Stream Reference Manual

This manual gives detailed reference information for the Stream programmer. It describes the tools in the Stream toolset, the Stream language, the Stream programming model and its Component, Pipeline and Kernel APIs, and the DSP MIPS standard library.

SWRM-00001-007

Stream Processors, Inc.
455 DeGuigne Drive
Sunnyvale, CA 94085-3890 USA
Telephone: +1.408.616.3338
Fax: +1.408.616.3337
Email: info@streamprocessors.com
Web: www.streamprocessors.com

Table of Contents

1 Introduction

This manual contains detailed reference information about the Stream tools from Stream Processors, Inc. (SPI). It describes details of Stream tools, the Stream language, the Stream application program interface (API), the Stream programming model (SPM), and the DSP MIPS standard library. A companion volume, *Stream User's Guide*, contains introductory tutorial information about program development using the Stream tools, the Stream API, and the Stream programming model.

1.1 Typographical conventions

This manual indicates a definition by setting the defined word in italic type. Italic type is also used for emphasis and to indicate a placeholder which may take on different values; for example, an *n*-bit object might contain 8, 16 or 32 bits. Boldface type indicates literals in the C language or literal filenames; for example, **int** is a C keyword. Monospace typeface `Courier` is used for Stream program examples.

1.2 Document revision history

Document number	Date	Release version	Description
SWRM-00001-001	December 2007	RapiDev 1.0	Initial release
SWRM-00001-002	January 2008	RapiDev 1.0.1	Revision
SWRM-00001-003	April 2008	RapiDev 1.0.2	Revision
SWRM-00001-004	June 2008	Stream 2.0	Major revision
SWRM-00001-005	September 2008	Stream 2.1	Revision
SWRM-00001-006	December 2008	Stream 2.2	Revision
SWRM-00001-007	March 2009	Stream 2.3	Revision

2 Summary tables

This chapter gives summary tables for Stream tools, API types and API functions. Each entry contains a link to a page in this manual with a detailed description. *Stream User's Guide* gives more general introductory information about Stream tools and about the Stream programming model.

2.1 Tools summary

spc	Compiler
spide	Integrated development environment (debugger)
spperf	Performance analysis tool
sprun	Run a DSP MIPS executable
spsim	Simulate a DSP MIPS executable
Special SPM options	Special Stream programming model runtime options

2.2 Component API types summary

spi_binfo_valid_t	Buffer information
spi_buffer_flag_t	Buffer flag
spi_buffer_flags_t	Buffer flags
spi_buffer_t	Buffer
spi_cmd_t	Command
spi_component_flag_t	Component flag
spi_component_flags_t	Component flags
spi_component_instance_cmdhandler_fn_t	Instance command handler function
spi_component_instance_destroy_fn_t	Instance destroy function
spi_component_instance_execute_fn_t	Instance execute function
spi_component_instance_init_fn_t	Instance initialization function
spi_component_properties_fn_t	Component properties function
spi_component_t	Component
spi_connection_flag_t	Connection flag
spi_connection_flags_t	Connection flags
spi_connection_t	Connection
spi_execution_requirement_t	Execution requirement
spi_fb_pixel_type_t	Framebuffer pixel
spi_fb_pixel_types_t	Framebuffer pixels
spi_image_flag_t	Image flag
spi_image_flags_t	Image flags
spi_instance_context_t	Instance context
spi_instance_state_t	Instance state
spi_instance_t	Instance
spi_payload_timer_t	Payload timer
spi_payload_type_t	Payload type
spi_pel_t	Processing element
spi_pels_t	Processing elements
spi_pool_flag_t	Buffer pool flag
spi_pool_flags_t	Buffer pool flags
spi_portdir_t	Port direction
spi_provider_t	Component provider
spi_resource_t	Processing resource
spi_resources_t	Processing resources
spi_response_context_t	Command response context
spi_response_errno_t	Command response error
spi_response_handler_fn_t	Command response handler function
spi_response_t	Command response
spi_schedgroup_properties_t	Scheduling group properties
spi_spm_flag_t	Stream programming model flag
spi_spm_flags_t	Stream programming model flags
spi_timer_t	Timer

2.3 Component API functions summary

The Component API functions section of this manual lists all Component API macros and functions alphbetically. The summary table below organizes them by categories:

- *Definition macros* define components and scheduling groups. These macros may be used only as declarations with file scope at the top level of a source file, not within functions.
- *Global functions* apply to global Stream programming model properties.
- *Instance-specific functions* apply to the current component instance.
- The remaining categories correspond to Stream programming model concepts: buffers, commands, and so on.

Definition:

SPI_COMPONENT_NEW	Define a component
SPI_SCHEDGROUP_NEW	Define a scheduling group

Global:

spi_connect	Connect two component instances
spi_get_buffer_heap_highwater	Get buffer heap highwater mark
spi_get_buffer_heap_size	Get current buffer heap size
spi_get_connection	Get the connection with a given name
spi_get_instance	Get the instance with a given name
spi_get_log	Get the log with a given name
spi_get_pool	Get the pool with a given name
spi_get_time	Get the system time
spi_get_timer	Get the timer with a given name
spi_load_image	Load an image on a processing element
spi_spm_start	Start the Stream programming model runtime
spi_spm_stop	Stop the Stream programming model runtime

Instance-specific:

spi_get_component	Get the component for the current instance
spi_get_name	Get the name of the current instance
spi_get_priority	Get the priority of the current instance
spi_get_state	Get the state of the current instance
spi_instance_new	Create a new component instance
spi_set_priority	Set scheduling priority of current instance
spi_set_state	Set the state of the current instance

Buffer:

spi_buffer_clone	Clone a buffer
spi_buffer_close	Close a buffer
spi_buffer_free	Free a buffer
spi_buffer_get_info	Get buffer information
spi_buffer_get_info_size	Get size of buffer information
spi_buffer_get_size	Get buffer size
spi_buffer_merge	Merge buffers
spi_buffer_new	Create a new buffer
spi_buffer_open	Open a buffer
spi_buffer_set_info	Set buffer information

Command:

spi_cmd_free	Free a command
spi_cmd_get_desc	Get a command description
spi_cmd_get_id	Get a command ID
spi_cmd_get_name	Get a command name
spi_cmd_get_payload	Get a command payload
spi_cmd_get_payload_size	Get the size of a command payload
spi_cmd_get_payload_type	Get the type of a command payload
spi_cmd_get_response_payload_type	Get the type of a command response payload
spi_cmd_register	Register a command
spi_cmd_send	Send a command
spi_cmd_send_response	Send a command response

Component:

spi_component_find	Find a component
spi_component_get_desc	Get a component description
spi_component_get_name	Get a component name
spi_component_get_provider	Get the provider of a component
spi_component_get_version	Get the version number of a component
spi_component_set_flags	Set component flags
spi_component_set_resource_requirements	Set component resource requirements

Connection:

spi_connection_get_depth	Get the depth of a connection
spi_connection_get_name	Get the name of a connection
spi_connection_is_empty	Check if connection is empty
spi_connection_is_full	Check if connection is full
spi_connection_new	Create a new connection
spi_connection_pop	Pop a buffer from a connection
spi_connection_push	Push a buffer to a connection

Execution requirement:

spi_exec_req_activate	Activate an execution requirement
spi_exec_req_delete	Delete an execution requirement
spi_exec_req_is_satisfied	True if execution requirement is satisfied
spi_exec_req_register	Register an execution requirement

Framebuffer:

spi_fb_get_line_length	Get framebuffer line length
spi_fb_get_pixel_type	Get framebuffer pixel type
spi_fb_get_xres	Get framebuffer horizontal resolution
spi_fb_get_yres	Get framebuffer vertical resolution
spi_fb_is_available	Check if framebuffer is available
spi_fb_pool_new	Create a new framebuffer buffer pool

Log:

spi_log	Write a log message
spi_log_get_desc	Get a log description
spi_log_get_enable_mask	Get the current enable mask of a log
spi_log_get_name	Get a log name
spi_log_new	Create a log
spi_log_set_enable_mask	Set the enable mask of a log

Pool:

spi_pool_free	Free a buffer pool
spi_pool_get_avail_buffer_count	Get the number of buffers acailable from a pool
spi_pool_get_buffer	Get a buffer from a pool
spi_pool_get_desc	Get a pool description
spi_pool_get_name	Get a pool name
spi_pool_new	Create a new buffer pool

Port:

spi_port_export	Export a port on a contained instance
spi_port_get_connection	Get a connection on a port
spi_port_get_connection_count	Get the current connection count on a port
spi_port_get_desc	Get a port description
spi_port_get_dir	Get the direction of a port
spi_port_get_max_connection_count	Get the maximum connection count of a port
spi_port_get_name	Get a port name
spi_port_register	Register a port

Provider:

spi_provider_get_name	Get a provider name

Response:

spi_response_free	Free a command response
spi_response_get_errno	Get a command response error number
spi_response_get_payload	Get a command response payload
spi_response_get_payload_size	Get the size of a command response payload
spi_response_get_payload_type	Get the type of a command response payload
spi_response_set_handler	Set a command response handler
spi_response_strerror	Get a textual description of an error code

Scheduling group:

spi_schedgroup_component_find	Find a component in a scheduling group
spi_schedgroup_register_component	Register a scheduling group component
spi_schedgroup_set_controlled_resources	Set the resources controlled by a scheduling group
spi_schedgroup_set_min_stacksize	Set the minimum stacksize for a scheduling group
spi_schedgroup_set_processing_elements	Set the processing elements for a scheduling group

Timer:

spi_timer_get_desc	Get a timer description
spi_timer_get_name	Get a timer name
spi_timer_get_nanoseconds	Get timer count in nanosecond ticks
spi_timer_get_start_count	Get timer start count
spi_timer_get_total_nanoseconds	Get total timer count in nanosecond ticks
spi_timer_new	Create a timer
spi_timer_start	Start a timer
spi_timer_stop	Stop a timer

Trace:

spi_trace_is_enabled	Check if tracing is enabled
spi_trace_start	Start tracing
spi_trace_stop	Stop tracing

2.4 Pipeline API functions summary

spi_barrier	Wait for completion of all DPU activity	
spi_count	Return number of valid data records in a stream	
spi_load_2d_index	Load 2d blocks using index stream	[Storm-2 only]
spi_load_2d_index_crop	Load cropped 2d blocks using index stream	[Storm-2 only]
spi_load_2d_stride	Load 2d blocks using stided access pattern	[Storm-2 only]
spi_load_2d_stride_crop	Load cropped 2d blocks using strided access pattern	[Storm-2 only]
spi_load_block	Load a block of contiguous data	
spi_load_index	Load data using index stream	
spi_load_stride	Load data using strided access pattern	
spi_out	Return value of kernel scalar output parameter	
spi_print_stream_data	Print LRF data for a stream	
spi_store_2d_index	Store 2d blocks using index stream	[Storm-2 only]
spi_store_2d_index_crop	Store cropped 2d blocks using index stream	[Storm-2 only]
spi_store_2d_stride	Store 2d blocks using strided access pattern	[Storm-2 only]
spi_store_2d_stride_crop	Store cropped 2d blocks using strided access pattern	[Storm-2 only]
spi_store_block	Store a block of contiguous data	
spi_store_index	Store data using index stream	
spi_store_stride	Store data using strided access pattern	

2.5 Kernel API library functions summary

spi_array_read	Read data from array stream
spi_array_write	Write data to array stream
spi_cond_read	Read data from conditional stream
spi_cond_write	Write data to conditional stream
spi_eos	Check for end of stream
spi_printf	Print data (functional mode only)
spi_read	Read data from sequential stream
spi_write	Write data to sequential stream

2.6 Kernel API intrinsic operations summary

spi_add	Scalar addition	
spi_and	Scalar bitwise AND	
spi_eq	Scalar equality	
spi_laneid	Lane ID	
spi_le	Scalar less than or equal to	
spi_lt	Scalar less than	
spi_ne	Scalar not equal to	
spi_not	Scalar bitwise NOT	
spi_or	Scalar bitwise OR	
spi_perm	Scalar permute data	
spi_select	Scalar select	
spi_shift	Scalar logical shift	
spi_shifta	Scalar arithmetic shift	
spi_shuffle	Scalar shuffle bytes	
spi_shuffled	Scalar shuffle bytes dual	
spi_sub	Scalar subtraction	
spi_vabd	Vector absolute difference	
spi_vadd	Vector addition	
spi_vaddc	Vector addition with carry	
spi_vadds	Vector saturating addition	
spi_vand	Vector bitwise AND	
spi_vclip	Vector clip	
spi_vdivstep	Vector divide step	
spi_vdotna	Vector accumulating negative dot product	
spi_vdotp	Vector dot product	[Storm-2 only]
spi_vdotpa	Vector accumulating positive dot product	
spi_veq	Vector equal to	
spi_vffone	Vector find first ‘1’	
spi_vle	Vector less than or equal to	
spi_vlsbs	Vector least significant bits	
spi_vlt	Vector less than	
spi_vmax	Vector maximum	
spi_vmin	Vector minimum	
spi_vmula	Vector multiplication accumulating	
spi_vmuld	Vector multiplication dual output	
spi_vmulha32	Vector accumulating multiplication high	
spi_vmulha16	Vector accumulating multiplication high	
spi_vmulla32	Vector accumulating multiplication low	
spi_vmulra	Vector accumulating multiplication rounding	
spi_vne	Vector not equal to	
spi_vnorm	Vector normalize	
spi_vnot	Vector bitwise NOT	
spi_vor	Vector bitwise OR	
spi_vperm	Vector permute data	
spi_vrandl	Vector reduction AND across lanes (scalar result)	
spi_vrandlv	Vector reduction AND across lanes (vector result)	
spi_vrorl	Vector reduction OR across lanes (scalar result)	
spi_vrorlv	Vector reduction OR across lanes (vector result)	
spi_vsad	Vector sum of absolute differences	[Storm-2 only]

spi_vsclip	Vector shift and clip	[Storm-2 only]
spi_vselect	Vector select	
spi_vselectd	Vector select	
spi_vshift	Vector logical shift	
spi_vshifta	Vector arithmetic shift	
spi_vshuffle	Vector shuffle bytes	
spi_vshuffled	Vector shuffle bytes dual	
spi_vsub	Vector subtraction	
spi_vsubc	Vector subtraction with carry	
spi_vsubs	Vector saturating subtraction	
spi_vsuma	Vector accumulating sum	
spi_vxor	Vector bitwise XOR	
spi_xor	Scalar logical XOR	

3 Tools

The Stream toolset includes program development tools to compile, simulate, and run application programs. This chapter describes the available Stream tools.

3.1 spc

3.1.1 Usage

spc [*option* ...] *file* ...

3.1.2 Options

-c	Compile source files, do not link.
-D *name*[=*text*]	Define symbol *name* to preprocessor.
--dump_lrf	Dump LRF allocation data.
-E	Expand: stop after preprocessing.
-g	Generate debug information for device mode; functional mode **-z** always generates debug information.
-h	Display usage message.
-I *path*	Add *path* to include search path.
--include *file*	Force include of *file.*
-l *name*	Library: add **lib***name***.a** to link (see caveat below).
-L *path*	Add *path* to library search path (see caveat below).
--lint	Stop after checking syntax and semantics.
-m *device*	Machine: generate code for *device* (**sp16**, **sp8, sp2x**, **sp16_sys**, **sp8_sys**, **sp2x_sys**) [default: **sp16**].
-m testbench	Compile standalone DSP MIPS executable with **spi_main** mainline (functional, **sp16**, **sp8**, or **sp2x**).
--map *file*	Generate mapfile in *file* (for device executables only).
--nodefaultlibs	Do not link default libraries.
-O*n*	Pass optimization level *n* (0, 1, 2, or 3) to **gcc** or **mips-gcc.**
-o *file*	Generate output to *file* [default: **a.out** for executables, *file***.o** for unlinked objects].
-p	Profile: generate profile information.
--save-temps	Save all intermediate files.
--tmp *dir*	Use given *dir* for temporary files during compilation.
-U *name*	Undefine symbol *name* to preprocessor.

-v	Verbose: generate verbose information output.
--version	Version: print version information.
-w	Inhibit all warnings [default].
-W*flags*	Pass warning *flags* to **gcc**; **-Wall** enables all warnings.
-X *phase arg*	Pass *arg* to given compilation *phase.*
-z	Build functional mode executable.

3.1.3 Description

The Stream Processors compiler **spc** compiles and links a Stream program. The user can build an executable program in a single step (compiling plus linking) with **spc** as follows:

```
$ spc -o prog.device.out -m sp16 prog.sc
```

spc supplies option-specific preprocessor definitions that programs can use for conditionalization, as indicated in the table in the Predefined macros section of the Stream language chapter below.

spc can also compile and link in separate steps:

```
$ spc -c -m sp16 prog.sc
$ spc -o prog.device.out -m sp16 prog.o
```

The user can the standard librarian **ar** to create an object archive that contains one or more object files compiled from Stream sources. When the user uses **spc** to link object files into an executable, the command line should use the appropriate compilation options (**-g**, **-m**, **-p**), as **spc** may perform additional compilation steps when creating the executable.

If an object archive contains one or more SPM components, the **spc** command line to build an executable using the archive must explicitly specify linker directives that force the executable to contain the desired components. For example, suppose archive **lib_sys.a** contains object files for three components, with component ids **C1_COMP_ID**, **C2_COMP_ID**, and **C3_COMP_ID** used in the **SPI_COMPONENT_NEW** macros that define the components. To build a System MIPS executable that can use components 1 and 3, the **spc** command line must supply two **-Xld** options that force the linker to pull in the code for the desired components from the archive:

```
$ spc -o app.out -m sp16_sys -Xld -uC1_COMP_ID -Xld -uC3_COMP_ID app.c lib_sys.a
```

Without the special linker options, the linker would not pull in the code for the desired components from the archive, so the resulting executable **app.out** would not contain the components.

3.1.4 See also

Stream User's Guide gives detailed examples of the use of **spc** to build various flavors of programs (functional mode, testbench, debug mode, profile mode, complete applications, and so on).

3.2 spide

3.2.1 Usage

spide [*option* ...]

3.2.2 Description

The Stream Processors integrated development environment (IDE) **spide** provides a graphical user interface (GUI) for the development of Stream programs. The programmer can use **spide** to edit, build, run, and debug a Stream program.

3.2.3 Options

By default, **spide** uses a maximum of 128M of memory. Option **-vmargs -Xmx***msize* sets the memory limit to the given *msize* instead. For example, to set the maximum memory size to 256M, invoke **spide** as follows:

```
$ spide -vmargs -Xmx256M &
```

By default, **spide** uses a workspace selected by an interactive dialog when it begins. To use the workspace in *workspacedir* instead, invoke **spide** as follows:

```
$ spide -data workspacedir &
```

3.2.4 See also

Stream User's Guide gives a detailed introduction to the use of **spide**.

3.3 spperf

3.3.1 Usage

spperf *name*.**out** *name*.**sbt** *name*_**tcs**.**sbt -o** *name*.**html** [**--stm** *name*.**stm**] [**--time_unit** *time_unit*]

3.3.2 Options

-o *name*.**html**	Generate HTML file *name*.**html**.
--stm *name*.**stm**	Generate XML file *name*.**stm** for **spide**.
-t *time_unit* **--time_unit** *time_unit*	Use the given *time_unit*: **s**, **ms**, **us**, **ns**, or **ps** (default: **us**).

3.3.3 Description

spperf provides program performance information as an HTML file. It generates an HTML file from binary trace information produced by the simulator **spsim**.

To use **spperf**, first make sure the Stream program enables tracing by calling **spi_trace_start** and **spi _trace_stop**.

Next, compile the program using **spc** option **-p**. Compiling with the **-p** option generates an executable that contains analysis information about kernels and stream operations in the program. The executable generates trace information in .**sbt** files at runtime.

Next, simulate the program with **spsim**, supplying option **--spi_trace_file=***name***.sbt** after the image file. This generates two binary files *name***.sbt** and *name***_tcs.sbt** that contain program trace information.

Next, run **spperf**. **spperf** takes as input the executable file produced by **spc** and the two .**sbt** trace files produced during simulation by **spsim**. It produces as output an **.html** file that contains performance information. Optionally, it also produces an .**stm** XML file that may be used by the Stream IDE.

Finally, use a browser to view the performance information in the generated HTML file.

3.3.4 See also

spc, **spsim**

Stream User's Guide gives detailed examples of the use of **spperf**.

3.4 sprun

3.4.1 Usage

sprun [*option* ...] *DSP_image* [*SPM_option* ...] [*argument* ...]

3.4.2 Options

-d *size* Set minimum DSP MIPS memory size to *size* (advanced option).

-e Redirect DSP MIPS **stderr** to **/dev/dsp_mips_stderr;** DSP MIPS **stdout** still goes to **/dev/dsp_mips_stdout**.

-h Help: print **sprun** help information.

-l Do not reset all cores (advanced option).

-n Do not redirect DSP MIPS console; do not wait for application termination.

-q Quiet: do no print SPI monitor banner.

-r Redirect DSP MIPS console to **/dev/dsp_mips_console**; redirects both **stdout** and **stderr** unless **-e**.

-s Stop all cores (advanced option; takes no additional arguments).

3.4.3 Description

System MIPS Linux command **sprun** runs a DSP MIPS program *DSP_image* compiled for a Stream Processors hardware device. Each *option* is an option to **sprun** (not to the program), *DSP_image* is the DSP MIPS executable program image to run, and the trailing *argument* list is passed to the DSP MIPS program.

The *Special SPM options* section below describes special Stream programming model options in the argument list that are recognized by **spi_spm_start**.

3.4.4 Caveats

If **sprun** reports an error such as “failed to open...” or “failed to allocate...”, reset DSP MIPS with **sprun -s**, then try again.

By default, the Stream runtime allocates 1MB for DSP MIPS memory. If *DSP_image* requires more than 1MB of memory, use **sprun** option **-d** to allocate additional DSP MIPS memory; do not use special *SPM_option* **--spi_dsp_memory_size** instead.

sprun only executes programs compiled for DSP MIPS on a Stream Processors device. Compilation in functional mode with **-z** creates an executable which runs directly on the host PC, not under **sprun**.

3.4.5 See also

Special SPM options, **spi_spm_start**, **spsim**

Stream User's Guide gives detailed examples of the use of **sprun**.

3.5 spsim

3.5.1 Usage

spsim [*option* ...] *image* [*SPM_option* ...] [*argument* ...]

3.5.2 Options

Option	Meaning	Default
--architecture *arch*	Use architecture *arch*.	SP16
--dpu_clock *n*	Set DPU clock to *n* MHz.	500 MHz
--dsp_mips_clock *n*	Set DSP MIPS clock to *n* MHz.	280 MHz
--dsp_mips_cycle_acc *n*	Enable/disable cycle-accurate DSP MIPS simulation.	Enabled
--help [or **-h**]	Print help message.	
--logging_level *n*	Set logging level to *n*.	2
--mem_clock *n*	Set memory clock to *n* MHz.	233 MHz
--mem_model *model*	Use given memory *model* (**simple** or **arbitrated**).	**arbitrated**
--show_tick	Display time periodically.	
--uart_port *n*	Use port *n* for UART1 (debug monitor).	10410
--version [or **-v**]	Print version information.	

3.5.3 Description

spsim simulates a Stream program compiled for a Stream Processors device. **spsim** usage is similar to the usage of **sprun**. Each *option* is an option to **spsim** (*not* to the program), *image* is the executable program image to simulate, and each *argument* is a program argument passed to the program *image*.

By default, the Stream runtime allocates 1MB for DSP MIPS memory. If *image* loads a DSP MIPS program that requires more than 1MB of memory, use *SPM_option* **--spi_dsp_memory_size** or **--spi_dsp_memory_size_image** to allocate additional DSP MIPS memory.

Performance data from the simulator (for example, timer data) is very accurate for DSP MIPS performance and for long kernels, and is quite accurate for other operations (for example, stream operations). However, it may differ significantly from hardware performance data in extreme cases.

The *Special SPM options* section below describes special Stream programming model options in the argument list that are recognized by **spi_spm_start**.

3.5.4 See also

Special SPM options, **spi_spm_start**, **sprun**

Stream User's Guide gives detailed examples of the use of **spsim**.

3.6 Special SPM options

3.6.1 Usage

sprun [*option* ...] *image* [*SPM_option* ...] [*argument* ...]
spsim [*option* ...] *image* [*SPM_option* ...] [*argument* ...]

3.6.2 Options

--spi_default_stack_size=*size*	Set the stack size for scheduling groups to the given *size* (in bytes); the default is 64K bytes. A scheduling group may specify an explicit minimum stack size with **spi_schedgroup_set_min_stack_size**.
--spi_dsp_memory_size=*size*	Set the amount of memory reserved for the DSP MIPS image. Default value: **1M**.
--spi_dsp_memory_size_image=*image*	Use *image* to set the amount of memory reserved for DSP MIPS.
--spi_dsp_stderr_file=*file*	Redirect DSP MIPS standard error output to *file*. By default, DSP MIPS **stderr** output goes to System MIPS **stderr**.
--spi_dsp_stdout_file=*file*	Redirect DSP MIPS standard output to *file*. By default, DSP MIPS **stdout** output goes to System MIPS **stdout**.
--spi_init_file=*file*	Create instances and connections as specified in initialization file *file*. Multiple initialization files may be specified with repeated options.
--spi_log_dir=*dir*	Keep separate log files in directory *dir*. By default, the runtime writes all logs intermixed to **stdout**.
--spi_log_mask=*log,mask*	Use the given enable *mask* for the given *log*. The *log* may be predefined log **debug** or **error**, or a user-defined log; ***** means all logs. By default, SPM enables all **error** log levels and disables all **debug** log levels.
--spi_log_timestamps=[**0**|**1**]	Disable or enable timestamps on log entries (default: enabled).
--spi_shared_memory_size=*size*	Set the amount of shared memory reserved for **sp_buffer_t** objects. Default value **0** indicates that all memory not required by System MIPS or DSP MIPS should be shared.
--spi_tcs_logging_level=*n*	Set the simulator logging level to *n*.
--spi_trace_buffer_size=*n*	Set the trace buffer size to *n* bytes (default: 32768).
--spi_trace_file=*file*	Write stream command trace results to *file*.

3.6.3 Description

The Stream programming model recognizes some special **--spi_*** options to **spi_spm_start**. A program calls **spi_spm_start** to start the Stream programming model runtime:


```
int main(int argc, char *argv[]) {
        ...
        spi_spm_start("progname", &argc, argv, SPI_SPM_FLAG_NONE);
        /* handle argc/argv here, after special options have been removed */
        ...
```

spi_spm_start recognizes the special options listed above and removes them from the supplied **argc**/**argv** argument list. These options control Stream programming model behavior. *size* can include trailing **K**, **M**, or **G** (uppercase or lowercase) to indicate kilobytes, megabytes or gigabytes.

spi_load_image similarly recognizes special arguments and removes them from its argument list.

By default, the Stream runtime allocates 1MB for DSP MIPS memory. If *image* loads a DSP MIPS program that requires more than 1MB of memory, use *SPM_option* --**spi_dsp_memory_size** or --**spi_dsp_memory_size_image** in the **spi_spm_start** or **spi_load_image** argument list to allocate additional DSP MIPS memory.

3.6.4 See also

spi_load_image, **spi_spm_start**, **sprun**, **spsim**

Stream User's Guide gives detailed examples of the use of special SPM options.

4 Stream language extensions

This chapter describes added Stream keywords, macros defined by **spc**, standard C types, and DPU types.

4.1 Added keywords

The Stream language uses the following keywords in addition to the usual C keywords. In addition, all tokens beginning with s**pi_** or **SPI_** are reserved for the Stream implementation.

array_in	kernel parameter attribute	identifies an array input stream
array_io	kernel parameter attribute	identifies an array input/output stream
array_out	kernel parameter attribute	identifies an array output stream
cond_in	kernel parameter attribute	identifies a conditional input stream
cond_out	kernel parameter attribute	identifies a conditional output stream
in	kernel parameter attribute	identifies a scalar input parameter or a sequential input stream
inline	type modifier	indicates an inlined kernel function
int16x2	DPU basic type	two 16-bit signed integers packed into a 32-bit word
int32x1	DPU basic type	one 32-bit signed integer
int8x4	DPU basic type	four 8-bit signed integers packed into a 32-bit word
kernel	type modifier	declares a kernel function for execution on the DPU
lrf_address	stream declaration attribute	specifies the LRF address of a stream
offset	stream attribute	specifies the offset of a substream
out	kernel parameter attribute	identifies a scalar output parameter or a sequential output stream
pragma	keyword	identifies a pragma
__repeat__	keyword	repeats a block of code a constant number of times (for loop unrolling)
seq_in	kernel parameter attribute	identifies a sequential input stream
seq_out	kernel parameter attribute	identifies a sequential output stream
size	stream declaration attribute stream attribute	specifies the size of a stream specifies the size of a substream
stream	type modifier	declares a stream
type	kernel parameter attribute	specifies the type of a stream or the type of a scalar kernel parameter
uint16x2	DPU basic type	two 16-bit unsigned integers packed into a 32-bit word
uint32x1	DPU basic type	one 32-bit unsigned integer
uint8x4	DPU basic type	four 8-bit unsigned integers packed into a 32-bit word
vec	type modifier	indicates a vector variable that holds **SPI_LANES** individual values (rather than a scalar variable that holds one value)

4.2 Predefined macros

The Stream compiler **spc** defines the following macros for use in Stream code.

Definition	**Description**	**spc** option
SPI_DEBUG	Defined if compiling for debug.	**-g**
SPI_DEVICE_DSP	Defined if compiling for DSP MIPS.	**-m sp16**, **-m sp8**, or **-m sp2x**
SPI_DEVICE_SP16	Defined if compiling for SP16 DSP MIPS.	**-m sp16**
SPI_DEVICE_SP16_SYS	Defined if compiling for SP16 System MIPS.	**-m sp16_sys**
SPI_DEVICE_SP8	Defined if compiling for SP8 DSP MIPS.	**-m sp8**
SPI_DEVICE_SP8_SYS	Defined if compiling for SP8 System MIPS.	**-m sp8_sys**
SPI_DEVICE_SP2X	Defined if compiling for Storm-2 DSP MIPS.	**-m sp2x**
SPI_DEVICE_SP2X_SYS	Defined if compiling for Storm-2 System MIPS.	**-m sp2x_sys**
SPI_DEVICE_SYS	Defined if compiling for System MIPS.	**-m sp16_sys**, **-m sp8_sys**, or **-m sp2x_sys**
SPI_FUNCTIONAL	Defined if compiling for functional mode (with **-z**).	**-z**
SPI_LANES	The number of lanes on the processor (16 or 8).	
SPI_LRF_SIZE	The LRF size in words per lane (4096 on SP16 and SP8).	
SPI_PROFILE	Defined if compiling for profiling.	**-p**
SPI_RELEASE	Defined if compiling with no **-g**, **-p**, or **-z** option.	
SPI_STREAMC	Defined if compiling Stream source (**.sc**, not **.c**).	
SPI_TESTBENCH	Defined if compiling testbench version.	**-m testbench**
SPI_TOOLS_VERSION_MAJOR	The major number of the SPI toolset.	
SPI_TOOLS_VERSION_MINOR	The minor number of the SPI toolset.	
SPI_TOOLS_VERSION_PATCH	The patch number of the SPI toolset.	

4.3 Types

Stream programs can use standard C data types. Signed integers use 2's complement representation. Floating point types use IEEE format. Stream stores multibyte data in littleendian format. If **unsigned integer i** contains 0x03020100, Stream stores its bytes to successive increasing memory locations as 0x00, 0x01, 0x02, 0x03. Similarly, if **unsigned short s** contains 0x0100, Stream stores its bytes to successive increasing memory locations as 0x00, 0x01.

Type	Unsigned variant	Bits	Bytes
char	**unsigned char**	8	1
short	**unsigned short**	16	2
int **long**	**unsigned int** **unsigned long**	32	4
pointer		32	4
float		32	4
double **long double**		64	8
long long	**unsigned long long**	64	8

4.4 DPU basic types

Kernels defined in Stream programs can use only special DPU basic types. All DPU basic types contain 32 bits.

Type	Synonyms	Description
int	**int32x1**	one 32-bit signed integer
int16x2		two 16-bit signed integers packed into a 32-bit word
int8x4		four 8-bit signed integers packed into a 32-bit word
unsigned int	**uint32x1** **unsigned int32x1**	one 32-bit unsigned integer
unsigned int16x2	**uint16x2**	two 16-bit unsigned integers packed into a 32-bit word
unsigned int8x4	**uint8x4**	four 8-bit unsigned integers packed into a 32-bit word

5 Component API

This chapter describes the Stream programming model Component API that supports high-level multi-core task level parallelism on a stream processor. It describes each type and each function defined by the Component API.

5.1 Component API types

This section presents an alphabetical listing of each type that the Stream programming model Component API defines. A summary table in the Component API types summary section of this document lists each Component API type. *Stream User's Guide* provides more general information on the Stream programming model and the Component API.

A program should `#include "spi_spm.h"` to get definitions for the Stream programming model Component API.

5.1.1 spi_binfo_valid_t

5.1.1.1 Prototype

```
#include "spi_spm.h"

typedef struct {
        unsigned int      valid_size;
} spi_binfo_valid_t;
```

5.1.1.2 Description

Type **spi_binfo_valid_t** contains information about a buffer. Member **valid_size** indicates the number of valid data bytes in the buffer.

5.1.1.3 See also

spi_buffer_flags_t

5.1.2 spi_buffer_flag_t

5.1.2.1 Prototype

```
#include "spi_spm.h"

typedef enum {
        SPI_BUFFER_FLAG_NONE          = 0,          /* none */
        SPI_BUFFER_FLAG_CACHED        = (1<<0),     /* cached */
        SPI_BUFFER_FLAG_READONLY      = (1<<1),     /* readonly */
        SPI_BUFFER_FLAG_UNMAPPED      = (1<<2),     /* unmapped */
        SPI_BUFFER_FLAG_TEMP          = (1<<3)      /* temporary */

} spi_buffer_flag_t;
```

5.1.2.2 Description

Enumerated type **spi_buffer_flag_t** defines buffer flags. Flags indicate special properties of a buffer. Flag values are unique powers of 2, so buffer flags may be OR'ed together as **spi_buffer_flags_t**. By default, a buffer resides in uncached memory, is not read only, is not unmapped, and is not temporary.

Temporary buffers can be allocated and used only within the execute function of a component. The Stream component runtime frees temporary buffers automatically when the execute function returns; they should not be freed manually.

5.1.2.3 See also

spi_buffer_flags_t

5.1.3 spi_buffer_flags_t

5.1.3.1 Prototype

#include "spi_spm.h"

typedef unsigned int spi_buffer_flags_t;

5.1.3.2 Description

Type **spi_buffer_flags_t** represents a set of **spi_buffer_flag_t** buffer flags OR'ed together.

spi_buffer_open opens a buffer; it takes a set of buffer flags as an argument.

5.1.3.3 See also

spi_buffer_flag_t, **spi_buffer_open**

5.1.4 spi_buffer_t

5.1.4.1 Prototype

#include "spi_spm.h"

5.1.4.2 Description

Opaque type **spi_buffer_t** represents a buffer in the Stream programming model. A buffer is a block of shared memory used to pass data between component instances.

spi_buffer_new creates a new buffer. **spi_pool_get_buffer** gets a buffer from a buffer pool.

5.1.4.3 See also

spi_buffer_flag_t, **spi_buffer_clone**, **spi_buffer_close**, **spi_buffer_free**, **spi_buffer_get_info**, **spi_buffer_get_size**, **spi_buffer_merge**, **spi_buffer_new**, **spi_buffer_open**, **spi_buffer_set_info**, **spi_connection_pop**, **spi_connection_push**, **spi_pool_get_buffer**

5.1.5 spi_cmd_t

5.1.5.1 Prototype

#include "spi_spm.h"

5.1.5.2 Description

Opaque type **spi_cmd_t** represents a command sent between component instances by **spi_cmd_send**.

5.1.5.3 See also

spi_cmd_free, **spi_cmd_get_desc**, **spi_cmd_get_id**, **spi_cmd_get_name**, **spi_cmd_get_payload**, **spi_cmd_get_payload_size**, **spi_cmd_register**, **spi_cmd_send**, **spi_cmd_send_response**

5.1.6 spi_component_flag_t

5.1.6.1 Prototype

```
#include "spi_spm.h"

typedef enum {
        SPI_COMPONENT_FLAG_NONE                 = 0,          /* none */
        SPI_COMPONENT_FLAG_BLOCKING             = (1 << 0),   /* blocking */
        SPI_COMPONENT_FLAG_AUTO_INSTANTIATE     = (1<<1)      /* auto-instantiate */
} spi_component_flag_t;
```

5.1.6.2 Description

Enumerated type **spi_component_flag_t** defines component flags. Flags indicate special properties of a component. Flag values are unique powers of 2, so component flags may be OR'ed together as **spi_component_flags_t**.

SPI_COMPONENT_FLAG_BLOCKING indicates that a component instance may block when running its execute function. A scheduling group that contains an instance of a blocking component cannot contain any other instances.

An instance of a component is created automatically when its scheduling group starts if its properties function uses **spi_component_set_flags** to set **SPI_COMPONENT_FLAG_AUTO_INSTANTIATE**.

5.1.6.3 See also

spi_component_flags_t, **spi_component_set_flags**

5.1.7 spi_component_flags_t

5.1.7.1 Prototype

#include "spi_spm.h"

typedef unsigned int spi_component_flags_t;

5.1.7.2 Description

Type **spi_component_flags_t** represents a set of **spi_component_flag_t** component flags OR'ed together.

spi_component_set_flags sets component flags from a component's properties function.

5.1.7.3 See also

spi_component_flag_t, **spi_component_set_flags**

5.1.8 spi_component_instance_cmdhandler_fn_t

5.1.8.1 Prototype

#include "spi_spm.h"

typedef void (spi_component_instance_cmdhandler_fn_t)
(
spi_instance_context_t *context***,**
spi_cmd_t *cmd*
);

5.1.8.2 Description

spi_component_instance_cmdhandler_fn_t is the type of a component instance command handler function, as specified as an argument to the **SPI_COMPONENT_NEW** macro. The instance command handler is called when an instance receives a command. It takes an instance-specific context argument, or **NULL** if the instance has no custom context, in addition to the command argument.

The instance must call **spi_cmd_send_response** to respond to the *cmd*. It may respond immediately, within the command handler function. Alternatively, it may save the *cmd* within the instance's context and respond at a later time. The instance must free the *cmd* object when it is no longer needed.

5.1.8.3 See also

SPI_COMPONENT_NEW, **spi_cmd_free**

5.1.9 spi_component_instance_destroy_fn_t

5.1.9.1 Prototype

#include "spi_spm.h"

typedef void (spi_component_instance_destroy_fn_t)(spi_instance_context_t *context***);**

5.1.9.2 Description

spi_component_instance_destroy_fn_t is the type of a component instance destroy function, as specified as an argument to the **SPI_COMPONENT_NEW** macro. The instance destroy function is called when a component instance is destroyed. It takes an instance-specific context argument, or **NULL** if the instance has no custom context. It returns no value.

5.1.9.3 See also

SPI_COMPONENT_NEW

5.1.10 spi_component_instance_execute_fn_t

5.1.10.1 Prototype

#include "spi_spm.h"

typedef void (spi_component_instance_execute_fn_t)(spi_instance_context_t *context***);**

5.1.10.2 Description

spi_component_instance_exec_fn_t is the type of a component instance execution function, as specified as an argument to the **SPI_COMPONENT_NEW** macro. The instance execute functions executes the component instance. It takes an instance-specific context argument, or **NULL** if the instance has no custom context. It returns no value.

5.1.10.3 See also

SPI_COMPONENT_NEW

5.1.11 spi_component_instance_init_fn_t

5.1.11.1 Prototype

#include "spi_spm.h"

typedef spi_instance_context_t (spi_component_instance_init_fn_t)(void);

5.1.11.2 Description

spi_component_instance_init_fn_t is the type of a component instance initialization function, as specified as an argument to the **SPI_COMPONENT_NEW** macro. The instance initialization function is called when **spi_instance_new** creates a component instance. It returns a context of type **spi_instance_context_t** that identifies the specific component instance.

5.1.11.3 See also

SPI_COMPONENT_NEW, **spi_instance_new**

5.1.12 spi_component_properties_fn_t

5.1.12.1 Prototype

#include "spi_spm.h"

typedef void (spi_component_properties_fn_t)(void);

5.1.12.2 Description

spi_component_properties_fn_t is the type of a component properties function, as specified as an argument to the **SPI_COMPONENT_NEW** macro. A component properties function takes no arguments and returns no value. It may call the following component API functions:

> **spi_component_set_flags** to set component properties,
> **spi_component_set_resource_requirements** to set component resource requirements,
> **spi_cmd_register** to register commands,
> **spi_port_register** to register ports, and
> **spi_exec_req_register** to register execution requirements.

However, it may not call many other component API functions (for example, **spi_exec_req_activate** to activate an execution requirement), and a runtime error will occur if it attempts to do so.

5.1.12.3 See also

spi_cmd_register, SPI_COMPONENT_NEW, **spi_component_set_flags**, **spi_component_set_resource_requirements**, **spi_exec_req_register**, **spi_port_register**

5.1.13 spi_component_t

5.1.13.1 Prototype

#include "spi_spm.h"

5.1.13.2 Description

Opaque type **spi_component_t** represents a Stream programming model component. Macro **SPI_COMPONENT_NEW** defines a component. **spi_component_find** and **spi_schedgroup_component_find** find a component.

5.1.13.3 See also

spi_component_find, **SPI_COMPONENT_NEW**, **spi_get_component**, **spi_schedgroup_component_find**

5.1.14 spi_connection_flag_t

5.1.14.1 Prototype

```
#include "spi_spm.h"

typedef enum {
        SPI_CONNECTION_FLAG_NONE            = 0,          /* none */
        SPI_CONNECTION_FLAG_INCOMING        = (1<<0),     /* incoming connection */
        SPI_CONNECTION_FLAG_OUTGOING        = (1<<1)      /* outgoing connection */
} spi_connection_flag_t;
```

5.1.14.2 Description

Enumerated type **spi_connection_flag_t** defines connection flags. Flag values are unique powers of 2, so connection flags may be OR'ed together as **spi_connection_flags_t**.

spi_connection_new connects to a contained component instance.

5.1.14.3 See also

spi_connection_flags_t, **spi_connection_new**

5.1.15 spi_connection_flags_t

5.1.15.1 Prototype

#include "spi_spm.h"

typedef unsigned int spi_connection_flags_t;

5.1.15.2 Description

Type **spi_connection_flags_t** represents a set of **spi_connection_flags_t** connection flags OR'ed together.

5.1.15.3

5.1.15.4 See also

spi_connection_flag_t

5.1.16 spi_connection_t

5.1.16.1 Prototype

#include "spi_spm.h"

5.1.16.2 Description

Opaque type **spi_connection_t** represents a connection between two component instances.

spi_connect connects two component instances. **spi_connection_new** connects to a contained instance.

5.1.16.3 See also

spi_connect, **spi_connection_get_depth**, **spi_connection_get_name**, **spi_connection_is_empty**, **spi_connection_is_full**, **spi_connection_new**, **spi_connection_pop**, **spi_connection_push**

5.1.17 spi_execution_requirement_t

5.1.17.1 Prototype

```
#include "spi_spm.h"

typedef enum {
        SPI_EXEC_ALWAYS         = 0,    /* always satisfied */
        SPI_EXEC_NEVER          = 1,    /* never satisfied */
        SPI_EXEC_ALLOF          = 2,    /* requires all of a set of requirements */
        SPI_EXEC_ANYOF          = 3,    /* requires any of a set of requirements */
        SPI_EXEC_PORT_ALLOF     = 4,    /* requires all connections of a port or set of ports to be ready */
        SPI_EXEC_PORT_ANYOF     = 5,    /* requires any connection of a port or set of ports to be ready */
        SPI_EXEC_FD_READ        = 6,    /* requires ability to read from a set of file descriptors */
        SPI_EXEC_FD_WRITE       = 7,    /* requires ability to write to a set of file descriptors */
        SPI_EXEC_POOL           = 8     /* requires that each of a set of pools must have avail. buffer */
} spi_execution_requirement_t;
```

5.1.17.2 Description

Enumerated type **spi_execution_requirement_t** defines an execution requirement for **spi_exec_req_register**.

5.1.17.3

5.1.17.4 See also

spi_exec_req_activate, **spi_exec_req_delete**, **spi_exec_req_is_satisfied**, spi**_exec_req_register**

5.1.18 spi_fb_pixel_type_t

5.1.18.1 Prototype

```
#include "spi_spm.h"

typedef enum {
        SPI_FB_LUT8          = (1 << 0),      /* 8-bit indexed color */
        SPI_FB_RGB555        = (1 << 1),      /* RGB 5-5-5-x (most significant bit unused) */
        SPI_FB_RGBA5551      = (1 << 2),      /* RGBA 5-5-5-1 */
        SPI_FB_RGB565        = (1 << 3),      /* RGB 5-6-5 */
        SPI_FB_RGB24         = (1 << 4),      /* RGB 8-8-8 */
        SPI_FB_RGB32         = (1 << 5),      /* RGB 8-8-8-x (most significant byte unused) */
        SPI_FB_RGBA32        = (1 << 6)       /* RGBA 8-8-8-8 */
} spi_fb_pixel_type_t;
```

5.1.18.2 Description

Enumerated type **spi_fb_pixel_type_t** defines Stream programming model framebuffer pixel types. Flag values are unique powers of 2, so image loading flags may be OR'ed together as **spi_fb_pixel_types_t** .

5.1.18.3 See also

spi_fb_pixel_types_t

5.1.19 spi_fb_pixel_types_t

5.1.19.1 Prototype

#include "spi_spm.h"

typedef unsigned int spi_fb_pixel_types_t;

5.1.19.2 Description

Type **spi_fb_pixel_types_t** represents a set of **spi_fb_pixel_type_t** flags. **spi_fb_get_pixel_type** gets the pixel type of a framebuffer.

5.1.19.3 See also

spi_fb_get_pixel_type, **spi_fb_pixel_type_t**

5.1.20 spi_image_flag_t

5.1.20.1 Prototype

```
#include "spi_spm.h"

typedef enum {
        SPI_IMAGE_FLAG_NONE     = 0,        /* none */
        SPI_IMAGE_FLAG_UART     = (1<<0)    /* redirect stdout and stderr to UART on device */
} spi_image_flag_t;
```

5.1.20.2 Description

Enumerated type **spi_image_flag_t** defines flags that control image loading. Flag values are unique powers of 2, so image loading flags may be OR'ed together as **spi_image_flags_t**.

5.1.20.3

5.1.20.4 See also

spi_image_flags_t

5.1.21 spi_image_flags_t

5.1.21.1 Prototype

#include "spi_spm.h"

typedef unsigned int spi_image_flags_t;

5.1.21.2 Description

Type **spi_image_flags_t** represents a set of **spi_image_flag_t** flags. **spi_load_image** observes the specified flags when it loads a DSP MIPS image.

5.1.21.3 See also

spi_image_flag_t, **spi_load_image**

5.1.22 spi_instance_context_t

5.1.22.1 Prototype

#include "spi_spm.h"

5.1.22.2 Description

Opaque type **spi_instance_context_t** represents a custom per-instance context. A Stream programming model application may contain multiple instances of the same component. Each instance may be uniquely identified by a per-instance context. The context must be a pointer or an object that can be cast to a pointer without loss of data. The functions supplied as arguments to a **SPI_COMPONENT_NEW** component definition define and manipulate instance contexts.

5.1.22.3 See also

SPI_COMPONENT_NEW

5.1.23 spi_instance_state_t

5.1.23.1 Prototype

```
#include "spi_spm.h"

typedef enum {
        SPI_INSTANCE_STATE_UNKNOWN,   /* unknown */
        SPI_INSTANCE_STATE_STOPPED,   /* stopped */
        SPI_INSTANCE_STATE_PAUSED,    /* paused */
        SPI_INSTANCE_STATE_RUNNING    /* running */
} spi_instance_state_t;
```

5.1.23.2 Description

Enumerated type **spi_instance_state_t** defines component instance states. A component instance is either stopped, paused, or running.

A stopped instance is in state **SPI_INSTANCE_STATE_STOPPED**. This is a terminal state: once an instance is stopped, it remains stopped. The execute function of a stopped instance will never be invoked, even if all its execution requirements are satisfied. The command handler function of a stopped instance will never be invoked, even if the instance receives a command.

A paused instance is in state **SPI_INSTANCE_STATE_PAUSED**. A paused instance may change state to stopped or running. The execute function of a paused instance will never be invoked, even if all its execution requirements are satisfied. However, the command handler function of a paused instance will be called when the instance receives a command. **spi_instance_new** starts newly created instances in the paused state (though the component's instance initialization function may change it to the running state).

A running instance is in state **SPI_INSTANCE_STATE_RUNNING**. A running instance may change state to stopped or paused. The execute function of a running instance will be called when all its execution requirements are satisfied. The command handler function of the instance will be called when the instance receives a command.

spi_get_state gets the current instance state and **spi_set_state** sets the current instance state. **spi_cmd_send** can issue a built-in command **SPI_CMD_STOP**, **SPI_CMD_PAUSE**, or **SPI_CMD_START** to change the current state.

5.1.23.3 See also

spi_cmd_send, **spi_get_state**, **spi_instance_new**, **spi_set_state**

5.1.24 spi_instance_t

5.1.24.1 Prototype

#include "spi_spm.h"

5.1.24.2 Description

Opaque type **spi_instance_t** represents a Stream programming model component instance.

spi_instance_new creates a component instance from a component. **spi_get_instance** gets a component instance with a given name. **spi_cmd_send** sends a command to an instance. **spi_port_export** exports a port on a contained instance. **spi_connect** and **spi_connection_new** connect instances.

5.1.24.3 See also

spi_cmd_send, **spi_connect**, **spi_connection_new**, **spi_get_instance**, **spi_instance_new**, **spi_port_export**

5.1.25 spi_payload_timer_t

5.1.25.1 Prototype

```
#include "spi_spm.h"

typedef struct {
        unsigned int        count;          /* number of times timer was started */
        unsigned long long  nanoseconds;    /* total nanoseconds timer was running */
} spi_payload_timer_t;
```

5.1.25.2 Description

Type **spi_payload_timer_t** represents timer information returned by a **SPI_CMD_GET_TIMER** command response. Member **count** gives the number of times the timer was started. Member **nanoseconds** gives the total number of nanoseconds the timer was running.

5.1.25.3 See also

spi_cmd_send

5.1.26 spi_payload_type_t

5.1.26.1 Prototype

```
#include "spi_spm.h"

typedef enum {
        SPI_PAYLOAD_UNKNOWN         = (UINT_MAX),        /* unknown */
        SPI_PAYLOAD_NULL            = (UINT_MAX - 1),    /* NULL */
        SPI_PAYLOAD_INT32           = (UINT_MAX - 2),    /* 32-bit signed integer */
        SPI_PAYLOAD_UINT32          = (UINT_MAX - 3),    /* 32-bit unsigned integer */
        SPI_PAYLOAD_INT64           = (UINT_MAX - 4),    /* 64-bit signed integer */
        SPI_PAYLOAD_UINT64          = (UINT_MAX - 5),    /* 64-bit unsigned integer */
        SPI_PAYLOAD_FLOAT           = (UINT_MAX - 6),    /* 32-bit float */
        SPI_PAYLOAD_DOUBLE          = (UINT_MAX - 7),    /* 64-bit double */
        SPI_PAYLOAD_STRING          = (UINT_MAX - 8),    /* NUL-terminated string */
        SPI_PAYLOAD_TIMER           = (UINT_MAX - 9),    /* spi_payload_timer_t timer */
} spi_payload_type_t;
```

5.1.26.2 Description

Enumerated type **spi_payload_type_t** represents the type of a command payload or response. **spi_cmd_register** registers a command with a given payload type and response payload type.

5.1.26.3 See also

spi_cmd_get_payload_type, **spi_cmd_get_response_payload_type**, **spi_cmd_register, spi_response_get_payload_type**

5.1.27 spi_pel_t

5.1.27.1 Prototype

```
#include "spi_spm.h"

typedef enum {
        SPI_PEL_NONE            = 0,        /* none */
        SPI_PEL_SYSTEM_MIPS     = (1<<0),   /* System MIPS */
        SPI_PEL_DSP_MIPS,       = (1<<1),   /* DSP MIPS */
        SPI_PEL_DSU_MIPS        = (1<<2),   /* DSU (Storm-2) */

} spi_pel_t;
```

5.1.27.2 Description

Enumerated type **spi_pel_t** represents a processing element (i.e., a hardware resource) on which a scheduling group may execute. Processing element values are unique powers of 2, so processing elements may be OR'ed together as **spi_pels_t**.

