libatomic Reference

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

8-Bit Data Functions

sceAtomicAdd8, sceAtomicIncrement8, sceAtomicDecrement8, sceAtomicOr8, sceAtomicAnd8, sceAtomicExchange8, sceAtomicCompareAndSwap8, sceAtomicLoad8, sceAtomicStore8

Atomic operation for 8-bit data

Definition

```
#include <sce_atomic.h>
int8_t sceAtomicAdd8(
        volatile int8_t* ptr,
        int8_t value
);
int8_t sceAtomicIncrement8(
        volatile int8_t* ptr
);
int8_t sceAtomicDecrement8(
        volatile int8_t* ptr
);
int8_t sceAtomicOr8(
        volatile int8_t* ptr,
        int8_t value
);
int8_t sceAtomicAnd8(
        volatile int8_t* ptr,
        int8_t value
);
int8_t sceAtomicExchange8(
        volatile int8_t* ptr,
        int8_t swap
);
int8_t sceAtomicCompareAndSwap8(
        volatile int8_t* ptr,
        int8_t compare,
        int8_t swap
);
int8_t sceAtomicLoad8(
        volatile int8_t* ptr
);
void sceAtomicStore8(
        volatile int8_t* ptr,
        int8_t value
);
```

Arguments

ptr	Memory to be updated
value	Operation argument
swap	Value to be written
compare	Value to be compared

Return Values

Refer to Description.

Description

`sceAtomicAdd8(ptr, value)` executes an operation equivalent to the following code as an indivisible operation.

```
int8_t old = *ptr;
*ptr = old + value;
return old;
```

`sceAtomicIncrement8(ptr)` executes an operation equivalent to the following code as an indivisible operation.

```
int8_t old = *ptr;
*ptr = old + 1;
return old;
```

`sceAtomicDecrement8(ptr)` executes an operation equivalent to the following code as an indivisible operation.

```
int8_t old = *ptr;
*ptr = old - 1;
return old;
```

`sceAtomicOr8(ptr, value)` executes an operation equivalent to the following code as an indivisible operation.

```
int8_t old = *ptr;
*ptr = old | value;
return old;
```

`sceAtomicAnd8(ptr, value)` executes an operation equivalent to the following code as an indivisible operation.

```
int8_t old = *ptr;
*ptr = old & value;
return old;
```

`sceAtomicExchange8(ptr, swap)` executes an operation equivalent to the following code as an indivisible operation.

```
int8_t old = *ptr;
*ptr = swap;
return old;
```

`sceAtomicCompareAndSwap8(ptr, compare, swap)` executes an operation equivalent to the following code as an indivisible operation.

```
int8_t old = *ptr;
if(old == compare){
   *ptr = swap;
}
return old;
```

`sceAtomicLoad8(ptr)` executes an operation equivalent to the following code as an indivisible operation.

```
return *ptr;
```

`sceAtomicStore8(ptr, value)` executes an operation equivalent to the following code as an indivisible operation.

```
*ptr = value;
```

