Shader Compiler User's Guide

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Document serial number: 000004892117

Table of Contents

About This Document

The purpose of this guide is to describe the PlayStation®Vita shader compiler.

Conventions

The typographical conventions used in this guide are explained in this section.

Notes

Additional advice or related information is presented as a 'note' surrounded by a box. For example:

> **Note:** Vertex attributes are bound by name in the API and one symbol is generated per vector.

Text

- Names of keyboard functions or keys are formatted in a bold serif font. For example, **Ctrl**, **Delete**, **F9**.
- File names, source code and command-line text are formatted in a fixed-width font. For example:

```
psp2cgc -profile <profile name> input.cg
```

1 Using the Shader Compiler

This chapter describes how to use the PlayStation®Vita shader compiler.

Language Specification

The PlayStation®Vita shader compiler implements the Cg 2.2 specification with the PlayStation®Vita system. You can find this specification here:

http://developer.download.nvidia.com/cg/Cg_2.2/Cg-2.2_October2009_LanguageSpecification.pdf

Command-line Usage

The basic syntax required for a compile is:

```
psp2cgc -profile <profile name> input.cg
```

By default, this command writes a shader binary called `input_cg.gxp`. However, you can specify the `output` file name as follows:

```
psp2cgc -profile <profile name> input.cg -o output.gxp
```

Compiler Options

The tables below describe the options that can be used with the `psp2cgc` command.

Table 1 Basic Compiler Options

Option	Description
`-h / -help`	Prints a brief command-line reference.
`-v / -version`	Prints the version number.
`-I<directory>`	Adds `<directory>` as a search path.
`-D<name>[=<value>]`	Defines a new macro called `<name>` and optionally sets it to `<value>`.
`-include <file>`	Treats `<file>` as if it was included on the first line of the input source file.
`-profile <profile>`	The target profile for the compilation. See Chapter 2, Profiles for details.
`-o <filename>`	Explicitly specifies the name of the file to be written.
`-entry <function>`	Sets `<function>` to be the entry point. Note that `<function>` must not be overloaded or a struct method.
`-nostdlib`	Disables the Cg standard library. If your shader does not need Cg standard library functions, use this option to reduce compilation time.

Table 2 Language-Specific Compiler Options

Option	Description
`-W<level>`	Sets the warning level to `<level>`. `<level>` can be a value in the range [0,4] and defaults to 1. If `<level>` is set to 0, all warnings are disabled.
`-Wperf`	Enables warnings about potential performance hazards.
`-Werror`	Treats all warnings as errors.
`-Wsuppress=id[,id...]`	Suppress specified warnings.
`-pedantic`	Warns about non-standard extensions being used.
`-pedantic-error`	Errors on non-standard extensions being used.
`-fx` `-nofx`	`-fx` enables CgFX support. Note that this is currently pass-through only. `-nofx` disables CgFX support.

Table 3 Code Generation Compiler Options

Option	Description
-O<level>	Sets the optimization level to <level>. <level> can be a value in the range [0,4] and defaults to 3. Note that level 4 optimizations are considered experimental.
-fastmath -nofastmath	-fastmath allows the compiler to perform algebraic transformations such as a*4 instead of a+a+a+a. -nofastmath disables this functionality. The default value is -fastmath.
-fastprecision -nofastprecision	Enables or disables optimizations that may reduce numeric accuracy. The default value is -nofastprecision.
-bestprecision	Disables all optimizations that may affect precision. Implies -nofastmath and -nofastprecision.
-fastint -nofastint	-fastint allows the compiler to reduce the range of float to int conversions to the [-32768, 32767] range if the int value is used only as an array index or loop counter. -nofastint disables this functionality. The default value is -fastint.
-invpos	Generates a position invariant vertex program.

Table 4 Miscellaneous Compiler Options

Option	Description
-lang <locale>	Changes the display language of the compiler warnings and errors. Valid locale settings are en and jp.
-nocolor	Disables coloring of warnings and errors.
-E	Enables preprocessing-only mode. No shader binary will be written.
-P	When used with -E, disables generation of #line directives.
-C	When used with -E, preserves comments in the output.
-cache	Outputs a cache file for shader association. For details about this subject, refer to the *Razor User's Guide*.
-xmlcache	Outputs a cache file in XML form for shader association. For details about this subject, refer to the *Razor User's Guide*.
-cachedir <path>	The path that the SDB file is written to. If this argument is not provided, the SDB file is written into the same directory as the GXP file.
-sdb <file>	Sets a custom name for the shader association cache file.
-xmlsdb <file>	Sets a custom name for the XML form of the shader association cache file.

Table 5 Dependency Compiler Options

Option	Description
-M / -MM	Output dependencies with -E implied.
-MD / -MMD	Enables dependency file generation without -E implied.
-MP	Adds phony targets for all inputs to the main file.
-MF <file>	Explicitly specifies the name of the dependency file to be written.
-MT <target>	Changes the name of the target rule emitted in the dependency file.
-MQ <target>	Like -MT, but generates escape sequences for all characters that have a special meaning in makefiles.
-MG	Causes missing files to be added to the dependency list without an error being raised. This option may only be used in preprocessing-only modes.

Note: The dependency generation flags listed here are intended to be compatible with gcc.

2 Profiles

This chapter describes the profiles supported by the shader compiler.

Overview

The following profiles are supported by the shader compiler:

Table 6 Supported Profiles

Name	Description
sce_vp_psp2	Vertex program.
sce_fp_psp2	Fragment program.

Semantics

Semantics are used to explicitly assign hardware resources to variables that define an *interface* for the shader. A variable's declaration context determines whether or not the variable is considered to define the interface. Table 7 lists the available contexts.

Table 7 Variable Declaration Contexts

Category	Declaration Contexts
Vertex attribute	An input argument to the entry function of a vertex program.
Varying	An output argument or return value of the entry function in a vertex program. An input argument to the entry function of a fragment program.
Fragment shader output	An output argument or return value of the entry function in a fragment program.
Uniform	A declaration at the global scope without a "static" qualifier. An input argument to the entry function with a "uniform" qualifier.

Note: If a declaration does not fall into one of the categories above, a semantic can be provided, but will have no effect. The following example illustrates the rules:

```
float4 uniform1 : BUFFER[0];

float4 mainVP(
    // all input arguments are vertex attributes
    float4 vertexAttribute1,
    half2 vertexAttribute2,

    // all output arguments are varyings
    out fixed3 varying2 : TEXCOORD0,
    out float varying3 : TEXCOORD1,

    // all input arguments with a 'uniform' qualifier are uniforms
    uniform half4 uniform2 : BUFFER[0]

    // the return value of entry function is a varying as well
    ) : POSITION
{
    ...
}
```

Semantics do not always have to be specified as part of the declarator itself. Alternatively, they may be applied for individual members of a struct:

```
struct a2v
{
    float4 vertexAttribute1;
    half2 vertexAttribute2;
};
struct v2f
{
    float4 varying1 : POSITION;
    fixed3 varying2 : TEXCOORD0;
    float varying3 : TEXCOORD1;
};
float4 uniform1 : BUFFER[0];
half4 uniform2 : BUFFER[0];

void mainVP(
    a2v input,
    out v2f output
    )
{
    ...
}
```

Note: In the example above, the `v2f` struct could be re-used as input to the fragment program. Fragment profiles may not read from variables with a `POSITION` semantic; however, they do allow such variables to be declared.

Vertex Input Semantics

The PlayStation®Vita vertex profile allows attributes both with and without semantics. In either case, hardware registers are allocated in order of declaration. There is no direct mapping between semantics and particular hardware resources; semantics serve only as an alternative to name-based lookups. As a result of this, different semantics will always point to different hardware registers.

Table 8 Vertex Attribute Semantics

Semantic Name	Index Range
ATTR	0–15
BOOL	0–15
BINORMAL	0–15
BLENDINDICES	0–15
BLENDWEIGHT	0–15
COLOR	0–15
DIFFUSE	0–15
FOGCOORD	0–15
NORMAL	0–15
POINTSIZE	0–15
POSITION	0–15
SPECULAR	0–15
TANGENT	0–15
TEXCOORD	0–15

Table 9 Special Semantics

Semantic	Description	Supported Types
`INDEX`	Index for the currently shaded vertex.	`char, unsigned char, short, unsigned short, int, unsigned int`
`INSTANCE`	Instance for the currently shaded vertex.	`char, unsigned char, short, unsigned short, int, unsigned int`

Hardware registers for input arguments to a vertex program with the special semantics in Table 9 are always allocated last. Such inputs arguments are not considered to be vertex attributes and do not require any input data from the vertex stream.

```
void main(
    // one symbol: 'attr0'(no semantic)
    in float4 attr0,

    // two symbols: 'attr1[0]'(COLOR0) and 'attr1[1]'(COLOR1)
    in half2 attr1[2] : COLOR0,

    // three symbols: 'attr2[0]'(ATTR0), 'attr2[1]'(ATTR1) and 'attr2[2]'(ATTR2)
    in float3x3 attr2 : ATTR0
    )
{
     ...
}
```

Note: For vertex attributes, one symbol is generated per vector. No symbols are generated for vertex program inputs with special semantics.

Vertex Output Semantics

The use of appropriate vertex output semantics and data formats plays a crucial role in writing effective shaders, as each kind of semantic is linked to specific parameter buffer formats. Therefore, using the most effective representation for each varying should be a primary consideration when optimizing shaders.

- `COLOR` varyings should be used where possible as they are always more compact than an equivalent `TEXCOORD`.
- `TEXCOORD` varyings stored at half precision in the parameter buffer are interpolated at full precision, making them a viable alternative in many cases.
- The parameter buffer only ever encodes 2, 3, or 4 coefficient `TEXCOORD` varyings. Unused coefficients waste parameter buffer space and should be avoided where possible.

For further information regarding the parameter buffer, refer to the *libgxm Overview*.

Where multiple parameter buffer formats are available, the one closest to the user declared type will be used. For example, in the case of a `fixed TEXCOORD`, `half2` will be chosen.

Table 10 Vertex Output Semantics

Semantic	Description	Parameter Buffer Formats
`POSITION, HPOS`	Output vertex position.	`float3/float4`
`COLOR0-COLOR1, COL0-COL1`	Low precision color interpolant.	`unsigned char4`
`TEXCOORD0-TEXCOORD9, TEX0-TEX9`	Texture coordinate interpolants.	`half[2-4], float[2-4]`

Semantic	Description	Parameter Buffer Formats
TEXCOORD0_HALF-TEXCOORD9_HALF	Similar to TEXCOORD0-TEXCOORD9, but stored at half precision in the parameter buffer. See Interpolant Precision and Parameter Buffer Packing Format for details.	half[2-4]
FOG, FOGC	Fog coordinate.	float
PSIZE, PSIZ	Point sprite size.	N/A
CLP0-CLP7	Distances to user-defined clipping planes.	N/A

Where non-packed array types are used or the array dimensionality is greater than 1, multiple resources will be reserved:

```
void main(
    out float2x2 vary0 : TEXCOORD0,        // allocates TEXCOORD0 and TEXCOORD1
    out float vary1[2] : TEXCOORD2,        // allocates TEXCOORD2 and TEXCOORD3
    out packed float vary2[2]: TEXCOORD4   // allocates TEXCOORD4
    )
{
    ...
}
```

Fragment Input Semantics

Table 11 Fragment Input Semantics

Semantic	Description	Iterated Formats
COLOR0-COLOR1, COL0-COL1	Low precision color interpolant.	unsigned char4, fixed4, half4, float4
COLOR0_CENTROID-COLOR1_CENTROID	Like COLOR0-COLOR1, but sampling at the pixel centroid.	unsigned char4, fixed4, half4, float4
FOG, FOGC	Interpolated fog value.	float
FOG_CENTROID	Like FOG, but sampling at the pixel centroid.	float
TEXCOORD0-TEXCOORD9, TEX0-TEX9, TEXCOORD0_HALF-TEXCOORD9_HALF	Texture coordinate interpolants.	fixed[1-4], half[1-4], float[1-4]
TEXCOORD0_CENTROID-TECXOORD9_CENTROID	Like TEXCOORD0-TEXCOORD9, but sampling at the pixel centroid.	fixed[1-4], half[1-4], float[1-4]
WPOS	Window position coordinates.	float4
WPOS_CENTROID	Like WPOS, but sampling at the pixel centroid.	float4
SPRITECOORD, POINTCOORD	Coordinate over a point sprite from [0,0] to [1,1]. This semantic must be used only if point sprites with generated UVs are being rendered.	fixed2, half2, float2
FRAGCOLOR	This is part of an extension. Supplies the previous color value of the sample. Please refer to the Programmable Blending section for details.	Same format as used for the COLOR0 output.
FACE	A value less than zero if the fragment being rendered is back facing, greater than zero if it is front facing.	float

Unlike varying outputs, inputs permit multiple variables using the same semantic to alias:

```
void main(
    in float2x2 vary0 : TEXCOORD0,         // iterates TEXCOORD0 and TEXCOORD1
    in float2 vary1[2] : TEXCOORD0,        // iterates TEXCOORD0 and TEXCOORD1
    in packed float vary2[4] : TEXCOORD0   // iterates TEXCOORD0
    )
{
    ...
}
```

Fragment Output Semantics

Table 12 Fragment Output Semantics

Semantic	Description
COLOR0, COL0	Output color for the color surface.
DEPTH, DEPR	Output for depth replace shaders.

```
void main(
    out float3 out0 : COLOR0,    // writes to the color surface
    out float out2 : DEPTH        // writes the depth replace value for this
fragment
    )
{
    ...
}
```

Uniform Semantics

Table 13 Uniform Semantics

Semantic	Description
BUFFER0-13	The uniform buffer the variable should be bound to. Where multiple variables are bound to the same buffer, they are concatenated in order of declaration.
TEXUNIT0-15, register(s#)	The texture unit a sampler should be bound to.

```
// define uniforms for the default buffer
float4 defaultUni0;
half4 defaultUni1[4];
myStruct defaultUni2;

// define uniform buffer 0 layout
float2 bufferUni0 : BUFFER[0];
half bufferUni1[16] : BUFFER[0];
myStruct bufferUni2 : BUFFER[0];

// define samplers
sampler2D smp0;              // implicitly bound to TEXUNIT0
sampler2D smp1 : TEXUNIT1;   // explicitly bound to TEXUNIT1
sampler2D smp2 : register(s2);  // explicitly bound to TEXUNIT2
```

Uniform Buffers

Uniform values are stored inside uniform buffers, there are two types of uniform buffers:

- Default Uniform Buffer.
- User Declared Uniform Buffers.