5.1.27.3 See also

spi_pels_t, **spi_schedgroup_t**

5.1.28 spi_pels_t

5.1.28.1 Prototype

#include "spi_spm.h"

typedef unsigned int spi_pels_t;

5.1.28.2 Description

Type **spi_pels_t** represents a set of **spi_pel_t** processing element flags OR'ed together.

SPI_PEL_ALL is the set of all processing elements, namely ((**spi_pels_t**)(**SPI_PEL_SYSTEM_MIPS | SPI_PEL_DSP_MIPS**) on Storm-1 or ((**spi_pels_t**)(**SPI_PEL_SYSTEM_MIPS | SPI_PEL_DSP_MIPS | SPI_PEL_DSU_MIPS**) on Storm-2.

5.1.28.3 See also

spi_pel_t

5.1.29 spi_pool_flag_t

5.1.29.1 Prototype

```
#include "spi_spm.h"

typedef enum {
        SPI_POOL_FLAG_NONE              = 0,            /* none */
        SPI_POOL_FLAG_CONTIGUOUS        = (1<<0),       /* create pool in contiguous shared memory */
        SPI_POOL_FLAG_GROW              = (1<<1)        /* allow pool to grow as needed */
} spi_pool_flag_t;
```

5.1.29.2 Description

Enumerated type **spi_pool_flag_t** defines buffer pool flags. Flags indicate special buffer pool properties. Flag values are unique powers of 2, so pool flags may be OR'ed together as **spi_pool_flags_t**.

SPI_POOL_FLAG_CONTIGUOUS indicates that initial buffers should be created from a contiguous region of shared memory.

SPI_POOL_FLAG_GROW indicates that a pool may create additional buffers as needed. If set, **spi_pool_get_buffer** attempts to allocate additional buffers if no buffers are available. If not set, **spi_pool_get_buffer** returns **NULL** if no buffers are available.

5.1.29.3 See also

spi_pool_flags_t, **spi_pool_get_buffer**

5.1.30 spi_pool_flags_t

5.1.30.1 Prototype

#include "spi_spm.h"

typedef unsigned int spi_pool_flags_t;

5.1.30.2 Description

Type **spi_pool_flags_t** represents a set of **spi_pool_flag_t** pool flags OR'ed together.

spi_pool_new creates a buffer pool with a given buffer count, size and alignment, observing given pool flags.

5.1.30.3 See also

spi_pool_flag_t, **spi_pool_new**

5.1.31 spi_portdir_t

5.1.31.1 Prototype

```
#include "spi_spm.h"

typedef enum {
        SPI_PORT_UNKNOWN,       /* unknown port direction */
        SPI_PORT_IN,            /* input port */
        SPI_PORT_OUT            /* output port */
} spi_portdir_t;
```

5.1.31.2 Description

Enumerated type **spi_portdir_t** gives the allowed directions for a component instance connection port. A port is either an input port or an output port.

spi_port_register defines a port for a component. **spi_port_get_dir** returns the direction of a port.

5.1.31.3 See also

spi_port_get_dir, **spi_port_register**

5.1.32 spi_provider_t

5.1.32.1 Prototype

```
#include "spi_spm.h"

typedef enum {
        SPI_PROVIDER_UNKNOWN    = 0,    /* unknown provider */
        SPI_PROVIDER_SPI        = 1     /* Stream Processors, Inc. */
} spi_provider_t;
```

5.1.32.2 Description

Enumerated type **spi_provider_t** defines stream programming model providers. SPI assigns each organization that produces SPM components a unique provider id. The provider id allows the SPM to disambiguate identically named components or scheduling groups from different providers.

Negative provider IDs are reserved for internal use within an organization. Programmers may use negative provider IDs freely for internal components, but only assigned provider IDs should be used for externally exposed components.

SPI_COMPONENT_NEW defines a component, including its provider. **SPI_SCHEDGROUP_NEW** creates a new scheduling group with a specified provider. **spi_schedgroup_register_component** registers a component with a scheduling group. **spi_component_get_provider** returns the provider of a component.

5.1.32.3 See also

spi_component_get_provider, **SPI_COMPONENT_NEW**, **SPI_SCHEDGROUP_NEW**, **spi_schedgroup_register_component**

5.1.33 spi_resource_t

5.1.33.1 Prototype

```
#include "spi_spm.h"

typedef enum {
        SPI_RESOURCE_NONE       = 0,        /* no resource */
        SPI_RESOURCE_DPU        = (1<<0),   /* data parallel unit (DPU) */
        SPI_RESOURCE_DSU        = (1<<1),   /* data serial unit (Storm-2) */
        SPI_RESOURCE_ME         = (1<<2)    /* motion estimation unit (Storm-2) */

} spi_resource_t;
```

5.1.33.2 Description

Enumerated type **spi_resource_t** represents Stream programming model resources. A resource is a unique hardware or software entity. Resource values are unique powers of 2, so resources may be OR'ed together as **spi_resources_t**.

5.1.33.3 See also

spi_resources_t

5.1.34 spi_resources_t

5.1.34.1 Prototype

#include "spi_spm.h"

typedef unsigned int spi_resources_t;

5.1.34.2 Description

Type **spi_resources_t** represents a set of **spi_resource_t** resources OR'ed together.

Functions **spi_schedgroup_set_controlled_resources** and **spi_component_set_resource_requirements** set the resource requirements for a scheduling group and for a component.

5.1.34.3 See also

spi_component_set_resource_requirements, **spi_resource_t**, **spi_schedgroup_set_controlled_resources**

5.1.35 spi_response_context_t

5.1.35.1 Prototype

#include "spi_spm.h"

5.1.35.2 Description

Opaque type **spi_response_context_t** represents a Stream programming model per-command response context. The context must be a pointer type or a type that can be cast to a pointer type without loss of data.

5.1.35.3 See also

5.1.36 spi_response_errno_t

5.1.36.1 Prototype

```
#include "spi_spm.h"

typedef enum {
        SPI_RESPONSE_ERRNO_OK,                      /* ok */
        SPI_RESPONSE_ERRNO_FAIL,                            /* failure */
        SPI_RESPONSE_ERRNO_NULL,                    /* NULL response */
        SPI_RESPONSE_ERRNO_UNKNOWN_CMD,             /* unknown command */
        SPI_RESPONSE_ERRNO_INVALID_PAYLOAD,                 /* invalid payload */
        SPI_RESPONSE_ERRNO_STOPPED,                 /* receiving instance is stopped */
        SPI_RESPONSE_ERRNO_DISCONNECTED,            /* component instance disconnected */
        SPI_RESPONSE_ERRNO_MAX                      /* max response error code value */
} spi_response_errno_t;
```

5.1.36.2 Description

Enumerated type **spi_reponse_errno_t** defines built-in command response error codes.

5.1.36.3 See also

spi_cmd_send_response, **spi_response_get_errno**, **spi_response_strerror**

5.1.37 spi_response_handler_fn_t

5.1.37.1 Prototype

#include "spi_spm.h"

typedef void (spi_response_handler_fn_t)
(
spi_response_t *response***,**
spi_response_context_t *context*
);

5.1.37.2 Description

Type **spi_response_handler_fn_t** represents a command response handler function. **spi_response_set_handler** specifies a command response handler.

5.1.37.3 See also

spi_response_context_t, **spi_repsponse_set_handler**, **spi_response_t**

5.1.38 spi_response_t

5.1.38.1 Prototype

#include "spi_spm.h"

5.1.38.2 Description

Opaque type **spi_response_t** represents a command response token. When **spi_cmd_send** sends a command, it returns a **spi_reponse_t** token as a proxy for the actual command response. When the receiving instance finishes processing the command and returns a response, the sending instance uses the command reponse token to identify the command.

spi_response_set_handler sets a handler function to handle a command response.

5.1.38.3 See also

5.1.39 spi_schedgroup_properties_t

5.1.39.1 Prototype

#include "spi_spm.h"

typedef void (spi_schedgroup_properties_fn_t)(void);

5.1.39.2 Description

Type **spi_schedgroup_properties_t** represents the type of a scheduling group properties function. Macro **SPI_SCHEDGROUP_NEW** specifies a properties function as part of the scheduling group definition.

5.1.39.3 See also

SPI_SCHEDGROUP_NEW, **spi_schedgroup_register_component**, **spi_schedgroup_set_controlled_resources**, **spi_schedgroup_set_processing_element**

5.1.40 spi_spm_flag_t

5.1.40.1 Prototype

```
#include "spi_spm.h"

typedef enum {
        SPI_SPM_FLAG_NONE                = 0,          /* none */
        SPI_SPM_FLAG_NO_REGISTRY         = (1<<0),     /* do not start system-wide registry */
        SPI_SPM_FLAG_NO_RESET            = (1<<1)      /* do not reset DSP MIPS (and DSU if Storm-2)
*/

} spi_spm_flag_t;
```

5.1.40.2 Description

Enumerated type **spi_spm_flag_t** defines Stream programming model flags. Flag values are unique powers of 2, so pool flags may be OR'ed together as **spi_spm_flags_t**. **spi_spm_start** takes a set of Stream programming model flags as an argument.

SPI_SPM_FLAG_NO_REGISTRY indicates that the Stream programming model runtime should not start the system-wide registry. The entire system should contain only one registry. For example, if an application includes both a System MIPS part and a DSP MIPS part and each calls **spi_spm_start**, the second **spi_spm_start** call (typically, the one not on System MIPS) should specify the **SPI_SPM_FLAG_NO_REGISTRY** flag.

SPI_SPM_FLAG_NO_RESET indicates that System MIPS should not reset DSP MIPS (and, on Storm-2, not reset the DSU). It may be used only on System MIPS.

5.1.40.3 See also

spi_spm_flags_t, **spi_spm_start**

5.1.41 spi_spm_flags_t

5.1.41.1 Prototype

#include "spi_spm.h"

typedef unsigned int spi_spm_flags_t;

5.1.41.2 Description

Type **spi_spm_flags_t** represents a set of **spi_spm_flag_t** Stream programming model flags OR'ed together.

5.1.41.3 See also

spi_spm_flag_t

5.1.42 spi_timer_t

5.1.42.1 Prototype

#include "spi_spm.h"

5.1.42.2 Description

Type **spi_timer_t** represents a Stream programming model timer.

spi_timer_new creates a timer. **spi_get_timer** gets a timer with a given name. **spi_timer_start** and **spi_timer_stop** start and stop a timer. **spi_timer_get_nanoseconds** and **spi_timer_get_total_nanoseconds** return timings from a timer.

See **spi_get_timer** for a description of built-in timers provided by the Stream programming model.

5.1.42.3 See also

spi_get_timer, **spi_timer_get_desc**, **spi_timer_get_name**, **spi_timer_get_nanoseconds**, **spi_timer_get_start_count**, **spi_timer_get_total_nanoseconds**, **spi_timer_new**, **spi_timer_start**, **spi_timer_stop**

5.2 Component API functions

This section presents an alphabetical listing of each function that the Stream programming model Component API defines. A table in the Component API functions summary section of this document lists each Component API function. The *Stream User's Guide* provides more general information on the Stream programming model and on the Component API.

A program should `#include "spi_spm.h"` to get definitions for the Stream programming model Component API.

5.2.1 spi_buffer_clone

5.2.1.1 Prototype

#include "spi_spm.h"

spi_buffer_t
spi_buffer_clone(spi_buffer_t *buffer***);**

5.2.1.2 Description

spi_buffer_clone clones the given *buffer*. Upon successful completion, both the original *buffer* and the returned buffer represent the same region of shared memory. Only the component instance that owns a buffer may clone it.

5.2.1.3 Return value

On success, **spi_buffer_clone** returns a new buffer that represents the same region of shared memory as *buffer*. On failure, it returns **NULL**.

5.2.1.4 See also

spi_buffer_free, **spi_buffer_merge**

5.2.2 spi_buffer_close

5.2.2.1 Prototype

#include "spi_spm.h"

int
spi_buffer_close(spi_buffer_t *buffer***);**

5.2.2.2 Description

spi_buffer_close closes the given *buffer*. When it completes, the pointer returned by **spi_buffer_open** for the buffer becomes invalid. Only the component instance that owns a buffer may close it.

5.2.2.3 Return value

spi_buffer_close returns zero on success, non-zero on failure.

5.2.2.4 See also

spi_buffer_open

5.2.3 spi_buffer_free

5.2.3.1 Prototype

#include "spi_spm.h"

int
spi_buffer_free(spi_buffer_t *buffer***);**

5.2.3.2 Description

spi_buffer_free frees the given *buffer*. If *buffer* has active clones, then *buffer* must not have been written. If the buffer has been written and has active clones, then **spi_buffer_merge** must be used instead to free the buffer. The shared memory that *buffer* represents becomes available for reuse once all buffer clones have been freed or merged.

spi_buffer_new allocates a buffer. **spi_pool_get_buffer** gets a buffer from a buffer pool.

5.2.3.3 Return value

spi_buffer_free returns zero on success and non-zero on failure.

5.2.3.4 See also

spi_buffer_clone, **spi_buffer_merge**, **spi_buffer_new**, **spi_pool_get_buffer**

5.2.4 spi_buffer_get_info

5.2.4.1 Prototype

#include "spi_spm.h"

void *
spi_buffer_get_info(spi_buffer_t *buffer***);**

5.2.4.2 Description

spi_buffer_get_info gets the information associated with the given *buffer*. The returned pointer references an object owned by *buffer*. When the ownership of *buffer* is released, the returned pointer becomes invalid. Before ownership of *buffer* is released, the information associated with *buffer* may be modified via the returned pointer.

SPI_MAX_BUFFER_INFO_SIZE defines the maximum size (in bytes) of the information attached to a buffer.
spi_buffer_get_info_size gets the size of the info associated with a buffer.

5.2.4.3 Return value

spi_buffer_get_info returns a pointer to the information associated with *buffer*.

5.2.4.4 See also

spi_buffer_get_info_size, **spi_buffer_set_info**

5.2.5 spi_buffer_get_info_size

5.2.5.1 Prototype

#include "spi_spm.h"

unsigned int
spi_buffer_get_info_size(spi_buffer_t *buffer***);**

5.2.5.2 Description

spi_buffer_get_info_size gets the size of the information associated with the given *buffer*.

SPI_MAX_BUFFER_INFO_SIZE defines the maximum size (in bytes) of the information attached to a buffer.
spi_buffer_get_info gets the info associated with a buffer.

5.2.5.3 Return value

spi_buffer_get_info_size returns the size of the information associated with *buffer*, or zero if the buffer has no associated informationi.

5.2.5.4 See also

spi_buffer_get_info, **spi_buffer_set_info**

5.2.6 spi_buffer_get_size

5.2.6.1 Prototype

#include "spi_spm.h"

unsigned int
spi_buffer_get_size(spi_buffer_t *buffer***);**

5.2.6.2 Description

spi_buffer_get_size gets the size in bytes of the given *buffer*. Each buffer has a fixed size and a known alignment.

5.2.6.3 Return value

spi_buffer_get_size returns the size in bytes of *buffer*.

5.2.6.4 See also

5.2.7 spi_buffer_merge

5.2.7.1 Prototype

```
#include "spi_spm.h"

spi_buffer_t
spi_buffer_merge(
        spi_buffer_t *          buffers,
        unsigned int            count
        );
```

5.2.7.2 Description

spi_buffer_merge merges the given array of cloned *buffers* of length *count*. All of the *buffers* must be clones of the same buffer. **spi_buffer_merge** frees the *buffers* and returns a new buffer that represents the shared memory of the cloned *buffers*.

5.2.7.3 Return value

On success, **spi_buffer_merge** returns a buffer representing the same memory as the given cloned buffers. On failure, it returns **NULL**.

5.2.7.4 See also

spi_buffer_clone, **spi_buffer_free**

5.2.8 spi_buffer_new

5.2.8.1 Prototype

```
#include "spi_spm.h"

spi_buffer_t
spi_buffer_new(
        unsigned int            size,
        unsigned int            alignment,
        spi_buffer_flags_t      flags
        );
```

5.2.8.2 Description

spi_buffer_new creates a new buffer with the given *size* in bytes and the given *alignment*. An *alignment* of 0 indicates the buffer should use the default alignment (cache-aligned, to a multiple of 32 bytes). The *flags* specify desired buffer flags; for SP8 and SP16, it should always be **SPI_BUFFER_FLAG_NONE**. The new buffer does not belong to a buffer pool; it resides in uncached memory, is not read only, and is not unmapped.

5.2.8.3 Return value

On success, **spi_buffer_new** returns a buffer with the given properties. On failure, it returns **NULL**.

5.2.8.4 See also

spi_pool_new

5.2.9 spi_buffer_open

5.2.9.1 Prototype

```
#include "spi_spm.h"

void *
spi_buffer_open(
        spi_buffer_t            buffer,
        spi_buffer_flags_t      flags
        );
```

5.2.9.2 Description

spi_buffer_open returns a pointer to the shared memory for the given *buffer*. The *flags* indicate special buffer properties, such as whether the buffer is readonly or resides in cached memory.

spi_buffer_close closes a buffer.

5.2.9.3 Return value

spi_buffer_open returns a pointer to the contents of the *buffer* on success or **NULL** on failure.

5.2.9.4 See also

spi_buffer_close

5.2.10 spi_buffer_set_info

5.2.10.1 Prototype

```
#include "spi_spm.h"

int
spi_buffer_set_info(
        spi_buffer_t    buffer,
        void *          info,
        unsigned int    size
        );
```

5.2.10.2 Description

spi_buffer_set_info sets the information associated with the given *buffer* to the *size* bytes at location *info*. When **spi_connection_push** pushes a buffer onto a connection, a copy of the associated information is attached to the pushed buffer. Thus, changing the *info* object after the push does not change the information associated with the buffer; the information may be changed before pushing the buffer.

SPI_MAX_BUFFER_INFO_SIZE defines the maximum size (in bytes) of the information attached to a buffer.

If *info* is **NULL**, no information is associated with *buffer*.

5.2.10.3 Return value

spi_buffer_set_info returns zero on success or non-zero on failure.

5.2.10.4 See also

spi_buffer_get_info

5.2.11 spi_cmd_free

5.2.11.1 Prototype

#include "spi_spm.h"

int
spi_cmd_free(spi_cmd_t *command***);**

5.2.11.2 Description

spi_cmd_free frees the given *command*, including its payload.

5.2.11.3 Return value

spi_cmd_free returns zero on success or non-zero on failure.

5.2.11.4 See also

spi_cmd_get_id, **spi_cmd_get_name**, **spi_cmd_register**

5.2.12 spi_cmd_get_desc

5.2.12.1 Prototype

#include "spi_spm.h"

const char *
spi_cmd_get_desc(spi_cmd_t *command***);**

5.2.12.2 Description

spi_cmd_get_desc gets the description of the given *command*.

spi_cmd_register defines a command, specifying a description.

5.2.12.3 Return value

spi_cmd_get_desc returns the description of *command*. The returned string should not be modified.

5.2.12.4 See also

spi_cmd_get_id, **spi_cmd_get_name**, **spi_cmd_register**

5.2.13 spi_cmd_get_id

5.2.13.1 Prototype

#include "spi_spm.h"

unsigned int
spi_cmd_get_id(spi_cmd_t *command***);**

5.2.13.2 Description

spi_cmd_get_id gets the ID for the given *command*.

spi_cmd_register defines a command, specifying an ID.

5.2.13.3 Return value

spi_cmd_get_id returns the ID for *command*.

5.2.13.4 See also

spi_cmd_get_desc, **spi_cmd_get_name**, **spi_cmd_register**

5.2.14 spi_cmd_get_name

5.2.14.1 Prototype

#include "spi_spm.h"

const char *
spi_cmd_get_name(spi_cmd_t *command***);**

5.2.14.2 Description

spi_cmd_get_name gets the name of the given *command*.

spi_cmd_register defines a command, speciifying a name.

5.2.14.3 Return value

spi_cmd_get_name returns the name of *command*. The returned string should not be modified.

5.2.14.4 See also

spi_cmd_get_desc, **spi_cmd_get_id**, **spi_cmd_register**

5.2.15 spi_cmd_get_payload

5.2.15.1 Prototype

#include "spi_spm.h"

void *
spi_cmd_get_payload(spi_cmd_t *command***);**

5.2.15.2 Description

spi_cmd_get_payload get the payload of the given *command*. The actual payload depends on the command.
spi_cmd_get_payload_size returns the payload size of a command.

spi_cmd_send sends a command with a specified payload and payload size.

5.2.15.3 Return value

spi_cmd_get_payload returns the payload of *command*.

5.2.15.4 See also

spi_cmd_get_payload_size, **spi_cmd_send**

5.2.16 spi_cmd_get_payload_size

5.2.16.1 Prototype

#include "spi_spm.h"

unsigned int
spi_cmd_get_payload_size(spi_cmd_t *command***);**

5.2.16.2 Description

spi_cmd_get_payload_size gets the payload size in bytes of the given *command*. **spi_cmd_get_payload** gets the payload.

spi_cmd_register registers a command, defining its payload type. **spi_cmd_send** sends a command with a specified payload and payload size.

5.2.16.3 Return value

spi_cmd_get_payload_size returns the *command* payload size in bytes. A return value of zero indicates that the payload is just the payload pointer itself; it may be an actual pointer or a value that can be safely cast to a pointer. For example, an integer value can be sent as a payload by casting it to **void *** and sending it as the payload with a payload size of 0.

5.2.16.4 See also

spi_cmd_get_payload, **spi_cmd_register**, **spi_cmd_send**

5.2.17 spi_cmd_get_payload_type

5.2.17.1 Prototype

#include "spi_spm.h"

spi_payload_type_t
spi_cmd_get_payload_type(spi_cmd_t *command***);**

5.2.17.2 Description

spi_cmd_get_payload_type gets the payload type of the given *command*.

spi_cmd_get_payload gets the payload. **spi_cmd_send** sends a command with a specified payload and payload type.

5.2.17.3 Return value

spi_cmd_get_payload_type returns the *command* payload type.

5.2.17.4 See also

spi_cmd_get_payload, **spi_cmd_send**

5.2.18 spi_cmd_get_response_payload_type

5.2.18.1 Prototype

#include "spi_spm.h"

spi_payload_type_t
spi_cmd_get_response_payload_type(spi_cmd_t *command***);**

5.2.18.2 Description

spi_cmd_get_response_payload_type gets the response payload type of the given *command*.

spi_cmd_send_response sends a command repsonse with a specified response payload and response payload type.
spi_response_get_payload gets a payload response.

5.2.18.3 Return value

spi_cmd_get_response_payload_type returns the response payload type of the given *command*.

5.2.18.4 See also

spi_cmd_send_response, **spi_response_get_payload**

5.2.19 spi_cmd_register

5.2.19.1 Prototype

```
#include "spi_spm.h"

int
spi_cmd_register(
        unsigned int            id,
        const char *            description,
        spi_payload_type_t      command_payload_t,
        spi_payload_type_t      response_payload_t
        );
```

5.2.19.2 Description

spi_cmd_register defines a command for a component. The parameters define the numerical *id*, *description*, *command_payload_type*, and *response_payload_type* of the command.

spi_cmd_register uses the stringized *id* argument as the name of the command. For example, if the header file for component FOO defines **#define FOO_CMD_DOIT 1** and a program uses **FOO_CMD_DOIT** as the *id* argument to **spi_cmd_register**, the name of the command will be "FOO_CMD_DOIT".

5.2.19.3 Return value

spi_cmd_register returns zero on success, non-zero on failure.

5.2.19.4 See also

spi_cmd_get_desc, **spi_cmd_get_id**, **spi_cmd_get_name**, **spi_cmd_get_payload_type, spi_cmd_get_response_payload_type, spi_cmd_send**

5.2.20 spi_cmd_send

5.2.20.1 Prototype

```
#include "spi_spm.h"

spi_response_t
spi_cmd_send(
        spi_instance_t    instance,
        unsigned int      id,
        void *            payload,
        unsigned int      payload_size
        );
```

5.2.20.2 Description

spi_cmd_send sends a command to a component *instance*. Parameters *id*, *payload*, and *payload_size* specify the command, its payload and its payload size in bytes. The payload and its size depend on the command, as given in the **spi_cmd_register** call that registers the command. **SPI_MAX_COMMAND_PAYLOAD_SIZE** defines the maximum size (in bytes) of a command payload.

The behavior of **spi_cmd_send** differs depending on whether it is called from a Stream application or from a component instance. A **spi_cmd_send** call from a Stream application returns only when the receiving instance returns a response; that is, the **spi_cmd_send** blocks while awaiting a response. The application should free the **spi_response_t** object returned by **spi_cmd_send** with **spi_response_free** when it is no longer needed.

In contrast, a **spi_cmd_send** call from a component instance always returns a **spi_response_t** response token immediately, before the receiving instance returns the actual response. If the sending instance does not need to be notified of the actual response, it should free the returned **spi_response_t** with **spi_response_free**. If the sending instance does need to be notified of the actual response, it should call **spi_response_set_handler** to register a response handler. The sending instance will execute the registered response handler when it receives the actual response from the receiving instance. The sending instance should free the **spi_response_t** response once it is no longer needed.

spi_instance_new creates a component instance, returning an instance handle. As there is no other way to obtain an instance handle, **spi_cmd_send** can only send commands to instances within the hierarchy of instances created under an instance, not to arbitrary instances.

Built-in command IDs **SPI_CMD_START**, **SPI_CMD_PAUSE**, and **SPI_CMD_STOP** start, pause, and stop a component instance. spi_instance_state_t describes instance states. These commands have no payload and no response payload, so each has a command payload type and a response payload type of **SPI_PAYLOAD_NONE**.

Build-in command ID **SPI_CMD_GET_TIMER** gets a timer in an instance. The command payload is the timer name (command payload type **SPI_PAYLOAD_STRING**) and the command response is a **spi_payload_timer_t** object containing timer information (repsonse payload type **SPI_PAYLOAD_TIMER**).

Built-in command IDs **SPI_CMD_GET_PRIORITY** and **SPI_CMD_SET_PRIORITY** get and set the scheduling priority of a component instance. **SPI_CMD_GET_PRIORITY** has no payload (command payload type **SPI_PAYLOAD_NULL**); its response (response payload type **SPI_PAYLOAD_UINT32**) is the priority of the instance. The command payload of **SPI_CMD_SET_PRIORITY** is the priority for the instance (command payload type **SPI_PAYLOAD_UINT32**); it has no response payload (response payload type **SPI_PAYLOAD_NULL**).

5.2.20.3 Return value

spi_cmd_send returns a response handle of type **spi_response_t** if it sends the command successfully, or NULL on failure. As noted in the Description above, it returns immediately if called from a component instance, while it blocks if called from the application.

The receiving *instance* returns a handle. Eventually, the command handler of the receiving *instance* (specified by the **spi_component_instance_cmdhandler_fn_t** function in the **SPI_COMPONENT_NEW** component definition) finishes processing the command. It then calls **spi_cmd_send_response** to send an actual command response. The reponse handler of the sending instance (specified by **spi_response_set_handler**) handles the response, using the **spi_response_t** context that **spi_cmd_send** returned to identify the command.

5.2.20.4 See also

spi_cmd_register, **spi_cmd_send_response**, **spi_component_instance_cmdhandler_fn_t**, **SPI_COMPONENT_NEW**, **spi_instance_state_t**, **spi_response_set_handler**, **spi_response_t**, **spi_payload_timer_t**

5.2.21 spi_cmd_send_response

5.2.21.1 Prototype

#include "spi_spm.h"

```
int
spi_cmd_send_response(
        spi_cmd_t                 command,
        spi_response_errno_t      error_code,
        void *                    payload,
        unsigned int              payload_size
        );
```

5.2.21.2 Description

spi_cmd_send_response sends a response to the given *command*. The response consists of an *error_code* and a *payload*, with a *payload_size* in bytes, as given in the **spi_cmd_register** call that registers the command. Error code **SPI_RESPONSE_ERRNO_OK** indicates that *command* completed successfully; any other error code indicates that *command* failed.

SPI_MAX_RESPONSE_PAYLOAD_SIZE defines the maximum size (in bytes) of a response payload.

5.2.21.3 Return value

spi_cmd_send_response returns zero if the response was sent, non-zero otherwise.

5.2.21.4 See also

spi_cmd_register, **spi_response_set_handler**

5.2.22 spi_component_find

5.2.22.1 Prototype

```
#include "spi_spm.h"

spi_component_t
spi_component_find(
        const char *      name,
        spi_provider_t    provider,
        const char *      min_version,
        const char *      max_version
        );
```

5.2.22.2 Description

spi_component_find finds a Stream programming model component with the given *name* and *provider* and a version number that falls into the range specified by the given *min_version* and *max_version*. If **spi_component_find** finds multiple versions of the requested component, it returns the one with the latest version that matches the given criteria. *min_version* and *max_version* are strings in format "[<>]*w.x.y.z*", where *w* is the major version, *x* is the minor version, *y* is the patch version, and *z* is the local version. The minimum version is inclusive unless min_version begins with '>'. Similarly, the maximum version is inclusive unless max_version begins with '<'. A minimum or maximum version of **NULL** indicates that there is no requirement on the minimum or maximum.

A component may be assigned to multiple schedule groups in an image with **spi_schedgroup_register_component**, and a component also may be used in multiple images. **spi_component_find** searches all scheduling groups in the system to find a component, so it will fail if it finds the same component in multiple scheduling groups. In this case, the program should instead use **spi_schedgroup_component_find** to find the component in a specified scheduling group.

SPI_COMPONENT_NEW defines a component, specifying its name, provider, and version number.

5.2.22.3 Return value

spi_component_find returns a component that matches the name, provider, and version criteria on success. It returns **NULL** on failure.

5.2.22.4 See also

SPI_COMPONENT_NEW, **spi_provider_t**, **spi_schedgroup_component_find**, **spi_schedgroup_register_component**

5.2.23 spi_component_get_desc

5.2.23.1 Prototype

#include "spi_spm.h"

const char *
spi_component_get_desc(spi_component_t *component***);**

5.2.23.2 Description

spi_component_get_desc gets the description of the given Stream programming model *component*.

Macro **SPI_COMPONENT_NEW** defines a component, speciifying a description.

5.2.23.3 Return value

spi_component_get_desc returns the description of *component*. The returned string should not be modified.

5.2.23.4 See also

spi_component_get_name, **SPI_COMPONENT_NEW**

5.2.24 spi_component_get_name

5.2.24.1 Prototype

#include "spi_spm.h"

const char *
spi_component_get_name(spi_component_t *component***);**

5.2.24.2 Description

spi_component_get_name gets the name of the given Stream programming model *component*.

Macro **SPI_COMPONENT_NEW** defines a component, specifying a name.

5.2.24.3 Return value

spi_component_get_name returns the *component* name. The returned string should not be modified.

5.2.24.4 See also

spi_component_get_desc, **SPI_COMPONENT_NEW**

5.2.25 spi_component_get_provider

5.2.25.1 Prototype

#include "spi_spm.h"

spi_provider_t
spi_component_get_provider(spi_component_t *component***);**

5.2.25.2 Description

spi_component_get_provider gets the provider of the given Stream programming model *component*.

SPI_COMPONENT_NEW defines a component, specifying its provider.

5.2.25.3 Return value

spi_component_get_provider returns the provider of *component*.

5.2.25.4 See also

SPI_COMPONENT_NEW, **spi_provider_t**

5.2.26 spi_component_get_version

5.2.26.1 Prototype

#include "spi_spm.h"

const char *
spi_component_get_version(spi_component_t *component***);**

5.2.26.2 Description

spi_component_get_version gets the version of the given Stream programming model *component* as a string. The returned string should be freed by the caller when no longer needed.

5.2.26.3 Return value

spi_component_get_version returns a string giving the version version of *component*.

5.2.26.4 See also

SPI_COMPONENT_NEW

5.2.27 SPI_COMPONENT_NEW

5.2.27.1 Prototype

#include "spi_spm.h"

```
SPI_COMPONENT_NEW(
        const char *                                id,
        const char *                                name,
        spi_provider_t                              provider,
        const char *                                description,
        const char *                                version,
        spi_component_properties_fn_t               prop_fn,
        spi_component_instance_init_fn_t            init_fn,
        spi_component_instance_destroy_fn_t         dest_fn,
        spi_component_instance_execute_fn_t         exec_fn,
        spi_component_instance_cmdhandler_fn_t      cmdh_fn
        );
```

5.2.27.2 Description

Macro **SPI_COMPONENT_NEW** defines a component with the given *id*, *name*, *provider*, *description*, and *version*. The version should be in the format "*w.x.y.z*", where *w* is the major version, *x* is the minor version, *y* is the patch version, and *z* is the local version. **SPI_COMPONENT_NEW** must be used as a declaration with file scope; it should not be invoked as a function call.

Five function arguments define the operation of the component:

- Properties function *prop_fn* sets the properties of the component.
  - If **NULL**, component properties assume default values.
- Initialization function *init_fn* initializes an instance of the component.
  - If **NULL**, no instance initialization is performed, and the custom context for each instance will be NULL.
- Destructor function *dest_fn* destroys an instance of the component.
  - If **NULL**, no function is invoked when an instance is destroyed.
- Execution function *exec_fn* executes an instance of the component.
  - If **NULL**, no function is invoked to execute the instance.
- Command handler function *cmdh_fn* handles commands sent to a component instance.
  - If **NULL**, no function is invoked when an instance receives a command.

A component properties function executes once, when the Stream programming model runtime begins execution; it can set component properties and resource requirements and register commands, ports, and execution requirements that apply to all instances of the component. A component instance initialization function executes each time **spi_instance_new** creates a new component instance. A component execution function executes when the component is running and specified execution properties are met; for example, a component might begin execution when input data is available on its input port and space is available on its output port. A component command handler handles component-specific commands.

5.2.27.3 Return value

None.

5.2.27.4 See also

spi_component_get_desc, **spi_component_get_name**, **spi_component_instance_cmdhandler_fn_t, spi_component_instance_destroy_fn_t, spi_component_instance_execute_fn_t, spi_component_instance_init_fn_t, spi_component_properties_fn_t**, **spi_instance_new**

5.2.28 spi_component_set_flags

5.2.28.1 Prototype

#include "spi_spm.h"

void
spi_component_set_flags(spi_component_flags_t *flags***);**

5.2.28.2 Description

spi_component_set_flags sets the flags for a component to the given *flags* value. It may be called only from the component properties function; calling **spi_component_set_flags** from any other context causes a fatal error.

5.2.28.3 Return value

None.

5.2.28.4 See also

SPI_COMPONENT_NEW, **spi_component_properties_fn_t**

5.2.29 spi_component_set_resource_requirements

5.2.29.1 Prototype

#include "spi_spm.h"

void
spi_component_set_resource_requirements(spi_resources_t *resources***);**

5.2.29.2 Description

spi_component_set_resource_requirements sets the resources required for a component to the given *resources* value. It may be called only from the component properties function; calling **spi_component_set_resource_requirements** from any other context causes a fatal error.

If a component requires the DPU, it must call **spi_component_set_resource_requirements(SPI_RESOURCE_DPU)** from its properties function.

5.2.29.3 Return value

None.

5.2.29.4 See also

spi_component_properties_fn_t, **spi_resources_t**

5.2.30 spi_connect

5.2.30.1 Prototype

```
#include "spi_spm.h"

spi_connection_t
spi_connect(
        const char *      name,
        spi_instance_t    from_instance,
        unsigned int      from_port_id,
        spi_instance_t    to_instance,
        unsigned int      to_port_id,
        unsigned int      depth
        );
```

5.2.30.2 Description

spi_connect creates a new connection with the given *name* between two previously created component instances. It connects output port *from_port_id* of component instance *from_instance* to input port *to_port_id* of component instance *to_instance* with a FIFO of the given *depth*.

spi_connection_new creates a new connection to a contained component instance.

5.2.30.3 Return value

spi_connect returns a new connection on success or **NULL** on failure.

5.2.30.4 See also

spi_connection_new, **spi_instance_t**

5.2.31 spi_connection_get_depth

5.2.31.1 Prototype

#include "spi_spm.h"

unsigned int
spi_connection_get_depth(spi_connection_t *connection***);**

5.2.31.2 Description

spi_connection_get_depth gets the maximum number of buffers that can be held in the FIFO represented by the *connection*, called its depth.

spi_connect and **spi_connection_new** define a connection, specifying its depth.

5.2.31.3 Return value

spi_connection_get_depth returns the maximum number of buffers that can be held by *connection*.

5.2.31.4 See also

spi_connect, **spi_connection_new**

5.2.32 spi_connection_get_name

5.2.32.1 Prototype

#include "spi_spm.h"

const char *
spi_connection_get_name(spi_connection_t *connection***);**

5.2.32.2 Description

spi_connection_get_name gets the name of the given *connection.* **spi_connect** and **spi_connection_new** specify a connection name when they create a connection.

5.2.32.3 Return value

spi_connection_get_name returns the *connection* name. The returned string should not be modified in place.

5.2.32.4 See also

spi_connect, **spi_connection_new**

5.2.33 spi_connection_is_empty

5.2.33.1 Prototype

#include "spi_spm.h"

int
spi_connection_is_empty(spi_connection_t *connection***);**

5.2.33.2 Description

spi_connection_is_empty determines whether the given *connection* is empty.

5.2.33.3 Return value

spi_connection_is_empty returns non-zero if *connection* is empty.

5.2.33.4 See also

spi_connection_is_full

5.2.34 spi_connection_is_full

5.2.34.1 Prototype

#include "spi_spm.h"

int
spi_connection_is_full(spi_connection_t *connection***);**

5.2.34.2 Description

spi_connection_is_full determines whether the given *connection* is full.

5.2.34.3 Return value

spi_connection_is_full returns non-zero if *connection* is full

5.2.34.4 See also

spi_connection_is_empty

5.2.35 spi_connection_new

5.2.35.1 Prototype

```
#include "spi_spm.h"

spi_connection_t
spi_connection_new(
        const char *                name,
        spi_instance_t              instance,
        unsigned int                port,
        unsigned int                depth,
        spi_connection_flags_t      flags
        );
```

5.2.35.2 Description

spi_connection_new creates a new connection with the given *name* to a *port* on a contained component *instance*. The new connection consists of a FIFO with up to *depth* buffers. *flags* indicates the direction of the connection. If the connection is incoming (if **SPI_CONNECTION_FLAG_INCOMING**), buffers can be popped from the connection. If the connection is outgoing (if **SPI_CONNECTION_FLAG_OUTGOING**), buffers can be pushed onto the connection.

spi_connect creates a new connection between two existing component instances.

5.2.35.3 Return value

spi_connection_new returns a new connection on success or **NULL** on failure.

5.2.35.4 See also

spi_connect, **spi_connection_flags_t**, **spi_instance_t**

5.2.36 spi_connection_pop

5.2.36.1 Prototype

```
#include "spi_spm.h"

spi_buffer_t
spi_connection_pop(
        spi_connection_t        connection,
        int                     timeout
        );
```

5.2.36.2 Description

spi_connection_pop pops a buffer from the given *connection*. A component instance acquires ownership of a buffer when it pops it from a connection.

Parameter *timeout* controls the behavior of the pop. If *timeout* is negative, **spi_connection_pop** does not return until the pop succeeds. If *timeout* is zero, **spi_connection_pop** fails if it cannot perform the pop immediately. Otherwise, **spi_connection_pop** fails if it cannot perform the pop within *timeout* milliseconds.

5.2.36.3 Return value

spi_connection_pop returns the popped buffer on success or **NULL** on failure.

5.2.36.4 See also

spi_connection_push, **spi_connection_t**

5.2.37 spi_connection_push

5.2.37.1 Prototype

```
#include "spi_spm.h"

int
spi_connection_push(
        spi_connection_t        connection,
        spi_buffer_t            buffer,
        int                     timeout
        );
```

5.2.37.2 Description

spi_connection_push pushes the given *buffer* onto the given *connection*. A component instance releases ownership of a buffer when it pushes it to a connection.

Parameter *timeout* controls the behavior of the push. If *timeout* is negative, **spi_connection_push** does not return until the push succeeds. If *timeout* is zero, **spi_connection_push** fails if it cannot perform the push immediately. Otherwise, **spi_connection_push** fails if it cannot perform the push within *timeout* milliseconds.

5.2.37.3 Return value

spi_connection_push returns zero if the push succeeds, non-zero on failure.

5.2.37.4 See also

spi_connection_pop, **spi_connection_t**

5.2.38 spi_exec_req_activate

5.2.38.1 Prototype

#include "spi_spm.h"

int
spi_exec_req_activate(unsigned int id**);**

5.2.38.2 Description

spi_exec_req_activate activates the execution requirement with the given *id*. By default, the execute function of an instance is called only when *all* its registered execution requirements are satisfied. **spi_exec_req_activate** specifies that a single execution requirement controls invocation of the instance's execute function instead.

5.2.38.3 Return value

spi_exec_req_activate returns zero on success or non-zero on failure.

5.2.38.4 See also

spi_exec_req_delete, **spi_exec_req_register**

5.2.39 spi_exec_req_delete

5.2.39.1 Prototype

#include "spi_spm.h"

int
spi_exec_req_delete(**unsigned int** *id*)**;**

5.2.39.2 Description

spi_exec_req_delete deletes the execution requirement with the given *id*.

5.2.39.3 Return value

spi_exec_req_delete returns zero if the execution requirement is deleted, non-zero on failure.

5.2.39.4 See also

spi_exec_req_register

5.2.40 spi_exec_req_is_satisfied

5.2.40.1 Prototype

#include "spi_spm.h"

int
spi_exec_req_is_satisfied(unsigned int *id***);**

5.2.40.2 Description

spi_exec_req_is_satisfied determines whether the execution requirement with the given *id* is satisfied. This function should be called only from a component's execute function, before it calls any other SPM functions. Its result may be inaccurate if it is invoked outside the component's execute function or after other SPM functions have been called.

5.2.40.3 Return value

spi_exec_req_is_satisfied returns non-zero (true) if the execution requirement is satisfied, zero (false) if not.

5.2.40.4 See also

spi_exec_req_activate, spi**_exec_req_register**

5.2.41 spi_exec_req_register

5.2.41.1 Prototype

```
#include "spi_spm.h"

int
spi_exec_req_register(
        unsigned int                    id,
        const char *                    description,
        spi_execution_requirement_t     req_type,
        unsigned int                    req_count,
        ...);
```

5.2.41.2 Description

spi_exec_req_register defines an execution requirement with the given numerical *id* and *description* for a component instance. Execution requirements control when a component instance's execute function (specified by the **spi_component_instance_execute_fn_t** parameter in the **SPI_COMPONENT_NEW** component definition) is called. **spi_exec_req_register** may be called multiple times to create multiple execution requirements. By default, an instance's execute function is invoked only when *all* its execution requirements are satisfied (but: see also **spi_exec_req_activate**). **spi_exec_req_is_satisfied** determines whether a given part of a complex execution requirement is satisfied.

The variable argument portion of the argument list depends on the *req_type*. The available requirement types are:

- **SPI_EXEC_ALWAYS** is always satisfied. *req_count* must be zero and other parameters are ignored.
- **SPI_EXEC_NEVER** is never satisfied. *req_count* must be zero and other parameters are ignored.
- **SPI_EXEC_ALLOF** specifies that all of a set of *req_count* additional requirements must be satisfied. Additional arguments after *req_count* give the IDs of the additional requirements.
- **SPI_EXEC_ANYOF** specifies that one or more of a set of *req_count* additional requirements must be satisfied. Additional arguments after *req_count* give the IDs of the additional requirements.
- **SPI_EXEC_PORT_ALLOF** specifies that a buffer must be available on all connections attached to a set of input ports, or that a free entry must be available on all connections attached to a set of output ports. Parameter *req_count* gives the number of port IDs specified in the additional arguments. Additional arguments after *req_count* give the port IDs.
- **SPI_EXEC_PORT_ANYOF** specifies that a buffer must be available on any one of the connections attached to a set of input ports, or that a free entry must be available on any one of the connections attached to a set of output ports. Parameter *req_count* gives the number of port IDs specified in the additional arguments. Additional arguments after *req_count* give the port IDs.
- **SPI_EXEC_FD_READ** specifies that all of a set of *req_count* file descriptors must be ready for reading. Additional arguments after *req_count* give the file descriptors.

- **SPI_EXEC_FD_WRITE** specifies that all of a set of *req_count* file descriptors must be ready for writing. Additional arguments after *req_count* give the file descriptors.

- **SPI_EXEC_POOL** specifies that each of a set of *req_count* buffer pools must have an available buffer. Additional arguments after *req_count* give the pool IDs.

5.2.41.3 Return value

spi_exec_req_register returns zero on success or non-zero if it is unable to register the execution requirement.

5.2.41.4 See also

spi_exec_req_activate, **spi_exec_req_delete**, **spi_exec_req_is_satisfied**,**spi_execution_requirement_t**

5.2.42 spi_fb_get_line_length

5.2.42.1 Prototype

#include "spi_spm.h"

unsigned int
spi_fb_get_line_length(unsigned int *idx***);** **/*** in bytes */

5.2.42.2 Description

spi_fb_get_line_length gets the line length in bytes of a framebuffer in the framebuffer pool with the given index *idx*.

spi_fb_pool_new creates a framebuffer pool with a given index.

5.2.42.3 Return value

spi_fb_get_line_length returns the line length in bytes of a framebuffer in the framebuffer pool with the given index *idx*.

5.2.42.4 See also

spi_fb_get_xres, **spi_fb_get_yres**, **spi_fb_pool_new**

5.2.43 spi_fb_get_pixel_type

5.2.43.1 Prototype

#include "spi_spm.h"

spi_fb_pixel_type_t
spi_fb_get_pixel_type(unsigned int *idx***);**

5.2.43.2 Description

spi_fb_get_pixel_type gets the pixel type of a framebuffer in the framebuffer pool with the given index *idx*.

spi_fb_pool_new creates a framebuffer pool with a given index. **spi_fb_get_line_length**, **spi_fb_get_xres**, and **spi_fb_get_yres** return respectively the line length, the horizontal resolution, and the vertical resolution of a framebuffer.

5.2.43.3 Return value

spi_get_fb_pixel_type returns the pixel type of a framebuffer in the framebuffer pool with the given index *idx*.

5.2.43.4 See also

spi_fb_get_line_length, **spi_fb_get_xres**, **spi_fb_get_yres**, **spi_fb_pixel_type_t**, **spi_fb_pool_new**

5.2.44 spi_fb_get_xres

5.2.44.1 Prototype

#include "spi_spm.h"

unsigned int
spi_fb_get_xres(unsigned int *idx***);** /***** in pixels */

5.2.44.2 Description

spi_fb_get_xres gets the horizontal (X) resolution in pixels of a framebuffer in the framebuffer pool with the given index *idx*.

spi_fb_pool_new creates a framebuffer pool with a given index. **spi_fb_get_yres** gets the vertical (Y) resolution of a framebuffer.

5.2.44.3 Return value

spi_fb_get_xres returns the horizontal (X) resolution in pixels of a framebuffer in the framebuffer pool with the given index *idx*.

5.2.44.4 See also

spi_fb_get_line_length, **spi_fb_get_yres**, **spi_fb_pool_new**

5.2.45 spi_fb_get_yres

5.2.45.1 Prototype

#include "spi_spm.h"

unsigned int
spi_fb_get_yres(unsigned int *idx***);** /***** in pixels */

5.2.45.2 Description

spi_fb_get_yres gets the vertical (Y) resolution in pixels of a framebuffer in the framebuffer pool with the given index *idx*.

spi_fb_pool_new creates a framebuffer pool with a given index. **spi_fb_get_xres** gets the horizontal (X) resolution of a framebuffer.

5.2.45.3 Return value

spi_fb_get_yres returns the vertical (Y) resolution in pixels of a framebuffer in the framebuffer pool with the given index *idx*.

5.2.45.4 See also

spi_fb_get_line_length, **spi_fb_get_xres**, **spi_fb_pool_new**

5.2.46 spi_fb_is_available

5.2.46.1 Prototype

#include "spi_spm.h"

int
spi_fb_is_available(unsigned int *idx***);**

5.2.46.2 Description

spi_fb_is_available determines whether a framebuffer is available from the framebuffer pool with the given index *idx*.

spi_fb_pool_new creates a framebuffer buffer pool with a given index.

5.2.46.3 Return value

spi_fb_is_available returns nonzero if a framebuffer is available from the framebuffer pool with the given index *idx*.

5.2.46.4 See also

spi_fb_pool_new

5.2.47 spi_fb_pool_new

5.2.47.1 Prototype

```
#include "spi_spm.h"

spi_pool_t
spi_fb_pool_new(
        const char *        name,                   /* name */
        const char *        desc,                   /* description */
        unsigned int        fb_idx,                 /* index */
        unsigned int        initial_buffer_count,   /* initial buffer count */
        spi_pool_flags_t    flags                   /* flags */
);
```

5.2.47.2 Description

spi_fb_pool_new creates a buffer pool for buffers that represent framebuffers. It creates a new pool with the given *name*, description *desc*, and index *idx*. The pool initially contains *inital_buffer_count* buffers. The given *flags* should be either **SPI_POOL_FLAG_NONE** or **SPI_POOL_FLAG_GROW**; the pool will grow as needed if *flags* includes **SPI_POOL_FLAG_GROW**.

5.2.47.3 Return value

spi_fb_pool_new returns a new pool on success or **NULL** on failure.

5.2.47.4 See also

spi_pool_flags_t, **spi_pool_new**, **spi_pool_t**

5.2.48 spi_get_buffer_heap_highwater

5.2.48.1 Prototype

#include "spi_spm.h"

unsigned long long
spi_get_buffer_heap_highwater(void);

5.2.48.2 Description

spi_get_buffer_heap_highwater gets the total number of bytes currently used by the shared memory heap for buffer allocation. Due to memory fragmentation, not all of this memory is necessarily in current use by buffers. The buffer allocator attempts to reuse contiguous blocks of unused memory to avoid raising the highwater mark unnecessarily.

spi_get_buffer_heap_size gets the size of the shared memory heap.

5.2.48.3 Return value

spi_get_buffer_heap_highwater returns the total number of bytes currently used by the shared memory heap used for buffer allocation.

5.2.48.4 See also

spi_get_buffer_heap_size

5.2.49 spi_get_buffer_heap_size

5.2.49.1 Prototype

#include "spi_spm.h"

unsigned long long
spi_get_buffer_heap_size(void);

5.2.49.2 Description

spi_get_buffer_heap_size gets the size of the shared memory heap used for buffer allocation.

spi_get_buffer_heap_highwater gets the number of bytes currently in use by the shared memory heap.

5.2.49.3 Return value

spi_get_buffer_heap_size returns the size of the shared memory heap.

5.2.49.4 See also

spi_get_buffer_heap_highwater

5.2.50 spi_get_component

5.2.50.1 Prototype

#include "spi_spm.h"

spi_component_t
spi_get_component(void);

5.2.50.2 Description

spi_get_component gets the component used to instantiate the current instance.

spi_instance_new creates a component instance.

5.2.50.3 Return value

spi_get_component returns the component that instantiated the current instance.

5.2.50.4 See also

spi_get_name, **spi_get_state**, **spi_instance_new**

5.2.51 spi_get_connection

5.2.51.1 Prototype

#include "spi_spm.h"

spi_connection_t
spi_get_connection(const char ****name*);**

5.2.51.2 Description

spi_get_connection gets a connection between the active instance and a contained instance by name.

spi_connection_new creates a connection.

5.2.51.3 Return value

spi_get_connection returns the connection with the given *name*, or **NULL** if it does not exist..

5.2.51.4 See also

spi_connection_new

5.2.52 spi_get_instance

5.2.52.1 Prototype

#include "spi_spm.h"

spi_instance_t
spi_get_instance(const char ****name*);**

5.2.52.2 Description

spi_get_instance gets the instance with the given *name*. It is used to obtain an instance handle after an initialization file creates instances. It applies only within the context in which an instance was created.

spi_instance_new creates a component instance.

5.2.52.3 Return value

spi_get_instance returns the instance with the given *name*, or **NULL** if it does not exist..

5.2.52.4 See also

spi_instance_new

5.2.53 spi_get_log

5.2.53.1 Prototype

#include "spi_spm.h"

spi_log_t
spi_get_log(const char * *name***);**

5.2.53.2 Description

spi_get_log gets the log with the given *name*.

spi_log_new creates a log with a specified name.

5.2.53.3 Return value

spi_get_log returns the log with the given *name*, or **NULL** if the named log does not exist.

5.2.53.4 See also

spi_log_new

5.2.54 spi_get_name

5.2.54.1 Prototype

#include "spi_spm.h"

const char *
spi_get_name(void);

5.2.54.2 Description

spi_get_name gets the name of the current instance.

spi_instance_new creates a component instance with a specified name.

5.2.54.3 Return value

spi_get_name returns the name of the current instance. The returned string should not be modified.

