NAME

ATOMIC_VAR_INIT, **atomic_init**, **atomic_load**, **atomic_store**, **atomic_exchange**, **atomic_compare_exchange_strong**, **atomic_compare_exchange_weak**, **atomic_fetch_add**, **atomic_fetch_and**, **atomic_fetch_or**, **atomic_fetch_sub**, **atomic_fetch_xor**, **atomic_is_lock_free** - type-generic atomic operations

SYNOPSIS

#include <stdatomic.h>

_Atomic(T) *v* = ATOMIC_VAR_INIT(c);
void
atomic_init(*_Atomic(T) *object*, *T value*);

T
atomic_load(*_Atomic(T) *object*);

T
atomic_load_explicit(*_Atomic(T) *object*, *memory_order order*);

void
atomic_store(*_Atomic(T) *object*, *T desired*);

void
atomic_store_explicit(*_Atomic(T) *object*, *T desired*, *memory_order order*);

T
atomic_exchange(*_Atomic(T) *object*, *T desired*);

T
atomic_exchange_explicit(*_Atomic(T) *object*, *T desired*, *memory_order order*);

_Bool
atomic_compare_exchange_strong(*_Atomic(T) *object*, *T *expected*, *T desired*);

_Bool
atomic_compare_exchange_strong_explicit(*_Atomic(T) *object*, *T *expected*, *T desired*, *memory_order success*, *memory_order failure*);

_Bool
atomic_compare_exchange_weak(*_Atomic(T) *object*, *T *expected*, *T desired*);

_Bool
atomic_compare_exchange_weak_explicit(*_Atomic(T) *object*, *T *expected*, *T desired*, *memory_order success*, *memory_order failure*);

T
atomic_fetch_add(*_Atomic(T) *object*, *T operand*);

T
atomic_fetch_add_explicit(*_Atomic(T) *object*, *T operand*, *memory_order order*);

T
atomic_fetch_and(*_Atomic(T) *object*, *T operand*);

T
atomic_fetch_and_explicit(*_Atomic(T) *object*, *T operand*, *memory_order order*);

T
atomic_fetch_or(*_Atomic(T) *object*, *T operand*);

T
atomic_fetch_or_explicit(*_Atomic(T) *object*, *T operand*, *memory_order order*);

T
atomic_fetch_sub(*_Atomic(T) *object*, *T operand*);

T
atomic_fetch_sub_explicit(*_Atomic(T) *object*, *T operand*, *memory_order order*);

T
atomic_fetch_xor(*_Atomic(T) *object*, *T operand*);

T
atomic_fetch_xor_explicit(*_Atomic(T) *object*, *T operand*, *memory_order order*);

_Bool
atomic_is_lock_free(*const _Atomic(T) *object*);

DESCRIPTION

The header <*stdatomic.h*> provides type-generic macros for atomic operations. Atomic operations can be used by multithreaded programs to provide shared variables between threads that in most cases may

be modified without acquiring locks.

Atomic variables are declared using the **_Atomic**() type specifier. These variables are not type-compatible with their non-atomic counterparts. Depending on the compiler used, atomic variables may be opaque and can therefore only be influenced using the macros described.

The **atomic_init**() macro initializes the atomic variable *object* with a *value*. Atomic variables can be initialized while being declared using **ATOMIC_VAR_INIT**().

The **atomic_load**() macro returns the value of atomic variable *object*. The **atomic_store**() macro sets the atomic variable *object* to its *desired* value.

The **atomic_exchange**() macro combines the behaviour of **atomic_load**() and **atomic_store**(). It sets the atomic variable *object* to its desired *value* and returns the original contents of the atomic variable.

The **atomic_compare_exchange_strong**() macro stores a *desired* value into atomic variable *object*, only if the atomic variable is equal to its *expected* value. Upon success, the macro returns true. Upon failure, the *desired* value is overwritten with the value of the atomic variable and false is returned. The **atomic_compare_exchange_weak**() macro is identical to **atomic_compare_exchange_strong**(), but is allowed to fail even if atomic variable *object* is equal to its *expected* value.

The **atomic_fetch_add**() macro adds the value *operand* to atomic variable *object* and returns the original contents of the atomic variable.

The **atomic_fetch_and**() macro applies the *and* operator to atomic variable *object* and *operand* and stores the value into *object*, while returning the original contents of the atomic variable.

The **atomic_fetch_or**() macro applies the *or* operator to atomic variable *object* and *operand* and stores the value into *object*, while returning the original contents of the atomic variable.

The **atomic_fetch_sub**() macro subtracts the value *operand* from atomic variable *object* and returns the original contents of the atomic variable.

The **atomic_fetch_xor**() macro applies the *xor* operator to atomic variable *object* and *operand* and stores the value into *object*, while returning the original contents of the atomic variable.

The **atomic_is_lock_free**() macro returns whether atomic variable *object* uses locks when using atomic operations.

BARRIERS

The atomic operations described previously are implemented in such a way that they disallow both the compiler and the executing processor to re-order any nearby memory operations across the atomic operation. In certain cases this behaviour may cause suboptimal performance. To mitigate this, every atomic operation has an **_explicit**() version that allows the re-ordering to be configured.

The *order* parameter of these **_explicit**() macros can have one of the following values.

memory_order_relaxed No operation orders memory.

memory_order_consume
Perform consume operation.

memory_order_acquire Acquire fence.

memory_order_release Release fence.

memory_order_acq_rel Acquire and release fence.

memory_order_seq_cst
Sequentially consistent acquire and release fence.

The previously described macros are identical to the **_explicit**() macros, when *order* is memory_order_seq_cst.

COMPILER SUPPORT

These atomic operations are typically implemented by the compiler, as they must be implemented type-generically and must often use special hardware instructions. As this interface has not been adopted by most compilers yet, the <*stdatomic.h*> header implements these macros on top of existing compiler intrinsics to provide forward compatibility.

This means that certain aspects of the interface, such as support for different barrier types may simply be ignored. When using GCC, all atomic operations are executed as if they are using memory_order_seq_cst.

Instead of using the atomic operations provided by this interface, ISO/IEC 9899:2011 ("ISO C11") allows the atomic variables to be modified directly using built-in language operators. This behaviour cannot be emulated for older compilers. To prevent unintended non-atomic access to these variables, this header file places the atomic variable in a structure when using an older compiler.

When using GCC on architectures on which it lacks support for built-in atomic intrinsics, these macros

may emit function calls to fallback routines. These fallback routines are only implemented for 32-bits and 64-bits datatypes, if supported by the CPU.

SEE ALSO

pthread(3), atomic(9)

STANDARDS

These macros attempt to conform to ISO/IEC 9899:2011 ("ISO C11").

HISTORY

These macros appeared in FreeBSD 10.0.

AUTHORS

Ed Schouten <*ed@FreeBSD.org*>
David Chisnall <*theraven@FreeBSD.org*>