When a uniform is declared without the `BUFFER[x]` semantic, it is allocated inside the *default uniform buffer* which has the following characteristics:

- The compiler removes unreferenced symbols as part of dead code elimination.
- The compiler can choose to allocate uniforms in an order which is different from the declaration order.

When uniforms are declared using the `BUFFER[x]` semantics they are allocated inside the *user declared uniform buffer* #*x*, and:

- The compiler never dead strip uniforms - this preserves the memory layout of the buffer.
- The compiler allocates uniforms in the order in which they are declared.

User declared uniform buffers are provided so that it is possible for shader programmers to share the data layout of uniform variables between different shaders as well as between their GPU code and CPU code.

If required, it is possible to enable dead stripping of uniforms from a user declared uniform buffer by using the `strip_buffer` pragma, see Chapter 4, Pragmas for more information.

psp2cgc also supports writable user declared uniform buffers. Writable uniform buffers allow data to be stored from a vertex or fragment program to an area of mapped memory. In order to declare a writable uniform buffer the `readwrite_buffer` pragma should be used. By default symbol information is not generated for user declared uniform buffers marked with the `readwrite_buffer` pragma. To force psp2cgc to include symbol information for writable uniform buffers, the buffer must be marked with the `enable_buffer_symbols` pragma; see Chapter 4, Pragmas for more information on these pragmas.

It is not permitted to perform writes to writable user declared uniform buffers during the selective rate phase of a fragment program. Attempting to do so will result in a compile time error.

> **Note:** This will only occur if your write to a writable user declared uniform buffer has a dependency on the results of reading the fragment color in a shader that is declared with `__msaa`. Please refer to Programmable Blending With MSAA for more information on generating code in a selective rate phase.

The compiler will also emit an error when a shader tries to write to a writable user declared uniform buffer before a discard statement or a write into the `DEPTH` binding. You should always make sure that writes into writable uniform buffers are performed after a discard or depth write.

A maximum of 14 user declared uniform buffers can be used for a single shader.

When defining the layout of a user declared uniform buffer it is important to consider the following alignment restrictions:

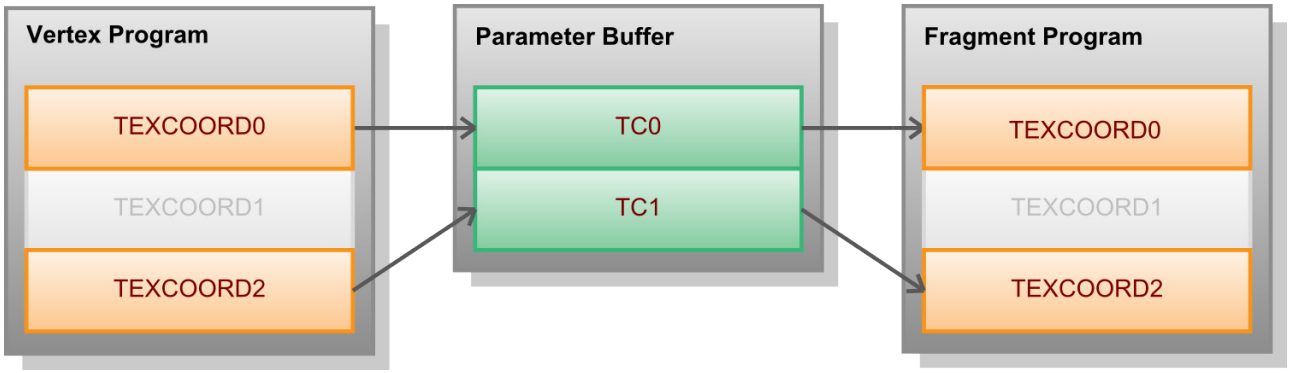
- Floating point vector elements must start on a 64-bit boundary.
- All other data types must start on a 32-bit boundary.

Avoiding Shader Linking

When a fragment program is used in libgxm, it normally requires a vertex program to link with. This ensures that all varyings read by the fragment program are written to the parameter buffer by the vertex program. It also accounts for the SGX compacting `TEXCOORD` interpolants.

For example, consider a vertex program that writes to `TEXCOORD0` and `TEXCOORD2`. When the texture coordinates are written into the parameter buffer, the gaps in the coordinate usage are skipped. As a result, the two coordinates are written to parameter buffer locations TC0 and TC1, respectively. Therefore, the fragment program must be *linked*, resulting in the mapping shown in Figure 1.

Figure 1 Sparse Varying Mapping



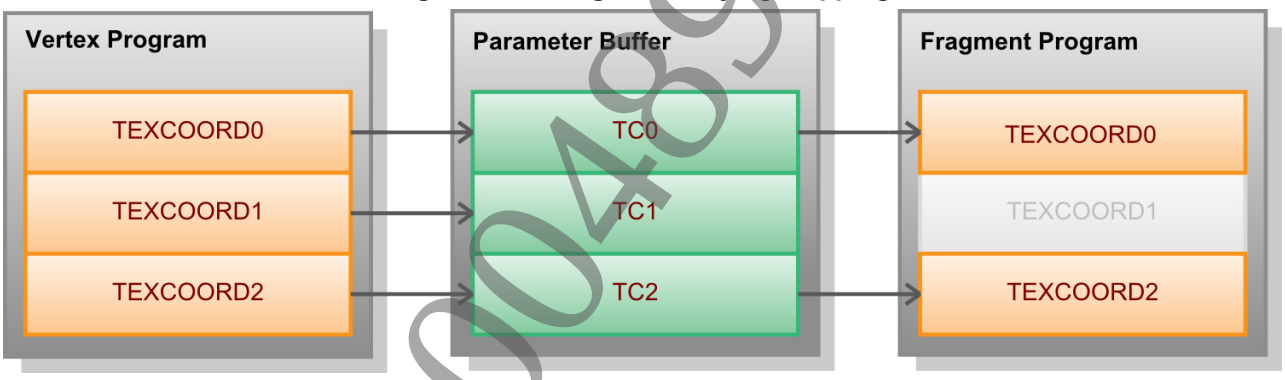
However, you can avoid the link step (passing NULL for the vertex program at runtime) by doing one of the following:

- In the vertex program, write a contiguous range of `TEXCOORD` semantics from 0 to N.
- In the fragment program semantics, manually account for the parameter buffer remapping.

Figure 2 shows how a contiguous range could be written from the vertex program to avoid the linking stage.

Note: In this case, the parameter buffer is unnecessarily large, because more data is stored than required by the fragment program.

Figure 2 Contiguous Varying Mapping



Standard Library

All PlayStation®Vita profiles support the full Cg standard library with the exception of certain texture functions and `frexp()`. Additionally the function `floatToIntBits()` has different behavior from the Cg specification, which says that the NaN floating point values are canonicalized to the integer value `0x7fc00000` regardless of the specific NaN encoding or the sign bit of the NaN. The psp2cgc standard library does not perform this conversion and will return an integer value that corresponds bit-per-bit to the original floating point number.

Texture Functions

PlayStation®Vita profiles support the texture functions shown in Table 14.

Table 14 Texture Functions

Prototype
`half4 tex2D(sampler2D, float3)`
`unsigned int4 tex2D(usampler2D: float2, float2: float2)`
`int4 tex2D(isampler2D, float2, float2, float2)`
`half4 tex2D(sampler2D, float2, float2, float2)`

Prototype
unsigned int4 tex2D(usampler2D, float2)
int4 tex2D(isampler2D, float2)
half4 tex2D(sampler2D, float2)
half4 texCUBE(samplerCUBE, float4, float3, float3)
half4 texCUBE(samplerCUBE, float4)
unsigned int4 texCUBE(usamplerCUBE, float3: float3: float3)
int4 texCUBE(isamplerCUBE, float3, float3, float3)
half4 texCUBE(samplerCUBE, float3, float3, float3)
unsigned int4 texCUBE(usamplerCUBE, float3)
int4 texCUBE(isamplerCUBE, float3)
half4 texCUBE(samplerCUBE, float3)
half4 tex1Dproj(sampler1D, float3)
unsigned int4 tex1Dproj(usampler1D, float2)
int4 tex1Dproj(isampler1D, float2)
half4 tex1Dproj(sampler1D, float2)
half4 tex2Dproj(sampler2D, float4)
unsigned int4 tex2Dproj(usampler2D, float3)
int4 tex2Dproj(isampler2D, float3)
half4 tex2Dproj(sampler2D, float3)
unsigned int4 texCUBEproj(usamplerCUBE, float4)
int4 texCUBEproj(isamplerCUBE, float4)
half4 texCUBEproj(samplerCUBE, float4)
unsigned int4 tex1Dbias(usampler1D, float4)
int4 tex1Dbias(isampler1D, float4)
half4 tex1Dbias(sampler1D, float4)
unsigned int4 tex2Dbias(usampler2D, float4)
int4 tex2Dbias(isampler2D, float4)
half4 tex2Dbias(sampler2D, float4)
unsigned int4 texCUBEbias(usamplerCUBE, float4)
int4 texCUBEbias(isamplerCUBE, float4)
half4 texCUBEbias(samplerCUBE, float4)
unsigned int4 tex1Dlod(usampler1D, float4)
int4 tex1Dlod(isampler1D, float4)
half4 tex1Dlod(sampler1D, float4)
unsigned int4 tex2Dlod(usampler2D, float4)
int4 tex2Dlod(isampler2D, float4)
half4 tex2Dlod(sampler2D, float4)
unsigned int4 texCUBElod(usamplerCUBE, float4)
int4 texCUBElod(isamplerCUBE, float4)
half4 texCUBElod(samplerCUBE, float4)

Built-in Macros

The following built-in macros are available:

```
#define __psp2__ 1
#define __SCE__ 1
```

3 Language Extensions

This chapter describes the language extensions provided by the shader compiler.

Additional Cg Keywords

psp2cgc supports all keywords that are part of the Cg language standard plus few additional extensions detailed in the following table:

Keyword	Description
`break`	Causes control to pass to the statement following the innermost enclosing `while`, `do` or `for` loop.
`continue`	Causes control to pass to the end of the innermost enclosing `while`, `do` or `for` loop, at which point the loop continuation condition is re-evaluated.

Texture Query Result Formats

To express the behavior of the SGX texture unit, the shader compiler provides a language extension that allows the user to specify explicitly the format in which the texture unit should return the results of a query.

If this directive is not specified, the compiler uses the format defined by the standard library interfaces, with one exception: if a `float` sampler is used, the compiler can choose between returning a `half4` or `float4` to improve performance. If `half` precision is not sufficient, the result format should be specified explicitly to avoid loss of precision due to optimizations.

All queries from the same sampler must return the same number of coefficients. For example, if a sampler returns a `half2` in one query, it cannot be used to return a `half4` in another query. Furthermore, libgxm provides validation functions at runtime to ensure that textures are queried only in formats with which they are compatible.

For more information about the formats returned by the texture unit, refer to the *GPU User's Guide*.

The syntax for explicitly specifying a texture query format is `tex2D<FORMAT>` where `FORMAT` is a numeric Cg type.

Note: Integral types must be used with `usampler` and `isampler` types for `unsigned` and `signed` integer result formats, respectively. For example, to use a result format of `unsigned char4`, a 2D sampler must be declared as `usampler2D`.

```
sampler2D smpShadow;
sampler2D smpAlbedo;

float4 main(
    float2 texCoord : TEXCOORD0,
    float4 projCoord : TEXCOORD1
    ) : COLOR
{
    // half precision is sufficient for the albedo. Let the compiler pick
    // the optimal result format automatically.
    half3 const albedo = h3tex2D(smpAlbedo, texCoord);
    // Enforce a float precision query for the shadow test.
    float const shadow = tex2Dproj<float>(smpShadow, projCoord);
    return albedo * shadow;
}
```

Note how the prefixed texture function `h3tex2D` is being used. The result format extension does not place any restriction on the prefixes that can be used with texture functions. As with other Cg implementations, the prefix is purely syntactic sugar for an explicit cast. Therefore, the line

```
half3 const albedo = h3tex2D(smpAlbedo, texCoord);
```

in the example above is equivalent to:

```
half3 const albedo = (half3)tex2D(smpAlbedo, texCoord);
```

Column Major Matrices

By default, matrices in Cg are stored in row major ordering, which may be expressed explicitly using the `row_major` keyword. In addition to this, psp2cgc supports `column_major` storage as an extension.

