# 纹理格式确定及数据上传

## OpenGL 层teximage

- 分析限定条件openglcore, opengl, 非压缩纹理，纹理为非代理纹理

```
/**
 * 实现所有的 glTexImage1D/2D/3D 函数以及 glCompressedTexImage1D/2D/3D 函数的通用代码。
 * \param compressed 仅在调用 glCompressedTexImage1D/2D/3D 时为 GL_TRUE。
 * \param format 用户的图像格式（仅在非压缩情况下使用）。
 * \param type 用户的图像类型（仅在非压缩情况下使用）。
 * \param imageSize 仅在调用 glCompressedTexImage1D/2D/3D 时使用。
 */
static ALWAYS_INLINE void
teximage(struct gl_context *ctx, GLboolean compressed, GLuint dims,
         GLenum target, GLint level, GLint internalFormat,
         GLsizei width, GLsizei height, GLsizei depth,
         GLint border, GLenum format, GLenum type,
         GLsizei imageSize, const GLvoid *pixels, bool no_error)
{ const char *func = compressed ? "glCompressedTexImage" : "glTexImage";
   struct gl_pixelstore_attrib unpack_no_border;
   const struct gl_pixelstore_attrib *unpack = &ctx->Unpack;
    ...

    // 进行格式转换检验
   if (compressed) {
      /* 对于 glCompressedTexImage()，驱动程序在纹理格式上没有选择余地，
       * 因为我们永远不会重新编码用户的压缩图像数据。internalFormat 在前面已经检查过了。
       */
      texFormat = _mesa_glenum_to_compressed_format(internalFormat);
   }
   else {
        // 选择非压缩纹理格式的格式
      texFormat = _mesa_choose_texture_format(ctx, texObj, target, level,
                                              internalFormat, format, type);
   }

   if (_mesa_is_proxy_texture(target)) {
      ...
   }
   else {
      const GLuint face = _mesa_tex_target_to_face(target);
      struct gl_texture_image *texImage;
      _mesa_lock_texture(ctx, texObj);
      {
         texImage = _mesa_get_tex_image(ctx, texObj, target, level);

         if (!texImage) {
            _mesa_error(ctx, GL_OUT_OF_MEMORY, "%s%uD", func, dims);
         }
         else {
            ctx->Driver.FreeTextureImageBuffer(ctx, texImage);
```

```
            _mesa_init_teximage_fields(ctx, texImage,
                                       width, height, depth,
                                       border, internalFormat, texFormat);

            /* 将纹理交给驱动程序。 <pixels> 可能为 null。 */
            if (width > 0 && height > 0 && depth > 0) {
               if (compressed) {
                  ctx->Driver.CompressedTexImage(ctx, dims, texImage,
                                                 imageSize, pixels);
               }
               else {
                  ctx->Driver.TexImage(ctx, dims, texImage, format,
                                       type, pixels, unpack);
                        [jump state_tracker st_TexImage]
               }
            }

            check_gen_mipmap(ctx, target, texObj, level);

            _mesa_update_fbo_texture(ctx, texObj, face, level);

            _mesa_dirty_texobj(ctx, texObj);
         }
      }
      _mesa_unlock_texture(ctx, texObj);
   }
}
```

- 该接口首先将参数格式类型转换为mesa内的mesa_format， 然后调用st_TexImage上传

## 压缩纹理格式的mesa_format格式确定

- 对于压缩格式通过_mesa_glenum_to_compressed_format

## 非压缩纹理mesa_format 格式的确定

非压缩纹理格式通过_mesa_choose_texture_format选择相应的mesa格式

```
mesa_format
_mesa_choose_texture_format(struct gl_context *ctx,
                            struct gl_texture_object *texObj,
                            GLenum target, GLint level,
                            GLenum internalFormat, GLenum format, GLenum type)
{
   mesa_format f;

   f = ctx->Driver.ChooseTextureFormat(ctx, target, internalFormat,
                                       format, type);
                [jump state_tracker st_ChooseTextureFormat]
   assert(f != MESA_FORMAT_NONE);
   return f;
}
```

# state_tracker

## 纹理图像上传相关接口

```
void
st_init_texture_functions(struct dd_function_table *functions)
{
   functions->ChooseTextureFormat = st_ChooseTextureFormat;
   functions->QueryInternalFormat = st_QueryInternalFormat;
   functions->TexImage = st_TexImage;
   functions->TexSubImage = st_TexSubImage;
   functions->CompressedTexSubImage = st_CompressedTexSubImage;
   functions->CopyTexSubImage = st_CopyTexSubImage;
   ...
}
```

## 非压缩纹理格式选择 st_ChooseTextureFormat

```
/**
 * Called via ctx->Driver.ChooseTextureFormat().
 */
mesa_format
st_ChooseTextureFormat(struct gl_context *ctx, GLenum target,
                       GLint internalFormat,
                       GLenum format, GLenum type)
{
   struct st_context *st = st_context(ctx);
   enum pipe_format pFormat;
   mesa_format mFormat;
   unsigned bindings;
   bool is_renderbuffer = false;
   enum pipe_texture_target pTarget;

   if (target == GL_RENDERBUFFER) {
      pTarget = PIPE_TEXTURE_2D;
      is_renderbuffer = true;
   } else {
      pTarget = gl_target_to_pipe(target);
   }

   if (target == GL_TEXTURE_1D || target == GL_TEXTURE_1D_ARRAY) {
      /* We don't do compression for these texture targets because of
       * difficulty with sub-texture updates on non-block boundaries, etc.
       * So change the internal format request to an uncompressed format.
       */
      internalFormat =
        _mesa_generic_compressed_format_to_uncompressed_format(internalFormat);
   }
```

```
   /* GL textures may wind up being render targets, but we don't know
    * that in advance.  Specify potential render target flags now for formats
    * that we know should always be renderable.
    */
   bindings = PIPE_BIND_SAMPLER_VIEW;
   if (_mesa_is_depth_or_stencil_format(internalFormat))
      bindings |= PIPE_BIND_DEPTH_STENCIL;
   else if (is_renderbuffer || internalFormat == 3 || internalFormat == 4 ||
            internalFormat == GL_RGB || internalFormat == GL_RGBA ||
            internalFormat == GL_RGB8 || internalFormat == GL_RGBA8 ||
            internalFormat == GL_BGRA ||
            internalFormat == GL_RGB16F ||
            internalFormat == GL_RGBA16F ||
            internalFormat == GL_RGB32F ||
            internalFormat == GL_RGBA32F)
      bindings |= PIPE_BIND_RENDER_TARGET;

   pFormat = st_choose_format(st, internalFormat, format, type,
                              pTarget, 0, 0, bindings, GL_TRUE);

   if (pFormat == PIPE_FORMAT_NONE && !is_renderbuffer) {
      /* try choosing format again, this time without render target bindings */
      pFormat = st_choose_format(st, internalFormat, format, type,
                                 pTarget, 0, 0, PIPE_BIND_SAMPLER_VIEW,
                                 GL_TRUE);
   }

   if (pFormat == PIPE_FORMAT_NONE) {
      mFormat = _mesa_glenum_to_compressed_format(internalFormat);
      if (st_compressed_format_fallback(st, mFormat))
          return mFormat;

      /* no luck at all */
      return MESA_FORMAT_NONE;
   }

   mFormat = st_pipe_format_to_mesa_format(pFormat);

   return mFormat;
}


/**
 * 给定用于纹理或表面的 OpenGL internalFormat 值，返回最匹配的 PIPE_FORMAT_x，
 * 如果没有匹配，则返回 PIPE_FORMAT_NONE。例如，在 glTexImage2D 中调用此函数。
 *
 * bindings 参数通常设置了 PIPE_BIND_SAMPLER_VIEW，以及如果希望具有渲染到纹理的能力，则还设置了
 * PIPE_BINDING_RENDER_TARGET 或 PIPE_BINDING_DEPTH_STENCIL。
 *
 * \param internalFormat 传递给 glTexImage2D 的用户值
 * \param target PIPE_TEXTURE_x 中的一个
 * \param bindings PIPE_BIND_x 标志的位掩码。
 * \param allow_dxt 表示是否可以返回 DXT 格式。这只在 internalFormat 命名了通用或特定压缩格式时才重要。
```

```
 *                  这应该仅在从 gl[Copy]TexImage() 中调用时发生。
 */
enum pipe_format
st_choose_format(struct st_context *st, GLenum internalFormat,
                 GLenum format, GLenum type,
                 enum pipe_texture_target target, unsigned sample_count = 0,
                 unsigned storage_sample_count = 0,
                 unsigned bindings, boolean allow_dxt= true)
{
   struct pipe_screen *screen = st->pipe->screen;
   unsigned i;
   int j;
   enum pipe_format pf;

   // 处理internalFormat=rgb, rgba
   pf = find_exact_format(internalFormat, format, type);
   if (pf != PIPE_FORMAT_NONE &&
       screen->is_format_supported(screen, pf, target, sample_count,
                                   storage_sample_count, bindings)) {
      goto success;
   }

   /* For an unsized GL_RGB but a 2_10_10_10 type, try to pick one of the
    * 2_10_10_10 formats.  This is important for
    * GL_EXT_texture_type_2_10_10_10_EXT support, which says that these
    * formats are not color-renderable.  Mesa's check for making those
    * non-color-renderable is based on our chosen format being 2101010.
    */
   if (type == GL_UNSIGNED_INT_2_10_10_10_REV) {
      if (internalFormat == GL_RGB)
         internalFormat = GL_RGB10;
      else if (internalFormat == GL_RGBA)
         internalFormat = GL_RGB10_A2;
   }

   /* search table for internalFormat */
   for (i = 0; i < ARRAY_SIZE(format_map); i++) {
      const struct format_mapping *mapping = &format_map[i];
      for (j = 0; mapping->glFormats[j]; j++) {
         if (mapping->glFormats[j] == internalFormat) {
            // 找到一个支持该格式的pipe格式
            pf = find_supported_format(screen, mapping->pipeFormats,
                                       target, sample_count,
                                       storage_sample_count, bindings,
                                       allow_dxt);
------------------------------------------------------------------
                      uint i;
                      for (i = 0; formats[i]; i++) {
                         if (screen->is_format_supported(screen, formats[i],
target,
                                                          sample_count,
storage_sample_count,
                                                          bindings)) {
                                    [jump radeonsi si_is_format_supported]
                            if (!allow_dxt && util_format_is_s3tc(formats[i])) {
```

```
                                /* we can't return a dxt format, continue
searching */
                                continue;
                             }

                             return formats[i];
                          }
                       }
                       return PIPE_FORMAT_NONE;

             goto success;
          }
       }
    }
    _mesa_problem(NULL, "unhandled format!\n");
    return PIPE_FORMAT_NONE;
}
```

