Memory Refresh

Memory refresh is the process of periodically reading information from an area of computer memory and immediately rewriting the read information to the same area without modification, for the purpose of preserving the information. Memory refresh is a background maintenance process required during the operation of semiconductor dynamic random-access memory (DRAM), the most widely used type of computer memory, and in fact is the defining characteristic of this class of memory.

In a DRAM chip, each bit of memory data is stored as the presence or absence of an electric charge on a small capacitor on the chip. As time passes, the charges in the memory cells leak away, so without being refreshed the stored data would eventually be lost. To prevent this, external circuitry periodically reads each cell and rewrites it, restoring the charge on the capacitor to its original level. Each **memory refresh cycle** refreshes a succeeding area of memory cells, thus repeatedly refreshing all the cells in a consecutive cycle. This process is conducted automatically, in the background, by the memory circuitry, while the computer is on, and is transparent to the user. While a refresh cycle is occurring the memory is not available for normal read and write operations, but in modern memory this "overhead" time is not large enough to significantly slow down memory operation.

Electronic memory that does not require refreshing is available, called static random-access memory (SRAM).SRAM circuits require more area, because an SRAM memory cell requires four to six transistors, compared to a single transistor and a capacitor for DRAM. As a result, data density is much lower in SRAM than in DRAM, and SRAM has higher price per bit. Therefore, DRAM is used for the main memory in computers, video game consoles, graphics cards and applications requiring large capacities and low cost.The need for memory refresh makes DRAM timing significantly more complicated than SRAM circuits, but the density and cost advantages of DRAM justify this complexity.

ECC Memory

Error-correcting code memory (**ECC memory**) is a type of computer data storage that can detect and correct the most common kinds of internal data corruption. ECC memory is used in most computers where data corruption cannot be tolerated under any circumstances, such as for scientific or financial computing.

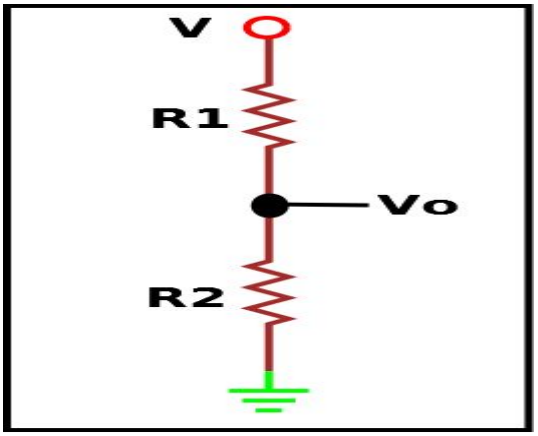
Several approaches have been developed to deal with unwanted bit-flips, including immunity-aware programming, RAM parity memory, and ECC memory.

This problem can be mitigated by using DRAM modules that include extra memory bits and memory controllers that exploit these bits. These extra bits are used to record parity or to use an error-correcting code(ECC). Parity allows the detection of all single-bit errors (actually, any odd number of wrong bits). The most common error correcting code, a single-error correction and double-error detection (SECDED) Hamming code, allows a single-bit error to be corrected and (in the usual configuration, with an extra parity bit) double-bit errors to be detected. Chipkill ECC

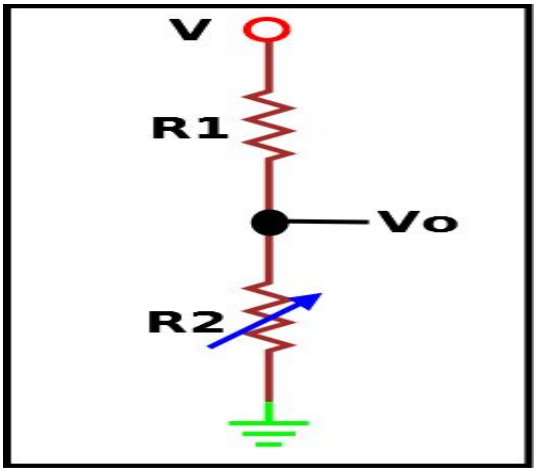
is a more effective version that also corrects for multiple bit errors, including the loss of an entire memory chip.

What is a transistor, how does it work, and how can it be used as an amplifier or switch?

To understand how an amplifier works, you need to first understand how a Voltage Divider circuit functions. Below is a simple Voltage Divider Circuit.