These specifiers affect only the internal coefficient ordering in registers or main memory; the functionality expressed by a shader is unaffected. However, due to the different resource association, shader performance may vary.

To change the default ordering of matrices, please refer to the `pack_matrix` pragma documentation.

```
column_major float3x4 myMtx;
float3 main(float4 myAttr) : POSITION
{
    return mul(myMtx, myAttr);
}
```

Explicit Fragment Shader Output Formats

To give users direct access to the pixel formats supported by the SGX, psp2cgc supports the `__regformat` keyword:

```
void main(out __regformat unsigned short2 result : COLOR0) {
   return 1;
}
```

Note: the `__regformat` keyword must be used only with the `COLOR0` output. Furthermore, the type of the variable must be exactly 32bits or 64bits in size.

The use of `__regformat` specifies that a variable must be stored in the format that was declared. If `__regformat` is not specified, psp2cgc behaves like other Cg implementations and ignores the format declared by the user. Instead, psp2cgc uses the default output format, which is `half4`.

In many cases, the SGX can convert natively from a `half4` output register format to the format of the color surface. In these cases, the `__regformat` keyword is unnecessary. For a full list of the available conversions, refer to the *GPU User's Guide.*

Where no native hardware conversion is available, users may either rely on the shader patcher to insert code for the conversion, or can use the `__regformat` syntax. Using the shader patcher is convenient, but could increase the instruction count of the shader and requires the compiler outputs to be in `half4` format.

The example above illustrates the case where a `SCE_GXM_COLOR_BASE_FORMAT_U16U16` color surface is used. Using the `__regformat` syntax has two benefits:

- A `half` cannot represent the full range of an `unsigned short`; therefore, without the `__regformat` keyword, the shader output may be incorrect.
- When the shader patcher is used, psp2cgc first converts its outputs to a `half2` and then the shader patcher appends code to convert them back to an `unsigned short2`, hence degrading performance.

Reinterpreting Numeric Values Using bit_cast

To allow for numeric values being reinterpreted in a different format (for example, for packing data) psp2cgc provides the `bit_cast` extension:

```
uniform unsigned int2 myUniform;
half4 main() {
    return bit_cast<half4>(myUniform);
}
```

The types involved in `bit_cast` must be vectors or scalars. Any other type such as a sampler or struct must not be used as either the source or destination type of the cast.

Furthermore, if the destination type is larger than the source type, all bits that were not present in the source type must be considered invalid. For example, when casting from a `fixed3` to an `unsigned int` type, the top two bits of the resulting integer should not be referenced since only the bottom 30 bits are valid.

In the C language, the equivalent of a `bit_cast` is type punning performed via pointer aliasing or a union. For example, consider the following `bit_cast`:

```
unsigned int myIntValue = 0x3f800000;
float myFloatValue = bit_cast<float>(myIntValue);
```

The equivalent C code (ignoring strict aliasing rules) is:

```
unsigned int myIntValue = 0x3f800000;
float myFloatValue = *((float*)&myIntValue);
```

Avoiding Fragment Program Patching Using __nativecolor

In order to allow for a traditional blending API, where blend equations can be specified via libgxm, shaders compiled with psp2cgc do not write the final shader output by default. Instead, the new color data is written into temporary storage so that additional code generated by libgxm can read this value and perform the blending.

As a result of this, shaders need to be patched by libgxm at runtime even if no blending is ultimately performed. In some of these cases, this may also result in an additional instruction being appended even though it would not be required had the shader written directly into the output storage.

Users can avoid this patching by using the `__nativecolor` qualifier. When specified as part of the entry function declaration, psp2cgc will generate a shader that writes into the output storage directly. This means however that blending may no longer be specified via libgxm for this shader.

In addition, the fragment's `COLOR0` output must be declared with the `__regformat` specifier as libgxm cannot insert code that would implicitly convert the default `half4` format to one that is compatible with the color surface used.

```
usampler2D smp;
__nativecolor __regformat unsigned char4 main(half2 coord : TEXCOORD0) {
    return tex2D<unsigned char4>(smp, coord);
}
```

Note: the `__nativecolor` specifier must be used only with entry functions as it is a property that applies to the entire shader.

Programmable Blending

User defined programmable blending is the combination of a number of extensions. Before reading this section, please familiarize yourself with `__nativecolor` and `__regformat` as they are fundamental to this feature.

Because the `__nativecolor` specifier grants us direct access to the shader output storage, we are also able to read back its contents prior to writing to it. This previous color value is accessible via the `FRAGCOLOR` semantic. It's important to understand that the `FRAGCOLOR` input and `COLOR0` output refer to the same hardware resource – so both must be declared to be the same size.

The following example shows a trivial additive blending case:

```
__nativecolor void main(
    unsigned char4 newColor : COLOR0,
    __regformat unsigned char4 lastColor : FRAGCOLOR,
    out __regformat unsigned char4 outColor : COLOR0)
{
    outColor = newColor + lastColor;
}
```

Programmable Blending With MSAA

Using programmable blending with MSAA requires that users know at compile time whether MSAA will be used with the shader at runtime, in addition to the output register format provided by the `__nativecolor` extension.

In the normal case, the qualifier `__msaa` should be used on the entry-point to inform the shader compiler that the output will be blended while MSAA is active.

The following example shows a trivial additive blending case with MSAA:

```
__nativecolor __msaa main(
    unsigned char4 newColor : COLOR0,
    __regformat unsigned char4 lastColor : FRAGCOLOR,
    out __regformat unsigned char4 outColor : COLOR0)
{
    outColor = newColor + lastColor;
}
```

The compiler will split execution of the shader into two phases; the first phase run at pixel rate and will execute all statements that have no dependency on the `FRAGCOLOR` value, the second phase run at selective rate (for each valid sample) and will execute all code depending on the `FRAGCOLOR` value.

Shader code executed at selective rate is not allowed to execute a discard statement or to write to the `DEPTH` pixel output binding. Moreover, any instruction that requires gradient calculation is forbidden. In these cases the shader compiler will emit a compilation error; if the shader must sample a texture in the selective phase of a shader, it should do so using one of the texture sample functions that do not implicitly compute gradients.

The qualifier `__msaa` is actually an alias for `__msaa4x`, which signifies that the blending phase can be used with both 2xMSAA and 4xMSAA at runtime. The number of temporary registers available during this selective rate blending phase is quite limited in this mode. To make more temporary registers available in this phase, the qualifier `__msaa2x` can be used, but the resulting program may then only be used with 2xMSAA at runtime. If register limits are not hit in `__msaa4x` mode, which is expected to be the case for most fragment programs, then there is no code generation difference between `__msaa4x` and `__msaa2x`.

Offline Vertex Unpacking

In order to allow for a traditional shader API where the in memory format of vertex attributes is specified only at runtime, the compiler always assumes that input vertex attributes are provided with full precision floating point format; libgxm prepends runtime generated code to the shader body, that converts each vertex attribute from the in memory format to `float4` format.

If the user knows in what format vertex attributes will be stored in memory they can also choose to generate a shader which is specialized for his particular vertex attribute setup.

Vertex attribute format specialization has few advantages:

- Redundant format conversions are removed, for example skinning indices do not get converted from integer to floating point and back again before being used as an index.
- The compiler can often generate more efficient unpack code using vector instructions.
- Redundant move operations are eliminated: often the compiler can unpack directly into output registers.

The shader programmer uses the `__regformat` qualifier to specify the runtime format of a vertex attribute. A mix of `__regformat` and non `__regformat` vertex attributes can also be used inside the same shader, for such cases libgxm will generate conversion code only for the non `__regformat` attributes.

Programmers should be aware that it is their responsibility to ensure that the vertex attribute format used at runtime matches the format declared in the shader.

The following example shows how this syntax can be used to represent vertex attributes with arbitrary formats:

```
float4 mainVP(
    __regformat unsigned int bitPackedData[8],
    __regformat unsigned char4 boneIndex
    ) : POSITION
{
    ...
}
```

Avoiding Symbols Dead Stripping

By default the shader compiler strips all unused uniforms which are part of the default uniform buffer as well as all unused input vertex attributes.

To avoid dead stripping of symbols the `__nostrip` keyword can be used on uniforms and vertex attributes, as in the following example:

```
// do not strip uniform1
float4 __nostrip uniform1;

float4 mainVP(
    // do not strip vertex attributes
    float4 __nostrip __regformat vertexAttribute1,
    half2 __nostrip __regformat vertexAttribute2,
    ...
    ) : POSITION
{
    ...
}
```

For vertex attributes, this feature can be used to ensure that attributes declared consecutively will always be assigned to consecutive registers, assuming no registers are required for alignment. This allows for multiple consecutive attributes in memory to be transferred to the shader as a single operation, by treating them as a single large untyped attribute at runtime.

Please note that needlessly increasing the number of vertex attributes or uniforms using the `__nostrip` keyword increases register pressure, which can impact performance.

Interpolant Precision and Parameter Buffer Packing Format

The GPU expects that all vertex program outputs are provided at `float` precision. However, when vertex program outputs are packed by the hardware into the parameter buffer, texture coordinates can either be packed as `float` values or `half` values.

In order to control this packing operation without affecting the precision of computation (which can risk adding redundant format conversion operations), this packing format must be specified as part of the vertex program output semantic for each texture coordinate. The type of the output variable is ignored.

To choose `half` as the format for parameter buffer storage shader programmers can use the `TEXCOORD#_HALF` syntax, in the following example `vtex0` will be stored in the parameter buffer using `half` precision:

```
void vmain (
  in float4 vin: POSITION,
  out float4 vout: HPOS,
  out float4 vtex0 : TEXCOORD0_HALF)
{
    ...
}
```

Please note that this format is only used to encode the value at each vertex. Interpolation for the fragment program is always done at full precision, regardless of whether the parameter buffer format is `float` or `half`. For example, if a vertex program is passing a half texture coordinate straight through to a fragment program, `TEXCOORD#_HALF` should always be used as there will be no quality difference.

Note that if the vertex output attribute is declared with half precision, then the `TEXCOORD#_HALF` semantic is redundant since the compiler will use this information to decide the format of the attribute inside the parameter buffer. Programmers should be aware that, as part of the hardware design, every vertex program must write their final results as 32-bit floating point values regardless of the format used for the parameter buffer. Declaring an output as half will likely cause the compiler to insert a conversion instruction, at the end of the shader, that converts the value from half to floating point. Due to these extra conversion operations, when specifying that a texture coordinate should be stored as half in the parameter buffer, it is better to declare as `floatN` and use `TEXCOORD#_HALF` then declare its variable at half precision.

fastblend Operations

`psp2cgc` provides a number of unique intrinsics that can be used to perform blending operations in shader code. The use of these intrinsics is strongly advised since they allow the compiler to select the best instructions sequence to match the quality and performance of the equivalent run-time blending.

Important: the blend intrinsics are available only for low precision types `unsigned char` and `fixed` with 1 to 4 coefficients. Furthermore: whilst these functions are most commonly used in the context of programmable blending, these intrinsics may be used for arbitrary computations.

Additionally the result of the fastblend instrinsic is always saturated and, in the case of the `unsigned char` overload, the operation is carried out as if the operands were 0:8 fixed point values as opposed to 8-bit twos-complement integer.

fastblend

```
TYPE __fastblend(
    TYPE src1,
    TYPE src2,
    int colorBlendOperation,
    int colorFactor1,
    int colorFactor2,
    int alphaBlendOperation,
    int alphaFactor1,
    int alphaFactor2);
```

A number of pre-defined macros can be used to select the blend operation and blend factors. For `colorBlendOperation` and `alphaBlendOperation`, these are:

```
__BLEND_OP_ADD
__BLEND_OP_SUB
__BLEND_OP_MIN
__BLEND_OP_MAX
```

For `colorFactor1` and `colorFactor2`, possible blend factors are:

```
__BLEND_FACTOR_ZERO
__BLEND_FACTOR_ONE
__BLEND_FACTOR_SRC1
__BLEND_FACTOR_ONE_MINUS_SRC1
__BLEND_FACTOR_SRC2
__BLEND_FACTOR_ONE_MINUS_SRC2
__BLEND_FACTOR_SRC1_ALPHA
__BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA
__BLEND_FACTOR_SRC2_ALPHA
__BLEND_FACTOR_ONE_MINUS_SRC2_ALPHA
```

For `alphaFactor1` and `alphaFactor2`, possible blend factors are:

```
__BLEND_FACTOR_ZERO
__BLEND_FACTOR_ONE
__BLEND_FACTOR_SRC1_ALPHA
__BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA
__BLEND_FACTOR_SRC2_ALPHA
__BLEND_FACTOR_ONE_MINUS_SRC2_ALPHA
```

Please note that, due to the nature of the underlying native instruction, when `__BLEND_OP_MIN` or `__BLEND_OP_MAX` are used, the blending factors are completely ignored.

