10                    // Vertex data Type 10
KM_VERTEXTYPE_11                    // Vertex data Type 11
KM_VERTEXTYPE_12                    // Vertex data Type 12
KM_VERTEXTYPE_13                    // Vertex data Type 13
KM_VERTEXTYPE_14                    // Vertex data Type 14
KM_VERTEXTYPE_15                    // Vertex data Type 15
KM_VERTEXTYPE_16                    // Vertex data Type 16
KM_VERTEXTYPE_17                    // Vertex data Type 17
```

`StructSize` (input)
Size of the vertex data structure. Specify a size according to the selected vertex data type in the `sizeof(KMVERTEX_01)` format.

Caution If the format of the vertex parameters written here does not match the format of global parameters specified by preceding `kmSetVertexRenderState`, the operation may be adversely affected.

Return values:

`KMSTATUS_SUCCESS`	Vertex data successfully written
`KMSTATUS_NOT_ENOUGH_MEMORY`	Insufficient vertex data buffer space

<**This function is defined as a macro of inline expansion because high-speed execution is demanded.**>

To use the macro, code the following:

```
#define _KM_USE_VERTEX_MACRO_
#include <kamui.h>
#include <kamuix.h>
```

3.7.5 Setting a User Clipping Area (for Buffer Mode)

```
KMSTATUS kmSetUserClipping(
                            PKMVERTEXBUFFDESC pBufferDesc,
                            KMLISTTYPE ListType,
                            KMINT32 Xmin, KMINT32 Ymin,
                            KMINT32 Xmax, KMINT32 Ymax)
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	♥

Explanation:
This function specifies a user clipping area. The user clipping area specified using this function is effective for polygons for which `KM_USERCLIP_INSIDE` or `KM_USERCLIP_OUTSIDE` is specified in the `UserClipMode` member of the `KMVERTEXCONTEXT` structure.

Arguments:
`pBufferDesc` (input)
Input a pointer to a vertex data buffer descriptor of `PKMVERTEXBUFFDESC` type.

`ListType` (input)
Specify a vertex data list in which a user clipping area is specified.
One of the following values can be specified:

```
KMLISTTYPE
KM_OPAQUE_POLYGON        = 0   // Opaque polygon
KM_OPAQUE_MODIFIER       = 1   // Opaque modifier volume
KM_TRANS_POLYGON         = 2   // Translucent/transparent
                                  polygon
KM_TRANS_MODIFIER        = 3   // Translucent/transparent
                                  modifier volume
KM_PUNCHTHROUGH_POLYGON  = 4   // Punchthrough polygon (for only
                                  CLX2)
```

`Xmin, Ymin, Xmax, Ymax` (input)

Specify the coordinates of the top left and bottom right corners of the user clip area. Specify the values in tiles (1 = 32 pixels).

Just the six low-order bits of `Xmin` and `Xmax` are valid. As for `Ymin` and `Ymax`, just the four low-order bits are valid.

Caution **It is impossible to clip part of a vertex strip when it is being registered. Specifically, do not issue `kmSetUserClipping` to clip part of a vertex strip for which registration has been started using `kmStartVertexStrip` before `ENDOFSTRIP` is specified by `kmSetVertex`.**

Return value:

`KMSTATUS_SUCCESS`	Set successfully

3.7.6 Notifying the End of Vertex Data Writing

```
KMSTATUS kmRender(VOID)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function notifies KAMUI that all vertex data of a single scene has been written. The renderer begins rendering for a back buffer after data expansion has been completed.

Argument:
None

Return value:

`KMSTATUS_SUCCESS`	Notified successfully

3.7.7 Rendering into the Texture Memory

```
KMSTATUS kmRenderTexture(PKMSURFACEDESC pTextureSurface)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function notifies KAMUI that all vertex data of a single scene has been written. The renderer begins rendering for a texture surface specified after vertex data expansion is completed.
Usually, when 640 x 480 rendering is performed, a 1,024 x 512 texture surface is specified as the rendering destination. As a result, the result of rendering is written to the UV coordinates (0.0f,0.0f)-(0.625f,0.9375f) of this texture. (The lowest line of the texture serves as the highest line of the screen.) If a texture surface less than the screen resolution is specified as the rendering destination surface, the upper-left part of the screen is rendered to the texture. For example, if rendering is performed to a 256 x 256 texture surface where the screen resolution is 640 x 480, the upper-left part (0,0)-(255,255) of the screen is written to the texture.

Argument:
`pTextureSurface` (output)
Texture of which rendering result is stored

Caution When using this function, make sure that BPP of the frame buffer is the same as BPP of the texture at the rendering destination by using `kmSetDisplayMode`. Otherwise, the performance will drop. Note that the texture surface specified here must be of RECTANGLE or STRIDE type.

Return values:

`KMSTATUS_SUCCESS`	Notified successfully
`KMSTATUS_INVALID_TEXTURE`	Invalid texture specified

3.7.8 Specifying a Modifier Volume List

```
KMSTATUS kmUseAnotherModifier(
                    KMLISTTYPE kmModifierListType)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function uses the modifier volume list specified by input parameter `kmModifierListType` as another modifier volume list.
This function only rewrites the pointer to OPAQUE-MODIFIER and TRANS-MODIFIER of the region array in native data. If KM_OPAQUE_MODIFIER is specified and TRANS-MODIFIER object data is registered, the data of OPAQUE-MODIFIER is overwritten (the converse holds true).

Argument:
`kmModifierListType` (input)
This argument specifies the usage of the modifier volume list. Specify it in either of the following ways:
`KM_OPAQUE_MODIFIER`
Also uses Opaque Modifier as Trans Modifier.
`KM_TRANS_MODIFIER`
Also uses Trans Modifier as Opaque Modifier.

Return values:

`KMSTATUS_SUCCESS`	Set successfully
`KMSTATUS_INVALID_LIST_TYPE`	Setting failed

3.7.9 Obtaining Current Writing Position of VertexBuffer

```
KMDWORD kmGetCurrentVertexOffset ( KMLISTTYPE ListType )
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function returns the number of vertex data items currently held in the specified list type vertex data buffer, using a 32-bit word number.
By using the value obtained by this function in combination with `kmChangeVertexOffset` or `kmChangeVertexPCW`, the value of `VertexBuffer` can be changed.

Argument:
`ListType` (input)

```
KM_OPAQUE_POLYGON         = 0   // Opaque polygon
KM_OPAQUE_MODIFIER        = 1   // Opaque modifier volume
KM_TRANS_POLYGON          = 2   // Translucent/transparent
                                   polygon
KM_TRANS_MODIFIER         = 3   // Translucent/transparent
                                   modifier volume
KM_PUNCHTHROUGH_POLYGON   = 4   // Punchthrough polygon (for only
                                   CLX2)
```

Return value:
The current writing position in vertex data of the specified list type is returned. The value is a 32-bit word offset from the beginning of VertexBuffer of the specified list type (i.e., currently used capacity of the vertex data buffer).

3.7.10 Changing Current Writing Position of VertexBuffer

```
KMSTATUS kmChangeVertexOffset (
                            KMLISTTYPE ListType,
                            KMDWORD VertexOffset )
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function changes the current writing position of the vertex data buffer of the specified list type.

Arguments:
`ListType` (input)

```
KM_OPAQUE_POLYGON        = 0   // Opaque polygon
KM_OPAQUE_MODIFIER       = 1   // Opaque modifier volume
KM_TRANS_POLYGON         = 2   // Translucent/transparent
                                  polygon
KM_TRANS_MODIFIER        = 3   // Translucent/transparent
                                  modifier volume
KM_PUNCHTHROUGH_POLYGON  = 4   // Punchthrough polygon (for only
                                  CLX2)
```

`VertextOffset` (input)
This argument specifies the writing position of the specified list type. The value obtained by `kmGetCurrentVertexOffset` is used.

Return values:

`KMSTATUS_SUCCESS`	Set successfully
`KMSTATUS_INVALID_VALUE`	A value that must not be specified is specified

3.7.11 Direct Rewriting of Vertex Control Word

```
KMSTATUS kmChangeVertexPCW (
                            KMLISTTYPE ListType,
                            KMDWORD VertexPCW,
                            KMDWORD IncPtr)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function writes a value specified by VertexPCW to the current writing position of the vertex data buffer of the specified list type. The pointer of the vertex data buffer is incremented by IncPtr. It is used to abort a strip after strip writing of a triangle is finished.
Exercise care when using this function because it may destroy matching in the vertex data buffer. This function must not be used when direct mode is used to write vertex data.

Arguments:
`ListType` (input)

```
KM_OPAQUE_POLYGON        = 0   // Opaque polygon
KM_OPAQUE_MODIFIER       = 1   // Opaque modifier volume
KM_TRANS_POLYGON         = 2   // Translucent/transparent
                                  polygon
KM_TRANS_MODIFIER        = 3   // Translucent/transparent
                                  modifier volume
KM_PUNCHTHROUGH_POLYGON  = 4   // Punchthrough polygon (for only
                                  CLX2)
```

`VertexPCW` (input)
This argument specifies a type of a control word.

```
KM_VERTEXPARAM_NORMAL      = 0xE0000000  //  Normal vertex data
KM_VERTEXPARAM_ENDOFSTRIP  = 0xF0000000  //  Vertex data at the
                                             end of the strip
```

`IncPtr` (input)
This argument specifies how much the pointer of the vertex data buffer advances after the parameter specified by `VertexPCW` has been written. Usually, specify the size of the vertex data structure because it is used like `sizeof(...)/4`.

Return value:

`KMSTATUS_SUCCESS`	Set successfully

3.7.12 Flushing VertexBuffer

```
KMSTATUS kmFlushVertexBuffer (KMLISTTYPE ListType)
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	√	♥

Note COSMOS+ARC1 only supports Opaque.

Explanation:
This function flushes `VertexBuffer` of the specified list type. KAMUI begins transferring the contents of a vertex data buffer of a specified list type to the hardware in DMA mode. This function is used to save the capacity of the vertex data buffer used for `Opaque` list by using the fact that the list is sent from `Opaque` with the `CLX1/2` and by sending the list little by little. This function must not be used with a direct mode function. If the previous `DMA` transfer has not been completed when this function is called, waiting for the completion of `DMA` takes place. Make sure that waiting does not occur when using this function.

Argument:
`ListType` (input)

```
KM_OPAQUE_POLYGON         = 0   // Opaque polygon
KM_OPAQUE_MODIFIER        = 1   // Opaque modifier volume
KM_TRANS_POLYGON          = 2   // Translucent/transparent
                                   polygon
KM_TRANS_MODIFIER         = 3   // Translucent/transparent
                                   modifier volume
KM_PUNCHTHROUGH_POLYGON   = 4   // Punchthrough polygon (for only
                                   CLX2)
```

Return value:

`KMSTATUS_SUCCESS`	Set successfully

3.8 REGISTERING VERTEX DATA IN THE DIRECT MODE

In the direct mode, the following functions are used to register vertex data and to direct the start of rendering.

The direct mode cannot be used together with the buffer mode. Which mode to use is specified using `kmSetSystemConfiguration`.

3.8.1 Allocating the Native Data Buffer

```
KMSTATUS kmCreateTABuffer(
                    PKMVERTEXBUFFDESC pBufferDesc,
                    KMINT32 TABuffer)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	-

Explanation:
This function allocates a native data buffer in frame buffer memory. In the direct mode, vertex data is registered by sending it directly to the tiling accelerator, using the following functions.

`kmStartVertexStripDirect`
`kmSetVertexDirect`
`kmSetUserClippingDirect`
`kmSetEndOfListDirect`
`kmRenderDirect`

Caution It is impossible to use `kmCreateVertexBuffer` and `kmCreateTABuffer` simultaneously in one application program. Do not allocate a texture surface before calling this function. Holly (CLX1/2) cannot use the function, because it uses a different method to generate a native data buffer. `kmSetSystemConfiguration` should be used for Holly (CLX1/2).

Arguments:
`pBufferDesc` (input)
This is a pointer to a vertex buffer descriptor of `PKMVERTEXBUFFDESC` type.

`TABuffer` (input)
Native data buffer size (in bytes).

Return values:

`KMSTATUS_SUCCESS`	Native data buffer successfully allocated
`KMSTATUS_NOT_ENOUGH_MEMORY`	Insufficient memory

3.8.2 Starting the Registration of Vertex Data Strips

```
KMSTATUS kmStartVertexStripDirect(VOID)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function begins the registration of vertex data strips. It writes rendering parameters (created by `kmProcessVertexRenderState` and registered with the system by `kmSetVertexRenderState`) directly to the tiling accelerator. It is necessary to simultaneously supply each type of vertex data for the same scene to the tiling accelerator.
Example: After opaque and translucent polygons are registered in the stated order, it is impossible to register the opaque polygon again.

Caution If the rendering parameter information written by this function does not match the type of a vertex parameter written later by `kmSetVertex`, normal operation is not guaranteed. Once the registration of vertices in a list is completed, it is necessary to notify the tiling accelerator of the completion of registration using `kmSetEndOfListDirect` before starting registration in another list.

Argument:
None

Return value:

`KMSTATUS_SUCCESS`	Rendering parameters successfully written

3.8.3 Directly Writing Vertex Data

```
KMSTATUS kmSetVertexDirect(PVOID pVertex,
                           KMVERTEXTYPE VertexType,
                           KMINT32 StructSize)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
The function directly writes the vertex data specified by `pVertex` into the tiling accelerator. It is necessary to simultaneously supply each type of vertex data for the same scene to the tiling accelerator.

Arguments:
`pVertex` (input)
Pointer to the vertex data structure

`VertexType` (input)
Vertex data type.
Specify one of the following:

```
KM_VERTEXTYPE_00                    // Vertex data Type 0
KM_VERTEXTYPE_01                    // Vertex data Type 1
KM_VERTEXTYPE_02                    // Vertex data Type 2
KM_VERTEXTYPE_03                    // Vertex data Type 3
KM_VERTEXTYPE_04                    // Vertex data Type 4
KM_VERTEXTYPE_05                    // Vertex data Type 5
KM_VERTEXTYPE_06                    // Vertex data Type 6
KM_VERTEXTYPE_07                    // Vertex data Type 7
KM_VERTEXTYPE_08                    // Vertex data Type 8
KM_VERTEXTYPE_09                    // Vertex data Type 9
KM_VERTEXTYPE_10                    // Vertex data Type 10
KM_VERTEXTYPE_11                    // Vertex data Type 11
KM_VERTEXTYPE_12                    // Vertex data Type 12
KM_VERTEXTYPE_13                    // Vertex data Type 13
KM_VERTEXTYPE_14                    // Vertex data Type 14
KM_VERTEXTYPE_15                    // Vertex data Type 15
KM_VERTEXTYPE_16                    // Vertex data Type 16
KM_VERTEXTYPE_17                    // Vertex data Type 17
```

`StructSize` (input)
Vertex data type size. Specify a size according to the selected vertex data type in the `sizeof(KMVERTEX_01)` format.