Memory Ordering

The above functions do not perform memory ordering control. If memory ordering control is required, use the following functions.

```
int8_t sceAtomicAdd8Acquire(volatile int8_t* ptr,int8_t value);
int8_t sceAtomicAdd8Release(volatile int8_t* ptr,int8_t value);
int8_t sceAtomicAdd8AcqRel(volatile int8_t* ptr,int8_t value);
int8_t sceAtomicAdd8Relaxed(volatile int8_t* ptr,int8_t value);

int8_t sceAtomicIncrement8Acquire(volatile int8_t* ptr);
int8_t sceAtomicIncrement8Release(volatile int8_t* ptr);
int8_t sceAtomicIncrement8AcqRel(volatile int8_t* ptr);
int8_t sceAtomicIncrement8Relaxed(volatile int8_t* ptr);

int8_t sceAtomicDecrement8Acquire(volatile int8_t* ptr);
int8_t sceAtomicDecrement8Release(volatile int8_t* ptr);
int8_t sceAtomicDecrement8AcqRel(volatile int8_t* ptr);
int8_t sceAtomicDecrement8Relaxed(volatile int8_t* ptr);

int8_t sceAtomicOr8Acquire(volatile int8_t* ptr,int8_t value);
int8_t sceAtomicOr8Release(volatile int8_t* ptr,int8_t value);
int8_t sceAtomicOr8AcqRel(volatile int8_t* ptr,int8_t value);
int8_t sceAtomicOr8Relaxed(volatile int8_t* ptr,int8_t value);

int8_t sceAtomicAnd8Acquire(volatile int8_t* ptr,int8_t value);
int8_t sceAtomicAnd8Release(volatile int8_t* ptr,int8_t value);
int8_t sceAtomicAnd8AcqRel(volatile int8_t* ptr,int8_t value);
int8_t sceAtomicAnd8Relaxed(volatile int8_t* ptr,int8_t value);

int8_t sceAtomicExchange8Acquire(volatile int8_t* ptr,int8_t swap);
int8_t sceAtomicExchange8Release(volatile int8_t* ptr,int8_t swap);
int8_t sceAtomicExchange8AcqRel(volatile int8_t* ptr,int8_t swap);
int8_t sceAtomicExchange8Relaxed(volatile int8_t* ptr,int8_t swap);

int8_t sceAtomicCompareAndSwap8Acquire(volatile int8_t* ptr,int8_t
compare,int8_t swap);
int8_t sceAtomicCompareAndSwap8Release(volatile int8_t* ptr,int8_t
compare,int8_t swap);
int8_t sceAtomicCompareAndSwap8AcqRel(volatile int8_t* ptr,int8_t
compare,int8_t swap);
int8_t sceAtomicCompareAndSwap8Relaxed(volatile int8_t* ptr,int8_t
compare,int8_t swap);

int8_t sceAtomicLoad8Acquire(volatile int8_t* ptr);
int8_t sceAtomicLoad8Release(volatile int8_t* ptr);
int8_t sceAtomicLoad8AcqRel(volatile int8_t* ptr);
int8_t sceAtomicLoad8Relaxed(volatile int8_t* ptr);

void sceAtomicStore8Acquire(volatile int8_t* ptr,int8_t value);
void sceAtomicStore8Release(volatile int8_t* ptr,int8_t value);
void sceAtomicStore8AcqRel(volatile int8_t* ptr,int8_t value);
void sceAtomicStore8Relaxed(volatile int8_t* ptr,int8_t value);
```

Functions with `Acquire` at the end guarantee that the memory operations performed by these functions will always be executed before later memory operations.

Functions with `Release` at the end guarantee that the memory operations performed before these functions were called will always be executed before the memory operations of these functions in other threads.

Functions with `AcqRel` at the end guarantee that the memory operations performed before these functions were called will always be executed before the memory operations of these functions, and guarantees that the memory operations of these functions will always be executed before later memory operations.

Functions with `Relaxed` at the end do not guarantee the order of memory operations before or after and are the same as functions with nothing added at the end.

16-Bit Data Functions

sceAtomicAdd16, sceAtomicIncrement16, sceAtomicDecrement16, sceAtomicOr16, sceAtomicAnd16, sceAtomicExchange16, sceAtomicCompareAndSwap16, sceAtomicLoad16, sceAtomicStore16

Atomic operation for 16-bit data

Definition

```
#include <sce_atomic.h>
int16_t sceAtomicAdd16(
        volatile int16_t* ptr,
        int16_t value
);
int16_t sceAtomicIncrement16(
        volatile int16_t* ptr
);
int16_t sceAtomicDecrement16(
        volatile int16_t* ptr
);
int16_t sceAtomicOr16(
        volatile int16_t* ptr,
        int16_t value
);
int16_t sceAtomicAnd16(
        volatile int16_t* ptr,
        int16_t value
);
int16_t sceAtomicExchange16(
        volatile int16_t* ptr,
        int16_t swap
);
int16_t sceAtomicCompareAndSwap16(
        volatile int16_t* ptr,
        int16_t compare,
        int16_t swap
);
int16_t sceAtomicLoad16(
        volatile int16_t* ptr
);
void sceAtomicStore16(
        volatile int16_t* ptr,
        int16_t value
);
```

Arguments

ptr	Memory to be updated
value	Operation argument
swap	Value to be written
compare	Value to be compared

Return Values

Refer to Description.