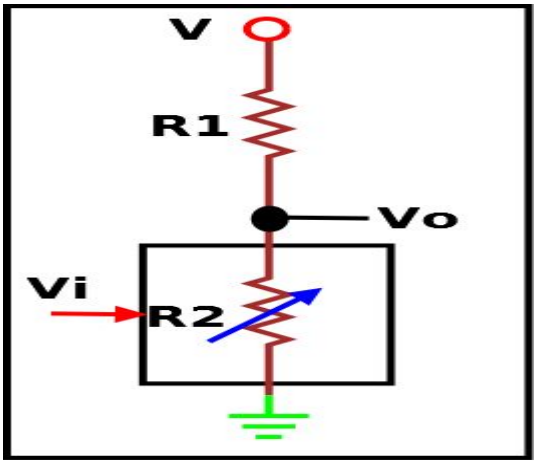


The output Vo depends on V, R1 and R2. For example if V = 100V, R1 = 40 Ohms and R2 = 60 Ohms. Then Vo = V * (R2/(R1+R2) = 100 * (60/100) = 60V. By changing the values of V, R1 and R2 the output Vo can be changed. Now let us change the resistor R2 with a Variable Resistor.

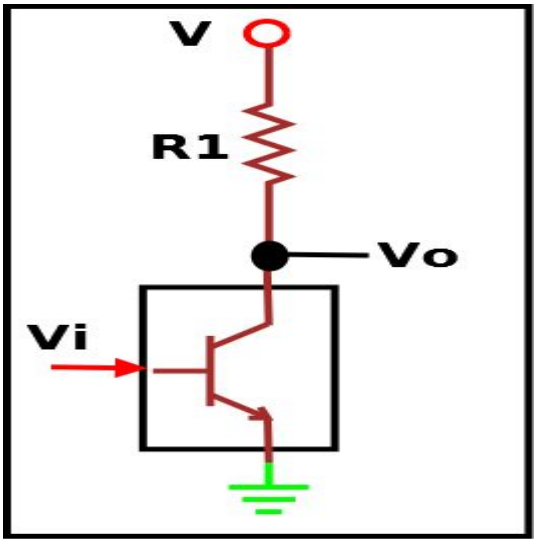


In the above circuit V and R1 are fixed and R2 is a variable. So, if we change R2, Vo will change. We generally call this as Regulator. Now let us

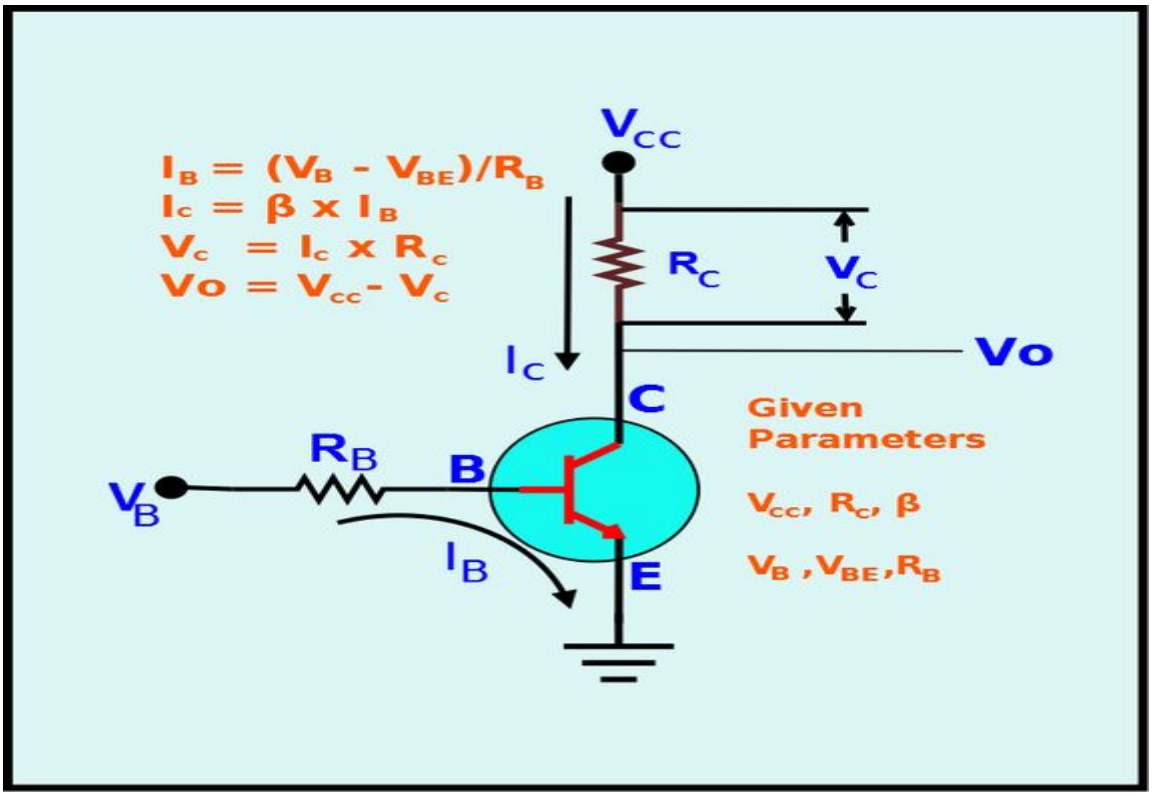
have a variable resistor whose resistance can be changed by the voltage instead of manual control.