- st_ChooseTextureFormat首先取定pipe_target, bindings参数，然后根据st_choose_format 接口获取pipe_format,
- 之后通过st_pipe_format_to_mesa_format 获取mesa_format的格式，
- 在st_choose_format内部，首先对于internalFormat等于rgb和rgba的方式直接通过find_exact_format查表的方式找出对应的pipe_format格式后调用radeonsi接口进行格式支持判断，如果没有找到和对于其他内部格式则通过format_map搜索表的方式找出支持该内部格式的定义好的pipe_format的即和，然后在find_supported_format内部进行一一判断找到首个支持该格式类型后返回

## 对于内部格式GL_RGBA格式的处理

此时使用rgba8888_tbl 进行特殊处理

```
static const struct exact_format_mapping rgba8888_tbl[] =
{
   { GL_RGBA,     GL_UNSIGNED_INT_8_8_8_8,        PIPE_FORMAT_ABGR8888_UNORM },
   { GL_ABGR_EXT, GL_UNSIGNED_INT_8_8_8_8_REV,    PIPE_FORMAT_ABGR8888_UNORM },
   { GL_RGBA,     GL_UNSIGNED_INT_8_8_8_8_REV,    PIPE_FORMAT_RGBA8888_UNORM },
   { GL_ABGR_EXT, GL_UNSIGNED_INT_8_8_8_8,        PIPE_FORMAT_RGBA8888_UNORM },
   { GL_BGRA,     GL_UNSIGNED_INT_8_8_8_8,        PIPE_FORMAT_ARGB8888_UNORM },
   { GL_BGRA,     GL_UNSIGNED_INT_8_8_8_8_REV,    PIPE_FORMAT_BGRA8888_UNORM },
   { GL_RGBA,     GL_UNSIGNED_BYTE,               PIPE_FORMAT_R8G8B8A8_UNORM },
   { GL_ABGR_EXT, GL_UNSIGNED_BYTE,               PIPE_FORMAT_A8B8G8R8_UNORM },
   { GL_BGRA,     GL_UNSIGNED_BYTE,               PIPE_FORMAT_B8G8R8A8_UNORM },
   { 0,           0,                              0                          }
};
```

### 对于内部格式GL_RGB 的处理

使用格式映射表 rgbx8888_tbl 找到符合指定format和type的 PIPE_FORMAT

```
static const struct exact_format_mapping rgbx8888_tbl[] =
{
   { GL_RGBA,     GL_UNSIGNED_INT_8_8_8_8,        PIPE_FORMAT_XBGR8888_UNORM },
   { GL_ABGR_EXT, GL_UNSIGNED_INT_8_8_8_8_REV,    PIPE_FORMAT_XBGR8888_UNORM },
   { GL_RGBA,     GL_UNSIGNED_INT_8_8_8_8_REV,    PIPE_FORMAT_RGBX8888_UNORM },
   { GL_ABGR_EXT, GL_UNSIGNED_INT_8_8_8_8,        PIPE_FORMAT_RGBX8888_UNORM },
   { GL_BGRA,     GL_UNSIGNED_INT_8_8_8_8,        PIPE_FORMAT_XRGB8888_UNORM },
   { GL_BGRA,     GL_UNSIGNED_INT_8_8_8_8_REV,    PIPE_FORMAT_BGRX8888_UNORM },
   { GL_RGBA,     GL_UNSIGNED_BYTE,               PIPE_FORMAT_R8G8B8X8_UNORM },
   { GL_ABGR_EXT, GL_UNSIGNED_BYTE,               PIPE_FORMAT_X8B8G8R8_UNORM },
   { GL_BGRA,     GL_UNSIGNED_BYTE,               PIPE_FORMAT_B8G8R8X8_UNORM },
   { 0,           0,                              0                          }
};
```

## 对于非GL_RENDERBUFFER渲染目标可根据gl_target_to_pipe 总结如下target和pTarget的对应关系S

| GLenum | pipe_texture_target |
|---|---|
| GL_TEXTURE_1D | PIPE_TEXTURE_1D |
| GL_PROXY_TEXTURE_1D | PIPE_TEXTURE_1D |
| GL_TEXTURE_2D | PIPE_TEXTURE_2D |
| GL_PROXY_TEXTURE_2D | PIPE_TEXTURE_2D |
| GL_TEXTURE_EXTERNAL_OES | PIPE_TEXTURE_2D |
| GL_TEXTURE_2D_MULTISAMPLE | PIPE_TEXTURE_2D |
| GL_PROXY_TEXTURE_2D_MULTISAMPLE | PIPE_TEXTURE_2D |
| GL_TEXTURE_RECTANGLE_NV | PIPE_TEXTURE_RECT |
| GL_PROXY_TEXTURE_RECTANGLE_NV | PIPE_TEXTURE_RECT |
| GL_TEXTURE_3D | PIPE_TEXTURE_3D |
| GL_PROXY_TEXTURE_3D | PIPE_TEXTURE_3D |
| GL_TEXTURE_CUBE_MAP_ARB | PIPE_TEXTURE_CUBE |
| GL_PROXY_TEXTURE_CUBE_MAP_ARB | PIPE_TEXTURE_CUBE |
| GL_TEXTURE_CUBE_MAP_POSITIVE_X | PIPE_TEXTURE_CUBE |
| GL_TEXTURE_CUBE_MAP_NEGATIVE_X | PIPE_TEXTURE_CUBE |
| GL_TEXTURE_CUBE_MAP_POSITIVE_Y | PIPE_TEXTURE_CUBE |
| GL_TEXTURE_CUBE_MAP_NEGATIVE_Y | PIPE_TEXTURE_CUBE |

| GLenum | pipe_texture_target |
|---|---|
| GL_TEXTURE_CUBE_MAP_POSITIVE_Z | PIPE_TEXTURE_CUBE |
| GL_TEXTURE_CUBE_MAP_NEGATIVE_Z | PIPE_TEXTURE_CUBE |
| GL_TEXTURE_1D_ARRAY_EXT | PIPE_TEXTURE_1D_ARRAY |
| GL_PROXY_TEXTURE_1D_ARRAY_EXT | PIPE_TEXTURE_1 |

## 关于bindings的确定

首先对于深度或者模板内部格式则将绑定点指定为 PIPE_BIND_DEPTH_STENCIL

| 内部格式（`internalFormat`） | 绑定点（`bindings`） |
|---|---|
| OTHER | PIPE_BIND_SAMPLER_VIEW |
| 3 | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_RENDER_TARGET |
| 4 | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_RENDER_TARGET |
| GL_RGB | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_RENDER_TARGET |
| GL_RGBA | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_RENDER_TARGET |
| GL_RGB8 | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_RENDER_TARGET |
| GL_RGBA8 | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_RENDER_TARGET |
| GL_BGRA | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_RENDER_TARGET |
| GL_RGB16F | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_RENDER_TARGET |
| GL_RGBA16F | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_RENDER_TARGET |
| GL_RGB32F | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_RENDER_TARGET |
| GL_RGBA32F | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_RENDER_TARGET |
| GL_DEPTH_COMPONENT | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_DEPTH_STENCIL |
| GL_DEPTH_COMPONENT16 | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_DEPTH_STENCIL |
| GL_DEPTH_COMPONENT24 | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_DEPTH_STENCIL |
| GL_DEPTH_COMPONENT32 | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_DEPTH_STENCIL |
| GL_STENCIL_INDEX | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_DEPTH_STENCIL |
| GL_STENCIL_INDEX1_EXT | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_DEPTH_STENCIL |
| GL_STENCIL_INDEX4_EXT | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_DEPTH_STENCIL |
| GL_STENCIL_INDEX8_EXT | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_DEPTH_STENCIL |
| GL_STENCIL_INDEX16_EXT | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_DEPTH_STENCIL |
| GL_DEPTH_STENCIL_EXT | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_DEPTH_STENCIL |

| 内部格式(`internalFormat`) | 绑定点(`bindings`) |
|---|---|
| GL_DEPTH24_STENCIL8_EXT | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_DEPTH_STENCIL |
| GL_DEPTH_COMPONENT32F | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_DEPTH_STENCIL |
| GL_DEPTH32F_STENCIL8 | PIPE_BIND_SAMPLER_VIEW \| PIPE_BIND_DEPTH_STENCIL |