The following example illustrates an optimal shader that performs a non-dependent texture read as well as blend operation:

```
usampler2D mySampler;
__nativecolor __regformat unsigned char4 main(
    __regformat unsigned char4 fragColor : FRAGCOLOR,
    float2 vTexCoord : TEXCOORD0
    ) : COLOR0
{
    unsigned char4 color = tex2D<unsigned char4>(mySampler, vTexCoord);
    return __fastblend(
        color,
        fragColor,
        __BLEND_OP_ADD,
        __BLEND_FACTOR_SRC1_ALPHA,
        __BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA,
        __BLEND_OP_ADD,
        __BLEND_FACTOR_SRC1_ALPHA,
        __BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA);
}
```

fastblend3

```
TYPE __fastblend3(
    TYPE src1,
    TYPE src2,
    TYPE src3,
    int colorBlendOperation,
    int colorFactor1,
    int colorFactor2,
    int alphaBlendOperation,
    int alphaFactor1);
```

A number of pre-defined macros can be used to select the blend operation and blend factors. For `colorBlendOperation` and `alphaBlendOperation`, these are:

```
__BLEND_OP_ADD
__BLEND_OP_SUB
```

For `colorFactor1` and `colorFactor2`, possible blend factors are:

```
__BLEND_FACTOR_ZERO
__BLEND_FACTOR_ONE
__BLEND_FACTOR_SRC1
__BLEND_FACTOR_ONE_MINUS_SRC1
__BLEND_FACTOR_SRC2
__BLEND_FACTOR_ONE_MINUS_SRC2
__BLEND_FACTOR_SRC3
__BLEND_FACTOR_ONE_MINUS_SRC3
__BLEND_FACTOR_SRC1_ALPHA
__BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA
__BLEND_FACTOR_SRC2_ALPHA
__BLEND_FACTOR_ONE_MINUS_SRC2_ALPHA
__BLEND_FACTOR_SRC3_ALPHA
__BLEND_FACTOR_ONE_MINUS_SRC3_ALPHA
```

For `alphaFactor1`, possible blend factors are:

```
__BLEND_FACTOR_ZERO
__BLEND_FACTOR_ONE
__BLEND_FACTOR_SRC1_ALPHA
__BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA
__BLEND_FACTOR_SRC2_ALPHA
__BLEND_FACTOR_ONE_MINUS_SRC2_ALPHA
__BLEND_FACTOR_SRC3_ALPHA
__BLEND_FACTOR_ONE_MINUS_SRC3_ALPHA
```

Please note that, due to the nature of the underlying native instruction, the `alphaFactor2` cannot be specified, its value will be implicitly set to the same value used for `colorFactor2` .

fastblend with writemask

An additional intrinsic function is provided, which allows for blending and color masking to be performed with only one instruction. The only limitation when compared with the `__fastblend` intrinsic is that the alpha factors must be the same as the color ones (hence they do not have a separate function argument).

```
unsigned TYPE __fastblend_writemask(
    TYPE src1,
    TYPE src2,
    int colorBlendOperation,
    int factor1,
    int factor2
    int alphaBlendOperation,
    TYPE passThrough,
    int writeMask);
```

In addition to the parameters of the `__fastblend` intrinsic, `__fastblend_writemask` accepts a previous value and a bitmask to be used as a per-component writemask. The final result will be the blend where the writemask bit is set, otherwise the component is taken from the pass-through parameter.

The same restriction with regard to `__BLEND_OP_MIN` and `__BLEND_OP_MAX` which are in place for `__fastblend` applies also to __fastblend_writemask: if `__BLEND_OP_MIN` or `__BLEND_OP_MAX` operation is used then blending factors are completely ignored .

The `writemask` can be constructed by or-ing together these pre-defined macros:

```
__BLEND_WMASK_R
__BLEND_WMASK_G
__BLEND_WMASK_B
__BLEND_WMASK_A
__BLEND_WMASK_RGB
__BLEND_WMASK_RGBA
```

Example usage:

```
usampler2D mySampler;
__nativecolor __regformat unsigned char4 main(
    __regformat unsigned char4 fragColor : FRAGCOLOR,
    float2 vTexCoord : TEXCOORD0
    ) : COLOR0
{
    unsigned char4 color = tex2D<unsigned char4>(mySampler, vTexCoord);
    return __fastblend_writemask(
        color,
        fragColor,
        __BLEND_OP_ADD,
        __BLEND_FACTOR_SRC1_ALPHA,
        __BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA,
        __BLEND_OP_ADD,
        fragColor,
        __BLEND_WMASK_RGB);
}
```

Saturated U8 Operations

The USSE has native instructions for performing 4-way SIMD addition and multiplications on 8-bit integer with a saturated result. psp2cgc exposes two specific intrinsic functions to take advantage of them:

```
unsigned char4 __add_sat(unsigned char4 op0, unsigned char4 op1);
unsigned char4 __mul_sat(unsigned char4 op0, unsigned char4 op1);
```

Note that the `__mul_sat` intrinsic will perform a normal twos-complement multiplication and as such cannot be emulated with a `__fastblend` since the latter performs a fixed point multiplication (multiply and shift). The `__add_sat` intrinsic can be emulated via `__fastblend`, but is provided as a short cut.

It is also important to note that using `__mul_sat` is different from writing `saturate(uchar4 * uchar4)`. The compiler is technically not allowed to fold the saturate into the multiplication since the result of the multiplication is expected to have normal wrap-around semantic.

Tile and Pixel Coordinates

When executing fragment programs the USSE has access to additional read-only registers that contain data pertaining to the current fragment that is being shaded. psp2cgc exposes a number of intrinsic functions to facilitate access to these registers:

```
unsigned char2 __tile_xy(void);
unsigned short __pixel_x(void);
unsigned short __pixel_y(void);
```

The `__tile_xy()` intrinsic function returns an `unsigned char2` containing the pixel coordinates of the top-left of the tile currently being shaded, counted in 16 pixel units. For programs executing in non-MSAA mode this has the implication that the function will only assume even values. The `__pixel_x()` and `__pixel_y()` intrinsic functions return an `unsigned short` containing the pixel coordinates within the current tile. For programs executing in MSAA mode, the x and y pixel coordinate is the same for each sample of a pixel.

Table 15 Ranges of __tile_xy for different MSAA modes

MSAA Mode	Valid Range
No MSAA	[0..254] (only even values), [0..254] (only even values)
2x MSAA	[0..254] (only even values), [0..255]
4x MSAA	[0..255], [0..255]

Table 16 Ranges of __pixel_x for different MSAA modes

MSAA Mode	Valid Range
No MSAA	[0..31]
2x MSAA	[0..31]
4x MSAA	[0..15]

Table 17 Ranges of __pixel_y for different MSAA modes

MSAA Mode	Valid Range
No MSAA	[0..31]
2x MSAA	[0..15]
4x MSAA	[0..15]

These intrinsic functions can be very useful when calculating the index of the current fragment within a writable user declared uniform buffer.

```
unsigned int calculatePixelIndexLinear(const unsigned short stride) {
    const unsigned short tileMultiplier = 16;
    const unsigned int x = tileMultiplier * __tile_xy().x + __pixel_x();
    const unsigned int y = tileMultiplier * __tile_xy().y + __pixel_y();
    const unsigned int pixelIndex = y * stride + x;
    return pixelIndex;
}

// calculating the index for a linear surface.
auxSurface[calculatePixelIndexLinear(stride)] = someData;
```

Special Texture Query Functions

texXD_info

```
unsigned char4 tex1D_info(sampler1D,  float1 uv);
unsigned char4 tex1D_info(usampler1D, float1 uv);
unsigned char4 tex1D_info(isampler1D, float1 uv);
unsigned char4 tex1D_info(sampler1D,  float1 uv, float1 dx, float1 dy);
unsigned char4 tex1D_info(usampler1D, float1 uv, float1 dx, float1 dy);
unsigned char4 tex1D_info(isampler1D, float1 uv, float1 dx, float1 dy);
unsigned char4 tex1Dlod_info(sampler1D, float4 uv);
unsigned char4 tex1Dlod_info(usampler1D, float4 uv);
unsigned char4 tex1Dlod_info(isampler1D, float4 uv);
unsigned char4 tex1Dbias_info(sampler1D, float4 uv);
unsigned char4 tex1Dbias_info(usampler1D, float4 uv);
unsigned char4 tex1Dbias_info(isampler1D, float4 uv);
unsigned char4 tex2D_info(sampler2D,  float2 uv);
unsigned char4 tex2D_info(usampler2D, float2 uv);
unsigned char4 tex2D_info(isampler2D, float2 uv);
unsigned char4 tex2D_info(sampler2D,  float2 uv, float2 dx, float2 dy);
unsigned char4 tex2D_info(usampler2D, float2 uv, float2 dx, float2 dy);
unsigned char4 tex2D_info(isampler2D, float2 uv, float2 dx, float2 dy);
unsigned char4 tex2Dlod_info(sampler2D, float4 uv);
unsigned char4 tex2Dlod_info(usampler2D, float4 uv);
unsigned char4 tex2Dlod_info(isampler2D, float4 uv);
```

```
unsigned char4 tex2Dbias_info(sampler2D, float4 uv);
unsigned char4 tex2Dbias_info(usampler2D, float4 uv);
unsigned char4 tex2Dbias_info(isampler2D, float4 uv);
unsigned char4 texCUBE_info(samplerCUBE,  float3 uv);
unsigned char4 texCUBE_info(usamplerCUBE, float3 uv);
unsigned char4 texCUBE_info(isamplerCUBE, float3 uv);
unsigned char4 texCUBE_info(samplerCUBE,  float3 uv, float3 dx, float3 dy);
unsigned char4 texCUBE_info(usamplerCUBE, float3 uv, float3 dx, float3 dy);
unsigned char4 texCUBE_info(isamplerCUBE, float3 uv, float3 dx, float3 dy);
unsigned char4 texCUBElod_info(samplerCUBE, float4 uv);
unsigned char4 texCUBElod_info(usamplerCUBE, float4 uv);
unsigned char4 texCUBElod_info(isamplerCUBE, float4 uv);
unsigned char4 texCUBEbias_info(samplerCUBE, float4 uv);
unsigned char4 texCUBEbias_info(usamplerCUBE, float4 uv);
unsigned char4 texCUBEbias_info(isamplerCUBE, float4 uv);
```

The `texXD_info` family of functions allow programmers to query filtering information for a specified texture query, without actually performing the query itself.

All `texXD_info` functions return an `unsigned char4` value so formed:

Table 18 texXD_info Result Components

Result Component	Content
`result.x`	Bilinear U fraction
`result.y`	Bilinear V fraction
`result.z`	Trilinear fraction
`result.w`	LOD value used for mipmap selection

Gather Texture Query Functions

tex2D_gather4

```
void tex2D_gather4(sampler2D, float2 uv, out float[4]);
void tex2D_gather4(sampler2D, float2 uv, out float2[4]);
void tex2D_gather4(sampler2D, float2 uv, out float4[4]);
void tex2D_gather4(sampler2D, float2 uv, out half[4]);
void tex2D_gather4(sampler2D, float2 uv, out half2[4]);
void tex2D_gather4(sampler2D, float2 uv, out half4[4]);
void tex2D_gather4(usampler2D, float2 uv, out unsigned char[4]);
void tex2D_gather4(usampler2D, float2 uv, out unsigned char2[4]);
void tex2D_gather4(usampler2D, float2 uv, out unsigned char4[4]);
void tex2D_gather4(isampler2D, float2 uv, out signed char[4]);
void tex2D_gather4(isampler2D, float2 uv, out signed char2[4]);
void tex2D_gather4(isampler2D, float2 uv, out signed char4[4]);
void tex2D_gather4(usampler2D, float2 uv, out unsigned short[4]);
void tex2D_gather4(usampler2D, float2 uv, out unsigned short2[4]);
void tex2D_gather4(usampler2D, float2 uv, out unsigned short4[4]);
void tex2D_gather4(isampler2D, float2 uv, out signed short[4]);
void tex2D_gather4(isampler2D, float2 uv, out signed short2[4]);
void tex2D_gather4(isampler2D, float2 uv, out signed short4[4]);
void tex2D_gather4(usampler2D, float2 uv, out unsigned int[4]);
void tex2D_gather4(usampler2D, float2 uv, out unsigned int2[4]);
void tex2D_gather4(usampler2D, float2 uv, out unsigned int4[4]);
void tex2D_gather4(isampler2D, float2 uv, out signed int[4]);
void tex2D_gather4(isampler2D, float2 uv, out signed int2[4]);
void tex2D_gather4(isampler2D, float2 uv, out signed int4[4]);
void tex2D_gather4(sampler2D, float2 uv, float2 dx, float2 dy, out float[4]);
void tex2D_gather4(sampler2D, float2 uv, float2 dx, float2 dy, out float2[4]);
void tex2D_gather4(sampler2D, float2 uv, float2 dx, float2 dy, out float4[4]);
void tex2D_gather4(sampler2D, float2 uv, float2 dx, float2 dy, out half[4]);
```

```
void tex2D_gather4(sampler2D, float2 uv, float2 dx, float2 dy, out half2[4]);
void tex2D_gather4(sampler2D, float2 uv, float2 dx, float2 dy, out half4[4]);
void tex2D_gather4(usampler2D, float2 uv, float2 dx, float2 dy, out unsigned
char[4]);
void tex2D_gather4(usampler2D, float2 uv, float2 dx, float2 dy, out unsigned
char2[4]);
void tex2D_gather4(usampler2D, float2 uv, float2 dx, float2 dy, out unsigned
char4[4]);
void tex2D_gather4(isampler2D, float2 uv, float2 dx, float2 dy, out signed
char[4]);
void tex2D_gather4(isampler2D, float2 uv, float2 dx, float2 dy, out signed
char2[4]);
void tex2D_gather4(isampler2D, float2 uv, float2 dx, float2 dy, out signed
char4[4]);
void tex2D_gather4(usampler2D, float2 uv, float2 dx, float2 dy, out unsigned
short[4]);
void tex2D_gather4(usampler2D, float2 uv, float2 dx, float2 dy, out unsigned
short2[4]);
void tex2D_gather4(usampler2D, float2 uv, float2 dx, float2 dy, out unsigned
short4[4]);
void tex2D_gather4(isampler2D, float2 uv, float2 dx, float2 dy, out signed
short[4]);
void tex2D_gather4(isampler2D, float2 uv, float2 dx, float2 dy, out signed
short2[4]);
void tex2D_gather4(isampler2D, float2 uv, float2 dx, float2 dy, out signed
short4[4]);
void tex2D_gather4(usampler2D, float2 uv, float2 dx, float2 dy, out unsigned
int[4]);
void tex2D_gather4(usampler2D, float2 uv, float2 dx, float2 dy, out unsigned
int2[4]);
void tex2D_gather4(usampler2D, float2 uv, float2 dx, float2 dy, out unsigned
int4[4]);
void tex2D_gather4(isampler2D, float2 uv, float2 dx, float2 dy, out signed int[4]);
void tex2D_gather4(isampler2D, float2 uv, float2 dx, float2 dy, out signed
int2[4]);
void tex2D_gather4(isampler2D, float2 uv, float2 dx, float2 dy, out signed
int4[4]);
```

The `tex2D_gather4` family of functions allows programmers to make a texture query and retrieve pixel data before any filtering is performed by the hardware. Each version of the function has one output parameter, which will contain, on exit, the contents of the four pixels surrounding the specified `uv` coordinates.