Description

`sceAtomicAdd16(ptr, value)` executes an operation equivalent to the following code as an indivisible operation.

```
int16_t old = *ptr;
*ptr = old + value;
return old;
```

`sceAtomicIncrement16(ptr)` executes an operation equivalent to the following code as an indivisible operation.

```
int16_t old = *ptr;
*ptr = old + 1;
return old;
```

`sceAtomicDecrement16(ptr)` executes an operation equivalent to the following code as an indivisible operation.

```
int16_t old = *ptr;
*ptr = old - 1;
return old;
```

`sceAtomicOr16(ptr, value)` executes an operation equivalent to the following code as an indivisible operation.

```
int16_t old = *ptr;
*ptr = old | value;
return old;
```

`sceAtomicAnd16(ptr, value)` executes an operation equivalent to the following code as an indivisible operation.

```
int16_t old = *ptr;
*ptr = old & value;
return old;
```

`sceAtomicExchange16(ptr, swap)` executes an operation equivalent to the following code as an indivisible operation.

```
int16_t old = *ptr;
*ptr = swap;
return old;
```

`sceAtomicCompareAndSwap16(ptr, compare, swap)` executes an operation equivalent to the following code as an indivisible operation.

```
int16_t old = *ptr;
if(old == compare){
   *ptr = swap;
}
return old;
```

`sceAtomicLoad16(ptr)` executes an operation equivalent to the following code as an indivisible operation.

```
return *ptr;
```

`sceAtomicStore16(ptr, value)` executes an operation equivalent to the following code as an indivisible operation.

```
*ptr = value;
```

Memory Ordering

The above functions do not perform memory ordering control. If memory ordering control is required, use the following functions.

```
int16_t sceAtomicAdd16Acquire(volatile int16_t* ptr,int16_t value);
int16_t sceAtomicAdd16Release(volatile int16_t* ptr,int16_t value);
int16_t sceAtomicAdd16AcqRel(volatile int16_t* ptr,int16_t value);
int16_t sceAtomicAdd16Relaxed(volatile int16_t* ptr,int16_t value);

int16_t sceAtomicIncrement16Acquire(volatile int16_t* ptr);
int16_t sceAtomicIncrement16Release(volatile int16_t* ptr);
int16_t sceAtomicIncrement16AcqRel(volatile int16_t* ptr);
int16_t sceAtomicIncrement16Relaxed(volatile int16_t* ptr);

int16_t sceAtomicDecrement16Acquire(volatile int16_t* ptr);
int16_t sceAtomicDecrement16Release(volatile int16_t* ptr);
int16_t sceAtomicDecrement16AcqRel(volatile int16_t* ptr);
int16_t sceAtomicDecrement16Relaxed(volatile int16_t* ptr);

int16_t sceAtomicOr16Acquire(volatile int16_t* ptr,int16_t value);
int16_t sceAtomicOr16Release(volatile int16_t* ptr,int16_t value);
int16_t sceAtomicOr16AcqRel(volatile int16_t* ptr,int16_t value);
int16_t sceAtomicOr16Relaxed(volatile int16_t* ptr,int16_t value);

int16_t sceAtomicAnd16Acquire(volatile int16_t* ptr,int16_t value);
int16_t sceAtomicAnd16Release(volatile int16_t* ptr,int16_t value);
int16_t sceAtomicAnd16AcqRel(volatile int16_t* ptr,int16_t value);
int16_t sceAtomicAnd16Relaxed(volatile int16_t* ptr,int16_t value);

int16_t sceAtomicExchange16Acquire(volatile int16_t* ptr,int16_t swap);
int16_t sceAtomicExchange16Release(volatile int16_t* ptr,int16_t swap);
int16_t sceAtomicExchange16AcqRel(volatile int16_t* ptr,int16_t swap);
int16_t sceAtomicExchange16Relaxed(volatile int16_t* ptr,int16_t swap);

int16_t sceAtomicCompareAndSwap16Acquire(volatile int16_t* ptr,int16_t
compare,int16_t swap);
int16_t sceAtomicCompareAndSwap16Release(volatile int16_t* ptr,int16_t
compare,int16_t swap);
int16_t sceAtomicCompareAndSwap16AcqRel(volatile int16_t* ptr,int16_t
compare,int16_t swap);
int16_t sceAtomicCompareAndSwap16Relaxed(volatile int16_t* ptr,int16_t
compare,int16_t swap);

int16_t sceAtomicLoad16Acquire(volatile int16_t* ptr);
int16_t sceAtomicLoad16Release(volatile int16_t* ptr);
int16_t sceAtomicLoad16AcqRel(volatile int16_t* ptr);
int16_t sceAtomicLoad16Relaxed(volatile int16_t* ptr);

void sceAtomicStore16Acquire(volatile int16_t* ptr,int16_t value);
void sceAtomicStore16Release(volatile int16_t* ptr,int16_t value);
void sceAtomicStore16AcqRel(volatile int16_t* ptr,int16_t value);
void sceAtomicStore16Relaxed(volatile int16_t* ptr,int16_t value);
```