Caution **If Parameter Control Word of the vertex data at the end of the strip is not `KM_VERTEXPARAM_ENDOFSTRIP`, the operation is not guaranteed. If the format of the vertex parameters written here does not match the format of rendering parameters specified by preceding `kmSetVertexRenderState`, the operation may be adversely affected.**

Return value:

`KMSTATUS_SUCCESS`	Vertex data successfully written

3.8.4 Setting a User Clipping Area (for Direct Mode)

```
KMSTATUS kmSetUserClippingDirect(
                            KMINT32 Xmin, KMINT32 Ymin,
                            KMINT32 Xmax, KMINT32 Ymax)
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	√

Explanation:
This function specifies a user clipping area. The user clipping area specified using this function is effective for polygons for which `KM_USERCLIP_INSIDE` or `KM_USERCLIP_OUTSIDE` is specified in the `UserClipMode` member of the `KMVERTEXCONTEXT` structure.
For the difference between this function and `kmSetUserClipping`, see Section 3.6.

Arguments:
`Xmin, Ymin, Xmax, Ymax` (input)
Specify the coordinates of the upper-left and lower-right corners of the user clip area. Specify the values in tiles (1 = 32 pixels).
Just the six low-order bits of `Xmin` and `Xmax` are valid. As for `Ymin` and `Ymax`, just the four low-order bits are valid.

Return value:
`KMSTATUS_SUCCESS` Set successfully

Caution `kmSetUserClippingDirect` must not be issued between `kmStartVertexStripDirect` and `kmSetVertexDirect`.

3.8.5 Notifying the End of Vertex Registration

```
KMSTATUS kmSetEndOfListDirect(VOID)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function directly notifies the end of a vertex data list of a specific type to the tiling accelerator. Vertex data must be classified by the type of vertex and input to the tiling accelerator in the same scene. The end of the type of the currently registered vertex is reported by issuing `kmSetEndOfListDirect` at the end of each of five types of vertices.

Argument:
None

Caution Do not call this function if `kmSetVertex` is used for vertex data registration.

Return value:

`KMSTATUS_SUCCESS`	Notified successfully

3.8.6 Notifying the End of Vertex Data Writing

```
KMSTATUS kmRenderDirect(VOID)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function notifies KAMUI that all vertex data of a single scene has been written. The renderer begins rendering for a back buffer after vertex data expansion has been completed.

Argument:
None

Return value:

`KMSTATUS_SUCCESS`	Notified successfully

3.8.7 Rendering into the Texture Memory

```
KMSTATUS kmRenderTextureDirect(
                    PKMSURFACEDESC pTextureSurface)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function notifies KAMUI that all vertex data of a single scene has been written. The renderer begins rendering for a texture surface specified after vertex data expansion has been completed.
Usually, when 640 x 480 rendering is performed, a 1,024 x 512 texture surface is specified as the rendering destination. As a result, the result of rendering is written to the UV coordinates (0.0f,0.0f)-(0.625f,0.9375f) of this texture. (The lowest line of the texture serves as the highest line of the screen.) If a texture surface less than the screen resolution is specified as the rendering destination surface, the upper-left part of the screen is rendered to the texture. For example, if rendering is performed to a 256 x 256 texture surface where the screen resolution is 640 x 480, the upper-left part (0,0)-(255,255) of the screen is written to the texture.

Argument:
`pTextureSurface` (output)
Texture of which rendering result is stored

Caution When using this function, make sure that BPP of the frame buffer is the same as BPP of the texture at the rendering destination by using `kmSetDisplayMode`. Otherwise, the performance will drop. Note that the texture surface specified here must be of RECTANGLE/STRIDE type.

Return values:

`KMSTATUS_SUCCESS`	Notified successfully
`KMSTATUS_INVALID_TEXTURE`	Invalid texture specified

3.9 CALLBACK FUNCTIONS AND CALLBACK AUXILIARY FUNCTIONS

KAMUI can specify callback functions that modify rendering conditions and do other operation at a particular timing. The functions are called at particular events (end of rendering, for example), irrespective of the normal processing flow.

3.9.1 Specifying a Rendering End Callback Function

```
KMSTATUS kmSetEORCallback(PKMCALLBACKFUNC pEORCallback,
                          PVOID pCallbackArguments)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function specifies the callback function to be called at the end of rendering.
Code the callback function in the following format:
`VOID EORCallbackFunc(PVOID pCallbackArguments);`
`pCallbackArguments` (input): Pointer to the parameter set at the specification

Arguments:
`pEORCallback` (input)
Pointer to the function to be called at the end of rendering
If NULL is specified, the callback function is canceled.

`pCallbackArguments` (input)
Pointer to argument to be passed to the function called at callback

Return value:
`KMSTATUS_SUCCESS` Specified successfully

3.9.2 Specifying a V-Sync Callback Function

```
KMSTATUS kmSetVSyncCallback(PKMCALLBACKFUNC pVSyncCallback,
                            PVOID pCallbackArguments)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function specifies the callback function to be called at an entry into the vertical flyback segment (Vsync).
Code the callback function in the following format:
`VOID VSyncCallbackFunc(PVOID pCallbackArguments);`
`pCallbackArguments` (input): Pointer to the parameter set at the specification

Arguments:
`pVSyncCallback` (input)
Pointer to the function to be called at an entry into Vsync
If NULL is specified, the callback function is canceled.

`pCallbackArguments` (input)
Pointer to argument to be passed to the function called at callback

Return value:
`KMSTATUS_SUCCESS` Specified successfully

3.9.3 Specifying a V-Sync Wait Callback Function

```
KMSTATUS kmSetWaitVSyncCallback(
                        PKMCALLBACKFUNC pWaitVSyncCallback,
                        PVOID pCallbackArguments)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function specifies a callback function to be called during a Vsync wait state. It is used when reading from CD-ROM is carried out in the background or when processing is performed asynchronously with other V periods. Do not try to call too large function or to use an endless loop in a callback.

Code the callback function in the following format:
`VOID WaitVSyncCallbackFunc(PVOID pCallbackArguments);`
`pCallbackArguments` (input): Pointer to the parameter set at the specification

Arguments:
`pWaitVSyncCallback` (input)
Pointer to the function in a Vsync wait state.
If NULL is specified, the callback function is canceled.

`pCallbackArguments` (input)
Pointer to an argument to be passed to the function called at callback

Return value:
`KMSTATUS_SUCCESS` Specified successfully

3.9.4 Specifying an H-Sync Interrupt Callback Function

```
KMSTATUS kmSetHSyncCallback(PKMCALLBACKFUNC pHSyncCallback,
                            PVOID pCallbackArguments)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function specifies the callback function to be called at an entry into the horizontal flyback segment (Hsync). Code the callback function in the following format:

```
VOID HSyncCallbackFunc(PVOID pCallbackArguments);
```

`pCallbackArguments` (input): Pointer to the parameter set at the specification

Arguments:

`pHSyncCallback` (input)

Pointer to the function to be called at an entry into Hsync
If NULL is specified, the callback function is canceled.

`pCallbackArguments` (input)

Pointer to argument to be passed to the function called at callback. The line number specified by `KmSetHSyncLine` is not passed. Hold the value specified by `kmSetHSyncLine` in the area specified by this pointer. Alternatively, obtain the current scanline count using `kmGetCurrentScanline()`.

Return value:

`KMSTATUS_SUCCESS`	Specified successfully

3.9.5 Setting the H-Sync Interrupt Line

```
KMSTATUS kmSetHSyncLine(KMINT32 nInterruptLine)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function specifies the display line on which an interrupt is caused.

Argument:
`nInterruptLine` (input)
Specify the line on which an interrupt is caused. Specify a value within the range of 0 to 240/480.

Return values:
`KMSTATUS_SUCCESS` Set successfully

3.9.6 Reading the Current H-Sync Line

```
KMSTATUS kmGetCurrentScanline(PKMINT32 pScanline)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function reads the current H-Sync line. Zero is returned for the first line of the display area and a negative value is returned for a line of an area other than the display area. The negative value is counted up to -1 which indicates one line before the first line of the display area.

Argument:
`pScanline` (input)
Pointer to `KMINT32` where the current H-Sync line is stored

Return value:
`KMSTATUS_SUCCESS` Set successfully

3.9.7 Specifying a Texture Memory Overflow Callback Function

```
KMSTATUS kmSetTexOverflowCallback(
                      PKMCALLBACKFUNC pTexOverflowCallback,
                       PVOID pCallbackArguments)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function registers a callback function that is called when a texture surface is to be allocated by `kmCreateTextureSurface` or `kmCreateCombinedTextureSurface`.
Code the callback function in the following format:
`VOID TexOverflowCallbackFunc(PVOID pCallbackArguments);`
`pCallbackArguments` (input): Pointer to the parameter set at the specification

Arguments:
`pTexOverflowCallback` (input)
Pointer to the callback function to be called at texture overflow
If NULL is specified, the callback function is canceled.

`pCallbackArguments` (input)
Pointer to argument to be passed to the function called at callback

Return value:
`KMSTATUS_SUCCESS` Specified successfully

3.9.8 Specifying a Strip Buffer Overrun Callback Function

```
KMSTATUS kmSetStripOverRunCallback (
                          PKMCALLBACKFUNC pStripOverRunCallback,
                          PVOID pCallbackArguments)
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	♥

Explanation:
This function specifies the callback function to be called when the rendering of the next strip is not completed during the display period of the vertical dimension of a strip buffer.
Code the callback function in the following format:
`VOID StripOverRunCallbackFunc(PVOID pCallbackArguments);`
`pCallbackArguments` (input): Pointer to the parameter set at the specification

Arguments:
`pStripOverRunCallback` (input)
Pointer to the callback function
If NULL is specified, the callback function is canceled.

`pCallbackArguments` (input)
Pointer to argument to be passed to the function called at callback

Return value:
`KMSTATUS_SUCCESS` Specified successfully

3.9.9 Specifying a Vertex Data Transfer End Callback Function

```
KMSTATUS kmSetEndOfVertexCallback (
                            PKMCALLBACKFUNC pEndOfVertexCallback,
                             PVOID pCallbackArguments)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function specifies the callback function to be called at the end of transfer of the data of one scene from KAMUI to the rendering hardware.
Code the callback function in the following format:
`VOID EndOfVertexCallbackFunc(PVOID pCallbackArguments);`
`pCallbackArguments` (input): Pointer to the parameter set at the specification

Arguments:
`pEndOfVertexCallback` (input)
Pointer to the callback function
If NULL is specified, the callback function is canceled.

`pCallbackArguments` (input)
Pointer to argument to be passed to the function called at callback

Return value:
`KMSTATUS_SUCCESS` Specified successfully

3.9.10 Specifying a YUV Converter End Callback Function

```
KMSTATUS kmSetEndOfYUVCallback (
                        PKMCALLBACKFUNC pEndOfYUVCallback,
                        PVOID pCallbackArguments)
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	♥

Explanation:
This function specifies a callback function to be called when YUV converter processing (started by `kmLoadYUVTexture`) ends. The YUV converter is incorporated in the Holly PowerVR hardware (tiling accelerator). It is designed for conversion from YUV420 format to YUV422 format.
Code the callback function in the following format:

```
VOID EndOfYUVCallbackFunc(PVOID pCallbackArguments);
```

`pCallbackArguments` (input): Pointer to the parameter set at the specification

Arguments:

`pEndOfYUVCallback` (input)
Pointer to the callback function.
If NULL is specified, the callback function is canceled.

`pCallbackArguments` (input)
Pointer to an argument to be passed to the function called at callback

Return value:

`KMSTATUS_SUCCESS`	Specified successfully

3.10 OTHER FUNCTIONS

3.10.1 Stopping the Frame Buffer Display

```
KMSTATUS kmStopDisplayFrameBuffer (VOID)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function stops the frame buffer display. The function just causes the CRT controller to stop display and does not change the frame buffer status.

Argument:
None

Return value:

`KMSTATUS_SUCCESS`	Success

3.10.2 Obtaining the Version Information

```
KMSTATUS kmGetVersionInfo(PKMVERSIONINFO pVersionInfo)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

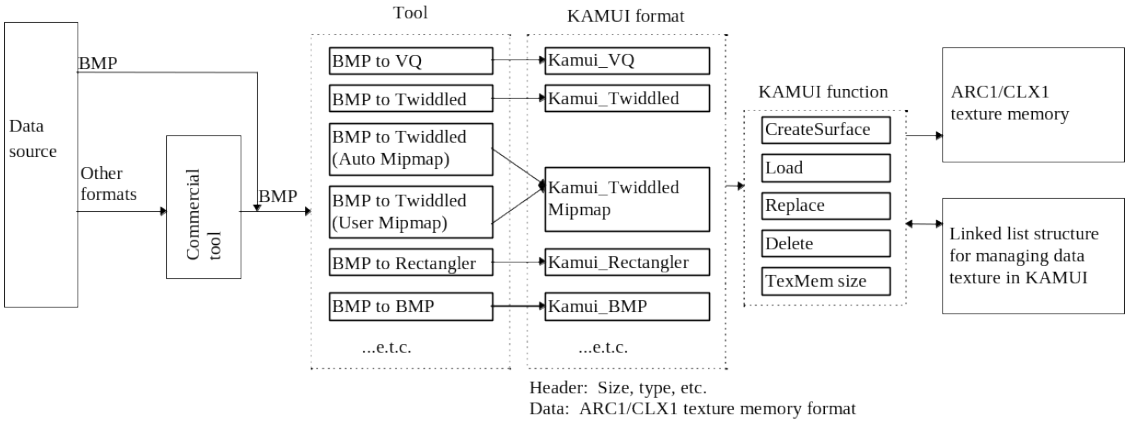
Explanation:
This function obtains the version information of the library. For the contents of the version information structure, see the structure list.
The version information becomes definite only after the `kmInitDevice` function has been issued. This function should be called after the `kmInitDevice` function has been issued.

Argument:
`pVersionInfo` (output)
Indicates a pointer to the KMVERSIONINFO structure allocated in advance.

Return value:
`KMSTATUS_SUCCESS` Success

3.11 TEXTURE HANDLING FUNCTIONS OF KAMUI

The figure below shows the texture flow with KAMUI.



The texture control function of KAMUI transfers texture in the image format on texture memory. The texture must be converted beforehand by a tool into the KAMUI texture format having the image on texture memory.

KAMUI uses the following procedures to register and use textures.

- Texture data preparation
  The application program prepares texture data in system memory. It also allocates the SurfaceDesc structure (see Section 5.1) for saving texture information. The application program need not set the contents of the structure.
- Texture surface preparation
  The `kmCreateTextureSurface` function is used to allocate a texture surface in frame buffer memory. KAMUI fixes the contents of the SurfaceDesc structure. The texture surface must be allocated separately for each texture.
- Texture data loading
  The `kmLoadTexture` function is used to transfer data to a texture surface.
- VERTEXCONTEXT setting
  The address of the SurfaceDesc structure for a texture is set in VERTEXCONTEXT (pTextureSurfaceDesc member) for the vertex data for which the texture is to be used.

For the details of the texture format, see Chapter 6.