This function can be used to efficiently implement special texture filter functions: for example, one could downsample a depth buffer using min or max downsampling functions without having to issue four separate texture queries and associated `uv` coordinate computations.

For floating-point result types, the GPU exposes an additional specialization of the function, which also allows the programmer to obtain the bilinear filtering coefficients that would have been used if the hardware were to perform the filtering itself. This additional specialization is defined as follows:

```
void tex2D_gather4(sampler2D, float2 uv, out float[4] pixels, out half4 coeffs);
void tex2D_gather4(sampler2D, float2 uv, float2 dx, float2 dy, out float[4] pixels,
out half4 coeffs);
```

4 Pragmas

This chapter describes the pragmas supported by psp2cgc.

Uniform Buffer Locations

Uniform buffers which are not marked with the `readwrite_buffer` pragma and declared using the `BUFFER` semantic, may be located in either memory or registers. To set the location for a buffer, psp2cgc supports `register_buffer` and `memory_buffer` pragmas.

Note: If no location is specified explicitly, the buffer will be located in memory.

```
#pragma register_buffer BUFFER[0] // sets uniform buffer 0 to be in registers
#pragma memory_buffer BUFFER1     // sets uniform buffer 1 to be in memory
```

If multiple settings for the same buffer are provided in the same file, the location provided in the last pragma is used throughout.

If a uniform buffer is marked with the `readwrite_buffer` pragma then it will always be located in memory. psp2cgc does not require you to explicitly specify the location for the buffer.

```
#pragma readwrite_buffer BUFFER0     // sets uniform buffer 0 to be writable
float auxSurface[256*256] : BUFFER0; // writable buffer, implicitly in memory
```

Uniform Buffer Symbols

Uniform buffers which are not marked with the `readwrite_buffer` pragma and declared using the `BUFFER` semantic will by default generate symbol data which is included in the resulting GXP file. However, if a user declared uniform buffer is marked with the `readwrite_buffer` pragma then by default no symbol data is included in the resulting GXP file. Generation of symbol data can be enabled or disabled by using the `enable_buffer_symbols` and `disable_buffer_symbols` pragmas respectively. It should be noted that using the `enable_buffer_symbols` pragma can significantly increase the size of the GXP file created by psp2cgc as each element of an array of structures will be present as a symbol in the GXP file.

```
float4x4 skinMatrices[64] : BUFFER0; // symbols will be generated.

#pragma disable_buffer_symbols BUFFER1
float4x4 skinMatrices[64] : BUFFER1; // symbols will not be generated.

#pragma readwrite_buffer BUFFER2
float auxSurface[256*256] : BUFFER2; // symbols will not be generated.

#pragma enable_buffer_symbols BUFFER3
#pragma readwrite_buffer BUFFER3
float auxSurface[256*256] : BUFFER3; // symbols will be generated.
```

Uniform Buffer Dead Stripping

User defined uniform buffers are, by default, never stripped of dead symbols.

This is to ensure that the binary layout of a uniform buffer matches specific data structures declared using the CPU compiler. If such behavior is *not* required it is possible to enable dead stripping of uniforms in user defined uniform buffer by using the following pragma:

```
#pragma strip_buffer BUFFER[0] // enable dead stripping of uniform buffer 0
```

Warnings

To change the severity of warnings, psp2cgc supports the following pragma syntax:

```
#pragma warning (push)
#pragma warning (pop)
#pragma warning (specifier : warning_ids [; specifier : warning_ids ...])
```

The `push` and `pop` operations can be used to store and restore the state of the diagnostics.

The third line of syntax shown above uses a `specifier` to achieve a particular effect. Table 19 lists all the specifiers supported by psp2cgc:

Table 19 Warning pragma Specifiers

Specifier	Description
`default`	Resets the severity of a warning to the default. Always enables the warning.
`disable`	Warnings of the specified type will no longer be emitted.
`error`	The specified warnings will be treated as errors.

The `warning_ids` are a white-space separated list of the numbers that are found in the psp2cgc output; for example:

```
shader.cg(1,1) : warning D5203: parameter 'arg' is unreferenced
```

where `5203` is the warning ID.

Note: The "`D`" prefix in the message serves only to clarify the meaning of the number; therefore, it should be omitted from the `warning_ids` list.

For warnings with an ID greater than 6000, which are associated with code generation, the warning settings that are active at the beginning of the function will be used. If a warning pragma appears in the body of a function, it will take effect after the end of the function.

```
// Disable warnings 5203, 5204 and treat 5205 as an error
#pragma warning (disable: 5203 5204; error: 5205)

// Disable warning 5203 only for a certain function
#pragma warning (push)
#pragma warning (disable: 5203)
... function declaration ...
#pragma warning (pop)
```

Loop Code Generation

To provide users with more detailed control over the loop code being generated, psp2cgc supports the following pragma syntax:

```
#pragma loop (unroll: always)
#pragma loop (unroll: never)
#pragma loop (unroll: default)
```

The pragma always applies to the next loop only. If multiple pragmas are specified before a loop is found, the last setting takes precedence.

In the case where `default` is specified, the compiler determines whether or not to unroll the loop.

Note: The `always` hint can be used only if the trip count of the loop can be determined at compile time. If this is not the case, a warning is emitted, indicating that the hint could not be applied (see the following example).

```
uniform float myUniform;
...

// this loop will not be unrolled
#pragma loop (unroll: never)
for (int i=0; i<10; ++i)
{
    // this loop will be unrolled with a trip count of 10
#   pragma loop (unroll: always)
    for (int j=0; j<10; ++j)
        ...

    // this loop will not be unrolled, as the trip count is unknown
    for (int k=0; k<(int)myUniform; ++k)
        ...
}
```

Branch Code Generation

The compiler provides detailed user control on whether branches should be kept in final code or flattened by using conditional moves or predication. psp2cgc supports the following pragma syntax:

```
#pragma branch (flatten: always)
#pragma branch (flatten: never)
#pragma branch (flatten: default)
```

The pragma always applies to the next conditional block only. If multiple pragmas are specified before a condition is found, the last setting takes precedence.

In the case where `default` is specified, the compiler determines whether or not to flatten the branch.

Note: Branch pragmas are always considered hints for the compiler, programmers should look at final generated code using psp2shaderperf in order to verify the expected result.

Command-line Arguments

Some command-line arguments may be specified inside the shader code, using argument pragmas. These pragmas behave like command-line arguments in that they apply to the entire shader.

If the same property is changed by more than one pragma, the last value is used for the entire program.

The syntax for argument pragmas is:

```
#pragma argument (specifier [; specifier ...])
```

where `specifier` may be one of the following values:

```
O0-O4, fastmath, nofastmath, fastprecision, nofastprecision, bestprecision,
invpos
```

For details about the effect of these properties, refer to Table 3, Code Generation Compiler Options.

The following example shows how to use argument pragmas to ensure that a program uses all possible arithmetic optimizations:

```
#pragma argument (O3; fastmath; fastprecision)
```

Default Matrix Ordering

The default storage order of matrices is `row_major` as in other Cg implementations. However, this can be changed using the following pragma:

```
#pragma pack_matrix (column_major) // sets default order to column major
#pragma pack_matrix (row_major)    // sets default order to row major
```

The pragma may be used multiple times throughout a program and will affect all following declarations.

Position Invariance

When `-fastmath` and/or `-fastprecision` are used, it is possible for the compiler to generate slightly different permutation for the code that calculates the output position of a vertex program for different shaders, even when the original shader code for this calculation is equivalent.

The drawback is that the LSBs of the computed output position can differ between different shaders, which can lead to graphical artifacts when the user relies on the position computation to be invariant.

To avoid this problem it is possible to tell the compiler to generate a position invariant vertex program:

```
#pragma position_invariant <func>
```

The pragma argument is the name of the vertex program entry function.

Bank Clash Adjustment

Typically, the lower the number of registers that a shader requires, the higher the number of threads (pixel or vertex) can be active concurrently, making the hiding of memory latency more effective. For this reason, psp2cgc always tries to minimize the number of registers used while compiling a shader.

However, due to the implementation of register banking in the GPU, there are extreme cases where increasing the number of registers that a shader requires actually reduces bank clashes and leads to improved overall performance.

Starting from SDK 2.100, the shader patcher (a component of the runtime graphics library, libgxm) will increase the number of registers required by a shader if it detects such cases.

This adjustment can be disabled by using the following pragma:

```
#pragma disable_bank_clash_adjustment
```

5 Inspecting GXP Files with cgnm

This chapter describes the `psp2cgnm` utility provided with the shader compiler.

Command-line Usage

The basic syntax for retrieving the symbol information associated with a GXP program is:

```
psp2cgnm input.gxp
```

A list of all the symbol information stored for `input.gxp` is emitted to standard out.

Command-line Options

Table 20 describes the options that can be used with `psp2cgnm`.

Table 20 Basic Options

Option	Description
`-h / -help`	Prints a brief command-line reference.
`-v / -version`	Prints the version number.

Parameter Syntax

The parameter information emitted by `psp2cgnm` is equivalent to that which can be queried using the functions described in Chapter 9, "Using the Shader API" in the *libgxm Overview*.

Note: The output has a one-to-one correspondence with the `SceGxmProgramParameter` entries in the `GXP` file.

Table 21 Parameter Notation

Parameter Category	Notation
Vertex attribute	`attr <name> <semantic> <type> <resource>`
Sampler	`smp <name> <kind> <result_formats> <result_width> <resource>`
Uniform	`uni <name> <type> <array_size> <resource> <container>`
Uniform buffer	`unibuf <resource> <byte_size>`

Where a type is emitted, the Cg notation is used. The only exception to this is unsigned integral types: a single `u` is prepended instead of the full `unsigned` keyword. For example, `unsigned int4` is emitted as `uint4`.

For details regarding the meaning of the other parameter attributes, refer to the *libgxm Overview*.

Property Syntax

Properties apply to the shader as a whole and describe whether certain features or hardware resources are being used. For more details, refer to Chapter 9, "Using the Shader API" in the *libgxm Overview*.

Table 22 Property Notation

Notation	Description
`discard 1`	Emitted if a live discard statement was present in the shader.
`depth_write 1`	Emitted if the shader replaces the depth value of the shaded fragment.

6 Using psp2gxpstrip

This chapter describes the `psp2gxpstrip` utility provided with the shader compiler.

Command-line Usage

The basic syntax for removing symbol information from a GXP file is:

```
psp2gxpstrip -o output.gxp input.gxp
```

The resulting GXP file will be equivalent to the input GXP file, but all symbol information will have been removed from it. As a result the output GXP file should be smaller in size.

Command-line Options

Table 20 describes the options that can be used with `psp2gxpstrip`.

Table 23 Basic Options

Option	Description
`-h / -help`	Prints a brief command-line reference.
`-v / -version`	Prints the version number.
`-o <filename>`	Specifies the name of the file to be written.
`-lang <locale>`	Changes the display language for the messages printed by the tool. Valid locale settings are `en` and `jp`.

Appendix - Shader Compiler Reference

Introduction

The DLL version of psp2cgc allows tools to compile shaders without having to spawn an external process. Users may also choose to replace the native file system with a virtualized one, by using a set of callbacks.

The compiler may be invoked in one of three modes:

Table 24 Shader Compiler Modes

Mode	Description
scePsp2CgcCompileProgram	Performs a full compile, returning a ScePsp2CgcCompileOutput object containing both the binary and SDB.
scePsp2CgcPreprocessProgram	Performs only pre-processing and returns the resulting text.
scePsp2CgcGenerateDependencies	Generates a list of dependencies for the primary file provided.

Each mode has a corresponding destroy function: scePsp2CgcDestroyCompileOutput, scePsp2CgcDestroyPreprocessOutput and scePsp2CgcDestroyDependencyOutput.