In the above circuit the value of R2 is changed by the voltage Vi. If we change Vi then Vo is changed. The relation between Vo and Vi is called Amplification Factor. Here is a surprise. The resistor who resistance is changed by voltage (current) is nothing but a **Transistor**.



Actually Transistor acts as only a variable resistor. The value of resistor between Collector and Emitter is changed by the base current.



The Transistor acts as a Regulator (Variable Resistor) or a switch(ON/OFF).

The Transistor has 3 operating modes.

1. Cut-off (Switch - OFF)
2- Saturation (Switch - ON)
3- Active (Regulator).

1. **Cut-off Mode**

Vb < Vbe (Generally 0.7V)

So Ib = 0A

Ic = 0A

Vc = Ic x Rc = 0V

Vo = Vcc - Vc = Vcc

2. **Saturation Mode**

Ic > Ic.max

Ic.max = Vcc/Rc

Ic = β *Ib*

Ic = Ic.max

Vc = Ic x Rc = Vcc

Vo = Vcc - Vc = 0V

3. **Active Mode**

0 < Ic < Ic.max

Ib = (Vb - Vbe)/Rb

Ic = β *x* Ib

Vc = Ic x Rc

Vo = Vcc - Vc

0V < Vo < Vcc

When a Transistor acts as a Regulator, it is called an **Amplifier.**

When a Transistor acts as a Switch, it is called a **Gate**.

Transistor in Active Mode - Analog Electronics

Transistor in Cutoff/Saturation Mode - Digital Electronics

MIPS (originally an acronym for Microprocessor without Interlocked Pipeline Stages)

Double-precision floating-point format is a computer number format that occupies 8 bytes (64 bits) in computer memory and represents a wide, dynamic range of values by using a floating point.

DMIPS

Dhrystone Million instructions per second

Dhrystone is a synthetic computing benchmark program developed in 1984 by Reinhold P. Weicker intended to be representative of system (integer) programming. The Dhrystone grew to become representative of general processor (CPU) performance. The name "Dhrystone" is a pun on a different benchmark algorithm called Whetstone.

Difference between ARM and Thumb states

Thumb state is an added advantage in ARM to reduce the code size. Infact, ARM has single instruction set. Thumb is short hand 16-bit representation of a subset of those instructions. The processor fetches these 16-bit instructions and expands into 32-bit equivalent. So, ARM and Thumb differs only in how the instructions are fetched and interpreted, not in execution. So, speed is not at all changed. CPU has dedicated hardware within the chip to interpret the Thumb instructions.