3.11.1 Loading Texture Data

```
KMSTATUS kmLoadTexture(PKMSURFACEDESC pSurfaceDesc,
                       PKMDWORD pTexture,
                       KMBOOLEAN bAutoMipMap,
                       KMBOOLEAN bDither)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	√

Explanation:
This function loads the texture on main memory specified by `pTexture` into the texture memory area allocated by `kmCreateTextureSurface`.
The format and size of the texture to be read are identified by the surface descriptor specified by `pSurfaceDesc`. If the actual format and size of the texture are different from the contents of the surface descriptor specified by `pSurfaceDesc`, the display is illegal.
In the `Holly (CLX1/2)` version of KAMUI, `bAutoMipMap` and `bDither` cannot be set to `TRUE`. They must always be set to `FALSE`.
In addition, mipmap dither generation is the responsibility of the application program.

If the start address of texture data in system memory is aligned with a 32-byte boundary, and its size is a multiple of 32 bytes, the DMA mode is used to transfer the texture data to texture memory, so that high-speed transfer becomes possible.

If the DMA mode is used to transfer texture data, it is possible to select whether to wait until the transfer ends. If `kmSetSystemConfiguration` sets the `KM_CONFIGFLAG_NOWAIT_FINISH_TEXTUREDMA flag`, the function ends without waiting for the completion of a DMA transfer. In this case, the `kmQueryFinishLastTextureDMA` function can be used to check for the end of DMA transfer.

If the CPU directly rewrites texture data in main memory before it is loaded, it is necessary to purge the cache before executing the load function in order to maintain cache coherency. (Specifically, execute the SH4 ocbwb instruction.)

Arguments:
`pSurfaceDesc` (input)
Texture surface allocated by `kmCreateTextureSurface`, `kmCreateCombinedTextureSurface`, or `kmCreateContiguousTextureSurface`

`pTexture` (input)

Pointer to the pixel data portion of the texture in main memory. The address specified for this pointer is the first address of the texture file of KAMUI texture format + 16. Specify an address aligned with a 32-byte boundary (16 bytes of the header portion of KAMUI are skipped).

If this address is not on a 32-byte boundary, DMA transfer cannot be used, resulting in the processing being slow.

`bAutoMipMap` (input)

(This option cannot be specified for CLX1/2. If it is set to `TRUE` for CLX1/2, `KMSTATUS_INVALID_TEXTURE_TYPE` is returned.)

Specify whether MIPMAP is automatically generated. When `TRUE` is specified, MIPMAP is automatically generated. If `TRUE` is specified, the pixel data must have `KM_TEXTURE_BMP` as a category code of texture type and be a square texture of 512 x 512 texels or less.

In this case, the read texture is converted into `KM_TEXTURE_TWIDDLED_MM` format. Therefore, the type of the surface specified by `pSurfaceDesc` must be `KM_TEXTURE_TWIDDLED_MM`.

If `TRUE` is specified as `bAutoMipMap`, a work area of 512 KB is allocated in the stack. The operation is not guaranteed unless sufficient memory is allocated.

`bDither` (input)

(This option cannot be specified for CLX1/2. If it is set to `TRUE` for CLX1/2, `KMSTATUS_INVALID_TEXTURE_TYPE` is returned.)

Specifies whether dither is applied to the texture to be read. If `TRUE` is specified, dither is applied. If `TRUE` is specified, the pixel data must have `KM_TEXTURE_BMP` as a category code of texture type and be a square texture of 512 x 512 texels or less.

In this case, the read texture is converted into `KM_TEXTURE_TWIDDLED`/`KM_TEXTURE_TWIDDLED_MM` format. Therefore, the type of the surface specified by `pSurfaceDesc` must be `KM_TEXTURE_TWIDDLED`/`KM_TEXTURE_TWIDDLED_MM`.

If `TRUE` is specified for `bDither`, a work area of 512 KB is allocated in the stack.

The operation is not guaranteed unless sufficient memory is allocated.

Return values:

`KMSTATUS_SUCCESS`	Read successfully
`KMSTATUS_INVALID_ADDRESS`	The specified area (`Surface`) is not allocated.
`KMSTATUS_INVALID_TEXTURE_TYPE`	Invalid texture type specified
`KMSTATUS_INVALID_MIPMAPED_TEXTURE`	Use of AutoMIPMAP/AutoDither is attempted for a texture for which use of MIPMAP/Dither is inhibited.

3.11.2 Loading Texture Data Blocks

```
KMSTATUS kmLoadTextureBlock(
                      PKMSURFACEDESC pSurfaceDesc,
                      PKMDWORD pTexture,
                      KMUINT32 nBlockNum,
                      KMUINT32 nBlockSize)
```

IRIS+ARC1	COSMOS+ARC1	Holly
◆	◆	♥

Explanation:
This function loads texture blocks from a main memory area specified by `pTexture` into a texture memory area allocated using `kmCreateTextureSurface`.

Texture data is divided into blocks before it is loaded. It makes it possible to load large texture data without allocating a large work area in main memory.

To load texture data by dividing it in `BUFFSIZE`*32 byte units, for example, code the following:

```
i = 0;
Load the first BUFFSIZE*32 byte block into pTexture;
while(KMSTATUS_SUCCESS == kmLoadTextureBlock(
                            &TexSurfaceDesc,
                            pTexture,
                            i++,
                            BUFFSIZE
                            )) {
      Load the next BUFFSIZE*32 byte block into pTexture;
}
```

Even if the size of the whole texture data is not an integer multiple of the block size, loading is performed normally.
It is impossible to change the `BUFFSIZE` value in a loop in which one set of texture blocks is being loaded. If the value is changed, the texture display becomes illegal.

The format and size of the texture data to be loaded are identified by the surface descriptor specified by `pSurfaceDesc`. If the actual format and size of the texture data are different from the contents of the surface descriptor specified by `pSurfaceDesc`, the display becomes illegal.

If the start address of texture data in system memory is on a 32-byte boundary, and its size is a multiple of 32 bytes, the DMA mode is used to transfer the texture data to texture memory, so that high-speed transfer becomes possible.

If the DMA mode is used to transfer texture data, it is possible to select whether to wait until the transfer ends. If `kmSetSystemConfiguration` sets the `KM_CONFIGFLAG_NOWAIT_FINISH_TEXTUREDMA flag`, the function ends without waiting for the completion of DMA transfer. In this case, the `kmQueryFinishLastTextureDMA` function can be used to check for the end of DMA transfer.

If the CPU directly rewrites texture data into main memory before it is loaded, it is necessary to purge the cache before executing the load function in order to maintain cache coherency. (Specifically, execute the SH4 ocbwb instruction.)

Arguments:

`pSurfaceDesc` (input)
Texture surface allocated by `kmCreateTextureSurface`, `kmCreateCombinedTextureSurface`, or `kmCreateContiguousTextureSurface`

`pTexture` (input)
Pointer to the beginning of a texture block in main memory. If the pointer is not on a 32-byte boundary, DMA transfer cannot be used, resulting in the processing becoming slow.

`nBlockNum` (input)
Specify a texture block number from 0 to n (n varies with the format and size).

`nBlockSize` (input)
Specify the size of a texture block in 32-byte units, that is, an actual block size (in bytes) divided by 32. Even if the size of the entire texture block is not an integer multiple of the block size, loading is performed normally.

Caution This function does not support texture data of Small VQ format. If pSurfaceDesc of Small VQ format is specified, KMSTATUS_INVALID_TEXTURE_TYPE is returned. ARC1 does not support a format (like VQ, VQ-mipmap, or Twiddled-mipmap) that involves interleaving.

Return values:

`KMSTATUS_SUCCESS`	Read successfully
`KMSTATUS_INVALID_BLOCKNUMBER`	Illegal block number
`KMSTATUS_INVALID_ADDRESS`	Specified area (`Surface`) not allocated.
`KMSTATUS_INVALID_TEXTURE_TYPE`	Invalid texture type specified.

3.11.3 Loading Part of Texture Data

```
KMSTATUS kmLoadTexturePart(
                    PKMSURFACEDESC pSurfaceDesc,
                    PKMDWORD pTexture,
                    KMUINT32 nOffset,
                    KMUINT32 nSize)
```

IRIS+ARC1	COSMOS+ARC1	Holly
◆	◆	♥

Explanation:
This function loads texture portions from a main memory area specified by `pTexture` into a texture memory area allocated using `kmCreateTextureSurface`.

Texture data is divided into portions before it is loaded. This makes it possible to load a large amount of texture data without allocating a large work area in main memory.

Unlike `kmLoadTextureBlock`, `kmLoadTexturePart` can load one texture data item by dividing it into portions of different sizes. The user is responsible for managing the size (offset from the beginning of the texture data) of each texture portion that has already been loaded.

The format and size of the texture data to be loaded are identified by the surface descriptor specified by `pSurfaceDesc`. If the actual format and size of the texture data are different from the contents of the surface descriptor specified by `pSurfaceDesc`, the display becomes illegal.

If the start address of texture data in system memory is on a 32-byte boundary, and its size is a multiple of 32 bytes, the DMA mode is used to transfer the texture data to texture memory, so that high-speed transfer becomes possible.

If the DMA mode is used to transfer texture data, it is possible to select whether to wait until the transfer ends. If `kmSetSystemConfiguration` sets the `KM_CONFIGFLAG_NOWAIT_FINISH_TEXTUREDMA flag`, the function ends without waiting for the completion of a DMA transfer. In this case, the `kmQueryFinishLastTextureDMA` function can be used to check for the end of DMA transfer.

If the CPU directly rewrites texture data into main memory before it is loaded, it is necessary to purge the cache before executing the load function in order to maintain cache coherency. (Specifically, execute the SH4 ocbwb instruction.)

Arguments:

`pSurfaceDesc` (input)

Texture surface allocated by `kmCreateTextureSurface`, `kmCreateCombinedTextureSurface`, or `kmCreateContiguousTextureSurface`

`pTexture` (input)

Pointer to the beginning of a texture data portion (work area) in main memory. If the pointer is not on a 32-byte boundary, DMA transfer cannot be used, resulting in the processing becoming slow.

`nOffset` (input)

Specify the size of the texture data portion that has already been loaded (the offset from the beginning of the entire texture data) in byte units. This size must be an integer multiple of 4, because it is used to obtain the address of the transfer destination texture area in frame buffer memory.

`nSize` (input)

Specify the size of the texture data portion to be loaded, in byte units. This size must be an integer multiple of 4. If nSize is greater than the size of the remaining texture portion (= texture size - nOffset), texture data loading is completed by loading only the rest of the texture data.

Caution This function does not support texture data of Small VQ format. If pSurfaceDesc for the Small VQ format is specified, KMSTATUS_INVALID_TEXTURE_TYPE is returned. ARC1 does not support a format (like VQ, VQ-mipmap, or Twiddled-mipmap) that involves interleaving.

Example:

```
nOffset = 0;
nSize   = ***;
Load the first nSize byte portion of texture data into an area
specified by pTexture;
while(KMSTATUS_SUCCESS == kmLoadTexturePart(...);
      nOffset = nOffset + nSize;
      nSize   = ????;
      Load the next nSize byte portion into an area specified by
      pTexture;
}
```

Return values:

`KMSTATUS_SUCCESS`	Read successfully
`KMSTATUS_INVALID_ADDRESS`	`nOffset` greater than the texture size.
`KMSTATUS_INVALID_TEXTURE_TYPE`	Invalid texture type specified.

3.11.4 Re-reading the Code Book Portion of VQ Texture

```
KMSTATUS kmLoadVQCodebook(  PKMSURFACEDESC pSurfaceDesc,
                            PKMDWORD pTexture)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function reads only the code book portion of the VQ or small VQ texture in main memory as specified by `pTexture` to the VQ or small VQ texture surface specified by `pSurfaceDesc`. It is used to rewrite only the code book of the VQ or small VQ texture already loaded and use the color palette effect.

If the start address of texture data in system memory is on a 32-byte boundary, and its size is a multiple of 32 bytes, the DMA mode is used to transfer the texture data to texture memory, so that high-speed transfer becomes possible.

If the DMA mode is used to transfer texture data, it is possible to select whether to wait until the transfer ends. If `kmSetSystemConfiguration` sets the `KM_CONFIGFLAG_NOWAIT_FINISH_TEXTUREDMA flag`, the function ends without waiting for the completion of DMA transfer. In this case, the `kmQueryFinishLastTextureDMA` function can be used to check for the end of DMA transfer.

If the CPU directly rewrites texture data into main memory before it is loaded, it is necessary to purge the cache before executing the load function in order to maintain cache coherency. (Specifically, execute the SH4 ocbwb instruction.)

Arguments:
`pSurfaceDesc` (input)
Texture surface allocated by `kmCreateTextureSurface` or `kmCreateCombinedTextureSurface`. The category of this surface must be one of the following types:

```
KM_TEXTURE_VQ
KM_TEXTURE_VQ_MM
KM_TEXTURE_SMALLVQ
KM_TEXTURE_SMALLVQ_MM
```

`pTexture` (input)
Pointer indicating a texture (code book) in main memory. Specify an address aligned with a 32-byte boundary. This does not have to be in the complete VQ or small VQ texture format, but a code book (its size in bytes is indicated below) must be included in the beginning.

The following table lists the relationships between the texture size and code book size.

Texture type/size	Code book size (byte)
VQ/VQ mipmap	0x800
16 x 16 small VQ	0x80
16 x 16 small VQ mipmap	0x80
32 x 32 small VQ	0x100
32 x 32 small VQ mipmap	0x200
64 x 64 small VQ	0x400

Return values:

`KMSTATUS_SUCCESS`	Read successfully
`KMSTATUS_INVALID_TEXTURE_TYPE`	Invalid texture surface specified

3.11.5 Reloading a Particular Mipmap Texture

```
KMSTATUS kmReLoadMipmap(PKMSURFACEDESC pSurfaceDesc,
                        PKMVOID pTexture,
                        KMINT32 nMipmapCount)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function overwrites the mipmap texture on main memory specified by `pTexture` and loads it into the texture memory area allocated by `kmCreateTextureSurface`.
The type of the texture (surface) that can be specified is one of the following types:

```
KM_TEXTURE_TWIDDLED_MM
KM_TEXTURE_VQ_MM
KM_TEXTURE_PALETTIZE4_MM
KM_TEXTURE_PALETTIZE8_MM
KM_TEXTURE_SMALLVQ_MM
```

The format and size of the texture to be read are identified by the surface descriptor specified by `pSurfaceDesc`.

[Reference]
Offset from the beginning of Twiddled mipmap file in KAMUI texture format to each mipmap level and the number of bytes of each mipmap level (except 16 bytes of header section)

SIZE	OFFSET	BYTES
1 x 1	6	2
2 x 2	8	8
4 x 4	16 (10h)	32 (20h)
8 x 8	48 (30h)	128 (80h)
16 x 16	176 (B0h)	512 (200h)
32 x 32	688 (2B0h)	2,048 (800h)
64 x 64	2,736 (AB0h)	8,192 (2000h)
128 x 128	10,928 (2AB0h)	32,768 (8000h)
256 x 256	43,696 (AAB0h)	13,1072 (20000h)
512 x 512	174,768 (2AAB0h)	524,288 (80000h)
1,024 x 1,024	699,056 (AAAB0h)	2,097,152 (200000h)

Offset from the beginning of VQ mipmap file in KAMUI texture format to each mipmap level and the number of bytes of each mipmap level (except 16 bytes of header section)

SIZE	OFFSET	BYTES
1 x 1	--	--
2 x 2	2,048 + 1	1
4 x 4	2,048 + 2	4
8 x 8	2,048 + 6	16 (10h)
16 x 16	2,048 + 22 (16h)	64 (40h)
32 x 32	2,048 + 86 (56h)	256 (100h)
64 x 64	2,048 + 342 (156h)	1,024 (400h)
128 x 128	2,048 + 1,366 (556h)	4,096 (1000h)
256 x 256	2,048 + 5,462 (1556h)	16,384 (4000h)
512 x 512	2,048 + 21,846 (5556h)	65,536 (10000h)
1,024 x 1,024	2,048 + 87,382 (15556h)	262,144 (40000h)

Offset from the beginning of the palettized 4-bpp mipmap file in KAMUI texture format to each mipmap level and the number of bytes of each mipmap level (except 16 bytes of the header section)

SIZE	OFFSET	BYTES
1 x 1	1	0.5
2 x 2	2	2
4 x 4	4	8
8 x 8	12 (0Ch)	32 (20h)
16 x 16	44 (2Ch)	128 (80h)
32 x 32	172 (ACh)	512 (200h)
64 x 64	684 (2ACh)	2,048 (800h)
128 x 128	2,732 (AACh)	8,192 (2000h)
256 x 256	10,924 (2AACh)	32,768 (8000h)
512 x 512	43,692 (AAACh)	131,072 (20000h)
1,024 x 1,024	174,764 (2AAACh)	524,288 (80000h)

Offset from the beginning of the palettized 8-bpp mipmap file in KAMUI texture format to each mipmap level and the number of bytes of each mipmap level (except 16 bytes of the header section)

SIZE	OFFSET	BYTES
1 x 1	3	1
2 x 2	4	4
4 x 4	8	16 (10h)
8 x 8	24 (18h)	64 (40h)
16 x 16	88 (58h)	256 (100h)
32 x 32	344 (158h)	1,024 (400h)
64 x 64	1,368 (558h)	4,096 (1000h)
128 x 128	5,464 (1558h)	16,384 (4000h)
256 x 256	21,848 (5558h)	65,536 (10000h)
512 x 512	87,384 (15558h)	262,144 (40000h)
1,024 x 1,024	349,528 (55558h)	1,048,576 (100000h)

If the start address of texture data in system memory is on a 32-byte boundary, and its size is a multiple of 32 bytes, the DMA mode is used to transfer the texture data to texture memory, so that high-speed transfer becomes possible.

If the DMA mode is used to transfer texture data, it is possible to select whether to wait until the transfer ends. If `kmSetSystemConfiguration` sets the `KM_CONFIGFLAG_NOWAIT_FINISH_TEXTUREDMA flag`, the function ends without waiting for the completion of DMA transfer. In this case, the `kmQueryFinishLastTextureDMA` function can be used to check for the end of DMA transfer.

If the CPU directly rewrites texture data into main memory before it is loaded, it is necessary to purge the cache before executing the load function in order to maintain cache coherency. (Specifically, execute the SH4 ocbwb instruction.)

Arguments:

`pSurfaceDesc` (input)

Texture surface allocated by `kmCreateTextureSurface` or `kmCreateCombinedTextureSurface`

<Reload destination>

`pTexture` (input)

Pointer indicating the pixel data portion of the texture in main memory. Indicates the beginning of the texture data of the mipmap level specified by `nMipmapCount`.

<Reload source>

`nMipmapCount` (input)

Specify the level of the mipmap texture to be read. One of the following enum values can be specified.

nMipmapCount	Texture Size
KM_MAPSIZE_1	1 x 1
KM_MAPSIZE_2	2 x 2
KM_MAPSIZE_4	4 x 4
KM_MAPSIZE_8	8 x 8
KM_MAPSIZE_16	16 x 16
KM_MAPSIZE_32	32 x 32
KM_MAPSIZE_64	64 x 64
KM_MAPSIZE_128	128 x 128
KM_MAPSIZE_256	256 x 256
KM_MAPSIZE_512	512 x 512
KM_MAPSIZE_1024	1,024 x 1,024

Caution No DMA transfer can be used in CLX1/2, resulting in this function being slower than `kmLoadTexture`. This is because, in texture data transfer by the function, the address of the data transfer source and destination areas is not necessarily on a 32-byte boundary.
The correct picture is not displayed if the code book at the reloading destination and that at the reloading source coincide when reloading VQ-Mipmap. Nothing is performed if 1 x 1 Mipmap is specified when reloading VQ-Mipmap.
The correct picture is not displayed if the texture palette data at the reloading destination and that at the reloading source coincide when reloading Palettized-Mipmap.

Return values:

KMSTATUS_SUCCESS	Success
KMSTATUS_INVALID_PARAMETER	Invalid parameter
KMSTATUS_INVALID_TEXTURE	Invalid texture specified

3.11.6 Reading the YUV-Format Texture Data

```
KMSTATUS kmLoadYUVTexture(PPKMSURFACEDESC ppSurfaceDesc,
                          PKMDWORD pTexture,
                          KMINT32 nWidth,
                          KMINT32 nHeight,
                          KMINT32 nFormat,
                          KMBOOLEAN bWaitEndOfDMA
                          )
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	♥

Explanation:
This function converts the YUV420-data/YUV422-data in main memory specified by pTexture into Non-Twiddled YUV422 texture and reads it into a texture memory area allocated by kmCreateTextureSurface/kmCreateCombinedTextureSurface/ kmCreateContiguousTextureSurface.
In doing so, the YUV-data converter built into tiling accelerator of the CLX1/2 is used. Because the output of the YUV-data converter is Non-Twiddled, the texture surface at the read destination specified by this API must be in either of the following formats:

```
KM_TEXTURE_RECTANGLE | KM_TEXTURE_YUV422   // Rectangular
KM_TEXTURE_STRIDE | KM_TEXTURE_YUV422      // Rectangular (with stride specification)
```

If two or more YUV-data are read successively at one time (nWidth x nHeight > 1), the size of each texture must be 16 x 16 texels. Exercise care in specifying the size of texture surface at the read destination specified by this function. In this case, texture surface at read destination must be allocated to contiguous addresses in the frame buffer. Specify the texture surface allocated by "kmCreateContiguousTextureSurface" function.

When one YUV-data item is loaded (when nWidth x nHeight = 1), both the vertical and horizontal texture sizes must be 16, 32, 64, 128, 256, 512, or 1,024. In this case, the texture sizes are identified according to the contents of ppSurfaceDesc, that is, a value specified at texture surface generation is used.

If the CPU directly rewrites texture data into main memory before it is loaded, it is necessary to purge the cache before executing the load function in order to maintain cache coherency. (Specifically, execute the SH4 ocbwb instruction.)

Arguments:
ppSurfaceDesc (input)
Pointer of pointer array to KMSURFACEDESC structure indicating the texture surface that has already been allocated

`pTexture` (input)

Pointer indicating YUV420-data/YUV422-data in main memory. Specify an address aligned with a 32-byte boundary.

If the address is not on a 32-byte boundary, the YUV converter cannot operate because of hardware constraints. In this case, `KMSTATUS_INVALID_ADDRESS` is returned.

`nWidth` and `nHeight` (input)

Specify the horizontal and vertical numbers of the 16 x 16 texel macro blocks to be loaded consecutively. A value between 1 and 64 can be specified. As many macro blocks as nWidth x nHeight are loaded.

`nFormat` (input)

Specifies the format of the data to be read. Specify either of the following:

```
KM_TEXTURE_YUV420    // Indicates that the input data is YUV420-data.
KM_TEXTURE_YUV422    // Indicates that the input data is YUV422-data.
```

`bWaitEndOfDMA` (input)

If TRUE is specified, the function waits until the DMA transfer of data to the YUV converter has been completed. This API does not end until DMA transfer ends. If FALSE is specified, the function does not wait until DMA transfer ends. To detect the end of DMA transfer, use the `kmSetEndOfYUVCallback` function.

Return values:

`KMSTATUS_SUCCESS`	Success
`KMSTATUS_INVALID_TEXTURE_TYPE`	Invalid texture specified
`KMSTATUS_INVALID_ADDRESS`	`pTexture` not on a `32`-byte boundary.

3.11.7 Deleting Texture Data

```
KMSTATUS kmFreeTexture(PKMSURFACEDESC pSurfaceDesc)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function releases a specified texture surface.

Argument:
`pSurfaceDesc` (I/O)
Texture surface allocated by `kmCreateTextureSurface`

Return values:

`KMSTATUS_SUCCESS`	Released successfully
`KMSTATUS_INVALID_ADDRESS`	Specified area (`Surface`) is not allocated.

3.11.8 Obtaining the Available Texture Memory Space

```
KMSTATUS kmGetFreeTextureMem(PKMUINT32 pSizeOfTexture PKMUNIT32
                             pMaxBlockSizeOfTexture
                             )
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function returns the unused capacity of the texture memory.
The texture memory is managed in block units. If it is repeatedly allocated and released, the texture memory is divided into many blocks. This API can check the total size (**`pSizeOfTexture`**) of all the vacant blocks of texture memory and the size of the largest vacant block (**`pMaxBlockSizeOfTexture`**).
Even if the total size of the vacant blocks is sufficient, if the size of the largest vacant block is not sufficient, `KMSTATUS_NOT_ENOUGH_MEMORY` (insufficient memory) is returned when a texture surface is allocated (`kmCreateTextureSurface` or `kmCreateCombinedTextureSurface`).
With ARC1, the VQ and mipmap textures are interleaved in memory. Therefore, these surfaces may not be secured even if `pMaxBlockSizeOfTexture` is sufficient.

To use the texture memory efficiently, secure and release as many texture surfaces as possible.

Argument:
`pSizeOfTexture` (input)
Pointer to the KMDWORD area to which the available texture memory space is returned

`pMaxBlockSizeOfTexture` (input)
Pointer to the KMDWORD area to which the largest vacant block in the texture memory is to be returned.

Return value:
`KMSTATUS_SUCCESS` Success

3.11.9 Reading the Texture in Texture Memory

```
KMSTATUS kmGetTexture(      PKMDWORD pTexture,
                            PKMSURFACEDESC pSurfaceDesc)
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function reads the texture in texture memory specified by `pSurfaceDesc` to the main memory specified by `pTexture`. Only the texture pixel data of the KAMUI texture format is output. No header is appended. If `SurfaceDesc` of the frame buffer is specified for `pSurfaceDesc`, the contents of the specified frame buffer can be read into main memory.

If the start address of texture data in system memory is on a 32-byte boundary, and its size is a multiple of 32 bytes, the DMA mode is used to transfer the texture data to texture memory, so that high-speed transfer becomes possible.

Arguments:
`pTexture` (output)
Pointer indicating the area in main memory where the texture is to be saved. Secure a multiple of 32 bytes, aligned with a 32-byte boundary.
<Read destination>

`pSurfaceDesc` (input)
Texture surface to which the texture is saved.
<Read source>

Return values:

`KMSTATUS_SUCCESS`	Read successfully
`KMSTATUS_INVALID_ADDRESS`	Specified texture surface is not allocated.

3.11.10 Garbage Collection of Texture Memory

```
KMSTATUS kmGarbageCollectTexture( VOID )
```

IRIS+ARC1	COSMOS+ARC1	Holly
♥	♥	♥

Explanation:
This function performs garbage collection for the frame buffer memory. If there is a vacant area at addresses lower than the already allocated texture surface, the texture is moved and aligned with the lower addresses.
The address of the texture is changed after this function has been called (the contents of `pSurface` of the `KMSURFACEDESC` structure are rewritten). Consequently, `kmProcessVertexRenderState` and `kmSetVertexRenderState` must be re-executed for all the `KMVERTEXCONTEXT` structures using texture after this function is used.
Note that the frame buffer area for display, native data buffer, and VQ/Mipmap texture area of ARC1 are not subject to garbage collection. To use the memory efficiently, call the functions that create a frame buffer area for display and native data buffer (`kmCreateFrameBufferSurface` and `kmCreateVertexBuffer API`) before creating the texture surface and, whenever possible, release and re-create these areas after the end of AP. With ARC1, allocate or release VQ/Mipmap texture areas at the same time and in combination.

Argument:
None

Return value:
`KMSTATUS_SUCCESS` Garbage collection successful

3.11.11 Checking for Texture Load DMA Transfer End

```
KMSTATUS kmQueryFinishLastTextureDMA(KMVOID)
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	♥

Explanation:
This function checks whether a DMA transfer started by the previous texture load function (kmLoadTexture, kmLoadTextureBlock, kmLoadTexturePart, kmLoadVQCodebook, or kmReLoadMipmap) has ended. The function is valid only if `kmSetSystemConfiguration` sets the `KM_CONFIGFLAG_NOWAIT_FINISH_TEXTUREDMA` `flag`. Otherwise, KMSTATUS_SUCCESS is returned.

Argument:
None

Return values:

`KMSTATUS_SUCCESS`	Previous texture load `DMA` transfer ended.
`KMSTATUS_NOT_FINISH_DMA`	Previous texture load `DMA` transfer not ended.

4. KAMUI UTILITY LIBRARY

This chapter explains the utility library.
Those functions that do not access the hardware but which are closely related to the hardware are supplied as a utility library. To use these functions, include "kmutil.h" in the source code of the application, and link "kmutil.lib".
The names of all the functions included in this library start with "kmu".

4.1 SELECTING ENVIRONMENTS

4.1.1 Selecting a Target Environment

```
KMSTATUS        kmuSetTarget( KMDWORD dwTarget )
```

Explanation:
This function specifies a target system for the KAMUI utility library. Before using the Kmutil library, execute this function.

Argument:
`dwTarget` (input)
Specifies a target system by selecting it from the following:

KMU_TARGET_ARC1	ARC1 is selected as the target.
KMU_TARGET_CLX1	CLX1 is selected as the target.
KMU_TARGET_CLX2	CLX2 is selected as the target.

Return value:

`KMSTATUS_SUCCESS`	Selected successfully

4.2 TEXTURE-RELATED FUNCTIONS

4.2.1 Conversion from KAMUI Bit Map Format to Twiddled Format

```
KMSTATUS       kmuCreateTwiddledTexture (
                              PKMDWORD      pOutputTexture,
                              PKMDWORD      pInputTexture,
                              KMBOOLEAN     bAutoMipMap,
                              KMBOOLEAN     bDither,
                              KMINT32       USize,
                              KMTEXTURETYPE nTextureType
                              )
```

Explanation:
This function converts a texture in KM_TEXTURE_BMP format (ABGR8888) in main memory into a texture in Twiddled/Twiddled Mipmap format. If TRUE is specified for bAutoMipMap, a mipmap is created automatically. If TRUE is specified for bDither, dither is effected.

Caution The contents of the input texture data are destroyed if mipmap or dither is specified.

Arguments:
pOutputTexture (output)
Address in main memory to which converted texture data is to be written.

pInputTexture (input)
Pointer indicating an input texture in KM_TEXTURE_BMP format.

bAutoMipMap (input)
Specifies whether a mipmap is created automatically. If TRUE is specified, a mipmap is automatically created (the output is in KM_TEXTURE_TWIDDLED_MM format). If FALSE is specified, a mipmap is not created (output is in KM_TEXTURE_TWIDDLED format).

bDither (input)
Specifies whether dither is effected. If TRUE is specified, dither is effected.