These functions must be called in order to free the resources associated with a job. Creation and destruction do not have to be performed on the same thread. Each of these functions depends chiefly on two inputs:

ScePsp2CgcCompileOptions: This structure is the equivalent of the command-line options for the standalone compiler. It may also be used as a compilation ID by extending the struct and passing the appropriate pointer to psp2cgc. This pointer will be provided on all callbacks.

ScePsp2CgcCallbackList: Provides the compiler with an interface to the file system, whether it is the interface provided by the operating system or a virtualized one. For details regarding the individual callbacks, refer to the documentation of ScePsp2CgcCallbackList.

Note: If a callback list is provided, it should always be initialized via scePsp2CgcInitializeCallbackList.

Note: Where indicated, the callbacks argument is optional. If no callbacks are provided, the operating system's file system will be used.

Datatypes

ScePsp2CgcCallbackDefaults

Classifies default callbacks.

Definition

```
#include <psp2cgc.h>
typedef enum ScePsp2CgcCallbackDefaults {
    SCE_PSP2CGC_SYSTEM_FILES,
    SCE_PSP2CGC_TRIVIAL
} ScePsp2CgcCallbackDefaults;
```

Enumeration Values

Table 25 ScePsp2CgcCallbackDefaults Enumeration Values

Macro	Value	Description
SCE_PSP2CGC_SYSTEM_FILES	N/A	Default callback functions for using system files and paths.
SCE_PSP2CGC_TRIVIAL	N/A	Default callback functions for using only the "openFile" callback.

ScePsp2CgcDiagnosticLevel

Classifies the severity of a diagnostic.

Definition

```
#include <psp2cgc.h>
typedef enum ScePsp2CgcDiagnosticLevel {
    SCE_PSP2CGC_DIAGNOSTIC_LEVEL_INFO,
    SCE_PSP2CGC_DIAGNOSTIC_LEVEL_WARNING,
    SCE_PSP2CGC_DIAGNOSTIC_LEVEL_ERROR
} ScePsp2CgcDiagnosticLevel;
```

Enumeration Values

Table 26 ScePsp2CgcDiagnosticLevel Enumeration Values

Macro	Value	Description
SCE_PSP2CGC_DIAGNOSTIC_LEVEL_INFO	N/A	Informational message.
SCE_PSP2CGC_DIAGNOSTIC_LEVEL_WARNING	N/A	Warning.
SCE_PSP2CGC_DIAGNOSTIC_LEVEL_ERROR	N/A	Error.

ScePsp2CgcLocale

Classifies language.

Definition

```
#include <psp2cgc.h>
typedef enum ScePsp2CgcLocale {
    SCE_PSP2CGC_ENGLISH,
    SCE_PSP2CGC_JAPANESE
} ScePsp2CgcLocale;
```

Enumeration Values

Table 27 ScePsp2CgcLocale Enumeration Values

Macro	Value	Description
SCE_PSP2CGC_ENGLISH	N/A	English language setting.
SCE_PSP2CGC_JAPANESE	N/A	Japanese language setting.

ScePsp2CgcTargetProfile

Classifies the target profiles.

Definition

```
#include <psp2cgc.h>
typedef enum ScePsp2CgcTargetProfile {
    SCE_PSP2CGC_PROFILE_VP,
    SCE_PSP2CGC_PROFILE_FP
} ScePsp2CgcTargetProfile;
```

Enumeration Values

Table 28 ScePsp2CgcTargetProfile Enumeration Values

Macro	Value	Description
SCE_PSP2CGC_PROFILE_VP	N/A	Vertex program.
SCE_PSP2CGC_PROFILE_FP	N/A	Fragment program.

ScePsp2CgcParameterClass

Classifies the class of a shader parameter.

Definition

```
#include <psp2cgc.h>
typedef enum ScePsp2CgcParameterClass {
    ...
} ScePsp2CgcParameterClass;
```

Enumeration Values

Table 29 ScePsp2CgcParameterClass Enumeration Values

Macro	Value	Description
SCE_PSP2CGC_PARAMETERCLASS_INVALID	N/A	Invalid class.
SCE_PSP2CGC_PARAMETERCLASS_SCALAR	N/A	Scalar class.
SCE_PSP2CGC_PARAMETERCLASS_VECTOR	N/A	Vector class.
SCE_PSP2CGC_PARAMETERCLASS_MATRIX	N/A	Matrix class.
SCE_PSP2CGC_PARAMETERCLASS_STRUCT	N/A	Struct class.
SCE_PSP2CGC_PARAMETERCLASS_ARRAY	N/A	Array class.
SCE_PSP2CGC_PARAMETERCLASS_SAMPLER	N/A	Sampler class.
SCE_PSP2CGC_PARAMETERCLASS_UNIFORMBLOCK	N/A	Uniform block.

ScePsp2CgcParameterBaseType

Classifies the base type of a shader parameter.

Definition

```
#include <psp2cgc.h>
typedef enum ScePsp2CgcParameterBaseType {
    ...
} ScePsp2CgcParameterBaseType;
```

Enumeration Values

Table 30 ScePsp2CgcParameterBaseType Enumeration Values

Macro	Value	Description
SCE_PSP2CGC_BASETYPE_INVALID	N/A	Invalid type.
SCE_PSP2CGC_BASETYPE_FLOAT	N/A	32-bit floating point type.
SCE_PSP2CGC_BASETYPE_HALF	N/A	16-bit floating point type.
SCE_PSP2CGC_BASETYPE_FIXED	N/A	2:8 fixed point type.
SCE_PSP2CGC_BASETYPE_BOOL	N/A	Bool type.
SCE_PSP2CGC_BASETYPE_CHAR	N/A	8-bit integer type.
SCE_PSP2CGC_BASETYPE_UCHAR	N/A	Unsigned 8-bit integer type.
SCE_PSP2CGC_BASETYPE_SHORT	N/A	16-bit integer type.
SCE_PSP2CGC_BASETYPE_USHORT	N/A	Unsigned 16-bit integer type.
SCE_PSP2CGC_BASETYPE_INT	N/A	32-bit integer type.
SCE_PSP2CGC_BASETYPE_UINT	N/A	Unsigned 32-bit integer type.
SCE_PSP2CGC_BASETYPE_SAMPLER1D	N/A	1D sampler type.
SCE_PSP2CGC_BASETYPE_ISAMPLER1D	N/A	Signed integer 1D sampler type.
SCE_PSP2CGC_BASETYPE_USAMPLER1D	N/A	Unsigned integer 1D sampler type.
SCE_PSP2CGC_BASETYPE_SAMPLER2D	N/A	2D sampler type.
SCE_PSP2CGC_BASETYPE_ISAMPLER2D	N/A	Signed integer 2D sampler type.
SCE_PSP2CGC_BASETYPE_USAMPLER2D	N/A	Unsigned integer 2D sampler type.
SCE_PSP2CGC_BASETYPE_SAMPLERCUBE	N/A	Cube sampler type.
SCE_PSP2CGC_BASETYPE_ISAMPLERCUBE	N/A	Signed integer Cube sampler type.

Macro	Value	Description
SCE_PSP2CGC_BASETYPE_USAMPLERCUBE	N/A	Unsigned integer Cube sampler type.
SCE_PSP2CGC_BASETYPE_SAMPLERRECT	N/A	Rect sampler type.
SCE_PSP2CGC_BASETYPE_ISAMPLERRECT	N/A	Signed integer Rect sampler type.
SCE_PSP2CGC_BASETYPE_USAMPLERRECT	N/A	Unsigned integer Rect sampler type.
SCE_PSP2CGC_BASETYPE_ARRAY	N/A	Array type.
SCE_PSP2CGC_BASETYPE_STRUCT	N/A	Struct type.
SCE_PSP2CGC_BASETYPE_UNIFORMBLOCK	N/A	Uniform block type.

ScePsp2CgcParameterMemoryLayout

Classifies the memory layout of a matrix parameter.

Definition

```
#include <psp2cgc.h>
typedef enum ScePsp2CgcParameterMemoryLayout {
    ...
} ScePsp2CgcParameterMemoryLayout;
```

Enumeration Values

Table 31 ScePsp2CgcParameterMemoryLayout Enumeration Values

Macro	Value	Description
SCE_PSP2CGC_MEMORYLAYOUT_INVALID	N/A	Invalid memory layout.
SCE_PSP2CGC_MEMORYLAYOUT_COLUMN_MAJOR	N/A	Column major layout.
SCE_PSP2CGC_MEMORYLAYOUT_ROW_MAJOR	N/A	Row major layout.

ScePsp2CgcParameterVariability

Classifies the variability of a parameter.

Definition

```
#include <psp2cgc.h>
typedef enum ScePsp2CgcParameterVariability {
    ...
} ScePsp2CgcParameterVariability;
```

Enumeration Values

Table 32 ScePsp2CgcParameterVariability Enumeration Values

Macro	Value	Description
SCE_PSP2CGC_VARIABILITY_INVALID	N/A	Invalid variability.
SCE_PSP2CGC_VARIABILITY_VARYING	N/A	Parameter is varying.
SCE_PSP2CGC_VARIABILITY_UNIFORM	N/A	Parameter is uniform.

ScePsp2CgcParameterDirection

Classifies the direction of a parameter.

Definition

```
#include <psp2cgc.h>
typedef enum ScePsp2CgcParameterDirection {
    ...
} ScePsp2CgcParameterDirection;
```

Enumeration Values

Table 33 ScePsp2CgcParameterDirection Enumeration Values

Macro	Value	Description
SCE_PSP2CGC_DIRECTION_INVALID	N/A	Invalid direction.
SCE_PSP2CGC_DIRECTION_IN	N/A	Parameter is input.
SCE_PSP2CGC_DIRECTION_OUT	N/A	Parameter is output.

ScePsp2CgcCallbackList

Lists the user-defined callbacks for compiler operations.

Definition

```
#include <psp2cgc.h>
typedef struct ScePsp2CgcCallbackList {
    ScePsp2CgcCallbackOpenFile openFile;
    ScePsp2CgcCallbackReleaseFile releaseFile;
    ScePsp2CgcCallbackLocateFile locateFile;
    ScePsp2CgcCallbackAbsolutePath absolutePath;
    ScePsp2CgcCallbackReleaseFileName releaseFileName;
    ScePsp2CgcCallbackFileDate fileDate;
} ScePsp2CgcCallbackList;
```

Members

openFile	The callback used to open a file.
releaseFile	The callback used to release an opened file (optional).
locateFile	The callback used to indicate a file exists (optional).
absolutePath	The callback used to indicate the absolute path of a file (optional).
releaseFileName	The callback used to release an absolute path string (optional).
fileDate	The callback used to indicate file modification date (optional).

Description

The ScePsp2CgcCallbackList structure is used in each of scePsp2CgcCompileProgram, scePsp2CgcPreprocessProgram, and scePsp2CgcGenerateDependencies in the same fashion. In order to initialize instances of this structure, please always use scePsp2CgcInitializeCallbackList.

For details regarding the individual callbacks, please refer to their respective documentations.

ScePsp2CgcCompileOptions

Describes the input data for a compilation job.

Definition

```
#include <psp2cgc.h>
typedef struct ScePsp2CgcCompileOptions {
    const char *mainSourceFile;
    ScePsp2CgcTargetProfile targetProfile;
    const char *entryFunctionName;
    uint32_t searchPathCount;
    const char *const *searchPaths;
    uint32_t macroDefinitionCount;
    const char *const *macroDefinitions;
    uint32_t includeFileCount;
    const char *const *includeFiles;
    uint32_t suppressedWarningsCount;
    const char *const *suppressedWarnings;
    ScePsp2CgcLocale locale;
    int32_t useFx;
```

```
    int32_t noStdlib;
    int32_t optimizationLevel;
    int32_t useFastmath;
    int32_t useFastprecision;
    int32_t useFastint;
    int32_t positionInvariant;
    int32_t warningsAsErrors;
    int32_t performanceWarnings;
    int32_t warningLevel;
    int32_t pedantic;
    int32_t pedanticError;
    int32_t xmlCache;
    int32_t stripSymbols;

} ScePsp2CgcCompileOptions;
```

Members

mainSourceFile	The main Cg source file to compile.
targetProfile	The target profile.
entryFunctionName	The name of the entry function. Usually "main".
searchPathCount	The number of search paths for include files.
searchPaths	The search paths for include files.
macroDefinitionCount	The number of macro definitions provided.
macroDefinitions	The macro definitions in the form: MACRONAME or MACRONAME=VALUE.
includeFileCount	The number of files to force-include.
includeFiles	The files to include before the main source file.
suppressedWarningsCount	The total count of warnings to suppress.
suppressedWarnings	The numeric IDs of the warnings to suppress.
locale	The language to use in diagnostics.
useFx	Equivalent to -fx if non-zero, -nofx otherwise.
noStdlib	Equivalent to -nostdlib if non-zero.
optimizationLevel	Equivalent to -O?. Valid range is 0–4.
useFastmath	Equivalent to -fastmath if non-zero.
useFastprecision	Equivalent to -fastprecision if non-zero.
useFastint	Equivalent to -fastint if non-zero.
positionInvariant	Equivalent to -invpos if non-zero.
warningsAsErrors	Equivalent to -Werror if non-zero.
performanceWarnings	Equivalent to -Wperf if non-zero.
warningLevel	Equivalent to -W?. Valid range is 0–4.
pedantic	Equivalent to -pedantic if non-zero.
pedanticError	Equivalent to -pedantic-error if non-zero.
xmlCache	Equivalent to -xmlcache if non-zero.
stripSymbols	Set to non zero to generate a stripped GXP file. This option is equivalent to using the offline compiler to generate a GXP file and subsequently using psp2gxpstrip to strip the result of the compilation. See Chapter 6, Using psp2gxpstrip.