ARM (Advanced RISC Machines)	Thumb ("T" in the core's full name specifies Thumb. Eg: ARM7TDMI)
Needs more memory than Thumb, since all are RISC 32-bit instructions	Reduced memory consumption with 16-bit insturctions (have some 32-bit instructions too)
Totally 18 Registers: R1-R12, SP(R13), LR(R14), PC(R15), CPSR, SPSR used in exceptions	Totally 12 Registers: R1-R7, SP, LR, PC, CPSR (These 12 registers have to be used to pass data between ARM and Thumb state)
T bit = 0 of CPSR represents ARM state	T bit = 1 of CPSR represents Thumb state
Branch instruction (BX or BLX) to the address with LSB set to 0 enters ARM state. (Eg: BX 0x80000000)	Branch instruction (BX or BLX) to the address with LSB set to 1 enters Thumb state. (Eg: BX 0x80000001)

When exception occurs, ARM state is entered.	-
When return from Exception, if T bit of SPSR is set to 0, returns to ARM	When return from Exception, if T bit of SPSR is set to 1, returns to Thumb
-	No way (instruction) to access status or coprocessor registers
Load and store instructions of R13 register manipulates stack	Has stack mnemonics PUSH, POP
-	Advantages: Reduced memory, 16-bit bus can be used without compromising with speed Disadvantages: As above said, No way (instruction) to access status or coprocessor registers. Some functions that can be accomplished in a single ARM instruction can only be simulated with a sequence of Thumb instructions.

Pipeline

Instruction pipelining is a technique that implements a form of parallelism called instruction-level parallelism within a single processor. It therefore allows faster CPU throughput (the number of instructions that can be executed in a unit of time) than would otherwise be possible at a given clock rate. The basic instruction cycle is broken up into a series called a pipeline. Rather than processing each instruction sequentially (finishing one instruction before starting the next), each instruction is split up into a sequence of steps so different steps can be executed in parallel and instructions can be processed concurrently (starting one instruction before finishing the previous one).

How caching works

CPU caches are small pools of memory that store information the CPU is most likely to need next. Which information is loaded into cache depends on sophisticated algorithms and certain assumptions about programming code. The goal of the cache system is to ensure that the CPU has the next

bit of data it will need already loaded into cache by the time it goes looking for it (also called a cache hit).

A cache miss, on the other hand, means the CPU has to go scampering off to find the data elsewhere. This is where the L2 cache comes into play — while it's slower, it's also much larger. Some processors use an inclusive cache design (meaning data stored in the L1 cache is also duplicated in the L2 cache) while others are exclusive (meaning the two caches never share data). If data can't be found in the L2 cache, the CPU continues down the chain to L3 (typically still on-die), then L4 (if it exists) and main memory (DRAM).

SDRAM (synchronous DRAM)

SDRAM (synchronous DRAM) is a generic name for various kinds of dynamic random access memory (DRAM) that are synchronized with the clock speed that the microprocessor is optimized for. This tends to increase the number of instructions that the processor can perform in a given time. The speed of SDRAM is rated in MHz rather than in nanoseconds (ns). This makes it easier to compare the bus speed and the RAM chip speed. You can convert the RAM clock speed to nanoseconds by dividing the chip speed into 1 billion ns (which is one second). For example, an 83 MHz RAM would be equivalent to 12 ns.

How does free work

When you malloc a block, it actually allocates a bit more memory than you asked for. This extra memory is used to store information such as the size of the allocated block, and a link to the next free/used block in a chain of blocks, and sometimes some "guard data" that helps the system to detect if you write past the end of your allocated block. Also, most allocators will round up the total size and/or the start of your part of the memory to a multiple of bytes (e.g. on a 64-bit system it may align the data to a multiple of 64 bits (8 bytes) as accessing data from non-aligned addresses can be more difficult and inefficient for the processor/bus), so you may also end up with some "padding" (unused bytes).

When you free your pointer, it uses that address to find the special information it added to the beginning (usually) of your allocated block. If you pass in a different address, it will

access memory that contains garbage, and hence its behaviour is undefined (but most frequently will result in a crash)

Later, if you free() the block but don't "forget" your pointer, you may accidentally try to access data through that pointer in the future, and the behaviour is undefined. Any of the following situations might occur:

- the memory might be put in a list of free blocks, so when you access it, it still happens to contain the data you left there, and your code runs normally.
- the memory allocator may have given (part of) the memory to another part of your program, and that will presumably have then overwritten (some of) your old data, so when you read it, you'll get garbage which might cause unexpected behaviour or crashes from your code. Or you will write over the other data, causing the other part of your program to behave strangely at some point in the future.
- the memory could have been returned to the operating system (a "page" of memory that you're no longer using can be removed from your address space, so there is no longer any memory available at that address - essentially an unused "hole" in your application's memory). When your application tries to access the data a hard memory fault will occur and kill your process.