5.2.54.4 See also

spi_get_component, **spi_get_state**, **spi_instance_new**

5.2.55 spi_get_pool

5.2.55.1 Prototype

#include "spi_spm.h"

spi_pool_t
spi_get_pool(const char * *name***);**

5.2.55.2 Description

spi_get_pool gets the buffer pool with the given *name*.

spi_pool_new creates a pool with a specified name.

5.2.55.3 Return value

spi_get_pool returns the pool with the given *name*, or **NULL** if the named pool does not exist.

5.2.55.4 See also

spi_pool_new

5.2.56 spi_get_priority

5.2.56.1 Prototype

#include "spi_spm.h"

int
spi_get_priority(void);

5.2.56.2 Description

spi_get_priority gets the scheduling priority of the current instance.

spi_set_priority sets the instance priority.

5.2.56.3 Return value

spi_get_priority returns the current instance priority (from highest priority 0 to lowest priority 15), or -1 on error.

5.2.56.4 See also

spi_set_priority

5.2.57 spi_get_state

5.2.57.1 Prototype

#include "spi_spm.h"

spi_instance_state_t
spi_get_state(void);

5.2.57.2 Description

spi_get_state gets the state of the current instance (stopped, paused, or running).

5.2.57.3 Return value

spi_get_state returns the state of the current instance, or **SPI_INSTANCE_STATE_UNKNOWN** if it is unable to determine the state.

5.2.57.4 See also

spi_get_name, **spi_instance_state_t**, **spi_set_state**

5.2.58 spi_get_time

5.2.58.1 Prototype

#include "spi_spm.h"

unsigned long long
spi_get_time(void);

5.2.58.2 Description

spi_get_time gets the current system time. The current system time is the number of nanosecond ticks since the last system reset.

5.2.58.3 Return value

spi_get_time returns the current system time.

5.2.58.4 See also

5.2.59 spi_get_timer

5.2.59.1 Prototype

#include "spi_spm.h"

spi_timer_t
spi_get_timer(const char * *name***);**

5.2.59.2 Description

spi_get_timer gets the timer with the given *name*.

5.2.59.3 Return value

spi_get_timer returns the timer with the given *name*, or **NULL** if the named timer does not exist.

spi_timer_new creates a timer with a given name and description.

The Stream programming model includes built-in timers with the following names:

- **SPI_TIMER_CMDHANDLER** measures the time spent in the command handler function of a component.
- **SPI_TIMER_DSU** measures the time that the DSU is active (Storm-2).
- **SPI_TIMER_EXECUTE** measures the time spent in the execute function of a component.
- **SPI_TIMER_KERNEL** measures the time required to execute the last invoked kernel. For this timer, **spi_timer_start** and **spi_timer_stop** have no effect, the return value of **spi_timer_get_start_count** is undefined, and **spi_timer_get_total_nanoseconds** returns the same value as **spi_timer_get_nanoseconds**. To use **SPI_TIMER_KERNEL,** a program should call **spi_barrier** before a kernel call (to assure the completion of previous kernels), then call the kernel, then call **spi_timer_get_nanoseconds** or **spi_timer_get_total_nanoseconds** to get the timer value; the program does not need a second **spi_barrier** call before getting the time.
- **SPI_TIMER_LOAD_DSP** measures the time required to load a DSP MIPS image using **spi_load_image**(**SPI_PEL_DSP_MIPS**, ...).
- **SPI_TIMER_LOAD_DSU** measures the time required to load a DSU image using **spi_load_image**(**SPI_PEL_DSU_MIPS**, ...) (Storm-2).
- **SPI_TIMER_ME** measures the time that the ME is active (Storm-2).
- **SPI_TIMER_SPM** starts when the Stream programming model starts. The Stream programming model runtime never stops this timer, so a program can use it to measure elapsed time since the runtime started.
- **SPI_TIMER_STARTUP** measures the startup time of the Stream programming model runtime.

5.2.59.4 See also

spi_barrier, **spi_timer_get_nanoseconds**, **spi_timer_get_start_count**, **spi_timer_get_total_nanoseconds**, **spi_timer_new**, **spi_timer_start**, **spi_timer_stop**

5.2.60 spi_instance_new

5.2.60.1 Prototype

```
#include "spi_spm.h"

spi_instance_t
spi_instance_new(
        const char *            name,
        spi_component_t         component
        );
```

5.2.60.2 Description

spi_instance_new creates a new instance of the given *component*. It gives the new instance the supplied *name*. It assigns the new instance a default priority of 8. The instance starts in the paused state **SPI_INSTANCE_STATE_PAUSED**.

5.2.60.3 Return value

spi_instance_new returns the new component instance on success or **NULL** on failure.

5.2.60.4 See also

SPI_NEW_COMPONENT

5.2.61 spi_load_image

5.2.61.1 Prototype

```
#include "spi_spm.h"

int
spi_load_image(
        spi_pel_t          pel,
        const char *       pathname,
        char               *argv[],
        char               *envp[],
        spi_image_flags_t flags
        );
```

5.2.61.2 Description

spi_load_image loads an image at the given *pathname* onto processing element *pel* while observing the given *flags*. The loaded image starts execution immediately with arguments and environment given by *argv* and *envp*. Like **spi_spm_start**, **spi_load_image** looks for special SPM options in the argument list and removes any special SPM options it finds.

By default, the standard output and standard error of the loaded image are redirected to **stdout** and **stderr** of the application that invoked **spi_load_image**.

5.2.61.3 Return value

spi_load_image returns zero on success, non-zero for failure.

5.2.61.4 See also

Special SPM options, **spi_image_flags_t**, **spi_pel_t**, **spi_spm_start**

5.2.62 spi_log

5.2.62.1 Prototype

```
#include "spi_spm.h"

void
spi_log(
        spi_log_t       log,
        unsigned int    level,
        const char *    fmt,
        ...
        );
```

5.2.62.2 Description

spi_log logs a message at the given *level* to the given *log* with **printf**-style formatting, using *fmt* to format trailing arguments. The *level* is a bit mask; the message is logged only if the log enable mask matches one of the specified bits.

SPI_LOG_DEBUG is the name of the predefined debug log. It supports the following debug log levels:

- **SPI_LOG_LEVEL_DEBUG** (1 << 0) generic debug logging; log failed SPM calls
- **SPI_LOG_LEVEL_DEBUG_EXECUTION** (1 << 1) log SPM event execution
- **SPI_LOG_LEVEL_DEBUG_SCHEDULER** (1 << 2) log SPM component scheduler activity
- **SPI_LOG_LEVEL_DEBUG_MEMORY** (1 << 3) log SPM buffer heap allocation
- **SPI_LOG_LEVEL_DEBUG_KERNEL** (1 << 4) kernel logging
- **SPI_LOG_LEVEL_DEBUG_INTERNAL** (1 << 31) internal SPM runtime debug logging

SPI_LOG_ERROR is the name of the predefined error log. It supports the following error log levels:

- **SPI_LOG_LEVEL_ERROR_FATAL** (1 << 0) fatal error logging
- **SPI_LOG_LEVEL_ERROR_ASSERT** (1 << 1) assertion logging

spi_log_new defines a log. **spi_log_set_enable_mask** sets the enable mask of a log.

By default, the SPM runtime disables all debug log levels, enables all error log levels, and intermixes timestamped output from all logs on **stdout**. **spi_spm_start** recognizes special SPM command-line options that control log behavior: **--spi_log_dir=***dir* specifies a log file directory, **--spi_log_mask=***log,mask* specifies an enable mask for a log, and **--spi_log_timestamps=**[0|1] disables or enables log entry timestamps.

5.2.62.3 Return value

None.

5.2.62.4 See also

spi_log_set_enable_mask, **spi_log_new**, **spi_spm_start**

5.2.63 spi_log_get_desc

5.2.63.1 Prototype

#include "spi_spm.h"

const char *
spi_log_get_desc(spi_log_t *log***);**

5.2.63.2 Description

spi_log_get_desc gets the description of the given *log*.

spi_log_new defines a log with a specified desciption.

5.2.63.3 Return value

spi_log_get_desc returns the description of *log*. The returned string should not be modified.

5.2.63.4 See also

spi_log, **spi_log_new**

5.2.64 spi_log_get_enable_mask

5.2.64.1 Prototype

#include "spi_spm.h"

unsigned int
spi_log_get_enable_mask(spi_log_t *log***);**

5.2.64.2 Description

spi_log_get_enable_mask gets the enable mask for the given *log*. The enable mask is a bit mask; each bit indicates whether the corresponding log level is enabled for the log. If a log level is not enabled, then messages of that level are not logged.

spi_log_set_enable_mask sets the enable mask for a log.

5.2.64.3 Return value

spi_log_get_enable_mask returns the 32-bit enable mask for *log*.

5.2.64.4 See also

spi_log_set_enable_mask

5.2.65 spi_log_get_name

5.2.65.1 Prototype

#include "spi_spm.h"

const char *
spi_log_get_name(spi_log_t *log***);**

5.2.65.2 Description

spi_log_get_name gets the name of the given *log*.

spi_log_new defines a log with a specified name.

5.2.65.3 Return value

spi_log_get_name returns the name of *log*. The returned string should not be modified.

5.2.65.4 See also

spi_log, **spi_log_new**

5.2.66 spi_log_new

5.2.66.1 Prototype

#include "spi_spm.h"

```
spi_log_t
spi_log_new(
        const char *    name,
        const char *    description
        );
```

5.2.66.2 Description

spi_log_new defines a log for a component. The parameters define the *name* and *description* of the log. The enable mask of the new log is initially 0.

spi_log_get_desc and **spi_log_get_name** return the name and description of a log. **spi_log_set_enable_mask** and **spi_log_get_enable_mask** set and get the enable mask of a log. **spi_log** prints a message to a log.

5.2.66.3 Return value

spi_log_new returns a new log on success, or **NULL** if unable to create the log.

5.2.66.4 See also

spi_portdir_t

5.2.67 spi_log_set_enable_mask

5.2.67.1 Prototype

```
#include "spi_spm.h"

int
spi_log_set_enable_mask(
        spi_log_t       log,
        unsigned int    mask
        );
```

5.2.67.2 Description

spi_log_set_enable_mask sets the enable mask for the given *log* to *mask*. The enable mask is a bit mask; each bit indicates whether the corresponding log level is enabled for the log. If a log level is not enabled, then messages of that level are not logged.

spi_log_get_enable_mask gets the enable mask of a log.

5.2.67.3 Return value

spi_log_set_enable_mask returns 0 on success or nonzero on failure.

5.2.67.4 See also

spi_log_get_enable_mask

5.2.68 spi_pool_free

5.2.68.1 Prototype

#include "spi_spm.h"

int
spi_pool_free(spi_pool_t *pool***);**

5.2.68.2 Description

spi_pool_free frees the given buffer *pool*, which should be a pool allocated by **spi_pool_new**. Freeing the pool does not affect buffers from the pool that are currently in use; each region of shared memory controlled by the pool will be released when it is no longer used by a buffer.

5.2.68.3 Return value

spi_pool_free returns a zero on success or nonzero on failure.

5.2.68.4 See also

.
spi_pool_new

5.2.69 spi_pool_get_avail_buffer_count

5.2.69.1 Prototype

#include "spi_spm.h"

unsigned int
spi_pool_get_avail_buffer_count(spi_pool_t *pool***);**

5.2.69.2 Description

spi_pool_get_avail_buffer_count gets the number of buffers currently available from the given buffer *pool*.

spi_pool_get_buffer gets a buffer from a pool.

5.2.69.3 Return value

spi_pool_get_avail_buffer_count returns the number of buffers currently available from the given buffer *pool*.

5.2.69.4 See also

.
spi_pool_get_buffer, **spi_pool_new**

5.2.70 spi_pool_get_buffer

5.2.70.1 Prototype

#include "spi_spm.h"

spi_buffer_t
spi_pool_get_buffer(spi_pool_t *pool***);**

5.2.70.2 Description

spi_pool_get_buffer gets a buffer from the given buffer *pool*. The shared memory block that the buffer represents has the size and alignment specified when **spi_pool_new** created the pool.

5.2.70.3 Return value

spi_pool_get_buffer returns a buffer on success or **NULL** on failure.

5.2.70.4 See also

.
spi_buffer_flags_t, **spi_pool_new**

5.2.71 spi_pool_get_desc

5.2.71.1 Prototype

#include "spi_spm.h"

const char *
spi_pool_get_desc(spi_pool_t *pool***);**

5.2.71.2 Description

spi_pool_get_desc gets the description of the given buffer *pool*.

spi_pool_new creates a buffer pool with a specified description.

5.2.71.3 Return value

spi_pool_get_desc returns the description of the buffer pool. The returned string should not be modified.

5.2.71.4 See also

spi_pool_get_name, **spi_pool_new**

5.2.72 spi_pool_get_name

5.2.72.1 Prototype

#include "spi_spm.h"

const char *
spi_pool_get_name(spi_pool_t *pool***);**

5.2.72.2 Description

spi_pool_get_name gets the name of the given buffer *pool*.

spi_pool_new creates a buffer pool with a name.

5.2.72.3 Return value

spi_pool_get_name returns the name of the buffer pool. The returned string should not be modified.

5.2.72.4 See also

spi_pool_get_desc, **spi_pool_new**

5.2.73 spi_pool_new

5.2.73.1 Prototype

```
#include "spi_spm.h"

spi_pool_t
spi_pool_new(
        const char *        name,
        const char *        description,
        unsigned int        count,
        unsigned int        size,
        unsigned int        alignment,
        spi_pool_flags_t    flags
        );
```

5.2.73.2 Description

spi_pool_new creates a new buffer pool with the given *name* and *description*. The new buffer pool initially contains *count* buffers, each with the given *size* in bytes and given *alignment*. The new buffers will observe the given *flags*, which specify whether the pool may grow dynamically as needed. An *alignment* of 0 specifies the default alignment (cache-line aligned, to a multiple of 32 bytes).

spi_fb_pool_new creates a new buffer pool specifically for framebuffers.

5.2.73.3 Return value

spi_pool_new returns a new pool if it creates the pool successfully, **NULL** otherwise.

spi_pool_get_buffer gets a buffer from a pool. **spi_pool_get_name** and **spi_pool_get_desc** get the name and descirption of a buffer pool. **spi_pool_free** frees a pool.

5.2.73.4 See also

spi_fb_pool_new, **spi_pool_flags_t**, **spi_pool_free**, **spi_pool_get_buffer**, **spi_pool_get_desc**, **spi_pool_get_name**

5.2.74 spi_port_export

5.2.74.1 Prototype

```
#include "spi_spm.h"

int
spi_port_export(
        unsigned int      port,
        spi_instance_t    instance,
        unsigned int      export_port
        );
```

5.2.74.2 Description

spi_port_export exports the given *export_port* on the contained *instance* as *port*. Any connection to *port* is forwarded to *export_port* on the contained *instance*.

5.2.74.3 Return value

spi_port_export returns zero on success, non-zero on failure.

5.2.74.4 See also

spi_instance_t

5.2.75 spi_port_get_connection

5.2.75.1 Prototype

```
#include "spi_spm.h"

spi_connection_t
spi_port_get_connection(
        unsigned int    id,
        unsigned int    index
        );
```

5.2.75.2 Description

spi_port_get_connection gets the connection with the given *index* on the port with the given *id*. Valid indices are from 0 to *n* - 1, where *n* is the current number of connections attached to the port. **spi_port_get_connection_count** returns the current nuimber of connections attached to a port, so a program can iterate through possible *index* values to find connections with a given port *id*.

5.2.75.3 Return value

spi_port_get_connection returns the requested connection on success or **NULL** on failure.

5.2.75.4 See also

spi_port_get_connection_count

5.2.76 spi_port_get_connection_count

5.2.76.1 Prototype

#include "spi_spm.h"

unsigned int
spi_port_get_connection_count(unsigned int *id***);**

5.2.76.2 Description

spi_port_get_connection_count gets the current number of connections on the port with the given *id*.
spi_port_get_max_connection_count returns the maximum number of connections allowed for a port.

5.2.76.3 Return value

spi_port_get_connection_count returns the number of connections currently attached to port *id.*

5.2.76.4 See also

spi_port_get_max_connection_count

5.2.77 spi_port_get_desc

5.2.77.1 Prototype

#include "spi_spm.h"

const char *
spi_port_get_desc(unsigned int *id***);**

5.2.77.2 Description

spi_port_get_desc gets the description of the port with the given *id*.

spi_port_register creates a port with a specified description.

5.2.77.3 Return value

spi_port_get_desc returns the description of port *id*. The returned string should not be modified.

5.2.77.4 See also

spi_port_get_name, **spi_port_register**

5.2.78 spi_port_get_dir

5.2.78.1 Prototype

#include "spi_spm.h"

spi_portdir_t
spi_port_get_dir(unsigned int *id***);**

5.2.78.2 Description

spi_port_get_dir gets the direction of the port with the given *id* in the component instance containing the port.

spi_port_register defines a port for a component with a specified direction.

5.2.78.3 Return value

spi_port_get_dir returns the direction of port *id*.

5.2.78.4 See also

spi_portdir_t, **spi_port_register**

5.2.79 spi_port_get_max_connection_count

5.2.79.1 Prototype

#include "spi_spm.h"

unsigned int
spi_port_get_max_connection_count(unsigned int *id***);**

5.2.79.2 Description

spi_port_get_max_connection_count gets the maximum number of connections allowed on the port with the given *id*.

spi_port_get_connection_count gets the current connection count for a port.

5.2.79.3 Return value

spi_port_get_max_connection_count returns the maximum number of connections allowed on port *id*. A value of zero indicates that the port allows an unlimited number of connections.

5.2.79.4 See also

spi_port_get_connection_count

5.2.80 spi_port_get_name

5.2.80.1 Prototype

#include "spi_spm.h"

const char *
spi_port_get_name(unsigned int *id***);**

5.2.80.2 Description

spi_port_get_name gets the name of the port with the given *id*.

5.2.80.3 Return value

spi_port_get_name returns the name of port *id*. The returned string should not be modified.

spi_port_register creates a port with a specified name.

5.2.80.4 See also

spi_port_get_desc, **spi_port_register**

5.2.81 spi_port_register

5.2.81.1 Prototype

```
#include "spi_spm.h"

int
spi_port_register(
        unsigned int    id,
        const char *    description,
        spi_portdir_t   direction,
        unsigned int    connections
        );
```

5.2.81.2 Description

spi_port_register defines a port for a component. The parameters define the numerical *id*, *description*, *direction*, and maximum number of *connections* allowed on the port. If *connections* is 0, the port allows an unlimited number of connections.

5.2.81.3 Return value

spi_port_register returns zero on success, non-zero on failure..

5.2.81.4 See also

spi_portdir_t

5.2.82 spi_provider_get_name

5.2.82.1 Prototype

#include "spi_spm.h"

const char *
spi_provider_get_name(spi_provider_t *provider***);**

5.2.82.2 Description

spi_provider_get_name gets the name of the given *provider*. A provider is an organization that provides Stream programming model components.

5.2.82.3 Return value

spi_provider_get_name returns the name of the *provider*. The returned string should not be modified.

5.2.82.4 See also

spi_provider_t

5.2.83 spi_response_free

5.2.83.1 Prototype

#include "spi_spm.h"

int
spi_response_free(spi_response_t *response***);**

5.2.83.2 Description

spi_response_free frees the given command *response*, including the response payload.

5.2.83.3 Return value

spi_response_get_errno returns zero on success, non-zero on failure.

5.2.83.4 See also

spi_cmd_send_response

5.2.84 spi_response_get_errno

5.2.84.1 Prototype

#include "spi_spm.h"

spi_response_errno_t
spi_response_get_errno(spi_response_t *response***);**

5.2.84.2 Description

spi_response_get_errno gets the error code of the given command *response*.

spi_cmd_send_response sends a command with a specified error code. **spi_response_strerror** gets the textual desciption of an error code.

5.2.84.3 Return value

spi_response_get_errno returns the error code of the command *response*.

5.2.84.4 See also

spi_cmd_send_response, **spi_response_strerror**

5.2.85 spi_response_get_payload

5.2.85.1 Prototype

#include "spi_spm.h"

void *
spi_response_get_payload(spi_response_t *response***);**

5.2.85.2 Description

spi_response_get_payload gets the payload of the given command *response*. The actual content of the payload depends on the command that corresponds to the *response*. **spi_reponse_get_payload_size** gets the payload size of a response.

spi_cmd_send_response sends a command with a specified payload and payload size.

5.2.85.3 Return value

spi_response_get_payload returns the payload of the command *response*.

5.2.85.4 See also

spi_cmd_send_response, **spi_reponse_get_payload_size**

5.2.86 spi_response_get_payload_size

5.2.86.1 Prototype

#include "spi_spm.h"

unsigned int
spi_response_get_payload_size(spi_response_t *response***);**

5.2.86.2 Description

spi_response_get_payload_size gets the size in bytes of the payload of the given command *response*. A payload size of zero indicates that the response payload is just the payload pointer itself. **SPI_MAX_RESPONSE_PAYLOAD_SIZE** gives the maximum size (in bytes) of a response payload.

spi_reponse_get_payload gets the payload of a response. **spi_cmd_send_response** sends a command with a specified payload and payload size.

5.2.86.3 Return value

spi_reponse_get_payload_size returns the payload size in bytes. If the payload size is zero, the payload may be a pointer or a value that can be passed as a pointer without loss of precision; for example, an integer value may be sent as a response payload by casting it to **void *** and sending it as the payload.

5.2.86.4 See also

spi_cmd_send_response, **spi_reponse_get_payload**

5.2.87 spi_response_get_payload_type

5.2.87.1 Prototype

#include "spi_spm.h"

spi_payload_type_t
spi_response_get_payload_type(spi_response_t *response***);**

5.2.87.2 Description

spi_response_get_payload_type gets the payload type of the given *response*. **spi_response_get_payload** gets the response payload.

spi_send_response sends a command response with a specified payload and payload type.

5.2.87.3 Return value

spi_response_get_payload_type returns the *response* payload type.

5.2.87.4 See also

spi_response_get_payload, **spi_send_response**

5.2.88 spi_response_set_handler

5.2.88.1 Prototype

```
#include "spi_spm.h"

int
spi_response_set_handler(
        spi_response_t                  response,
        spi_response_handler_fn_t       fn,
        spi_response_context_t          context
        );
```

5.2.88.2 Description

spi_response_set_handler sets the handler function for a command *response* to the given *fn*. If *fn* is **NULL**, no function is invoked when the command response is received. If *fn* is not **NULL**, the *context* is passed to the command response handler *fn* when the reponse is received.

When **spi_cmd_send** sends a command, the receiving instance returns a handle immediately. Eventually, the command handler of the receiving instance (specified by the **spi_component_instance_cmdhandler_fn_t** function in the **SPI_COMPONENT_NEW** component definition) finishes processing the command. It then calls **spi_cmd_send_response** to send an actual command response. The reponse handler of the sending instance (specified by **spi_response_set_handler**) handles the response, using the **spi_response_t** response token that **spi_cmd_send** returned to identiry the command.

5.2.88.3 Return value

spi_response_set_handler returns 0 on success, non-zero on failure.

5.2.88.4 See also

spi_cmd_send, spi**_response_context_t**, **spi_response_handler_fn_t**, **spi_response_t**

5.2.89 spi_response_strerror

5.2.89.1 Prototype

#include "spi_spm.h"

const char *
spi_response_strerror(spi_response_errno_t *error_code***);**

5.2.89.2 Description

spi_response_strerror gets the textual description of the given response *error_code*.

spi_cmd_send_response sends a command with a specified error code. **spi_response_errno** gets the error code of a response.

5.2.89.3 Return value

spi_response_strerror returns the textual description of response *error_code*. The returned string should not be modified.

5.2.89.4 See also

spi_cmd_send_response, **spi_response_errno**

5.2.90 spi_schedgroup_component_find

5.2.90.1 Prototype

```
#include "spi_spm.h"

spi_component_t
spi_schedgroup_component_find(
        const char *      schedgroup_name,
        spi_provider_t    schedgroup_provider,
        const char *      component_name,
        spi_provider_t    component_provider,
        const char *      min_version,
        const char *      max_version
        );
```

5.2.90.2 Description

spi_schedgroup_component_find is like **spi_component_find**, but it searches for a component within a single scheduling group rather than across the entire system. A component may be assigned to multiple schedule groups in an image with **spi_schedgroup_register_component**, and a component also may be used in multiple images. **spi_component_find** searches all scheduling groups in the system to find a component, so it will fail if it finds the same component in multiple scheduling groups. In this case, the program should instead use **spi_schedgroup_component_find** to find the component in a specified scheduling group.

spi_schedgroup_component_find finds a Stream programming model component with the given *component_name* and *component_provider* and a version number that falls into the range specified by the given *min_version* and *max_version* within a scheduling group with the given *schedgroup_name* and *schedgroup_provider*. If **spi_schedgroup_component_find** finds multiple versions of the requested component, it returns the one with the latest version that matches the given criteria. *min_version* and *max_version* are strings in format "[<>]*w.x.y.z*", where *w* is the major version, *x* is the minor version, *y* is the patch version, and *z* is the local version. The minimum version is inclusive unless min_version begins with '>'. Similarly, the maximum version is inclusive unless max_version begins with '<'. A minimum or maximum version of **NULL** indicates that there is no requirement on the minimum or maximum.

5.2.90.3 Return value

spi_schedgroup_component_find returns a component in the scheduling group that matches the name, provider, and version criteria on success. It returns **NULL** on failure.

5.2.90.4 See also

spi_component_find, **spi_provider_t**, **spi_schedgroup_register_component**

5.2.91 SPI_SCHEDGROUP_NEW

5.2.91.1 Prototype

#include "spi_spm.h"

```
SPI_SCHEDGROUP_NEW(
        const char *                      name,
        spi_provider_t                    provider,
        const char *                      description,
        spi_schedgroup_properties_fn_t    prop_fn
        );
```

5.2.91.2 Description

Macro **SPI_SCHEDGROUP_NEW** defines a scheduling group with the given *name*, *provider*, *description*, and properties function *prop_fn* If *prop_fn* is **NULL**, then the scheduling group has no properties function; all properties will be set to default values.. **SPI_SCHEDGROUP_NEW** must be used as a declaration with file scope; it should not be invoked as a function call.

Properties function *prop_fn* can call **spi_schedgroup_register_component** to register a component within a scheduling group. It can also call **spi_schedgroup_set_processing_element** and **spi_schedgroup_set_controlled_resources** to set scheduling group processing element and resource requirements.

5.2.91.3 Return value

None.

5.2.91.4 See also

spi_provider_t, **spi_schedgroup_properties_fn_t**, **spi_schedgroup_register_component**, **spi_schedgroup_set_controlled_resources**, **spi_schedgroup_set_processing_element**

5.2.92 spi_schedgroup_register_component

5.2.92.1 Prototype

#include "spi_spm.h"

```
void
spi_schedgroup_register_component(
        const char *      name,
        spi_provider_t    provider,
        const char *      min_version,
        const char *      max_version
        );
```

5.2.92.2 Description

spi_schedgroup_register_component registers a component with a scheduling group. The given parameters specify a *name*, *provider*, and minimum and maximum version numbers of the component. The scheduling group can create instances of a component once the component is registered with a scheduling group.

A component is assigned to the default scheduling group of an image if the program does not explicitly register the component with **spi_schedgroup_register_component**. A component may be assigned to multiple scheduling groups within an image.

If **spi_schedgroup_register_component** finds multiple versions of the requested component, it returns the one with the latest version that matches the given version criteria. *min_version* and *max_version* are strings in format "[<>]*w.x.y.z*", where *w* is the major version, *x* is the minor version, *y* is the patch version, and *z* is the local version. The minimum version is inclusive unless *min_version* begins with '>'. Similarly, the maximum version is inclusive unless *max_version* begins with '<'. A minimum or maximum version of **NULL** indicates that there is no requirement on the minimum or maximum.

5.2.92.3 Return value

None. A fatal error occurs if the requested component is not available.

5.2.92.4 See also

spi_provider_t

5.2.93 spi_schedgroup_set_controlled_resources

5.2.93.1 Prototype

#include "spi_spm.h"

void
spi_schedgroup_set_controlled_resources(spi_resources_t *resources***);**

5.2.93.2 Description

spi_schedgroup_set_controlled_resources defines the *resources* that a scheduling group requires during execution. It may be called only from the scheduling group properties function; calling it in any other context causes a fatal error.

5.2.93.3 Return value

None.

5.2.93.4 See also

spi_resources_t

5.2.94 spi_schedgroup_set_min_stacksize

5.2.94.1 Prototype

#include "spi_spm.h"

void
spi_schedgroup_set_min_stacksize(unsigned int *size***);**

5.2.94.2 Description

spi_schedgroup_set_min_stacksize sets the minimum stack size required for a scheduling group to the given *size* (in bytes). It may be called only from the scheduling group properties function; calling it in any other context causes a fatal error.

5.2.94.3 Return value

None.

5.2.94.4 See also

spi_resources_t

5.2.95 spi_schedgroup_set_processing_elements

5.2.95.1 Prototype

#include "spi_spm.h"

void
spi_schedgroup_set_processing_elements(spi_pels_t *pels***);**

5.2.95.2 Description

spi_schedgroup_set_processing_elements defines the processing elements *pels* on which the scheduling group is allowed to execute. It may be called only from the scheduling group properties function; calling it in any other context causes a fatal error.

By default, a processing group can execute on any processing element (**SPI_PEL_ALL**).

5.2.95.3 Return value

None.

5.2.95.4 See also

spi_pels_t

5.2.96 spi_set_priority

5.2.96.1 Prototype

#include "spi_spm.h"

int
spi_set_priority(unsigned int *priority***);**

5.2.96.2 Description

spi_set_priority sets the scheduling priority of the current instance to the given *priority*. Priority 0 is the highest priority level, priority 15 is the lowest. The default priority for an instance is 8.

spi_get_priority gets the current instance priority.

5.2.96.3 Return value

spi_set_priority returns zero if it changes the priority as requested, non-zero on failure.

5.2.96.4 See also

spi_get_priority

5.2.97 spi_set_state

5.2.97.1 Prototype

#include "spi_spm.h"

int
spi_set_state(spi_instance_state_t *state***);**

5.2.97.2 Description

spi_set_state sets the state of the current instance to the given *state* (stopped, paused, or running).

spi_get_state gets the current instance state.

5.2.97.3 Return value

spi_set_state returns zero if it changes the state as requested, non-zero on failure.

5.2.97.4 See also

spi_get_state, **spi_instance_state_t**

5.2.98 spi_spm_start

5.2.98.1 Prototype

```
int
spi_spm_start(
        char *              name,
        int *               argcp,
        char **             argv,
        spi_spm_flags_t     flags
        );
```

5.2.98.2 Description

spi_spm_start starts the Stream programming model runtime. It must be called before an application calls any other Stream programming model function. Parameter *name* supplies the name of the application and parameters **argcp* and *argv* supply program arguments.

The Stream programming model runtime looks for special options in the argument list:

--**spi_default_stack_size=***size*
: Set the stack size for scheduling groups to the given *size* (in bytes); the default is 64K bytes. A scheduling group may specify an explicit minimum stack size with **spi_schedgroup_set_min_stack_size**. Suffixes: [**k**|**m**|**g**].

--**spi_dsp_memory_size=***size*
: Reserve *size* memory for the DSP MIPS image (default: 1M). Suffixes: [**k**|**m**|**g**].

--**spi_dsp_memory_size_image=***image*

--**spi_dsu_memory_size=***size* [Storm-2 only]
: Reserve *size* memory for the DSU MIPS image (default: 1M). Suffixes: [**k**|**m**|**g**].

--**spi_dsu_memory_size_image=***image* [Storm-2 only]
: Use *image* to set the amount of memory reserved for DSU MIPS.

--**spi_dsp_stderr_file=***file*
: Redirect DSU MIPS standard error to *file*. By default, DSU MIPS **stderr** output goes to System MIPS **stderr**.

--**spi_dsp_stdout_file=***file*
: Redirect DSP MIPS standard output to *file*. By default, DSP MIPS **stdout** output goes to System MIPS **stdout**.

--**spi_dsu_stderr_file=***file* [Storm-2 only]
: Redirect DSU MIPS standard error to *file*. By default, DSU MIPS **stderr** output goes to System MIPS **stderr**.

--**spi_dsu_stdout_file=***file* [Storm-2 only]
: Redirect DSU MIPS standard output to *file*. By default, DSU MIPS **stdout** output goes to System MIPS **stdout**.

--**spi_init_file=***file*
: Create instances and connections and execute commands as specified in initialization file *file*. Multiple initialization files may be specified with repeated --**spi_init_fil**e options. *Stream User's Guide* describes the format of initialization files.

--**spi_logdir=***dir*
: Write log files to the given *dir*. By default, logging writes to **stdout**. With this option, logging for each image and instance is written to a separate file in directory *dir*, which must exist.

--spi_log_mask={**error**|**debug**|*},*mask*
Set the enable mask for the error log or debug log (or both) to the given *mask* value
--spi_log_timestamps=[**0**|**1**]
Disable or enable (default) timestamps on log entries.
--spi_shared_memory_size=*size*
Reserve *size* for **spi_buffer_t** ojects. The default value 0 indicates that all memory not used by DSP MIPS or System MIPS should be used for shared memory.
--spi_tcs_logging level=*level*
Set the simulator logging level to *level*.
--spi_trace_buffer_size=*sizze*
Set the stream command trace buffer size to *size*.
--spi_trace_file=*file*
Write stream command trace output to *file*.

The runtime adjusts ***argcp** and **argv** to remove any special options.

spi_spm_stop stops the Stream programming model runtime.

5.2.98.3 Return value

spi_spm_start returns zero if the runtime starts correctly, non-zero on failure.

5.2.98.4 See also

Special SPM options, **spi_load_image**, **spi_spm_flags_t**, **spi_spm_stop**

5.2.99 spi_spm_stop

5.2.99.1 Prototype

#include "spi_spm.h"

int
spi_spm_stop(void);

5.2.99.2 Description

spi_spm_stop stops the Stream programming model runtime. Stopping the runtime stops and destroys all instances that the application created.

spi_spm_start stops the Stream programming model runtime.

5.2.99.3 Return value

spi_spm_stop returns zero if the runtime stops correctly, non-zero on failure.

5.2.99.4 See also

spi_spm_start

5.2.100 spi_timer_get_desc

5.2.100.1 Prototype

#include "spi_spm.h"

const char *
spi_timer_get_desc(spi_timer_t *timer***);**

5.2.100.2 Description

spi_timer_get_desc returns the description of the given *timer*.

spi_timer_new defines a timer, specifying its description.

5.2.100.3 Return value

spi_timer_get_desc returns the description of *timer*. The returned string should not be modified.

5.2.100.4 See also

spi_timer_new, **spi_timer_get_name**

5.2.101 spi_timer_get_name

5.2.101.1 Prototype

#include "spi_spm.h"

const char *
spi_timer_get_name(spi_timer_t *timer***);**

5.2.101.2 Description

spi_timer_get_name returns the name of the given *timer*.

spi_timer_new defines a timer, specifying its name.

5.2.101.3 Return value

spi_timer_get_name returns the name of *timer*. The returned string should not be modified.

5.2.101.4 See also

spi_timer_new, **spi_timer_get_desc**

5.2.102 spi_timer_get_nanoseconds

5.2.102.1 Prototype

#include "spi_spm.h"

unsigned long long
spi_timer_get_nanoseconds(spi_timer_t *timer***);**

5.2.102.2 Description

If the given *timer* is currently running, **spi_timer_get_nanoseconds** gets the time in nanoseconds since it started. If *timer* is currently stopped, **spi_timer_get_nanoseconds** gets the time in nanoseconds for its last start/stop interval.

spi_timer_get_nanoseconds introduces considerable overhead compared to **spi_timer_start** and **spi_timer_stop**. For accurate timing, stop the timer before calling **spi_timer_get_nanoseconds** rather than calling it while the timer is running.

spi_timer_new defines a timer. **spi_timer_get_total_nanoseconds** returns the total time for all start/stop intervals of a timer.

5.2.102.3 Return value

spi_timer_get_nanoseconds returns the time in nanoseconds since *timer* started (if it is running) or for its most recent start/stop interval (if it is stopped).

5.2.102.4 See also

spi_timer_new, **spi_timer_get_total_nanoseconds**, **spi_timer_start**, **spi_timer_stop**

5.2.103 spi_timer_get_start_count

5.2.103.1 Prototype

#include "spi_spm.h"

unsigned int
spi_timer_get_start_count(spi_timer_t *timer***);**

5.2.103.2 Description

spi_timer_get_start_count gets the number of times the given *timer* has been started.

spi_timer_new defines a timer.

5.2.103.3 Return value

spi_timer_get_start_count returns the number of times *timer* has been started. The value of **spi_timer_get_start_count** is undefined for the built-in **SPI_TIMER_KERNEL** timer.

5.2.103.4 See also

spi_timer_new,**spi_timer_start**

5.2.104 spi_timer_get_total_nanoseconds

5.2.104.1 Prototype

#include "spi_spm.h"

unsigned long long
spi_timer_get_total_nanoseconds(spi_timer_t *timer***);**

5.2.104.2 Description

spi_timer_get_total_nanoseconds gets the total time in nanoseconds for all start/stop intervals of the given *timer*. If *timer* is currently running, the total time includes the time since it was last started.

spi_timer_get_total_nanoseconds introduces considerable overhead compared to **spi_timer_start** and **spi_timer_stop**. For accurate timing, stop the timer before calling **spi_timer_get_total_nanoseconds** rather than calling it while the timer is running.

spi_timer_new defines a timer for a component. **spi_timer_get_nanoseconds** returns the time for a single start/stop interval of a timer.

5.2.104.3 Return value

spi_timer_get_total_nanoseconds returns the time in nanoseconds for all start/stop intervals of *timer*. For the built-in **SPI_TIMER_KERNEL** timer, **spi_timer_get_total_nanoseconds** returns the same value as **spi_timer_get_nanoseconds**.

5.2.104.4 See also

spi_timer_new, **spi_timer_get_nanoseconds**

5.2.105 spi_timer_new

5.2.105.1 Prototype

```
#include "spi_spm.h"

spi_timer_t
spi_timer_new(
        const char *        name,
        const char *        description,
        );
```

5.2.105.2 Description

spi_timer_new defines a timer for a component. The parameters define the *name* and *description* of the timer.

spi_timer_start and **spi_timer_stop** start and stop a timer. **spi_timer_get_start_count** returns the number of times a timer has been started. **spi_timer_get_nanoseconds** returns the elapsed time since a timer started. **spi_timer_get_total_nanoseconds** returns the total elapsed time for a timer. **spi_timer_get_name** and **spi_timer_get_desc** return the name and description of a timer. **spi_get_timer** gets a timer with a given name.

5.2.105.3 Return value

spi_timer_new returns a new timer.

5.2.105.4 See also

spi_get_timer, **spi_timer_get_desc**, , **spi_timer_get_name**, **spi_timer_get_nanoseconds**, **spi_timer_get_start_count**, **spi_timer_get_total_nanoseconds**, **spi_timer_start**, **spi_timer_stop**

5.2.106 spi_timer_start

5.2.106.1 Prototype

#include "spi_spm.h"

int
spi_timer_start(spi_timer_t *timer***);**

5.2.106.2 Description

spi_timer_start starts the given *timer*. If *timer* is already running, **spi_timer_start** has no effect. **spi_timer_start** has no effect fot the built-in **SPI_TIMER_KERNEL** timer.

spi_timer_stop stops a given timer. **spi_timer_new** defines a timer for a component.

5.2.106.3 Return value

spi_timer_start returns zero if *timer* is started successfully or is already running. It returns non-zero on failure.

5.2.106.4 See also

spi_timer_new, **spi_timer_stop**

5.2.107 spi_timer_stop

5.2.107.1 Prototype

#include "spi_spm.h"

int
spi_timer_stop(spi_timer_t *timer***);**

5.2.107.2 Description

spi_timer_stop stops the given *timer*. If *timer* is already stopped, **spi_timer_stop** has no effect. **spi_timer_stop** has no effect fot the built-in **SPI_TIMER_KERNEL** timer.

spi_timer_start starts a given timer. **spi_timer_new** defines a timer for a component.

5.2.107.3 Return value

spi_timer_stop returns zero if *timer* is stopped successfully or is already stopped. It returns non-zero on failure.

5.2.107.4 See also

spi_timer_new, **spi_timer_start**

5.2.108 spi_trace_is_enabled

5.2.108.1 Prototype

#include "spi_spm.h"

int
spi_trace_is_enabled(void);

5.2.108.2 Description

spi_trace_is_enabled determines if program tracing is enabled.

If tracing is enabled, **spi_trace_start** starts program tracing and **spi_trace_stop** stops program tracing.

5.2.108.3 Return value

spi_trace_is_enabled returns non-zero if tracing is enabled or zero otherwise.

5.2.108.4 See also

spi_trace_start, **spi_trace_stop**

5.2.109 spi_trace_start

5.2.109.1 Prototype

#include "spi_spm.h"

int
spi_trace_start(void);

5.2.109.2 Description

spi_trace_start starts program tracing. If tracing is already started, **spi_trace_start** has no effect.

spi_trace_stop stops program tracing. **spi_trace_is_enabled** determines if tracing is enabled.

5.2.109.3 Return value

spi_trace_start returns zero if tracing is started successfully or is already running. It returns non-zero on failure.

5.2.109.4 See also

spi_trace_is_enabled, **spi_trace_stop**

5.2.110 spi_trace_stop

5.2.110.1 Prototype

#include "spi_spm.h"

int
spi_trace_stop(void);

5.2.110.2 Description

spi_trace_stop stops program tracing. If the tracing is already stopped, **spi_trace_stop** has no effect.

spi_trace_start stops program tracing. **spi_trace_is_enabled** determines if tracing is enabled.

5.2.110.3 Return value

spi_trace_stop returns zero if tracing is stopped successfully or is already stopped. It returns non-zero on failure.

5.2.110.4 See also

spi_trace_is_enabled, **spi_trace_stop**

6 Pipeline API

This chapter describes Stream programming model Pipeline API functions that support stream processor on-chip memory management. Pipeline API functions may be used on DSP MIPS only, not on System MIPS or within kernel functions. Pipeline API stream functions issue commands to the stream controller hardware unit of the stream processor, so data transfers occur asynchronously with DSP MIPS operation. Stream functions may be used only in Stream code (in a **.sc** file), not in ordinary C code (in a **.c** file).

Stream functions **spi_load_*** load stream data to the LRF and stream functions **spi_store_*** store stream data from the LRF; different forms of these functions specify different data access patterns. **spi_count** returns the number of valid data items in a stream. **spi_out** returns the value of a kernel scalar output parameter.

6.1 Pipeline API stream functions

Stream processor hardware supports several different data transfer patterns that allow an application to distribute and replicate data according to the needs of the kernel. Functions **spi_load_block** loads a block of contiguous data records from memory to the LRF. Function **spi_load_index** loads records from locations specified by an index stream; the index stream defines the offsets of desired records in memory. Function **spi_load_stride** loads records from locations in memory separated by a given stride. Functions **spi_store_block**, **spi_store_index**, and **spi_store_stride** similarly store records from the LRF to memory with a specific transfer pattern.

A table in the Pipeline API functions summary section of this document lists each Pipeline API function.

6.1.1 spi_barrier

6.1.1.1 Prototype

void
spi_barrier(**void**);

6.1.1.2 Description

spi_barrier causes DSP MIPS to wait for completion of all DPU activity before continuing. After the **spi_barrier,** all kernel function execution will be complete and all data transfers between buffers and streams will be complete.

For example, a program might call **spi_barrier** before stopping a timer after a kernel call, to assure that the timer measures the total time required by the kernel. Without the **spi_barrier** call, the kernel could still be running when the timer is stopped.

6.1.1.3 Return value

None.

6.1.2 spi_count

6.1.2.1 Prototype

```
int
spi_count(                  // returns the number of valid records in stream
        stream type str     // stream
);
```

6.1.2.2 Description

spi_count returns the number of valid data records currently in stream *str*. A stream's record count is undefined when the stream is declared. Writing to an output stream sets the count to the number of records written to the stream. Reading or updating a stream does not change its count.

In summary:

- The count of a stream is initially *undefined.*
- A stream *write* sets the stream count to the actual number of records written. For a sequential stream, the number of records will always be a multiple of the number of lanes in the device. For a conditional stream, the number of records will be the actual number of records written by the operation, not necessarily a multiple of **SPI_LANES**.
- A stream *read* or *update* does not change stream count.
- Using a substream (e.g. `my_stream(0, 16)`) never changes the count of a stream, even when used as an output.

The table below shows the effect of various operations on **spi_count**. Here **write** means that the operation sets **spi_count**, **update** means the operation updates the contents of the stream but leaves **spi_count** unchanged, and **read** means that the operation does not change the contents or the **spi_count** of the stream.

Operation	Stream	Substream
spi_load_*	Write	Update
spi_store_*	Read	Read
Use as index to **spi_load_*** or **spi_store_***	Read	Read
Kernel sequential or conditional input	Read	Read
Kernel array input	Read	Read
Kernel sequential or conditional output	Write	Update
Kernel array output	Update	Update
Kernel array I/O	Update	Update

6.1.2.3 Return value

spi_count returns the count of the stream *str* as defined above, or -1 if an error occurs.

6.1.3 spi_load_2d_index

6.1.3.1 Prototype

```
void
spi_load_2d_index(                          // Storm-2 only
        stream type     str,                // Destination stream
        spi_buffer_t    buffer,             // Source data buffer
        unsigned int    offset,             // Byte offset from beginning of buffer
        unsigned int    width,              // Image width in bytes
        unsigned int    x_stride,           // Horizontal (X) stride in bytes
        unsigned int    blocks_per_group,   // Number of blocks to load per group
        stream int16x2  xy_indices,         // Coordinates for start of each group of blocks
        unsigned int    block_width,        // Block width in bytes
        unsigned int    block_height,       // Block height in rows
        unsigned int    count               // Number of blocks to load
);
```

6.1.3.2 Description

Storm-2 function **spi_load_2d_index** transfers data from records at the given *offset* in *buffer* to the local register file (LRF) of a stream processor. It is similar to **spi_load_index**, but with additional arguments that allow it to manipulate two-dimensional (2d) images efficiently.

spi_load_2d_index loads a total of *count* 2d blocks from an image of the given *width*. Each block is a rectangle of dimensions *block_width* by *block_height*. If *block_width* is not a multiple of 4, **spi_load_2d_index** 0-pads each row of each block in the LRF to the next word multiple. The basetype *type* of stream *str* must represent a single *block_width* (plus padding if necessary) by *block_height* block.

Index stream *xy_indices* contains packed halfwords giving the x and y coordinates (in bytes, with x in the low-order halfword and y in the high-order halfword) of the beginning of each group of blocks. *x_stride* gives the horizontal stride in bytes between successive blocks in a horizontal group, and *blocks_per_group* gives the number of blocks in each group. Index (-1, -1) (that is, 0xFFFFFFFFp2) loads a group of 0-filled blocks to the LRF; any other negative index produces a runtime error.

In pseudocode:

```
n = 0;
foreach (x, y) in  xy_indices {
        for (i = 0; i < blocks_per_group; i++) {
                if (x, y) == (-1, -1) {
                        load block_width by block_height 0-filled block to lane n
                } else {
                        load block_width by block_height block from (x, y) to lane n
                        x += x_stride;
                }
                n = (n + 1) % SPI_LANES;
        }
}
```

offset may be any 32-bit value. *block_height* must be less than 256. All other parameters must be less than 65536.

The argument *buffer* should not be open. If a program calls **spi_load_2d_index** with an open *buffer*, the runtime will cause the program to exit with a failure status in functional mode or debug mode.

spi_load_2d_index_clip is like **spi_load_2d_index**, but with additional arguments that allow it to crop the loaded image blocks.

6.1.3.3 Return value

None.

6.1.3.4 See also

spi_load_2d_index_crop, **spi_load_2d_stride**, **spi_load_index**, **spi_store_2d_index**

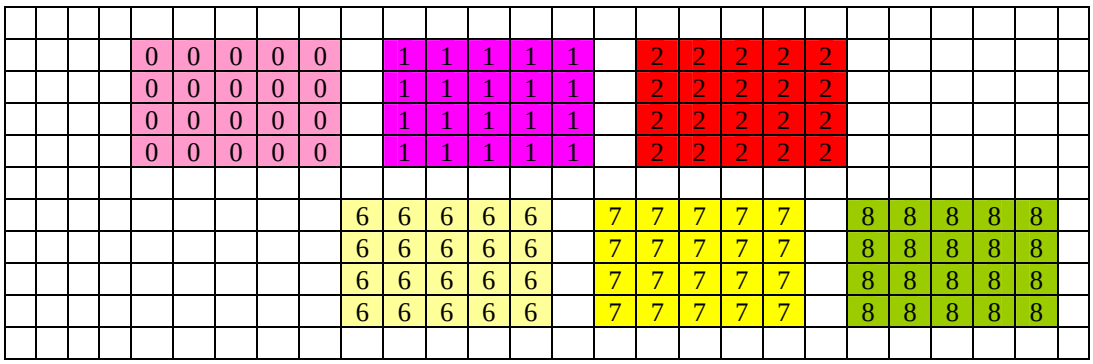
6.1.3.5 Example

The following example loads nine 5 by 4 blocks (including three 0-filled blocks) from a 27 by 11 image.

```
struct b5x4 {
        uint8x4 row0_lo;        // low-order 4 bytes of row 0
        uint8x4 row0_hi;        // 5th byte of row 0 in low-order byte, 0-padding in higher bytes
        uint8x4 row1_lo;        // row 1
        uint8x4 row1_hi;
        uint8x4 row2_lo;        // row 2
        uint8x4 row2_hi;
        uint8x4 row3_lo;        // row 3
        uint8x4 row3_hi;
};
spi_buffer_t            buffer;
stream struct b5x4      s;
stream int16x2          xy_indices;
```

Suppose index stream **xy_indices** contains three entries, specifying coordinates (4, 1), (-1, -1), and (9, 6); that is, it contains 0x00010004p2, 0xFFFFFFFFp2, and 0x00060009p2.

In **buffer**, with each box representing one byte:



Then:

```
spi_load_2d_index(
        s,
        buffer,
        0,              // byte offset from beginning of buffer
        27,             // image width in bytes
        6,              // x-stride between blocks in bytes
        3,              // number of blocks in each group
        xy_indices,     // index stream
        5,              // block width in bytes
        4,              // block height in rows
        9               // number of blocks to load
);
```

Each block is 5 bytes wide, so **spi_load_2d_index** 0-pads each block row to 8 bytes (2 words) in the LRF. After **spi_load_2d_index**, stream **s** in the LRF of a 4-lane stream processor contains (with each box representing one 4-byte word):

Lane	0	1	2	3
low address	0 **row0_lo**	1 **row0_lo**	2 **row0_lo**	3 **row0_lo** (0)
	0 **row0_hi**	1 **row0_hi**	2 **row0_hi**	3 **row0_hi** (0)
	0 **row1_lo**	1 **row1_lo**	2 **row1_lo**	3 **row1_lo** (0)
	0 **row1_hi**	1 **row1_hi**	2 **row1_hi**	3 **row1_hi** (0)
	0 **row2_lo**	1 **row2_lo**	2 **row2_lo**	3 **row2_lo** (0)
	0 **row2_hi**	1 **row2_hi**	2 **row2_hi**	3 **row2_hi** (0)
	0 **row3_lo**	1 **row3_lo**	2 **row3_lo**	3 **row3_lo** (0)
	0 **row3_hi**	1 **row3_hi**	2 **row3_hi**	3 **row3_hi** (0)
	4 **row0_lo** (0)	5 **row0_lo** (0)	6 **row0_lo**	7 **row0_lo**
	4 **row0_hi** (0)	5 **row0_hi** (0)	6 **row0_hi**	7 **row0_hi**
	4 **row1_lo** (0)	5 **row1_lo** (0)	6 **row1_lo**	7 **row1_lo**
	4 **row1_hi** (0)	5 **row1_hi** (0)	6 **row1_hi**	7 **row1_hi**
	4 **row2_lo** (0)	5 **row2_lo** (0)	6 **row2_lo**	7 **row2_lo**
	4 **row2_hi** (0)	5 **row2_hi** (0)	6 **row2_hi**	7 **row2_hi**
	4 **row3_lo** (0)	5 **row3_lo** (0)	6 **row3_lo**	7 **row3_lo**
	4 **row3_hi** (0)	5 **row3_hi** (0)	6 **row3_hi**	7 **row3_hi**
	8 **row0_lo**			
	8 **row0_hi**			
	8 **row1_lo**			
	8 **row1_hi**			
	8 **row2_lo**			
	8 **row2_hi**			
	8 **row3_lo**			
high address	8 **row3_hi**			

spi_load_2d_index loads 0s for the records shown in gray, as blocks 3, 4, and 5 correspond to an index stream entry of (-1, -1).

