HCS12, like many embedded chips, maps device registers into RAM

In both C and assembly, register accesses look just like global variable accesses

However, these registers do not act like RAM

  Each read may return a different value
  Writes may be ignored
  Reads and writes can be side effecting
Your compiler tries very hard to generate fast code, and performs many optimizations

- Performing math on constants at compile time
- Eliminating dead code
- Turning multiplies by power-of-two into shifts
- Eliminating redundant memory operations
- Caching frequently-used variables in registers

These optimizations are usually good, but may be very bad when they are applied to device register accesses
You write this code:

```c
extern char MY_PTJ @ ( 0x00000268 ) ;

void Out(unsigned char data) {
    MY_PTJ = 0;
    PTT=data;
    MY_PTJ = 1;
}
```

CodeWarrior for HCS12 gives you this:

```
STAB _PTT
LDAB #1
STAB MY_PTJ
RTS
```

What is wrong?
Bad Optimization Example

Better register declaration:

```c
extern volatile char MY_PTJ @ ( 0x00000268 ) ;
```

```c
void Out(unsigned char data) {
    MY_PTJ = 0;
    PTT=data;
    MY_PTJ = 1;
}
```

For the same C code, CodeWarrior for HCS12 gives you this:

```asm
CLR MY_PTJ
STAB _PTT
LDAB #1
STAB MY_PTJ
RTS
```

Is it right now?
Without `volatile` it is impossible to reliably access device registers from C code.
Interrupt Synchronization

This kind of code is common:

```c
int flag = 0;

void interrupt 7 my_isr (void) {
    // isr code...
    flag = 1;
}

void wait_for_interrupt (void) {
    while (1) {
        if (flag == 1) return;
    }
}
```

What happens when we compile this?
ColdFire Translation

_wait_for_interrupt:
  link   a6,#0
  tst.l  _flag
  seq    d0
  extb.l d0
  neg.l  d0

loop:
  tst.l  d0
  bne.s  loop
  unlk   a6
  rts
_wait_for_interrupt:
    link    a6,#0
loop:
    tst.l   _flag
    beq.s   loop
    unlk    a6
    rts
Without `volatile` it is impossible to reliably synchronize between main and interrupts in C code.
Always make any variable volatile when it:
- Represents a device register
- Is used to communicate with interrupt handlers
- Is used to communicate between threads

What happens if you forget?
volatile is a “storage qualifier” in C
Like const it lets you tell the compiler something special about a variable
Any C type can be marked as volatile
Composite types (structs and arrays) can be all volatile, or contain volatile fields/elements
What does volatile mean?

The “volatile rules” that the compiler must obey are:

- Every assignment to a volatile variable in C code must turn into a store to that variable in the compiler output.
- Every read from a volatile variable in C code must turn into a load from that variable in the compiler output.
- The order of operations on volatile variables in C code cannot be changed in the compiler output.
  - This applies to individual volatile variables and to collections of them.
  - No guarantee is made about the relative ordering of volatile and non-volatile accesses.

Basically, volatile variables are not optimized by the compiler.
You write this code which uses `buffer_ready` to tell an interrupt handler that the buffer has been initialized:

```c
volatile int buffer_ready;
char buffer[BUF_SIZE];

void buffer_init() {
    int i;
    for (i=0; i<BUF_SIZE; i++)
        buffer[i] = 0;
    buffer_ready = 1;
}
```

The compiler can move the store to `buffer_ready` above the initialization loop. Solutions?
Volatile != Atomic

Accesses to volatile variables are not necessarily atomic.
For correct interrupt synchronization, you need to both use
volatiles and also begin_critical() / end_critical()
Does this make sense?

const volatile int x;
You can make a “pointer to volatile int”
int volatile *x;
You can make a “volatile pointer to int”
int *volatile x;
You can make a “volatile pointer to volatile int”
int volatile *volatile x;
The volatile rules apply to each volatile variable.
How to know which of these to use?
Even good embedded developers have to think hard about this!
Typedefs can help avoid confusion:

typedef volatile char my_volatile_char;
my_volatile_char *pc;
Volatile Summary

Learn to love volatile