ScePsp2CgcCompileOutput

Describes the output of a compilation process.

Definition

```
#include <psp2cgc.h>
typedef struct ScePsp2CgcCompileOutput {
    const uint8_t *programData;
    uint32_t programSize;
    const char *sdbData;
```

```
    const char *sdbExt;
    int32_t sdbDataSize;
    int32_t diagnosticCount;
    const ScePsp2CgcDiagnosticMessage *diagnostics;
} ScePsp2CgcCompileOutput;
```

Members

programData	The compiled program binary data.
programSize	The compiled program size.
sdbData	The contents of the SDB file (optional).
sdbExt	The extension of the SDB file (optional).
sdbDataSize	The size in byte of the array pointed by the *sdbData* member.
diagnosticCount	The number of diagnostics.
diagnostics	The diagnostic message array.

Description

On failure, *programData* will be 0 and the *diagnosticCount* will be non-zero. On success, *programData* will be non-zero and one or more non-error diagnostics may be present.

ScePsp2CgcDependencyOutput

Describes the output of dependency generation.

Definition

```
#include <psp2cgc.h>
typedef struct ScePsp2CgcDependencyOutput {
    int32_t dependencyTargetCount;
    const char *const *dependencyTargets;
    int32_t diagnosticCount;
    const ScePsp2CgcDiagnosticMessage *diagnostics;
} ScePsp2CgcDependencyOutput;
```

Members

dependencyTargetCount	The number of dependencies.
dependencyTargets	The individual target rules.
diagnosticCount	The number of diagnostics.
diagnostics	The diagnostic message array.

ScePsp2CgcDiagnosticMessage

Describes a single compiler diagnostic.

Definition

```
#include <psp2cgc.h>
typedef struct ScePsp2CgcDiagnosticMessage {
    ScePsp2CgcDiagnosticLevel level;
    uint32_t code;
    const ScePsp2CgcSourceLocation *location;
    const char *message;
} ScePsp2CgcDiagnosticMessage;
```

Members

level	The severity of the diagnostic.
code	A unique code for each kind of diagnostic.
location	The location for which the diagnostic is reported (optional).
message	The diagnostic message.

ScePsp2CgcPreprocessOutput

Describes the output of a preprocessor process.

Definition

```
#include <psp2cgc.h>
typedef struct ScePsp2CgcPreprocessOutput {
    const char *program;
    uint32_t programSize;
    int32_t diagnosticCount;
    const ScePsp2CgcDiagnosticMessage *diagnostics;
} ScePsp2CgcPreprocessOutput;
```

Members

program	The preprocessor output.
programSize	The preprocessor output size.
diagnosticCount	The number of diagnostics.
diagnostics	The diagnostic message array.

Description

On failure, program will be 0 and the *diagnosticCount* will be non-zero. On success, program will be non-zero and one or more non-error diagnostics may be present.

ScePsp2CgcSourceLocation

Describes a location in the source code.

Definition

```
#include <psp2cgc.h>
typedef struct ScePsp2CgcSourceLocation {
    const ScePsp2CgSourceFile *file;
    uint32_t lineNumber;
    uint32_t columnNumber;
} ScePsp2CgcSourceLocation;
```

Members

file	The file containing the location.
lineNumber	The line number of the location.
columnNumber	The column number of the location.

ScePsp2CgSourceFile

Describes a source file.

Definition

```
#include <psp2cgc.h>
typedef struct ScePsp2CgSourceFile {
    const char *fileName;
    const char *text;
    uint32_t size;
} ScePsp2CgSourceFile;
```

Members

fileName	The relative or absolute name of the file.
text	The contents of the source file.
size	The size of the 'text' array in bytes.

ScePsp2CgcParameter

An opaque handle for a shader parameter.

Definition

```
#include <psp2cgc.h>
typedef void const * ScePsp2CgcParameter;
```

Functions

scePsp2CgcCompileProgram

Compiles a Cg program for the PlayStation®Vita system.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE ScePsp2CgcCompileOutput const *__cdecl
scePsp2CgcCompileProgram(
    const ScePsp2CgcCompileOptions *options,
    const ScePsp2CgcCallbackList *callbacks,
    void *userData
);
```

Arguments

options	Indicates the compile options for compiling a program. Also doubles as a session ID where multiple compiles are being run.
callbacks	Defines the callbacks to be used for file system access. If not provided, the default file system of the OS will be used (optional).
userData	An opaque pointer to user data that the compiler passes unaltered to each callback invoked.

Return Values

A ScePsp2CgcCompileOutput object, containing the binary and SDB outputs of the compile. Must be destroyed using scePsp2CgcDestroyCompileOutput. If 0 is returned, the input arguments were malformed.

Description

Compiles a Cg program for the PlayStation®Vita system using the options provided.

scePsp2CgcDestroyCompileOutput

Releases all allocations associated with a compiled program.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE void __cdecl scePsp2CgcDestroyCompileOutput(
    ScePsp2CgcCompileOutput const *output
);
```

Arguments

output	The result from a call to scePsp2CgcCompileProgram, to be destroyed.

Return Values

None

scePsp2CgcDestroyDependencyOutput

Releases all allocations associated with the dependency output.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE void __cdecl scePsp2CgcDestroyDependencyOutput(
    ScePsp2CgcDependencyOutput const *output
);
```

Arguments

output	The result from a call to scePsp2CgcGenerateDependencies, to be destroyed.

Return Values

None

scePsp2CgcDestroyPreprocessOutput

Releases all allocations associated with a compiled program.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE void __cdecl scePsp2CgcDestroyPreprocessOutput(
    ScePsp2CgcPreprocessOutput const *output
);
```

Arguments

output	The result from a call to scePsp2CgcPreprocessProgram, to be destroyed.

Return Values

None

scePsp2CgcGenerateDependencies

Generates dependency information for a program.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE ScePsp2CgcDependencyOutput const *__cdecl
scePsp2CgcGenerateDependencies(
    const ScePsp2CgcCompileOptions *options,
    const ScePsp2CgcCallbackList *callbacks,
    void *userData,
    const char *targetName,
    int32_t emitPhonies
);
```

Arguments

options	Indicates the compile options for generating the file dependencies for a program. Also doubles as a session ID where multiple compiles are being run.
callbacks	Defines the callbacks to be used for file system access. If not provided, the default file system of the OS will be used (optional).
userData	An opaque pointer to user data that the compiler passes unaltered to each callback invoked.
targetName	The name of the main target rule. Because we have no concept of the final binary name at this point, this must always be provided.
emitPhonies	Non-zero indicates that phony dependencies should be generated.

Return Values

A ScePsp2CgcDependencyOutput object, containing a list of file names and their dependent files. Should be destroyed using scePsp2CgcDestroyDependencyOutput. If 0 is returned, the input arguments were malformed.

Description

Generates dependency information for the given compilation options.

scePsp2CgcInitializeCallbackList

Initializes the callback list with the default values.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE void __cdecl scePsp2CgcInitializeCallbackList(
    ScePsp2CgcCallbackList *callbacks,
    ScePsp2CgcCallbackDefaults defaults
);
```

Arguments

callbacks	The callbacks struct to be initialized.
defaults	Indicates which set of default callbacks to initialize from.

Return Values

None

Description

There are two kinds of defaults available:

- SCE_PSP2CGC_SYSTEM_FILES uses the native file system of the operating system in the same manner as the command-line version of psp2cgc. This is the default behavior if no callback structure is provided for a compilation/pre-processing/dependency job.
- SCE_PSP2CGC_TRIVIAL provides placeholder implementations of all callbacks but 'openFile'. The latter must always be provided by the user.

In the trivial case, the following defaults will be used:

- *releaseFile*: Does nothing.
- *locateFile*: Returns the same path it was called with.
- *absolutePath*: Returns the same path it was called with.
- *releaseFileName*: Does nothing.
- *fileDate*: Returns the current time.

scePsp2CgcInitializeCompileOptions

Initializes the compile options with the default values.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE void __cdecl scePsp2CgcInitializeCompileOptions(
    ScePsp2CgcCompileOptions *options
);
```

Arguments

options	The option structure that should be initialized.

Return Values

None

Description

The following fields must be initialized by the user:

- *mainSourceFile*
- *targetProfile*

All other fields may be left in the state set by this function.

scePsp2CgcPreprocessProgram

Preprocesses a Cg program for the PlayStation®Vita system.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE ScePsp2CgcPreprocessOutput const *__cdecl
scePsp2CgcPreprocessProgram(
    const ScePsp2CgcCompileOptions *options,
    const ScePsp2CgcCallbackList *callbacks,
    void *userData,
    int32_t emitLineDirectives,
    int32_t emitComments
);
```

Arguments

options	Indicates the compile options for preprocessing a program. Also doubles as a session ID where multiple compiles are being run.
callbacks	Defines the callbacks to be used for file system access. If not provided, the default file system of the OS will be used (optional).
userData	An opaque pointer to user data that the compiler passes unaltered to each callback invoked.
emitLineDirectives	Indicates whether to emit line directives in the output.
emitComments	Indicates whether to retain comments in the output.

Return Values

A ScePsp2CgcPreprocessOutput object, containing the UTF-8 encoded text of the preprocessed program. Must be destroyed using scePsp2CgcDestroyPreprocessOutput. If 0 is returned the input arguments were malformed.

Description

Compiles a program for the PlayStation®Vita system using the options provided. Callbacks may be provided, but are optional. (see note on struct ScePsp2CgcCallbackList)

scePsp2CgcGetFirstParameter

Gets a handle to the first parameter of the shader.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE ScePsp2CgcParameter __cdecl
scePsp2CgcGetFirstParameter(
    const ScePsp2CgcCompileOutput *prog
);
```

Arguments

prog	The result of a successful compilation.

Return Values

A ScePsp2CgcParameter object pointing to the first global parameter of the shader.

Description

Returns a handle to the first global parameter of a shader.

Global parameters of a shader are not returned in any particular declaration order, so the first global parameter could be any parameter. It is guaranteed that the first global parameter for each shader will always be the same and the ordering will not change between compilations.

scePsp2CgcGetNextParameter

Gets a handle to the next parameter in list.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE ScePsp2CgcParameter __cdecl scePsp2CgcGetNextParameter(
    ScePsp2CgcParameter param
);
```

Arguments

param	Current parameter in list.

Return Values

A ScePsp2CgcParameter object pointing to the next parameter in list.

Description

Returns a handle to the next parameter in list.

scePsp2CgcGetParameterByName

Gets a handle to a parameter given its name.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE ScePsp2CgcParameter __cdecl
scePsp2CgcGetParameterByName(
    const ScePsp2CgcCompileOutput *prog,
    char const *name
);
```

Arguments

prog	The result of a successful compilation.
name	Parameter name

Return Values

A ScePsp2CgcParameter object pointing to the named parameter, or NULL if the parameter does not exist.

Description

Returns a handle to a named parameter.

scePsp2CgcGetParameterName

Returns the name of a parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE const char *__cdecl scePsp2CgcGetParameterName(
    ScePsp2CgcParameter param
);
```

Arguments

param The input parameter.

Return Values

A null-terminated string containing the name of the parameter.

Description

Returns a null-terminated string containing the name of the input parameter.

scePsp2CgcGetParameterSemantic

Returns the parameter semantic.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE const char *__cdecl scePsp2CgcGetParameterSemantic(
    ScePsp2CgcParameter param
);
```

Arguments

param The input parameter.

Return Values

A null-sterminated string containing the parameter semantic or NULL if no semantic was declared.

Description

Returns a null-terminated string containing the semantic of the input parameter or NULL if no semantic was declared.

scePsp2CgcGetParameterUserType

Returns the user-declared type that was used to declare the parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE const char *__cdecl scePsp2CgcGetParameterUserType(
    ScePsp2CgcParameter param
);
```

Arguments

param The input parameter.

Return Values

A null-terminated string containing the user-declared type of the parameter semantic or NULL if no semantic was declared.

Description

Returns a null-terminated string containing the user-declared type of the parameter semantic or NULL if no semantic was declared.

scePsp2CgcGetParameterClass

Returns the class of the input parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE ScePsp2CgcParameterClass __cdecl
scePsp2CgcGetParameterClass(
    ScePsp2CgcParameter param
);
```

Arguments

param The input parameter.

Return Values

The class for the input parameter.

Description

Returns a member of the ScePsp2CgcParameterClass enum set describing the class for the input parameter.

scePsp2CgcGetParameterVariability

Returns the variability of the input parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE ScePsp2CgcParameterVariability __cdecl
scePsp2CgcGetParameterVariability(
    ScePsp2CgcParameter param
);
```

Arguments

param The input parameter.

Return Values

The variability for the input parameter.

Description

Returns a member of the ScePsp2CgcParameterVariability enum set describing the variability for the input parameter.

scePsp2CgcGetParameterDirection

Returns the direction of the input parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE ScePsp2CgcParameterDirection __cdecl
scePsp2CgcGetParameterDirection(
    ScePsp2CgcParameter param
);
```

Arguments

param The input parameter.

Return Values

The direction for the input parameter.

Description

Returns a member of the ScePsp2CgcParameterDirection enum set describing the direction for the input parameter.

scePsp2CgcGetParameterBaseType

Returns the base type of the input parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE ScePsp2CgcParameterBaseType __cdecl
scePsp2CgcGetParameterBaseType(
    ScePsp2CgcParameter param
);
```

Arguments

param The input parameter.

