Accredited Standards Committee X3
Information Processing Systems
Operating under the procedures of
American National Standards Institute

Doc No: X3J16/95-0107 WG21/N0707
Date: May 30, 1995
Project: Programming Language C++
Reply to: Michael Eager
(eager@mri.com)

Volatile Semantics

Introduction

The semantics of the volatile keyword are described in very limited terms in both the ANSI C standard and the Working Paper [1.8--Program execution]. Essentially, the only semantics defined is that at a sequence point all volatile objects have their values assigned.

There is a note in [7.1.5.1--The cv-qualifiers] that "volatile is a hint to to the processor to avoid aggressive optimization involving the object because the value of the object may be changed by means undetectable by a processor." Exactly what aggressive optimizations are is left to the imagination.

A flurry of about 100 messages on the core reflector in February and March revealed that "everyone knows what volatile means" but that everyone's definition is different. Lacking a clear definition, it is difficult to provide either compiler writers or users with guidance on what volatile should do and how it should be used.

The Problem

The current semantics for volatile are to provide minimal support for signal handling through the use of the sig_atomic_t type (which is completely undefined). This type is intended to be accessed atomically. A volatile object of this time will have the last value assigned if a signal handler is entered. This is a very limited application.

Volatile is used in a number of different contexts:

- -- Access to machine registers (both fetch and store)
- -- Reference to memory mapped hardware (I/O ports or clock)
- -- Implementation of shared memory interlocks
- -- Forcing explicit access to memory for each fetch or store

In a number of cases, the number of accesses and whether they are fetches or stores is critical to the correct operation of the program.

Lacking clear semantics for volatile means that the behavior of the program which is most critical to the application is undefined. The programmer is unable to write C++ code which are assured to perform the operations intended.

Additionally, although the semantics of volatile are essentially undefined, the Working Paper makes volatile a different type, requiring additional effort on the part of the user (for example, to explicity create copy constructors) which are not required if the volatile qualifier is not used. This also creates incompatibilities with C with no clear benefit or purpose for the differences.

The Strawmen

There are a number of arguments which have been proposed for why we should not define volatile semantics.

Arg 1. Volatile is inherently hardware dependent; let's leave it completely implementation defined.

Ans 1. Being implementation defined is more restrictive than just being hardware dependent. It means that two different implementations for the same hardware, or even two different versions of the same compiler may have different behaviors. And there is no way to decide which has the correct behavior.

Arg 2. Let's leave it to the implementation to define their volatile semantics.

Ans 2. Volatile is not listed as implementation defined, which would require that the implementation provide a description. Nor does the standard provide a range of possible behaviors, as required in [1.3--Definitions].

Arg 3. Volatile requires the description of how to mask interrupts and lock threads [see c++std-core-5253].

Ans 3. Nope. You might use a volatile object to implement a thread locking scheme, but that is derived from the type, not basic to its use.

Arg 4. The user can write constructors and member functions which implement the desired semantics for volatile [see c++std-core-5214]. Volatile is a signal that we are doing something outside of the language which cannot be described within C or C++ [see c++std-core-5256].

Ans 4. These two seem to contradict each other. If the meaning of access to a volatile object is ill-defined, there is little reason to expect that it will be better defined in a member function. And if it truely is something that cannot be defined within the language, such as issuing hardware specific instructions, probably one needs to descend to assembly language.

Arg 5. We don't have enough experience with using volatile to define it clearly, and it is much too complicated.

Ans 5. Volatile has been in use and misuse for ten years; this should give more than enough experience with it's use. It's much less complex that many other areas of C++, for example, the compilatin model.

Proposed Changes
----

1. Section 1.8 -- Program Execution

Paragraph 7 contains the following:

> Accessing an object designated by a volatile lvalue, modifying an object, modifying a file, or calling a function that does any of those operations are all _side effects_, which are changes in the state of the execution environment. Evaluation of an expression might produce side effects. At certain specified points in the

execution sequence called sequence points, all side effects of
previous evaluations shall be complete and no side effects of
subsequent evaluations shall have taken place.

I propose that this be followed with:

Every access to an object (of certain implementation defined types[*]) designated by a volatile lvalue, whether to obtain the object's value or to set it, must occur in exactly the order required by the abstract machine model. No access shall occur prior to the sequence point preceeding the expression containing the reference to the volatile lvalue, nor after the sequence point following the instruction. No other access to a volatile lvalue shall occur.

[*] It may not be possible to read or write a volatile type in a single access for some types, e.g., bitfields. It may also not be possible to access certain types without also accessing adjacent types, e.g., when the type can only be accessed as part of a larger unit. These types are specified by the implementation. Certainly, sig_atomic_t should be in this list.

There may be no more and no fewer accesses to an object referenced by a volatile lvalue than are expressed in the source code.

Some of the exmaples mentioned in previous discussions are resolved by this change:

Ex 1. `int foo () { volatile int a = 0; return a; }`

This must result in a single store to a to set it to zero. There must be a single load of a prior to returning its value.

Ex 2. `void foo () { volatile int a; a = 0; }`

This must result in a single store to a to set it to zero. Instructions which do read/modify/write may not be used.

Ex. 3. `void foo () { volatile int a, b; a = b = 0; }`

This must result in a single store of 0 to b, then a single store of 0 to a. Section 3.10(p6) says "whenever an lvalue appears in a context where an lvalue is not expected, the lvalue is converted to an rvalue".

Ex 4. `struct X { volatile int a:8; volatile int b:24; } x; x.a = 5;`

It is implementation defined whether there is a single store to the 8 bits containing a, or whether b is also accessed when a is accessed.

2. Adopt solution 2 from 95-0056:

A copy constructor and copy assignment operator may have the form

`X ([cv] X&), operator= ([cv] X&)`

If a class does not have a user-declared copy constructor or copy assigment operator, one is implicitly declared.