`USize` (input)

Specifies the number of texels per side of texture. Select one of the following:

```
KM_MAPSIZE_8
KM_MAPSIZE_16
KM_MAPSIZE_32
KM_MAPSIZE_64
KM_MAPSIZE_128
KM_MAPSIZE_256
KM_MAPSIZE_512
KM_MAPSIZE_1024
```

`nTextureType` (input)

Specifies the pixel format of the converted texture. Select one of the following:

```
KM_TEXTURE_ARGB1555
KM_TEXTURE_RGB565
KM_TEXTURE_ARGB4444
```

Return values:

`KMSTATUS_SUCCESS`	Converted successfully
`KMSTATUS_INVALID_TEXTURE_TYPE`	Invalid texture type specified

4.2.2 Conversion from Rectangle Format to Windows BMP Format

```
KMSTATUS      kmuConvertFBtoBMP (
                              PKMDWORD      pOutputData,
                              PKMDWORD      pInputData,
                              KMINT32       nWidth,
                              KMINT32       nHeight,
                              KMBPPMODE     nBpp
                              )
```

Explanation:
This function converts the contents of the frame buffer (rectangle format) read into main memory by `kmGetTexture` into pixel data in Windows full-color BMP format (BGR888) and writes it into memory. This is a debug function in that it saves the contents of the frame buffer in Windows BMP format.
This function does not create the 54 bytes of the header in Windows BMP format.

Arguments:
`pOutputData` (output)
Address of main memory into which the converted pixel data is to be written.

`pInputData` (input)
Pointer indicating the contents of the frame buffer. Pointer to the pixel data of the frame buffer read by specifying a descriptor of the frame buffer surface by using `kmGetTexture`.

`nWidth` and `nHeight` (input)
Specifies the screen size of the read frame buffer.

`nBpp` (input)
Specifies the pixel format of the read frame buffer. One of the following can be specified.

```
KM_DSPBPP_RGB565
KM_DSPBPP_RGB555
KM_DSPBPP_ARGB4444
KM_DSPBPP_ARGB1555
```

Return value:

`KMSTATUS_SUCCESS`	Converted successfully

4.3 FUNCTIONS RELATED TO VERTEXCONTEXT

4.3.1 Multipass VERTEXCONTEXT Automatic Generation

```
KMUPASSSTATUS kmuGeneratePassTable(
        PKMVERTEXCONTEXT    pVertexContext,
        KMUINT32            nNumContext,
        PPKMVERTEXCONTEXT   ppVertexContextTable,
        PKMUINT32           pPass
        )
```

IRIS+ARC1	COSMOS+ARC1	Holly
-	-	√

Explanation:
This function generates the context for each pass of the multipass process (Trilinear) according to the rendering specification (context) set by the user.

A multipass process requires that `VERTEXCONTEXT` be set (when a trilinear filter is used). This function automatically generates `VERTEXCONTEXT` to relieve the user from the task of setting it for individual passes.

When a trilinear filter is used in pVertexContext, specifying a VERTEXCONTEXT value for pass 1 generates the VERTEXCONTEXT required for each pass according to the specified value. (The opaque polygon uses a two-pass process, while the transparent polygon uses a three-pass process.) When a trilinear filter is used for the transparent polygon, the blending mode for pass 3 can be set to any value. However, this function sets the blending mode as follows:

SRCBlendingMode = KM_SRCALPHA
DSTBlendingMode = KM_INVSRCALPHA

If NULL is specified in ppVertexContextTable, only the required number of passes is returned to pPass.

Arguments:

`pVertexContext` (input)
Pointer to the context for specifying rendering conditions

`nNumContext` (input)
Specifies the number of entries (passes) in the prepared `pVertexContextTable`. If the specified value is smaller than the number of actually required passes, `KMU_PASS_ERROR_VERTEXCONTEXT_PASS` is returned. In this case, the function ends only by setting the number of required passes in `pPass`.

`ppVertexContextTable` (output)
Specifies a pointer to an array of pointers to `VERTEXCONTEXT` where the generated multipass context is to be received. If `NULL` is specified in this argument, only the number of required passes is returned to `pPass`.

`pPass` (output)
KAMUI returns the number of multipasses required in the specified rendering to this argument.

Return value:

`KMU_PASS_OK`	Set successfully
`KMU_PASS_ERROR_VERTEXCONTEXT`	Invalid (`NULL`) `VERTEXCONTEXT`
`KMU_PASS_ERROR_VERTEXCONTEXT_PASS`	The number of specified passes is insufficient.

4.3.2 Checking VERTEXCONTEXT

```
KMUPASSSTATUS kmuCheckPassTable(
        PPKMVERTEXCONTEXT    ppVertexContextTable,
        KMUINT32             nNumContext,
        PKMUINT32            pPass
        )
```

IRIS+ARC1	COSMOS+ARC1	Holly
√	√	√

Explanation:
This function checks whether the content of each context in the specified `VERTEXCONTEXT` table is correct. The function is intended mainly for debugging when multiple passes are used.

Arguments:
`ppVertexContextTable` (input)
Specifies a pointer to an array of pointers to the prepared `VERTEXCONTEXT`.

`nNumContext` (input)
Specifies the entries (passes) in the prepared `ppVertexContextTable`.

`pPass` (output)
If an error is detected, KAMUI sets the invalid `VERTEXCONTEXT` in `pPass`.
(If `KMU_PASS_OK` is returned, the contents of `pPass` will be undefined.)

Return values:

`KMU_PASS_OK`	Set successfully
`KMU_PASS_ERROR_VERTEXCONTEXT`	Invalid (`NULL`) `VertexContext`
`KMU_PASS_ERROR_VERTEXCONTEXT_PASS`	Invalid `nNumContext` (less than 1)
`KMU_PASS_ERROR_SPRITE_SETTING`	Invalid combination of `sprite` settings
`KMU_PASS_ERROR_TRILINEAR_SETTING`	Invalid combination of `trilinear` settings
`KMU_PASS_ERROR_BUMP_SETTING`	Invalid combination of `bump` settings
`KMU_PASS_ERROR_BLENDINGMODE_SETTING`	Invalid combination of `blending` settings
`KMU_PASS_ERROR_MODIFIER_SETTING`	Invalid combination of `modifier` settings
`KMU_PASS_ERROR_PARAMTYPE`	Invalid `parameter type`
`KMU_PASS_ERROR_LISTTYPE`	Invalid `list type`

KMU_PASS_ERROR_PIXELFORMAT	Invalid pixel format

KMU_PASS_ERROR_MIPMAP_D_ADJUST
: Invalid Mipmap_D_Adjust

KMU_PASS_ERROR_SHADINGMODE
: Invalid shading mode (for ARC1)

KMU_PASS_ERROR_FOGMODE
: Invalid fog mode (for ARC1)

KMU_PASS_ERROR_FILTERMODE
: Invalid filter mode (for ARC1)

KMU_PASS_ERROR_TEXTURESHADINGMODE
: Invalid texture shading mode (for ARC1)

5. STRUCTURES

5.1 FRAME BUFFER/TEXTURE SURFACE STRUCTURE

```
typedef struct tagKMSURFACEDESC
{
        KMDWORD SurfaceType;              // 0... FrameBuffer 1...Texture
                //      KM_SURFACETYPE_FRAMEBUFFER
                //      KM_SURFACETYPE_TEXTURE
                //      KM_SURFACETYPE_SMALLVQ_TEXTURE
                // Note  The higher 16 bits of this member are reserved for Ninja.
        KMDWORD BitDepth;                 // Indicates number of bits per pixel (e.g.,
                                             16 for 16 bpp)
                //      KM_BITDEPTH_16
                //      KM_BITDEPTH_24
                //      KM_BITDEPTH_32
        KMDWORD PixelFormat;              // 1555, 4444, etc.
                //      KM_PIXELFORMAT_ARGB1555
                //      KM_PIXELFORMAT_RGB565
                //      KM_PIXELFORMAT_ARGB4444
                //      KM_PIXELFORMAT_YUV422
                //      KM_PIXELFORMAT_BUMP
                //      KM_PIXELFORMAT_PALETTIZED_4BPP
                //      KM_PIXELFORMAT_PALETTIZED_8BPP
        union{
                KMDWORD USize;            // Usize 8 – 1024
                //      KM_MAPSIZE_8
                //      KM_MAPSIZE_16
                //      KM_MAPSIZE_32
                //      KM_MAPSIZE_64
                //      KM_MAPSIZE_128
                //      KM_MAPSIZE_256
                //      KM_MAPSIZE_512
                //      KM_MAPSIZE_1024
                KMDWORD nWidth;           // For Frame Buffer, Horizontal Size
        }u0;
        union{
                KMDWORD VSize;            // Vsize 8 – 1024
                //      KM_MAPSIZE_8
```

```
                //       KM_MAPSIZE_16
                //       KM_MAPSIZE_32
                //       KM_MAPSIZE_64
                //       KM_MAPSIZE_128
                //       KM_MAPSIZE_256
                //       KM_MAPSIZE_512
                //       KM_MAPSIZE_1024
                KMDWORD nHeight;          // For Frame Buffer, Vertical Size
        }u1;
        union {
                KMDWORD dwTextureSize;    // Texture Size (byte)
                KMDWORD dwFrameBufferSize; // FrameBuffer Size (byte)
        }uSize;
        KMDWORD  fSurfaceFlags;           // Surface Flags
        PKMDWORD pSurface;                // Pointer to Surface Instance
        PKMDWORD pVirtual;                /* Texture instance(Virtual  address on SH4)*/
        PKMDWORD pPhysical;               /* Texture instance(physical address on SH4)*/
}KMSURFACEDESC, *PKMSURFACEDESC;
```

fSurfaceFlags

```
----  For texture
bit 0   0... Non MipMap           1... MipMapped
bit 2   0... Rectangle            1... Twiddled
bit 3   0... NonVQ                1... VQed Texture
bit 4   0... NonStride            1... Stride Select
bit 5   0... Non Palettized       1... Palettized

Specify the OR of the following flags:
KM_SURFACEFLAGS_MIPMAPED          0x001
KM_SURFACEFLAGS_TWIDDLED          0x004
KM_SURFACEFLAGS_VQ                0x008
KM_SURFACEFLAGS_STRIDE            0x010
KM_SURFACEFLAGS_PALETTIZED        0x020

----  For frame buffer
bit 0   0... Full-Screen Buffer  1... StripBuffer
```

5.2 VERSION INFORMATION STRUCTURE

```
typedef struct KMVERSIONINFO
{
         KMDWORD kmMajorVersion;  // 1 for IRIS, 2 for COSMOS, 3 for HOLLY
         KMDWORD kmLocalVersion;
         KMDWORD kmFrameBufferSize; // Total Size of Texture and Frame Buffer
} KMVERSIONINFO, *PKMVERSIONINFO;
```

5.3 VERTEX CONTEXT

```
typedef struct tagKMVERTEXCONTEXT
{
KMDWORD                   RenderState;              // Render Context

/* for Global Parameter */
KMPARAMTYPE               ParamType                 // Parameter Type
KMLISTTYPE                ListType                  // List Type
KMCOLORTYPE               ColorType                 // Color Type
KMUVFORMAT                UVFormat                  // UV format

/* for ISP/TSP Instruction Word */
KMDEPTHMODE               DepthMode;                // Depth Mode Specification
KMCULLINGMODE             CullingMode;              // Culling Mode
KMSCREENCOORDINATION      ScreenCoordination;       // Screen Coordination
                                                    (ignored by Holly)
KMSHADINGMODE             ShadingMode;              // Shading Mode
KMMODIFIER                SelectModifier;           // Modifier Volume Valiant
KMBOOLEAN                 bZWriteDisable;           // Z Write Disable

/* for TSP Control Word */
KMBLENDINGMODE            SRCBlendingMode;          // Source Blending Mode
KMBLENDINGMODE            DSTBlendingMode;          // Desitination Blending Mode
KMBOOLEAN                 bSRCSel;                  // Source Select
KMBOOLEAN                 bDSTSel;                  // Distination Select
KMFOGMODE                 FogMode;                  // Fogging
KMBOOLEAN                 bUseSpecular;             // Specular Highlight
KMBOOLEAN                 bUseAlpha;                // Alpha
KMBOOLEAN                 bIgnoreTextureAlpha;      // Ignore Texture Alpha
KMFLIPMODE                FlipUV;                   // Flip U,V
KMCLAMPMODE               ClampUV;                  // Clamp U,V
KMFILTERMODE              FilterMode;               // Texture Filter
KMBOOLEAN                 bSuperSample;             // Anisotoropic Filter
```

```
KMDWORD                   MipMapAdjust;             // Mipmap D Adjust
KMTEXTURESHADINGMODE      TextureShadingMode;       // Texture Shading Mode
KMBOOLEAN                 bColorClamp;              // Color Clamp
KMDWORD                   PaletteBank;              // Palette Bank

/* for Texture Control Bits/Address */
PKMSURFACEDESC            pTextureSurfaceDesc;      // Texture DESC Pointer

/* FaceColor Setting for Intensity */
KMFLOAT                   fFaceColorAlpha;          // Face Color Alpha
KMFLOAT                   fFaceColorRed;            // Face Color Red
KMFLOAT                   fFaceColorGreen;          // Face Color Green
KMFLOAT                   fFaceColorBlue;           // Face Color Blue

/* Specular Highlight Specification for Intensity */
KMFLOAT                   fOffsetColorAlpha;        // Specular Alpha
KMFLOAT                   fOffsetColorRed;          // Specular Red
KMFLOAT                   fOffsetColorGreen;        // Specular Green
KMFLOAT                   fOffsetColorBlue;         // Specular Blue

/* Internal Variables */
KMDWORD                   GLOBALPARAMBUFFER;        // Global Parameter Buffer
KMDWORD                   ISPPARAMBUFFER;           // ISP Parameter Buffer
KMDWORD                   TSPPARAMBUFFER;           // TSP Parameter Buffer
KMDWORD                   TexturePARAMBUFFER;       // TextureParameter Buffer

/* for ModifierInstruction */
KMDWORD             ModifierInstruction;      /* ModifierInstruction*/
KMFLOAT             fBoundingBoxXmin;         /* BoundingBoxXmin(ShadowVolume) */
KMFLOAT             fBoundingBoxYmin;         /* BoundingBoxYmin(ShadowVolume) */
KMFLOAT             fBoundingBoxXmax;         /* BoundingBoxXmax(ShadowVolume) */
KMFLOAT             fBoundingBoxYmax;         /* BoundingBoxYmax(ShadowVolume) */

/* Added on Ver.1.30 */
KMBOOLEAN                 bDCalcExact;              // D-param calc
KMSTRIPLENGTH             StripLength               // Strip Length
KMUSERCLIPMODE            UserClipMode              // UserClip Mode

} KMVERTEXCONTEXT, *PKMVERTEXCONTEXT;
```

5.4 PACKED 32-BIT COLORS

```
typedef union _tagKMPACKEDARGB
{
	KMDWORD	dwPacked;
	struct {
		BYTE	bBlue;
		BYTE	bGreen;
		BYTE	bRed;
		BYTE	bAlpha;
	}byte;
}KMPACKEDARGB,*PKMPACKEDARGB;
```

5.5 PALETTE DEFINITION STRUCTURE

```
typedef union _tagKMPALETTEDATA
{
	struct {
		KMWORD wDummy;
		KMWORD wPaletteData;	/* for 16 bpp */
	}16bpp;
	KMDWORD dwPaletteData;		/* for 32 bpp */
}KMPALETTEDATA,*PKMPALETTEDATA;
```

6. TEXTURE FORMAT

6.1 TEXTURE FORMATS SUPPORTED BY ARC1 AND CLX1/2

The ARC1 and CLX1/2 supports the following textures. ARC1 in the figure indicates the texture supported by both ARC1 and CLX1/2. CLX1 indicates the texture supported only by CLX1/2. X indicates a format that is not supported.