This is why it is important to make sure you don't use a pointer after freeing the memory it points at - the best practice for this is to set the pointer to NULL after freeing the memory, because you can easily test for NULL, and attempting to access memory via a NULL pointer will cause a bad but **consistent** behaviour, which is much easier to debug.

Setting a bit

Use the bitwise OR operator (`|`) to set a bit.

```
number |= 1UL << n;
```

Clearing a bit

Use the bitwise AND operator (`&`) to clear a bit.

```
number &= ~(1UL << n);
```

Toggling a bit

The XOR operator (^) can be used to toggle a bit.

```
number ^= 1UL << n;
```

Checking a bit

You didn't ask for this, but I might as well add it.

To check a bit, shift the number n to the right, then bitwise AND it:

```
bit = (number >> n) & 1U;
```

What is the difference between a mutex and a semaphore?

There are many issues unique to multithreaded programming, the most prominent being *atomicity* and *ordering*. Whats important to realize is that these are completely unrelated to each other.

Atomicity:

An illusion that a section of code either executes completely, or doesn't execute at all. This illusion can be provided by allowing only one thread to execute that code at a time. Atomicity is a key requirement for generating consistent results with respect to a memory location. It is particularly useful for the parts of your program that modify shared state, say globals.

Ordering:

Different threads might be running on different cores. However, since there is no such thing as a global clock, at times its imperative to achieve ordering of instructions across various cores, for correctness. For example, it might be a correctness requirement to *execute instruction X of thread T1 running on core C1 before instruction Y of thread T2 running on core C2*.

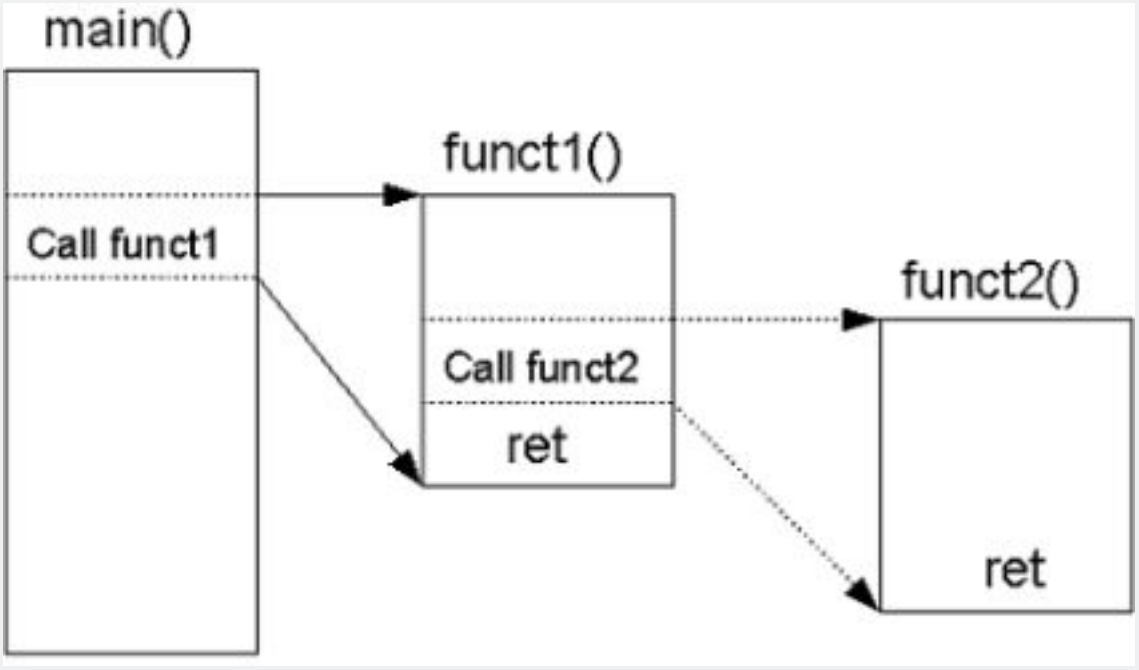
A mutex is used to meet the atomicity requirement. However, it does not impose any ordering. In other words, **given two threads, use of a mutex can't specify, which thread will acquire the mutex first**, and hence execute the critical section before the other. The only assurance is that if one thread is executing the critical section, the other will be kept out of it.