## TexImage /st_TexImage

st_TexImage 内部会调用st_TexSubImage， 之间流程见纹理分析

## TexSubImage / st_TexSubImage

```
static void
st_TexSubImage(struct gl_context *ctx, GLuint dims,
               struct gl_texture_image *texImage,
               GLint xoffset, GLint yoffset, GLint zoffset,
               GLint width, GLint height, GLint depth,
               GLenum format, GLenum type, const void *pixels,
               const struct gl_pixelstore_attrib *unpack)
{
   struct st_context *st = st_context(ctx);
   struct st_texture_image *stImage = st_texture_image(texImage);
   struct st_texture_object *stObj = st_texture_object(texImage->TexObject);
   struct pipe_context *pipe = st->pipe;
   struct pipe_screen *screen = pipe->screen;
   struct pipe_resource *dst = stImage->pt;
   struct pipe_resource *src = NULL;
   struct pipe_resource src_templ;
   struct pipe_transfer *transfer;
   struct pipe_blit_info blit;
   enum pipe_format src_format, dst_format;
   mesa_format mesa_src_format;
   GLenum gl_target = texImage->TexObject->Target;
   unsigned bind;
   GLubyte *map;
   unsigned dstz = texImage->Face + texImage->TexObject->MinLayer;
   unsigned dst_level = 0;
   bool throttled = false;

   // 清理缓存和无效的读像素缓存
   st_flush_bitmap_cache(st);
   st_invalidate_readpix_cache(st);

   // 如果纹理对象和纹理图像使用的是相同的底层资源，则更新目标层级
   if (stObj->pt == stImage->pt)
      dst_level = texImage->TexObject->MinLevel + texImage->Level;

   // 检查纹理格式是否支持 TexSubImage 操作
   assert(!_mesa_is_format_etc2(texImage->TexFormat) &&
          !_mesa_is_format_astc_2d(texImage->TexFormat) &&
          texImage->TexFormat != MESA_FORMAT_ETC1_RGB8);

   // 如果目标纹理为空，则使用回退操作
```

```
    if (!dst)
        goto fallback;

    // 尝试使用快速memcpy路径的texture_subdata
    // memcpy不涉及格式转换略过
    if (pixels &&
        !_mesa_is_bufferobj(unpack->BufferObj) &&
        _mesa_texstore_can_use_memcpy(ctx, texImage->_BaseFormat,
                                      texImage->TexFormat, format, type,
                                      unpack)) {
         ...
    }

    // 如果不偏好使用基于blit的纹理传输，则使用回退操作
    if (!st->prefer_blit_based_texture_transfer) {
        goto fallback;
    }

    // 深度-模板格式的回退，因为某些驱动程序中的stencil blit实现不完整
    if (format == GL_DEPTH_STENCIL) {
        goto fallback;
    }

    // 如果基本内部格式和纹理格式不匹配，则无法使用基于blit的TexSubImage
    if (texImage->_BaseFormat !=
        _mesa_get_format_base_format(texImage->TexFormat)) {
        goto fallback;
    }

    // 确定目标格式是否受支持
    if (format == GL_DEPTH_COMPONENT || format == GL_DEPTH_STENCIL)
        bind = PIPE_BIND_DEPTH_STENCIL;
    else
        bind = PIPE_BIND_RENDER_TARGET;

    dst_format = util_format_linear(dst->format);
    dst_format = util_format_luminance_to_red(dst_format);
    dst_format = util_format_intensity_to_red(dst_format);

    // 如果目标格式不受支持，则使用回退操作
    if (!dst_format ||
        !screen->is_format_supported(screen, dst_format, dst->target,
                                     dst->nr_samples, dst->nr_storage_samples,
                                     bind)) {
        goto fallback;
    }

    // 尝试使用PBO上传
    if (_mesa_is_bufferobj(unpack->BufferObj)) {
        if (try_pbo_upload(ctx, dims, texImage, format, type, dst_format,
                           xoffset, yoffset, zoffset,
                           width, height, depth, pixels, unpack))
            return;
    }

    // 如果纹理格式已经匹配，直接使用memcpy进行快速上传
```

```
if (_mesa_format_matches_format_and_type(texImage->TexFormat, format,
                                         type, unpack->SwapBytes, NULL)) {
   goto fallback;
}

// 选择源格式
src_format = st_choose_matching_format(st, PIPE_BIND_SAMPLER_VIEW,
                                       format, type, unpack->SwapBytes);
if (!src_format) {
   goto fallback;
}

mesa_src_format = st_pipe_format_to_mesa_format(src_format);

// 如果不能使用memcpy进行源纹理临时数据的上传，则使用回退操作
if (!_mesa_texstore_can_use_memcpy(ctx,
                          _mesa_get_format_base_format(mesa_src_format),
                          mesa_src_format, format, type, unpack)) {
   goto fallback;
}

// 对于立方体贴图，TexSubImage只设置单个立方体贴图面
if (gl_target == GL_TEXTURE_CUBE_MAP) {
   gl_target = GL_TEXTURE_2D;
}
// 对于立方体贴图数组，上传时需要使用2D数组
if (gl_target == GL_TEXTURE_CUBE_MAP_ARRAY) {
   gl_target = GL_TEXTURE_2D_ARRAY;
}

// 初始化源纹理的描述
memset(&src_templ, 0, sizeof(src_templ));
src_templ.target = gl_target_to_pipe(gl_target);
src_templ.format = src_format;
src_templ.bind = PIPE_BIND_SAMPLER_VIEW;
src_templ.usage = PIPE_USAGE_STAGING;

// 将OpenGL纹理维度转换为Gallium纹理维度
st_gl_texture_dims_to_pipe_dims(gl_target, width, height, depth,
                                &src_templ.width0, &src_templ.height0,
                                &src_templ.depth0, &src_templ.array_size);

// 检查非2的幂次方纹理是否受支持
if (!screen->get_param(screen, PIPE_CAP_NPOT_TEXTURES) &&
    (!util_is_power_of_two_or_zero(src_templ.width0) ||
     !util_is_power_of_two_or_zero(src_templ.height0) ||
     !util_is_power_of_two_or_zero(src_templ.depth0))) {
   goto fallback;
}

// 防止内存使用超限
util_throttle_memory_usage(pipe, &st->throttle,
                           width * height * depth *
                           util_format_get_blocksize(src_templ.format));
throttled = true;
```

```
    // 创建源纹理
    src = screen->resource_create(screen, &src_templ);
    if (!src) {
       goto fallback;
    }

    // 映射源纹理像素
    pixels = _mesa_validate_pbo_teximage(ctx, dims, width, height, depth,
                                         format, type, pixels, unpack,
                                         "glTexSubImage");
    if (!pixels) {
       // 这是一个GL错误
       pipe_resource_reference(&src, NULL);
       return;
    }

    // 转换为Gallium坐标
    if (gl_target == GL_TEXTURE_1D_ARRAY) {
       zoffset = yoffset;
       yoffset = 0;
       depth = height;
       height = 1;
    }

    // 映射源纹理内存
    map = pipe_transfer_map_3d(pipe, src, 0, PIPE_TRANSFER_WRITE, 0, 0, 0,
                               width, height, depth, &transfer);
    if (!map) {
       _mesa_unmap_teximage_pbo(ctx, unpack);
       pipe_resource_reference(&src, NULL);
       goto fallback;
    }

    // 上传像素数据（使用memcpy）
    {
       const uint bytesPerRow = width * util_format_get_blocksize(src_format);
       GLuint row, slice;

       for (slice = 0; slice < (unsigned) depth; slice++) {
          if (gl_target == GL_TEXTURE_1D_ARRAY) {
             // 1D数组纹理，需要将Gallium坐标转换为GL坐标
             void *src = _mesa_image_address2d(unpack, pixels,
                                                width, depth, format,
                                                type, slice, 0);
             memcpy(map, src, bytesPerRow);
          }
          else {
             ubyte *slice_map = map;

             for (row = 0; row < (unsigned) height; row++) {
                void *src = _mesa_image_address(dims, unpack, pixels,
                                                 width, height, format,
                                                 type, slice, row, 0);
                memcpy(slice_map, src, bytesPerRow);
                slice_map += transfer->stride;
             }
```

```
            }
            map += transfer->layer_stride;
        }
    }

    // 解除源纹理内存映射
    pipe_transfer_unmap(pipe, transfer);
    _mesa_unmap_teximage_pbo(ctx, unpack);

    // 填充blit
    ...

    // 执行blit操作
    st->pipe->blit(st->pipe, &blit);

    // 释放源纹理资源
    pipe_resource_reference(&src, NULL);
    return;

fallback:
    // 如果未进行内存节流，则进行内存节流
    if (!throttled) {
        util_throttle_memory_usage(pipe, &st->throttle,
                                   width * height * depth *
                                   _mesa_get_format_bytes(texImage->TexFormat));
    }
    // 执行回退操作
    _mesa_store_texsubimage(ctx, dims, texImage, xoffset, yoffset, zoffset,
                            width, height, depth, format, type, pixels,
                            unpack);
}
```

- 该接口内部首先根据格式类型等信息尝试memcpy形式的blit操作确定纹理，否则使用 *mesa_store_texsubimage 这种像素填充式存储操作,而在*mesa_store_texsubimage内部会进行格式转换,不过这里的格式并非和描述格式对应主要是格式转换时使用填充

