By Jan Kristoffersen, RAMTEX Engineering ApS

Atomic operations with multi-threaded environments.

This document summarize some aspects and thoughts about atomic operations in multi-treaded environments. The intend is to propose some solution which later may be added to both C and C++.

In general atomic operations can be looked upon at in two slightly different ways :

A Use of object or resource reservation. If one thread reserves a resource then competing threads are told: "This resource is locked, go and do something else, or give up your time slot".
This kind of atomic operations are traditional implemented by use of concepts like mutex or semaphores to give a thread mutually exclusive access to a resource. The concept requires use of a scheduler. Primary design issues are avoidance of deadlocks and perhaps the amount of time a resource is locked.

B Use of execution locking . If one thread needs exclusive access to an object then all, or just competing threads, are prevented from even getting executing time for as long as the locking last. The concept used is: "I do not want anybody to interrupt me for a while".
This kind of atomic object access can only be realized if supported by hardware.
The concept does not necessarily require use of a scheduler. Thread deadlocks can not take place. A primary design issues is the amount of time a resource is locked.

Method B applies both for hosted environments and for freestanding environments.
This document focus on method B when using the word *atomic*.
For comparison the *atomic* protection concept described in this document is an abstraction in level with *spin-locks* (concept used by Linux).

The *execution locking* method is a low-level approach which require that special machine instruction are generated by the compiler. For instance machine instructions for access locking in multi-processor environments or interrupt blocking in single-processor environments.

The main purpose for standardizing the syntax for atomic operations is to make it possible to write compiler and processor independent driver code which operates in a multi threaded (or multi-processor) environment.

A typical application where *atomic* support is useful is in software fifo-buffer implementations. Here one end of the fifo may be activated from interrupt level and the other from main level. If the language has support for *atomic* operations then the code implementing such fifo drivers could be completely compiler and processor independent.

The nature of atomic operations

The purpose of atomic operations is often not just to protect a single object, but to protect an execution sequence involving one or more operations on one or more objects.

Often the code which absolutely *must* be executed in a atomic fashion in order to make a system fail-safe will involve relatively few and simple object operations like test-and-set , store-and-increment, fetch-and-decrement etc.

Atomic activation / deactivation is symmetric by nature, in the sense that an atomic-activation operation must always be followed by an atomic deactivation operation (otherwise we no longer have a multi-threaded system).

The methods available to a compiler vendor for implementing atomic operations depends very much on the processor architecture and on the operating system. The syntax for atomic operations should be able to completely encapsulate the underlying mechanisms used for creating atomic operations in order to assure source portability.

Protection of execution sequences are supported by hardware in nearly all processor architectures. The hardware support for atomic execution can for instance be: interrupt enable / disable mechanisms, modify / restore of interrupt mask levels, use of special instructions which protect "n" numbers of the following instruction fetches from being interrupted, use of hardware for bus-locking etc.

The importance of this is that focusing on *atomic execution sequences* has the potential of generating much more efficient code, than more high-level software mechanisms like use of mutex objects etc.

A proposed syntax and semantic for atomic
The proposed syntax is a keyword followed by a compound statement block.
The proposed keyword is **atomic** i.e. an addition to the core language. The word *atomic* is used here for the rest of the discussion.

```
atomic
  {
  // statement sequence to be executed in an atomic fashion
  }
```

The atomic block must be used inside a function block. At the '{' following the atomic keyword the compiler inserts any instructions required to in order to ensure that the compound statements in the block are executed in an atomic fashion. At the '}' terminating the compound statement block the compiler inserts any instructions required to terminate the atomic protection.

```
volatile int myflag;
int get_and_set(void)
   {
   int oldflag;
   atomic
```

```
    { // Start of atomic protection
    oldflag = myflag;
    myflag = 1;
    } // End of atomic protection
  return oldflag;
  }
```

The use of a compound statement block emphasis the fact that atomic activation / deactivation should be done in a symmetric fashion.

Nesting of atomic blocks

Nested invocation of atomic must be allowed and be execution safe in order to facilitate library design.

If nested invocation of atomic is used then deactivation of atomic protection will first take place at the end of the outer atomic block where atomic protection was first activated.

The following examples should generate valid code:

Example A:

```
void foo(void) {
  atomic {            // atomic is activated here
    // some code
    atomic {          // has no effect on atomic protection
       // some code
    }                 // has no effect on atomic protection
  // some code
  }                   // atomic is deactivated here
}
```

Example B:

```
void foo1(void)
{
  atomic {               // atomic is activated here
    foo2();
  }                      // atomic is deactivated here
}

void foo2(void) {
  atomic {               // has no effect on atomic protection
                         // when called from foo()
   // do something else
  }
}
```

Hidden objects

The above examples also indicates that a implementation of **atomic** involves use of at least two hidden objects:

One global object holding information about the global atomic protection state and one

temporary object for each **atomic** invocation holding information about the state of the global protection object at the time when entering the atomic compound statement block, so the global state can be restored at exit from the compound block.

Below the surface
Depending on the given execution environment the activation / deactivation of atomic execution may involve calls to the operating system or may be simple insertion of hardware instructions which activates / deactivates the atomic protection.
The compiler generated code for atomic execution could in principle on some platforms be as simple as this:

```
push      status_flag_register          // atomic {
clear     interrupt_enable_flag
// some machine instructions to be protected
pop       status_flag_register          //    }
```

Here the status_flag_register is the object holding the global atomic state information and the copy pushed on the stack is the temporary atomic state object.

Jump out of an atomic block
If the code contains a jump out of an atomic block the compiler should insert code which ends the atomic operation before the jump itself takes place. Jump into an atomic block should not be allowed.

```
void foo(void)
   {
   atomic
     {               // atomic protection activativation
     // some code
     goto myend;     // Protection deactivation before jump
     // some code
     }               // Normal atomic protection deactivation
   // some code
   myend:
   }
```

Escape functionality like **longjmp,** and in C++ **try**, **catch,** should include save / restore of the global atomic state object.

If the atomic keyword can not be supported
The atomic keyword is allowed to be ignored if the processor architecture and/or operating system does not support mechanisms for implementing atomic protection. In that case the syntax is equal to use of the compound statement block alone.

If OS scheduler functions are called
In an hosted environment if some Operating System scheduler function is called from within the *atomic* block, then no guaranties can be given whether the atomic protection are

maintained throughout the *atomic* block scope.

```
volatile int aval;
void foo(void)
   {
   atomic
     {         // atomic protection activativation
     aval = 1;
     Sleep(50);// Bad design. Some OS function is called
               // which may change thread / process
               // Atomic execution may be lost
     aval += 2;// No guarantie that aval get the value of 3
     }         // Atomic protection deactivation
   }
```

Final comment

It should be noted that the *atomic* keyword will not, and should not, make any protection from the consequences of a bad design or misused language features when executing in a multi-threaded environment. The language concept is still “trust the programmer”.

The purpose for the *atomic* keyword is only to make it possible to write portable code where atomic execution is needed for a short amount of time. Preferable *atomic* protection should be implemented by the compiler with as little runtime overhead as possible.