Texture format			ARGB 1555	RGB 565	ARGB 4444	YUV 422	Bump	ARGB 8888
Twiddled	Non VQ	Square	ARC1	ARC1	ARC1	CLX1	CLX1	X
		Square Mipmap	ARC1	ARC1	ARC1	CLX1	CLX1	X
		Rectangle	CLX1	CLX1	CLX1	CLX1	CLX1	X
		Palettized (4 or 8 bpp)	CLX1	CLX1	CLX1	X	X	CLX1
		Palettized Mipmap (4 or 8 bpp)	CLX1	CLX1	CLX1	X	X	CLX1
		Palettized Rectangle (4 or 8 bpp)	CLX1	CLX1	CLX1	X	X	CLX1
	VQ	Sqare	ARC1	ARC1	ARC1	X	X	X
		Square Mipmap	ARC1	ARC1	ARC1	X	X	X
	Small VQ	Sqare	ARC1	ARC1	ARC1	X	X	X
		Square Mipmap	ARC1	ARC1	ARC1	X	X	X
Scan Order	Non VQ	Rectangle	ARC1	ARC1	ARC1	CLX1	CLX1	X
		Stride	ARC1	ARC1	ARC1	CLX1	CLX1	X
	VQ	Square	X	X	X	X	X	X
		Square Mipmap	X	X	X	X	X	X
	Small VQ	Square	X	X	X	X	X	X
		Square Mipmap	X	X	X	X	X	X

The ARGB8888 color uses 32 bits (4 bytes) for each pixel. The other colors use 16 bits (2 bytes) for each pixel.

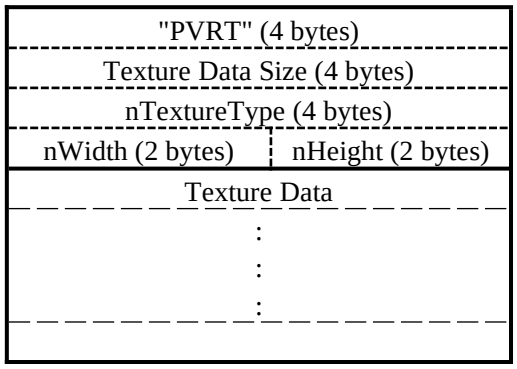
The number of bits of the index for Palettized 4 bpp is 4.
The number of bits of the index for Palettized 8 bpp is 8.
The number of bits of the index in VQ or small VQ format is 8.

When the texture is input or output (kmLoadTexture, kmLoadVQCodebook, kmReloadMipmap, kmLoadYUVTexture, or kmGetTexture), if the start address of texture data in system memory is on a 32-byte boundary, and its size is a multiple of 32 bytes, the DMA mode is used to transfer the texture data to texture memory, so that high-speed transfer is possible.

6.2 ARC1 AND CLX1/2 TEXTURE FORMATS

6.2.1 Texture Format of KAMUI

The texture format of KAMUI is the pixel data of a texture with headers (4 x 32 bits) indicating size and type appended. The headers are not referenced by KAMUI. Instead, KAMUI identifies the format of a texture based on information on the texture surface descriptor at the load destination specified by a load function. The headers are appended data for high-level applications such as Ninja.
When specifying the address of a texture by using a load function of KAMUI, skip the headers and specify the first address of pixel data.
If the start address of this pixel data portion is on a 32-byte boundary, and its size is a multiple of 32 bytes, the DMA mode is used to transfer the texture data to texture memory, so that high-speed transfer becomes possible. KAMUI defines an appropriate number of dummy bytes to make the size of the pixel data portion a multiple of 32 bytes.



"PVRT"

The KAMUI texture starts with four half-width characters "PVRT", which constitute an identifier indicating a KAMUI texture.

Texture Data Size

Saves the number of bytes of pixel data of the texture + 8. If two or more texture files are successively synthesized into one file, the first position of the next texture can be checked by using this value.

nTextureType

Specifies the format of the texture by using a category code and pixel format. The following enum values are ORed and specified (see the description of `kmCreateTextureSurface`).

Note that the higher 16 bits of this field are reserved for Ninja.

Category codes

KM_TEXTURE_TWIDDLED(0x00000100)

Texture in Twiddled-Non VQ-Square format

Texture of special pixel array used for PCX1/PCX2

KM_TEXTURE_TWIDDLED_MM(0x00000200)

Twiddled-Non VQ-Square format with mipmap

KM_TEXTURE_TWIDDLED_RECTANGLE(0x00000D00)

Twiddled-Non VQ-Rectangle format. Rectangular texture

Can be used with CLX1/2 only.

KM_TEXTURE_VQ(0x00000300)

Texture in Twiddled-VQ-Square format. Texture on which VQ (Vector Quantization) compression is performed. The first half of the texture data includes a code book table.

KM_TEXTURE_VQ_MM(0x00000400)

In Twiddled-VQ-Square format with mipmap

KM_TEXTURE_SMALLVQ(0x00000F00)

Texture in Twiddled-small-VQ-Square format. Texture on which VQ (Vector Quantization) compression is performed. The first half of the texture data includes a code book table.

KM_TEXTURE_SMALLVQ_MM(0x00001000)

In Twiddled-small-VQ-Square format with mipmap

KM_TEXTURE_PALETTIZE4(0x00000500)

KM_TEXTURE_PALETTIZE4_MM(0x00000600)

Texture in Twiddled-NonVQ-Palettized format

Palettized texture with index of 4 bpp (16 colors). Because only one palette can be used in a system, texture data does not include palette information. To set a palette, use `kmSetPaletteMode` or `kmSetPaletteData`.

KM_TEXTURE_PALETTIZE8(0x00000700)

KM_TEXTURE_PALETTIZE8_MM(0x00000800)

Palettized texture with an index of 8 bpp (256 colors) in Twiddled-Non VQ-Palettized format

KM_TEXTURE_RECTANGLE(0x00000900)

Rectangular texture in Scan Order-Rectangle format. The arrangement of the pixel data conforms to Windows BMP format.

KM_TEXTURE_STRIDE(0x00000B00)

Rectangular texture in Scan Order-Stride format. The arrangement of the pixel data conforms to Windows BMP format.

KM_TEXTURE_BMP(0x00000E00)

Input-dedicated format for MIPMAP automatic creation/dithering of `kmLoadTexture`. The texture data is ABGR8888 conforming to the pixel data in Windows BMP format (`24 bpp`). In this case, specifying the pixel format in a header is not necessary.

The texture in this format is Twiddled (or Twiddled Mipmap) when read by `kmLoadTexture`, and the pixel format is `RGB565`, `ARGB1555`, or `ARGB4444`. Specify the pixel format by `kmCreateTextureSurface`.

Pixel format codes

KM_TEXTURE_ARGB1555 (0x00000000)

Format consisting of one bit of alpha value and five bits of RGB values
The alpha value indicates transparent when it is 0 and opaque when it is 1.

KM_TEXTURE_RGB565 (0x00000001)

Format without alpha value and consisting of five bits of RB values and six bits of G value

KM_TEXTURE_ARGB4444 (0x00000002)

Format consisting of four bits of alpha value and four bits of RGB values.
The alpha value indicates completely transparent when it is 0x0 and completely opaque when it is 0xF.

KM_TEXTURE_YUV422 (0x00000003)

YUV422 format

KM_TEXTURE_BUMP (0x00000004)

Specifies a texture for bump mapping

ARGB8888 color can be specified in Palettized format only. In this case, it is set for a palette by kmPaletteMode API instead of nTextureType.

nWidth, nHeight

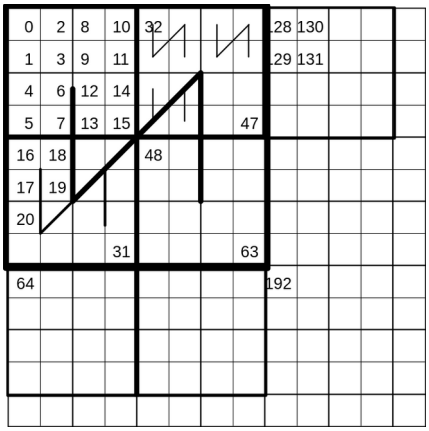
Specifies the horizontal and vertical sizes of a texture. The horizontal and vertical sizes of a texture must be 8, 16, 32, 64, 128, 256, 512, or 1,024.
For details, see the description of `kmCreateTextureSurface`.

Texture Data

Pixel data of a texture. Basically, conforms to the image in the texture memory of ARC1 and CLX1/2.
The address of the texture-specified parameter `pTexture` of the texture-related API of KAMUI is the first address of this portion. Note that the header may be skipped.
The start address of the pixel data should preferably be on a 32-byte boundary.
For details of the pixel data, see the following sections.

6.2.2 Twiddled Format and Twiddled Mipmap Format

Twiddled-NonVQ-Square and Twiddled-NonVQ-Square Mipmap formats are usually called Twiddled and Twiddled Mipmap formats. Twiddled format is the basic format of the PowerVR texture. Because this format executes texture filtering such as Bilinear filter at high speeds, addressing is optimized by hardware. Therefore, pixels in Twiddled format **must not be arranged in raster order**. The pixel order in Twiddled format is as shown below. Because Twiddled format is 16 bpp, it must be doubled to obtain the actual byte addresses.



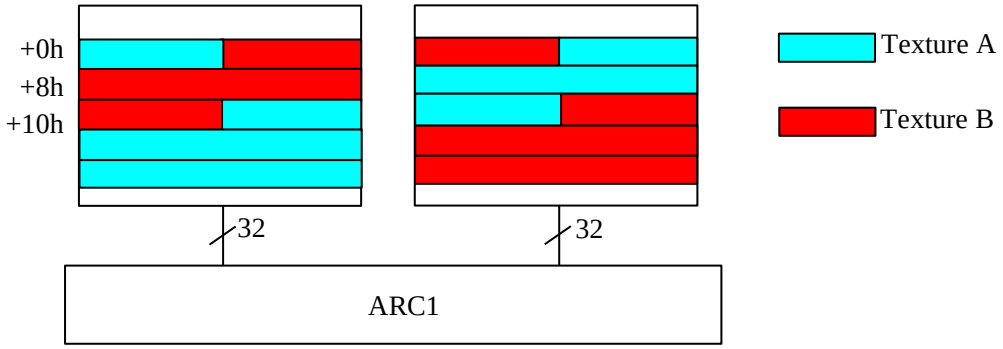
Twiddled format is as follows in KAMUI texture format:

+00h	KAMUI texture header (10h bytes)
+10h	Texture pixel data

The size of the pixel data of a texture is as follows (with ARC1, Twiddled format must be always a square).

Vertical/horizontal size	Pixel data size
8 x 8	80h bytes
16 x 16	200h bytes
32 x 32	800h bytes
64 x 64	2000h bytes
128 x 128	8000h bytes
256 x 256	20000h bytes
512 x 512	80000h bytes
1,024 x 1,024	200000h bytes

With ARC1, the mipmap texture in Twiddled format is specially arranged in memory. ARC1 has a 64-bit texture data bus. This bus is divided into two 32-bit banks and arranged so that each mipmap level is interleaved. To enhance the efficiency of the memory, two sets of textures are usually interleaved and registered as one set. The level of a mipmap is 1 x 1, i.e., the lowest level is registered first. An example of interleaving is shown below.



The byte address image in ARC1 viewed from the PCI/SH4 is as follows:

Bank 0

+00h	Dummy zero (6h bytes)
+06h	Texture A 1 x 1 map (2h bytes)
+08h	Texture B 2 x 2 map (8h bytes)
+10h	Texture A 4 x 4 map (20h bytes)
+30h	Texture B 8 x 8 map (80h bytes) :

Bank 1

+00h	Dummy zero (6h bytes)
+06h	Texture B 1 x 1 map (2h bytes)
+08h	Texture A 2 x 2 map (8h bytes)
+10h	Texture B 4 x 4 map (20h bytes)
+30h	Texture A 8 x 8 map (80h bytes) :

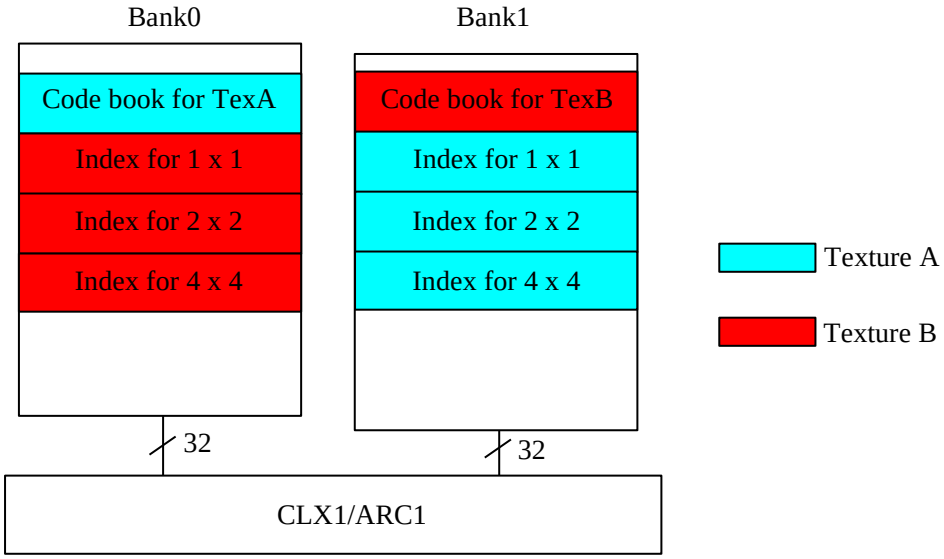
In the above example, two textures, A and B, are interleaved and located. With KAMUI, the start address of a texture is always aligned with an eight-byte boundary.
With KAMUI, the bank size is set to 4 MB. Where the start address of a 1 x 1 mipmap of texture A written into Bank0 is 00000006h, the start address of a 1 x 1 mipmap of texture B is 00400006h.

The Twiddled mipmap is interleaved and located only with ARC1. Because KAMUI absorbs the differences between the ARC1 and CLX1, interleaving is unnecessary in KAMUI texture format. To maintain compatibility with SGL, two dummy bytes are provided after each header.
In KAMUI texture format, the Twiddled mipmap format is as follows (Twiddled mipmap format must always be a square).