Functions with `Acquire` at the end guarantee that the memory operations performed by these functions will always be executed before later memory operations.

Functions with `Release` at the end guarantee that the memory operations performed before these functions were called will always be executed before the memory operations of these functions in other threads.

Functions with `AcqRel` at the end guarantee that the memory operations performed before these functions were called will always be executed before the memory operations of these functions, and guarantees that the memory operations of these functions will always be executed before later memory operations.

Functions with `Relaxed` at the end do not guarantee the order of memory operations before or after and are the same as functions with nothing added at the end.

32-Bit Data Functions

sceAtomicAdd32, sceAtomicIncrement32, sceAtomicDecrement32, sceAtomicOr32, sceAtomicAnd32, sceAtomicExchange32, sceAtomicCompareAndSwap32, sceAtomicLoad32, sceAtomicStore32

Atomic operation for 32-bit data

Definition

```
#include <sce_atomic.h>
int32_t sceAtomicAdd32(
        volatile int32_t* ptr,
        int32_t value
);
int32_t sceAtomicIncrement32(
        volatile int32_t* ptr
);
int32_t sceAtomicDecrement32(
        volatile int32_t* ptr
);
int32_t sceAtomicOr32(
        volatile int32_t* ptr,
        int32_t value
);
int32_t sceAtomicAnd32(
        volatile int32_t* ptr,
        int32_t value
);
int32_t sceAtomicExchange32(
        volatile int32_t* ptr,
        int32_t swap
);
int32_t sceAtomicCompareAndSwap32(
        volatile int32_t* ptr,
        int32_t compare,
        int32_t swap
);
int32_t sceAtomicLoad32(
        volatile int32_t* ptr
);
void sceAtomicStore32(
        volatile int32_t* ptr,
        int32_t value
);
```

Arguments

ptr	Memory to be updated
value	Operation argument
swap	Value to be written
compare	Value to be compared

Return Values

Refer to Description.

Description

`sceAtomicAdd32(`*`ptr`*`, `*`value`*`)` executes an operation equivalent to the following code as an indivisible operation.

```
int32_t old = *ptr;
*ptr = old + value;
return old;
```

`sceAtomicIncrement32(`*`ptr`*`)` executes an operation equivalent to the following code as an indivisible operation.

```
int32_t old = *ptr;
*ptr = old + 1;
return old;
```

`sceAtomicDecrement32(`*`ptr`*`)` executes an operation equivalent to the following code as an indivisible operation.

```
int32_t old = *ptr;
*ptr = old - 1;
return old;
```

`sceAtomicOr32(`*`ptr`*`, `*`value`*`)` executes an operation equivalent to the following code as an indivisible operation.

```
int32_t old = *ptr;
*ptr = old | value;
return old;
```

`sceAtomicAnd32(`*`ptr`*`, `*`value`*`)` executes an operation equivalent to the following code as an indivisible operation.

```
int32_t old = *ptr;
*ptr = old & value;
return old;
```