On the other hand, a semaphore can be used to impose ordering constraints in your execution. Considering the aforementioned example, you can block thread T2 just before it executes instruction Y, conditioned on whether T1 is done executing instruction X. This can be done by making T2 wait on the same condition variable that T1 signals, precisely the programming abstraction that the semaphore provides through its wait() and signal() operations.

Thus, a mutex can only be used to maintain atomicity whereas a semaphore can be used for both ordering and atomicity.

Function calls and stacking of parameters in embedded systems

Usually when you write C programs for microcontrollers you use functions that can be called any time in main program or in another function. Compiler compiles these functions as subroutines. These functions can be called from main function or from other functions â€“ this is called nesting (nested subroutines).



If you see the listings of compiled program you will see that subroutines are called by using call or rcall keyword. The argument of this instruction is an subroutine address which will be executed on the next processor cycle. Call instruction also writes return address from function to the stack to continue program after returning from function.

The next instruction begins execution at the start of the subroutine and returning from function is done after ret instruction which also restores an address of program counter in hardware level. Stack is also used to store all arguments of function. it usually depends on compiler host it is done.

But in general compiler generates code which:

- *Push all arguments to stack;*
- *Call the function;*
- *Allocate storage for all local variables in the stack;*
- *Perform the function;*
- *Deallocate local variables from stack;*
- *Return from function;*
- *Deallocate the space used by arguments.*

All these operations may depend on compiler and platform used. For instance AVR-GCC compiler is using registers for arguments and return values and if there are too many (over 9) of them then other are passed to stack. Read the FAQ. But it is

recommended not to use too many parameters in functions if you want to use code efficiently.

Types of constant

Pointer to constant.

*const int *ptr; or int const *ptr;*

Constant pointer to variable.

```
int *const ptr;
```

constant pointer to constant

```
const int *const ptr;
```

```
int const xxx;
 |    |    |
 |    |    +-----------------> xxx is a
 |    +----------------------> constant
 +---------------------------> integer
int const *ptr;
 |     |  | |
 |     |  | +----------------> ptr is a
 |     |  +------------------> pointer to a
 |     +---------------------> constant
 +---------------------------> integer
int * const ptr;
 |   | |     |
 |   | |     +---------------> ptr is a
 |   | +-----------------------> constant
 |   +-----------------------> pointer to an
 +---------------------------> integer
int const * const ptr;
 |    |   |   |    |
 |    |   |   |    +---------> ptr is a
 |    |   |   +--------------> constant
 |    |   +------------------> pointer to a
 |    +----------------------> constant
 +---------------------------> integer
int volatile * const ptr;
 |     |     |   |    |
 |     |     |   |    +------> ptr is a
 |     |     |   +-----------> constant
 |     |     +---------------> pointer to a
```

```
|     +---------------------> volatile
+--------------------------> integer
```

Can a variable be both const and volatile?

This is possible and mostly used in embedded system.The example is Interrupt Status Register as it is a status register , in the program we should not modify this Variable so it should be a constant.But this variable can be changed by the processor or hardware based on the interrupt condition. So when in the program ,we want to read the value of this variable , it should read the actual value without any optimisation.For this reason ,the variable can be declared as volatile too.

E.g.
Mapping of the register address with the pointer.

```
uint32_t *pStatusReg = (uint32_t *) 0x00020000;
```

Using the **pointer** (pStatusReg), we can read the value of the status register but problem is that pointed address is not qualified by the const so that program can change the value of the status register. It creates the problem because you will not get the proper value from the status register.

So to avoid this problem we have to qualify the address with const keyword.

```
uint32_t const *pStatusReg = (uint32_t *) 0x00020000;
```

Now const qualifier makes the value of status register nonmodifiable but still, there is one more problem. It can be possible the address that is pointing by the pointers can be changed because the pointer is not constant type, so we have to qualify the pointer with const qualifier.

```
uint32_t const * const pStatusReg = (uint32_t *) 0x00020000;
```

In the above expression pointer and address both qualified by **const qualifier**, so might be compiler put the value of the status register to a cache memory for the fast processing because they have found that program is not able to modify the value of the status register and the value of the status register will be the same in every read.

So here volatile can resolve your problems because when the data is marked as volatile the compiler won't make such an assumption (because the object can be changed at any time without any action being taken by the code) and so it will reread the value from the status register.

```
uint32_t const volatile * const pStatusReg = (uint32_t *) 0x00020000;

Register_Status = *pStatusReg ; // Read value from the Status Register
```