+00h	KAMUI texture header (10h bytes)
+10h	Dummy zero (2h bytes)
+12h	1 x 1 mipmap texture (2h bytes)
+14h	2 x 2 mipmap texture (8h bytes)
+1Ch	4 x 4 mipmap texture (20h bytes)
+3Ch	8 x 8 mipmap texture (80h bytes)
+BCh	16 x 16 mipmap texture (200h bytes)
+2BCh	32 x 32 mipmap texture (800h bytes)
+ABCh	64 x 64 mipmap texture (2000h bytes)
+2ABCh	128 x 128 mipmap texture (8000h bytes)
+AABCh	256 x 256 mipmap texture (20000h bytes)
+2AABCh	512 x 512 mipmap texture (80000h bytes)
+AAABCh	1,024 x 1,024 mipmap texture (200000h bytes)

6.2.3 VQ Format and VQ Mipmap Format

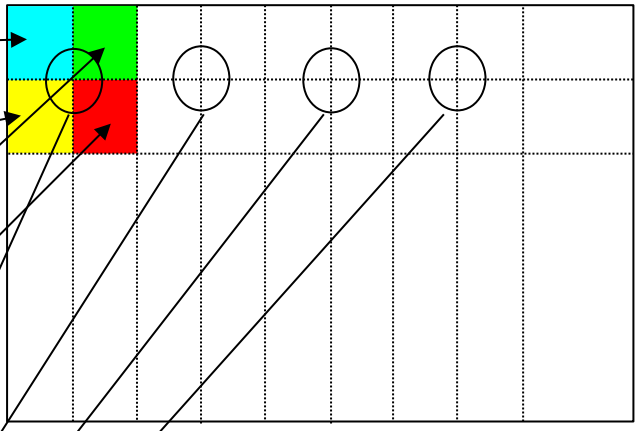
Twiddled-VQ-Square and Twiddled-VQ-Square-Mipmap formats are usually called VQ/VQ Mipmap formats.
VQ (Vector Quantization) format is a compressed texture format with a high compression rate. VQ format largely consists of two areas. One is a color table called a code book, while the other is index data that indicates the position of this code book. The VQ format structure is very close to the palette texture in that it expands index data by a code book and creates an image. However, this code book can be set for each texture.
In VQ format, each pixel index is arranged in Twiddled format. With the ARC1, the code book and index are interleaved in memory to enable high-speed access. The memory mapping if mipmap is effected is shown below (if mipmap is not effected, only the index of the registered size is registered).



A texture of VQ format consists of a code book and index data. The size of a code book is always 256 entries. Each entry contains data for four texels.

Code book

+0	Pixel A Low
+1	Pixel A High
+2	Pixel B Low
+3	Pixel B High
+4	Pixel C Low
+5	Pixel C High
+7	Pixel D Low
+8	Pixel D High
…….	…….
+7FE	Code book end



Index (Bank opposite to code book)

+0	Index#0
+1	Index#1
+2	Index#2
+3	Index#3
+4	
+5	
+7	
+8	
…….	…….

If a texture of 256 x 256 is compressed by VQ, the size is reduced to about 1/7, as follows:

(256 x 256 x 16) / ((128 x 128 x 8) + (256 x 16 x 4)) = 7.1:1

↑ Index ↑ Code book

The byte address image in ARC1 viewed from the PCI/SH4 is as follows:

[VQ]

	Bank 0		Bank 1	
+00h	Texture A Code book (800h bytes)		+00h	Texture B Code book (800h bytes)
+800h	Texture B Index		+800h	Texture A Index

[VQ Mipmap]

	Bank 0		Bank 1	
+00h	Texture A Code book (800h bytes)		+00h	Texture B Code book (800h bytes)
+800h	Dummy zero (1h byte)		+800h	Dummy zero (1h byte)
+801h	Texture B 2 x 2 mipmap index (1h byte)		+801h	Texture A 2 x 2 mipmap index (1h byte)
+802h	Texture B 4 x 4 mipmap index (4h bytes)		+802h	Texture A 4 x 4 mipmap index (4h bytes)
+806h	Texture B 8 x 8 mipmap index (10h bytes)		+806h	Texture A 8 x 8 mipmap index (10h bytes)
+816h	Texture B 16 x 16 mipmap index (40h bytes) :		+816h	Texture A 16 x 16 mipmap index (40h bytes) :
+???h	Dummy zero (Ah byte)		+???h	Dummy zero (Ah byte)

Note that, unlike in Twiddled format, the mipmap of each level is not interleaved. In VQ, the index of a 1 x 1 mipmap does not exist.

With KAMUI, the bank size is set to 4 MB. If the start address of a 2 x 2 mipmap of texture B written into Bank 0 is 00000801h, the start address of a 2 x 2 mipmap of texture A is 00400801h.

The code book is interleaved only with ARC1.

In KAMUI texture format, it is possible to safely ignore VQ interleaving.

The VQ format of KAMUI is as follows (VQ format must be always a square).

+00h	KAMUI texture header (10h bytes)
+10h	Code book (800h bytes)
+810h	Index

The size of the index of a texture is as follows:

Vertical/horizontal size	Index size
8 x 8	10h bytes[1]
16 x 16	40h bytes
32 x 32	100h bytes
64 x 64	400h bytes
128 x 128	1000h bytes
256 x 256	4000h bytes
512 x 512	10000h bytes
1,024 x 1,024	40000h bytes

The VQ mipmap format of KAMUI is as follows (VQ mipmap format must always be a square).

+00h	KAMUI texture header (10h bytes)
+10h	Code book (800h bytes)
+810h	Dummy zero (1h byte)
+811h	2 x 2 mipmap index (1h bytes)
+812h	4 x 4 mipmap index (4h bytes)
+816h	8 x 8 mipmap index (10h bytes)
+826h	16 x 16 mipmap index (40h bytes)
+866h	32 x 32 mipmap index (100h bytes)
+966h	64 x 64 mipmap index (400h bytes)
+D66h	128 x 128 mipmap index (1000h bytes)
+1D66h	256 x 256 mipmap index (4000h bytes)
+5D66h	512 x 512 mipmap index (10000h bytes)
+15D66h	1,024 x 1,024 mipmap index (40000h bytes)
+nnnh	Dummy zero (Ah byte)

Dummy data of 1 byte between the code book and index data or of 10 bytes after the index data is necessary for VQ mipmap texture.

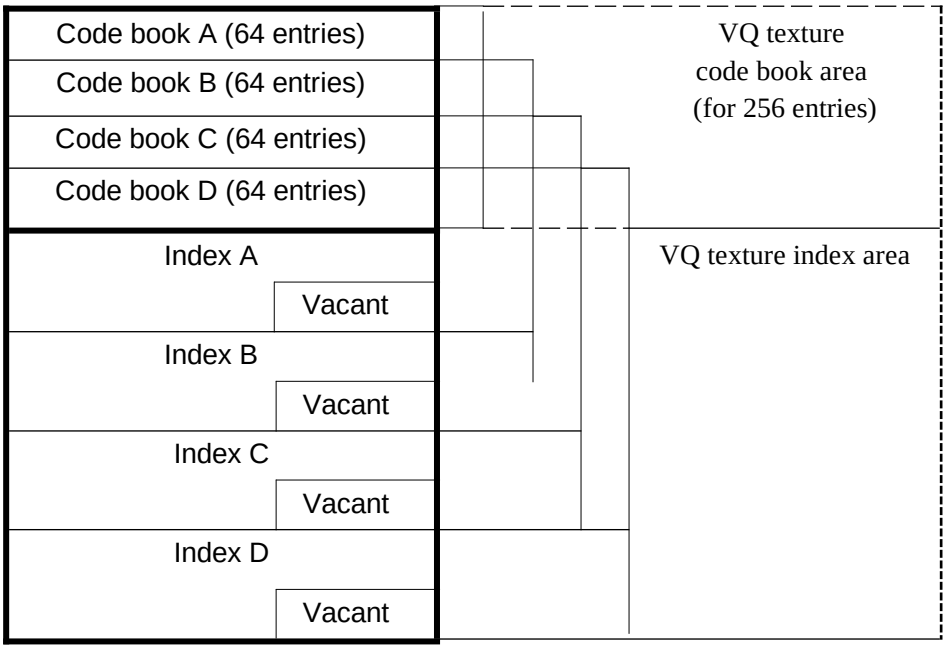
[1] The texture surface is allocated in 32-byte units, 8 x 8 VQ texture index data consumes 32 (20h) bytes of texture memory.

6.2.4 Small VQ and Small VQ Mipmap Formats

Twiddled-small-VQ-Square and Twiddled-small-VQ-Square-Mipmap formats are usually called the small VQ and small VQ Mipmap formats, respectively.

With the VQ/VQ Mipmap formats (described earlier), the compression ratio becomes low if the vertical and horizontal texture sizes are smaller than 64 x 64 texels, thus requiring a texture surface that is larger than that required by the Twiddled format, because the code book size is fixed to 256 entries (2,048 bytes). To avoid this problem, a VQ-compressed texture is prepared by reducing the code book size to make it match the vertical and horizontal texture sizes. This texture is called a small VQ texture.

The basic structure of the small VQ texture is the same as that in the VQ/VQ Mipmap format. For the VQ texture in the PowerVR hardware, the code book size can be selected from 16, 32, 64, 128, and 256 entries. The relative distance between the start address of the code book section and the start address of the index section must always be 2,048 bytes. So, a small-sized VQ texture can be defined by grouping (combining) several VQ textures having a small size, as shown below. This is a small VQ texture.



In this example, the code book is divided into four 64-entry sections, into each of which a small VQ texture is loaded. Each texture is referenced by specifying the start address of the corresponding code book.

To increase the memory use efficiency for the small VQ texture, it is necessary to minimize the vacant area in the index section shown in the above diagram. To satisfy this requirement, the index size must be smaller than or equal to the small VQ code book size. So, a maximum memory use efficiency can be attained with the following combinations of small VQ texture and code book sizes.

Size	Mip Map	Code book size		Index size (Bytes)	Vacant area size	Number of textures that can be grouped
		Entry	Bytes			
8 x 8	No	16	128 (80h)	16 (10h)	112 (70h)	16
	Yes	16	128 (80h)	32 (20h)	96 (60h)	16
16 x 16	No	16	128 (80h)	64 (40h)	64 (40h)	16
	Yes	16	128 (80h)	96 (60h)	32 (20h)	16
32 x 32	No	32	256 (100h)	256 (100h)	0	8
	Yes	64	512 (200h)	352 (160h)	160 (A0h)	4
64 x 64	No	128	1,024 (400h)	1,024 (400h)	0	2
	Yes	256	2,048 (800h)	1,376 (560h)	672 (2A0h)	1

As can be seen from the table, a combination of 32 x 32 or 64 x 64 and no-mipmap achieves the highest memory use efficiency (because the vacant area size is zero), and a combination of 64 x 64 and mipmap lowers the memory use efficiency (because the required code book and index sizes become larger than that of an ordinary VQ texture).

The total number of pixels in an 8 x 8 texture is 64. So, compression takes effect only if the code book is smaller than 16 entries. The PowerVR does no support a code book smaller than 16 entries, so using the small VQ format for 8 x 8 textures cannot take advantage of the compression. Similarly, the size of an 8 x 8 texture in the small VQ format combined with mipmap becomes larger than an 8 x 8 texture in the Twiddled format.

KAMUI does not support the combinations indicated by shading in the above table, because in these combinations, the Twiddled or ordinary VQ format offers a higher memory use efficiency than the small VQ format.

KAMUI groups small VQ textures having the same size when saving them into memory. The number of textures having the same size that are assembled into one group affects the memory use efficiency. The same texture memory capacity ([256 + 256]*8 = 4,096 bytes) is reserved for one 32 x 32 no-mipmap texture and eight 32 x 32 no-mipmap textures; however, kmGetFreeTextureMem offers the appropriate capacity for both cases.
When using small VQ textures, pay attention to how many are used simultaneously. The highest efficiency can be achieved by using a multiple of the quantity stated in the "Number of textures that can be grouped" column.

KAMUI performs the grouping of textures automatically when they are loaded.
The following tables list the small VQ formats prepared by the application program.

16 x 16 small VQ file format

+00h	KAMUI texture header (10h bytes)
+10h	Code book (80h bytes)
+90h	Index (40h bytes)

32 x 32 small VQ file format

+00h	KAMUI texture header (10h bytes)
+10h	Code book (100h bytes)
+110h	Index (100h bytes)

64 x 64 small VQ file format

+00h	KAMUI texture header (10h bytes)
+10h	Code book (400h bytes)
+410h	Index (400h bytes)

The following tables list the small VQ mipmap formats.

16 x 16 small VP mipmap

+00h	KAMUI texture header (10h bytes)
+10h	Code book (80h bytes)
+90h	Dummy zero (1h byte)
+91h	2 x 2 mipmap index (1h bytes)
+92h	4 x 4 mipmap index (4h bytes)
+96h	8 x 8 mipmap index (10h bytes)
+A6h	16 x 16 mipmap index (40h bytes)
+E6h	Dummy zero (Ah byte)

32 x 32 small VQ mipmap

+00h	KAMUI texture header (10h bytes)
+10h	Code book (200h bytes)
+210h	Dummy zero (1h byte)
+211h	2 x 2 mipmap index (1h bytes)
+212h	4 x 4 mipmap index (4h bytes)
+216h	8 x 8 mipmap index (10h bytes)
+226h	16 x 16 mipmap index (40h bytes)
+266h	32 x 32 mipmap index (100h bytes)
+366h	Dummy zero (Ah byte)

6.2.5 Palettized 4-bpp/8-bpp Format

Twiddled-NonVQ-Palettized format is usually called Palettized format.
There are two types of Palettized format: 4 bpp and 8 bpp. A system has only one palette with a total of 1,024 entries. For Palettized-4 bpp, the 1,024 entries are divided into 64 banks (1,024 entries/16 colors = 64 banks). For Palettized-8 bpp, the 1,024 entries are divided into 4 banks (1,024 entries/256 colors = 4 banks). Each bank is not physically separated, but is created by obtaining a pointer to the 1,024 entries through calculation. The 4 bpp texture and 8 bpp texture can exist together in one scene, but the overlapping entries of the 1,024 entries are shared. Therefore, changing the contents of the palette affects both the 4 bpp and 8 bpp textures.

The bank of the palette can be specified in VERTEX (polygon) units. A bank number is specified by PaletteBank member of KMVERTEXCONTEXT. The entry that can actually be used is selected as follows using the palette bank number (PaletteBank) and index value (index_data) of each texel of a texture.

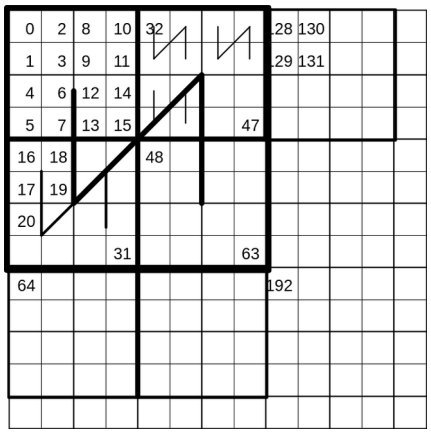
```
if (PixelFormat == 8BPP)
{
   palette_entry = (PaletteBank << 4) & 0x300 + index_data;
}

if (PixelFormat == 4BPP)
{
   palette_entry = (PaletteBank << 4) + index_data;
}
```

A value of 0 to 63 can be specified as PaletteBank in the case of 4 bpp. It is also 0 to 63 for 8 bpp. In this case, however, only the higher 2 bits of the 6 bits are valid, and only four types of values can be used: 0 (0 to 15), 16 (16 to 31), 32 (32 to 47), and 48 (48 to 63).

The format in memory is almost the same as the Twiddled format. However, four dots constitute 1 (16-bit) word for 4 bpp, and four dots constitute 2 (16-bit) words for 8 bpp.