`sceAtomicExchange32(`*`ptr`*`, `*`swap`*`)` executes an operation equivalent to the following code as an indivisible operation.

```
int32_t old = *ptr;
*ptr = swap;
return old;
```

`sceAtomicCompareAndSwap32(`*`ptr`*`, `*`compare`*`, `*`swap`*`)` executes an operation equivalent to the following code as an indivisible operation.

```
int32_t old = *ptr;
if(old == compare){
   *ptr = swap;
}
return old;
```

`sceAtomicLoad32(`*`ptr`*`)` executes an operation equivalent to the following code as an indivisible operation.

```
return *ptr;
```

`sceAtomicStore32(`*`ptr`*`, `*`value`*`)` executes an operation equivalent to the following code as an indivisible operation.

```
*ptr = value;
```

Memory Ordering

The above functions do not perform memory ordering control. If memory ordering control is required, use the following functions.

```
int32_t sceAtomicAdd32Acquire(volatile int32_t* ptr,int32_t value);
int32_t sceAtomicAdd32Release(volatile int32_t* ptr,int32_t value);
int32_t sceAtomicAdd32AcqRel(volatile int32_t* ptr,int32_t value);
int32_t sceAtomicAdd32Relaxed(volatile int32_t* ptr,int32_t value);

int32_t sceAtomicIncrement32Acquire(volatile int32_t* ptr);
int32_t sceAtomicIncrement32Release(volatile int32_t* ptr);
int32_t sceAtomicIncrement32AcqRel(volatile int32_t* ptr);
int32_t sceAtomicIncrement32Relaxed(volatile int32_t* ptr);

int32_t sceAtomicDecrement32Acquire(volatile int32_t* ptr);
int32_t sceAtomicDecrement32Release(volatile int32_t* ptr);
int32_t sceAtomicDecrement32AcqRel(volatile int32_t* ptr);
int32_t sceAtomicDecrement32Relaxed(volatile int32_t* ptr);

int32_t sceAtomicOr32Acquire(volatile int32_t* ptr,int32_t value);
int32_t sceAtomicOr32Release(volatile int32_t* ptr,int32_t value);
int32_t sceAtomicOr32AcqRel(volatile int32_t* ptr,int32_t value);
int32_t sceAtomicOr32Relaxed(volatile int32_t* ptr,int32_t value);

int32_t sceAtomicAnd32Acquire(volatile int32_t* ptr,int32_t value);
int32_t sceAtomicAnd32Release(volatile int32_t* ptr,int32_t value);
int32_t sceAtomicAnd32AcqRel(volatile int32_t* ptr,int32_t value);
int32_t sceAtomicAnd32Relaxed(volatile int32_t* ptr,int32_t value);

int32_t sceAtomicExchange32Acquire(volatile int32_t* ptr,int32_t swap);
int32_t sceAtomicExchange32Release(volatile int32_t* ptr,int32_t swap);
int32_t sceAtomicExchange32AcqRel(volatile int32_t* ptr,int32_t swap);
int32_t sceAtomicExchange32Relaxed(volatile int32_t* ptr,int32_t swap);

int32_t sceAtomicCompareAndSwap32Acquire(volatile int32_t* ptr,int32_t
compare,int32_t swap);
int32_t sceAtomicCompareAndSwap32Release(volatile int32_t* ptr,int32_t
compare,int32_t swap);
int32_t sceAtomicCompareAndSwap32AcqRel(volatile int32_t* ptr,int32_t compare,
int32_t swap);
int32_t sceAtomicCompareAndSwap32Relaxed(volatile int32_t* ptr,int32_t
compare,int32_t swap);

int32_t sceAtomicLoad32Acquire(volatile int32_t* ptr);
int32_t sceAtomicLoad32Release(volatile int32_t* ptr);
int32_t sceAtomicLoad32AcqRel(volatile int32_t* ptr);
int32_t sceAtomicLoad32Relaxed(volatile int32_t* ptr);

void sceAtomicStore32Acquire(volatile int32_t* ptr,int32_t value);
void sceAtomicStore32Release(volatile int32_t* ptr,int32_t value);
void sceAtomicStore32AcqRel(volatile int32_t* ptr,int32_t value);
void sceAtomicStore32Relaxed(volatile int32_t* ptr,int32_t value);
```

Functions with `Acquire` at the end guarantee that the memory operations performed by these functions will always be executed before later memory operations.