Return Values

The base type for the input parameter.

Description

Returns a member of the ScePsp2CgcParameterBaseType enum set describing the base type for the input parameter.

scePsp2CgcIsParameterReferenced

Returns non-zero if the parameter is referenced in the shader.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE int32_t __cdecl scePsp2CgcIsParameterReferenced(
    ScePsp2CgcParameter param
);
```

Arguments

param The input parameter.

Return Values

Non-zero if the parameter is referenced, zero if it is never referenced.

Description

Returns a non-zero value if the parameter is referenced in the shader otherwise it returns a zero value.

scePsp2CgcGetParameterResourceIndex

Returns the hardware index associated with the shader parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE uint32_t __cdecl scePsp2CgcGetParameterResourceIndex(
    ScePsp2CgcParameter param
);
```

Arguments

param The input parameter.

Return Values

Zero-based index for the resource associated with the parameter.

Description

Returns a zero-based resource index for the input parameter.

This value is currently meaningful for uniform parameters only and it represents the offset from the start of the uniform buffer where the parameter is allocated.

scePsp2CgcGetParameterBufferIndex

Returns the buffer index where the uniform parameter is located.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE uint32_t __cdecl scePsp2CgcGetParameterBufferIndex(
    ScePsp2CgcParameter param
);
```

Arguments

param The input parameter.

Return Values

Index of the uniform buffer where the parameter is allocated.

Description

Returns the index for the uniform buffer where the parameter is allocated.

This value is meaningful for uniform parameters only.

scePsp2CgcGetFirstStructParameter

Returns a handle to the parameter representing the first member for a structure parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE ScePsp2CgcParameter __cdecl
scePsp2CgcGetFirstStructParameter(
    ScePsp2CgcParameter param
);
```

Arguments

param The input parameter.

Return Values

Handle to first member for the input parameter or NULL if *param* is not a structure parameter.

Description

Returns a handle to the parameter representing the first member for a struct parameter.

scePsp2CgcGetFirstUniformBlockParameter

Returns a handle to the parameter representing the first member of a uniform block parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE ScePsp2CgcParameter __cdecl
scePsp2CgcGetFirstUniformBlockParameter(
    ScePsp2CgcParameter param
);
```

Arguments

param The input parameter.

Return Values

Handle to first member for the input parameter or NULL if *param* is not a uniform block parameter.

Description

Returns a handle to the parameter representing the first member of a uniform block parameter.

scePsp2CgcGetArraySize

Returns the size of the array represented by the input parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE uint32_t __cdecl scePsp2CgcGetArraySize(
    ScePsp2CgcParameter param
);
```

Arguments

param The input parameter.

Return Values

The size of the array parameter.

Description

Returns the size of the array represented by the input parameter.

scePsp2CgcGetArrayParameter

Returns a parameter handle for the *i*-th element of the input array parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE ScePsp2CgcParameter __cdecl
scePsp2CgcGetArrayParameter(
    ScePsp2CgcParameter param,
    uint32_t index
);
```

Arguments

param	The input parameter.
index	The index of the array element.

Return Values

A parameter handle or NULL if the parameter does not exist.

Description

Returns a parameter handle for the *i*-th element of the input array parameter or NULL if the parameter does not exists.

scePsp2CgcGetParameterVectorWidth

Returns the vector width of a vector parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE uint32_t __cdecl scePsp2CgcGetParameterVectorWidth(
    ScePsp2CgcParameter param
);
```

Arguments

param	The input parameter.

Return Values

The vector width of the input parameter or zero if the parameter is not a vector parameter.

Description

Returns the vector width of the input parameter or zero if the parameter is not a vector parameter.

scePsp2CgcGetParameterColumns

Returns the number of columns of a matrix parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE uint32_t __cdecl scePsp2CgcGetParameterColumns(
    ScePsp2CgcParameter param
);
```

Arguments

param	The input parameter.

Return Values

The number of columns of the input parameter or zero if the parameter is not a matrix parameter.

Description

Returns the number of columns of the input parameter or zero if the parameter is not a matrix parameter.

scePsp2CgcGetParameterRows

Returns the number of rows of a matrix parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE uint32_t __cdecl scePsp2CgcGetParameterRows(
    ScePsp2CgcParameter param
);
```

Arguments

param	The input parameter.

Return Values

The number of rows of the input parameter or zero if the parameter is not a matrix parameter.

Description

Returns the number of rows of the input parameter or zero if the parameter is not a matrix parameter.

scePsp2CgcGetParameterMemoryLayout

Returns the memory layout of the input matrix parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE ScePsp2CgcParameterMemoryLayout __cdecl
scePsp2CgcGetParameterMemoryLayout(
    ScePsp2CgcParameter param
);
```

Arguments

param	The input parameter.

Return Values

The memory layout for the input parameter.

Description

Returns a member of the ScePsp2CgcParameterMemoryLayout enum set describing the layout for the input matrix parameter.

scePsp2CgcGetRowParameter

Returns a parameter handle for the *i*-th row of the input matrix parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE ScePsp2CgcParameter __cdecl scePsp2CgcGetRowParameter(
    ScePsp2CgcParameter param,
    uint32_t index
);
```

Arguments

param	The input parameter.
index	The index of the row.

Return Values

A parameter handle or NULL if the parameter does not exist.

Description

Returns a parameter handle for the *i*-th row of the input matrix parameter or NULL if the parameter does not exists.

scePsp2CgcGetSamplerQueryFormatWidth

Returns the query format component count for a sampler parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE uint32_t __cdecl scePsp2CgcGetSamplerQueryFormatWidth(
    ScePsp2CgcParameter param
);
```

Arguments

param The input sampler parameter.

Return Values

The query format component count for the sampler parameter.

Description

Returns the query format component count for a sampler parameter. For example it returns 4 for an `half4` or `char4` query format, or 2 for an `half2`, or 1 for a `float` query format.

scePsp2CgcGetSamplerQueryFormatPrecisionCount

Returns the number of query precision format used for a specified sampler parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE uint32_t __cdecl
scePsp2CgcGetSamplerQueryFormatPrecisionCount(
    ScePsp2CgcParameter param
);
```

Arguments

param The input sampler parameter.

Return Values

Returns the number of query precision format used for a specified sampler parameter.

Description

Returns the number of query precision format used for a specified sampler parameter. For example it returns 1 if the sampler is always queried at `half` precision, or 2 if the sampler is queried at `half` precision and also at `char` precision.

scePsp2CgcGetSamplerQueryFormatPrecision

Returns the query format precision used for a specified sampler parameter.

Definition

```
#include <psp2cgc.h>
SCE_PSP2CGC_INTERFACE ScePsp2CgcParameterBaseType __cdecl
scePsp2CgcGetSamplerQueryFormatPrecision(
    ScePsp2CgcParameter param,
    uint32_t index
);
```

Arguments

param	The input sampler parameter.
index	The index of the query format

Return Values

Returns the query format precision used for a specified sampler parameter.

Description

Returns the query format precision used for a specified sampler parameter, since a sampler can be queried at multiple precision the index parameter tells which precision to return. The number of available precision can be queried by calling scePsp2CgcGetSamplerQueryFormatPrecisionCount().

Callback Functions

ScePsp2CgcCallbackAbsolutePath

A callback used to retrieve the absolute path name for a given file.

Definition

```
#include <psp2cgc.h>
typedef const char * (*ScePsp2CgcCallbackAbsolutePath)(
    const char *fileName,
    const ScePsp2CgcSourceLocation *includedFrom,
    const ScePsp2CgcCompileOptions *compileOptions,
    void *userData
);
```

Arguments

fileName	The (possibly relative) file path for an include file, as provided by ScePsp2CgcCallbackLocateFile.
includedFrom	The location of the include directive.
compileOptions	The original options pointer used to invoke this compile.
userData	Opaque pointer to user data.

Return Values

The absolute file path or 0 if the file could not be located.

Description

Files are uniquely identified by absolute paths. If two include files lead to the same absolute path, the previously found file is used and no call to ScePsp2CgcCallbackOpenFile will be made. This function allows for a translation from a relative path scheme to an absolute path scheme.

If there is no valid absolute path for the given file, 0 should be returned. If a non-zero string is returned, the caller takes ownership and will release the allocation via the ScePsp2CgcCallbackReleaseFileName callback. This string will be the name passed to ScePsp2CgcCallbackOpenFile.

ScePsp2CgcCallbackFileDate

Provides date information for the named file.

Definition

```
#include <psp2cgc.h>
typedef int32_t (*ScePsp2CgcCallbackFileDate)(
    const ScePsp2CgSourceFile *file,
    const ScePsp2CgcSourceLocation *includedFrom,
    const ScePsp2CgcCompileOptions *compileOptions,
    void *userData,
    int64_t *timeLastStatusChange,
    int64_t *timeLastModified
);
```

Arguments

file	A file object returned from ScePsp2CgcCallbackOpenFile.
includedFrom	The location of the include directive.
compileOptions	The original options pointer used to invoke this compile.
userData	Opaque pointer to user data.
timeLastStatusChange	A pointer to the time of last status change (i.e. *creationTime*).
timeLastModified	A pointer to the time of last modification.

Return Values

Non-zero for success, 0 on failure.

Description

If the date attributes could not be read, 0 is returned and the results will be considered invalid. On success, *timeLastStatusChange* and *timeLastModified* will have been updated and a non-zero value is returned. The time values are encoded as `time_t`.

ScePsp2CgcCallbackLocateFile

A callback used to search for a named file.

Definition

```
#include <psp2cgc.h>
typedef const char * (*ScePsp2CgcCallbackLocateFile)(
    const char *fileName,
    const ScePsp2CgcSourceLocation *includedFrom,
    uint32_t searchPathCount,
    const char * const *searchPaths,
    const ScePsp2CgcCompileOptions *compileOptions,
    void *userData,
    const char **errorString
);
```

Arguments

fileName	The name of the file to be located.
includedFrom	The location of the include directive. Set to 0 for primary files.
searchPathCount	The number of search paths provided via *searchPaths*.
searchPaths	The array of search paths. The size is provided via *searchPathCount*.
compileOptions	The original options pointer used to invoke this compile.
userData	Opaque pointer to user data.
errorString	If 0 is returned and if this pointer is non-zero, it should be updated to contain a diagnostic message.

Return Values

A relative or absolute path to the file or 0 if the file could not be located.

Description

This function will search in all provided paths for the named file. If the file could not be located, 0 is returned and *errorString* will have been updated to a representative message, explaining why the file could not be located. On success, a non-zero string is returned. The caller takes ownership and will release the allocation via the ScePsp2CgcCallbackReleaseFileName callback.

ScePsp2CgcCallbackOpenFile

A callback used when the compiler needs to open a file.

Definition

```
#include <psp2cgc.h>
typedef ScePsp2CgSourceFile * (*ScePsp2CgcCallbackOpenFile)(
    const char *fileName,
    const ScePsp2CgcSourceLocation *includedFrom,
    const ScePsp2CgcCompileOptions *compileOptions,
    void *userData,
    const char **errorString
);
```

Arguments

fileName	The absolute path of the file to be opened.
includedFrom	The include location. Set to 0 for a primary file.
compileOptions	The original options pointer used to invoke this compile.
userData	Opaque pointer to user data.
errorString	If 0 is returned and this pointer is non-zero, this output param should be updated to contain a diagnostic message.

Return Values

A ScePsp2CgSourceFile object to be destroyed later with ScePsp2CgcCallbackReleaseFile.

Description

The *includedFrom* location will either be 0, if a primary file is being opened, or it will point to the location of the include directive. If the file could not be opened, 0 is returned and *errorString* will be updated to point to a representative message, explaining why the file could not be opened. On success, a non-zero pointer is returned. The caller takes ownership, calling ScePsp2CgcCallbackReleaseFile when the returned ScePsp2CgSourceFile is no longer required.

ScePsp2CgcCallbackReleaseFile

A callback used when the compiler needs to release file data.

Definition

```
#include <psp2cgc.h>
typedef void (*ScePsp2CgcCallbackReleaseFile)(
    const ScePsp2CgSourceFile *file,
    const ScePsp2CgcCompileOptions *compileOptions,
    void *userData
);
```

Arguments

file	The ScePsp2CgSourceFile object to be destroyed.
compileOptions	The original options pointer used to invoke this compile.
userData	Opaque pointer to user data.

Return Values

None

Description

This function is used to release memory that was previously allocated as part of a call to ScePsp2CgcCallbackOpenFile. This will be called once per *fileName* associated with that ScePsp2CgSourceFile object when the resources for the compile session are released.

ScePsp2CgcCallbackReleaseFileName

A callback for the compiler to release a file name.

Definition

```
#include <psp2cgc.h>
typedef void (*ScePsp2CgcCallbackReleaseFileName)(
    const char *fileName,
    const ScePsp2CgcCompileOptions *compileOptions,
    void *userData
);
```

Arguments

fileName	The file name string to be destroyed.
compileOptions	The original options pointer used to invoke this compile.
userData	Opaque pointer to user data.

Return Values

None

Description

This function is used to release memory that was previously allocated as part of a call to ScePsp2CgcCallbackLocateFile or ScePsp2CgcCallbackAbsolutePath.