6.1.4 spi_load_2d_index_crop

6.1.4.1 Prototype

```
void
spi_load_2d_index_crop(                    // Storm-2 only
    stream type      str,                  // Destination stream
    spi_buffer_t     buffer,               // Source data buffer
    unsigned int     offset,               // Byte offset from beginning of buffer
    unsigned int     width,                // Image width in bytes
    unsigned int     crop_width,           // Crop width in bytes
    unsigned int     crop_height,          // Crop height in rows
    unsigned int     x_stride,             // Horizontal (X) stride in bytes
    unsigned int     blocks_per_group,     // Number of blocks to load per group
    stream int16x2   xy_indices,           // Coordinates for start of each group of blocks
    unsigned int     block_width,          // Block width in bytes
    unsigned int     block_height,         // Block height in rows
    unsigned int     count                 // Number of blocks to load
);
```

6.1.4.2 Description

Storm-2 function **spi_load_2d_index_crop** transfers data from records at the given *offset* in *buffer* to the local register file (LRF) of a stream processor. It is similar to **spi_load_2d_index**, but with additional arguments that allow it to crop the loaded image blocks. The **spi_load_2d_index** page gives pseudocode that describes its operation.

spi_load_2d_index_crop loads a total of *count* 2d blocks from an image of the given *width*. Each block is a rectangle of dimensions *block_width* by *block_height*. If *block_width* is not a multiple of 4, **spi_load_2d_index_crop** 0-pads each row of each block in the LRF to the next word multiple. The basetype *type* of stream *str* must represent a single *block_width* (plus padding if necessary) by *block_height* block.

Index stream *xy_indices* contains packed halfwords giving the x and y coordinates (in bytes, with x in the low-order halfword and y in the high-order halfword) of the beginning of each group of blocks. *x_stride* gives the horizontal stride in bytes between successive blocks in a horizontal group, and *blocks_per_group* gives the number of blocks in each group. Index (-1, -1) (that is, 0xFFFFFFFFp2) loads a group of 0-filled blocks to the LRF; any other negative index produces a runtime error.

crop_width and *crop_height* specify the dimensions of a cropping window relative to the given *offset*. **spi_load_2d_index_crop** loads 0 for any part of a block that falls outside the cropping rectangle.

offset may be any 32-bit value. *block_height* must be less than 256. All other parameters must be less than 65536.

The argument *buffer* should not be open. If a program calls **spi_load_2d_index_crop** with an open *buffer*, the runtime will cause the program to exit with a failure status in functional mode or debug mode.

6.1.4.3 Return value

None.

6.1.4.4 See alsospi_load_2d_index, spi_load_2d_stride_crop, spi_load_index, spi_store_2d_index_crop

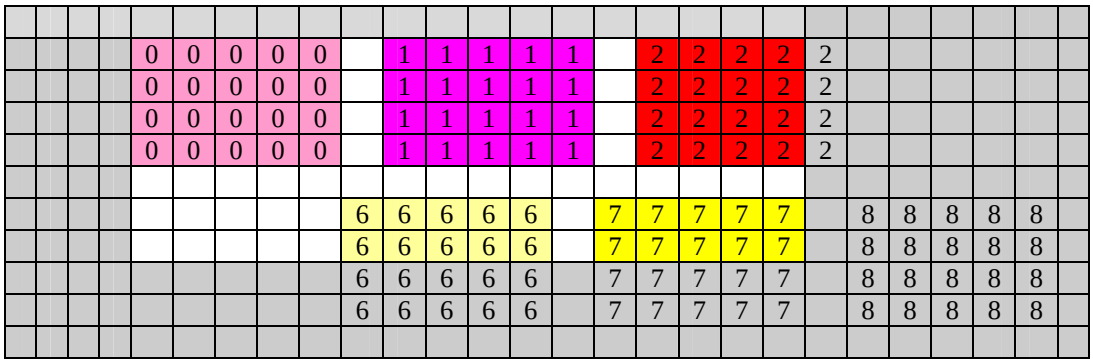
6.1.4.5 Example

The following example loads nine 5 by 4 blocks (including three 0-filled blocks) from a 27 by 11 image.

```
struct b5x4 {
        uint8x4 row0_lo;        // low-order 4 bytes of row 0
        uint8x4 row0_hi;        // 5th byte of row 0 in low-order byte, 0-padding in higher bytes
        uint8x4 row1_lo;        // row 1
        uint8x4 row1_hi;
        uint8x4 row2_lo;        // row 2
        uint8x4 row2_hi;
        uint8x4 row3_lo;        // row 3
        uint8x4 row3_hi;
};
spi_buffer_t            buffer;
stream struct b5x4      s;
stream int16x2          xy_indices;
```

Suppose index stream *xy_indices* contains three entries, specifying coordinates (0, 0), (-1, -1), and (5, 5); that is, it contains 0x00000000p2, 0xFFFFFFFFp2, and 0x00050005p2.

In **buffer**, with each box representing one byte:



Then:

```
spi_load_2d_index_crop(
        s,
        buffer,
        1 * 27 + 4,         // byte offset from beginning of buffer
        27,                 // image width in bytes
        16,                 // crop width in bytes
        7,                  // crop height in rows
        6,                  // x-stride between blocks in bytes
        3,                  // number of blocks in each group
        xy_indices,         // index stream
        5,                  // block width in bytes
        4,                  // block height in rows
        9                   // number of blocks to load
);
```

Each block is 5 bytes wide, so **spi_load_2d_index_crop** 0-pads each block row to 8 bytes (2 words) in the LRF. After **spi_load_2d_index_crop**, stream **s** in the LRF of a 4-lane stream processor contains (with each box representing one 4-byte word):

Lane	0	1	2	3
low address	0 **row0_lo**	1 **row0_lo**	2 **row0_lo**	3 **row0_lo** (0)
	0 **row0_hi**	1 **row0_hi**	2 **row0_hi** (0)	3 **row0_hi** (0)
	0 **row1_lo**	1 **row1_lo**	2 **row1_lo**	3 **row1_lo** (0)
	0 **row1_hi**	1 **row1_hi**	2 **row1_hi** (0)	3 **row1_hi** (0)
	0 **row2_lo**	1 **row2_lo**	2 **row2_lo**	3 **row2_lo** (0)
	0 **row2_hi**	1 **row2_hi**	2 **row2_hi** (0)	3 **row2_hi** (0)
	0 **row3_lo**	1 **row3_lo**	2 **row3_lo**	3 **row3_lo** (0)
	0 **row3_hi**	1 **row3_hi**	2 **row3_hi** (0)	3 **row3_hi** (0)
	4 **row0_lo** (0)	5 **row0_lo** (0)	6 **row0_lo**	7 **row0_lo**
	4 **row0_hi** (0)	5 **row0_hi** (0)	6 **row0_hi**	7 **row0_hi**
	4 **row1_lo** (0)	5 **row1_lo** (0)	6 **row1_lo**	7 **row1_lo**
	4 **row1_hi** (0)	5 **row1_hi** (0)	6 **row1_hi**	7 **row1_hi**
	4 **row2_lo** (0)	5 **row2_lo** (0)	6 **row2_lo** (0)	7 **row2_lo** (0)
	4 **row2_hi** (0)	5 **row2_hi** (0)	6 **row2_hi** (0)	7 **row2_hi** (0)
	4 **row3_lo** (0)	5 **row3_lo** (0)	6 **row3_lo** (0)	7 **row3_lo** (0)
	4 **row3_hi** (0)	5 **row3_hi** (0)	6 **row3_hi** (0)	7 **row3_hi** (0)
	8 **row0_lo** (0)			
	8 **row0_hi** (0)			
	8 **row1_lo** (0)			
	8 **row1_hi** (0)			
	8 **row2_lo** (0)			
	8 **row2_hi** (0)			
	8 **row3_lo** (0)			
high address	8 **row3_hi** (0)			

spi_load_2d_index loads 0s for the records shown in gray. It 0-fills blocks 3, 4, and 5 because they correspond to an index stream entry of (-1, -1), while it 0-fills the shaded portions of blocks 2, 6, 7, and 8 because they fall outside the cropping window.

6.1.5 spi_load_2d_stride

6.1.5.1 Prototype

```
void
spi_load_2d_stride(                          // Storm-2 only
        stream type     str,                 // Destination stream
        spi_buffer_t    buffer,              // Source data buffer
        unsigned int    offset,              // Byte offset from beginning of buffer
        unsigned int    width,               // Image width in bytes
        unsigned int    x_stride,            // Horizontal (X) stride in bytes
        unsigned int    blocks_per_group,    // Number of blocks to load per group
        unsigned int    y_stride,            // Vertical (Y) stride in rows
        unsigned int    block_width,         // Block width in bytes
        unsigned int    block_height,        // Block height in rows
        unsigned int    count                // Number of blocks to load
);
```

6.1.5.2 Description

Storm-2 function **spi_load_2d_stride** transfers data records from the given *offset* in *buffer* to the local register file (LRF) of a stream processor. It is similar to **spi_load_stride**, but with additional arguments that allow it to load rectangular portions of a two-dimensional (2d) image efficiently.

spi_load_2d_stride loads a total of *count* 2d blocks from an image of the given *width*. Each block is a rectangle of dimensions *block_width* by *block_height*. If *block_width* is not a multiple of 4, **spi_load_2d_stride** 0-pads each row of each block in the LRF to the next word multiple. The basetype *type* of stream *str* must represent a single *block_width* (plus padding if necessary) by *block_height* block.

x_stride gives the horizontal stride in bytes between records loaded to successive lanes, *blocks_per_group* gives the number of blocks in a horizontal group, and *y_stride* gives the vertical stride in rows between successive groups of records.

In pseudocode:

```
x0 = offset % width;
y = offset / width;
n = 0;
for (b = 0; b < count; b += blocks_per_group) {
        x = x0;
        for (i = 0; i < blocks_per_group && b + i < count; i++) {
                load block_width by block_height block from (x, y) to lane n
                x += x_stride;
                n = (n + 1) % SPI_LANES;
        }
        y += y_stride;
}
```

offset may be any 32-bit value. *block_height* and *y_stride* must be less than 256. All other parameters must be less than 65536.

The argument *buffer* should not be open. If a program calls **spi_load_2d_stride** with an open *buffer*, the runtime will cause the program to exit with a failure status in functional mode or debug mode.

spi_load_2d_stride_crop is like **spi_load_2d_stride**, but with additional arguments that allow it to crop the loaded image blocks.

6.1.5.3 Return value

None.

6.1.5.4 See also

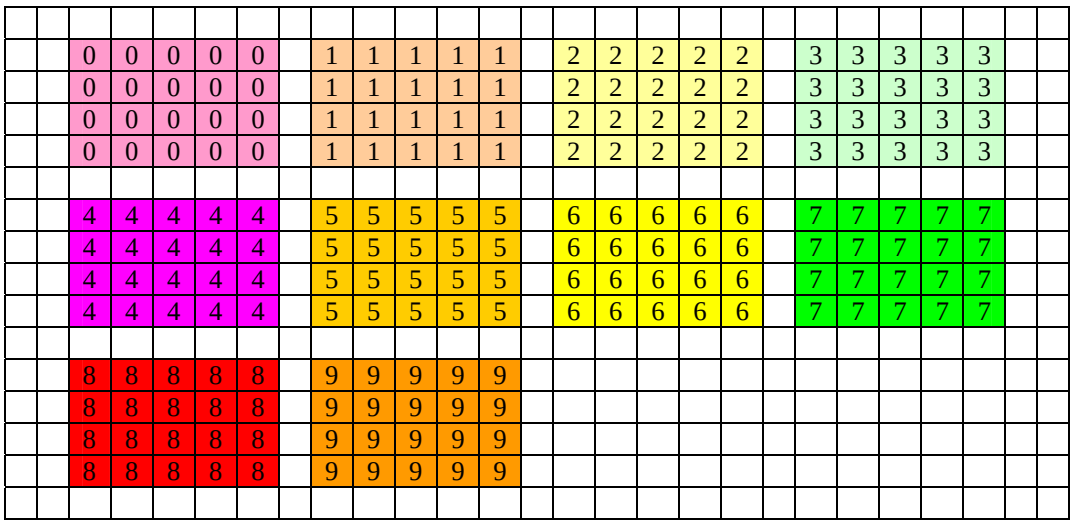
spi_load_2d_index, spi_load_2d_stride_crop, **spi_load_stride**, **spi_store_2d_stride**

6.1.5.5 Example

The following example loads ten 5 by 4 blocks from a 27 by 16 image.

```
struct b5x4 {
        uint8x4 row0_lo;        // low-order 4 bytes of row 0
        uint8x4 row0_hi;        // 5th byte of row 0 in low-order byte, 0-padding in higher bytes
        uint8x4 row1_lo;        // row 1
        uint8x4 row1_hi;
        uint8x4 row2_lo;        // row 2
        uint8x4 row2_hi;
        uint8x4 row3_lo;        // row 3
        uint8x4 row3_hi;
};
spi_buffer_t            buffer;
stream struct b5x4      s;
```

In **buffer**, with each box representing one byte:



Then:

```
spi_load_2d_stride(
        s,
        buffer,
        1 * 27 + 2,         // byte offset of first block
        27,                 // image width in bytes
        6,                  // x-stride between blocks in bytes
        4,                  // number of blocks before y-stride
        5,                  // y-stride between blocks
        5,                  // block width in bytes
        4,                  // block height in rows
        10                  // number of blocks to load
);
```

Each block is 5 bytes wide, so **spi_load_2d_stride** 0-pads each block row to 8 bytes (2 words) in the LRF. After **spi_load_2d_stride**, stream **s** in the LRF of a 4-lane stream processor contains (with each box representing one 4-byte word):

Lane	0	1	2	3
low address	0 **row0_lo**	1 **row0_lo**	2 **row0_lo**	3 **row0_lo**
	0 **row0_hi**	1 **row0_hi**	2 **row0_hi**	3 **row0_hi**
	0 **row1_lo**	1 **row1_lo**	2 **row1_lo**	3 **row1_lo**
	0 **row1_hi**	1 **row1_hi**	2 **row1_hi**	3 **row1_hi**
	0 **row2_lo**	1 **row2_lo**	2 **row2_lo**	3 **row2_lo**
	0 **row2_hi**	1 **row2_hi**	2 **row2_hi**	3 **row2_hi**
	0 **row3_lo**	1 **row3_lo**	2 **row3_lo**	3 **row3_lo**
	0 **row3_hi**	1 **row3_hi**	2 **row3_hi**	3 **row3_hi**
	4 **row0_lo**	5 **row0_lo**	6 **row0_lo**	7 **row0_lo**
	4 **row0_hi**	5 **row0_hi**	6 **row0_hi**	7 **row0_hi**
	4 **row1_lo**	5 **row1_lo**	6 **row1_lo**	7 **row1_lo**
	4 **row1_hi**	5 **row1_hi**	6 **row1_hi**	7 **row1_hi**
	4 **row2_lo**	5 **row2_lo**	6 **row2_lo**	7 **row2_lo**
	4 **row2_hi**	5 **row2_hi**	6 **row2_hi**	7 **row2_hi**
	4 **row3_lo**	5 **row3_lo**	6 **row3_lo**	7 **row3_lo**
	4 **row3_hi**	5 **row3_hi**	6 **row3_hi**	7 **row3_hi**
	8 **row0_lo**	9 **row0_lo**		
	8 **row0_hi**	9 **row0_hi**		
	8 **row1_lo**	9 **row1_lo**		
	8 **row1_hi**	9 **row1_hi**		
	8 **row2_lo**	9 **row2_lo**		
	8 **row2_hi**	9 **row2_hi**		
	8 **row3_lo**	9 **row3_lo**		
high address	8 **row3_hi**	9 **row3_hi**		

6.1.6 spi_load_2d_stride_crop

6.1.6.1 Prototype

```
void
spi_load_2d_stride_crop(                    // Storm-2 only
    stream type     str,                    // Destination stream
    spi_buffer_t    buffer,                 // Source data buffer
    unsigned int    offset,                 // Byte offset from beginning of buffer
    unsigned int    width,                  // Image width in bytes
    unsigned int    crop_width,             // Crop width in bytes
    unsigned int    crop_height,            // Crop height in rows
    unsigned int    x_stride,               // Horizontal (X) stride in bytes
    unsigned int    blocks_per_group,       // Number of blocks to load per group
    unsigned int    y_stride,               // Vertical (Y) stride in rows
    unsigned int    block_width,            // Block width in bytes
    unsigned int    block_height,           // Block height in rows
    unsigned int    count                   // Number of blocks to load
);
```

6.1.6.2 Description

Storm-2 function **spi_load_2d_stride_crop** transfers data records from the given *offset* in *buffer* to the local register file (LRF) of a stream processor. It is similar to **spi_load_2d_stride**, but with additional arguments that allow it to crop the loaded image blocks. The **spi_load_2d_stride** page gives pseudocode that describes its operation.

spi_load_2d_stride_crop loads a total of *count* 2d blocks from an image of the given *width*. Each block is a rectangle of dimensions *block_width* by *block_height*. If *block_width* is not a multiple of 4, **spi_load_2d_stride_crop** 0-pads each row of each block in the LRF to the next word multiple. The basetype *type* of stream *str* must represent a single *block_width* (plus padding if necessary) by *block_height* block.

x_stride gives the horizontal stride in bytes between records loaded to successive lanes, *blocks_per_group* gives the number of blocks in a horizontal group, and *y_stride* gives the vertical stride in rows between successive groups of records.

crop_width and *crop_height* specify the dimensions of a cropping window relative to the given *offset*. **spi_load_2d_stride_crop** loads 0 for any part of a block that falls outside the cropping rectangle.

offset may be any 32-bit value. *block_height* and *y_stride* must be less than 256. All other parameters must be less than 65536.

The argument *buffer* should not be open. If a program calls **spi_load_2d_stride** with an open *buffer*, the runtime will cause the program to exit with a failure status in functional mode or debug mode.

6.1.6.3 Return value

None.

6.1.6.4 See also

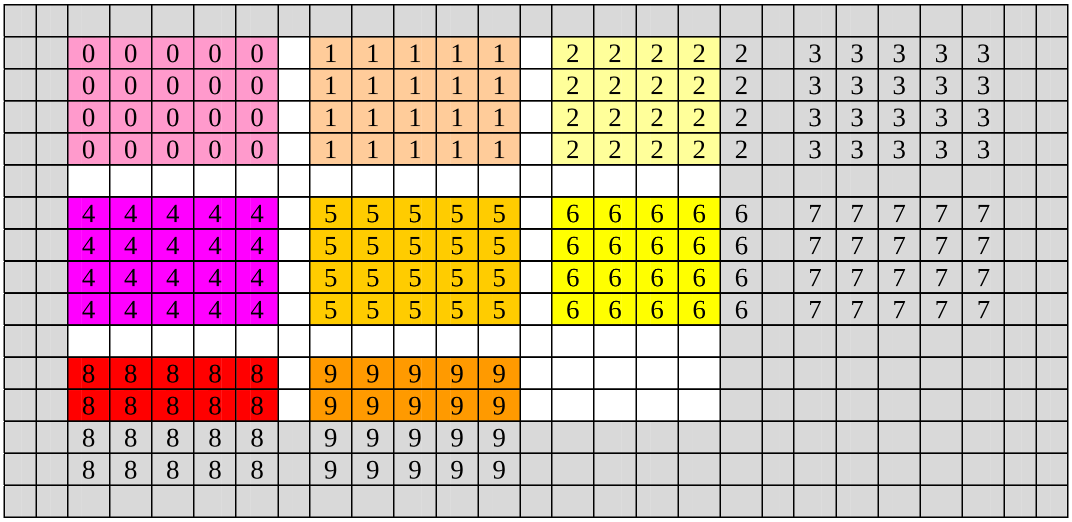
spi_load_2d_index_crop, **spi_load_2d_stride**, **spi_load_stride**, **spi_store_2d_stride_crop**

6.1.6.5 Example

The following example loads ten 5 by 4 blocks from a 27 by 16 image, using a 16 by 12 cropping window.

```
struct b5x4 {
            uint8x4  row0_lo;          // low-order 4 bytes of row 0
            uint8x4  row0_hi;          // 5th byte of row 0 in low-order byte, 0-padding in higher bytes
            uint8x4  row1_lo;          // row 1
            uint8x4  row1_hi;
            uint8x4  row2_lo;          // row 2
            uint8x4  row2_hi;
            uint8x4  row3_lo;          // row 3
            uint8x4  row3_hi;
};
spi_buffer_t                buffer;
stream struct b5x4          s;
```

In **buffer**, with each box representing one byte:



		0	0	0	0	0		1	1	1	1	1		2	2	2	2	2		3	3	3	3	3		
		0	0	0	0	0		1	1	1	1	1		2	2	2	2	2		3	3	3	3	3		
		0	0	0	0	0		1	1	1	1	1		2	2	2	2	2		3	3	3	3	3		
		0	0	0	0	0		1	1	1	1	1		2	2	2	2	2		3	3	3	3	3		
		4	4	4	4	4		5	5	5	5	5		6	6	6	6	6		7	7	7	7	7		
		4	4	4	4	4		5	5	5	5	5		6	6	6	6	6		7	7	7	7	7		
		4	4	4	4	4		5	5	5	5	5		6	6	6	6	6		7	7	7	7	7		
		4	4	4	4	4		5	5	5	5	5		6	6	6	6	6		7	7	7	7	7		
		8	8	8	8	8		9	9	9	9	9														
		8	8	8	8	8		9	9	9	9	9														
		8	8	8	8	8		9	9	9	9	9														
		8	8	8	8	8		9	9	9	9	9														

Then:

```
spi_load_2d_stride_crop(
        s,
        buffer,
        1 * 27 + 2,         // byte offset of first block
        27,                 // image width in bytes
        16,                 // crop width in bytes
        12,                 // crop height in rows
        6,                  // x-stride between blocks in bytes
        4,                  // number of blocks before y-stride
        5,                  // y-stride between blocks
        5,                  // block width in bytes
        4,                  // block height in rows
        10                  // number of blocks to load
);
```

Each block is 5 bytes wide, so **spi_load_2d_stride_crop** 0-pads each block row to 8 bytes (2 words) in the LRF. After **spi_load_2d_stride_crop**, stream **s** in the LRF of a 4-lane stream processor contains (with each box representing one 4-byte word):

Lane	0	1	2	3
low address	0 **row0_lo**	1 **row0_lo**	2 **row0_lo**	3 **row0_lo** (0)
	0 **row0_hi**	1 **row0_hi**	2 **row0_hi** (0)	3 **row0_hi** (0)
	0 **row1_lo**	1 **row1_lo**	2 **row1_lo**	3 **row1_lo** (0)
	0 **row1_hi**	1 **row1_hi**	2 **row1_hi** (0)	3 **row1_hi** (0)
	0 **row2_lo**	1 **row2_lo**	2 **row2_lo**	3 **row2_lo** (0)
	0 **row2_hi**	1 **row2_hi**	2 **row2_hi** (0)	3 **row2_hi** (0)
	0 **row3_lo**	1 **row3_lo**	2 **row3_lo**	3 **row3_lo** (0)
	0 **row3_hi**	1 **row3_hi**	2 **row3_hi** (0)	3 **row3_hi** (0)
	4 **row0_lo**	5 **row0_lo**	6 **row0_lo**	7 **row0_lo** (0)
	4 **row0_hi**	5 **row0_hi**	6 **row0_hi** (0)	7 **row0_hi** (0)
	4 **row1_lo**	5 **row1_lo**	6 **row1_lo**	7 **row1_lo** (0)
	4 **row1_hi**	5 **row1_hi**	6 **row1_hi** (0)	7 **row1_hi** (0)
	4 **row2_lo**	5 **row2_lo**	6 **row2_lo**	7 **row2_lo** (0)
	4 **row2_hi**	5 **row2_hi**	6 **row2_hi** (0)	7 **row2_hi** (0)
	4 **row3_lo**	5 **row3_lo**	6 **row3_lo**	7 **row3_lo** (0)
	4 **row3_hi**	5 **row3_hi**	6 **row3_hi** (0)	7 **row3_hi** (0)
	8 **row0_lo**	9 **row0_lo**		
	8 **row0_hi**	9 **row0_hi**		
	8 **row1_lo**	9 **row1_lo**		
	8 **row1_hi**	9 **row1_hi**		
	8 **row2_lo** (0)	9 **row2_lo** (0)		
	8 **row2_hi** (0)	9 **row2_hi** (0)		
	8 **row3_lo** (0)	9 **row3_lo** (0)		
high address	8 **row3_hi** (0)	9 **row3_hi** (0)		

spi_load_2d_stride_crop loads 0s for the records shown in gray, as they lie outside the cropping window.

6.1.7 spi_load_block

6.1.7.1 Prototype

```
void
spi_load_block(
    stream type    str,        // Destination stream
    spi_buffer_t   buffer,     // Source data buffer
    unsigned int   offset,     // Byte offset from beginning of buffer
    unsigned int   count       // Number of records to load
);
```

6.1.7.2 Description

spi_load_block transfers a block of contiguous data records from the given byte *offset* in *buffer* to the local register file (LRF) of a stream processor. This allows DSP MIPS to pass input data to a kernel as an input stream.

On Storm-1 (SP16 and SP8), if *count* is not a multiple of **SPI_LANES**, the contents of LRF locations immediately past the given *count* are undefined. On Storm-2, the contents of LRF locations past the given count remain unchanged.

On Storm-1, the low-order two bits of *offset* are ignored. All bits of *offset* are meaningful on Storm-2.

The argument *buffer* should not be open. If a program calls **spi_load_block** with an open *buffer*, the runtime will cause the program to exit with a failure status in functional mode or debug mode.

6.1.7.3 Return value

None.

6.1.7.4 See also

spi_load_index, **spi_load_stride**, **spi_store_block**

6.1.7.5 Example

The diagrams below show the result of **spi_load_block(s, b, 0, 8)** on a 4-lane stream processor. Each box represents a record. *count* is 8, so **spi_load_block** loads 8 records. The record numbering corresponds to the order of records in memory.

Buffer **b** in memory:

0	1	2	3	4	5	6	7

Stream **s** in LRF (lanes shown horizontally, LRF address shown vertically):

Lanes

0	1	2	3
4	5	6	7

6.1.8 spi_load_index

6.1.8.1 Prototype

```
void
spi_load_index(
    stream type     str,                // Destination stream
    spi_buffer_t    buffer,             // Source data buffer
    unsigned int    offset,             // Byte offset from beginning of buffer
    stream int      indices,            // Index stream supplies byte offset for each group of records
    unsigned int    recs_per_lane,      // Number of successive records loaded to each lane in a group
    unsigned int    lanes_per_group,    // Number of successive lanes in a group: 0, 1, 2, 4, or 8
    unsigned int    count               // Number of records to load
);
```

6.1.8.2 Description

spi_load_index transfers data from records at the given *offset* in *buffer* to the local register file (LRF) of a stream processor. This allows DSP MIPS to pass input data to a kernel as an input stream. Index stream *indices* contains nonnegative byte offsets that specify the beginning of each group of records in memory. The indices are in bytes, not in words or in records. On Storm-1 (SP16 and SP8), any negative index produces an error at runtime. On Storm-2, an index of -1 loads 0, and any other negative index produces an error at runtime.

The value of *lanes_per_group* must be 0, 1, 2, 4, or 8. If *lanes_per_group* is 0, **spi_load_index** replicates each record in each lane of the LRF.

On Storm-1, if *lanes_per_group* is 1, *recs_per_lane* times the size of a record in words (i.e., **sizeof**(*type*) / 4) must be less than 2048; otherwise, *recs_per_lane* times the size of a record in words must be less than 16. These restrictions do not apply on Storm-2.

If *lanes_per_group* is nonzero, *count* / (*recs_per_lane* * *lanes_per_group*) gives the number of groups of records to transfer. On Storm-1, the size of index stream *indices* must be equal to the number of groups rounded up to the next multiple of **SPI_LANES**. On Storm-2, the size of index stream *indices* must be equal to the number of groups.

On Storm-1, if *count* is not a multiple of **SPI_LANES**, the contents of LRF locations immediately past the given *count* are undefined. On Storm-2, the contents of LRF locations past the given count remain unchanged.

On Storm-1, the low-order two bits of *offset* are ignored. All bits of *offset* are meaningful on Storm-2.

The argument *buffer* should not be open. If a program calls **spi_load_index** with an open *buffer*, the runtime will cause the program to exit with a failure status in functional mode or debug mode.

6.1.8.3 Return value

None.

6.1.8.4 See also

spi_load_2d_index, **spi_load_2d_index_crop**, **spi_load_block**, **spi_load_stride**, **spi_store_index**

6.1.8.5 Examples

The diagrams below show the result of **spi_load_index(s, b, 0, ix, 3, 2, 24)** with a record size of one word (four bytes) on a 4-lane stream processor. *count* is 24, so **spi_load_index** loads 24 records. *recs_per_lane* is 3, so in each group of records it stores 3 successive records to each lane. *lanes_per_group* is 2, so it stores records to 2 lanes in each group of records. Each box represents a record. The record numbering corresponds to the order of records in memory.

Index stream **ix** contains four values:

100	12	72	40

Assuming a record size of one word (four bytes), these indices correspond to records 25, 3, 18, and 10, respectively.

Buffer **b** in memory:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39

Here each group of records contains 6 records (*recs_per_lane* * *lanes_per_group* is 3 * 2).

Stream **s** in LRF (lanes shown horizontally, LRF address shown vertically):

Lanes

25	28	3	6
26	29	4	7
27	30	5	8
18	21	10	13
19	22	11	14
20	23	12	15

The next example uses a value of 0 for *lanes_per_group*, so it replicates records in each lane of the processor.
The diagrams below show the result of **spi_load_index(s, b, 0, ix, 3, 0, 48)** on a 4-lane stream processor with a record size of one word (four bytes). Each block represents a record. *count* is 48, so **spi_load_stride** loads 48 records.

Index stream **ix** contains four values:

64	12	52	40

Since the record size is one word (four bytes), these indices correspond to records 16, 3, 13, and 10, respectively.

Buffer **b** in memory:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Each group of records contains 3 records (*recs_per_lane* is 3).

Stream **s** in LRF (lanes shown horizontally, LRF address shown vertically):

Lanes

16	16	16	16
17	17	17	17
18	18	18	18
3	3	3	3

4	4	4	4
5	5	5	5
13	13	13	13
14	14	14	14
15	15	15	15
10	10	10	10
11	11	11	11
12	12	12	12

6.1.9 spi_load_stride

6.1.9.1 Prototype

```
void
spi_load_stride(
    stream type     str,                // Destination stream
    spi_buffer_t    buffer,             // Source data buffer
    unsigned int    offset,             // Byte offset from beginning of buffer
    unsigned int    stride,             // Stride in bytes between each group of records
    unsigned int    recs_per_lane,      // Number of successive records loaded to each lane in a group
    unsigned int    lanes_per_group,    // Number of successive lanes in a group: 0, 1, 2, 4, or 8
    unsigned int    count               // Number of records to load
);
```

6.1.9.2 Description

spi_load_stride transfers data records from the given *offset* in *buffer* to the local register file (LRF) of a stream processor. This allows DSP MIPS to pass input data to a kernel as an input stream. The given *stride* defines the byte offset between each group of transferred records in memory.

The value of *lanes_per_group* must be 0, 1, 2, 4, or 8. If *lanes_per_group* is 0, **spi_load_stride** replicates each record in each lane of the LRF.

On Storm-1 (SP16 and SP8), if *lanes_per_group* is 1, *recs_per_lane* times the size of a record in words (i.e., **sizeof**(*type*) / 4) must be less than 2048. Otherwise, *recs_per_lane* times the size of a record in words must be less than 16. These restrictions do not apply on Storm-2.

On Storm-1, the value of *stride* must be a multiple of 4 and must be less than 4096. These restrictions do not apply on Storm-2.

On Storm-1, if *count* is not a multiple of **SPI_LANES**, the contents of LRF locations immediately past the given *count* are undefined. On Storm-2, the contents of LRF locations past the given count remain unchanged.

On Storm-1, the low-order two bits of *offset* are ignored. All bits of *offset* are meaningful on Storm-2.

The argument *buffer* should not be open. If a program calls **spi_load_stride** with an open *buffer*, the runtime will cause the program to exit with a failure status in functional mode or debug mode.

6.1.9.3 Return value

None.

6.1.9.4 See also

spi_load_2d_stride, **spi_load_2d_stride_crop**, **spi_load_block**, **spi_load_index**, **spi_store_stride**

6.1.9.5 Examples

The diagrams below show the result of **spi_load_stride(s, b, 0, 10 * sizeof(***type***), 3, 2, 24)** on a 4-lane stream processor. *count* is 24, so **spi_load_stride** loads 24 records. *recs_per_lane* is 3, so it stores 3 successive records to each lane in

each group of records. *lanes_per_group* is 2, so it stores records to 2 lanes in each group of records. *stride* is 10 times the size of a record, so it strides 10 records between each group of records in memory. Each box represents a record. The record numbering corresponds to the order of records in memory.

Buffer **b** in memory:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39

Here each group of records contains 6 records (*recs_per_lane* * *lanes_per_group* is 3 * 2) but *stride* is 10 records, so **spi_load_stride** does not use the four records between each group.

Stream **s** in LRF (lanes shown horizontally, LRF address shown vertically):

Lanes

0	3	10	13
1	4	11	14
2	5	12	15
20	23	30	33
21	24	31	34
22	25	32	35

The next example uses a value of 0 for *lanes_per_group*, so it replicates records in each lane of the processor. The diagrams below show the result of **spi_load_stride(s, b, 0, 5 * sizeof(*type*), 3, 0, 48)** on a 4-lane stream processor. Each block represents a record. *count* is 48, so **spi_load_stride** loads 48 records.

Buffer **b** in memory:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Each group of records contains 3 records (*recs_per_lane* is 3) but *stride* is 5 records, so **spi_load_stride** does not use the two records between each group.

Stream **s** in LRF (lanes shown horizontally, LRF address shown vertically):

Lanes

0	0	0	0
1	1	1	1
2	2	2	2
5	5	5	5
6	6	6	6
7	7	7	7
10	10	10	10
11	11	11	11
12	12	12	12
15	15	15	15
16	16	16	16
17	17	17	17

6.1.10 spi_out

6.1.10.1 Prototype

type **spi_out**(*variable*);

6.1.10.2 Description

spi_out returns the value of *variable*, which must be a local variable of 32-bit integer type passed to a previously invoked kernel as a scalar **out** parameter. A variable that a Stream program uses as a scalar **out** parameter in a kernel call may only be used as an argument to **spi_out** or as an argument to another kernel call.

6.1.10.3 Return value

spi_out returns the value stored in *variable* by the kernel that it was passed to as an **out** parameter.

6.1.11 spi_print_stream_data

6.1.11.1 Prototype

void
spi_print_stream_data(
    stream *type str* // Stream from which to print data
);

6.1.11.2 Description

Debug function **spi_print_stream_data** prints the lane register file (LRF) data for stream *str*. It can be called from Stream code to show the contents of the LRF. The function prints stream data with an index and an offset; the index is the value used to refer to the data if the stream is an array stream, and the offset is the byte offset of the data from the start of the LRF.

6.1.11.3 Return value

None.

6.1.11.4 Example

```
#include "spi_spm.h"

int spi_main(int argc, char *argv[])
{
        int           i, *a;
        spi_buffer_t  buffer;
        stream int    my_stream(64);

        buffer = spi_buffer_new(64 * sizeof(int), 0, SPI_BUFFER_FLAG_NONE);
        a = (int *)spi_buffer_open(buffer, 0);
        for (i = 0; i < 64; ++i)
                a[i] = i;
        spi_buffer_close(buffer);
        spi_load_block(my_stream, buffer, 0, 64);
        spi_print_stream_data(my_stream);
        return 0;
}
```

This generates the following output:

```
Index Offset   Lane 0   Lane 1  ...  Lane 14  Lane 15
----- ------ -------- --------  ... -------- --------
    0     40        0        1  ...        e        f
    1     80       10       11  ...       1e       1f
    2     c0       20       21  ...       2e       2f
    3    100       30       31  ...       3e       3f
```

6.1.12 spi_store_2d_index

6.1.12.1 Prototype

```
void
spi_store_2d_index(                          // Storm-2 only
        stream type      str,                // Source stream
        spi_buffer_t     buffer,             // Destination data buffer
        unsigned int     offset,             // Byte offset from beginning of buffer
        unsigned int     width,              // Image width in bytes
        unsigned int     x_stride,           // Horizontal (X) stride in bytes
        unsigned int     blocks_per_group,   // Number of blocks to store per group
        stream int16x2   xy_indices,         // Coordinates for start of each group of blocks
        unsigned int     block_width,        // Block width in bytes
        unsigned int     block_height        // Block height in rows
);
```

6.1.12.2 Description

Storm-2 function **spi_store_2d_index** transfers data transfers data from the local register file (LRF) of a stream processor to the given *offset* in *buffer*. It is similar to **spi_store_index**, but with additional arguments that allow it to store rectangular portions of a two-dimensional (2d) image efficiently.

spi_store_2d_index stores 2d blocks to an image of the given *width*. It uses the current stream count (**spi_count**(*str*) for an ordinary stream or the substream length for a substream) to determine the number of records to store. Each block is a rectangle of dimensions *block_width* by *block_height*. If *block_width* is not a multiple of 4, **spi_store_2d_index** 0-pads each row of each block in the LRF to the next word multiple. The basetype *type* of stream *str* must represent a single *block_width* (plus padding if necessary) by *block_height* block.

Index stream *xy_indices* contains packed halfwords giving the x and y coordinates (in bytes, with x in the low-order halfword and y in the high-order halfword) of the beginning of each group of blocks. *x_stride* gives the horizontal stride in bytes between successive blocks in a horizontal group, and *blocks_per_group* gives the number of blocks in each group. Index (-1, -1) (that is, 0xFFFFFFFFp2) stores nothing; any other negative index produces a runtime error.

In pseudocode:

```
n = 0;
foreach (x, y) in  xy_indices {
        for (i = 0; i < blocks_per_group; i++) {
                if (x, y) == (-1, -1) {
                        skip block_width by block_height block from lane n; store nothing
                } else {
                        store block_width by block_height block from lane n to (x, y)
                        x += x_stride;
                }
                n = (n + 1) % SPI_LANES;
        }
}
```

offset may be any 32-bit value. *block_height* must be less than 256. All other parameters must be less than 65536.

The argument *buffer* should not be open. If a program calls **spi_store_2d_index** with an open *buffer*, the runtime will cause the program to exit with a failure status in functional mode or debug mode.

6.1.12.3 Return value

None.

6.1.12.4 See also

spi_count, **spi_load_2d_index**, **spi_store_2d_index_crop**, **spi_store_2d_stride**, **spi_store_index**

6.1.12.5 Example

The following example stores six 5 by 4 blocks from a 27 by 11 image.

```
struct b5x4 {
        uint8x4 row0_lo;        // low-order 4 bytes of row 0
        uint8x4 row0_hi;        // 5th byte of row 0 in low-order byte, 0-padding in higher bytes
        uint8x4 row1_lo;        // row 1
        uint8x4 row1_hi;
        uint8x4 row2_lo;        // row 2
        uint8x4 row2_hi;
        uint8x4 row3_lo;        // row 3
        uint8x4 row3_hi;
};
spi_buffer_t            buffer;
stream struct b5x4      s;
stream int16x2          xy_indices;
```

Suppose index stream *xy_indices* contains three entries, specifying coordinates (4, 1), (-1, -1), and (9, 6); that is, it contains 0x00010004p2, 0xFFFFFFFFp2, and 0x00060009p2.

Since the blocks are 5 bytes wide, each block row is 0-padded to 8 bytes (2 words) in the LRF. Suppose stream **s** in the LRF of a 4-lane stream processor contains (with each box representing one 4-byte word):

Lane	0	1	2	3
low address	0 **row0_lo**	1 **row0_lo**	2 **row0_lo**	3 **row0_lo**
	0 **row0_hi**	1 **row0_hi**	2 **row0_hi**	3 **row0_hi**
	0 **row1_lo**	1 **row1_lo**	2 **row1_lo**	3 **row1_lo**
	0 **row1_hi**	1 **row1_hi**	2 **row1_hi**	3 **row1_hi**
	0 **row2_lo**	1 **row2_lo**	2 **row2_lo**	3 **row2_lo**
	0 **row2_hi**	1 **row2_hi**	2 **row2_hi**	3 **row2_hi**
	0 **row3_lo**	1 **row3_lo**	2 **row3_lo**	3 **row3_lo**
	0 **row3_hi**	1 **row3_hi**	2 **row3_hi**	3 **row3_hi**
	4 **row0_lo**	5 **row0_lo**	6 **row0_lo**	7 **row0_lo**
	4 **row0_hi**	5 **row0_hi**	6 **row0_hi**	7 **row0_hi**
	4 **row1_lo**	5 **row1_lo**	6 **row1_lo**	7 **row1_lo**
	4 **row1_hi**	5 **row1_hi**	6 **row1_hi**	7 **row1_hi**
	4 **row2_lo**	5 **row2_lo**	6 **row2_lo**	7 **row2_lo**
	4 **row2_hi**	5 **row2_hi**	6 **row2_hi**	7 **row2_hi**
	4 **row3_lo**	5 **row3_lo**	6 **row3_lo**	7 **row3_lo**
	4 **row3_hi**	5 **row3_hi**	6 **row3_hi**	7 **row3_hi**
	8 **row0_lo**			
	8 **row0_hi**			
	8 **row1_lo**			
	8 **row1_hi**			
	8 **row2_lo**			
	8 **row2_hi**			
	8 **row3_lo**			
high address	8 **row3_hi**			

spi_store_2d_index stores nothing for the records shown in gray, as they correspond to an index stream entry of (-1, -1).

Then:

```
spi_store_2d_index(
        s,
        buffer,
        0,                      // byte offset from beginning of buffer
        27,                     // image width in bytes
        6,                      // x-stride between blocks in bytes
        3,                      // number of blocks in each group
        xy_indices,             // index stream
        5,                      // block width in bytes
        4                       // block height in rows
);
```

stores the stream **s** data from the LRF to **buffer** as follows, with each box representing one byte:

				0	0	0	0	0		1	1	1	1	1		2	2	2	2	2						
				0	0	0	0	0		1	1	1	1	1		2	2	2	2	2						
				0	0	0	0	0		1	1	1	1	1		2	2	2	2	2						
				0	0	0	0	0		1	1	1	1	1		2	2	2	2	2						
									6	6	6	6	6		7	7	7	7	7		8	8	8	8	8	
									6	6	6	6	6		7	7	7	7	7		8	8	8	8	8	
									6	6	6	6	6		7	7	7	7	7		8	8	8	8	8	
									6	6	6	6	6		7	7	7	7	7		8	8	8	8	8	

6.1.13 spi_store_2d_index_crop

6.1.13.1 Prototype

```
void
spi_store_2d_index_crop(                    // Storm-2 only
    stream type     str,                    // Source stream
    spi_buffer_t    buffer,                 // Destination data buffer
    unsigned int    offset,                 // Byte offset from beginning of buffer
    unsigned int    width,                  // Image width in bytes
    unsigned int    crop_width,             // Cropping width in bytes
    unsigned int    crop_height,            // Cropping height in rows
    unsigned int    x_stride,               // Horizontal (X) stride in bytes
    unsigned int    blocks_per_group,       // Number of blocks to store per group
    stream int16x2  xy_indices,             // Coordinates for start of each group of blocks
    unsigned int    block_width,            // Record width in bytes
    unsigned int    block_height            // Record height in rows
);
```

6.1.13.2 Description

Storm-2 function **spi_store_2d_index_crop** transfers data transfers data from the local register file (LRF) of a stream processor to the given *offset* in *buffer*. It is similar to **spi_store_2d_index**, but with additional arguments that allow it to crop the stored image blocks. The **spi_store_2d_index** page gives pseudocode that describes its operation.

spi_store_2d_index_crop stores 2d blocks to an image of the given *width*. It uses the current stream count (**spi_count**(*str*) for an ordinary stream or the substream length for a substream) to determine the number of records to store. Each block is a rectangle of dimensions *block_width* by *block_height*. If *block_width* is not a multiple of 4, **spi_store_2d_index_crop** 0-pads each row of each block in the LRF to the next word multiple. The basetype *type* of stream *str* must represent a single *block_width* (plus padding if necessary) by *block_height* block.

Index stream *xy_indices* contains packed halfwords giving the x and y coordinates (in bytes, with x in the low-order halfword and y in the high-order halfword) of the beginning of each group of blocks. *x_stride* gives the horizontal stride in bytes between successive blocks in a horizontal group, and *blocks_per_group* gives the number of blocks in each group. Index (-1, -1) (that is, 0xFFFFFFFFp2) stores nothing; any other negative index produces a runtime error.

crop_width and *crop_height* specify the dimensions of a cropping window relative to the given *offset*. **spi_store_2d_index_crop** stores nothing for any part of a block that falls outside the cropping rectangle.

offset may be any 32-bit value. *block_height* must be less than 256. All other parameters must be less than 65536.

The argument *buffer* should not be open. If a program calls **spi_store_2d_index_crop** with an open *buffer*, the runtime will cause the program to exit with a failure status in functional mode or debug mode.

6.1.13.3 Return value

None.

6.1.13.4 See also

spi_count, **spi_load_2d_index_crop**, **spi_store_2d_index**, **spi_store_2d_stride_crop**, **spi_store_index**

6.1.13.5 Example

The following example stores six 5 by 4 blocks from a 27 by 11 image.

```
struct b5x4 {
        uint8x4 row0_lo;        // low-order 4 bytes of row 0
        uint8x4 row0_hi;        // 5th byte of row 0 in low-order byte, 0-padding in higher bytes
        uint8x4 row1_lo;        // row 1
        uint8x4 row1_hi;
        uint8x4 row2_lo;        // row 2
        uint8x4 row2_hi;
        uint8x4 row3_lo;        // row 3
        uint8x4 row3_hi;
};
spi_buffer_t            buffer;
stream struct b5x4      s;
stream int16x2          xy_indices;
```

Suppose index stream *xy_indices* contains three entries, specifying coordinates (4, 1), (-1, -1), and (9, 6); that is, it contains 0x00010004p2, 0xFFFFFFFFp2, and 0x00060009p2.

Since the blocks are 5 bytes wide, each block row is 0-padded to 8 bytes (2 words) in the LRF. Suppose stream **s** in the LRF of a 4-lane stream processor contains (with each box representing one 4-byte word):

Lane	0	1	2	3
low address	0 **row0_lo**	1 **row0_lo**	2 **row0_lo**	3 **row0_lo**
	0 **row0_hi**	1 **row0_hi**	2 **row0_hi**	3 **row0_hi**
	0 **row1_lo**	1 **row1_lo**	2 **row1_lo**	3 **row1_lo**
	0 **row1_hi**	1 **row1_hi**	2 **row1_hi**	3 **row1_hi**
	0 **row2_lo**	1 **row2_lo**	2 **row2_lo**	3 **row2_lo**
	0 **row2_hi**	1 **row2_hi**	2 **row2_hi**	3 **row2_hi**
	0 **row3_lo**	1 **row3_lo**	2 **row3_lo**	3 **row3_lo**
	0 **row3_hi**	1 **row3_hi**	2 **row3_hi**	3 **row3_hi**
	4 **row0_lo**	5 **row0_lo**	6 **row0_lo**	7 **row0_lo**
	4 **row0_hi**	5 **row0_hi**	6 **row0_hi**	7 **row0_hi**
	4 **row1_lo**	5 **row1_lo**	6 **row1_lo**	7 **row1_lo**
	4 **row1_hi**	5 **row1_hi**	6 **row1_hi**	7 **row1_hi**
	4 **row2_lo**	5 **row2_lo**	6 **row2_lo**	7 **row2_lo**
	4 **row2_hi**	5 **row2_hi**	6 **row2_hi**	7 **row2_hi**
	4 **row3_lo**	5 **row3_lo**	6 **row3_lo**	7 **row3_lo**
	4 **row3_hi**	5 **row3_hi**	6 **row3_hi**	7 **row3_hi**
	8 **row0_lo**			
	8 **row0_hi**			
	8 **row1_lo**			
	8 **row1_hi**			
	8 **row2_lo**			
	8 **row2_hi**			
	8 **row3_lo**			

high address	8 **row3_hi**			

spi_store_2d_index_crop stores nothing for the records shown in gray, as they correspond to an index stream entry of (-1, -1).

Then:

```
spi_store_2d_index_crop(
        s,
        buffer,
        1 * 27 + 4,         // byte offset from beginning of buffer
        27,                 // image width in bytes
        16,                 // crop width in bytes
        7,                  // crop height in rows
        6,                  // x-stride between blocks in bytes
        3,                  // number of blocks in each group
        xy_indices,         // index stream
        5,                  // block width in bytes
        4                   // block height in rows
);
```

stores the stream **s** data from the LRF to **buffer** as follows, with each box representing one byte:

				0	0	0	0	0		1	1	1	1	1		2	2	2	2	2						
				0	0	0	0	0		1	1	1	1	1		2	2	2	2	2						
				0	0	0	0	0		1	1	1	1	1		2	2	2	2	2						
				0	0	0	0	0		1	1	1	1	1		2	2	2	2	2						
									6	6	6	6	6		7	7	7	7	7		8	8	8	8	8	
									6	6	6	6	6		7	7	7	7	7		8	8	8	8	8	
									6	6	6	6	6		7	7	7	7	7		8	8	8	8	8	
									6	6	6	6	6		7	7	7	7	7		8	8	8	8	8	

spi_store_2d_index_crop stores nothing for the records shown in gray, as they lie outside the cropping window.

6.1.14 spi_store_2d_stride

6.1.14.1 Prototype

```
void
spi_store_2d_stride(                        // Storm-2 only
    stream type     str,                    // Source stream
    spi_buffer_t    buffer,                 // Destination data buffer
    unsigned int    offset,                 // Byte offset from beginning of buffer
    unsigned int    width,                  // Image width in bytes
    unsigned int    x_stride,               // Horizontal (X) stride in bytes
    unsigned int    blocks_per_group,       // Number of blocks to store per group
    unsigned int    y_stride,               // Vertical (Y) stride in rows
    unsigned int    block_width,            // Record width in bytes
    unsigned int    block_height            // Record height in rows
);
```

6.1.14.2 Description

Storm-2 function **spi_store_2d_stride** transfers data from the local register file (LRF) of a stream processor to the given *offset* in *buffer* in DSP MIPS memory. It is similar to **spi_store_stride**, but with additional arguments that allow it to store rectangular portions of a two-dimensional (2d) image efficiently.

spi_store_2d_stride stores 2d blocks to an image of the given *width*. It uses the current stream count (**spi_count**(*str*) for an ordinary stream or the substream length for a substream) to determine the number of blocks to store.

Each block is a rectangle of dimensions *block_width* by *block_height*. If *block_width* is not a multiple of 4, **spi_store_2d_stride** 0-pads each row of each block in the LRF to the next word multiple. The basetype *type* of stream *str* must represent a single *block_width* (plus padding if necessary) by *block_height* record.

x_stride gives the horizontal stride in bytes between records stored from successive lanes, *blocks_per_group* gives the number of blocks in a horizontal group, and *y_stride* gives the vertical stride in rows between successive groups of records.

In pseudocode:

```
x0 = offset % width;
y = offset / width;
n = 0;
for (b = 0; b < count; b++) {
        x = x0;
        for (i = 0; i < blocks_per_group; i++) {
                store block_width by block_height block from lane n to (x, y)
                x += x_stride;
                n = (n + 1) % SPI_LANES;
        }
        y += y_stride;
}
```

offset may be any 32-bit value. *block_height* and *y_stride* must be less than 256. All other parameters must be less than 65536.

The argument *buffer* should not be open. If a program calls **spi_store_2d_stride** with an open *buffer*, the runtime will cause the program to exit with a failure status in functional mode or debug mode.

spi_store_2d_stride_crop is like **spi_store_2d_stride**, but with additional arguments that allow it to crop the stored image blocks.

6.1.14.3 Return value

None.