Functions with `Release` at the end guarantee that the memory operations performed before these functions were called will always be executed before the memory operations of these functions in other threads.

Functions with `AcqRel` at the end guarantee that the memory operations performed before these functions were called will always be executed before the memory operations of these functions, and guarantees that the memory operations of these functions will always be executed before later memory operations.

Functions with `Relaxed` at the end do not guarantee the order of memory operations before or after and are the same as functions with nothing added at the end.

64-Bit Data Functions

sceAtomicAdd64, sceAtomicIncrement64, sceAtomicDecrement64, sceAtomicOr64, sceAtomicAnd64, sceAtomicExchange64, sceAtomicCompareAndSwap64, sceAtomicLoad64, sceAtomicStore64

Atomic operation for 64-bit data

Definition

```
#include <sce_atomic.h>
int64_t sceAtomicAdd64(
        volatile int64_t* ptr,
        int64_t value
);
int64_t sceAtomicIncrement64(
        volatile int64_t* ptr
);
int64_t sceAtomicDecrement64(
        volatile int64_t* ptr
);
int64_t sceAtomicOr64(
        volatile int64_t* ptr,
        int64_t value
);
int64_t sceAtomicAnd64(
        volatile int64_t* ptr,
        int64_t value
);
int64_t sceAtomicExchange64(
        volatile int64_t* ptr,
        int64_t swap
);
int64_t sceAtomicCompareAndSwap64(
        volatile int64_t* ptr,
        int64_t compare,
        int64_t swap
);
int64_t sceAtomicLoad64(
        volatile int64_t* ptr
);
void sceAtomicStore64(
        volatile int64_t* ptr,
        int64_t value
);
```

Arguments

ptr	Memory to be updated
value	Operation argument
swap	Value to be written
compare	Value to be compared

Return Values

Refer to Description.

Description

`sceAtomicAdd64(`*`ptr`*`, `*`value`*`)` executes an operation equivalent to the following code as an indivisible operation.

```
int64_t old = *ptr;
*ptr = old + value;
return old;
```

`sceAtomicIncrement64(`*`ptr`*`)` executes an operation equivalent to the following code as an indivisible operation.

```
int64_t old = *ptr;
*ptr = old + 1;
return old;
```

`sceAtomicDecrement64(`*`ptr`*`)` executes an operation equivalent to the following code as an indivisible operation.

```
int64_t old = *ptr;
*ptr = old - 1;
return old;
```

`sceAtomicOr64(`*`ptr`*`, `*`value`*`)` executes an operation equivalent to the following code as an indivisible operation.

```
int64_t old = *ptr;
*ptr = old | value;
return old;
```

`sceAtomicAnd64(`*`ptr`*`, `*`value`*`)` executes an operation equivalent to the following code as an indivisible operation.

```
int64_t old = *ptr;
*ptr = old & value;
return old;
```

`sceAtomicExchange64(`*`ptr`*`, `*`swap`*`)` executes an operation equivalent to the following code as an indivisible operation.

```
int64_t old = *ptr;
*ptr = swap;
return old;
```

`sceAtomicCompareAndSwap64(`*`ptr`*`, `*`compare`*`, `*`swap`*`)` executes an operation equivalent to the following code as an indivisible operation.

```
int64_t old = *ptr;
if(old == compare){
   *ptr = swap;
}
return old;
```

`sceAtomicLoad64(`*`ptr`*`)` executes an operation equivalent to the following code as an indivisible operation.

```
return *ptr;
```

`sceAtomicStore64(`*`ptr`*`, `*`value`*`)` executes an operation equivalent to the following code as an indivisible operation.

```
*ptr = value;
```

