Volatile

When you have excluded the impossible, whatever remains, however improbable, must be the truth

Sir Arthur Conan Doyle/Sherlock Holmes

While developing and testing a project the optimization settings for the compiler are usually set to a "Debug" environment. This hinders the compiler from some optimizations as

- Putting local variables into CPU-registers
- Skipping strict stack-frame allocations when unnecessary
- Removing unreferenced code blocks

These optimization would hinder the debug-process as how the debugger should know in which register a variable is maintained or where a local var is to find when there is no stack frame.
So near the project's end the compiler-settings get changed to perform more optimization, let the optimize target be better speed or fewer flash.
And all of a sudden the formerly running project stops working!
Switching the compiler back to debug-mode - project works, switching to optimization - project does not work.
Compiler error??? There might be some, but they are very rare. Since the poor developer cannot find an explanation he sticks to the "debug" settings and crosses his fingers that he will not run into limitations of speed or flash usage.
The explanation is: the cause has to do with interrupts, but the location of the error is somewhere else.
Let us assume an interrupt handler that does nothing more than setting a flag (a global variable) which is tested at a different part of the program.
The handler looks like

```
...
     FlagToTest = TRUE;
...
```

The polling in main() looks like

```
...
     while(! FlagToTest) Wait();
...
```

When compiling the "while" some funny things happen in my GCC: The call to "Wait() gets optimized-out since that function is considered to do nothing which is quite right.
A second optimization step is to drag the "! FlagToTest" out of the while loop and put it in front of it leaving an infinite loop behind.

```
     if(! FlagToTest)
     {
          while(1);
     }
```

From the point of optimization this is totally correct, the access of the var "FlagToTest" was taken out of the loop because it is an invariant (not changed within the loop).
As a matter of fact, this is exactly what we do not want! Fortunately there is a type qualifier we could apply to the declaration of the variable which will prevent the optimizer from taking the var out of a loop: "volatile"
So a declaration as

```
volatile char FlagToTest = 0;     // May be changed by an interrupt handler
```

will not suffer from the above over-optimization.

This is crucial: Every global variable that can get altered by an interrupt-handler must be declared as "volatile".