6.1.14.4 See also

spi_count, **spi_load_2d_stride**, **spi_store_2d_index**, **spi_store_2d_stride_crop**, **spi_store_stride**

6.1.14.5 Example

The following example stores ten 5 by 4 blocks into a 27 by 16 image.

```
struct b5x4 {
        uint8x4 row0_lo;        // low-order 4 bytes of row 0
        uint8x4 row0_hi;        // 5th byte of row 0 in low-order byte, 0-padding in higher bytes
        uint8x4 row1_lo;        // row 1
        uint8x4 row1_hi;
        uint8x4 row2_lo;        // row 2
        uint8x4 row2_hi;
        uint8x4 row3_lo;        // row 3
        uint8x4 row3_hi;
};
spi_buffer_t            buffer;
stream struct b5x4      s;
```

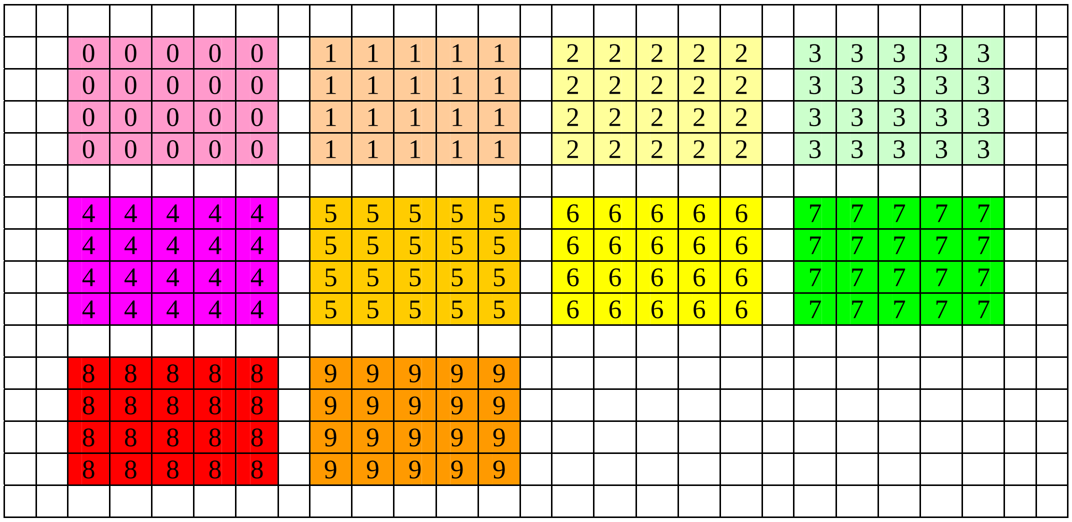
Since the blocks are 5 bytes wide, each block row is 0-padded to 8 bytes (2 words) in the LRF. Suppose stream **s** in the LRF of a 4-lane stream processor contains (with each box representing one 4-byte word):

Lane	0	1	2	3
low address	0 **row0_lo**	1 **row0_lo**	2 **row0_lo**	3 **row0_lo**
	0 **row0_hi**	1 **row0_hi**	2 **row0_hi**	3 **row0_hi**
	0 **row1_lo**	1 **row1_lo**	2 **row1_lo**	3 **row1_lo**
	0 **row1_hi**	1 **row1_hi**	2 **row1_hi**	3 **row1_hi**
	0 **row2_lo**	1 **row2_lo**	2 **row2_lo**	3 **row2_lo**
	0 **row2_hi**	1 **row2_hi**	2 **row2_hi**	3 **row2_hi**
	0 **row3_lo**	1 **row3_lo**	2 **row3_lo**	3 **row3_lo**
	0 **row3_hi**	1 **row3_hi**	2 **row3_hi**	3 **row3_hi**
	4 **row0_lo**	5 **row0_lo**	6 **row0_lo**	7 **row0_lo**
	4 **row0_hi**	5 **row0_hi**	6 **row0_hi**	7 **row0_hi**
	4 **row1_lo**	5 **row1_lo**	6 **row1_lo**	7 **row1_lo**
	4 **row1_hi**	5 **row1_hi**	6 **row1_hi**	7 **row1_hi**
	4 **row2_lo**	5 **row2_lo**	6 **row2_lo**	7 **row2_lo**
	4 **row2_hi**	5 **row2_hi**	6 **row2_hi**	7 **row2_hi**
	4 **row3_lo**	5 **row3_lo**	6 **row3_lo**	7 **row3_lo**
	4 **row3_hi**	5 **row3_hi**	6 **row3_hi**	7 **row3_hi**
	8 **row0_lo**	9 **row0_lo**		
	8 **row0_hi**	9 **row0_hi**		
	8 **row1_lo**	9 **row1_lo**		
	8 **row1_hi**	9 **row1_hi**		
	8 **row2_lo**	9 **row2_lo**		
	8 **row2_hi**	9 **row2_hi**		
	8 **row3_lo**	9 **row3_lo**		
high address	8 **row3_hi**	9 **row3_hi**		

Then:

```
spi_store_2d_stride(
        s,
        buffer,
        1 * 27 + 2,             // byte offset of first block
        27,                     // image width in bytes
        6,                      // x-stride between blocks in bytes
        4,                      // number of blocks before y-stride
        5,                      // y-stride between blocks
        5,                      // block width in bytes
        4                       // block height in rows
);
```

stores the stream **s** data from the LRF to **buffer** as follows, with each box representing one byte:



		0	0	0	0	0		1	1	1	1	1		2	2	2	2	2		3	3	3	3	3		
		0	0	0	0	0		1	1	1	1	1		2	2	2	2	2		3	3	3	3	3		
		0	0	0	0	0		1	1	1	1	1		2	2	2	2	2		3	3	3	3	3		
		0	0	0	0	0		1	1	1	1	1		2	2	2	2	2		3	3	3	3	3		
		4	4	4	4	4		5	5	5	5	5		6	6	6	6	6		7	7	7	7	7		
		4	4	4	4	4		5	5	5	5	5		6	6	6	6	6		7	7	7	7	7		
		4	4	4	4	4		5	5	5	5	5		6	6	6	6	6		7	7	7	7	7		
		4	4	4	4	4		5	5	5	5	5		6	6	6	6	6		7	7	7	7	7		
		8	8	8	8	8		9	9	9	9	9														
		8	8	8	8	8		9	9	9	9	9														
		8	8	8	8	8		9	9	9	9	9														
		8	8	8	8	8		9	9	9	9	9														

6.1.15 spi_store_2d_stride_crop

6.1.15.1 Prototype

```
void
spi_store_2d_stride_crop(                    // Storm-2 only
    stream type     str,                     // Source stream
    spi_buffer_t    buffer,                  // Destination data buffer
    unsigned int    offset,                  // Byte offset from beginning of buffer
    unsigned int    width,                   // Image width in bytes
    unsigned int    crop_width,              // Cropping width in bytes
    unsigned int    crop_height,             // Cropping height in rows
    unsigned int    x_stride,                // Horizontal (X) stride in bytes
    unsigned int    blocks_per_group,        // Number of blocks to store per group
    unsigned int    y_stride,                // Vertical (Y) stride in rows
    unsigned int    block_width,             // Record width in bytes
    unsigned int    block_height             // Record height in rows
);
```

6.1.15.2 Description

Storm-2 function **spi_store_2d_stride_crop** transfers data from the local register file (LRF) of a stream processor to the given *offset* in *buffer* in DSP MIPS memory. It is similar to **spi_store_2d_stride**, but with additional arguments that allow it to crop the stored image blocks. The **spi_store_2d_stride** page gives pseudocode that describes its operation.

spi_store_2d_stride_crop stores 2d blocks to an image of the given *width*. It uses the current stream count (**spi_count**(*str*) for an ordinary stream or the substream length for a substream) to determine the number of blocks to store.

Each block is a rectangle of dimensions *block_width* by *block_height*. If *block_width* is not a multiple of 4, **spi_store_2d_stride_crop** 0-pads each row of each block in the LRF to the next word multiple. The basetype *type* of stream *str* must represent a single *block_width* (plus padding if necessary) by *block_height* record.

x_stride gives the horizontal stride in bytes between records stored from successive lanes, *blocks_per_group* gives the number of blocks in a horizontal group, and *y_stride* gives the vertical stride in rows between successive groups of records.

crop_width and *crop_height* specify the dimensions of a cropping window relative to the given *offset*. **spi_store_2d_stride_crop** stores nothing for any part of a block that falls outside the cropping rectangle.

offset may be any 32-bit value. *block_height* and *y_stride* must be less than 256. All other parameters must be less than 65536.

The argument *buffer* should not be open. If a program calls **spi_store_2d_stride_crop** with an open *buffer*, the runtime will cause the program to exit with a failure status in functional mode or debug mode.

6.1.15.3 Return value

None.

6.1.15.4 See also

spi_count, **spi_load_2d_stride_crop**, **spi_store_2d_index_crop**, **spi_store_2d_stride**, **spi_store_stride**

6.1.15.5 Example

The following example stores ten 5 by 4 blocks into a 27 by 16 image, using a 16 by 12 cropping window.

```
struct b5x4 {
        uint8x4 row0_lo;        // low-order 4 bytes of row 0
        uint8x4 row0_hi;        // 5th byte of row 0 in low-order byte, 0-padding in higher bytes
        uint8x4 row1_lo;        // row 1
        uint8x4 row1_hi;
        uint8x4 row2_lo;        // row 2
        uint8x4 row2_hi;
        uint8x4 row3_lo;        // row 3
        uint8x4 row3_hi;
};
spi_buffer_t            buffer;
stream struct b5x4      s;
```

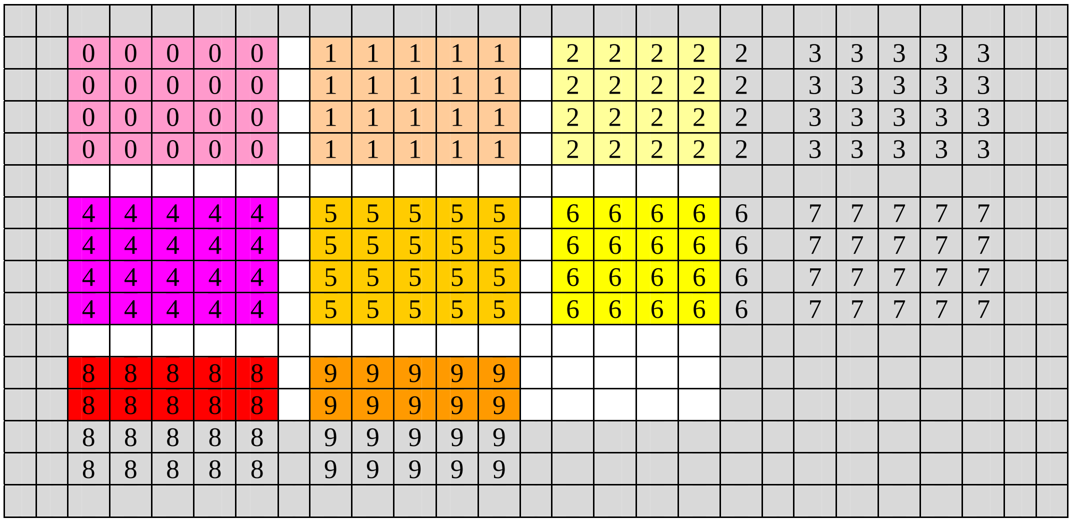
Since the blocks are 5 bytes wide, each block row is 0-padded to 8 bytes (2 words) in the LRF. Suppose stream **s** in the LRF of a 4-lane stream processor contains (with each box representing one 4-byte word):

Lane	0	1	2	3
low address	0 **row0_lo**	1 **row0_lo**	2 **row0_lo**	3 **row0_lo**
	0 **row0_hi**	1 **row0_hi**	2 **row0_hi**	3 **row0_hi**
	0 **row1_lo**	1 **row1_lo**	2 **row1_lo**	3 **row1_lo**
	0 **row1_hi**	1 **row1_hi**	2 **row1_hi**	3 **row1_hi**
	0 **row2_lo**	1 **row2_lo**	2 **row2_lo**	3 **row2_lo**
	0 **row2_hi**	1 **row2_hi**	2 **row2_hi**	3 **row2_hi**
	0 **row3_lo**	1 **row3_lo**	2 **row3_lo**	3 **row3_lo**
	0 **row3_hi**	1 **row3_hi**	2 **row3_hi**	3 **row3_hi**
	4 **row0_lo**	5 **row0_lo**	6 **row0_lo**	7 **row0_lo**
	4 **row0_hi**	5 **row0_hi**	6 **row0_hi**	7 **row0_hi**
	4 **row1_lo**	5 **row1_lo**	6 **row1_lo**	7 **row1_lo**
	4 **row1_hi**	5 **row1_hi**	6 **row1_hi**	7 **row1_hi**
	4 **row2_lo**	5 **row2_lo**	6 **row2_lo**	7 **row2_lo**
	4 **row2_hi**	5 **row2_hi**	6 **row2_hi**	7 **row2_hi**
	4 **row3_lo**	5 **row3_lo**	6 **row3_lo**	7 **row3_lo**
	4 **row3_hi**	5 **row3_hi**	6 **row3_hi**	7 **row3_hi**
	8 **row0_lo**	9 **row0_lo**		
	8 **row0_hi**	9 **row0_hi**		
	8 **row1_lo**	9 **row1_lo**		
	8 **row1_hi**	9 **row1_hi**		
	8 **row2_lo**	9 **row2_lo**		
	8 **row2_hi**	9 **row2_hi**		
	8 **row3_lo**	9 **row3_lo**		
high address	8 **row3_hi**	9 **row3_hi**		

Then:

```
spi_store_2d_stride_crop(
        s,
        buffer,
        1 * 27 + 2,             // byte offset of first block
        27,                     // image width in bytes
        16,                     // crop width in bytes
        12,                     // crop height in rows
        6,                      // x-stride between blocks in bytes
        4,                      // number of blocks before y-stride
        5,                      // y-stride between blocks
        5,                      // block width in bytes
        4,                      // block height in rows
        10                      // number of blocks to load
);
```

stores the stream **s** data from the LRF to **buffer** as follows, with each box representing one byte:



		0	0	0	0	0		1	1	1	1	1		2	2	2	2	2		3	3	3	3	3		
		0	0	0	0	0		1	1	1	1	1		2	2	2	2	2		3	3	3	3	3		
		0	0	0	0	0		1	1	1	1	1		2	2	2	2	2		3	3	3	3	3		
		0	0	0	0	0		1	1	1	1	1		2	2	2	2	2		3	3	3	3	3		
		4	4	4	4	4		5	5	5	5	5		6	6	6	6	6		7	7	7	7	7		
		4	4	4	4	4		5	5	5	5	5		6	6	6	6	6		7	7	7	7	7		
		4	4	4	4	4		5	5	5	5	5		6	6	6	6	6		7	7	7	7	7		
		4	4	4	4	4		5	5	5	5	5		6	6	6	6	6		7	7	7	7	7		
		8	8	8	8	8		9	9	9	9	9														
		8	8	8	8	8		9	9	9	9	9														
		8	8	8	8	8		9	9	9	9	9														
		8	8	8	8	8		9	9	9	9	9														

spi_store_2d_stride_crop stores nothing for the records shown in gray, as they lie outside the cropping window.

6.1.16 spi_store_block

6.1.16.1 Prototype

```
void
spi_store_block(
        stream type     str,            // Source stream
        spi_buffer_t    buffer,         // Destination data buffer
        unsigned int    offset          // Byte offset from beginning of buffer
);
```

6.1.16.2 Description

spi_store_block transfers data from the local register file (LRF) of a stream processor to the given *offset* in *buffer*. This allows DSP MIPS to access data written by a kernel to an output stream.

spi_store_block uses the current stream count (**spi_count**(*str*) for an ordinary stream or the substream length for a substream) to determine the number of records to store. On Storm-1 (SP16 and SP8), **spi_store_block** always stores data from each lane of the processor, so it rounds up the count to a multiple of **SPI_LANES** if necessary. On Storm-2, it stores the number of records given by the stream count.

On Storm-1, the low-order two bits of *offset* are ignored. All bits of *offset* are meaningful on Storm-2.

The argument *buffer* should not be open. If a program calls **spi_store_block** with an open *buffer*, the runtime will cause the program to exit with a failure status in functional mode or debug mode.

6.1.16.3 Return value

None.

6.1.16.4 See also

spi_count, **spi_load_block**, **spi_store_index**, **spi_store_stride**

6.1.16.5 Example

The diagrams below show the result of **spi_store_block(s, b, 0)** for a stream containing 8 records on a 4-lane stream processor. Each box represents a record. The record numbering corresponds to the order of the records loaded from the LRF.

Stream **s** in LRF (lanes shown horizontally, LRF address shown vertically):

Lanes

0	1	2	3
4	5	6	7

Buffer **b** in memory:

0	1	2	3	4	5	6	7

6.1.17 spi_store_index

6.1.17.1 Prototype

```
void
spi_store_index(
    stream type     str,              // Source stream
    spi_buffer_t    buffer,           // Destination data buffer
    unsigned int    offset,           // Byte offset from beginning of buffer
    stream int      indices,          // Index stream supplies byte offset for each group of records
    unsigned int    recs_per_lane,    // Number of successive records loaded from each lane in a group
    unsigned int    lanes_per_group   // Number of successive lanes in a group: 1, 2, 4, or 8
);
```

6.1.17.2 Description

spi_store_index transfers data transfers data from the local register file (LRF) of a stream processor to the given *offset* in *buffer*. This allows DSP MIPS to access data written by a kernel to an output stream. Index stream *indices* contains nonnegative byte offsets that specify the beginning of each group of records in memory. The indices are in bytes, not in words or in records. On Storm-1 (SP16 and SP8), any negative index produces a runtime error. On Storm-2, an index of -1 stores nothing, and any other negative index produces an error at runtime.

spi_store_index uses the current stream count (**spi_count**(*str*) for an ordinary stream or the substream length for a substream) to determine the number of records to store.

The value of *lanes_per_group* must be one of 1, 2, 4, or 8. On Storm-1, if *lanes_per_group* is 1, *recs_per_lane* times the size of a record in words (i.e., **sizeof**(*type*) / 4) must be less than 2048; otherwise, *recs_per_lane* times the size of a record in words must be less than 16. These restrictions do not apply on Storm-2.

spi_count(*str*) / (*recs_per_lane* * *lanes_per_group*) gives the number of groups of records to transfer. On Storm-1, the size of index stream *indices* must be equal to the number of groups rounded up to the next multiple of **SPI_LANES**. On Storm-2, the size of index stream *indices* must be equal to the number of groups.

On Storm-1, the low-order two bits of *offset* are ignored. All bits of *offset* are meaningful on Storm-2.

The argument *buffer* should not be open. If a program calls **spi_store_index** with an open *buffer*, the runtime will cause the program to exit with a failure status in functional mode or debug mode.

6.1.17.3 Return value

None.

6.1.17.4 See also

spi_count, **spi_load_index**, **spi_store_2d_index**, **spi_store_2d_index_crop**, **spi_store_block**, **spi_store_stride**

6.1.17.5 Examples

The diagrams below show the result of **spi_store_index(s, b, 0, ix, 3, 2)** for a stream **s** that contains 24 records with a record size of one word (four bytes) on a 4-lane stream processor. *recs_per_lane* is 3, so in each group of records **spi_store_index** loads 3 successive records from each lane. *lanes_per_group* is 2, so it loads records from 2 lanes in

each group of records. Each box represents a record. The record numbering corresponds to the order of records loaded from the LRF.

Index stream **ix** contains four values:

100	12	72	40

Assuming the record size is one word (four bytes), these indices designate records 25, 3, 18, and 10, respectively. Stream **s** in LRF (lanes shown horizontally, LRF address shown vertically):

Lanes

0	3	6	9
1	4	7	10
2	5	8	11
12	15	18	21
13	16	19	22
14	17	20	23

Buffer **b** in memory:

			6	7	8	9	10	11		18	19	20	21	22	23			12	13
14	15	16	17		0	1	2	3	4	5									

Here each group of records contains 6 records (*recs_per_lane* * *lanes_per_group* is 3 * 2).

6.1.18 spi_store_stride

6.1.18.1 Prototype

```
void
spi_store_stride(
    stream type     str,              // Source stream
    spi_buffer_t    buffer,           // Destination data buffer
    unsigned int    offset,           // Byte offset from beginning of buffer
    unsigned int    stride,           // Stride in bytes between each group of records
    unsigned int    recs_per_lane,    // Number of successive records from each lane in a group
    unsigned int    lanes_per_group   // Number of successive lanes in a group: 1, 2, 4, or 8
);
```

6.1.18.2 Description

spi_store_stride transfers data from the local register file (LRF) of a stream processor to the given *offset* in *buffer* in DSP MIPS memory. This allows DSP MIPS to access data written by a kernel to an output stream. The given *stride* defines the byte offset between each group of transferred data records in memory.

spi_store_stride uses the current stream count (**spi_count**(*str*) for an ordinary stream or the substream length for a substream) to determine the number of records to store.

The value of *lanes_per_group* must be one of 1, 2, 4, or 8. On Storm-1 (SP16 and SP8), if *lanes_per_group* is 1, *recs_per_lane* times the size of a record in words (i.e., **sizeof**(*type*) / 4) must be less than 2048; otherwise, *recs_per_lane* times the size of a record in words must be less than 16. These restrictions do not apply on Storm-2.

On Storm-1, the value of *stride* must be a multiple of 4 and must be less than 4096. These restrictions do not apply on Storm-2.

On Storm-1, the low-order two bits of *offset* are ignored. All bits of *offset* are meaningful on Storm-2.

The argument *buffer* should not be open. If a program calls **spi_store_stride** with an open *buffer*, the runtime will cause the program to exit with a failure status in functional mode or debug mode.

6.1.18.3 Return value

None.

6.1.18.4 See also

spi_count, **spi_load_stride**, **spi_store_2d_index**, **spi_store_2d_index_crop**, **spi_store_block**, **spi_store_index**

6.1.18.5 Example

The diagrams below show the result of **spi_store_stride(s, b, 0, 10 * sizeof(***type***), 3, 2)** for a stream with 24 records on a 4-lane stream processor. *recs_per_lane* is 3, so it takes 3 successive records from each lane in each group of records. *lanes_per_group* is 2, so it takes records from 2 lanes in each group of records. *stride* is 10 records, so it strides 10 records between each stored group of records. Each box represents a record. The record numbering corresponds to the order of the records loaded from the LRF.

Stream **s** in LRF (lanes shown horizontally, LRF address shown vertically):

Lanes

0	3	6	9
1	4	7	10
2	5	8	11
12	15	18	21
13	16	19	22
14	17	20	23

Buffer **b** in memory:

0	1	2	3	4	5					6	7	8	9	10	11				
12	13	14	15	16	17					18	19	20	21	22	23				

Each group of records contains 6 records (*recs_per_lane* * *lanes_per_group* is 3 * 2) but *stride* is 10 records, so **spi_store_stride** does not store into the four records which separate each group.

7 Kernel API

The Stream programming model Kernel API supports low-level data-parallel programming by allowing a DSP MIPS program to define and execute a kernel function that runs on the stream processor DPU. This chapter describes kernel functions, kernel library functions, and kernel intrinsic functions. Kernel library functions and kernel intrinsic functions may be used only within kernel functions. Kernel functions must be defined in **.sc** source files, not in **.c** files.

7.1 Kernel API library functions

Kernel library functions perform stream and array input (**spi_*read**) and output (**spi_*write**) and check for end of stream on an input stream (**spi_eos**). These functions may be used only within kernel code. A summary table in the Kernel API library functions summary section of this document lists each Kernel API library function.

The programmer should be aware of the relative efficiency of different stream read and write operations. Kernel library functions **spi_read**, **spi_cond_read** and **spi_array_read** all read data from a stream. Sequential read **spi_read** is more efficient than conditional read **spi_cond_read**, and in turn **spi_cond_read** is more efficient than array read **spi_array_read**. Similarly, **spi_write**, **spi_cond_write** and **spi_array_write** all write data into a stream. Sequential write **spi_write** is more efficient than conditional write **spi_cond_write**, and in turn **spi_cond_write** is more efficient than array write **spi_array_write.**

7.1.1 spi_array_read

7.1.1.1 Prototype

void
spi_array_read(

stream *type*	*str*,	// Array stream
vec *type*	*data*,	// Destination for read
vec int	*index*	// Record index into stream (may be different for each lane!)

);

7.1.1.2 Description

spi_array_read reads vector *data* of the given *type* from array input or input/output stream *str*. It is used to read data in a non-sequential order or to read the same data multiple times. Since *index* is a vector, a single **spi_array_read** call can read from different array indices in different lanes.

spi_array_read does not conform to ordinary C function call semantics: it stores returned data values through the supplied *data* argument, as if the parameter were passed by reference rather than by value. The compiler **spc** issues an error if the type of variable *data* does not match the type of the stream **str**.

By default, **spc** allocates 1 Kbyte per lane to hold local arrays for a kernel. To change the default value for a kernel, use the **local_array_size** pragma.

The programmer should be aware of the relative efficiency of different stream read operations. **spi_array_read** is less efficient than **spi_cond_read**, and in turn **spi_cond_read** is less efficient than **spi_read**.

7.1.1.3 Return value

None.

7.1.1.4 Example

```
kernel void mask_data(
     stream int32x1    filter(array_in),       // Array stream
     stream int32x1    data(seq_in),    // Sequential stream
     stream int32x1    result(seq_out), // Sequential stream
     int               max_index(in))
{
     int32x1           i;
     vec int32x1 t, f, s;

     while (!spi_eos(data))
     {
          s = 0;
          for (i = 0; i <= max_index; ++i)
          {
                spi_read(data, t);
                spi_array_read(filter, f, i);

                s = s + (t * f);
          }
          spi_write(result, s);
     }
}
```

7.1.2 spi_array_write

7.1.2.1 Prototype

void
spi_array_write(

stream *type*	*str*,	// Array stream
vec *type*	*data*,	// Data to write
vec int	*index*	// Index into stream

);

7.1.2.2 Description

spi_array_write writes vector *data* of the given *type* to array output or input/output stream *str*. It is used to write data in a non-sequential order. Since *index* is a vector, a single **spi_array_write** call can write to different array indices in different lanes. The compiler **spc** issues an error if the type of variable *data* does not match the type of the stream **str**.

By default, **spc** allocates 1 Kbyte per lane to hold local arrays for a kernel. To change the default value for a kernel, use the **local_array_size** pragma.

The programmer should be aware of the relative efficiency of different stream write operations. **spi_array_write** is less efficient than **spi_cond_write**, and in turn **spi_cond_write** is less efficient than **spi_write**.

7.1.2.3 Return value

None.

7.1.2.4 Example

```
kernel void mask_data
(
     stream int32x1    filter(array_in),       // Array stream
     stream int32x1    data(seq_in),    // Sequential stream
     stream int32x1    result(array_out),      // Array stream
     int               max_index(in)
)
{
     int32x1           i;
     vec int32x1 t, f, s;

     while (!spi_eos(data))
     {
          s = 0;
          for (i = 0; i <= max_index; ++i)
          {
                spi_read(data, t);
                spi_array_read(filter, f, i);

                s = s + (t * f);
          }
          spi_array_write(result, s, t);
     }
}
```


7.1.3 spi_cond_read

7.1.3.1 Prototype

void
spi_cond_read(
    stream *type* *str*, // Conditional stream
    vec *type* *data*, // Destination for read
    vec int32x1 *do_read* // Determines which lanes get data from stream
);

7.1.3.2 Description

spi_cond_read conditionally reads a record of the given *type* from conditional input stream *str* to destination *data*. It is used when only a subset of DPU lanes should receive new data from the stream.

The *do_read* parameter determines which lanes receive data from the stream. If the low-order bit of the value of *do_read* for a lane is non-zero, then the lane gets new data from the stream, otherwise **spi_cond_read** returns the last valid record read from the stream. It is up to the programmer to ignore the data in lanes where *do_read* is false. Note that the condition depends on whether bit 0 of *do_read* is non-zero, not on whether *do_read* is non-zero.

spi_cond_read does not conform to ordinary C function call semantics: it stores returned data values through the supplied *data* argument, as if the parameter were passed by reference rather than by value. The compiler **spc** issues an error if the type of variable *data* does not match the type of the stream **str**.

The programmer should be aware of the relative efficiency of different stream read operations. **spi_cond_read** is less efficient than **spi_read**, but it is more efficient than **spi_array_read**.

7.1.3.3 Return value

None.

7.1.3.4 Example

```
kernel void mask_data
(
     int32x1     mask(in),           // Input scalar data mask
     stream int  data(cond_in),      // Conditional input stream
     stream int  result(seq_out)     // Sequential output stream
)
{
     vec int32x1 temp;
     vec int32x1 do_read;

     do_read = spi_laneid() & 1;        // 0 in even lanes, 1 in odd lanes
     while (!spi_eos(data)) {
          spi_cond_read(data, temp, do_read);  // read in odd lanes
          if (do_read && temp > 0x80)
                temp = temp & mask;           // mask if greater than 0x80
          spi_write(result, temp);             // write unconditionally
     }
}
```

7.1.4 spi_cond_write

7.1.4.1 Prototype

```
void
spi_cond_write(
        stream type     str,        // Conditional stream
        vec type        data,       // Data to write
        vec int32x1     do_write    // Determines which lanes write data to stream
);
```

7.1.4.2 Description

spi_cond_write conditionally writes records of the given *type* from the given vector *data* to conditional output stream *str*. It is used when only a subset of DPU lanes should write to the stream. The compiler **spc** issues an error if the type of variable *data* does not match the type of the stream **str**.

The *do_write* parameter determines which lanes write data. **spi_cond_write** writes *data* to the output stream *str* for each lane in which the low-order bit of the value of *do_write* is non-zero. Note that the condition depends on whether bit 0 of *do_write* is non-zero, not on whether *do_write* is non-zero.

Since an output stream is simply a sequence of records, a record written from a given lane by **spi_cond_write** might be read into a different lane when another kernel subsequently reads the stream. For example, suppose a kernel writes 16 records on SP16 by calling **spi_cond_write** twice with a flag which selects only odd lanes and another kernel reads the data with a sequential (non-conditional) **spi_read**; the **spi_read** would read data records from lanes 1, 3, ..., 15, 1, 3, ..., 15 into lanes 0 through 15.

The programmer should be aware of the relative efficiency of different stream write operations. **spi_cond_write** is less efficient than **spi_write**, but it is more efficient than **spi_array_write**. The compiler **spc** issues an error if the type of destination variable *data* does not match the type of the stream **str**.

7.1.4.3 Return value

None.

7.1.4.4 Example

```
kernel void mask_data(
     int32x1     mask(in),           // Input scalar data mask
     stream int  data(seq_in), // Sequential input stream
     stream int  result(cond_out)    // Conditional output stream
) {
     vec int32x1 temp;
     vec int32x1 do_write;

     do_write = spi_laneid() & 1;       // 0 in even lanes, 1 in odd lanes
     while (!spi_eos(data)) {
          spi_read(data, temp);
          temp = temp & mask;
          spi_cond_write(result, temp, do_write); // write in odd lanes
     }
}
```


7.1.5 spi_eos

7.1.5.1 Prototype

int
spi_eos(// returns -1 if end of stream
stream *type str* // sequential or conditional stream
);

7.1.5.2 Description

spi_eos returns -1 (for true) or 0 (for false) to indicate whether the last data from input stream *str* of the given *type* has been read. **spi_eos** can only be used with sequential or conditional input streams. The number of items that can be read from a stream before **spi_eos** returns true is equal to the value returned by **spi_count**(*str*) prior to calling the kernel function or the length specified in the substream attribute for the stream; it is not the size specified in the stream declaration.

7.1.5.3 Return value

Returns -1 (true) if no more data is available in a sequential stream; returns 0 (false) otherwise. Note that the true return value is -1, not 1.

7.1.5.4 Example

```
kernel void mask_data
(
     int32x1            mask(in),               // Input scalar data mask
     stream int32x1     data_in(seq_in),        // Input sequential stream
     stream int32x1     data_in(seq_out),       // Output sequential stream
     int32x1            count(out)              // Output scalar value
)
{
     vec int32x1 temp;

     count = 0;
     while (!spi_eos(data))
     {
          spi_read(data_in, temp);
          if (temp != 0) temp = temp & mask;
          count++;
          spi_write(data_out, temp);
     }
}
```

7.1.6 spi_printf

7.1.6.1 Prototype

int
spi_printf(
    const char * *format*, ...
);

7.1.6.2 Description

In functional mode, **spi_printf** writes formatted output to the standard output; it does nothing in other modes. **spi_printf** may be used within a kernel to print debugging information. As with standard **printf**, the given *format* specifies the format of the output, and trailing arguments in the argument list correspond to the conversion specifiers in *format*. **spi_printf** can perform most standard **printf** conversions.

spi_printf can print packed data and vector data. It prints packed data as 4 or 2 '|'-separated items, with each item unpacked to a 32-bit value. It prints vector data as **SPI_LANES** space-separated items.

Each conversion specifier in *format* is of the form:

%[*flags*][*width*][.*precision*][*length*]*specifier*

The *flags*, *width*, and *precision* tags are identical to the equivalent standard **printf** tags. Each *specifier* must be one of the tags listed below. **spi_printf** does not support standard **printf** specifiers **n**, **p**, or **s**.

c	Character
d or **i**	Signed decimal integer
e	Scientific notation using **e**
E	Scientific notation using **E**
f	Decimal floating point
g	Use shorter of **%e** or **%f**
G	Use shorter of **%E** or **%f**
o	Signed octal
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (capital letters)
%	**%%** produces a single **%** character

If present, the optional *length* must be one of the tags listed below. **spi_printf** does not support standard **printf** length tags **h**, **l**, or **L**.

b	Scalar packed bytes	**int8x4** or **uint8x4**
h	Scalar packed halfwords	**int16x2** or **uint16x2**
v	Vector of words	**vec int32x1** or **vec uint32x1**
vb	Vector of packed bytes	**vec int8x4** or **vec uint8x4**
vh	Vector of packed halfwords	**vec int16x2** or **vec uint16x2**

7.1.6.3 Return value

spi_printf returns the number of characters printed.

7.1.6.4 Example

```
#include "spi_spm.h"

kernel void k(
     int32x1            i(in),
     stream int32x1     s(seq_out))
{
     vec int32x1 vi;
     int8x4      b;

     vi = (vec int32x1)i;
     b = (int8x4)i;
     spi_printf("i=%x\n", i);
     spi_printf("i=0x%x %d 0%o\n", i, i, i);
     spi_printf("vec vi: %06vx\n", vi);
     spi_printf("packed b: %02bx\n", b);
     spi_write(s, vi);
}
```

7.1.7 spi_read

7.1.7.1 Prototype

void
spi_read(
    stream *type* *str*, // Sequential stream
    vec *type* *data* // Destination for read
);

7.1.7.2 Description

spi_read reads one record of vector *data* of the given *type* from sequential input stream *str*.

spi _read does not conform to ordinary C function call semantics: it stores returned data values through the supplied *data* argument, as if the parameter were passed by reference rather than by value. The compiler **spc** issues a warning if the type of variable *data* does not match the type of the stream **str**.

The programmer should be aware of the relative efficiency of different stream read operations. **spi_read** is more efficient than **spi_cond_read**, and in turn **spi_cond_read** is more efficient than **spi_array_read.**

The compiler **spc** generates special code for conditionals that contain only **spi_read** or **spi_write** calls, such as:

```
if (<scalar_expression>) { spi_read(s, vi); }
```

The generated code does not contain a branch, so this code may be used within a software pipelined inner loop.

7.1.7.3 Return value

None.

7.1.7.4 Example

```
kernel void mask_data(
     int32x1            mask(in),               // Input scalar data mask
     stream int32x1     data_in(seq_in), // Input sequential stream
     stream int32x1     data_in(seq_out),       // Output sequential stream
     int32x1            count(out)              // Returned scalar value
) {
     vec int32x1 temp;

     count = 0;
     while (!spi_eos(data))
     {
          spi_read(data_in, temp);
          temp = temp & mask;
          spi_write(data_out, temp);
          count++;
     }
}
```


7.1.8 spi_write

7.1.8.1 Prototype

void
spi_write(
    stream *type* *str*, // Sequential stream
    vec *type* *data* // Pointer to storage for result of read
);

7.1.8.2 Description

spi_write writes one record of vector *data* of the given *type* to sequential output stream *str*. The compiler **spc** issues a warning if the type of variable *data* does not match the type of the stream **str**.

The programmer should be aware of the relative efficiency of different stream write operations. **spi_write** is more efficient than **spi_cond_write**, and in turn **spi_cond_write** is more efficient than **spi_array_write.**

The compiler **spc** generates special code for conditionals that contain only **spi_read** or **spi_write** calls, such as:

```
if (<scalar_expression>) { spi_write(s, vi); }
```

The generated code does not contain a branch, so this code may be used within a software pipelined inner loop.

7.1.8.3 Return value

None.

7.1.8.4 Example

```
kernel void mask_data
(
     int32x1           mask,            // Input scalar data mask
     stream int32x1    data_in(seq_in), // Input sequential stream
     stream int32x1    data_in(seq_out),       // Output sequential stream
     int32x1           *count           // Returned scalar value
)
{
     vec int32x1 temp;

     count = 0;
     while (!spi_eos(data))
     {
          spi_read(data_in, temp);
          if (temp != 0) temp = temp & mask;
          spi_write(data_out, temp);
          count++;
     }
}
```

7.2 Kernel API intrinsic operations

Intrinsic operations represent operations provided by stream processor DPU hardware. Stream programs can use intrinsic operations *only* within kernel functions. A summary table in the Kernel API intrinsic functions summary section of this document lists each Kernel API intrinsic operation.

A single page below describes all the variants of an operation; for example, the **spi_vadd** page describes all available variants of the vector addition intrinsics **spi_vadd***. Prefix **spi_** identifies an intrinsic as an SPI-specific operation; **v** indicates that the intrinsic's arguments are vectors (not scalars); a memonic (e.g., **add** for addition here) identifies the operation; and a width suffix (e.g., **32i**, **32u, 32ui**, **16i**, **16u**, **8i**, or **8u**) identifies a width and signedness variant of the intrinsic.

In most cases, the description of an intrinsic does not explictly describe its signedness and width variant semantics. For example, the **spi_vadd** page simply says "Each **spi_vadd*** intrinsic returns the sum of its arguments." This implies (but does not explicitly say) that the signed **32i**, **16i** and **8i** variants use signed arithmetic to perform the addition, while the unsigned **32u**, **16u** and **8u** variants use unsigned arithmetic. Similarly, it implies (but does not explicitly say) that the packed halfword **16i** and **16u** variants perform two separate 16-bit sums and return a packed result, and that the packed byte **8i** and **8u** variants perform four separate 8-bit sums and return a packed result.

Similarly, in most cases, the description of an intrinsic that returns a vector result does not explicitly describe its vector semantics (namely, that it performs the same operation in each lane). For example, the **spi_vadd** description implies (but does not explicitly say) that it performs a separate sum (or multiple separate sums, for packed variants) in each lane of the processor. The Examples section shows the operation's sample arguments and result for a single lane only.

Some DPU hardware operations return two values; for example, hardware operation ADDC32 returns a 32-bit sum and a 32-bit carry. These operations have two corresponding intrinsic functions (e.g., **spi_vaddc32**, which returns a sum, and **spi_vaddc32_c**, which returns a carry); the compiler **spc** merges paired calls to the intrinsics into a single hardware operation for efficiency.

Storm-2 hardware supports all Storm-1intrinsic operations plus a few additional intrinsics. Except for the operations specificially described as Storm-2 only, intrinsic operations work on both Storm-1 and Storm-2 hardware.

7.2.1 Cycle counts

Each DPU hardware operation requires a fixed number of cycles, called the *latency* of the operation. The table below gives cycle counts for each intrinsic operation. The application programmer generally does not need to be concerned with cycle counts because the Stream compiler schedules operations, but for performance optimization it can be useful for the programmer to know cycle counts.

spi_add	3	**spi_vclip**	3	**spi_vor**	2
spi_and	2	**spi_vdivstep**	3	**spi_vperm**	5
spi_eq	2	**spi_vdotna**	6	**spi_vrandl**	5
spi_laneid	7*	**spi_vdotp**	6	**spi_vrandlv**	5
spi_le	3	**spi_vdotpa**	6	**spi_vrorl**	5
spi_lt	3	**spi_veq**	2	**spi_vrorlv**	5
spi_ne	2	**spi_vffone**	3	**spi_vsad**	6
spi_not	2	**spi_vle**	3	**spi_vsclip**	3
spi_or	2	**spi_vlsbs**	5	**spi_vselect**	2
spi_perm	5	**spi_vlt**	3	**spi_vselectd**	2
spi_select	2	**spi_vmax**	3	**spi_vshift**	3
spi_shift	3	**spi_vmin**	3	**spi_vshifta**	3
spi_shifta	3	**spi_vmula**	6	**spi_vshuffle**	2
spi_shuffle	2	**spi_vmuld**	6	**spi_vshuffled**	2
spi_shuffled	2	**spi_vmulha32**	6	**spi_vsub**	3
spi_sub	3	**spi_vmulha16**	6	**spi_vsubc**	3
spi_vabd	3	**spi_vmulla32**	6	**spi_vsubs**	3
spi_vadd	3	**spi_vmulra**	6	**spi_vsuma**	6
spi_vaddc	3	**spi_vne**	2	**spi_vxor**	2
spi_vadds	3	**spi_vnorm**	3	**spi_xor**	2
spi_vand	2	**spi_vnot**	2		

(*) The first **spi_laneid** in a kernel takes 7 cycles, but subsequent calls take 0 cycles.

7.2.2 spi_add

7.2.2.1 Prototype

int32x1 spi_add32i(int32x1 a, int32x1 b);

7.2.2.2 Description

Intrinsic **spi_add32i** (addition) returns the sum of its arguments. The DPU supports scalar addition of signed words only.

7.2.2.3 Return value

Returns **a** + **b**.

7.2.2.4 See also

spi_sub: like **spi_add**, but uses subtraction rather than addition.
spi_vadd: like **spi_add**, but for vector arguments rather than scalars.

7.2.2.5 Examples

Op	**a**	**b**	Result
spi_add32i	0xFFFFFFFF	0x00000003	0x00000002

7.2.3 spi_and

7.2.3.1 Prototypes

uint32x1 spi_and32(uint32x1 a, uint32x1 b);

uint16x2 spi_and16(uint16x2 a, uint16x2 b);

uint8x4 spi_and8(uint8x4 a, uint8x4 b);

7.2.3.2 Description

Each **spi_and*** (AND) intrinsic returns the bitwise AND of its arguments. The arguments may be scalar words, packed halfwords, or packed bytes.

spi_and32, **spi_and16** and **spi_and8** all represent the same DPU hardware operation. They have different names to allow strict type checking of arguments and result.

7.2.3.3 Return value

Each return value bit is 1 if the corresponding bit in both arguments is 1, otherwise it is 0.

7.2.3.4 See also

spi_not : like **spi_and**, but uses bitwise NOT rather than bitwise AND.
spi_or: like **spi_and**, but uses bitwise OR rather than bitwise AND.
spi_vand : like **spi_and**, but for vector arguments rather than scalars.
spi_xor: like **spi_and**, but uses bitwise XOR rather than bitwise AND.

7.2.3.5 Examples

Op	**a**	**b**	Result
spi_and32	0x12345678	0x00FF00FF	0x00340078
spi_and16	0x12345678	0x00FF00FF	0x00340078
spi_and8	0x12345678	0x00FF00FF	0x00340078

7.2.4 spi_eq

7.2.4.1 Prototypes

int32x1 **spi_eq32(uint32x1 a, uint32x1 b);**

int16x2 **spi_eq16(uint16x2 a, uint16x2 b);**

int8x4 **spi_eq8(uint8x4 a, uint8x4 b);**

7.2.4.2 Description

Each **spi_eq*** (equals) intrinsic returns the boolean value true (all 1 bits) if **a** equals **b** and the boolean value false (all 0 bits) otherwise. The arguments may be scalar words, packed halfwords, or packed bytes.

In ANSI C, the result of the equality operator '==' is 0 (false) or 1 (true). Because the DPU hardware returns all 1 bits (that is, 0xFF, 0xFFFF or 0xFFFFFFFF) rather than 1 for true, the result of arithmetic using the result of **spi_eq*** may differ from the result in ANSI C.

7.2.4.3 Return value

Returns true (all 1 bits) if **a** is equal to **b** and 0 if not.

7.2.4.4 See also

spi_le: like **spi_eq**, but uses less than or equal to rather than equals.
spi_lt: like **spi_eq**, but uses less than rather than equals.
spi_ne: like **spi_eq**, but uses not equals rather than equals.
spi_veq: like **spi_eq**, but for vector arguments rather than scalars.

7.2.4.5 Examples

Op	**a**	**b**	Result
spi_eq32i	0x00000002	0x00000002	0xFFFFFFFF
spi_eq32i	0x00000002	0xFFFFFFFF	0x00000000
spi_eq16i	0x00010002	0x0001FFFF	0xFFFF0000
spi_eq8i	0x01020304	0x01FFFF04	0xFF0000FF

7.2.5 spi_laneid

7.2.5.1 Prototype

vec int32x1 spi_laneid(void);

7.2.5.2 Description

Intrinsic **spi_laneid** (lane id) returns the lane number of each lane.

The first **spi_laneid** in a kernel takes 7 cycles, but subsequent calls take 0 cycles.

7.2.5.3 Return value

Returns 0 in lane 0, 1 in lane 1, etc.

7.2.5.4 Examples

Op	Result lane 0	Result lane 1	**...**	Result lane 14	Result lane 15
spi_laneid	0x00000000	0x00000001	...	0x0000000E	0x0000000F

7.2.6 spi_le

7.2.6.1 Prototype

int32x1 spi_le32i(int32x1 a, int32x1 b);

7.2.6.2 Description

Intrinsic **spi_le32i** (less than or equal to) returns the boolean value true (all 1 bits) if **a** is less than or equal to **b** and the boolean value false (all 0 bits) otherwise. The DPU supports the scalar version of this operation only for signed words.

In ANSI C, the result of the less than or equal to operator '<=' is 0 (false) or 1 (true). Because the DPU hardware returns all 1 bits (0xFFFFFFFF) rather than 1 for true, the result of arithmetic using the result of **spi_le32i** may differ from the result in ANSI C.

7.2.6.3 Return value

Returns true (all 1 bits) if **a** is less than or equal to **b** and 0 if not.

7.2.6.4 See also

spi_eq: like **spi_le**, but returns equal to rather than less than or equal to.
spi_lt: like **spi_le**, but returns strictly less than rather than less than or equal to.
spi_ne: like **spi_le**, but returns not equal to rather than less than or equal to.
spi_vle: like **spi_le**, but for vector arguments rather than scalars.

7.2.6.5 Examples

Op	**a**	**b**	Result
spi_le32i	0x00000002	0x00000002	0xFFFFFFFF
spi_le32i	0x00000002	0x00000003	0xFFFFFFFF
spi_le32i	0x00000002	0xFFFFFFFF	0x00000000

7.2.7 spi_lt

7.2.7.1 Prototype

int32x1 spi_lt32i(int32x1 a, int32x1 b);

7.2.7.2 Description

Intrinsic **spi_lt32i** (less than) returns the boolean value true (all 1 bits) if **a** is less than **b** and the boolean value false (all 0 bits) otherwise. The DPU supports the scalar version of this operation only for signed words.

In ANSI C, the result of the less than operator '<' is 0 (false) or 1 (true). Because the DPU hardware returns all 1 bits (0xFFFFFFFF) rather than 1 for true, the result of arithmetic using the result of **spi_lt32i** may differ from the result in ANSI C.

7.2.7.3 Return value

Returns true (all 1 bits) if **a** is less than **b** and 0 if not.

7.2.7.4 See also

spi_eq: like **spi_lt**, but returns equal to rather than strictly less than.
spi_le: like **spi_lt**, but returns less than or equal to rather than strictly less than.
spi_ne: like **spi_lt**, but returns not equal to rather than strictly less than.
spi_vlt: like **spi_lt**, but for vector arguments rather than scalars.

7.2.7.5 Examples

Op	**a**	**b**	Result
spi_lt32i	0x00000002	0x00000002	0x00000000
spi_lt32i	0x00000002	0xFFFFFFFF	0x00000000

7.2.8 spi_ne

7.2.8.1 Prototypes

int32x1 **spi_ne32(uint32x1 a, uint32x1 b);**

int16x2 **spi_ne16(uint16x2 a, uint16x2 b);**

int8x4 **spi_ne8(uint8x4 a, uint8x4 b);**

7.2.8.2 Description

Each **spi_ne*** (not equals) intrinsic returns the boolean value true (all 1 bits) if **a** is not equal to **b** and the boolean value false (all 0 bits) if **a** is equal to **b**. The arguments may be words, packed halfwords, or packed bytes.

In ANSI C, the result of the inequality operator '!=' is 0 (false) or 1 (true). Because the DPU hardware returns all 1 bits (that is, 0xFF, 0xFFFF or 0xFFFFFFFF) rather than 1 for true, the result of arithmetic using the result of **spi_ne*** may differ from the result of '!=' in ANSI C.

7.2.8.3 Return value

Returns true (all 1 bits) if **a** is not equal to **b** and 0 if not.

7.2.8.4 See also

spi_eq: like **spi_ne**, but uses equals rather than not equals**.**
spi_le: like **spi_ne**, but uses less than or equal to rather than not equals**.**
spi_lt: like **spi_ne**, but uses less than rather than not equals**.**
spi_vne: like **spi_ne**, but for vector arguments rather than scalars.

7.2.8.5 Examples

Op	**a**	**b**	Result
spi_ne32	0x00000002	0x00000002	0x00000000
spi_ne32	0x00000002	0xFFFFFFFF	0xFFFFFFFF
spi_ne16	0x00010002	0x0001FFFF	0x0000FFFF
spi_ne8	0x01020304	0x01FFFF04	0x00FFFF00

7.2.9 spi_not

7.2.9.1 Prototypes

uint32x1 spi_not32(uint32x1 a);

uint16x2 spi_not16(uint16x2 a);

uint8x4 spi_not8(uint8x4 a);

7.2.9.2 Description

Each **spi_not*** (NOT) intrinsic returns the bitwise complement of its argument. The argument may be a word, packed halfwords, or packed bytes.

spi_not32, **spi_not16** and **spi_not8** all represent the same DPU hardware operation. They have different names to allow strict type checking of arguments and result.

7.2.9.3 Return value

Each return value bit is 1 if the corresponding bit in **a** is 0, otherwise it is 1.

7.2.9.4 See also

spi_and: like **spi_not**, but uses bitwise AND rather than bitwise NOT.
spi_or: like **spi_not**, but uses bitwise OR rather than bitwise NOT.
spi_vnot: like **spi_not**, but for a vector argument rather than a scalar.
spi_xor: like **spi_not**, but uses bitwise XOR rather than bitwise NOT.

7.2.9.5 Examples

Op	**a**	Result
spi_not32	0x12345678	0xEDCBA987
spi_not16	0x12345678	0xEDCBA987
spi_not8	0x12345678	0xEDCBA987

7.2.10 spi_or

7.2.10.1 Prototypes

uint32x1 **spi_or32(uint32x1 a, uint32x1 b);**

uint16x2 **spi_or16(uint16x2 a, uint16x2 b);**

uint8x4 **spi_or8(uint8x4 a, uint8x4 b);**

7.2.10.2 Description

Each **spi_or*** (OR) intrinsic returns the bitwise OR of its arguments. The arguments may be words, packed halfwords, or packed bytes.

spi_or32, **spi_or16** and **spi_or8** all represent the same DPU hardware operation. They have different names to allow strict type checking of arguments and result.

7.2.10.3 Return value

Each return value bit is 1 if the corresponding bit in either argument is 1, otherwise it is 0.

7.2.10.4 See also

spi_and: like **spi_or**, but uses bitwise AND rather than bitwise OR.
spi_not: like **spi_or**, but uses bitwise NOT rather than bitwise OR.
spi_vor: like **spi_or**, but for vector arguments rather than scalars.
spi_xor: like **spi_or**, but uses bitwise XOR rather than bitwise OR.

7.2.10.5 Examples

Op	**a**	**b**	Result
spi_or32	0x12345678	0x00FF00FF	0x12FF56FF
spi_or16	0x12345678	0x00FF00FF	0x12FF56FF
spi_or8	0x12345678	0x00FF00FF	0x12FF56FF

7.2.11 spi_perm

7.2.11.1 Prototypes

uint32x1 **spi_perm32(uint32x1 a, vec uint32x1 b, uint32x1 c);**

uint16x2 **spi_perm16(uint16x2 a, vec uint16x2 b, uint16x2 c);**

uint8x4 **spi_perm8(uint8x4 a, vec uint8x4 b, uint8x4 c);**

7.2.11.2 Description

Each **spi_perm*** (permute) intrinsic permutes data between the lanes of the stream processor and the scalar operand register file (SORF). Argument **a** is a scalar control value, while **b** is vector data and **c** is scalar data.

A single hardware operation performs both **spi_perm*** and the corresponding **spi_vperm*,** returning both a scalar and a vector result. The compiler **spc** merges adjacent paired calls to these functions for efficiency.

7.2.11.3 Return value

For **spi_perm32**, control value **a** specifies a lane or scalar selector *n*. If *n* is between 0 and 15, **spi_perm32** returns the value of vector **b** in lane ***n*.** If *n* is 16, **spi_perm32** returns scalar **c**. If *n* is greater than 16, the behavior is undefined.

For **spi_perm16**, bit 0 of control value **a** specifies a halfword (0 or 1) *m* and bits 5:1 of **a** specify a lane (0 to 15) or scalar (16) *n*. If *n* is between 0 and 15, **spi_perm16** returns the value of halfword *m* of vector **b** in lane ***n*.** If *n* is 16, **spi_perm16** returns the value of halfword *m* of scalar **c**. If *n* is greater than 16, the behavior is undefined.