Memory Ordering

The above functions do not perform memory ordering control. If memory ordering control is required, use the following functions.

```
int64_t sceAtomicAdd64Acquire(volatile int64_t* ptr,int64_t value);
int64_t sceAtomicAdd64Release(volatile int64_t* ptr,int64_t value);
int64_t sceAtomicAdd64AcqRel(volatile int64_t* ptr,int64_t value);
int64_t sceAtomicAdd64Relaxed(volatile int64_t* ptr,int64_t value);

int64_t sceAtomicIncrement64Acquire(volatile int64_t* ptr);
int64_t sceAtomicIncrement64Release(volatile int64_t* ptr);
int64_t sceAtomicIncrement64AcqRel(volatile int64_t* ptr);
int64_t sceAtomicIncrement64Relaxed(volatile int64_t* ptr);

int64_t sceAtomicDecrement64Acquire(volatile int64_t* ptr);
int64_t sceAtomicDecrement64Release(volatile int64_t* ptr);
int64_t sceAtomicDecrement64AcqRel(volatile int64_t* ptr);
int64_t sceAtomicDecrement64Relaxed(volatile int64_t* ptr);

int64_t sceAtomicOr64Acquire(volatile int64_t* ptr,int64_t value);
int64_t sceAtomicOr64Release(volatile int64_t* ptr,int64_t value);
int64_t sceAtomicOr64AcqRel(volatile int64_t* ptr,int64_t value);
int64_t sceAtomicOr64Relaxed(volatile int64_t* ptr,int64_t value);

int64_t sceAtomicAnd64Acquire(volatile int64_t* ptr,int64_t value);
int64_t sceAtomicAnd64Release(volatile int64_t* ptr,int64_t value);
int64_t sceAtomicAnd64AcqRel(volatile int64_t* ptr,int64_t value);
int64_t sceAtomicAnd64Relaxed(volatile int64_t* ptr,int64_t value);

int64_t sceAtomicExchange64Acquire(volatile int64_t* ptr,int64_t swap);
int64_t sceAtomicExchange64Release(volatile int64_t* ptr,int64_t swap);
int64_t sceAtomicExchange64AcqRel(volatile int64_t* ptr,int64_t swap);
int64_t sceAtomicExchange64Relaxed(volatile int64_t* ptr,int64_t swap);

int64_t sceAtomicCompareAndSwap64Acquire(volatile int64_t* ptr,int64_t
compare,int64_t swap);
int64_t sceAtomicCompareAndSwap64Release(volatile int64_t* ptr,int64_t
compare,int64_t swap);
int64_t sceAtomicCompareAndSwap64AcqRel(volatile int64_t* ptr,int64_t compare,
int64_t swap);
int64_t sceAtomicCompareAndSwap64Relaxed(volatile int64_t* ptr,int64_t
compare,int64_t swap);

int64_t sceAtomicLoad64Acquire(volatile int64_t* ptr);
int64_t sceAtomicLoad64Release(volatile int64_t* ptr);
int64_t sceAtomicLoad64AcqRel(volatile int64_t* ptr);
int64_t sceAtomicLoad64Relaxed(volatile int64_t* ptr);

void sceAtomicStore64Acquire(volatile int64_t* ptr,int64_t value);
void sceAtomicStore64Release(volatile int64_t* ptr,int64_t value);
void sceAtomicStore64AcqRel(volatile int64_t* ptr,int64_t value);
void sceAtomicStore64Relaxed(volatile int64_t* ptr,int64_t value);
```

Functions with `Acquire` at the end guarantee that the memory operations performed by these functions will always be executed before later memory operations.

Functions with `Release` at the end guarantee that the memory operations performed before these functions were called will always be executed before the memory operations of these functions in other threads.

Functions with `AcqRel` at the end guarantee that the memory operations performed before these functions were called will always be executed before the memory operations of these functions, and guarantees that the memory operations of these functions will always be executed before later memory operations.

Functions with `Relaxed` at the end do not guarantee the order of memory operations before or after and are the same as functions with nothing added at the end.

Memory Barrier Functions

sceAtomicMemoryBarrier

Memory barrier function

Definition

```
#include <sce_atomic.h>
void sceAtomicMemoryBarrier(void);
```

Arguments

None

Return Values

None

Description

This function guarantees that the order of the memory operations before and after this function is called does not change.