## 使用_mesa_store_texsubimage进行纹理数据上传

```
/*
 * Fallback for Driver.TexSubImage().
 */
void
_mesa_store_texsubimage(struct gl_context *ctx, GLuint dims,
                        struct gl_texture_image *texImage,
                        GLint xoffset, GLint yoffset, GLint zoffset,
                        GLint width, GLint height, GLint depth,
                        GLenum format, GLenum type, const void *pixels,
                        const struct gl_pixelstore_attrib *packing)
{
    store_texsubimage(ctx, texImage,
                      xoffset, yoffset, zoffset, width, height, depth,
                      format, type, pixels, packing, "glTexSubImage");
}
```

```
/**
 * Helper function for storing 1D, 2D, 3D whole and subimages into texture
 * memory.
 * The source of the image data may be user memory or a PBO.  In the later
 * case, we'll map the PBO, copy from it, then unmap it.
 */
static void
store_texsubimage(struct gl_context *ctx,
                  struct gl_texture_image *texImage,
                  GLint xoffset, GLint yoffset, GLint zoffset,
                  GLint width, GLint height, GLint depth,
                  GLenum format, GLenum type, const GLvoid *pixels,
                  const struct gl_pixelstore_attrib *packing,
                  const char *caller)

{
   const GLbitfield mapMode = get_read_write_mode(format, texImage->TexFormat);
   const GLenum target = texImage->TexObject->Target;
   GLboolean success = GL_FALSE;
   GLuint dims, slice, numSlices = 1, sliceOffset = 0;
   GLint srcImageStride = 0;
   const GLubyte *src;

   switch (target) {
   case GL_TEXTURE_1D:
      dims = 1;
    ...
   }

   /* get pointer to src pixels (may be in a pbo which we'll map here) */
   // 如果
   src = (const GLubyte *)
      _mesa_validate_pbo_teximage(ctx, dims, width, height, depth,
                                  format, type, pixels, packing, caller);

        if (!_mesa_is_bufferobj(unpack->BufferObj)) {
           /* no PBO */
           return pixels;

        buf = (GLubyte *) ctx->Driver.MapBufferRange(ctx, 0,
                                                     unpack->BufferObj->Size,
                                                     GL_MAP_READ_BIT,
                                                     unpack->BufferObj,
                                                     MAP_INTERNAL);

        return ADD_POINTERS(buf, pixels);

   if (!src)
      return;

   /* compute slice info (and do some sanity checks) */
      numSlices = ...;
      sliceOffset = ...;
      height = ...;
      yoffset = ....;
      srcImageStride = ...;
```

```
   assert(numSlices == 1 || srcImageStride != 0);

   for (slice = 0; slice < numSlices; slice++) {
      GLubyte *dstMap;
      GLint dstRowStride;

      // 获取纹理对象资源的映射地址
      ctx->Driver.MapTextureImage(ctx, texImage,
                                  slice + sliceOffset,
                                  xoffset, yoffset, width, height,
                                  mapMode, &dstMap, &dstRowStride);
      if (dstMap) {
         /* Note: we're only storing a 2D (or 1D) slice at a time but we need
          * to pass the right 'dims' value so that GL_UNPACK_SKIP_IMAGES is
          * used for 3D images.
          */
         success = _mesa_texstore(ctx, dims, texImage->_BaseFormat,
                                  texImage->TexFormat,
                                  dstRowStride,
                                  &dstMap,
                                  width, height, 1,  /* w, h, d */
                                  format, type, src, packing);

         ctx->Driver.UnmapTextureImage(ctx, texImage, slice + sliceOffset);
      }

      src += srcImageStride;
   }
}


/**
 * Store user data into texture memory.
 * Called via glTex[Sub]Image1/2/3D()
 * \return GL_TRUE for success, GL_FALSE for failure (out of memory).
 */
GLboolean
_mesa_texstore(TEXSTORE_PARAMS)
{
   if (_mesa_texstore_memcpy(ctx, dims, baseInternalFormat,
                             dstFormat,
                             dstRowStride, dstSlices,
                             srcWidth, srcHeight, srcDepth,
                             srcFormat, srcType, srcAddr, srcPacking)) {
      return GL_TRUE;
   }

   if (_mesa_is_depth_or_stencil_format(baseInternalFormat)) {
      return texstore_depth_stencil(ctx, dims, baseInternalFormat,
                                    dstFormat, dstRowStride, dstSlices,
                                    srcWidth, srcHeight, srcDepth,
                                    srcFormat, srcType, srcAddr, srcPacking);
   } else if (_mesa_is_format_compressed(dstFormat)) {
      return texstore_compressed(ctx, dims, baseInternalFormat,
                                 dstFormat, dstRowStride, dstSlices,
```

```
                                    srcWidth, srcHeight, srcDepth,
                                    srcFormat, srcType, srcAddr, srcPacking);
   } else {
      return texstore_rgba(ctx, dims, baseInternalFormat,
                              dstFormat, dstRowStride, dstSlices,
                              srcWidth, srcHeight, srcDepth,
                              srcFormat, srcType, srcAddr, srcPacking);
   }
}
```

- 深度模板，压缩纹理无格式转换

### 对于rgaba纹理的存储上传中的内部格式转换

```
/**
 * This macro defines the (many) parameters to the texstore functions.
 * \param dims  either 1 or 2 or 3
 * \param baseInternalFormat  user-specified base internal format
 * \param dstFormat  destination Mesa texture format
 * \param dstX/Y/Zoffset  destination x/y/z offset (ala TexSubImage), in texels
 * \param dstRowStride  destination image row stride, in bytes
 * \param dstSlices  array of addresses of image slices (for 3D, array texture)
 * \param srcWidth/Height/Depth  source image size, in pixels
 * \param srcFormat  incoming image format
 * \param srcType  incoming image data type
 * \param srcAddr  source image address
 * \param srcPacking  source image packing parameters
 */
#define TEXSTORE_PARAMS \
    struct gl_context *ctx, GLuint dims, \
        MAYBE_UNUSED GLenum baseInternalFormat, \
        MAYBE_UNUSED mesa_format dstFormat, \
        GLint dstRowStride, \
        GLubyte **dstSlices, \
    GLint srcWidth, GLint srcHeight, GLint srcDepth, \
    GLenum srcFormat, GLenum srcType, \
    const GLvoid *srcAddr, \
    const struct gl_pixelstore_attrib *srcPacking


// dstFormat 为mesa_format, srcFormat为format, type 为原参数格式类型
static GLboolean
texstore_rgba(TEXSTORE_PARAMS)
{
   void *tempImage = NULL;
   int img;
   GLubyte *src, *dst;
   uint8_t rebaseSwizzle[4];
   bool transferOpsDone = false;

   /* 我们必须手动处理MESA_FORMAT_YCBCR，因为它是一种特殊情况，
    * _mesa_format_convert 不支持它。在这种情况下，我们只允许在YCBCR格式之间进行转换，
```

```
     * 它主要是一个memcpy操作。
     */
    if (dstFormat == MESA_FORMAT_YCBCR || dstFormat == MESA_FORMAT_YCBCR_REV) {
         ...
    }

    /* 我们必须手动处理GL_COLOR_INDEX，因为
     * _mesa_format_convert 不处理这种格式。因此，我们在这里的做法是
     * 先将其转换为RGBA ubyte，然后像往常一样从那里转换为dst。
     */
    if (srcFormat == GL_COLOR_INDEX) {
         ...
    } else if (srcPacking->SwapBytes) {
       /* 在调用 _mesa_format_convert 之前，我们必须处理字节交换的情况 */
       ...
    }

    int srcRowStride =
       _mesa_image_row_stride(srcPacking, srcWidth, srcFormat, srcType);

    uint32_t srcMesaFormat =
       _mesa_format_from_format_and_type(srcFormat, srcType);

    dstFormat = _mesa_get_srgb_format_linear(dstFormat);

    /* 如果我们有transferOps，那么我们需要先转换为RGBA float，
     * 然后应用transferOps，然后再转换为dst
     */
    void *tempRGBA = NULL;
    if (!transferOpsDone &&
        _mesa_texstore_needs_transfer_ops(ctx, baseInternalFormat, dstFormat)) {
        ....
    }

    src = (GLubyte *)
       _mesa_image_address(dims, srcPacking, srcAddr, srcWidth, srcHeight,
                           srcFormat, srcType, 0, 0, 0);

    bool needRebase;
    if (_mesa_get_format_base_format(dstFormat) != baseInternalFormat) {
       needRebase =
          _mesa_compute_rgba2base2rgba_component_mapping(baseInternalFormat,
                                                         rebaseSwizzle);
    } else {
       needRebase = false;
    }

    for (img = 0; img < srcDepth; img++) {
       _mesa_format_convert(dstSlices[img], dstFormat, dstRowStride,
                            src, srcMesaFormat, srcRowStride,
                            srcWidth, srcHeight,
                            needRebase ? rebaseSwizzle : NULL);
       src += srcHeight * srcRowStride;
    }
    return GL_TRUE;
 }
```

- 通过计算出源格式的像素首地址最终通过_mesa_format_convert将源格式的像素数据转换为目的格式通过MapTextureImage获取到的图像映射地址中，达到写入像素数据的结果
- 这里为了进行格式转换提出了一个新的mesa_format格式生成方法，即是通过_mesa_format_from_format_and_type ,这里的格式是为下一不格式转换时使用，作填充纹理数据用