For **spi_perm8**, bits 1:0 of **a** specifies a byte (0 to 3) *m* and bits 6:2 of **a** specify a lane (0 to 15) or scalar (16) *n*. If *n* is between 0 and 15, **spi_perm8** returns the value of byte *m* of vector **b** in lane ***n*.** If *n* is 16, **spi_perm8** returns the value of byte *m* of scalar **c**. If *n* is greater than 16, the behavior is undefined.

7.2.11.4 Notes

To extract a scalar value from a vector, use **spi_perm32:**

```
#define     spi_extract_scalar(v, i)      spi_perm32(i, v, 0);
a = spi_extract_scalar(v, 5); // extract scalar value from lane 5 of vec v
```

7.2.11.5 See also

spi_shuffle: like **spi_perm**, but shuffles bytes within a lane rather than permuting data between lanes.
spi_vperm: like **spi_perm**, but takes a vector control value and returns a vector result rather than a scalar.

7.2.11.6 Examples

Op	**a**	**b** lane 0	**b** lane 1	**...**	**b** lane 15	**c**	Result
spi_perm32	0x00000001	0x03020100	0x13121110	...	0xF3F2F1F0	0xDEADBEEF	0x13121110
spi_perm32	0x00000010	0x03020100	0x13121110	...	0xF3F2F1F0	0xDEADBEEF	0xDEADBEEF
spi_perm16	0x00030021	0x03020100	0x13121110	...	0xF3F2F1F0	0xDEADBEEF	0x1312DEAD
spi_perm8	0x00053E43	0x03020100	0x13121110	...	0xF3F2F1F0	0xDEADBEEF	0x0011F2DE

The following diagram illustrates the **spi_perm8** example from the table above. **spi_perm8** arguments and result are of type **uint8x4**, i.e., four packed unsigned bytes. The diagram shows each byte of each value separately.

Control argument **a**, with each byte also given in binary to show the lane (bits 6:2) and component (bits 1:0) it selects:

a:	00	05	3E	43
Binary	0.00000.00	0.00001.01	0.01111.10	0.10000.11
Lane	0	1	15	Scalar (16)
Component	0	1	2	3

Vector argument **b** and scalar argument **c,** with colors indicating the lanes and components selected by control argument **a**:

	Lane 0				Lane 1				...	Lane 15					Scalar			
b:	03	02	01	00	13	12	11	10	...	F3	F2	F1	F0	**c**:	DE	AD	BE	EF

The result:

Result:	00	11	F2	DE

7.2.12 spi_select

7.2.12.1 Prototypes

uint32x1 **spi_select32(uint32x1 a, uint32x1 b, uint32x1 c);**

uint16x2 **spi_select16(uint32x1 a, uint16x2 b, uint16x2 c);**

uint8x4 **spi_select8(uint32x1 a, uint8x4 b, uint8x4 c);**

7.2.12.2 Description

Each **spi_select*** (select) intrinsic selects byte *n* of argument **b** if the low-order bit (bit 0) of byte *n* of **a** is 1, otherwise it selects byte *n* of argument **c**.

spi_select32, **spi_select16** and **spi_select8** all represent the same DPU hardware operation. They have different names to allow strict type checking of arguments and result.

In ANSI C, conditional operations such as **a ? b : c** depend on whether **a** is nonzero (true, selecting **b**) or zero (false, selecting **c**). Because the DPU hardware checks bit 0 of each byte of condition **a**, the result of **spi_select** may differ from the result of **a ? b : c** in ANSI C.

Since all three versions of **spi_select*** use the bottom bit *of each byte* of the control value to decide which argument to select, the user should construct the control argument using DPU boolean intrinsics (which return all 1 bits for true) rather than the corresponding C boolean operators (which return 1 for true), or alternatively manipulate the C boolean result appropriately before using it as a **spi_select*** control argument.

7.2.12.3 Return value

In each byte, returns argument **b** if the low-order bit of **a** is 1, otherwise returns argument **c**.

7.2.12.4 See also

spi_vselect: like **spi_select**, but for vector arguments rather than scalars.

7.2.12.5 Examples

Op	**a**	**b**	**c**	Result
spi_select32	0x00000000	0x11223344	0x55667788	0x55667788
spi_select32	0xFFFFFFFF	0x11223344	0x55667788	0x11223344
spi_select32	0x0000FFFF	0x11223344	0x55667788	0x55663344
spi_select32	0x00FF00FF	0x11223344	0x55667788	0x55227744
spi_select32	0x00010203	0x11223344	0x55667788	0x55227744
spi_select16	0x00010203	0x11223344	0x55667788	0x55227744
spi_select8	0x00010203	0x11223344	0x55667788	0x55227744

7.2.13 spi_shift

7.2.13.1 Prototypes

uint32x1 **spi_shift32(uint32x1 a, int32x1 b);**

uint16x2 **spi_shift16(uint16x2 a, int32x1 b);**

uint8x4 **spi_shift8(uint8x4 a, int32x1 b);**

7.2.13.2 Description

Each **spi_shift*** (logical shift) intrinsic returns the value of **a** shifted by **b** bits. The arguments may be signed or unsigned words, packed halfwords, or packed bytes.

The same shift count **b** applies to each component for the packed variants; DPU hardware does not support separate per-component shift counts.

7.2.13.3 Return value

Returns the value of **a** shifted by **b** bits. If the shift count **b** is positive, it left shifts by **b** bits. If the shift count **b** is negative, it right shifts by **-b** bits. If **b** is zero, it simply returns **a**. If the magnitude of **b** is greater than or equal to the width of the type, **spi_shift** returns 0. Logical right shift zero-fills the high-order bit when right shifting.

7.2.13.4 See also

spi_shifta: like **spi_shift**, but uses arithmetic shift rather than logical shift.
spi_vshift: like **spi_shift**, but for vector arguments rather than scalars.

7.2.13.5 Examples

Op	**a**	**b**	Result
spi_shift32	0x4567ABCD	0x00000001	0x8ACF579A
spi_shift32	0x4567ABCD	0xFFFFFFFF	0x22B3D5A6
spi_shift32	0x4567ABCD	0x00000020	0x00000000
spi_shift32	0x4567ABCD	0xFFFFFFDF	0x00000000
spi_shift32	0xFFFFFFF0	0xFFFFFFFF	0x7FFFFFF8
spi_shift16	0x4567ABCD	0x00000001	0x8ACE579A
spi_shift16	0x4567ABCD	0xFFFFFFFF	0x22B355E6
spi_shift8	0x4567ABCD	0x00000001	0x8ACE569A
spi_shift8	0x4567ABCD	0xFFFFFFFF	0x22335566

7.2.14 spi_shifta

7.2.14.1 Prototypes

int32x1 **spi_shifta32(int32x1 a, int32x1 b);**

int16x2 **spi_shifta16(int16x2 a, int32x1 b);**

7.2.14.2 Description

Each **spi_shifta*** (arithmetic shift) intrinsic returns the value of **a** shifted by **b** bits. The arguments may be words or packed halfwords. The DPU does not support arithmetic shift for packed bytes.

The same shift count **b** applies to each component for the packed **int16x2** variant; the DPU hardware does not support separate per-component shift counts.

7.2.14.3 Return value

Returns the value of **a** shifted by **b** bits. If the shift count **b** is positive, it left shifts by **b** bits. If the shift count **b** is negative, it right shifts by **-b** bits. If **b** is zero, it simply returns **a**. Arithmetic right shift sign-fills the high-order bit during shifting.

If the shifted result overflows the range of the type, the result saturates. For the **int32x1** case (**int16x2** is similar), left shift counts 32 and above return saturated result 0x7FFFFFFF or 0x80000000, and right shift counts -32 and below return 0 for positive inputs and -1 (0xFFFFFFFF) for negative inputs.

7.2.14.4 See also

spi_shift: like **spi_shifta**, but uses logical shift rather than arithmetic shift.
spi_vshifta*: like **spi_shifta**, but for vector arguments rather than scalars.

7.2.14.5 Examples

Op	**a**	**b**	Result
spi_shifta32	0x4567ABCD	0x00000001	0x7FFFFFFFF
spi_shifta32	0x4567ABCD	0xFFFFFFFF	0x22B3D5E6
spi_shifta32	0x4567ABCD	0x00000020	0x7FFFFFFFF
spi_shifta32	0x4567ABCD	0xFFFFFFDF	0x00000000
spi_shifta32	0xFFFFFFF0	0xFFFFFFFF	0xFFFFFFF8
spi_shifta16	0x4567ABCD	0x00000001	0x7FFF8000
spi_shifta16	0x4567ABCD	0xFFFFFFFF	0x22B3D5E6

7.2.15 spi_shuffle

7.2.15.1 Prototypes

int32x1 spi_shufflei(uint32x1 a, int32x1 b, int32x1 c);

uint32x1 spi_shuffleu(uint32x1 a, uint32x1 b, uint32x1 c);

7.2.15.2 Description

Each **spi_shuffle*** (shuffle bytes) intrinsic uses the control information in **a** to perform byte reordering on **b** and **c**.

7.2.15.3 Return value

The value of each byte $\mathbf{a}_n$ of the control value **a** determines the corresponding byte $\mathbf{x}_n$ of the result **x,** as follows:

0, 1, 2, 3 — byte 0, 1, 2 or 3 of **b**
4, 5, 6, 7 — byte 0, 1, 2 or 3 of **c**
8, 9, 0xA, 0xB
spi_shufflei : 0 or 0xFF depending on the sign bit (bit 7, 0 or 1) of byte 0, 1, 2 or 3 of **b**
spi_shuffleu: 0
0xC, 0xD, 0xE, 0xF
spi_shufflei: 0 or 0xFF depending on the sign bit (bit 7, 0 or 1) of byte 0, 1, 2 or 3 of **c**
spi_shuffleu: 0

If the value of $\mathbf{a}_n$ is greater than 0xF, the result is undefined.

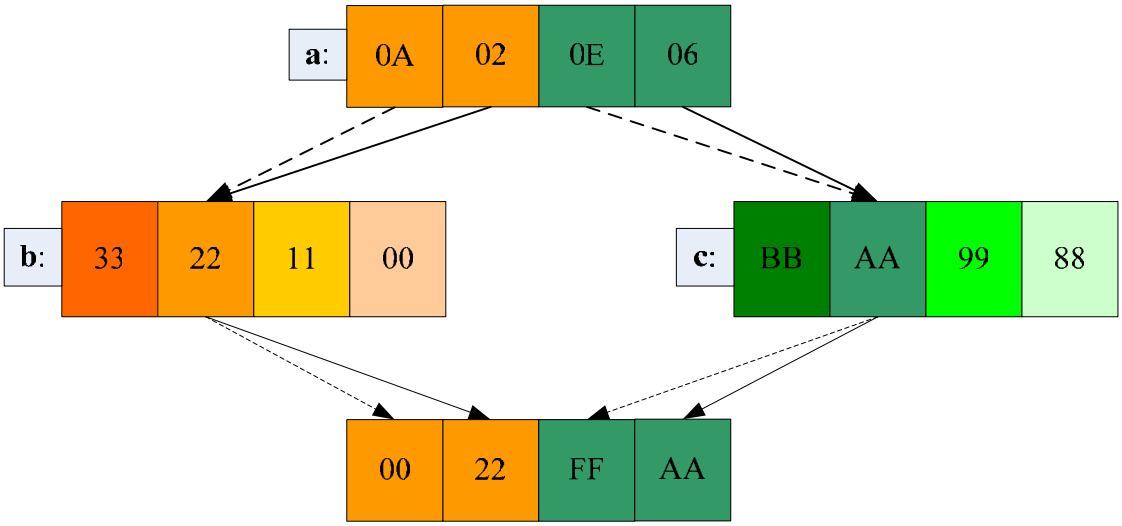
7.2.15.4 See also

spi_perm: like **spi_shuffle**, but permutes data between lanes rather than shuffling bytes within a lane.
spi_shuffled: like **spi_shuffle**, but interprets the control word as nibbles rather than bytes.
spi_vshuffle: like **spi_shuffle**, but for vector arguments rather than scalars.

7.2.15.5 Examples

Op	**a**	**b**	**c**	Result	Comment
spi_shufflei	0x00010203	0x33221100	0xBBAA9988	0x00112233	endianness byte swap
spi_shufflei	0x01000302	0x33221100	0xBBAA9988	0x11003322	halfword component swap
spi_shufflei	0x05010602	0x33221100	0xBBAA9988	0x9911AA22	byte shuffle
spi_shufflei	0x08020E06	0x33221100	0xBBAA9988	0x0022FFAA	sign-extend bytes to halfwords
spi_shuffleu	0x08020E06	0x33221100	0xBBAA9988	0x002200AA	zero-extend bytes to halfwords
spi_shufflei	0x0E0E0E06	0x33221100	0xBBAA9988	0xFFFFFFAA	sign-extend byte to word
spi_shuffleu	0x0E0E0E06	0x33221100	0xBBAA9988	0x000000AA	zero-extend byte to word
spi_shufflei	0x0E0E0504	0x33221100	0xBBAA9988	0xFFFF9988	sign-extend halfword to word
spi_shuffleu	0x0E0E0504	0x33221100	0xBBAA9988	0x00009988	zero-extend halfword to word

The following diagram illustrates the use of **spi_shufflei** to sign-extend selected bytes to halfwords using control word 0x0A020E06. Color here indicates the byte that each byte of the control word selects, while dotted arrows indicate sign-extension.



7.2.16 spi_shuffled

7.2.16.1 Prototypes

int32x1 **spi_shuffledi_hi(uint32x1 a, int32x1 b, int32x1 c);**

int32x1 **spi_shuffledi_lo(uint32x1 a, int32x1 b, int32x1 c);**

uint32x1 **spi_shuffledu_hi(uint32x1 a, uint32x1 b, uint32x1 c);**

uint32x1 **spi_shuffledu_lo(uint32x1 a, uint32x1 b, uint32x1 c);**

7.2.16.2 Description

Each **spi_shuffled*** (shuffle bytes dual) intrinsic performs byte reordering on **b** and **c** using control information in **a**.

A single hardware operation performs both **spi_shuffled*_hi** and the corresponding **spi_shuffled*_lo**, returning two results. The compiler **spc** merges adjacent paired calls to these functions for efficiency.

7.2.16.3 Return value

Each four-bit nibble of the control value **a** determines a byte of the result. For **spi_shuffled*_hi**, the high-order nibble of $\mathbf{a}_n$ determines byte $\mathbf{x}_n$ of the result **x**, while for **spi_shuffled*_lo** the low-order nibble of $\mathbf{a}_n$ determines byte $\mathbf{x}_n$ of the result **x**. The control values are as follows:

0, 1, 2, 3 — byte 0, 1, 2 or 3 of **b**
4, 5, 6, 7 — byte 0, 1, 2 or 3 of **c**
8, 9, 0xA, 0xB — **spi_shuffledi***: 0 or 0xFF depending on the sign bit (bit 7, 0 or 1) of byte 0, 1, 2 or 3 of **b**
spi_shuffledu*: 0

0xC, 0xD, 0xE, 0xF
spi_shuffledi*: 0 or 0xFF depending on the sign bit (bit 7, 0 or 1) of byte 0, 1, 2 or 3 of **c**
spi_shuffledu*: 0

7.2.16.4 See also

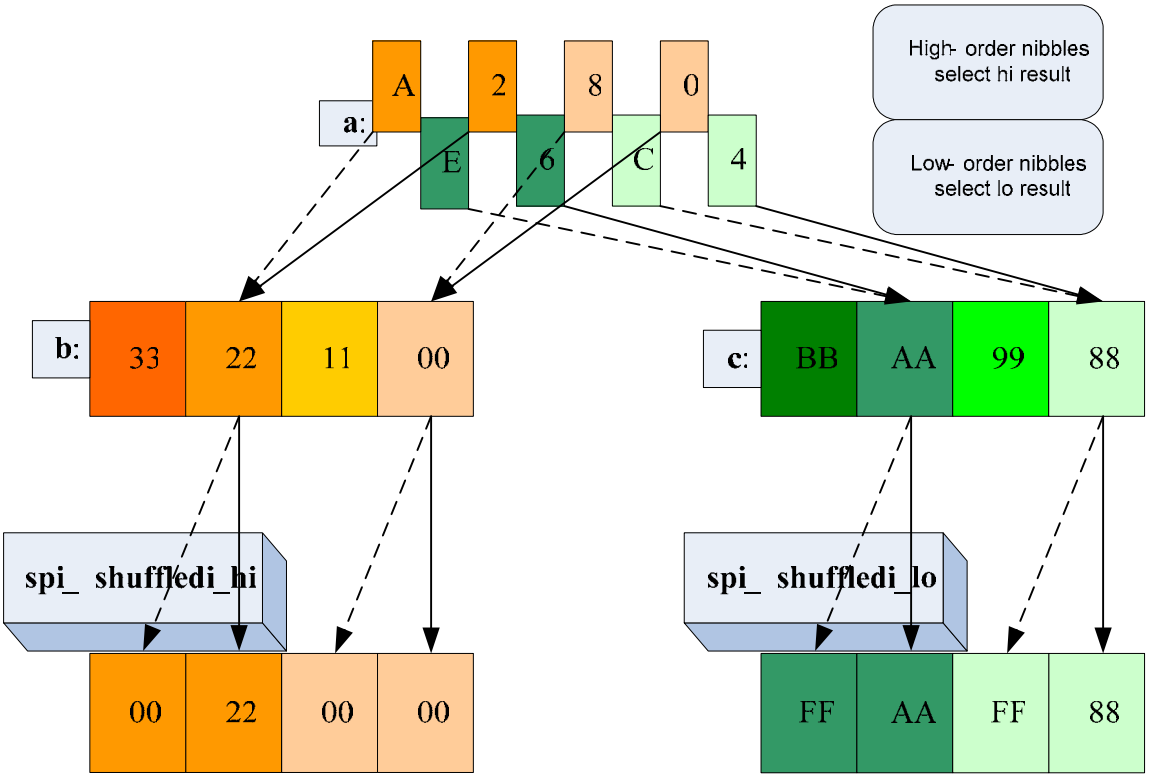
spi_shuffle: like **spi_shuffled**, but interprets control word as bytes rather than nibbles.
spi_vshuffled: like **spi_shuffled**, but takes vector arguments and returns a vector result rather than a scalar.

7.2.16.5 Examples

Op	**a**	**b**	**c**	**hi** result	**lo** result	Comment
spi_shuffledi_*	0x04152637	0x33221100	0xBBAA9988	0x00112233	0x8899AABB	endianness byte swap
spi_shuffledi_*	0x15043726	0x33221100	0xBBAA9988	0x11003322	0x9988BBAA	halfword component swap
spi_shuffledi_*	0x17345602	0x33221100	0xBBAA9988	0x11339900	0xBB88AA22	byte shuffle
spi_shuffledi_*	0x8C26EA62	0x33221100	0xBBAA9988	0x0022FFAA	0xFFAA0022	sign-extend bytes to halfwords
spi_shuffledu_*	0x8C26EA62	0x33221100	0xBBAA9988	0x002200AA	0x00AA0022	zero-extend bytes to halfwords

spi_shuffledi_*	0xEEEEEE62	0x33221100	0xBBAA9988	0xFFFFFFAA	0xFFFFFF22	sign-extend byte to word
spi_shuffledu_*	0xEEEEEE62	0x33221100	0xBBAA9988	0x000000AA	0x00000022	zero-extend byte to word
spi_shuffledi_*	0xEEEE7564	0x33221100	0xBBAA9988	0xFFFFBBAA	0xFFFF9988	sign-extend halfword to word
spi_shuffledu_*	0xEEEE7564	0x33221100	0xBBAA9988	0x0000BBAA	0x00009988	zero-extend halfword to word

The following diagram shows the use of **spi_shuffledi_*** with control argument 0xAE268C04 to sign-extend the low-order bytes of **b** and **c** to halfwords. Color here indicates the byte that each nibble of the control word selects, while dotted arrows indicate sign-extension.



7.2.17 spi_sub

7.2.17.1 Prototype

int32x1 spi_sub32i(int32x1 a, int32x1 b);

7.2.17.2 Description

Intrinsic **spi_sub32i** (subtraction) returns the difference of its arguments. The DPU supports only subtraction of signed words.

7.2.17.3 Return value

Returns **a - b**.

7.2.17.4 See also

spi_add: like **spi_sub**, but using addition rather than subtraction.
spi_vsub: like **spi_sub**, but for vector arguments rather than scalars.

7.2.17.5 Examples

Op	**a**	**b**	Result
spi_sub32i	0x00000010	0x00000003	0x0000000D

7.2.18 spi_vabd

7.2.18.1 Prototypes

vec uint32x1 spi_vabd32i(vec int32x1 a, vec int32x1 b);

vec uint16x2 spi_vabd16i(vec int16x2 a, vec int16x2 b);

vec uint16x2 spi_vabd16u(vec uint16x2 a, vec uint16x2 b);

vec uint8x4 spi_vabd8u(vec uint8x4 a, vec uint8x4 b);

7.2.18.2 Description

Each **spi_vabd*** (vector absolute difference) intrinsic returns the absolute value of the difference of its arguments. The arguments may be vectors of signed or unsigned words, packed halfwords, or packed bytes.

7.2.18.3 Return value

Returns | **a - b** |. The return value is always of unsigned type.

7.2.18.4 See also

spi_vsad: like **spi_vabd**, but returns a sum of absolute differences rather than a single absolute difference.

7.2.18.5 Examples

Op	**a**	**b**	Result
spi_vabd32i	0xFFFFFFFF	0x00000003	0x00000004
spi_vabd16i	0xFFFFFFFF	0x00000003	0x00010004
spi_vabd16u	0xFFFFFFFF	0x00000003	0xFFFFFFFC
spi_vabd8u	0xFFFFFFFF	0x00000003	0xFFFFFFFC

7.2.19 spi_vadd

7.2.19.1 Prototypes

vec int32x1 spi_vadd32i(vec int32x1 a, vec int32x1 b);

vec uint32x1 spi_vadd32u(vec uint32x1 a, vec uint32x1 b);

vec int16x2 spi_vadd16i(vec int16x2 a, vec int16x2 b);

vec uint16x2 spi_vadd16u(vec uint16x2 a, vec uint16x2 b);

vec int8x4 spi_vadd8i(vec int8x4 a, vec int8x4 b);

vec uint8x4 spi_vadd8u(vec uint8x4 a, vec uint8x4 b);

7.2.19.2 Description

Each **spi_vadd*** (vector addition) intrinsic returns the sum of its arguments. The arguments may be vectors of signed or unsigned words, packed halfwords, or packed bytes.

The signed and unsigned variant of each pair of **spi_vadd*[iu]** intrinsics represent the same DPU hardware operation. They have different names to allow strict type checking of arguments and result.

7.2.19.3 Return value

Returns **a** + **b**.

7.2.19.4 See also

spi_add: like **spi_vadd**, but for scalar arguments rather than vectors.
spi_vaddc: like **spi_vadd**, but with carry.
spi_vadds: like **spi_vadd**, but uses saturation arithmetic.
spi_vsub: like **spi_vadd**, but subtracts rather than adding.

7.2.19.5 Examples

Op	**a**	**b**	Result
spi_vadd32i	0xFFFFFFFF	0x00000003	0x00000002
spi_vadd32u	0xFFFFFFFF	0x00000003	0x00000002
spi_vadd16i	0xFFFFFFFF	0x00000003	0xFFFF0002
spi_vadd16u	0xFFFFFFFF	0x00000003	0xFFFF0002
spi_vadd8i	0xFFFFFFFF	0x00000003	0xFFFFFF02
spi_vadd8u	0xFFFFFFFF	0x00000003	0xFFFFFF02

7.2.20 spi_vaddc

7.2.20.1 Prototype

vec uint16x2 spi_vaddc8u_hi(vec uint8x4 a, vec uint8x4 b, vec uint8x4 c); // Storm-2 only

vec uint16x2 spi_vaddc8u_lo(vec uint8x4 a, vec uint8x4 b, vec uint8x4 c); // Storm-2 only

vec int32x1 spi_vaddc32(vec int32x1 a, vec int32x1 b, vec int32x1 c);

vec int32x1 spi_vaddc32_c(vec int32x1 a, vec int32x1 b, vec int32x1 c);

7.2.20.2 Description

spi_vaddc32 (vector addition with carry) adds its first two arguments plus the low-order bit of its third argument and returns the sum, while **spi_vaddc32_c** computes the same sum and returns the resulting carry. A single hardware operation performs both **spi_vaddc32** and **spi_vaddc32_c**, returning two results. The compiler **spc** merges adjacent paired calls to these functions for efficiency. The library currently provides only the signed form of**spi_vaddc32**, but the semantics of the corresponding unsigned operation are identical, so the user can perform the unsigned form by appropriate casting of arguments and result.

Each **spi_vaddc8u*** intrinsic performs a similar operation, but for packed byte arguments. **spi_vaddc8u_hi** computes the unsigned 16-bit sum of the high-order bytes of **a** and **b** plus the low bit of the high-order byte of **c**, packed with the similar 16-bit sum of the third-highest bytes; **spi_vaddc8u_lo** computes the similar packed result for the second-highest and lowest bytes. A single hardware operation performs both **spi_vaddc8u_hi** and **spi_vaddc8u_lo**, returning two results. **spi_vaddc8u*** intrinsics exist on Storm-2 only, not on Storm-1.

7.2.20.3 Return value

spi_vaddc32 returns the sum **a** + **b** plus the low-order bit of **c**, while **spi_vaddc32_c** returns the carry from the same sum.

spi_vaddc8u_hi returns byte3(**a**) + byte3(**b**) + bit0(byte3(**c**)) in the high-order halfword and byte1(**a**) + byte1(**b**) + bit0(byte1(**c**)) in the low-order halfword. **spi_vaddc8u_lo** returns byte2(**a**) + byte2(**b**) + bit0(byte2(**c**)) in the high-order halfword and byte0(**a**) + byte0(**b**) + bit0(byte0(**c**)) in the low-order halfword.

7.2.20.4 See also

spi_vadd: like **spi_vaddc**, but without carry.
spi_vsubc: like **spi_vaddc**, but subtracts rather than adding.

7.2.20.5 Examples

Op	**a**	**b**	**c**	**spi_vaddc32** result	**spi_vaddc32_c** result	
spi_vaddc32*	0x00000001	0x00000003	0x00000000	0x00000004	0x00000000	
spi_vaddc32*	0xFFFFFFFF	0x00000003	0x00000000	0x00000002	0x00000001	
spi_vaddc32*	0xFFFFFFFF	0x00000003	0x00000001	0x00000003	0x00000001	
spi_vaddc32*	0xFFFFFFFF	0x00000003	0x00000002	0x00000002	0x00000001	

Op	**a**	**b**	**c**	**spi_vaddc8u_hi** result	**spi_vaddc8u_lo** result	
spi_vaddc8u*	0x03020100	0x30201000	0x01010101	0x00340012	0x00230001	Storm-2 only
spi_vaddc8u*	0xFEFF0001	0x01010101	0x02020202	0x00FF0001	0x00000002	Storm-2 only

7.2.21 spi_vadds

7.2.21.1 Prototypes

vec int32x1 spi_vadds32i(vec int32x1 a, vec int32x1 b);

vec int16x2 spi_vadds16i(vec int16x2 a, vec int16x2 b);

vec uint16x2 spi_vadds16u(vec uint16x2 a, vec uint16x2 b);

vec uint16x2 spi_vadds16ui(vec uint16x2 a, vec int16x2 b);

vec uint8x4 spi_vadds8u(vec uint8x4 a, vec uint8x4 b);

vec uint8x4 spi_vadds8ui(vec uint8x4 a, vec int8x4 b);

7.2.21.2 Description

Each **spi_vadds*** (vector addition with saturation) intrinsic returns the sum of its arguments using saturation arithmetic. In addition to the signed and unsigned flavors, mixed variants allow saturating addition of unsigned plus signed, producing an unsigned saturated result.

7.2.21.3 Return value

Returns **a** + **b** using saturation arithmetic; if the sum underflows or overflows the range of the result type, it returns the minimum or maximum representable value.

7.2.21.4 See also

spi_vadd: like **spi_vadds**, but does not use saturation arithmetic.
spi_vsubs: like **spi_vadds**, but subtracts rather than adding.

7.2.21.5 Examples

Op	**a**	**b**	Result
spi_vadds32i	0x00000001	0xFFFFFFFF	0x00000000
spi_vadds16i	0x00000001	0xFFFFFFFF	0xFFFF0000
spi_vadds16u	0x00000001	0xFFFFFFFF	0xFFFFFFFF
spi_vadds16ui	0x00010002	0x0001FFFF	0x00020001
spi_vadds8u	0x00000001	0xFFFFFFFF	0fFFFFFFFF
spi_vadds8ui	0x01020304	0x10FF30F0	0x11013300

7.2.22 spi_vand

7.2.22.1 Prototypes

vec uint32x1 spi_vand32(vec uint32x1 a, vec uint32x1 b);

vec uint16x2 spi_vand16(vec uint16x2 a, vec uint16x2 b);

vec uint8x4 spi_vand8(vec uint8x4 a, vec uint8x4 b);

7.2.22.2 Description

Each **spi_vand*** (vector AND) intrinsic returns the bitwise AND of its arguments. The arguments may be vectors of words, packed halfwords, or packed bytes.

spi_vand32, **spi_vand16** and **spi_vand8** all represent the same DPU hardware operation. They have different names to allow strict type checking of arguments and result.

7.2.22.3 Return value

Each return value bit is 1 if the corresponding bit in both arguments is 1, otherwise it is 0.

7.2.22.4 See also

spi_and: like **spi_vand**, but for scalar arguments rather than vectors.
spi_vnot: like **spi_vand**, but uses bitwise NOT rather than bitwise AND.
spi_vor: like **spi_vand**, but uses bitwise OR rather than bitwise AND.
spi_vrandl: like **spi_vrandl**, but performs cross-lane reduction AND rather than AND in each lane.
spi_vxor: like **spi_vand**, but uses bitwise XOR rather than bitwise AND.

7.2.22.5 Examples

Op	**a**	**b**	Result
spi_vand32	0x12345678	0x00FF00FF	0x00340078
spi_vand16	0x12345678	0x00FF00FF	0x00340078
spi_vand8	0x12345678	0x00FF00FF	0x00340078

7.2.23 spi_vclip

7.2.23.1 Prototypes

vec int16x2 spi_vclip32i(vec int32x1 a, vec int32x1 b);

vec uint16x2 spi_vclip32u(vec uint32x1 a, vec uint32x1 b);

vec int8x4 spi_vclip16i(vec int16x2 a, vec int16x2 b);

vec uint8x4 spi_vclip16u(vec uint16x2 a, vec uint16x2 b);

7.2.23.2 Description

Each **spi_vclip*** (vector clip) intrinsic packs its arguments into a packed representation, clipping each argument which underflows or overflows the range of the result type. **spi_vclip32i** and **spi_vclip32u** pack two 32-bit arguments into a result containing two packed halfwords. **spi_vclip16i** and **spi_vclip16u** pack four 16-bit quantities from the two 32-bit arguments into a result containing four packed bytes.

7.2.23.3 Return value

spi_vclip32i and **spi_vclip32u** return a packed value with the clipped value of **a** in the high-order halfword and the clipped value of **b** in the low-order halfword.

spi_vclip16i and **spi_vclip16u** return a packed value with the high-order clipped halfword of **a** in the high-order byte, followed by the low-order clipped halfword of **a**, the high-order clipped halfword of **b** and finally the low-order clipped halfword of **b** in the low-order byte.

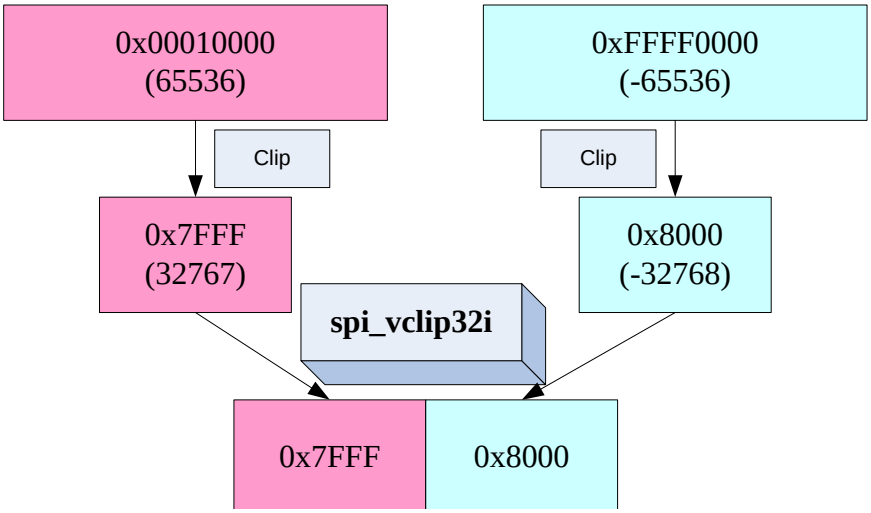
7.2.23.4 See also

spi_vsclip: like **spi_vclip**, but shifts its arguments before clipping.
spi_vshuffle: can be used to pack arguments without clipping.

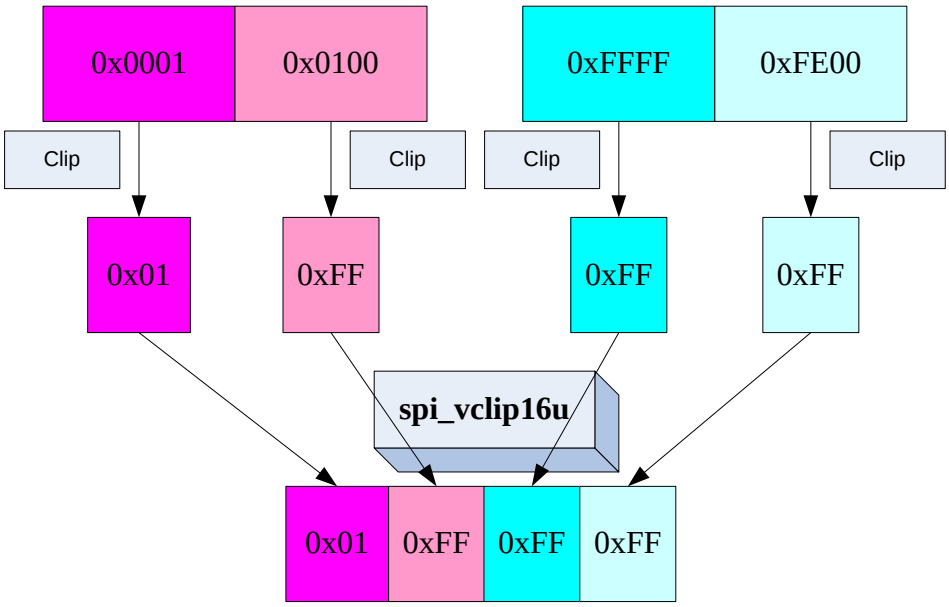
7.2.23.5 Examples

Op	**a**	**b**	Result
spi_vclip32i	0x00000000	0x00007FFF	0x00007FFF
spi_vclip32u	0x00000000	0x00007FFF	0x00007FFF
spi_vclip32i	0x00008000	0x0000FFFF	0x7FFF7FFF
spi_vclip32u	0x00008000	0x0000FFFF	0x8000FFFF
spi_vclip32i	0x00010000	0xFFFF0000	0x7FFF8000
spi_vclip32u	0x00010000	0xFFFF0000	0xFFFFFFFF
spi_vclip32i	0xFFFEFFFF	0xFFFFFFFF	0x8000FFFF
spi_vclip32u	0xFFFEFFFF	0xFFFFFFFF	0xFFFFFFFF
spi_vclip16i	0x00010100	0xFFFFFE00	0x017FFF80
spi_vclip16u	0x00010100	0xFFFFFE00	0x01FFFFFF

The following diagram shows the operation of **spi_vclip32i**.



The following diagram shows the operation of **spi_vclip16u**.



7.2.24 spi_vdivstep

7.2.24.1 Prototypes

vec uint32x1 spi_vdivstep32(vec uint32x1 a, vec uint32x1 b, vec uint32 c);

vec uint32x1 spi_vdivstep32_r(vec uint32x1 a, vec uint32x1 b, vec uint32 c);

vec uint16x2 spi_vdivstep16(vec uint16x2 a, vec uint16x2 b, vec uint16x2 c);

vec uint16x2 spi_vdivstep16_r(vec uint16x2 a, vec uint16x2 b, vec uint16x2 c);

7.2.24.2 Description

The DPU hardware operation DIVSTEP32 performs a 32-bit divide step operation and returns two results, a partial dividend and a partial remainder. A sequence of 32 DIVSTEP32 operations performs a 32-bit unsigned division/remainder operation on **uint32x1** operands. Similarly, a sequence of 16 DIVSTEP16 operations performs a 16-bit unsigned division/remainder on **uint16x2** operands.

spi_vdivstep32 (vector divide step) returns the dividend of a 32-bit divide step operation and **spi_vdivstep32_r** returns the remainder. Similarly, **spi_vdivstep16** returns the dividend of a 16-bit divide step operation and **spi_vdivstep16_r** returns the remainder. The compiler **spc** merges adjacent paired calls to these functions for efficiency. A division/remainder routine must use both calls, as partial dividend and partial remainder are both needed at each step of the algorithm; see the sample code in the Examples subsection below.

spi_vdivstep applies only to unsigned operands. For signed division and remainder, the program should take the absolute value of arguments before performing division/remainder and adjust quotient and remainder based on argument signs after.

The number of divide step operations required to perform a divide/remainder operation is equal to the number of significant bits in the dividend, so fewer than 32 or 16 divide steps are sufficient if an upper limit to the number of significant dividend bits is known in advance. For example, to perform a divide/remainder operation with a 20-bit dividend, left shift the dividend by 12 bits (32 - 20) to move the most significant dividend bit to bit 31, then use 20 divide step operations rather than 32.

7.2.24.3 Return value

spi_vdivstep32 and **spi_vdivstep32_r** return the divisor and remainder of a 32-bit divide step operation, respectively.
spi_vdivstep16 and **spi_vdivstep16_r** return the divisor and remainder of a 16-bit divide step operation, respectively.

A divide step performs the following calculation:

Left shift the combined quantity **a**:**c** (64 bits or 32 bits) by 1 bit to get new **a**:**c**
newc = (**a** < **b**) ? **c** : **c** | 1
newa = (a < **b**) ? **a** : **a - b**
a = newa
c = newc

7.2.24.4 Examples

Op	**a**	**b**	c	**spi_vdivstep*** result	**spi_vdivstep*_r** result	Comment
spi_vdivstep32*	0x00000000	0x00000004	0x00000023	0x00000046	0x00000000	35 / 4 step 1
spi_vdivstep32*	0x00000000	0x00000004	0x60000001	0xC0000002	0x00000000	35 / 4 step 30

Op	**a**	**b**	c	**spi_vdivstep*** result	**spi_vdivstep*_r** result	Comment
spi_vdivstep32*	0x00000000	0x00000004	0xC0000002	0x80000004	0x00000001	35 / 4 step 31
spi_vdivstep32*	0x00000001	0x00000004	0x80000004	0x00000008	0x00000003	35 / 4 step 32
spi_vdivstep16*	0x00000000	0x00070004	0x00610023	0x00C20046	0x00000000	95/7, 35/4 step 1
spi_vdivstep16*	0x00050000	0x00070004	0x20016001	0x4003C002	0x00030000	95/7, 35/4 step 14
spi_vdivstep16*	0x00030000	0x00070004	0x4003C002	0x80068004	0x00060001	95/7, 35/4 step 15
spi_vdivstep16*	0x00060001	0x00070004	0x80068004	0x000D0008	0x00060003	95/7, 35/4 step 16

The kernel function below performs 32-bit unsigned division and remainder. The **uint16x2** version is identical, but with 16 **spi_vdivstep16*** operations rather than 32 **spi_vdivstep32*** operations.

```
kernel      void  div32u(
      vec   uint32x1    dividend(in),
      vec   uint32x1    divisor(in),
      vec   uint32x1    quotient(out),
      vec   uint32x1    remainder(out)
      )
{
      vec   uint32x1    a, b, c, d;

      a = 0;
      b = divisor;
      c = dividend;

      __repeat__( ; 32) {
            d = spi_vdivstep32(a, b, c);
            a = spi_vdivstep32_r(a, b, c);
            c = d;
      }

      quotient = c;
      remainder = a;

}
```


7.2.25 spi_vdotna

7.2.25.1 Prototypes

vec int32x1 spi_vdotna16i_hi(vec int16x2 a, vec int16x2 b, vec uint32x1 c);

vec int32x1 spi_vdotna16i_lo(vec int16x2 a, vec int16x2 b, vec uint32x1 c);

vec int32x1 spi_vdotna16ui_hi(vec uint16x2 a, vec int16x2 b, vec uint32x1 c);

vec int32x1 spi_vdotna16ui_lo(vec uint16x2 a, vec int16x2 b, vec uint32x1 c);

7.2.25.2 Description

Each **spi_vdotna*** (vector dot product negative accumulating) intrinsic computes the negative dot product of **a** and **b**, sign extends the result to 64 bits, and adds **c** to the result. Operands **a** and **b** must be packed words; operand **a** is signed, operand **b** may be signed or signed. Operand c is always **uint32x1**.

A single hardware operation performs both **spi_vdotna16*_hi** and the corresponding **spi_vdotna16*_lo**, returning two results. The compiler **spc** merges adjacent paired calls to these functions for efficiency.

7.2.25.3 Return value

Each **spi_vdotna16*_hi** intrinsic returns the high-order 32 bits of the unsaturated result of hi(**a**) ***** hi(**b**) **-** lo(**a**) * lo(**b**) **+ c**; each **spi_vdotna16*_lo** intrinsic returns the low-order 32 bits of the same result.

7.2.25.4 See also

spi_vdotpa: like **spi_vdotna**, but uses positive dot product rather than negative.

7.2.25.5 Examples

Op	**a**	**b**	**c**	**hi** result	**lo** result
spi_vdotna16i_*	0x00020003	0x00050007	0x00000000	0xFFFFFFFF	0xFFFFFFF5
spi_vdotna16i_*	0x00020003	0x00050007	0x00000001	0xFFFFFFFF	0xFFFFFFF6
spi_vdotna16i_*	0xFFFF0003	0x0005FFFF	0x00000000	0xFFFFFFFF	0xFFFFFFFE
spi_vdotna16i_*	0xFFFF0003	0x0005FFFF	0x00000001	0xFFFFFFFF	0xFFFFFFFF
spi_vdotna16ui_*	0xFFFF0003	0x0005FFFF	0x00000000	0x00000000	0x0004FFFE
spi_vdotna16ui_*	0xFFFF0003	0x0005FFFF	0x00000001	0x00000000	0x0004FFFF

See **spi_vdotpa** for an illustrative diagram.

7.2.26 spi_vdotp

7.2.26.1 Prototypes

vec int32x1 spi_vdotp16x8i(vec int16x2 a, vec int16x2 b, vec int8x4 c); // Storm-2 only

7.2.26.2 Description

spi_vdotp16x8i (vector dot product, 16-bit x 8-bit) computes the 32-bit signed dot product of the four halfwords of arguments **a** and **b** with the four bytes of argument **c.** Operands **a** and **b** must be packed signed halfwords, and operand **c** must be signed packed bytes.

spi_vdotp16x8i exists on Storm-2 only, not on Storm-1.

7.2.26.3 Return value

spi_vdotp16x8i returns the 32-bit signed dot product hi(**a**) ***** byte3**(c) +** lo(**a**) * byte2**(c) +** hi(**b**) ***** byte1**(c) +** lo(**b**) * byte0**(c)**.

7.2.26.4 See also

spi_vdotpa: like **spi_vdotp**, but for halfword arguments.

7.2.26.5 Examples

Op	**a**	**b**	**c**	Result	
spi_vdotp16x8i	0x10000100	0x00100001	0x04030201	0x00004321	Storm-2 only
spi_vdotp16x8i	0xFFFE0001	0x0002FFFF	0xFEFF0102	0x00000003	Storm-2 only

7.2.27 spi_vdotpa

7.2.27.1 Prototypes

vec int32x1 spi_vdotpa16i_hi(vec int16x2 a, vec int16x2 b, vec uint32x1 c);

vec int32x1 spi_vdotpa16i_lo(vec int16x2 a, vec int16x2 b, vec uint32x1 c);

vec uint32x1 spi_vdotpa16u_hi(vec uint16x2 a, vec uint16x2 b, vec uint32x1 c);

vec uint32x1 spi_vdotpa16u_lo(vec uint16x2 a, vec uint16x2 b, vec uint32x1 c);

vec int32x1 spi_vdotpa16ui_hi(vec uint16x2 a, vec int16x2 b, vec uint32x1 c);

vec int32x1 spi_vdotpa16ui_lo(vec uint16x2 a, vec int16x2 b, vec uint32x1 c);

vec uint32x1 spi_vdotpa8u(vec uint8x4 a, vec uint8x4 b, vec uint32x1 c);

vec int32x1 spi_vdotpa8ui(vec uint8x4 a, vec int8x4 b, vec int32x1 c);

7.2.27.2 Description

Each **spi_vdotpa*** (vector dot product positive accumulating) intrinsic adds **c** to the positive dot product of **a** and **b; c** is always unsigned for **spi_vdotpa16***. Operands **a** and **b** can be packed bytes or packed words.

A single hardware operation performs both **spi_vdotpa*_hi** and the corresponding **spi_vdotpa*_lo**, returning two results. The compiler **spc** merges adjacent paired calls to these functions for efficiency.

7.2.27.3 Return value

Each **spi_vdotpa16*_hi** intrinsic returns the high-order 32 bits of the unsaturated result hi(**a**) * hi(**b**) + lo(**a**) * lo(**b**) + **c**; each **spi_vdotpa16*_lo** intrinsic returns the low-order 32 bits of the same result. If the result is signed, the intermediate result is sign-extended to 64 bits before adding **c**, but note that **c** is unsigned, not signed.

Each **spi_vdotpa8*** intrinsic computes the four products of the corresponding bytes of **a** and **b** and sums them with **c**.

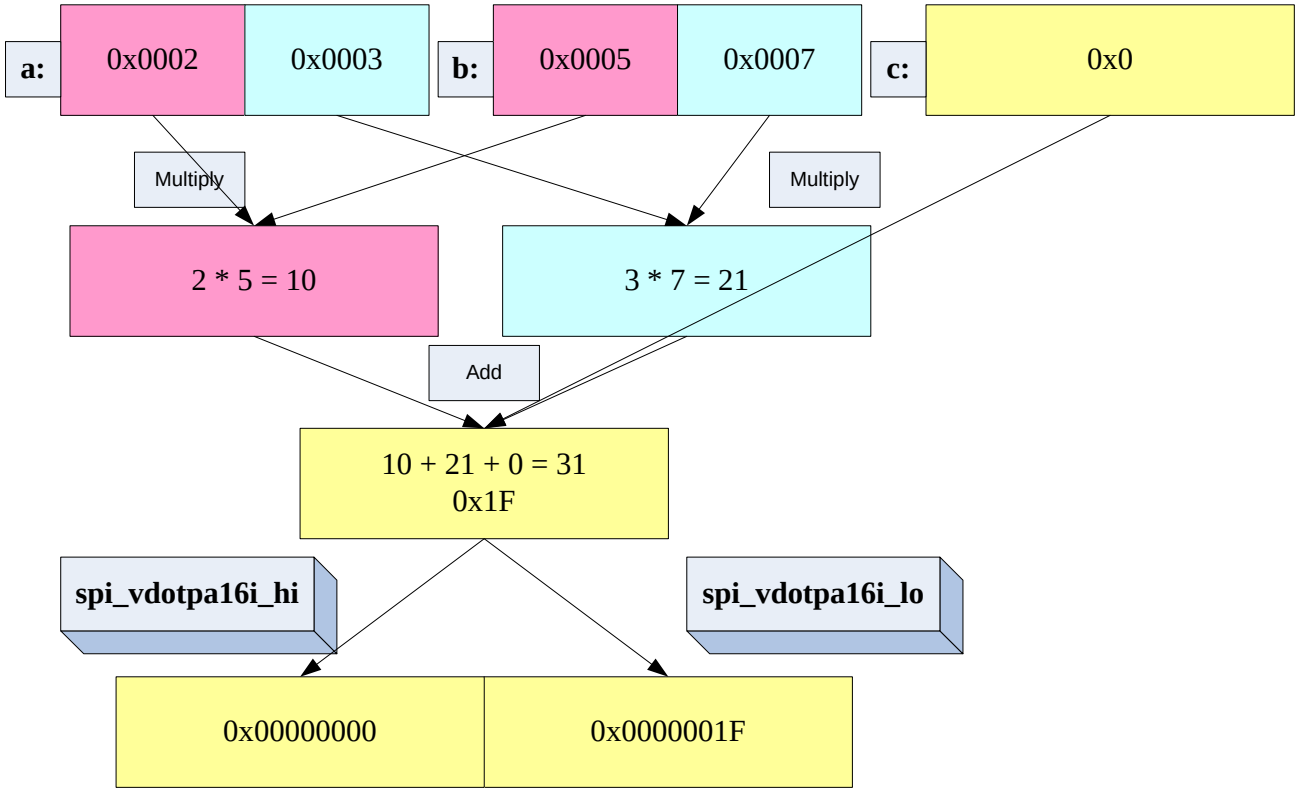
7.2.27.4 See also

spi_vdotna: like **spi_vdotpa**, but uses negative dot product rather than positive.
spi_vdotp: like **spi_vdotpa**, but computes a 4-component 16x8 dot product rather than a two-component 16x16 dot product.

7.2.27.5 Examples

Op	**a**	**b**	**c**	**hi** result	**lo** result
spi_vdotpa16i_*	0x00020003	0x00050007	0x00000000	0x00000000	0x0000001F
spi_vdotpa16i_*	0x00020003	0x00050007	0x00000001	0x00000000	0x00000020
spi_vdotpa16i_*	0xFFFF0003	0x0005FFFF	0x00000000	0xFFFFFFFF	0xFFFFFFF8
spi_vdotpa16i_*	0xFFFF0003	0x0005FFFF	0x00000001	0xFFFFFFFF	0xFFFFFFF9
spi_vdotpa16u_*	0xFFFF0003	0x0005FFFF	0x00000000	0x00000000	0x0007FFF8
spi_vdotpa16u_*	0xFFFF0003	0x0005FFFF	0x00000001	0x00000000	0x0007FFF9
spi_vdotpa8u	0xFE040608	0x03FB0709	0x00000000	0x00000758	
spi_vdotpa8ui	0xFE040608	0x03FB0709	0x00000000	0x00000358	

The following diagram shows the operation of spi_vdotpa16i_*.



7.2.28 spi_veq

7.2.28.1 Prototypes

vec int32x1 spi_veq32(vec uint32x1 a, vec uint32x1 b);

vec int16x2 spi_veq16(vec uint16x2 a, vec uint16x2 b);

vec int8x4 spi_veq8(vec uint8x4 a, vec uint8x4 b);

7.2.28.2 Description

Each **spi_veq*** (vector equals) intrinsic returns the boolean value true (all 1 bits) if **a** equals **b** and the boolean value false (all 0 bits) otherwise. The arguments may be scalars or vectors of words, packed halfwords, or packed bytes.

In ANSI C, the result of the equality operator '==' is 0 (false) or 1 (true). Because the DPU hardware returns all 1 bits (that is, 0xFF, 0xFFFF or 0xFFFFFFFF) rather than 1 for true, the result of arithmetic using the result of **spi_eq*** may differ from the result in ANSI C.

7.2.28.3 Return value

Returns true (all 1 bits) if **a** is equal to **b** and 0 if not.

7.2.28.4 See also

spi_eq: like **spi_veq**, but for scalar arguments rather than vectors.
spi_vle: like **spi_veq**, but uses less than or equal to rather than equals.
spi_vlt: like **spi_veq**, but uses less than rather than equals.
spi_vne: like **spi_veq**, but uses not equals rather than equals.

7.2.28.5 Examples

Op	**a**	**b**	Result
spi_veq32	0x00000002	0x00000002	0xFFFFFFFF
spi_veq32	0x00000002	0xFFFFFFFF	0x00000000
spi_veq16	0x00010002	0x0001FFFF	0xFFFF0000
spi_veq8	0x01020304	0x01FFFF04	0xFF0000FF

7.2.29 spi_vffone

7.2.29.1 Prototypes

vec int32x1 spi_vffone32(vec int32x1 a);

vec int16x2 spi_vffone16(vec int16x2 a);

7.2.29.2 Description

Each **spi_vffone*** (vector find first 1) intrinsic returns the number of leading 0 bits before the highest-order 1 bit in its argument, or the width of the type (32 or 16) if there is no 1 bit. The arguments may be vectors of signed or unsigned words or packed halfwords.