The format of a texture is as follows:



The address sequence dose not differ from that of Twiddled. This is packed in little Endian, in the order of (U = 0, V = 0), (U = 0, V = 1), (U = 1, V = 0), (U = 1, V = 1). Therefore, the address sequence is as follows in the case of 4 bpp and 8 bpp (Palettized/Palettized mipmap format must always be a square).

Texel arrangement for 4 bpp

Bits 15-12	Bits 11-8	Bits 7-4	Bits 3-0
Texel U = 1 V = 1	Texel U = 1 V = 0	Texel U = 0 V = 1	Texel U = 0 V = 0

Texel arrangement for 8 bpp

Bits 15-8	Bits 7-0
Texel U = 0 V = 1	Texel U = 0 V = 0

Palettized format in KAMUI texture format is as follows:

+00h	KAMUI texture header (10h bytes)
+10h	Texture pixel data

The size of the pixel data of a texture is as follows:

Vertical/horizontal size	Pixel data size	
	4 bpp	8 bpp
8 x 8	20h bytes	40h
16 x 16	80h bytes	100h
32 x 32	200h bytes	400h
64 x 64	800h bytes	1000h
128 x 128	2000h bytes	4000h
256 x 256	8000h bytes	10000h
512 x 512	20000h bytes	40000h
1,024 x 1,024	80000h bytes	100000h

The Palettized mipmap format in KAMUI texture format is as follows:

4 bpp			8 bpp	
Offset	Length	Data	Length	Offset
+00h	10h	KAMUI texture header	10h	+00h
+10h	1h	Dummy zero	3h	+10h
+11h	1h	1 x 1 mipmap index	1h	+13h
+12h	2h	2 x 2 mipmap index	4h	+14h
+14h	8h	4 x 4 mipmap index	10h	+18h
+1Ch	20h	8 x 8 mipmap index	40h	+28h
+3Ch	80h	16 x 16 mipmap index	100h	+68h
+BCh	200h	32 x 32 mipmap index	400h	+168h
+2BCh	800h	64 x 64 mipmap index	1000h	+568h
+ABCh	2000h	128 x 128 mipmap index	4000h	+1568h
+2ABCh	8000h	256 x 256 mipmap index	10000h	+5568h
+AABCh	20000h	512 x 512 mipmap index	40000h	+15568h
+2AABCh	80000h	1,024 x 1,024 mipmap index	100000h	+55568h
+????h	14h	Dummy zero	8h	+????

In the Palettized 4-bpp mipmap format, index data must be preceded by one byte of dummy data and followed by 20 bytes of dummy data. Similarly, in the Palettized 8-bpp mipmap format, index data must be preceded by three bytes of dummy data and followed by eight bytes of dummy data.

With ARC1, a palettized texture cannot be used.

6.2.6 Rectangle Format

Scan Order-Rectangle format is usually called Rectangle format. Note that, with the CLX1/2, Twiddled-NonVQ-Rectangle format texture also exists.
Rectangle format is a texture that can set different values for the U and V sizes. When this format is used, mipmap cannot be effected and the performance drops relative to that in Twiddled format.
In Rectangle format, however, access is extremely easy because pixel data are arranged in raster order (complicated addressing does not have to be performed, unlike in Twiddled format). Rectangle format can be subject to rendering[1] and, therefore, can be used for environment mapping.

The vertical/horizontal size of the texture that can be specified in Rectangle format is 8, 16, 32, 64, 128, 256, 512, or 1,024.

Rectangle format in KAMUI texture format is as follows:

+00h	KAMUI texture header (10h bytes)
+10h	Texture pixel data

Like the location of pixel data in the Windows BMP (24 bpp), 2-byte pixel data of (U = 0, V = 0) (lower left of texture) is first located, followed by (U = 1, V = 0), (U = 2, V = 0), ... (U = Umax, V = 0), (U = 0, V = 1) and so on.

The size of the pixel data of a texture can be calculated as follows:

Size =
Number of horizontal texels (U-size) x Number of vertical texels (V-size) x 2 (bytes per texel)

[1] However, the frame buffer must be the same as the color mode (1555, 4444, 565) specified for the texture in 16 bpp mode.

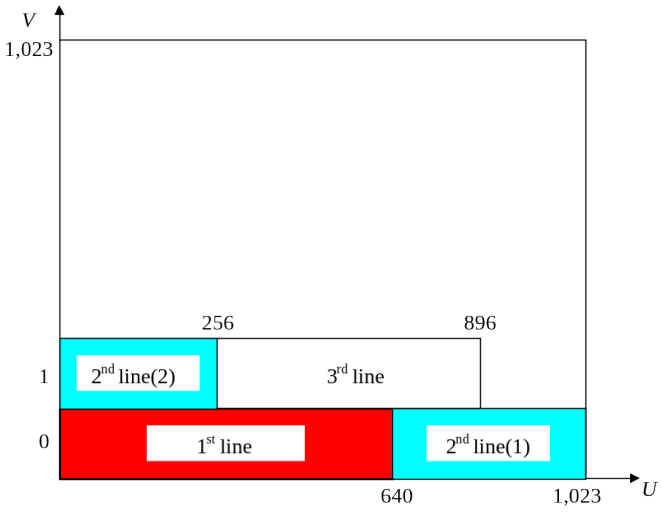
6.2.7 Stride Format

Scan Order-Stride format is usually called Stride format.
Stride format is a special type of Rectangle format. First, a global Stride value is set, after which texel is determined by using the following addressing.

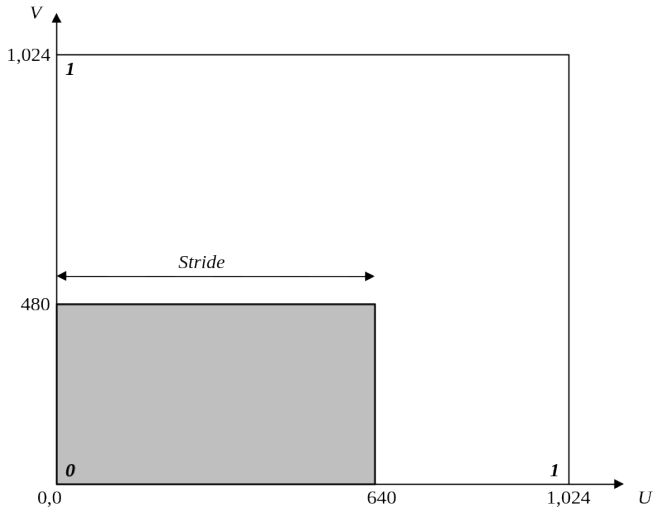
Addr = u + v*stride

Therefore, the number of horizontal texels of a texture can be varied by the Stride value. Note, however, that a Stride value must be a multiple of 32 (see the description of `kmSetStrideWidth`).

For example, when creating 640 x 480 areas in a texture of 1,024 x 1,024 when environment mapping is used, specify 640 as the Stride value. When rendering is performed on this texture surface by using `kmRenderTexture`, the portion on the first one line ((x,y) = (0,0)-(639,0)) on the screen is written to (U,V) = (0,0)-(639,0) of the texture surface, and the portion on the second line ((x,y) = (0,1)-(639,1) is written to (U,V) = (640,0)-(1023,0) and (0,1)-(255,1) of the texture surface.



When performing texture mapping by using this texture, the entire screen can be pasted as a texture where (U,V) = (0.0f,0.0f)-(0.625f,0.46875f).

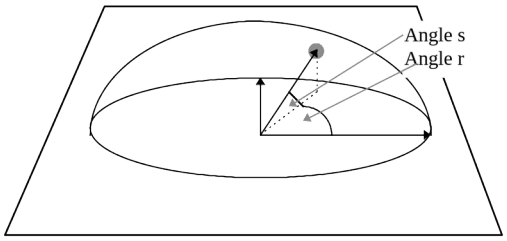


The Stride format of KAMUI is exactly the same as Rectangle format. The only difference is the flag of the texture type that is specified for nTextureType of a header.

6.2.8 BUMP-Mapping Format

With PVR2, BUMP-Mapping is implemented by the following technique. BUMP-Mapping information expresses projections and recesses on a surface by storing a normal vector as texture data, instead of pixel data of a normal texture. In addition, light source data is set to a bit of offset color, and the inner product of each dot is calculated based on this light source data. Therefore, **BUMP-Mapping can be used only with a polygon with texture offset**.
Within PVR2, a polar coordinate system is used for calculating BUMP-Mapping. The normal vector stored in a texture is expressed as follows:

$$\begin{aligned} x_s &= \cos(s')\cos(r') \\ y_s &= \sin(s') \\ z_s &= \cos(s')\sin(r') \end{aligned} \quad \text{where} \quad \begin{aligned} s' &= \pi/2 \frac{s}{256} \\ r' &= 2\pi \frac{r}{256} \end{aligned}$$



s and r are stored in the texture. In this case, s and r are 8-bit data and indicate an angle of elevation and azimuth angle of the BUMP data in the texture. Since the texture of the PVR2 is 16 bits long, one set of s and r exists per texel.
The light source vector for the polygon to which this texture is pasted is also expressed by using polar coordinates.

$$\begin{aligned} x_l &= \cos(t')\cos(q') \\ y_l &= \sin(t') \\ z_l &= \cos(t')\sin(q') \end{aligned}$$

In this case, t and q are expressed as follows, like the normal vector in a texture:

$$t' = \pi/2 \frac{t}{256}$$
$$q' = 2\pi \frac{q}{256}$$

In introducing this light source vector to a polygon, a Scale Factor is introduced. This Scale Factor is hereafter called Strength because it indicates the intensity of the light source.
By using Strength, the following K1, K2, and K3 are calculated.

$$k_1 = 1 - strength$$
$$k_2 = strength.\sin(t')$$
$$k_3 = strength.\cos(t')$$

Strength and K value are 8-bit values normalized to 1.
To give this light source information to a polygon, the following 32-bit offset color data is used.

Base color: xRGB			
K_1	K_2	K_3	q'

To use a Floating Color VertexType such as VertexType5, the above data is normalized to 1 and input. To use Packed Color VertexType such as VertexType4, it must be converted into 8-bit data and packed.
The ultimate brightness of each texel is calculated from the above texture normal vector and light source normal vector by the inner product.

$$\text{DotProduct} = \begin{bmatrix} x_s \\ y_s \\ z_s \end{bmatrix} . \begin{bmatrix} x_l \\ y_l \\ z_l \end{bmatrix} = x_s x_l + y_s y_l + z_s z_l$$

No further textures can be pasted to a polygon on which BUMP-Mapping has been performed. To paste a texture, the fact that translucent auto sort is GreaterEqual with the CLX1/2 is used and translucent polygon at the same coordinates as the BUMP-Mapped polygon are registered in duplicate. In this way, a texture can be pasted to the BUMP-Mapped polygon.

How to use a bump map

To make KAMUI display polygons using a bump map, make the following setting. This setting is only an example. Other settings may be able to offer diverse video representations.

Drawing polygons with only a bump map pasted

- Specifying a texture surface
  Specify a pixel format by setting nTextureType to KM_TEXTURE_BUMP. To generate a texture surface for a twiddled mipmap bump texture, for example, specify the following:
  nTextureType = KM_TEXTURE_TWIDDLED_MM|KM_TEXTURE_BUMP
- Setting VERTEXCONTEXT
  ShadingMode = KM_TEXTUREFLAT
  TextureShadingMode = KM_DECAL_ALPHA
  bUseSpecular = TRUE
  ShadingMode may be set to KM_TEXTUREGOURAUD, in which case, however, the base color applies gouraud. A luminance change made by a bump map is akin to flat shading.
- Vertex format
  It is necessary to use a vertex format that supports the specification of an offset color, such as TYPE03, TYPE04, TYPE11, or TYPE12.
- Specifying a vertex color
  A bump map is generated as a white texture where roughness is represented using an α value. If the vertex color is black, and KM_DECAL_ALPHA is specified, the roughness is represented in gray scale. The vertex color specified for polygons is added as shading to the bump. The use of a bump map imposes flat shading. So, the color of the third or subsequent vertex becomes effective.
- Specifying an offset color
  As stated above, the offset color of a polygon must be specified using parameters K1, K2, K3, and Q (in the stated order) for indicating the direction in which light strikes the polygon. The use of a bump map imposes flat shading. So, the offset color (K1, K2, K3, or Q) of the third or subsequent vertex becomes effective.

Overlapping a bump map with other textures

Using a translucent polygon can overlap a bump map with another texture. In this case, a polygon with a bump map pasted and a polygon with another texture pasted are drawn by overlapping them with each other in the same coordinate system. Either polygon can be translucent. The resulting picture display varies depending on which polygon is translucent and how the blend mode is specified.

The polygon with a bump map pasted is specified in the same way as stated earlier. The translucent polygon is specified in the same way as ordinary translucent polygons, as follows:

ListType = KM_TRANS_POLYGON
bUseAlpha = TRUE
TextureShadingMode = KM_MODULATE_ALPHA
SRCBlendingMode, DSTBlendingMode = Any blending mode

6.2.9 KAMUI Bit Map Format

Because KAMUI creates a texture on which mipmap and dither have automatically been performed when a texture is read by using `kmLoadTexture` API, it defines a texture in "KAMUI bit map format (KAMUI-BMP)" as its input format. The pixel format of this texture is ABGR8888 (note that the location is not ARGB) in conformance with pixel data in Windows BMP format (`24 bpp`).
(The Holly (CLX1/2) version of KAMUI does not support automatic mipmap and dither. So, it cannot use the KAMUI bit map format. See the descriptions about `kmLoadTexture` for details.)

The KAMUI bit map format in KAMUI texture format is as follows:

+00h	KAMUI texture header (10h bytes)
+10h	U = 0,V = 0 pixel data (Alpha) (1h byte)
+11h	U = 0,V = 0 pixel data (Blue) (1h byte)
+12h	U = 0,V = 0 pixel data (Green) (1h byte)
+13h	U = 0,V = 0 pixel data (Red) (1h byte)
+14h	U = 1,V = 0 pixel data (Alpha) (1h byte) : :

In the same manner as the Windows BMP (24 bpp) format, pixel data of 4 bytes of (U = 0, V = 0) (lower left of texture) are located first, followed by (U = 1, V = 0), (U = 2, V = 0), ... (U = Umax, V = 0), (U = 0, V = 1), and so on.

For automatic creation of mipmap and dither, a work area of the same capacity as in the Twiddled format is necessary in stack. Therefore, the vertical and horizontal size of a texture in bit map format handled by KAMUI is up to 512 texels

KAMUI bit map texture is converted into Twiddled/Twiddled Mipmap when it is read by `kmLoadTexture`. The pixel format at this time is converted into `KM_TEXTURE_ARGB1555`, `KM_TEXTURE_RGB565`, or `KM_TEXTURE_ARGB4444` in accordance with the setting of the surface at the load destination. When creating the load destination texture surface of KAMUI bit map texture by using `kmCreateTextureSurface`, specify `KM_TEXTURE_TWIDDLED` or `KM_TEXTURE_TWIDDLED_MM` as a category code, and `KM_TEXTURE_ARGB1555`, `KM_TEXTURE_RGB565`, or `KM_TEXTURE_ARGB4444` as a pixel format.

8. INDEX

G

I

K

L

M

N

O

P

R

S