**通过_mesa_format_from_format_and_type 生成格式**

```
/**
* 根据 OpenGL 的格式（GL_RGB、GL_RGBA等）、数据类型（GL_INT、GL_FLOAT等）返回相应的mesa_array_format 或普通的 mesa_format。
*
* 该函数通常用于从 GL 类型计算出 mesa 格式，以便调用 _mesa_format_convert。该函数不考虑字节交换，因此返回的类型假定不涉及字节交换。如果涉及字节交换，则客户端应在调用_mesa_format_convert 之前在其端处理。
*
* 该函数返回一个 uint32_t，可打包一个 mesa_format 或 mesa_array_format。客户端必须检查返回值上的 mesa 数组格式位（MESA_ARRAY_FORMAT_BIT），以确定返回的格式是 mesa_array_format还是 mesa_format。
*/
uint32_t
_mesa_format_from_format_and_type(GLenum format, GLenum type)
{
   bool is_array_format = true;
   uint8_t swizzle[4];
   bool normalized = false, is_float = false, is_signed = false;
   int num_channels = 0, type_size = 0;

   /* 从 OpenGL 数据类型中提取数组格式类型信息 */
   switch (type) {
   case GL_UNSIGNED_BYTE:
      type_size = 1;
      break;
   ....
      is_array_format = false;
      break;
   }

   /* 从 OpenGL 格式中提取数组格式的混合信息 */
   if (is_array_format)
      is_array_format = get_swizzle_from_gl_format(format, swizzle);

   /* 如果这是在检查数据类型和格式后的数组格式类型，创建数组格式 */
   if (is_array_format) {
      normalized = !_mesa_is_enum_format_integer(format);
      num_channels = _mesa_components_in_format(format);

      return MESA_ARRAY_FORMAT(type_size, is_signed, is_float,
                               normalized, num_channels,
                               swizzle[0], swizzle[1], swizzle[2], swizzle[3]);
   }

   /* 否则，这不是数组格式，因此返回与 OpenGL 格式和数据类型匹配的 mesa_format */
```

```
    switch (type) {
    // ... 省略了一些具体的格式映射 ...

    case GL_UNSIGNED_SHORT_8_8_MESA:
        if (format == GL_YCBCR_MESA)
            return MESA_FORMAT_YCBCR;
        break;
    case GL_UNSIGNED_SHORT_8_8_REV_MESA:
        if (format == GL_YCBCR_MESA)
            return MESA_FORMAT_YCBCR_REV;
        break;
    // ... 省略了一些具体的格式映射 ...

    default:
        break;
    }

    /* 如果运行到这里，意味着我们找不到与提供的 GL 格式/类型相匹配的 Mesa 格式。可能需要在这种情况下添加新的 Mesa 格式。 */
    unreachable("不支持的格式");
}
```

- 从这个函数可以看出对于 GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, GL_HALF_FLOAT, GL_HALF_FLOAT_OES,GL_FOAT 类型都当作MESA数组格式类型处理，而其他类型都当作非数组格式类型
- 对于数组格式类型

通过定义MESA_ARRAY_FORMAT返回对应的格式

```
/**
 * An enum useful to encode/decode information stored in a mesa_array_format
 */
enum {
    MESA_ARRAY_FORMAT_TYPE_IS_SIGNED = 0x4,
    MESA_ARRAY_FORMAT_TYPE_IS_FLOAT = 0x8,
    MESA_ARRAY_FORMAT_TYPE_NORMALIZED = 0x10,
    MESA_ARRAY_FORMAT_DATATYPE_MASK = 0xf,
    MESA_ARRAY_FORMAT_TYPE_MASK = 0x1f,
    MESA_ARRAY_FORMAT_TYPE_SIZE_MASK = 0x3,
    MESA_ARRAY_FORMAT_NUM_CHANS_MASK = 0xe0,
    MESA_ARRAY_FORMAT_SWIZZLE_X_MASK = 0x00700,
    MESA_ARRAY_FORMAT_SWIZZLE_Y_MASK = 0x03800,
    MESA_ARRAY_FORMAT_SWIZZLE_Z_MASK = 0x1c000,
    MESA_ARRAY_FORMAT_SWIZZLE_W_MASK = 0xe0000,
    MESA_ARRAY_FORMAT_BIT = 0x80000000
};


#define MESA_ARRAY_FORMAT(SIZE, SIGNED, IS_FLOAT, NORM, NUM_CHANS, \
      SWIZZLE_X, SWIZZLE_Y, SWIZZLE_Z, SWIZZLE_W) (                \
   (((SIZE >> 1)      ) & MESA_ARRAY_FORMAT_TYPE_SIZE_MASK) |      \
   (((SIGNED)    << 2 ) & MESA_ARRAY_FORMAT_TYPE_IS_SIGNED) |      \
   (((IS_FLOAT)  << 3 ) & MESA_ARRAY_FORMAT_TYPE_IS_FLOAT) |       \
```

```
    (((NORM)      << 4 ) & MESA_ARRAY_FORMAT_TYPE_NORMALIZED) |     \
    (((NUM_CHANS) << 5 ) & MESA_ARRAY_FORMAT_NUM_CHANS_MASK) |      \
    (((SWIZZLE_X) << 8 ) & MESA_ARRAY_FORMAT_SWIZZLE_X_MASK) |      \
    (((SWIZZLE_Y) << 11) & MESA_ARRAY_FORMAT_SWIZZLE_Y_MASK) |      \
    (((SWIZZLE_Z) << 14) & MESA_ARRAY_FORMAT_SWIZZLE_Z_MASK) |      \
    (((SWIZZLE_W) << 17) & MESA_ARRAY_FORMAT_SWIZZLE_W_MASK) |      \
    MESA_ARRAY_FORMAT_BIT)
```

- 对于非数组类型

| Type | Format | mesa_format |
|---|---|---|
| GL_UNSIGNED_SHORT_5_6_5 | GL_RGB | MESA_FORMAT_B5G6R5_UNORM |
| | GL_BGR | MESA_FORMAT_R5G6B5_UNORM |
| | GL_RGB_INTEGER | MESA_FORMAT_B5G6R5_UINT |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_UNSIGNED_SHORT_5_6_5_REV | GL_RGB | MESA_FORMAT_R5G6B5_UNORM |
| | GL_BGR | MESA_FORMAT_B5G6R5_UNORM |
| | GL_RGB_INTEGER | MESA_FORMAT_R5G6B5_UINT |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_UNSIGNED_SHORT_4_4_4_4 | GL_RGBA | MESA_FORMAT_A4B4G4R4_UNORM |
| | GL_BGRA | MESA_FORMAT_A4R4G4B4_UNORM |
| | GL_ABGR_EXT | MESA_FORMAT_R4G4B4A4_UNORM |
| | GL_RGBA_INTEGER | MESA_FORMAT_A4B4G4R4_UINT |
| | GL_BGRA_INTEGER | MESA_FORMAT_A4R4G4B4_UINT |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_UNSIGNED_SHORT_4_4_4_4_REV | GL_RGBA | MESA_FORMAT_R4G4B4A4_UNORM |
| | GL_BGRA | MESA_FORMAT_B4G4R4A4_UNORM |
| | GL_ABGR_EXT | MESA_FORMAT_A4B4G4R4_UNORM |
| | GL_RGBA_INTEGER | MESA_FORMAT_R4G4B4A4_UINT |
| | GL_BGRA_INTEGER | MESA_FORMAT_B4G4R4A4_UINT |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_UNSIGNED_SHORT_5_5_5_1 | GL_RGBA | MESA_FORMAT_A1B5G5R5_UNORM |
| | GL_BGRA | MESA_FORMAT_A1R5G5B5_UNORM |
| | GL_RGBA_INTEGER | MESA_FORMAT_A1B5G5R5_UINT |
| | GL_BGRA_INTEGER | MESA_FORMAT_A1R5G5B5_UINT |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_UNSIGNED_SHORT_1_5_5_5_REV | GL_RGBA | MESA_FORMAT_R5G5B5A1_UNORM |
| | GL_BGRA | MESA_FORMAT_B5G5R5A1_UNORM |
| | GL_RGBA_INTEGER | MESA_FORMAT_R5G5B5A1_UINT |
| | GL_BGRA_INTEGER | MESA_FORMAT_B5G5R5A1_UINT |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_UNSIGNED_BYTE_3_3_2 | GL_RGB | MESA_FORMAT_B2G3R3_UNORM |
| | GL_RGB_INTEGER | MESA_FORMAT_B2G3R3_UINT |
| --------------------------------- | -------------------------------- | ------------------------------ |

| Type | Format | mesa_format |
| --- | --- | --- |
| GL_UNSIGNED_BYTE_2_3_3_REV | GL_RGB | MESA_FORMAT_R3G3B2_UNORM |
| | GL_RGB_INTEGER | MESA_FORMAT_R3G3B2_UINT |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_UNSIGNED_INT_5_9_9_9_REV | GL_RGB | MESA_FORMAT_R9G9B9E5_FLOAT |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_UNSIGNED_INT_10_10_10_2 | GL_RGBA | MESA_FORMAT_A2B10G10R10_UNORM |
| | GL_RGBA_INTEGER | MESA_FORMAT_A2B10G10R10_UINT |
| | GL_BGRA | MESA_FORMAT_A2R10G10B10_UNORM |
| | GL_BGRA_INTEGER | MESA_FORMAT_A2R10G10B10_UINT |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_UNSIGNED_INT_2_10_10_10_REV | GL_RGB | MESA_FORMAT_R10G10B10X2_UNORM |
| | GL_RGBA | MESA_FORMAT_R10G10B10A2_UNORM |
| | GL_RGBA_INTEGER | MESA_FORMAT_R10G10B10A2_UINT |
| | GL_BGRA | MESA_FORMAT_B10G10R10A2_UNORM |
| | GL_BGRA_INTEGER | MESA_FORMAT_B10G10R10A2_UINT |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_UNSIGNED_INT_8_8_8_8 | GL_RGBA | MESA_FORMAT_A8B8G8R8_UNORM |
| | GL_BGRA | MESA_FORMAT_A8R8G8B8_UNORM |
| | GL_ABGR_EXT | MESA_FORMAT_R8G8B8A8_UNORM |
| | GL_RGBA_INTEGER | MESA_FORMAT_A8B8G8R8_UINT |
| | GL_BGRA_INTEGER | MESA_FORMAT_A8R8G8B8_UINT |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_UNSIGNED_INT_8_8_8_8_REV | GL_RGBA | MESA_FORMAT_R8G8B8A8_UNORM |
| | GL_BGRA | MESA_FORMAT_B8G8R8A8_UNORM |
| | GL_ABGR_EXT | MESA_FORMAT_A8B8G8R8_UNORM |
| | GL_RGBA_INTEGER | MESA_FORMAT_R8G8B8A8_UINT |
| | GL_BGRA_INTEGER | MESA_FORMAT_B8G8R8A8_UINT |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_UNSIGNED_SHORT_8_8_MESA | GL_YCBCR_MESA | MESA_FORMAT_YCBCR |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_UNSIGNED_SHORT_8_8_REV_MESA | GL_YCBCR_MESA | MESA_FORMAT_YCBCR_REV |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_UNSIGNED_INT_10F_11F_11F_REV | GL_RGB | MESA_FORMAT_R11G11B10_FLOAT |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_FLOAT | GL_DEPTH_COMPONENT | MESA_FORMAT_Z_FLOAT32 |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_UNSIGNED_INT | GL_DEPTH_COMPONENT | MESA_FORMAT_Z_UNORM32 |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_UNSIGNED_SHORT | GL_DEPTH_COMPONENT | MESA_FORMAT_Z_UNORM16 |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_UNSIGNED_INT_24_8 | GL_DEPTH_STENCIL | MESA_FORMAT_Z24_UNORM_S8_UINT |

| Type | Format | mesa_format |
| --- | --- | --- |
| --------------------------------- | -------------------------------- | ------------------------------ |
| GL_FLOAT_32_UNSIGNED_INT_24_8_REV | GL_DEPTH_STENCIL | MESA_FORMAT_Z32_FLOAT_S8X24_UINT |
| --------------------------------- | -------------------------------- | ------------------------------ |