7.2.29.3 Return value

Returns the number of leading 0 bits before the highest-order 1 bit.

7.2.29.4 Examples

Op	**a**	Result
spi_vffone32	0x00000000	0x00000020
spi_vffone32	0x04000000	0x00000005
spi_vffone32	0x11223344	0x00000003
spi_vffone16	0x00000000	0x00100010
spi_vffone16	0x04000000	0x00050010
spi_vffone16	0x11223344	0x00030002

7.2.30 spi_vle

7.2.30.1 Prototypes

vec int32x1 spi_vle32i(vec int32x1 a, vec int32x1 b);

vec int32x1 spi_vle32u(vec uint32x1 a, vec uint32x1 b);

vec int16x2 spi_vle16i(vec int16x2 a, vec int16x2 b);

vec int16x2 spi_vle16u(vec uint16x2 a, vec uint16x2 b);

vec int8x4 spi_vle8i(vec int8x4 a, vec int8x4 b);

vec int8x4 spi_vle8u(vec uint8x4 a, vec uint8x4 b);

7.2.30.2 Description

Each **spi_vle*** (vector less than or equal to) intrinsic returns the boolean value true (all 1 bits) if **a** is less than or equal to **b** and the boolean value false (all 0 bits) otherwise. The arguments may be vectors of signed or unsigned words, packed halfwords, or packed bytes.

In ANSI C, the result of the less than or equal to operator '<=' is 0 (false) or 1 (true). Because the DPU hardware returns all 1 bits (that is, 0xFF, 0xFFFF or 0xFFFFFFFF) rather than 1 for true, the result of arithmetic using the result of **spi_le*** may differ from the result in ANSI C.

7.2.30.3 Return value

Returns true (all 1 bits) if **a** is less than or equal to **b** and 0 if not.

7.2.30.4 See also

spi_le: like **spi_vle**, but for scalar arguments rather than vectors.
spi_veq: like **spi_vle**, but returns equal to rather than less than or equal to.
spi_vlt: like **spi_vle**, but returns strictly less than rather than less than or equal to.
spi_vne: like **spi_vle**, but returns not equal to rather than less than or equal to.

7.2.30.5 Examples

Op	**a**	**b**	Result
spi_vle32i	0x00000002	0x00000002	0xFFFFFFFF
spi_vle32i	0x00000002	0x00000003	0xFFFFFFFF
spi_vle32i	0x00000002	0xFFFFFFFF	0x00000000
spi_vle32u	0x00000002	0xFFFFFFFF	0xFFFFFFFF
spi_vle16i	0x00010002	0x0001FFFF	0xFFFF0000
spi_vle16u	0x00010002	0x0001FFFF	0xFFFFFFFF
spi_vle8i	0x01020304	0x0121FF03	0xFFFF0000
spi_vle8u	0x01020304	0x0121FF03	0xFFFFFF00

7.2.31 spi_vlsbs

7.2.31.1 Prototype

uint32x1 spi_vlsbs(vec uint32x1 a);

7.2.31.2 Description

Intrinsic **spi_vlsbs** (vector least significant bits) returns a word containing the least significant bit (bit 0) of **a** in each lane. It returns a scalar value, not a vector.

7.2.31.3 Return value

Bit *n* of the result is 0 if the least significant bit of **a** in lane *n* is 0, otherwise it is 1. Higher order bits of the result are always 0.

7.2.31.4 See also

spi_vrandl: like **spi_vlsbs**, but ANDs bits together rather than packing them into the result.
spi_vrorl: like **spi_vlsbs**, but ORs bits together rather than packing them into the result.

7.2.31.5 Examples

Op	**a** lane 0	**a** lane 1	**...**	**a** lane 14	**a** lane 15	Result	Comment
spi_vlsbs	0x00000000	0x00000000	...	0x00000000	0x00000000	0x00000000	lsb 0 in all lanes
spi_vlsbs	0x00000000	0x00000001	...	0x00000000	0x00000000	0x00000002	lsb non-0 in lane 1
spi_vlsbs	0x00000000	0x00000001	...	0x00000000	0x00000001	0x0000AAAA	lsb non-0 in odd lanes
spi_vlsbs	0x00000001	0x00000003	...	0x00007FFF	0x0000FFFF	0x0000FFFF	lsb non-0 in all lanes

7.2.32 spi_vlt

7.2.32.1 Prototypes

vec int32x1 spi_vlt32i(vec int32x1 a, vec int32x1 b);

vec int32x1 spi_vlt32u(vec uint32x1 a, vec uint32x1 b);

vec int16x2 spi_vlt16i(vec int16x2 a, vec int16x2 b);

vec int16x2 spi_vlt16u(vec uint16x2 a, vec uint16x2 b);

vec int8x4 spi_vlt8i(vec int8x4 a, vec int8x4 b);

vec int8x4 spi_vlt8u(vec uint8x4 a, vec uint8x4 b);

7.2.32.2 Description

Each **spi_vlt*** (vector less than) intrinsic returns the boolean value true (all 1 bits) if **a** is less than **b** and the boolean value false (all 0 bits) otherwise. The arguments may be vectors of signed or unsigned words, packed halfwords, or packed bytes.

In ANSI C, the result of the less than operator '<' is 0 (false) or 1 (true). Because the DPU hardware returns all 1 bits (that is, 0xFF, 0xFFFF or 0xFFFFFFFF) rather than 1 for true, the result of arithmetic using the result of **spi_lt*** may differ from the result in ANSI C.

7.2.32.3 Return value

Returns true (all 1 bits) if **a** is less than **b** and 0 if not.

7.2.32.4 See also

spi_lt: like **spi_vt**, but for scalar arguments rather than vectors.
spi_veq: like **spi_vlt**, but returns equal to rather than strictly less than.
spi_vle: like **spi_vlt**, but returns less than or equal to rather than strictly less than.
spi_vne: like **spi_vlt**, but returns not equal to rather than strictly less than.

7.2.32.5 Examples

Op	**a**	**b**	Result
spi_vlt32i	0x00000002	0x00000002	0x00000000
spi_vlt32i	0x00000002	0xFFFFFFFF	0x00000000
spi_vlt32u	0x00000002	0xFFFFFFFF	0xFFFFFFFF
spi_vlt16i	0x00010002	0x0001FFFF	0x00000000
spi_vlt16u	0x00010002	0x0001FFFF	0x0000FFFF
spi_vlt8i	0x01020304	0x0121FF03	0x00FF0000
spi_vlt8u	0x01020304	0x0121FF03	0x00FFFF00

7.2.33 spi_vmax

7.2.33.1 Prototypes

vec int32x1 spi_vmax32i(vec int32x1 a, vec int32x1 b);

vec int16x2 spi_vmax16i(vec int16x2 a, vec int16x2 b);

vec uint16x2 spi_vmax16u(vec uint16x2 a, vec uint16x2 b);

vec int8x4 spi_vmax8i(vec int8x4 a, vec int8x4 b);

vec uint8x4 spi_vmax8u(vec uint8x4 a, vec uint8x4 b);

7.2.33.2 Description

Each **spi_vmax*** (vector maximum) intrinsic returns the maximum of its arguments. The arguments may be signed or unsigned words, packed halfwords, or packed bytes.

A single hardware operation performs both **spi_vmax*** and the corresponding **spi_vmin***, returning two results. The compiler **spc** merges adjacent paired calls to these functions for efficiency.

7.2.33.3 Return value

Returns the maximum of **a** and **b**.

7.2.33.4 See also

spi_vmin: like **spi_vmax**, but returns the minimum rather than the maximum.

7.2.33.5 Examples

Op	**a**	**b**	Result
spi_vmax32i	0x00000001	0xFFFFFFFF	0x00000001
spi_vmax16i	0x00010002	0x0000FFFF	0x00010002
spi_vmax16u	0x00010002	0x0000FFFF	0x0001FFFF
spi_vmax8i	0x01020304	0x0001FF05	0x01020305
spi_vmax8u	0x01020304	0x0001FF05	0x0102FF05

7.2.34 spi_vmin

7.2.34.1 Prototypes

vec int32x1 spi_vmin32i(vec int32x1 a, vec int32x1 b);

vec int16x2 spi_vmin16i(vec int16x2 a, vec int16x2 b);

vec uint16x2 spi_vmin16u(vec uint16x2 a, vec uint16x2 b);

vec int8x4 spi_vmin8i(vec int8x4 a, vec int8x4 b);

vec uint8x4 spi_vmin8u(vec uint8x4 a, vec uint8x4 b);

7.2.34.2 Description

Each **spi_vmin*** (vector minimum) intrinsic returns the minumum of its arguments. The arguments may be signed or unsigned words, packed halfwords, or packed bytes.

A single hardware operation performs both **spi_vmin*** and the corresponding **spi_vmax***, returning two results. The compiler **spc** merges adjacent paired calls to these functions for efficiency.

7.2.34.3 Return value

Returns the minimum of **a** and **b**.

7.2.34.4 See also

spi_vmax: like **spi_vmin**, but returns the maximum rather than the minimum.

7.2.34.5 Examples

Op	**a**	**b**	Result
spi_vmin32i	0x00000001	0xFFFFFFFF	0xFFFFFFFF
spi_vmin16i	0x00010002	0x0000FFFF	0x0000FFFF
spi_vmin16u	0x00010002	0x0000FFFF	0x00000002
spi_vmin8i	0x01020304	0x0001FF05	0x0001FF04
spi_vmin8u	0x01020304	0x0001FF05	0x00010304

7.2.35 spi_vmula

7.2.35.1 Prototypes

vec int16x2 spi_vmula16i(vec int16x2 a, vec int16x2 b, vec int16x2 c);

vec uint16x2 spi_vmula16u(vec uint16x2 a, vec uint16x2 b, vec uint16x2 c);

vec uint8x4 spi_vmula8u(vec uint8x4 a, vec uint8x4 b, vec uint8x4 c);

7.2.35.2 Description

Each **spi_vmula*** (vector multiplication accumulating) intrinsic returns the saturated result of **a * b + c**. The arguments may be signed or unsigned packed halfwords or packed bytes. DPU hardware does not support full 32-bit * 32-bit to 64-bit multiplication directly.

7.2.35.3 Return value

Returns the saturated result of **a * b + c**.

7.2.35.4 Examples

Op	**a**	**b**	**c**	Result
spi_vmula16i	0x11223344	0x00030003	0x00000000	0x33667FFF
spi_vmula16i	0x11223344	0x00030003	0x00000001	0x33667FFF
spi_vmula16i	0x11223344	0xFFFFFFFF	0x00000000	0xEEDECCBC
spi_vmula16u	0x11223344	0x00030003	0x00000000	0x336699CC
spi_vmula8u	0x11223344	0x03FF03FF	0x00000000	0x33FF99FF

7.2.36 spi_vmuld

7.2.36.1 Prototypes

vec int32x spi_vmuld16i_hi(vec int16x2 a, vec int16x2 b);

vec int32x spi_vmuld16i_lo(vec int16x2 a, vec int16x2 b);

vec uint32x spi_vmuld16u_hi(vec uint16x2 a, vec uint16x2 b);

vec uint32x spi_vmuld16u_lo(vec uint16x2 a, vec uint16x2 b);

vec int32x spi_vmuld16ui_hi(vec uint16x2 a, vec int16x2 b);

vec int32x spi_vmuld16ui_lo(vec uint16x2 a, vec int16x2 b);

vec int16x2 spi_vmuld8i_hi(vec int8x4 a, vec int8x4 b);

vec int16x2 spi_vmuld8i_lo(vec int8x4 a, vec int8x4 b);

vec uint16x2 spi_vmuld8u_hi(vec uint8x4 a, vec uint8x4 b);

vec uint16x2 spi_vmuld8u_lo(vec uint8x4 a, vec uint8x4 b);

vec int16x2 spi_vmuld8ui_hi(vec uint8x4 a, vec int8x4 b);

vec int16x2 spi_vmuld8ui_lo(vec uint8x4 a, vec int8x4 b);

7.2.36.2 Description

Each **spi_vmuld*** (vector multiplication dual output) intrinsic returns a full-precision multiplication result (16-bit * 16-bit to 32-bit result, or a pair of 8-bit * 8-bit to 16-bit results). The arguments may be signed or unsigned packed halfwords or packed bytes. DPU hardware does not support 32-bit * 32-bit to 64-bit multiplication directly (see **spi_vmulha32** and **spi_vmulla32**).

A single hardware operation performs both **spi_muld*_hi** and the corresponding **spi_muld*_lo**, returning two results. The compiler **spc** merges adjacent paired calls to these functions for efficiency.

7.2.36.3 Return value

spi_vmuld16i_hi, **spi_vmuld16u_hi** and **spi_vmuld16ui_hi** return hi(**a) *** hi(**b)**.
spi_vmuld16i_lo, **spi_vmuld16u_lo** and **spi_vmuld16ui_lo** return lo(**a)** * lo(**b)**.

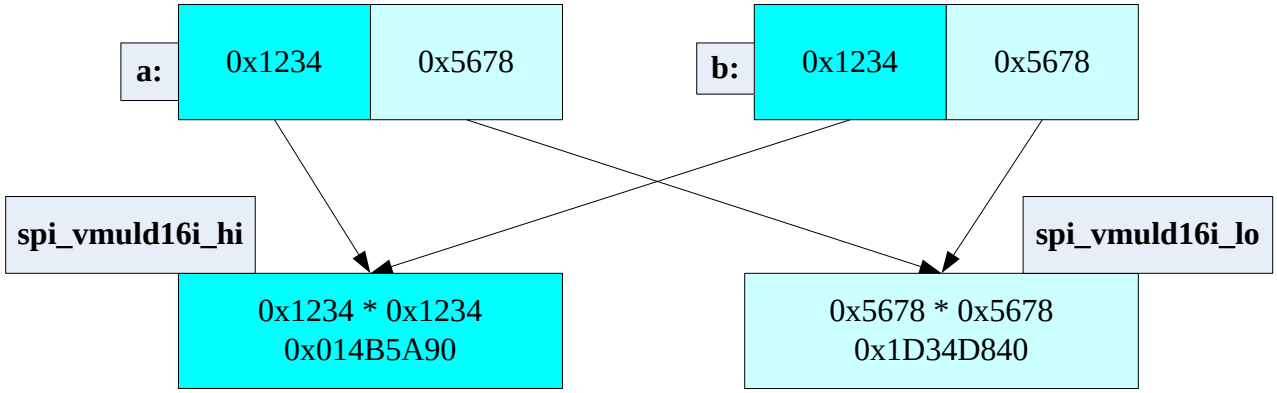
For **spi_vmuld8i_hi**, **spi_vmuld8u_hi** and **spi_vmuld8ui_hi**, the high-order halfword of the result returns the product of the high-order bytes byte3(**a)** * byte3(**b)** and the low-order halfword of the result returns byte2(**a) *** byte2(**b)**.
For **spi_vmuld8i_lo**, **spi_vmuld8u_lo** and **spi_vmuld8ui_lo**, the high-order halfword of the result returns byte1(**a)** * byte1(**b)** and the low-order halfword of the result returns the product of the low-order bytes byte0(**a) *** byte0(**b)**.

7.2.36.4 Examples

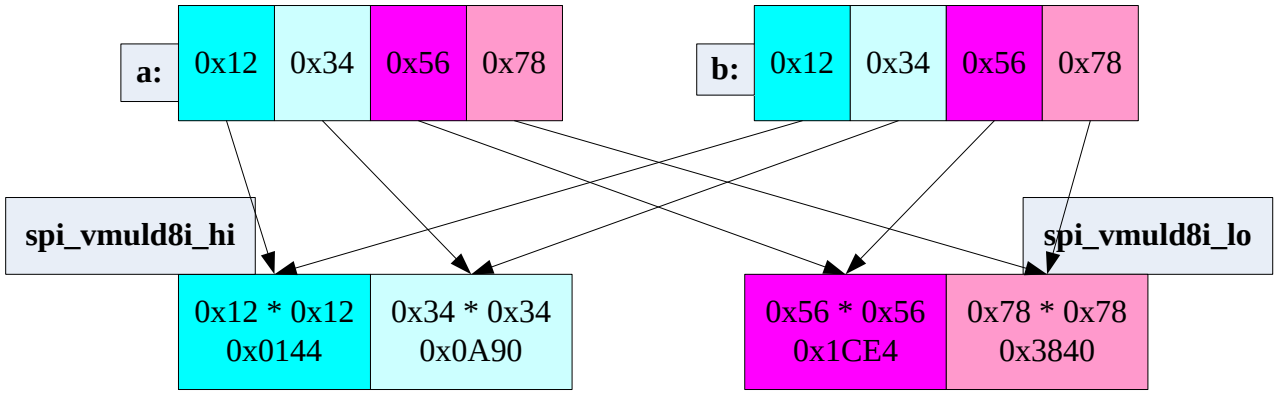
Op	**a**	**b**	**hi** result	**lo** result
spi_vmuld16i_*	0x12345678	0x12345678	0x014B5A90	0x1D34D840
spi_vmuld16u_*	0x12345678	0x12345678	0x014B5A90	0x1D34D840
spi_vmuld16ui_*	0x12345678	0x12345678	0x014B5A90	0x1D34D840
spi_vmuld16i_*	0x12345678	0xFFFFFFFF	0xFFFFEDCC	0xFFFFA988
spi_vmuld16u_*	0x12345678	0xFFFFFFFF	0x1233EDCC	0x5677A988

spi_vmuld16ui_*	0x12345678	0xFFFFFFFF	0xFFFFEDCC	0xFFFFA988
spi_vmuld8i_*	0x12345678	0x12345678	0x01440A90	0x1CE43840
spi_vmuld8u_*	0x12345678	0x12345678	0x01440A90	0x1CE43840
spi_vmuld8ui_*	0x12345678	0x12345678	0x01440A90	0x1CE43840
spi_vmuld8i_*	0x12345678	0xFFFFFFFF	0xFFEEFFCC	0xFFAAFF88
spi_vmuld8u_*	0x12345678	0xFFFFFFFF	0x11EE33CC	0x55AA7788
spi_vmuld8ui_*	0x12345678	0xFFFFFFFF	0xFFEEFFCC	0xFFAAFF88

The following diagram illustrates the first **spi_vmuld16i_*** example in the table above:



The following diagram illustrates the first **spi_vmuld8i_*** example in the table above:



7.2.37 spi_vmulha32

7.2.37.1 Prototypes

vec int32x1 spi_vmulha32i_hi(vec int16x2 a, vec int32x1 b, vec uint32x1 c);

vec uint32x1 spi_vmulha32i_lo(vec int16x2 a, vec int32x1 b, vec uint32x1 c);

7.2.37.2 Description

The **spi_vmulha32i*** (vector multiplication high with accumulation) intrinsics are useful when executing 32-bit by 32-bit to 64-bit multiplication or for executing 16-bit by 32-bit multiply-add operations with a 64-bit signed accumulator. They can be used with **spi_vmulla32ui**, **spi_vshuffledi** and **spi_add32i** to execute a signed 32-bit by 32-bit to 64-bit multiply, as shown in the example below.

A single hardware operation performs both **spi_vmulha32i_hi** and **spi_vmulha32i_lo**, returning two results. The compiler **spc** merges adjacent paired calls to these functions for efficiency.

7.2.37.3 Return value

spi_vmulha32i_hi computes the 48-bit product hi(**a**) * **b**, sign extends the result to 64 bits, adds **c**, and returns the high-order 32 bits of the 64 bit result, while **spi_vmulha32i_lo** returns the low-order 32 bits of the same 64-bit result.

7.2.37.4 See also

spi_vmulla32: like **spi_vmulha32**, but using the low-order halfword of **a** rather than the high-order halfword.

7.2.37.5 Examples

Op	**a**	**b**	**c**	**hi** result	**lo** result
spi_vmulha32i_*	0x1111AAAA	0x33333333	0x00000000	0x00000369	0xCCCCC963
spi_vmulha32i_*	0x1111AAAA	0x33333333	0x00000001	0x00000369	0xCCCCC964

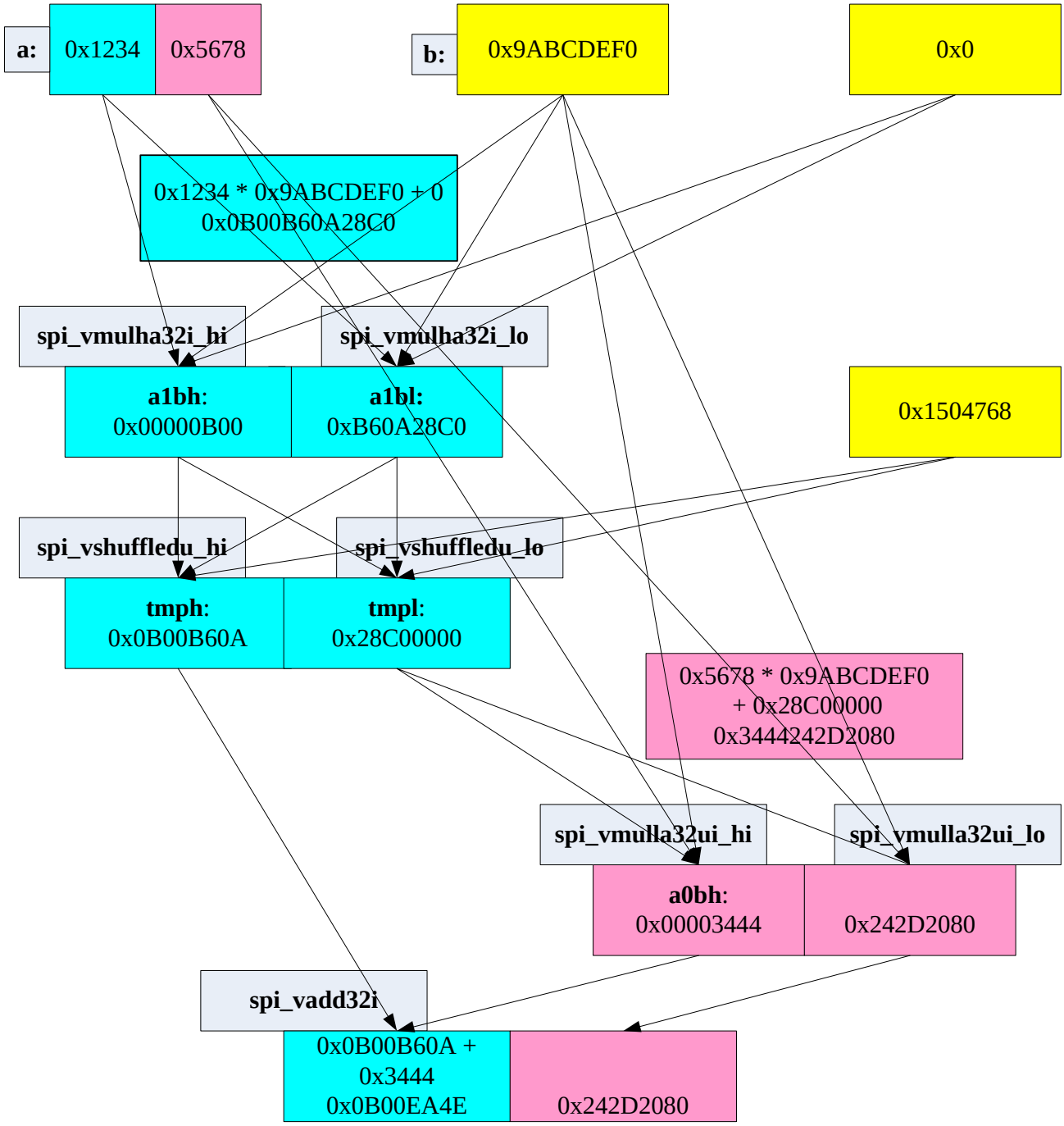
The following kernel function performs a signed 32-bit by 32-bit to 64-bit multiply. **spi_vmulha32i** computes the 48-bit partial product hi(**a**) * **b** and **spi_vshuffledu** shifts the 64-bit result left by 16 bits. Then **spi_vmulla32i** computes the 48-bit partial product lo(**a**) * **b** plus the low-order 32 bits of the previously computed partial product, and finally **spi_vadd32i** adds the high-order 32 bits of the previously computed partial product.

```
// Signed 32-bit by 32-bit to 64-bit multiplication.
kernel      void  mul64i(
      vec   int32x1           a(in),
      vec   int32x1           b(in),
      vec   int32x1           p_hi(out),
      vec   uint32x1          p_lo(out)
      )
{
      vec   int32x1           a1bh, a0bh, tmph;
      vec   uint32x1          a1bl, a0bl, tmpl;

      a1bh = spi_vmulha32i_hi((vec int16x2)a, b, 0);
      a1bl = spi_vmulha32i_lo((vec int16x2)a, b, 0);
      tmph = spi_vshuffledu_hi(0x15047868, (vec uint32x1)a1bh, a1bl);
      tmpl = spi_vshuffledu_lo(0x15047868, (vec uint32x1)a1bh, a1bl);
```

```
        a0bh = spi_vmulla32ui_hi((vec int16x2)a, b, tmpl);
        p_lo = spi_vmulla32ui_lo((vec int16x2)a, b, tmpl);
        p_hi = spi_vadd32i(tmph, a0bh);
}
```

The following diagram applies the above code to the product 0x12345678 * 0x9ABCDEF0 = 0x0B00EA4E242D2080.



7.2.38 spi_vmulha16

7.2.38.1 Prototypes

vec int16x2 spi_vmulha16i(vec int16x2 a, vec int16x2 b, vec uint16x2 c);

vec uint16x2 spi_vmulha16u(vec uint16x2 a, vec uint16x2 b, vec uint16x2 c);

vec int16x2 spi_vmulha16ui(vec uint16x2 a, vec int16x2 b, vec uint16x2 c);

7.2.38.2 Description

Each **spi_vmulha16*** (vector multiplication high with accumulation) intrinsic computes **a * b + c** and returns the high-order halfword of the shifted saturated result.

7.2.38.3 Return value

spi_vmulha16i computes **a** * **b** + **c**, , right shifts by 15 bits, and saturates the result.
spi_vmulha16u computes **a** * **b** + **c**, right shifts by 16 bits, and saturates the result.
spi_vmulha16ui computes **a** * **b** + **c** and right shifts by 16 bits. The result does not exceed the **uint16x2** range, so no saturation is required.

Argument **c** is added before shifting, not after; **c** is typically used for rounding to a desired precision (not for accumulation!).

7.2.38.4 Examples

Op	**a**	**b**	**c**	Result	Comment
spi_vmulha16i	0x4000C000	0x40004000	0x00000000	0x2000E000	.5 * .5 = .25 : -.5 * .5 = -.25
spi_vmulha16u	0x4000C000	0x40004000	0x00000000	0x10003000	.25 * .25 = .0625 : .75 * .25 = .1875
spi_vmulha16ui	0x4000C000	0x40004000	0x00000000	0x10003000	.25 * .5 = .125 : .75 * .5 = .375
spi_vmulha16i	0x01800180	0x01800180	0x00004000	0x00040005	truncation : rounding
spi_vmulha16u	0x11181118	0x10001000	0x00008000	0x01110112	truncation : rounding

7.2.39 spi_vmulla32

7.2.39.1 Prototypes

vec int32x1 spi_vmulla32i_hi(vec int16x2 a, vec int32x1 b, vec uint32x1 c);

vec uint32x1 spi_vmulla32i_lo(vec int16x2 a, vec int32x1 b, vec uint32x1 c);

vec int32x1 spi_vmulla32ui_hi(vec uint16x2 a, vec int32x1 b, vec uint32x1 c);

vec uint32x1 spi_vmulla32ui_lo(vec uint16x2 a, vec int32x1 b, vec uint32x1 c);

7.2.39.2 Description

The **spi_vmulla32i*** (vector multiplication low with accumulation) intrinsics are useful when executing 32-bit by 32-bit to 64-bit multiplication or for executing 16-bit by 32-bit multiply-add operations with a 64-bit signed accumulator. They can be used with **spi_vmulla32ui**, **spi_add32i**, and **spi_vshuffledi** to execute a 32-bit by 32-bit to 64-bit multiply.

A single hardware operation performs both **spi_vmulla32*_hi** and **spi_vmulla32*_lo**, returning two results. The compiler **spc** merges adjacent paired calls to these functions for efficiency.

7.2.39.3 Return value

spi_vmulla32i_hi computes the 48-bit product lo(**a**) * **b**, sign extends the result to 64 bits, adds **c**, and returns the high-order 32 bits of the 64 bit result, while **spi_vmulha32i_lo** returns the low-order 32 bits of the same 64-bit result. **spi_vmulla32ui_hi** and **spi_vmulha32ui_lo** are similar, but with unsigned **a** operand.

7.2.39.4 See also

spi_vmulha32: like **spi_vmulla32**, but using the high-order halfword of **a** rather than the low-order halfword.

7.2.39.5 Examples

Op	**a**	**b**	**c**	**hi** result	**lo** result
spi_vmulla32i_*	0x1111AAAA	0x33333333	0x00000000	0xFFFFEEEE	0xCCCCDDDE
spi_vmulla32i_*	0x1111AAAA	0x33333333	0x00000001	0xFFFFEEEE	0xCCCCDDDF
spi_vmulla32ui_*	0x1111AAAA	0x33333333	0x00000000	0x00002221	0xFFFFDDDE

The **spi_vmulha32** page contains sample Stream code that uses **spi_vmulla32** to perform a 32-bit times 32-bit to 64-bit product. It also includes a diagram that illustrates the use of **spi_vmulla32ui_***.

7.2.40 spi_vmulra

7.2.40.1 Prototypes

vec int16x2 spi_vmulra16i(vec int16x2 a, vec int16x2 b, vec int16x2 c);

vec uint16x2 spi_vmulra16u(vec uint16x2 a, vec uint16x2 b, vec uint16x2 c);

vec int16x2 spi_vmulra16ui(vec uint16x2 a, vec int16x2 b, vec int16x2 c);

7.2.40.2 Description

Each **spi_vmulra*** (multiplication with rounding and accumulation) intrinsic returns a rounded saturated result of **a * b + c**.

spi_vmulra16i is suitable for the fractional multiplication of signed values; it multiplies **a * b** to obtain a 32-bit value, adds 0x4000, shifts right 15 places, adds **c**, and saturates the result.

spi_vmulra16u is suitable for the fractional multiplication of unsigned values; it multiplies **a * b** to obtain a 32-bit value, adds 0x8000, shifts right 16 places, adds **c,** and saturates the result.

spi_vmulra16ui is suitable for fractional multiplication of unsigned times signed values; it multiplies **a * b** to obtain a 32-bit value, adds 0x8000, shifts right 16 places, adds **c,** and saturates the result.

7.2.40.3 Return value

Returns the rounded saturated result of **a * b + c**.

7.2.40.4 See also

spi_mulrs: like **spi_vmulra**, except returns **c - a * b** instead.

7.2.40.5 Examples

Op	**a**	**b**	**c**	Result
spi_vmulra16i	0x11223344	0x11111111	0x00000000	0x024906D6
spi_vmulra16i	0x11223344	0x11111111	0x00000001	0x024906D7
spi_vmulra16i	0x11223344	0xFE000000	0x00000000	0xFFBB0000
spi_vmulra16i	0x11223344	0xFE000000	0x00000001	0xFFBB0001
spi_vmulra16u	0x11223344	0x11111111	0x00000000	0x0124036B
spi_vmulra16u	0x11223344	0x11111111	0x00000001	0x0124036C
spi_vmulra16ui	0x11223344	0xFE000000	0x00000000	0x11000000
spi_vmulra16ui	0x11223344	0xFE000000	0x00000001	0x11000001

7.2.41 spi_vne

7.2.41.1 Prototypes

vec int32x1 spi_vne32(vec uint32x1 a, vec uint32x1 b);

vec int16x2 spi_vne16(vec uint16x2 a, vec uint16x2 b);

vec int8x4 spi_vne8(vec uint8x4 a, vec uint8x4 b);

7.2.41.2 Description

Each **spi_vne*** (vector not equals) intrinsic returns the boolean value true (all 1 bits) if **a** is not equal to **b** and the boolean value false (all 0 bits) if **a** is equal to **b**. The arguments may be vectors of words, packed halfwords, or packed bytes.

In ANSI C, the result of the inequality operator '!=' is 0 (false) or 1 (true). Because the DPU hardware returns all 1 bits (that is, 0xFF, 0xFFFF or 0xFFFFFFFF) rather than 1 for true, the result of arithmetic using the result of **spi_ne*** may differ from the result in ANSI C.

7.2.41.3 Return value

Returns true (all 1 bits) if **a** is not equal to **b** and 0 if not.

7.2.41.4 See also

spi_ne: like **spi_vne**, but for scalar arguments rather than vectors.
spi_veq: like **spi_vne**, but uses equals rather than not equals.
spi_vle: like **spi_vne**, but uses less than or equal to rather than not equals.
spi_vlt: like **spi_vne**, but uses less than rather than not equals.

7.2.41.5 Examples

Op	**a**	**b**	Result
spi_vne32	0x00000002	0x00000002	0x00000000
spi_vne32	0x00000002	0xFFFFFFFF	0xFFFFFFFF
spi_vne16	0x00010002	0x0001FFFF	0x0000FFFF
spi_vne8	0x01020304	0x01FFFF04	0x00FFFF00

7.2.42 spi_vnorm

7.2.42.1 Prototypes

vec int32x1 spi_vnorm32(vec int32x1 a);

vec int16x2 spi_vnorm16(vec int16x2 a);

7.2.42.2 Description

Each **spi_vnorm*** (vector normalize) intrinsic returns the number of redundant sign bits of its argument.

7.2.42.3 Return value

Returns the number of redundant sign bits of its argument.

7.2.42.4 Examples

Op	**a**	Result
spi_vnorm32	0x00000000	0x0000001F
spi_vnorm32	0x00000004	0x0000001C
spi_vnorm32	0xFFFFFFFC	0x0000001D
spi_vnorm16	0x0004FFFC	0x000C000D

7.2.43 spi_vnot

7.2.43.1 Prototype

vec uint32x1 spi_vnot32(vec uint32x1 a);

vec uint16x2 spi_vnot16(vec uint16x2 a);

vec uint8x4 spi_vnot8(vec uint8x4 a);

7.2.43.2 Description

Each **spi_vnot*** (vector NOT) intrinsic returns the bitwise complement of its argument. The argument may be a vector word, packed halfwords, or packed bytes.

spi_vnot32, **spi_vnot16** and **spi_vnot8** all represent the same DPU hardware operation. They have different names to allow strict type checking of arguments and result.

7.2.43.3 Return value

Each return value bit is 1 if the corresponding bit in **a** is 0, otherwise it is 1.

7.2.43.4 See also

spi_vand: like **spi_vnot**, but uses bitwise AND rather than bitwise NOT.
spi_vor: like **spi_vnot**, but uses bitwise OR rather than bitwise NOT.
spi_not: like **spi_vnot**, but for a vector argument rather than a scalar.
spi_vxor: like **spi_vnot**, but uses bitwise XOR rather than bitwise NOT.

7.2.43.5 Examples

Op	**a**	Result
spi_vnot32	0x12345678	0xEDCBA987
spi_vnot16	0x12345678	0xEDCBA987
spi_vnot8	0x12345678	0xEDCBA987

7.2.44 spi_vor

7.2.44.1 Prototypes

vec uint32x1 spi_vor32(vec uint32x1 a, vec uint32x1 b);

vec uint16x2 spi_vor16(vec uint16x2 a, vec uint16x2 b);

vec uint8x4 spi_vor8(vec uint8x4 a, vec uint8x4 b);

7.2.44.2 Description

Each **spi_vor*** (vector OR) intrinsic returns the bitwise OR of its arguments. The arguments may be words, packed halfwords, or packed bytes.

spi_vor32, **spi_vor16** and **spi_vor8** all represent the same DPU hardware operation. They have different names to allow strict type checking of arguments and result.

7.2.44.3 Return value

Each return value bit is 1 if the corresponding bit in either argument is 1, otherwise it is 0.

7.2.44.4 See also

spi_vand: like **spi_vor**, but uses bitwise AND rather than bitwise OR.
spi_vnot: like **spi_vor**, but uses bitwise NOT rather than bitwise OR.
spi_or: like **spi_vor**, but for vector arguments rather than scalars.
spi_vorl: like **spi_vor**, but performs cross-lane reduction OR rather than OR in each lane.
spi_vxor: like **spi_vor**, but uses bitwise XOR rather than bitwise OR.

7.2.44.5 Examples

Op	**a**	**b**	Result
spi_vor32	0x12345678	0x00FF00FF	0x12FF56FF
spi_vor16	0x12345678	0x00FF00FF	0x12FF56FF
spi_vor8	0x12345678	0x00FF00FF	0x12FF56FF

7.2.45 spi_vperm

7.2.45.1 Prototypes

vec uint32x1 spi_vperm32(vec uint32x1 a, vec uint32x1 b, uint32x1 c);

vec uint16x2 spi_vperm16(vec uint16x2 a, vec uint16x2 b, uint16x2 c);

vec uint8x4 spi_vperm8(vec uint8x4 a, vec uint8x4 b, uint8x4 c);

7.2.45.2 Description

The **spi_vperm*** (vector permute) intrinsics permute data between the lanes of the stream processor and the scalar operand register file (SORF). Argument **a** is a vector control value, **b** is vector data and **c** is scalar data.

A single hardware operation performs both **spi_perm*** and the corresponding **spi_vperm*,** returning both a scalar and a vector result. The compiler **spc** merges adjacent paired calls to these functions for efficiency.

7.2.45.3 Return value

Control value **a** is a vector, with a different value in each lane. The result of **spi_vperm*** in each lane depends on the value of **a** in that lane. The descriptions below apply identically to each lane, but since **a** has a different value in each lane the result of **spi_vperm*** differs in each lane.

For **spi_vperm32**, control value **a** specifies a lane or scalar selector *n*. If *n* is between 0 and 15, **spi_vperm32** returns the value of vector **b** in lane *n*. If *n* is 16, **spi_vperm32** returns scalar **c**. If *n* is greater than 16, the behavior is undefined.

For **spi_vperm16**, bit 0 of control value **a** specifies a halfword (0 or 1) *m* and bits 5:1 of **a** specify a lane (0 to 15) or scalar (16) *n*. If *n* is between 0 and 15, **spi_vperm16** returns the value of halfword *m* of vector **b** in lane *n***.** If *n* is 16, **spi_vperm16** returns the value of halfword *m* of scalar **c**. If *n* is greater than 16, the behavior is undefined.

For **spi_vperm8**, bits 1:0 of **a** specifies a byte (0 to 3) *m* and bits 6:2 of **a** specify a lane (0 to 15) or scalar (16) *n*. If *n* is between 0 and 15, **spi_vperm8** returns the value of byte *m* of vector **b** in lane *n***.** If *n* is 16, **spi_vperm8** returns the value of byte *m* of scalar **c**. If *n* is greater than 16, the behavior is undefined.

7.2.45.4 See also

spi_perm: like **spi_vperm**, but takes a scalar control value and returns a scalar result rather than a vector.
spi_vshuffle: like **spi_vperm**, but shuffles bytes within a lane rather than permuting data between lanes.

7.2.45.5 Examples

Op	**a** lane *n*	**b** lane 0	**b** lane 1	**...**	**b** lane 15	**c**	Result lane *n*
spi_vperm32	0x00000001	0x03020100	0x13121110	...	0xF3F2F1F0	0xDEADBEEF	0x13121110
spi_vperm32	0x00000010	0x03020100	0x13121110	...	0xF3F2F1F0	0xDEADBEEF	0xDEADBEEF
spi_vperm16	0x00030021	0x03020100	0x13121110	...	0xF3F2F1F0	0xDEADBEEF	0x1312DEAD
spi_vperm8	0x00053E43	0x03020100	0x13121110	...	0xF3F2F1F0	0xDEADBEEF	0x0011F2DE

The following diagram illustrates the **spi_vperm8** example from the table above. **spi_vperm8** arguments and result are of type **uint8x4**, i.e., four packed unsigned bytes. The diagram shows each byte of each value separately. Control value **a** is a vector, with a different value in each lane; this diagram shows the result for a single lane.

Control argument **a**, with each byte also given in binary to show the lane (bits 6:2) and component (bits 1:0) it selects:

a:	00	05	3E	43
Binary	0.00000.00	0.00001.01	0.01111.10	0.10000.11
Lane	0	1	15	Scalar (16)
Component	0	1	2	3

Vector argument **b** and scalar argument **c,** with colors indicating the lanes and components selected by control argument **a**:

	Lane 0				Lane 1				...	Lane 15					Scalar			
b:	03	02	01	00	13	12	11	10	...	F3	F2	F1	F0	**c**:	DE	AD	BE	EF

The result:

Result:	00	11	F2	DE

7.2.46 spi_vrandl

7.2.46.1 Prototype

uint32x1 spi_vrandl(vec uint32x1 a);

7.2.46.2 Description

spi_vrandl (vector reduction AND across lanes) performs a bitwise AND of the least significant bit (bit 0) of **a** in each lane. It returns a scalar value, not a vector.

7.2.46.3 Return value

Returns 0xFFFFFFFF (true) if the least significant bit of **a** in every lane is 1, otherwise returns 0 (false).

7.2.46.4 See also

spi_vand: like **spi_vrandl**, but ANDs two arguments in each lane rather than performing cross-lane reduction AND.
spi_vlsbs: like **spi_vrandl**, but packs least significant bits into a result rather than ANDing them together.
spi_vrandlv: like **spi_vrandl**, but returns a vector result rather than a scalar.
spi_vrorl: like **spi_vrandl**, but uses bitwise OR rather than bitwise AND.

7.2.46.5 Examples

Op	**a** lane 0	**a** lane 1	**...**	**a** lane 14	**a** lane 15	Result	Comment
spi_vrandl	0x00000000	0x00000000	...	0x00000000	0x00000000	0x00000000	lsb 0 in all lanes
spi_vrandl	0x00000000	0x00000001	...	0x00000000	0x00000000	0x00000000	lsb nonzero in 1 lane
spi_vrandl	0x00000001	0x00000003	...	0x00007FFF	0x0000FFFF	0xFFFFFFFF	lsb nonzero in all lanes

7.2.47 spi_vrandlv

7.2.47.1 Prototype

vec uint32x1 spi_vrandlv(vec uint32x1 a);

7.2.47.2 Description

spi_vrandlv (vector reduction AND across lanes with vector result) performs a bitwise AND of the least significant bit (bit 0) of **a** in each lane. It returns a vector with the same value in each lane.

7.2.47.3 Return value

Returns 0xFFFFFFFF (true) if the least significant bit of **a** in every lane is 1, otherwise returns 0 (false).

7.2.47.4 See also

spi_vand: like **spi_vrandlv**, but ANDs two arguments in each lane rather than performing cross-lane reduction AND.
spi_vlsbs: like **spi_vrandlv**, but packs least significant bits into a result rather than ANDing them together.
spi_vrandl: like **spi_vrandlv**, but returns a scalar result rather than a vector.
spi_vrorl: like **spi_vrandlv**, but uses bitwise OR rather than bitwise AND.

7.2.47.5 Examples

Op	**a** lane 0	**a** lane 1	**...**	**a** lane 14	**a** lane 15	Result	Comment
spi_vrandlv	0x00000000	0x00000000	...	0x00000000	0x00000000	0x00000000	lsb 0 in all lanes
spi_vrandlv	0x00000000	0x00000001	...	0x00000000	0x00000000	0x00000000	lsb nonzero in 1 lane
spi_vrandlv	0x00000001	0x00000003	...	0x00007FFF	0x0000FFFF	0xFFFFFFFF	lsb nonzero in all lanes

7.2.48 spi_vrorl

7.2.48.1 Prototype

uint32x1 spi_vrorl(vec uint32x1 a);

7.2.48.2 Description

spi_vrorl (vector reduction OR across lanes) performs a bitwise OR of the least significant bit (bit 0) of **a** in each lane. It returns a scalar value, not a vector.

7.2.48.3 Return value

Returns 0xFFFFFFFF (true) if the least significant bit of **a** in any lane is 1, otherwise returns 0 (false).

7.2.48.4 See also

spi_vor: like **spi_vrorl**, but ORs two arguments in each lane rather than performing cross-lane reduction OR.
spi_vrandl: like **spi_vrorl**, but uses bitwise AND rather than bitwise OR.
spi_vlsbs: like **spi_vrorl**, but packs least significant bits into a result rather than ORing them together.
spi_vrorlv: like **spi_vrorl**, but returns a vector result rather than a scalar.

7.2.48.5 Examples

Op	**a** lane 0	**a** lane 1	**...**	**a** lane 14	**a** lane 15	Result	Comment
spi_vrorl	0x00000000	0x00000000	...	0x00000000	0x00000000	0x00000000	lsb 0 in all lanes
spi_vrorl	0x00000000	0x00000001	...	0x00000000	0x00000000	0xFFFFFFFF	lsb nonzero in 1 lane
spi_vrorl	0x00000001	0x00000003	...	0x00007FFF	0x0000FFFF	0xFFFFFFFF	lsb nonzero in all lanes

7.2.49 spi_vsad

7.2.49.1 Prototype

vec uint32x1	**spi_vsad8u(vec uint8x4 a, vec uint8x4b);**	// Storm-2 only
vec uint16x2	**spi_vsaddh8u(vec uint8x4 a, vec uint8x4b, vec uint8x4c);**	// Storm-2 only
vec uint16x2	**spi_vsaddl8u(vec uint8x4 a, vec uint8x4b, vec uint8x4c);**	// Storm-2 only

7.2.49.2 Description

spi_vsad (vector sum of absolute differences) computes the sum of the absolute differences between four successive bytes. **spi_vsad8u** computes the sum of the absolute differences between the corresponding bytes of **a** and **b** and returns the sum as a single 32-bit value. Each **spi_vsadd*** intrinsic is similar, but instead computes a packed result containing two sums of the absolute differences between bytes of **a** and bytes of the eight-byte sequence in **b** and **c**, with differing byte offsets, as explained in detail below.

spi_vsad intrinsics exist on Storm-2 only, not on Storm-1.

7.2.49.3 Return value

spi_vsad8u returns | byte3(**a**) - byte3(**b**) | + | byte2(**a**) - byte2(**b**) | + | byte1(**a**) - byte1(**b**) | + | byte0(**a**) - byte0(**b**) |.

spi_vsaddh8u returns | byte3(**a**) - byte3(**b**) | + | byte2(**a**) - byte2(**b**) | + | byte1(**a**) - byte1(**b**) | + | byte0(**a**) - byte0(**b**) | in the low-order halfword and | byte3(**a**) - byte2(**b**) | + | byte2(**a**) - byte1(**b**) | + | byte1(**a**) - byte0(**b**) | + | byte0(**a**) - byte3(**c**) | in the high-order halfword. Thus, the low-order halfword performs the same computation as **spi_vsad8u**, while the high-order halfword performs a similar computation starting with byte 2 of **b**.

spi_vsaddl8u returns | byte3(**a**) - byte1(**b**) | + | byte2(**a**) - byte0(**b**) | + | byte1(**a**) - byte3(**c**) | + | byte0(**a**) - byte2(**c**) | in the low-order halfword and | byte3(**a**) - byte0(**b**) | + | byte2(**a**) - byte3(**c**) | + | byte1(**a**) - byte2(**c**) | + | byte0(**a**) - byte1(**c**) | in the high-order halfword. Thus, the low-order halfword performs the computation starting with byte 1 of **b**, while the high-order halfword performs the computation starting with the low-order byte of **b**.

7.2.49.4 See also

spi_vabd: like **spi_vsad**, but returns the absolute difference of its arguments rather than a sum of absolute differences.

7.2.49.5 Examples

Op	**a**	**b**	**c**	Result	
spi_vsad8u	0x00010203	0x00010203		0x00000000	Storm-2 only
spi_vsaddh8u	0x00010203	0x00010203	0x04050607	0x00040000	Storm-2 only
spi_vsaddl8u	0x00010203	0x00010203	0x04050607	0x000C0008	Storm-2 only

In the examples above, the **spi_vsad8u** result is | 0-0 | + | 1-1 | + | 2-2 | + | 3-3 | = 0 + 0 + 0 + 0 = 0, the high-order halfword **spi_vsaddh8u** result is | 0-1 | + | 1-2 | + | 2-3 | + | 3-4 | = 1 + 1 + 1 + 1 = 4, and so on.

7.2.50 spi_vsclip

7.2.50.1 Prototype

vec uint8x4 **spi_vsclip16i(uint32x1 a, vec int16x2 b, vec int16x2 c);** // Storm-2 only; **a** constant

vec int16x2 **spi_vsclip32i(uint32x1 a, vec int32x1 b, vec int32x1 c);** // Storm-2 only; **a** constant

7.2.50.2 Description

Each **spi_vsclip*** (vector shift and clip) intrinsic shifts its **b** and **c** arguments right by the value of the **a** argument, clips the intermediate results, and returns a packed result. Scalar shift count **a** must be a compile-time constant in the range 0 to 15 inclusive, otherwise **spc** will report an error. For **spi_vsclip16i**, arguments **b** and **c** are packed signed halfwords, the shifted signed intermediate results are clipped to unsigned bytes (0 to 255), and the result is packed unsigned bytes. For **spi_vsclip32i**, arguments **b** and **c** are signed words, the shifted signed intermediate results are clipped to signed halfwords (-32768 to 32767), and the result is packed signed halfwords.

spi_vsclip intrinsics exist on Storm-2 only, not on Storm-1.

7.2.50.3 Return value

For **spi_vsclip16i**, successive bytes of the result (high-order to low-order) contain clip8u(hi(**b**) >> **a**), clip8u(lo(**b**) >> **a**), clip8u(hi(**c**) >> **a**), and clip8u(lo(**c**) >> **a**).

For **spi_vsclip32i**, the high-order result halfword contains clip16s(**b**) and the low-order result halfword contains clip16s(**c**).

7.2.50.4 See also

spi_vclip: like **spi_vsclip**, but does not shift its arguments before clipping.

7.2.50.5 Examples

Op	**a**	**b**	**c**	Result	
spi_vsclip16i	0x00000000	0xFEDCBA98	0x76543210	0x0000FFFF	Storm-2 only
spi_vsclip16i	0x00000008	0xFEDCBA98	0x76543210	0x00007632	Storm-2 only
spi_vsclip32i	0x00000000	0xFEDCBA98	0x76543210	0x80007FFF	Storm-2 only
spi_vsclip32i	0x0000000C	0xFEDCBA98	0x76543210	0xEDCB7FFF	Storm-2 only

7.2.51 spi_vrorlv

7.2.51.1 Prototype

vec uint32x1 spi_vrorlv(vec uint32x1 a);

7.2.51.2 Description

spi_vrorlv (vector reduction OR across lanes with vector result) performs a bitwise OR of the least significant bit (bit 0) of **a** in each lane. It returns a vector that contains the same result in each lane.

7.2.51.3 Return value

Returns 0xFFFFFFFF (true) if the least significant bit of **a** in any lane is 1, otherwise returns 0 (false).

7.2.51.4 See also

spi_vor: like **spi_vrorlv**, but ORs two arguments in each lane rather than performing cross-lane reduction OR.
spi_vrandl: like **spi_vrorlv**, but uses bitwise AND rather than bitwise OR.
spi_vlsbs: like **spi_vrorlv**, but packs least significant bits into a result rather than ORing them together.
spi_vrorl: like **spi_vrorlv**, but returns a scalar result rather than a vector.

7.2.51.5 Examples

Op	**a** lane 0	**a** lane 1	**...**	**a** lane 14	**a** lane 15	Result	Comment
spi_vrorlv	0x00000000	0x00000000	...	0x00000000	0x00000000	0x00000000	lsb 0 in all lanes
spi_vrorlv	0x00000000	0x00000001	...	0x00000000	0x00000000	0xFFFFFFFF	lsb nonzero in 1 lane
spi_vrorlv	0x00000001	0x00000003	...	0x00007FFF	0x0000FFFF	0xFFFFFFFF	lsb nonzero in all lanes

7.2.52 spi_vselect

7.2.52.1 Prototypes

vec uint32x1 spi_vselect32(vec uint32x1 a, vec uint32x1 b, vec uint32x1 c);

vec uint16x2 spi_vselect16(vec uint32x1 a, vec uint16x2 b, vec uint16x2 c);

vec uint8x4 spi_vselect8(vec uint32x1 a, vec uint8x4 b, vec uint8x4 c);

7.2.52.2 Description

Each **spi_vselect*** (vector select) intrinsic selects byte *n* of argument **b** if the low-order bit (bit 0) of byte *n* of **a** is 1, otherwise it selects byte *n* of argument **c**.

spi_vselect32, **spi_vselect16** and **spi_vselect8** all represent the same DPU hardware operation. They have different names to allow strict type checking of arguments and result.