## 关于MESA_FORMAT_YCBCR

MESA_FORMAT_YCBCR 是 Mesa 3D 图形库中用于表示 YCbCr 格式的一个特殊格式。YCbCr 是一种颜色编码方法，常用于视频压缩和广播电视中。在 YCbCr 中，Y 表示亮度（Luma），Cb 和 Cr 表示色度（Chrominance）。

具体而言，MESA_FORMAT_YCBCR 是 Mesa 3D 图形库中定义的一个格式，用于表示包含 Y、Cb 和 Cr 成分的图像数据。这种格式通常涉及到视频处理和纹理映射等方面的操作。在代码中，对于 MESA_FORMAT_YCBCR 的处理是一种特殊情况，需要手动进行处理，因为通常的格式转换方法 `_mesa_format_convert` 不支持这种格式，而是通过调用 `_mesa_texstore_ycbcr` 函数来处理。

总的来说，MESA_FORMAT_YCBCR 是用于表示 YCbCr 数据的一种图像格式。

# 获取纹理图像资源映射地址 st_MapTextureImage

该接口和st_MapRenderbuffer基本用法一致，都是通过tranfer_map获取地址

```
/** called via ctx->Driver.MapTextureImage() */
static void
st_MapTextureImage(struct gl_context *ctx,
                   struct gl_texture_image *texImage,
                   GLuint slice, GLuint x, GLuint y, GLuint w, GLuint h,
                   GLbitfield mode,
                   GLubyte **mapOut, GLint *rowStrideOut)
{
   struct st_context *st = st_context(ctx);
   struct st_texture_image *stImage = st_texture_image(texImage);
   GLubyte *map;
   struct pipe_transfer *transfer;

   /* Check for unexpected flags */
   assert((mode & ~(GL_MAP_READ_BIT |
                    GL_MAP_WRITE_BIT |
                    GL_MAP_INVALIDATE_RANGE_BIT)) == 0);

   const enum pipe_transfer_usage transfer_flags =
      st_access_flags_to_transfer_flags(mode, false);

   map = st_texture_image_map(st, stImage, transfer_flags, x, y, slice, w, h, 1,
                              &transfer);
   if (map) {
      if (st_compressed_format_fallback(st, texImage->TexFormat)) {
         /* Some compressed formats don't have to be supported by drivers,
          * and st/mesa transparently handles decompression on upload (Unmap),
          * so that drivers don't see the compressed formats.
          *
          * We store the compressed data (it's needed for glGetCompressedTex-
          * Image and image copies in OES_copy_image).
          */
```

```
        unsigned z = transfer->box.z;
        struct st_texture_image_transfer *itransfer = &stImage->transfer[z];

        unsigned blk_w, blk_h;
        _mesa_get_format_block_size(texImage->TexFormat, &blk_w, &blk_h);

        unsigned y_blocks = DIV_ROUND_UP(texImage->Height2, blk_h);
        unsigned stride = *rowStrideOut = itransfer->temp_stride =
           _mesa_format_row_stride(texImage->TexFormat, texImage->Width2);
        unsigned block_size = _mesa_get_format_bytes(texImage->TexFormat);

        *mapOut = itransfer->temp_data =
           stImage->compressed_data +
           (z * y_blocks + (y / blk_h)) * stride +
           (x / blk_w) * block_size;
        itransfer->map = map;
      }
      else {
        /* supported mapping */
        *mapOut = map;
        *rowStrideOut = transfer->stride;
      }
   }
   else {
      *mapOut = NULL;
      *rowStrideOut = 0;
   }
}
```

- 该接口返回该图像bo资源的映射地址

# RadeonSI

## 格式支持判断接口

```
/**
 * Check if the given pipe_format is supported as a texture or
 * drawing surface.
 * \param bindings  bitmask of PIPE_BIND_*
 */
boolean (*is_format_supported)( struct pipe_screen *,
                                   enum pipe_format format,
                                   enum pipe_texture_target target,
                                   unsigned sample_count,
                                   unsigned storage_sample_count,
                                   unsigned bindings );



void si_init_screen_state_functions(struct si_screen *sscreen)
{
    sscreen->b.is_format_supported = si_is_format_supported;
}
```

分析限定条件非多重采样 此时sample_count = 0, storage_sample_count = 0

```
static boolean si_is_format_supported(struct pipe_screen *screen,
                                      enum pipe_format format,
                                      enum pipe_texture_target target,
                                      unsigned sample_count =0,
                                      unsigned storage_sample_count = 0,
                                      unsigned usage)
{
    struct si_screen *sscreen = (struct si_screen *)screen;
    unsigned retval = 0;

    if (sample_count > 1) {
        ...
    }

    if (usage & (PIPE_BIND_SAMPLER_VIEW |
             PIPE_BIND_SHADER_IMAGE)) {
        if (target == PIPE_BUFFER) {
            retval |= si_is_vertex_format_supported(
                screen, format, usage & (PIPE_BIND_SAMPLER_VIEW |
                                 PIPE_BIND_SHADER_IMAGE));
        } else {
            if (si_is_sampler_format_supported(screen, format))
                retval |= usage & (PIPE_BIND_SAMPLER_VIEW |
                           PIPE_BIND_SHADER_IMAGE);
        }
    }

    if ((usage & (PIPE_BIND_RENDER_TARGET |
              PIPE_BIND_DISPLAY_TARGET |
              PIPE_BIND_SCANOUT |
              PIPE_BIND_SHARED |
              PIPE_BIND_BLENDABLE)) &&
        si_is_colorbuffer_format_supported(format)) {
        retval |= usage &
              (PIPE_BIND_RENDER_TARGET |
               PIPE_BIND_DISPLAY_TARGET |
               PIPE_BIND_SCANOUT |
               PIPE_BIND_SHARED);
        if (!util_format_is_pure_integer(format) &&
            !util_format_is_depth_or_stencil(format))
            retval |= usage & PIPE_BIND_BLENDABLE;
    }

    if ((usage & PIPE_BIND_DEPTH_STENCIL) &&
        si_is_zs_format_supported(format)) {
        retval |= PIPE_BIND_DEPTH_STENCIL;
    }

    if (usage & PIPE_BIND_VERTEX_BUFFER) {
        retval |= si_is_vertex_format_supported(screen, format,
```

```
                              PIPE_BIND_VERTEX_BUFFER);
    }

    if ((usage & PIPE_BIND_LINEAR) &&
        !util_format_is_compressed(format) &&
        !(usage & PIPE_BIND_DEPTH_STENCIL))
        retval |= PIPE_BIND_LINEAR;

    return retval == usage;
}
```

# 格式最终形式-描述符格式的确定

最终上层传入的参数格式的都要通过描述符下发，这里的格式值的商城通过si_is_format_supported确定的目的格式类型
描述符有专门的字段表示。
资源有缓冲和纹理两类
分别对应SQ_BUF_RSRC_WORD 和 SQ_IMG_RSRC_WORD
格式与描述符下发绑定，根据texture分析， 纹理资源存放在sampler_view中， 通过上层接口teximage,teximagestorage,传入，在此不作分析处理

## 缓冲纹理 SQ_BUF_RSRC_WORD

确定接口

```
/**
 * Build the sampler view descriptor for a buffer texture.
 * @param state 256-bit descriptor; only the high 128 bits are filled in
 */
void
si_make_buffer_descriptor(struct si_screen *screen, struct r600_resource *buf,
              enum pipe_format format,
              unsigned offset, unsigned size,
              uint32_t *state)
{
    const struct util_format_description *desc;
    int first_non_void;
    unsigned stride;
    unsigned num_records;
    unsigned num_format, data_format;

    desc = util_format_description(format);
           switch (format) {
           case PIPE_FORMAT_NONE:
              return &util_format_none_description;
           case PIPE_FORMAT_B8G8R8A8_UNORM:
              return &util_format_b8g8r8a8_unorm_description;
           ...

    first_non_void = util_format_get_first_non_void_channel(format);

    stride = desc->block.bits / 8;
```

```
    num_format = si_translate_buffer_numformat(&screen->b, desc, first_non_void);
    data_format = si_translate_buffer_dataformat(&screen->b, desc,
first_non_void);

    num_records = size / stride;
    num_records = MIN2(num_records, (buf->b.b.width0 - offset) / stride);
    else if (screen->info.chip_class == VI)
        num_records *= stride;

    state[4] = 0;
    state[5] = S_008F04_STRIDE(stride);
    state[6] = num_records;
    state[7] = S_008F0C_DST_SEL_X(si_map_swizzle(desc->swizzle[0])) |
           S_008F0C_DST_SEL_Y(si_map_swizzle(desc->swizzle[1])) |
           S_008F0C_DST_SEL_Z(si_map_swizzle(desc->swizzle[2])) |
           S_008F0C_DST_SEL_W(si_map_swizzle(desc->swizzle[3])) |
           S_008F0C_NUM_FORMAT(num_format) |
           S_008F0C_DATA_FORMAT(data_format);
}
```

这个util_format_description 定义在u_format_table.c文件中

- 这里首先通过util_format_description 找到pipe_format对应的 util_format_description 格式描述符结构形式，每个pipe_format都对应一个全局定义的静态结构体
- 之后就是通过util_foramtget函数获取这个util_format_description中的 channel, num_format, data_format字段
- 最后存入state[7]字段

## 普通纹理 SQ_IMG_RSRC_WORD

```
/**
 * Build the sampler view descriptor for a texture.
 */
void
si_make_texture_descriptor(struct si_screen *screen,
               struct si_texture *tex,
               bool sampler,
               enum pipe_texture_target target,
               enum pipe_format pipe_format,
               const unsigned char state_swizzle[4],
               unsigned first_level, unsigned last_level,
               unsigned first_layer, unsigned last_layer,
               unsigned width, unsigned height, unsigned depth,
               uint32_t *state,
               uint32_t *fmask_state)
{
    struct pipe_resource *res = &tex->buffer.b.b;
    const struct util_format_description *desc;
    unsigned char swizzle[4];
    int first_non_void;
    unsigned num_format, data_format, type, num_samples;
    uint64_t va;
```

```
    desc = util_format_description(pipe_format);

    num_samples = desc->colorspace == UTIL_FORMAT_COLORSPACE_ZS ?
            MAX2(1, res->nr_samples) :
            MAX2(1, res->nr_storage_samples);

    ...

    first_non_void = util_format_get_first_non_void_channel(pipe_format);

    switch (pipe_format) {
    case PIPE_FORMAT_S8_UINT_Z24_UNORM:
        num_format = V_008F14_IMG_NUM_FORMAT_UNORM;
        break;
    default:
        if (first_non_void < 0) {
            if (util_format_is_compressed(pipe_format)) {
                switch (pipe_format) {
                case PIPE_FORMAT_DXT1_SRGB:
                case PIPE_FORMAT_DXT1_SRGBA:
                }
            } else if (desc->layout == UTIL_FORMAT_LAYOUT_SUBSAMPLED) {
                num_format = V_008F14_IMG_NUM_FORMAT_UNORM;
            } else {
                num_format = V_008F14_IMG_NUM_FORMAT_FLOAT;
            }
        } else if (desc->colorspace == UTIL_FORMAT_COLORSPACE_SRGB) {
            num_format = V_008F14_IMG_NUM_FORMAT_SRGB;
        } else {
            num_format = V_008F14_IMG_NUM_FORMAT_UNORM;

            switch (desc->channel[first_non_void].type) {
            case UTIL_FORMAT_TYPE_FLOAT:
                num_format = V_008F14_IMG_NUM_FORMAT_FLOAT;
            ...

        }
    }

    data_format = si_translate_texformat(&screen->b, pipe_format, desc,
first_non_void);
    if (data_format == ~0) {
        data_format = 0;
    }
    if (type == V_008F1C_SQ_RSRC_IMG_1D_ARRAY) {
            height = 1;
        depth = res->array_size;
    } else if (type == V_008F1C_SQ_RSRC_IMG_2D_ARRAY ||
           type == V_008F1C_SQ_RSRC_IMG_2D_MSAA_ARRAY) {
        if (sampler || res->target != PIPE_TEXTURE_3D)
            depth = res->array_size;
    } else if (type == V_008F1C_SQ_RSRC_IMG_CUBE)
        depth = res->array_size / 6;

    state[0] = 0;
```

```
    state[1] = (S_008F14_DATA_FORMAT_GFX6(data_format) |
            S_008F14_NUM_FORMAT_GFX6(num_format));
    state[2] = (S_008F18_WIDTH(width - 1) |
            S_008F18_HEIGHT(height - 1) |
            S_008F18_PERF_MOD(4));
    state[3] = (S_008F1C_DST_SEL_X(si_map_swizzle(swizzle[0])) |
            S_008F1C_DST_SEL_Y(si_map_swizzle(swizzle[1])) |
            S_008F1C_DST_SEL_Z(si_map_swizzle(swizzle[2])) |
            S_008F1C_DST_SEL_W(si_map_swizzle(swizzle[3])) |
            S_008F1C_BASE_LEVEL(num_samples > 1 ? 0 : first_level) |
            S_008F1C_LAST_LEVEL(num_samples > 1 ?
                    util_logbase2(num_samples) :
                    last_level) |
            S_008F1C_TYPE(type));
    state[4] = 0;
    state[5] = S_008F24_BASE_ARRAY(first_layer);
    state[6] = 0;
    state[7] = 0;

    if (screen->info.chip_class >= GFX9) {
    } else {
        state[3] |= S_008F1C_POW2_PAD(res->last_level > 0);
        state[4] |= S_008F20_DEPTH(depth - 1);
        state[5] |= S_008F24_LAST_ARRAY(last_layer);
    }

    if (tex->dcc_offset) {
        state[6] = S_008F28_ALPHA_IS_ON_MSB(vi_alpha_is_on_msb(pipe_format));
    } else {
        /* The last dword is unused by hw. The shader uses it to clear
         * bits in the first dword of sampler state.
         */
        if (screen->info.chip_class <= CIK && res->nr_samples <= 1) {
            if (first_level == last_level)
                state[7] = C_008F30_MAX_ANISO_RATIO;
            else
                state[7] = 0xffffffff;
        }
    }


}
```

- 普通纹理用法与缓冲纹理用法基本一致，不过这里通过si_translate_texformat获取data_format,而num_format直接通过case映射获取

# 附录寄存器相关

## SQ_BUF_RSRC_WORD3

**SQ_BUF_RSRC_WORD3 寄存器**是一个可读写的 32 位寄存器，用于配置缓冲区资源的一些参数。该寄存器的地址为 0x8f0c。

以下是字段的定义：

| 字段名称 | 位范围 | 默认值 | 描述 |
|---|---|---|---|
| DST_SEL_X | 2:0 | 0x0 | 目标数据混合 - X：x，y，z，w，0，1 |
| | | | **可能的值：** |
| | | | 00 - SQ_SEL_0：使用常数 0.0 |
| | | | 01 - SQ_SEL_1：使用常数 1.0 |
| | | | 02 - SQ_SEL_RESERVED_0：保留 |
| | | | 03 - SQ_SEL_RESERVED_1：保留 |
| | | | 04 - SQ_SEL_X：使用 X 分量 |
| | | | 05 - SQ_SEL_Y：使用 Y 分量 |
| | | | 06 - SQ_SEL_Z：使用 Z 分量 |
| | | | 07 - SQ_SEL_W：使用 W 分量 |
| DST_SEL_Y | 5:3 | 0x0 | 目标数据混合 - Y：x，y，z，w，0，1 |
| | | | **可能的值：** |
| | | | 00 - SQ_SEL_0：使用常数 0.0 |
| | | | 01 - SQ_SEL_1：使用常数 1.0 |
| | | | 02 - SQ_SEL_RESERVED_0：保留 |
| | | | 03 - SQ_SEL_RESERVED_1：保留 |
| | | | 04 - SQ_SEL_X：使用 X 分量 |
| | | | 05 - SQ_SEL_Y：使用 Y 分量 |
| | | | 06 - SQ_SEL_Z：使用 Z 分量 |
| | | | 07 - SQ_SEL_W：使用 W 分量 |
| DST_SEL_Z | 8:6 | 0x0 | 目标数据混合 - Z：x，y，z，w，0，1 |
| | | | **可能的值：** |
| | | | 00 - SQ_SEL_0：使用常数 0.0 |
| | | | 01 - SQ_SEL_1：使用常数 1.0 |
| | | | 02 - SQ_SEL_RESERVED_0：保留 |
| | | | 03 - SQ_SEL_RESERVED_1：保留 |
| | | | 04 - SQ_SEL_X：使用 X 分量 |
| | | | 05 - SQ_SEL_Y：使用 Y 分量 |

| 字段名称 | 位范围 | 默认值 | 描述 |
| --- | --- | --- | --- |
| | | | 06 - SQ_SEL_Z：使用 Z 分量 |
| | | | 07 - SQ_SEL_W：使用 W 分量 |
| DST_SEL_W | 11:9 | 0x0 | 目标数据混合 - W：x，y，z，w，0，1 |
| | | | **可能的值：** |
| | | | 00 - SQ_SEL_0：使用常数 0.0 |
| | | | 01 - SQ_SEL_1：使用常数 1.0 |
| | | | 02 - SQ_SEL_RESERVED_0：保留 |
| | | | 03 - SQ_SEL_RESERVED_1：保留 |
| | | | 04 - SQ_SEL_X：使用 X 分量 |
| | | | 05 - SQ_SEL_Y：使用 Y 分量 |
| | | | 06 - SQ_SEL_Z：使用 Z 分量 |
| | | | 07 - SQ_SEL_W：使用 W 分量 |
| NUM_FORMAT | 14:12 | 0x0 | 数值格式（unorm、snorm、float 等） |
| | | | **可能的值：** |
| | | | 00 - BUF_NUM_FORMAT_UNORM |
| | | | 01 - BUF_NUM_FORMAT_SNORM |
| | | | 02 - BUF_NUM_FORMAT_USCALED |
| | | | 03 - BUF_NUM_FORMAT_SSCALED |
| | | | 04 - BUF_NUM_FORMAT_UINT |
| | | | 05 - BUF_NUM_FORMAT_SINT |
| | | | 06 - BUF_NUM_FORMAT_SNORM_OGL |
| | | | 07 - BUF_NUM_FORMAT_FLOAT |
| DATA_FORMAT | 18:15 | 0x0 | 数据格式（8、16、8_8 等） |
| | | | **可能的值：** |
| | | | 00 - BUF_DATA_FORMAT_INVALID |
| | | | 01 - BUF_DATA_FORMAT_8 |
| | | | 02 - BUF_DATA_FORMAT_16 |
| | | | 03 - BUF_DATA_FORMAT_8_8 |
| | | | 04 - BUF_DATA_FORMAT_32 |