In ANSI C, conditional operations such as **a ? b : c** depend on whether **a** is nonzero (true, selecting **b**) or zero (false, selecting **c**). Because the DPU hardware checks bit 0 of each byte of condition **a**, the result of **spi_vselect** may differ from the result of **a ? b : c** in ANSI C.

Since all three versions of **spi_vselect*** use the bottom bit *of each byte* of the control value to decide which argument to select, the user should construct the control argument using DPU boolean intrinsics (which return all 1 bits for true) rather than the corresponding C boolean operators (which return 1 for true), or alternatively manipulate the C boolean result appropriately before using it as a **spi_vselect*** control argument.

7.2.52.3 Return value

In each byte, returns argument **b** if the low-order bit of **a** is 1, otherwise returns argument **c**.

7.2.52.4 See also

spi_select: like **spi_vselect**, but returns a scalar result rather than a vector.
spi_vselectd: like **spi_vselect**, but returns both the selected and the non-selected values.

7.2.52.5 Examples

Op	**a**	**b**	**c**	Result
spi_vselect32	0x00000000	0x11223344	0x55667788	0x55667788
spi_vselect32	0xFFFFFFFF	0x11223344	0x55667788	0x11223344
spi_vselect32	0x0000FFFF	0x11223344	0x55667788	0x55663344
spi_vselect32	0x00FF00FF	0x11223344	0x55667788	0x55227744
spi_vselect32	0x00010203	0x11223344	0x55667788	0x55227744
spi_vselect16	0x00010203	0x11223344	0x55667788	0x55227744
spi_vselect8	0x00010203	0x11223344	0x55667788	0x55227744

7.2.53 spi_vselectd

7.2.53.1 Prototypes

vec uint32x1 spi_vselectd32_hi(vec uint32x1 a, vec uint32x1 b, vec uint32x1 c);

vec uint32x1 spi_vselectd32_lo(vec uint32x1 a, vec uint32x1 b, vec uint32x1 c);

vec uint16x2 spi_vselectd16_hi(vec uint32x1 a, vec uint16x2 b, vec uint16x2 c);

vec uint16x2 spi_vselectd16_lo(vec uint32x1 a, vec uint16x2 b, vec uint16x2 c);

vec uint8x4 spi_vselectd8_hi(vec uint32x1 a, vec uint8x4 b, vec uint8x4 c);

vec uint8x4 spi_vselectd8_lo(vec uint32x1 a, vec uint8x4 b, vec uint8x4 c);

7.2.53.2 Description

spi_vselectd (vector select) is like **spi_vselect**, but it returns both the selected and nonselected values. Each **spi_vselectd*_hi** intrinsic selects byte *n* of argument **b** if the low-order bit (bit 0) of byte *n* of **a** is 0, otherwise it selects byte *n* of argument **c**. Each **spi_vselectd*_lo** intrinsic returns the non-selected bytes, namely byte *n* of argument **b** if the low-order bit of byte *n* of **a** is 1, otherwise byte *n* of argument **c**.

spi_vselectd32, **spi_vselectd16** and **spi_vselectd8** all represent the same DPU hardware operation. They have different names to allow strict type checking of arguments and result.

In ANSI C, conditional operations such as **a ? b : c** depend on whether **a** is nonzero (true, selecting **b**) or zero (false, selecting **c**). Because the DPU hardware checks bit 0 of each byte of condition **a**, the result of **spi_vselect** may differ from the result of **a ? b : c** in ANSI C.

Since all three versions of **spi_vselectd*** use the bottom bit *of each byte* of the control value to decide which argument to select, the user should construct the control argument using DPU boolean intrinsics (which return all 1 bits for true) rather than the corresponding C boolean operators (which return 1 for true), or alternatively manipulate the C boolean result appropriately before using it as a **spi_vselect*** control argument.

7.2.53.3 Return value

In each byte, **spi_vselectd*_hi** returns argument **b** if the low-order bit of **a** is 0, otherwise it returns argument **c**. In each byte, **spi_vselectd*_lo** returns argument **b** if the low-order bit of **a** is 1, otherwise it returns argument **c**.

7.2.53.4 See also

spi_vselect: like **spi_vselectd**, but returns only the selected result.

7.2.53.5 Examples

Op	**a**	**b**	**c**	***_lo** result	***_hi** result
spi_vselectd32	0x00000000	0x11223344	0x55667788	0x55667788	0x11223344
spi_vselectd32	0xFFFFFFFF	0x11223344	0x55667788	0x11223344	0x55667788
spi_vselecdt32	0x0000FFFF	0x11223344	0x55667788	0x55663344	0x11227788
spi_vselectd32	0x00FF00FF	0x11223344	0x55667788	0x55227744	0x11663388
spi_vselectd32	0x00010203	0x11223344	0x55667788	0x55227744	0x11663388
spi_vselectd16	0x00010203	0x11223344	0x55667788	0x55227744	0x11663388
spi_vselectd8	0x00010203	0x11223344	0x55667788	0x55227744	0x11663388

7.2.54 spi_vshift

7.2.54.1 Prototypes

vec uint32x1 spi_vshift32(vec uint32x1 a, vec int32x1 b);

vec uint16x2 spi_vshift16(vec uint16x2 a, vec int32x1 b);

vec uint8x4 spi_vshift8(vec uint8x4 a, vec int32x1 b);

7.2.54.2 Description

Each **spi_vshift*** (vector logical shift) intrinsic returns the value of **a** shifted by **b** bits. The arguments may be vectors of words, packed halfwords, or packed bytes.

The same **int32x1** shift count **b** applies to each subword component for the packed variants; **spi_vshift** does not support separate shift counts for subword components.

7.2.54.3 Return value

Returns the value of **a** shifted by **b** bits. If the shift count **b** is positive, it left shifts by **b** bits. If the shift count **b** is negative, it right shifts by **-b** bits. If **b** is zero, it simply returns **a**. If the magnitude of **b** is greater than or equal to the width of the type, **spi_shift** returns 0. Logical right shift zero-fills the high-order bit during shifting.

7.2.54.4 See also

spi_shift: like **spi_vshift**, but for scalar arguments rather than vectors.
spi_vshifta: like **spi_vshift**, but uses arithmetic shift rather than logical shift.

7.2.54.5 Examples

Op	**a**	**b**	Result
spi_vshift32	0x4567ABCD	0x00000001	0x8ACF579A
spi_vshift32	0x4567ABCD	0xFFFFFFFF	0x22B3D5E6
spi_vshift32	0x4567ABCD	0x00000020	0x00000000
spi_vshift32	0x4567ABCD	0xFFFFFFDF	0x00000000
spi_vshift32	0xFFFFFFF0	0xFFFFFFFF	0x7FFFFFF8
spi_vshift16	0x4567ABCD	0x00000001	0x8ACE579A
spi_vshift16	0x4567ABCD	0xFFFFFFFF	0x22B355E6
spi_vshift8	0x4567ABCD	0x00000001	0x8ACE569A
spi_vshift8	0x4567ABCD	0xFFFFFFFF	0x22335566

7.2.55 spi_vshifta

7.2.55.1 Prototypes

vec int32x1 spi_vshifta32(vec int32x1 a, vec int32x1 b);

vec int16x2 spi_vshifta16(vec int16x2 a, vec int32x1 b);

7.2.55.2 Description

Each **spi_vshifta*** (vector arithmetic shift) intrinsic returns the value of **a** shifted by **b** bits. The arguments may be vectors of words or packed halfwords. The DPU does not support arithmetic shift for packed bytes.

The same **int32x1** shift count **b** applies to each subword component for the packed **int16x2** variant; **spi_vshifta** does not support separate shift counts for subword components.

7.2.55.3 Return value

Returns the value of **a** shifted by **b** bits. If the shift count **b** is positive, it left shifts by **b** bits. If the shift count **b** is negative, it right shifts by **-b** bits. If **b** is zero, it simply returns **a**. Arithmetic right shift sign-fills the high-order bit during shifting.

If the shifted result overflows the range of the type, the result saturates. For the **int32x1** case (**int16x2** is similar), left shift counts 32 and above return saturated result 0x7FFFFFFF or 0x80000000, and right shift counts -32 and below return 0 for positive inputs and -1 (0xFFFFFFFF) for negative inputs.

7.2.55.4 See also

spi_shifta: like **spi_vshifta**, but for scalar arguments rather than vectors.
spi_vshift: like **spi_vshifta**, but uses logical shift rather than arithmetic shift.

7.2.55.5 Examples

Op	**a**	**b**	Result
spi_vshifta32	0x4567ABCD	0x00000001	0x7FFFFFFF
spi_vshifta32	0x4567ABCD	0xFFFFFFFF	0x22B3D5E6
spi_vshifta32	0x4567ABCD	0x00000020	0x7FFFFFFF
spi_vshifta32	0x4567ABCD	0xFFFFFFDF	0x00000000
spi_vshifta32	0xFFFFFFF0	0xFFFFFFFF	0xFFFFFFF8
spi_vshifta16	0x4567ABCD	0x00000001	0x7FFF8000
spi_vshifta16	0x4567ABCD	0xFFFFFFFF	0x22B3D5E6

7.2.56 spi_vshuffle

7.2.56.1 Prototypes

vec int32x1 spi_vshufflei(vec uint32x1 a, vec int32x1 b, vec int32x1 c);

vec uint32x1 spi_vshuffleu(vec uint32x1 a, vec uint32x1 b, vec uint32x1 c);

7.2.56.2 Description

Each **spi_vshuffle*** (vector shuffle bytes) intrinsic uses the control information in **a** to perform byte reordering on **b** and **c**.

7.2.56.3 Return value

The value of each byte $\mathbf{a}_n$ of the control value **a** determines the corresponding byte $\mathbf{x}_n$ of the result **x,** as follows:

0, 1, 2, 3 — byte 0, 1, 2 or 3 of **b**
4, 5, 6, 7 — byte 0, 1, 2 or 3 of **c**
8, 9, 0xA, 0xB
spi_vshufflei : 0 or 0xFF depending on the sign bit (bit 7, 0 or 1) of byte 0, 1, 2 or 3 of **b**
spi_vshuffleu: 0
0xC, 0xD, 0xE, 0xF
spi_vshufflei: 0 or 0xFF depending on the sign bit (bit 7, 0 or 1) of byte 0, 1, 2 or 3 of **c**
spi_vshuffleu: 0

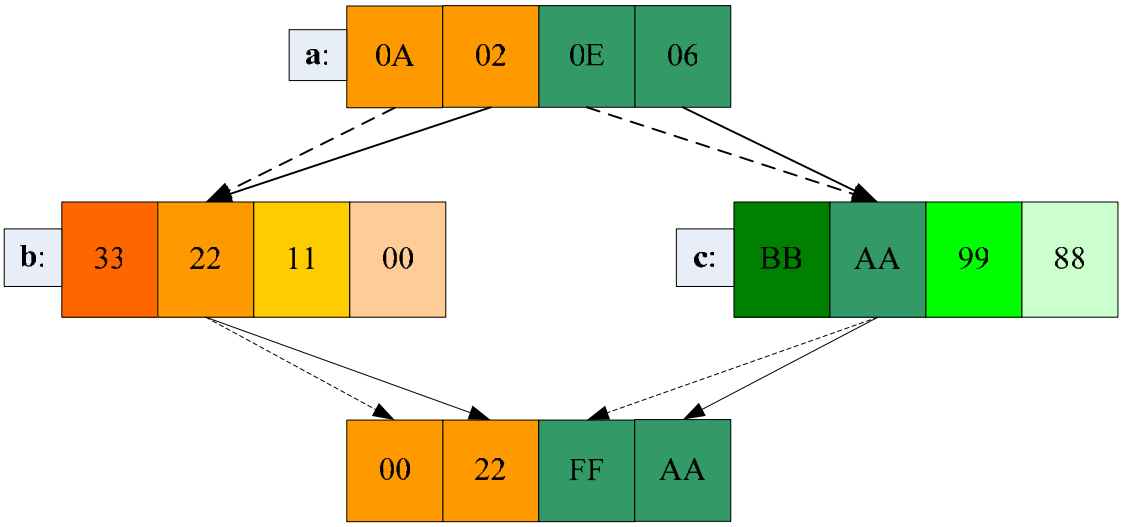
7.2.56.4 See also

spi_shuffle: like **spi_vshuffle**, but returns a scalar result rather than a vector.
spi_vclip: can be used to pack arguments, but with clipping.
spi_vperm: like **spi_vshuffle**, but permutes data between lanes rather than shuffling bytes within a lane.
spi_vshuffled: like **spi_vshuffle**, but interprets control value as nibbles rather than bytes.

7.2.56.5 Examples

Op	**a**	**b**	**c**	Result	Comment
spi_vshufflei	0x00010203	0x33221100	0xBBAA9988	0x00112233	endianness byte swap
spi_vshufflei	0x01000302	0x33221100	0xBBAA9988	0x11003322	halfword component swap
spi_vshufflei	0x05010602	0x33221100	0xBBAA9988	0x9911AA22	byte shuffle
spi_vshufflei	0x08020E06	0x33221100	0xBBAA9988	0x0022FFAA	sign-extend bytes to halfwords
spi_vshuffleu	0x08020E06	0x33221100	0xBBAA9988	0x002200AA	zero-extend bytes to halfwords
spi_vshufflei	0x0E0E0E06	0x33221100	0xBBAA9988	0xFFFFFFAA	sign-extend byte to word
spi_vshuffleu	0x0E0E0E06	0x33221100	0xBBAA9988	0x000000AA	zero-extend byte to word
spi_vshufflei	0x0E0E0504	0x33221100	0xBBAA9988	0xFFFF9988	sign-extend halfword to word
spi_vshuffleu	0x0E0E0504	0x33221100	0xBBAA9988	0x00009988	zero-extend halfword to word

The following diagram illustrates the use of **spi_vshufflei** to sign-extend selected bytes to halfwords using control word 0x0A020E06. Color here indicates the byte that each byte of the control word selects, with the dotted arrows indicating sign-extension.



7.2.56.6 Prototypes

vec int32x1 spi_vshuffledi_hi(vec uint32x1 a, vec int32x1 b, vec int32x1 c);

vec int32x1 spi_vshuffledi_lo(vec uint32x1 a, vec int32x1 b, vec int32x1 c);

vec uint32x1 spi_vshuffledu_hi(vec uint32x1 a, vec uint32x1 b, vec uint32x1 c);

vec uint32x1 spi_vshuffledu_lo(vec uint32x1 a, vec uint32x1 b, vec uint32x1 c);

7.2.56.7 Description

Each **spi_vshuffled*** (vector shuffle bytes dual) intrinsic performs byte reordering on **b** and **c** using control information in **a**.

A single hardware operation performs both **spi_vshuffled*_hi** and the corresponding **spi_vshuffled*_lo**, returning two results. The compiler **spc** merges adjacent paired calls to these functions for efficiency.

7.2.56.8 Return value

Each four-bit nibble of the control value **a** determines a byte of the result. For **spi_vshuffled*_hi**, the high-order nibble of $\mathbf{a}_n$ determines byte *n* of the result, while for **spi_vshuffled*_lo** the low-order nibble of $\mathbf{a}_n$ determines byte *n* of the result. The control values are:

0, 1, 2, 3 — byte 0, 1, 2 or 3 of **b**
4, 5, 6, 7 — byte 0, 1, 2 or 3 of **c**
8, 9, 0xA, 0xB — **spi_vshuffledi***: 0 or 0xFF depending on the sign bit (bit 7, 0 or 1) of byte 0, 1, 2 or 3 of **b**
spi_vshuffledu*: 0

0xC, 0xD, 0xE, 0xF
spi_vshuffledi*: 0 or 0xFF depending on the sign bit (bit 7, 0 or 1) of byte 0, 1, 2 or 3 of **c**
spi_vshuffledu*: 0

7.2.56.9 See also

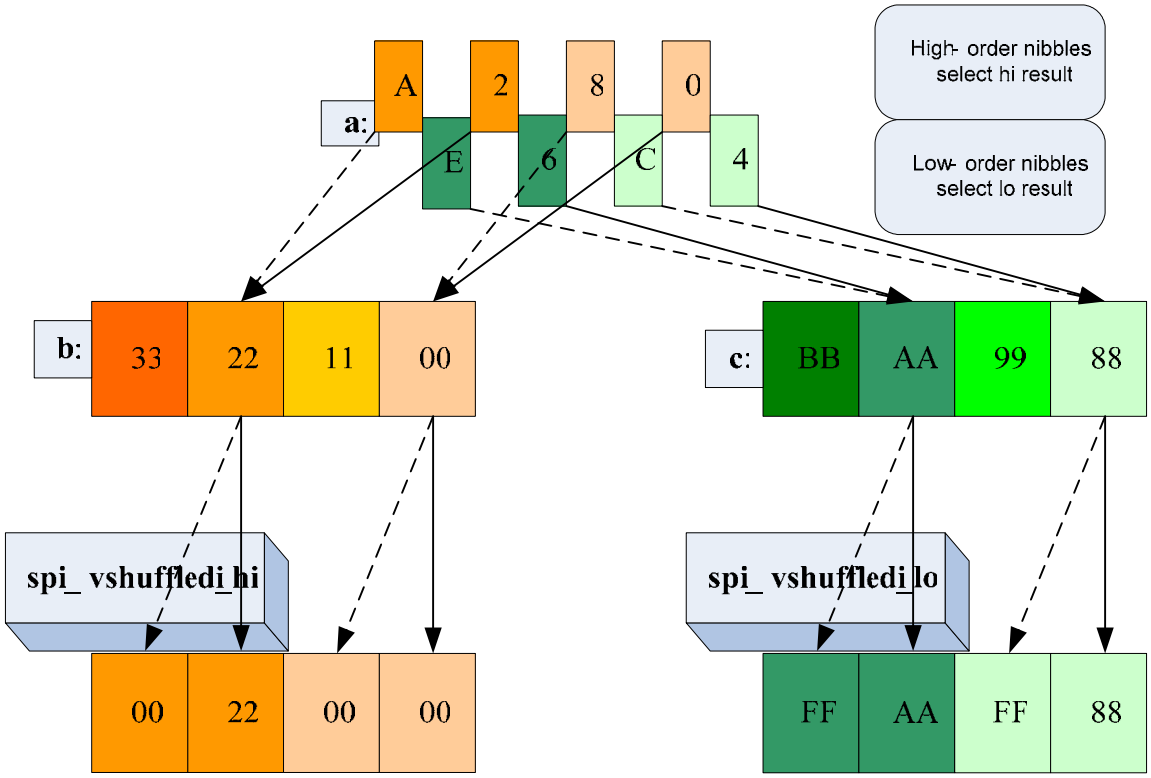
spi_shuffled: like **spi_vshuffled**, but returns a scalar result rather than a vector.
spi_vshuffle: like **spi_vshuffled**, but interprets control value as bytes rather than nibbles.

7.2.56.10 Examples

Op	**a**	**b**	**c**	**hi** result	**lo** result	Comment
spi_vshuffledi_*	0x04152637	0x33221100	0xBBAA9988	0x00112233	0x8899AABB	endianness byte swap
spi_vshuffledi_*	0x15043726	0x33221100	0xBBAA9988	0x11003322	0x9988BBAA	halfword component swap
spi_vshuffledi_*	0x17345602	0x33221100	0xBBAA9988	0x11339900	0xBB88AA22	byte shuffle
spi_vshuffledi_*	0x8C26EA62	0x33221100	0xBBAA9988	0x0022FFAA	0xFFAA0022	sign-extend bytes to halfwords
spi_vshuffledu_*	0x8C26EA62	0x33221100	0xBBAA9988	0x002200AA	0x00AA0022	zero-extend bytes to halfwords

Op	**a**	**b**	**c**	**hi** result	**lo** result	Comment
spi_vshuffledi_*	0xEEEEEE62	0x33221100	0xBBAA9988	0xFFFFFFAA	0xFFFFFF22	sign-extend byte to word
spi_vshuffledu_*	0xEEEEEE62	0x33221100	0xBBAA9988	0x000000AA	0x00000022	zero-extend byte to word
spi_vshuffledi_*	0xEEEE7564	0x33221100	0xBBAA9988	0xFFFFBBAA	0xFFFF9988	sign-extend halfword to word
spi_vshuffledu_*	0xEEEE7564	0x33221100	0xBBAA9988	0x0000BBAA	0x00009988	zero-extend halfword to word

The following diagram shows the use of **spi_vshuffledi_*** with control argument 0xAE268C04 to sign-extend the low-order bytes of each halfword of **b** and **c** to halfwords. Color here indicates the byte that each nibble of the control word selects, while dotted arrows indicate sign-extension.



7.2.57 spi_vsub

7.2.57.1 Prototype

vec int32x1 spi_vsub32i(vec int32x1 a, vec int32x1 b);

vec uint32x1 spi_vsub32u(vec uint32x1 a, vec uint32x1 b);

vec int16x2 spi_vsub16i(vec int16x2 a, vec int16x2 b);

vec uint16x2 spi_vsub16u(vec uint16x2 a, vec uint16x2 b);

vec int8x4 spi_vsub8i(vec int8x4 a, vec int8x4 b);

vec uint8x4 spi_vsub8u(vec uint8x4 a, vec uint8x4 b);

7.2.57.2 Description

Each **spi_vsub*** (vector subtraction) intrinsic returns the difference of its arguments. The arguments may be signed or unsigned words, packed halfwords, or packed bytes.

7.2.57.3 Return value

Returns **a** - **b**.

7.2.57.4 See also

spi_sub: like **spi_vsub**, but for scalar arguments rather than vectors.
spi_vadd: like **spi_vsub**, but using addition rather than subtraction.
spi_vsubc: like **spi_vsub**, but with carry.
spi_vsubs: like **spi_vsub**, but using saturation arithmetic.

7.2.57.5 Examples

Op	**a**	**b**	Result
spi_vsub32i	0x00000010	0x00000003	0x0000000D
spi_vsub32u	0x00000010	0x00000003	0x0000000D
spi_vsub16i	0x00100020	0x0001FFFF	0x000F0021
spi_vsub16u	0x00100020	0x0001FFFF	0x000F0021
spi_vsub8i	0x10203040	0x0102FFFE	0x0F1E3142
spi_vsub8u	0x10203040	0x0102FFFE	0x0F1E3142

7.2.58 spi_vsubc

7.2.58.1 Prototype

vec int32x1 spi_vsubc32(vec int32x1 a, vec int32x1 b, vec int32x1 c);

vec int32x1 spi_vsubc32_c(vec int32x1 a, vec int32x1 b, vec int32x1 c);

7.2.58.2 Description

spi_vsubc32 (vector subtraction with carry) returns the difference of its first two arguments minus one plus the low-order bit of its third argument, while **spi_vsubc32_c** performs the same computation and returns the resulting carry.

A single hardware operation performs both **spi_vsubc32** and **spi_vsubc32_c**, returning two results. The compiler **spc** merges adjacent paired calls to these functions for efficiency.

7.2.58.3 Return value

spi_vsubc32 returns the difference **a** - **b** - 1 plus the low-order bit of **c**, while **spi_vaddc32_c** returns the carry from the same computation.

7.2.58.4 See also

spi_vaddc: like **spi_vsubc**, but adds rather than subtracting.
spi_vsub; like **spi_vsubc**, but without carry.

7.2.58.5 Examples

Op	**a**	**b**	**c**	**spi_vsubc32** result	**spi_vsubc32_c** result
spi_vsubc32*	0x00000007	0x00000003	0x00000000	0x00000003	0x00000001
spi_vsubc32*	0x00000001	0x00000003	0x00000000	0xFFFFFFFD	0x00000000
spi_vsubc32*	0x00000001	0x00000003	0x00000001	0xFFFFFFFE	0x00000000

7.2.59 spi_vsubs

7.2.59.1 Prototypes

vec int32x1 spi_vsubs32i(vec int32x1 a, vec int32x1 b);

vec int16x2 spi_vsubs16i(vec int16x2 a, vec int16x2 b);

vec uint16x2 spi_vsubs16u(vec uint16x2 a, vec uint16x2 b);

vec uint16x2 spi_vsubs16ui(vec uint16x2 a, vec int16x2 b);

vec uint8x4 spi_vsubs8u(vec uint8x4 a, vec uint8x4 b);

vec uint8x4 spi_vsubs8ui(vec uint8x4 a, vec int8x4 b);

7.2.59.2 Description

Each **spi_vsubs*** (vector subtraction with saturation) intrinsic returns the difference of its arguments using saturation arithmetic. In addition to the signed and unsigned flavors, mixed variants allow saturating addition of unsigned plus signed, producing an unsigned saturated result.

7.2.59.3 Return value

Returns **a - b** using saturation arithmetic; if the sum underflows or overflows the range of the result type, it returns the minimum or maximum representable value.

7.2.59.4 See also

spi_vadds: like **spi_vsubs**, but adds rather than subtracting.
spi_vsub: like **spi_vsubs**, but not using saturation arithmetic.

7.2.59.5 Examples

Op	**a**	**b**	Result
spi_vsubs32i	0x7FFFFFFF	0xFFFFFFF	0x7FFFFFFF
spi_vsubs32i	0x80000000	0x00000001	0x80000000
spi_vsubs16i	0x00047FFF	0x0003FFFF	0x00017FFF
spi_vsubs16u	0x00047FFF	0x0003FFFF	0x00010000
spi_vsubs16ui	0x00047FFF	0x0003FFFF	0x00080000
spi_vsubs8u	0x01027FFF	0x7F7FFFFF	0x00000000
spi_vsubs8ui	0x00010203	0x00010203	0x000080FF

7.2.60 spi_vsuma

7.2.60.1 Prototypes

vec int32x1 spi_vsuma16i(vec int16x2 a, vec int16x2 b, vec int32x1 c);

vec uint32x1 spi_vsuma16u(vec uint16x2 a, vec uint16x2 b, vec uint32x1 c);

vec uint16x2 spi_vsuma8u(vec uint8x4 a, vec uint8x4 b, vec uint16x2 c);

7.2.60.2 Description

Each **spi_vsuma*** (vector sum accumulating) intrinsic returns **c** plus the sum of components of **a** and **b**.

7.2.60.3 Return value

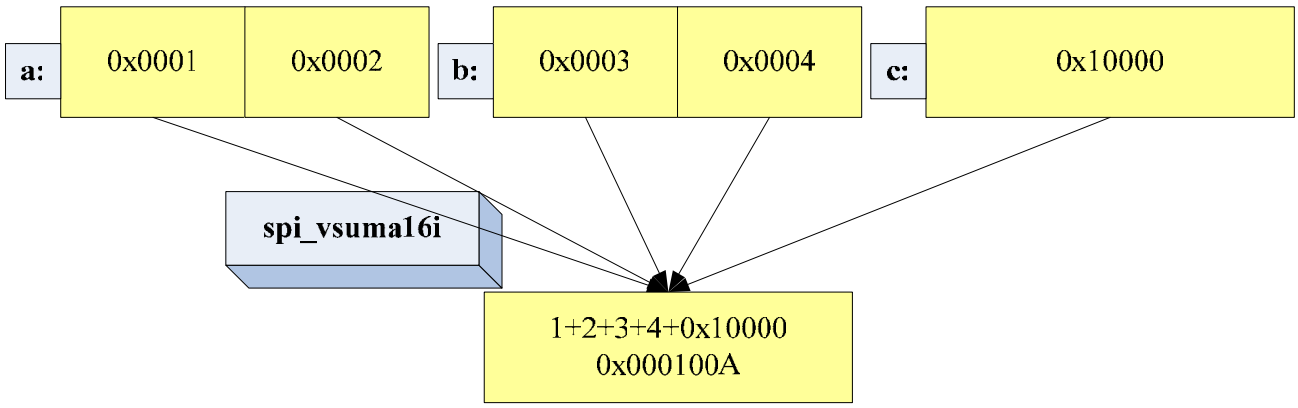
spi_vsuma16i and **spi_vsuma16u** return the 32-bit sum of **c** plus the sum of four 16-bit components of **a** and **b**.

spi_sumv8u returns a packed word containing two 16-bit sums: the high-order halfword contains the sum of high-order halfword of **c** plus the sum of the four 8-bit components from the high-order halfwords of **a** and **b,** while the low-order result halfword contains the low-order halfword of **c** plus the sum of the four 8-bit components from the low-order halfwords of **a** and **b**.

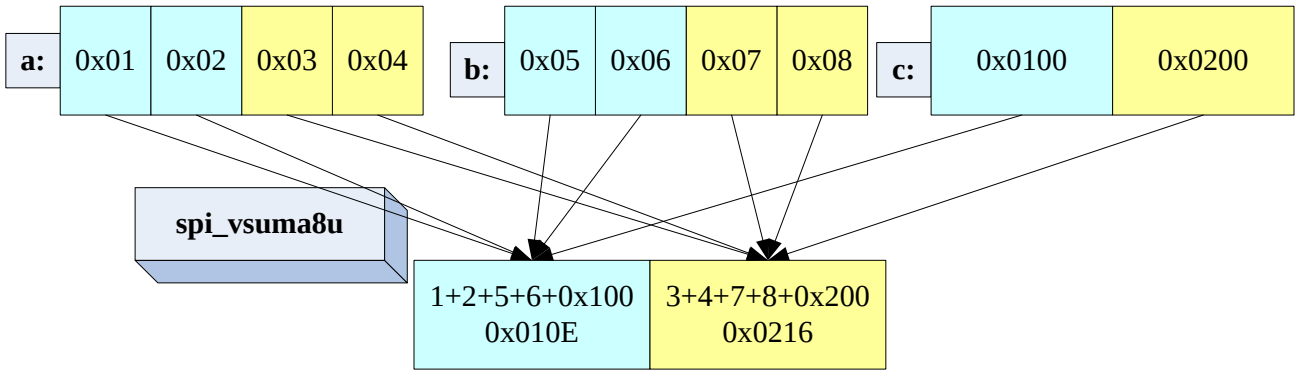
7.2.60.4 Examples

Op	**a**	**b**	**c**	Result
spi_vsuma16i	0x00010002	0x00030004	0x00010000	0x0001000A
spi_vsuma16u	0x00010002	0x00030004	0x00010000	0x0001000A
spi_vsuma8u	0x01020304	0x05060708	0x01000200	0x010E0216

The following diagram shows the operation of **spi_vsuma16i**.



The following diagram shows the operation of **spi_vsuma8u**.



7.2.61 spi_vxor

7.2.61.1 Prototypes

vec uint32x1 spi_vxor32(vec uint32x1 a, vec uint32x1 b);

vec uint16x2 spi_vxor16(vec uint16x2 a, vec uint16x2 b);

vec uint8x4 spi_vxor8(vec uint8x4 a, vec uint8x4 b);

7.2.61.2 Description

Each **spi_vxor*** (vector XOR) intrinsic returns the bitwise exclusive OR (XOR) of its arguments. The arguments may be vectors of words, packed halfwords, or packed bytes. The argument types must be identical; the result type is the same as the argument type.

spi_vxor32, **spi_vxor16** and **spi_vxor8** all represent the same DPU hardware operation. They have different names to allow strict type checking of arguments and result.

7.2.61.3 Return value

Each return value bit is 1 if the corresponding bit in exactly one argument is 1, otherwise it is 0.

7.2.61.4 See also

spi_vand: like **spi_vxor**, but uses bitwise AND rather than bitwise XOR.
spi_vnot: like **spi_vxor**, but uses bitwise NOT rather than bitwise XOR.
spi_vor: like **spi_vxor**, but uses bitwise OR rather than bitwise XOR.
spi_xor: like **spi_vxor**, but for scalar arguments rather than vectors.

7.2.61.5 Examples

Op	**a**	**b**	Result
spi_vxor32	0x12345678	0x00FF00FF	0x12CB5687
spi_vxor16	0x12345678	0x00FF00FF	0x12CB5687
spi_vxor8	0x12345678	0x00FF00FF	0x12CB5687

7.2.62 spi_xor

7.2.62.1 Prototypes

uint32x1 spi_xor32(uint32x1 a, uint32x1 b);

uint16x2 spi_xor16(uint16x2 a, uint16x2 b);

uint8x4 spi_xor8(uint8x4 a, uint8x4 b);

7.2.62.2 Description

Each **spi_xor*** (XOR) intrinsic returns the bitwise exclusive OR (XOR) of its arguments. The arguments may be words, packed halfwords, or packed bytes. The argument types must be identical; the result type is the same as the argument type.

spi_xor32, **spi_xor16** and **spi_xor8** all represent the same DPU hardware operation. They have different names to allow strict type checking of arguments and result.

7.2.62.3 Return value

Each return value bit is 1 if the corresponding bit in exactly one argument is 1, otherwise it is 0.

7.2.62.4 See also

spi_and: like **spi_xor**, but uses bitwise AND rather than bitwise XOR.
spi_not: like **spi_xor**, but uses bitwise NOT rather than bitwise XOR.
spi_or: like **spi_xor**, but uses bitwise OR rather than bitwise XOR.
spi_vxor: like **spi_xor**, but for vector arguments rather than scalars.

7.2.62.5 Examples

Op	**a**	**b**	Result
spi_xor32	0x12345678	0x00FF00FF	0x12CB5687
spi_xor16	0x12345678	0x00FF00FF	0x12CB5687
spi_xor8	0x12345678	0x00FF00FF	0x12CB5687

8 DSP MIPS standard library functions

This chapter describes the standard library functions available on DSP MIPS. Standard library functions are similar to Standard C functions; see *American National Standard for Programming Languages – C*, ANSI/ISO 9899-1990, for additional details. In some cases, DSP MIPS functions communicate with System MIPS to perform standard functions; for example, the implementation of DSP MIPS function **fopen** uses the standard C function **fopen** on System MIPS to open a file on the System MIPS Linux filesystem. As in Standard C, **FILE *** arguments to standard functions are called i/o streams; the user should not confuse this usage with the Stream type modifier **stream**.

The DSP MIPS runtime environment implements a subset of the Single Unix Specification 3 (SUS), System Interfaces volume (IEEE Std 1003.1, 2004 Edition), available online at www.opengroup.org/onlinepubs/009695399/. The runtime environment is *not* SUS conformant. Rather, it implements a useful subset of SUS but omits SUS features that are not relevant to the stream processor implementation.

For example, DSP MIPS essentially runs as a single process with multiple threads, with all threads sharing the same address space. Therefore, the runtime environment does not support process-related API calls (e.g., **fork** and **exec**). Similarly, it does not support asynchronous signals (no **kill**, **sigaction**, etc.).

The runtime environment fully supports some standard headers, including **float.h**, **iso646.h**, **limits.h**, **stdarg.h**, **stdbool.h**, **stddef.h**, **syslimits.h** and **varargs.h**. The subsections below specify the implemented subset of SUS functionality for other headers. Except as noted, the semantics of all implemented functions and macros follow SUS.

8.1.1 <assert.h>

8.1.1.1 Function

assert(int)

8.1.2 <ctype.h>

8.1.2.1 Notes

<cytpe.h> supports only the POSIX locale.

8.1.2.2 Functions

```
int     isalnum(int);
int     isalpha(int);
int     isascii(int);
int     isblank(int);
int     iscntrl(int);
int     isdigit(int);
int     isgraph(int);
int     islower(int);
int     isprint(int);
int     ispunct(int);
int     isspace(int);
int     isupper(int);
int     isxdigit(int);
int     tolower(int);
int     toupper(int);
```


8.1.3 <errno.h>

8.1.3.1 Notes

errno is implemented as a thread-specific variable.

8.1.3.2 Variable

errno

8.1.3.3 Macros

E2BIG
EACCES
EADDRINUSE
EADDRNOTAVAIL
EAGAIN
EALREADY
EBADF
EBADMSG
EBUSY
ECANCELED
ECONNABORTED
ECONNREFUSED
EDEADLK
EDESTADDRREQ
EDOM
EEXIST
EFAULT
EFBIG
EILSEQ
EINPROGRESS
EINTR
EINVAL
EIO
EISCONN
EMFILE
EMSGSIZE
ENAMETOOLONG
ENFILE
ENOBUFS
ENODEV
ENOENT
ENOMEM
ENOSPC
ENOSYS
ENOTCONN
ENOTTY
ENXIO
EOPNOTSUPP
EOVERFLOW
EPIPE
ERANGE
ESHUTDOWN
ESPIPE
ESRCH
ETIMEDOUT
EUSERS
EWOULDBLOCK

8.1.4 <math.h>

8.1.4.1 Notes

signgam is implemented as a thread-specific variable.

8.1.4.2 Variable

signgam

8.1.4.3 Macros

FP_ILOGB0	**HUGE_VALL**	**M_LOG2E**
FP_ILOGBNAN	**INFINITY**	**M_PI**
FP_INFINITE	**M_1_PI**	**M_PI_2**
FP_NAN	**M_2_PI**	**M_PI_4**
FP_NORMAL	**M_2_SQRTPI**	**M_SQRT1_2**
FP_SUBNORMAL	**M_E**	**M_SQRT2**
FP_ZERO	**M_LN10**	**NAN**
HUGE_VAL	**M_LN2**	
HUGE_VALF	**M_LOG10E**	

8.1.4.4 Functions

<math.h> implements only **double** versions of standard math functions, not **float** variants.

```
double  acos(double);
double  acosh(double);
double  asin(double);
double  asinh(double);
double  atan(double);
double  atan2(double, double);
double  atanh(double);
double  cbrt(double);
double  ceil(double);
double  copysign(double, double);
double  cos(double);
double  cosh(double);
double  erf(double);
double  erfc(double);
double  exp(double);
double  exp2(double);
double  expm1(double);
double  fabs(double);
double  fdim(double, double);
double  floor(double);
double  fma(double, double, double);
```



```
double  fmax(double, double);
double  fmin(double, double);
double  fmod(double, double);
int     fpclassify(double);
double  frexp(double, int*);
double  hypot(double, double);
int     ilogb(double);
int     isfinite(double);
int     isinf(double);
int     isnan(double);
int     isnormal(double);
double  j0(double);
double  j1(double);
double  jn(int, double);
double  ldexp(double, int);
double  lgamma(double x);
double  lgamma_r(double, int*);
double  log(double);
double  log10(double);
double  log1p(double);
double  log2(double);
double  logb(double);
long    lrint(double);
long    lround(double);
double  modf(double, double*);
double  nan(const char*);
double  nearbyint(double);
double  nextafter(double, double);
double  pow(double, double);
double  remainder(double, double);
double  remquo(double, double, int*);
double  rint(double);
double  round(double);
double  scalb(double, double);
double  scalbn(double, int);
int     signbit(double);
double  significand(double);
double  sin(double);
double  sinh(double);
double  sqrt(double);
double  sin(double);
double  tan(double);
double  tanh(double);
double  tgamma(double);
double  trunc(double);
double  y0(double);
double  y1(double);
double  yn(int, double);
```

8.1.5 <poll.h>

8.1.5.1 Notes

Internally, the DSP MIPS runtime environment only uses **POLLIN**, **POLLRDNORM**, **POLLOUT**, **POLLWRNORM**, **POLLPRI**, and **POLLNVAL**.

8.1.5.2 Type

nfds_t

8.1.5.3 Structure

struct pollfd;

8.1.5.4 Macros

POLLERR
POLLHUP
POLLIN
POLLNVAL
POLLOUT
POLLPRI
POLLRDBAND
POLLRDNORM
POLLWRBAND
POLLWRNORM

8.1.6 <pthread.h>

8.1.6.1 Notes

The current pthreads implementation only supports fixed priority-based scheduling (**SCHED_FIFO**). It does not support timeslicing. A thread will run without releasing the CPU until it blocks on I/O, sleeps, or a higher priority thread becomes runnable. At present, the use of different thread priorities is discouraged and should be avoided, as internal I/O mutexes do not implement any protection from unbounded priority inversion.

The current minimum stack size is 4KB, and the current maximum is 1MB. If no stacksize attribute is specified, a thread gets a default stack of 8KB.

A Stream programming model component should not use pthreads. An application can use pthreads, but only one thread in the application should call **spi_spm_start**, and only that thread may use the SPM APIs.

8.1.6.2 Types

pthread_t
pthread_attr_t

8.1.6.3 Macros

PTHREAD_CREATE_DETACHED
PTHREAD_CREATE_JOINABLE

8.1.6.4 Functions

```
int          pthread_attr_init(pthread_attr_t*);
int          pthread_attr_destroy(pthread_attr_t*);
int          pthread_attr_setstacksize(pthread_attr_t*, size_t);
int          pthread_attr_getstacksize(const pthread_attr_t*, size_t*);
int          pthread_attr_setdetachstate(pthread_attr_t*, int);
int          pthread_attr_getdetachstate(const pthread_attr_t*, int*);
int          pthread_create(pthread_t*, const pthread_attr_t*, void* (*)(void*), void*);
void         pthread_exit(void*);
int          pthread_detach(pthread_t);
int          pthread_join(pthread_t, void**);
pthread_t    pthread_self(void);
int          pthread_equal(pthread_t, pthread_t);
int          pthread_getschedparam(pthread_t, int*, struct sched_param*);
int          pthread_setschedparam(pthread_t, int, const struct sched_param*);
int          pthread_setschedprio(pthread_t, int);
```


8.1.7 <sched.h>

8.1.7.1 Macro

SCHED_FIFO

8.1.7.2 Structure

struct sched_param;

8.1.7.3 Functions

int sched_get_priority_max(int);
int sched_get_priority_min(int);
int sched_yield(void);

8.1.8 <stdint.h>

8.1.8.1 Types

int8_t
int16_t
int32_t
int64_t
uint8_t
uint16_t
uint32_t
uint64_t

8.1.9 <stdio.h>

8.1.9.1 Notes

The current **<stdio.h>** implementation is unbuffered, so all buffer-related calls (e.g., **setvbuf**, **ungetc**) are unimplemented.

DSP MIPS file i/o is redirected to System MIPS. The file namespace is identical to that of System MIPS; for example, `fopen("/etc/passwd", "r")` opens the same file as if it were called from a Linux user application running on System MIPS.

Output to console devices never blocks for an unbounded amount of time. This is obvious for UART output, because no flow control is used, so any output is sent out eventually. **stdout**/**stderr** output redirected to System MIPS uses a fixed-sized FIFO that drops output when it overflows. This can happen if System MIPS does not read from device **/dev/dsp_mips_console** fast enough. If it drops output, the runtime inserts a special marker "**** OVERFLOW ****" into the stream to show that output was dropped. The UART console never drops data, but it is much slower than a redirected console.

8.1.9.2 Macros

EOF	
FILENAME_MAX	// maximum file name size is 511 bytes (not including trailing '\0')
FOPEN_MAX	// maximum number of opened files is 32
SEEK_CUR	
SEEK_END	
SEEK_SET	
stderr	// default: System MIPS **stderr**
stdin	// default: System MIPS **stdin**
stdout	// default: System MIPS **stdout**

8.1.9.3 Type

FILE

8.1.9.4 Functions

```
void     clearerr(FILE*);
int      fclose(FILE*);
int      feof(FILE*);
int      ferror(FILE*);
int      fflush(FILE*);
int      fgetc(FILE*);
char *   fgets(char *, int, FILE *);
void     flockfile(FILE*);
FILE *   fopen(const char*, const char*);
int      fprintf(FILE*, const char*, ...);
int      fputc(int, FILE*);
```



```
int     fputs(const char*, FILE*);
size_t  fread(void*, size_t, size_t, FILE*);
int     fseek(FILE*, long, int);
int     fseeko(FILE*, off_t, int);
long    ftell(FILE*);
off_t   ftello(FILE*)
void    funlockfile(FILE*);
size_t  fwrite(const void*, size_t, size_t, FILE*);
int     getc(FILE*);
int     getchar(void);
void    perror(const char*);
int     printf(const char*, ...);
int     putc(int, FILE*);
int     putchar(int);
int     puts(const char*);
void    rewind(FILE*);
int     scanf(const char *);
int     snprintf(char*, size_t, const char*, ...);
int     sprintf(char*, const char*, ...);
int     sscanf(const char *, const char *, FILE *);
int     vfprintf(FILE*, const char*, va_list);
int     vsnprintf(char*, size_t, const char*, va_list);
int     vsprintf(char*, const char*, va_list);
```

8.1.10 <stdlib.h>

8.1.10.1 Macro

EXIT_FAILURE
EXIT_SUCCESS
RAND_MAX

8.1.10.2 Functions

void	**_Exit(int);**
void	**abort(void);**
int	**abs(int);**
int	**atexit(void (*func)(void));**
int	**atoi(const char*);**
long	**atol(const char*);**
long long	**atoll(const char*);**
void *	**calloc(size_t, size_t);**
void	**exit(int);**
void	**free(void*);**
char *	**getenv(const char*);**
long	**labs(long);**
long long	**llabs(long long) ;**
void *	**malloc(size_t);**
int	**posix_memalign(void**, size_t, size_t);**
int	**rand(void);**
void *	**realloc(void*, size_t);**
void	**srand(unsigned);**
long	**strtol(const char*, char**, int);**
long long	**strtoll(const char*, char**, int);**
unsigned long	**strtoul(const char*, char**, int);**
unsigned long long	**strtoull(const char*, char**, int);**

8.1.11 <string.h>

8.1.11.1 Functions

```
void *    memchr(const void*, int, size_t);
int       memcmp(const void*, const void*, size_t);
void *    memcpy(void*, const void*, size_t);
void *    memmove(void*, const void*, size_t);
void *    memset(void*, int, size_t);
char *    strcat(char*, const char*);
char *    strchr(const char*, int);
int       strcmp(const char*, const char*);
char *    strcpy(char*, const char*);
char *    strdup(const char*);
char *    strerror(int);
size_t    strlen(const char*);
char *    strncat(char*, const char*, size_t);
int       strncmp(const char*, const char*, size_t);
char *    strncpy(char*, const char*, size_t);
char *    strndup(const char*, size_t);          // a GNU extension, not part of SUS
char *    strrchr(const char*, int);
char *    strstr(const char *, const char *);
```

8.1.12 <strings.h>

8.1.12.1 Functions

```
int     ffs(int);
int     strcasecmp(const char*, const char*);
int     strncasecmp(const char*, const char*, size_t);
```


8.1.13 <sys/time.h>

8.1.13.1 Structure

struct timeval;

8.1.14 <sys/types.h>

8.1.14.1 Types

clock_t
off_t
suseconds_t
time_t
u_int8_t
u_int16_t
u_int32_t
u_int64_t
useconds_t

8.1.15 <time.h>

8.1.15.1 Notes

The value of **CLOCKS_PER_SEC** is not 1,000,000 as specified in *SUS*, but instead is the frequency of the MIPS CP0 counter (which is usually the MIPS clock frequency divided by 2).

8.1.15.2 Structure

struct timespec;

8.1.15.3 Macro

CLOCKS_PER_SEC // see note above

8.1.15.4 Functions

clock_t clock(void); // returns contents of MIPS CP0 counter
int nanosleep(const struct timespec*, struct timespec*); // see note above

8.1.16 <unistd.h>

8.1.16.1 Notes

The timer interrupt frequency determines the sleep resolution. Currently, the runtime uses a 10 ms system tick clock, so any sleep function delays thread execution for at least 10 ms.

8.1.16.2 Variable

extern char ** environ;

8.1.16.3 Functions

void _exit(int);
unsigned int sleep(unsigned int);
int usleep(useconds_t);

8.1.16.4 Macros

_POSIX_THREAD_ATTR_STACKSIZE
_POSIX_THREAD_PRIORITY_SCHEDULING
_POSIX_THREADS
_POSIX_VERSION

9 Glossary

The table below gives brief definitions of some common terms and acronyms. The second column identifies terms that are SPI-specific. For additional details on industry-standard terminology, see e.g. Wikipedia's excellent explanations.

ADC		analog to digital converter
AHB		AMBA High-Performance Bus
ALSA		Advanced Linux Sound Architecture
ALU		arithmetic-logical unit
AMBA		Advanced Microcontroller Bus Architecture
API		application programming interface
application		the top-level software that implements a Stream program
AVC		Advanced Video Coding
basetype	SPI	the type of each data record in a stream
BOA		a high performance web server
CIF		common intermediate format; a 352x288 video format
CODEC		coder / decoder (or compressor / decompressor): program that manipulates stream data
component	SPI	a high-level data-driven computation module
CPB		coded picture buffer
CRAMFS		compressed ROM filesystem
D1		digital video format (PAL 720x576 MPEG-2, NTSC 720x480 MPEG-2)
DAC		digital to analog converter
DHCP		dynamic host configuration protocol
DLL		dynamically loadable library
DMA		direct memory access
DPU	SPI	data parallel unit: the part of a stream processor that executes kernels
DSP		digital signal processor or digital signal processing
DSP MIPS	SPI	one of two MIPS processors (System MIPS and DSP MIPS) on a stream processor
Eclipse		extensible open development platform
FIFO		first in / first out queue
GPU		general purpose processing unit: part of a stream processor that executes Stream code
GUI		graphical user interface
H.264		video compression standard (a.k.a. MPEG-4 Part 10, a.k.a. AVC)
HD		high definition video
HDK	SPI	hardware development kit
HDMI		high-definition multimedia interface: a digital audio/video interface
IC		integrated circuit
IDE		integrated development environment
I frame		intra frame (coded without reference to other frames)
in-lane	SPI	per lane; each lane can only access in-lane LRF data directly, not data from other lanes
I/O		input/output
IPC		interprocess communication
ISA		instruction set architecture
JFFS		journaling flash filesystem
JTAG		Joint Test Action Group: IEEE 1149.1 standard for debugging ICs and embedded systems
kernel	SPI	a DPU function to perform a computationally intensive operation on streams
lane	SPI	one of multiple identical arithmetic processors in a stream processor (8 on SP8, 16 on SP16)
LGPL		Gnu Lesser General Public License
Linux		operating system (the operating system on System MIPS)

LRF	SPI	lane register file: local storage for communicating stream data to/from a kernel
mb		macroblock
me		motion estimation
mii		minimum iteration interval (of software pipelined loop)
MIPS		microprocessor without interlocked pipeline stages: a computer microprocessor architecture
MIPSsim		a simulator for MIPS programs
MPEG		Moving Picture Experts Group
MTD		memory technology device; a Linux subsystem for memory technology devices (e.g., flash)
module	SPI	an executable or library built from a **spide** project
NFS		network file system
NTSC		National Television Standards Committee; the television format used in the US and Japan
ORF	SPI	operand register file: local storage for each lane in a stream processor
OS		operating system
PAL		Phase Alternating Line; a television format used in Europe
PC		personal computer; also, program counter
PCM		Pulse Code Modulation: an encoding for digital audio data
P frame		predictively coded frame (coded with reference to other frames)
pipeline	SPI	top-level Stream function that performs stream operations
PPS		picture parameter set
project	SPI	a group of related files in a **spide** workspace
PSNR		peak signal-to-noise ratio
QCIF		quarter common intermediate format; a 176x144 video format
QP		quantization parameter
RAM		random access memory
RC		rate control
record	SPI	a structured data item that forms an element of a stream
RGB		an additive color model (red + green + blue)
RPC		remote procedure call
RTL		runtime library
RTP		real time transport protocol
RTSP		real time streaming protocol
SD		standard definition video
SDE		software development environment
SIMD		single instruction, multiple data; a variety of VLIW architecture design
SOC		system-on-a-chip
SORF	SPI	scalar operand register file: for communicating non-stream shared data to/from a kernel
SP16	SPI	a 16-lane stream processor from Stream Processors, Inc.
SP8	SPI	an 8-lane stream processor from Stream Processors, Inc.
spc	SPI	Stream Processors compiler
SPI	SPI	Stream Processors, Inc.; also Serial Peripheral Interface
spide	SPI	Stream Processors integrated development environment
SPM	SPI	Stream programming model
SPS		sequence parameter set
SRAM		static random access memory
Storm-1	SPI	a family of SPI processors, including SP16 and SP8
stream	SPI	a sequence of data records, each of identical type
striped	SPI	distributed across lanes, as with stream records
SUS		Single Unix Specification
SWP		software pipelining: rearranging loop operations to minimize iteration interval
System MIPS	SPI	one of two MIPS processors (System MIPS and DSP MIPS) on a stream processor
TCP		transfer control protocol

UART		universal asynchronous receiver/transmitter
V4L2		Video for Linux Two: a specification for Linux video
VBV		video buffering verifier (a.k.a. CPB)
vector	SPI	a variable with a separate value in each lane
VLIW		very large instruction word architecture
width	SPI	size in bits of each component of a basic data type (32, 16 or 8)
workspace	SPI	a directory containing **spide** metadata
YUV		a color model with one luma and two chrominance components.
YUV422		a YUV data format

10 Index

For additional information or product support, please contact:
Stream Processors, Inc., 455 DeGuigne Drive, Sunnyvale, CA 94085-3890, USA
Telephone: +1.408.616.3338 • FAX: +1.408.616.3337 • Email: info@streamprocessors.com • Web: www.streamprocessors.com

This document contains advance information on SPI products, some of which are in development, sampling or initial production phases.
The information and specifications contained herein are preliminary and are subject to change at the discretion of Stream Processors, Inc.