| 字段名称 | 位范围 | 默认值 | 描述 |
| --- | --- | --- | --- |
| | | | 05 - BUF_DATA_FORMAT_16_16 |
| | | | 06 - BUF_DATA_FORMAT_10_11_11 |
| | | | 07 - BUF_DATA_FORMAT_11_11_10 |
| | | | 08 - BUF_DATA_FORMAT_10_10_10_2 |
| | | | 09 - BUF_DATA_FORMAT_2_10_10_10 |
| | | | 10 - BUF_DATA_FORMAT_8_8_8_8 |
| | | | 11 - BUF_DATA_FORMAT_32_32 |
| | | | 12 - BUF_DATA_FORMAT_16_16_16_16 |
| | | | 13 - BUF_DATA_FORMAT_32_32_32 |
| | | | 14 - BUF_DATA_FORMAT_32_32_32_32 |
| | | | 15 - BUF_DATA_FORMAT_RESERVED_15 |
| ELEMENT_SIZE | 20:19 | 0x0 | 元素大小：2、4、8 或 16 字节。用于分页缓冲区寻址 |
| INDEX_STRIDE | 22:21 | 0x0 | 索引步幅：8、16、32 或 64。用于分页缓冲区寻址 |
| ADD_TID_ENABLE | 23 | 0x0 | 将线程 ID（0..63）添加到地址计算的索引中。主要用于临时缓冲区 |
| HASH_ENABLE | 25 | 0x0 | 如果为 true，则为缓冲区地址进行哈希以获得更好的缓存性能 |
| HEAP | 26 | 0x0 | 保留字段 |
| TYPE | 31:30 | 0x0 | 资源类型：必须为 BUFFER |
| | | | **可能的值：** |
| | | | 00 - SQ_RSRC_BUF |
| | | | 01 - SQ_RSRC_BUF_RSVD_1 |
| | | | 02 - SQ_RSRC_BUF_RSVD_2 |
| | | | 03 - SQ_RSRC_BUF_RSVD_3 |
| | | | 15 - IMG_NUM_FORMAT_RESERVED_15 |

# SQ_IMG_RSRC_WORD1

**SQ_IMG_RSRC_WORD1** 是一个可读写的 32 位寄存器，用于配置图像资源的一些参数。该寄存器的地址为 0x8f14。

以下是字段的定义：

| 字段名称 | 位范围 | 默认值 | 描述 |
|---|---|---|---|
| BASE_ADDRESS_HI | 7:0 | 0x0 | 图像基地址，位 47-40 |
| MIN_LOD | 19:8 | 0x0 | 最小 LOD，4.8 格式 |
| DATA_FORMAT | 25:20 | 0x0 | 数据格式（8、8_8、16 等） |
| | | | **可能的值：** |
| | | | 00 - IMG_DATA_FORMAT_INVALID |
| | | | 01 - IMG_DATA_FORMAT_8 |
| | | | 02 - IMG_DATA_FORMAT_16 |
| | | | 03 - IMG_DATA_FORMAT_8_8 |
| | | | 04 - IMG_DATA_FORMAT_32 |
| | | | 05 - IMG_DATA_FORMAT_16_16 |
| | | | 06 - IMG_DATA_FORMAT_10_11_11 |
| | | | 07 - IMG_DATA_FORMAT_11_11_10 |
| | | | 08 - IMG_DATA_FORMAT_10_10_10_2 |
| | | | 09 - IMG_DATA_FORMAT_2_10_10_10 |
| | | | 10 - IMG_DATA_FORMAT_8_8_8_8 |
| | | | 11 - IMG_DATA_FORMAT_32_32 |
| | | | 12 - IMG_DATA_FORMAT_16_16_16_16 |
| | | | 13 - IMG_DATA_FORMAT_32_32_32 |
| | | | 14 - IMG_DATA_FORMAT_32_32_32_32 |
| | | | 15 - IMG_DATA_FORMAT_RESERVED_15 |
| | | | 16 - IMG_DATA_FORMAT_5_6_5 |
| | | | 17 - IMG_DATA_FORMAT_1_5_5_5 |
| | | | 18 - IMG_DATA_FORMAT_5_5_5_1 |
| | | | 19 - IMG_DATA_FORMAT_4_4_4_4 |
| | | | 20 - IMG_DATA_FORMAT_8_24 |
| | | | 21 - IMG_DATA_FORMAT_24_8 |
| | | | 22 - IMG_DATA_FORMAT_X24_8_32 |
| | | | 23 - IMG_DATA_FORMAT_RESERVED_23 |
| | | | 24 - IMG_DATA_FORMAT_RESERVED_24 |
| | | | 25 - IMG_DATA_FORMAT_RESERVED_25 |

| 字段名称 | 位范围 | 默认值 | 描述 |
| --- | --- | --- | --- |
| | | | 26 - IMG_DATA_FORMAT_RESERVED_26 |
| | | | 27 - IMG_DATA_FORMAT_RESERVED_27 |
| | | | 28 - IMG_DATA_FORMAT_RESERVED_28 |
| | | | 29 - IMG_DATA_FORMAT_RESERVED_29 |
| | | | 30 - IMG_DATA_FORMAT_RESERVED_30 |
| | | | 31 - IMG_DATA_FORMAT_RESERVED_31 |
| | | | 32 - IMG_DATA_FORMAT_GB_GR |
| | | | 33 - IMG_DATA_FORMAT_BG_RG |
| | | | 34 - IMG_DATA_FORMAT_5_9_9_9 |
| | | | 35 - Reserved |
| | | | 36 - Reserved |
| | | | 37 - Reserved |
| | | | 38 - Reserved |
| | | | 39 - Reserved |
| | | | 40 - Reserved |
| | | | 41 - Reserved |
| | | | 42 - IMG_DATA_FORMAT_RESERVED_42 |
| | | | 43 - IMG_DATA_FORMAT_RESERVED_43 |
| | | | 44 - IMG_DATA_FORMAT_FMASK8_S2_F1 |
| | | | 45 - IMG_DATA_FORMAT_FMASK8_S4_F1 |
| | | | 46 - IMG_DATA_FORMAT_FMASK8_S8_F1 |
| | | | 47 - IMG_DATA_FORMAT_FMASK8_S2_F2 |
| | | | 48 - IMG_DATA_FORMAT_FMASK8_S4_F2 |
| | | | 49 - IMG_DATA_FORMAT_FMASK8_S4_F4 |
| | | | 50 - IMG_DATA_FORMAT_FMASK16_S16_F1 |
| | | | 51 - IMG_DATA_FORMAT_FMASK16_S8_F2 |
| | | | 52 - IMG_DATA_FORMAT_FMASK32_S16_F2 |
| | | | 53 - IMG_DATA_FORMAT_FMASK32_S8_F4 |
| | | | 54 - IMG_DATA_FORMAT_FMASK32_S8_F8 |
| | | | 55 - IMG_DATA_FORMAT_FMASK64_S16_F4 |

| 字段名称 | 位范围 | 默认值 | 描述 |
|---|---|---|---|
| | | | 56 - IMG_DATA_FORMAT_FMASK64_S16_F8 |
| | | | 57 - IMG_DATA_FORMAT_4_4 |
| | | | 58 - IMG_DATA_FORMAT_6_5_5 |
| | | | 59 - IMG_DATA_FORMAT_1 |
| | | | 60 - IMG_DATA_FORMAT_1_REVERSED |
| | | | 61 - IMG_DATA_FORMAT_32_AS_8 |
| | | | 62 - IMG_DATA_FORMAT_32_AS_8_8 |
| | | | 63 - IMG_DATA_FORMAT_32_AS_32_32_32_32 |
| NUM_FORMAT | 29:26 | 0x0 | 数字格式（unorm、snorm、float 等） |
| | | | **可能的值：** |
| | | | 00 - IMG_NUM_FORMAT_UNORM |
| | | | 01 - IMG_NUM_FORMAT_SNORM |
| | | | 02 - IMG_NUM_FORMAT_USCALED |
| | | | 03 - IMG_NUM_FORMAT_SSCALED |
| | | | 04 - IMG_NUM_FORMAT_UINT |
| | | | 05 - IMG_NUM_FORMAT_SINT |
| | | | 06 - IMG_NUM_FORMAT_SNORM_OGL |
| | | | 07 - IMG_NUM_FORMAT_FLOAT |
| | | | 08 - IMG_NUM_FORMAT_RESERVED_8 |
| | | | 09 - IMG_NUM_FORMAT_SRGB |
| | | | 10 - IMG_NUM_FORMAT_UBNORM |
| | | | 11 - IMG_NUM_FORMAT_UBNORM_OGL |
| | | | 12 - IMG_NUM_FORMAT_UBINT |
| | | | 13 - IMG_NUM_FORMAT_UBSCALED |
| | | | 14 - IMG_NUM_FORMAT_RESERVED_14 |
| | | | 15 - IMG_NUM_FORMAT_RESERVED_15 |