Solutions 7

Blue: Don't mark **Black: mark**

Question 1

If we change the contents of the register from a number which is larger than the current count value to a number which is smaller than the current count value, the counter will have to count all the way to its maximum value and wrap around around before the numbers match. This will introduce an unexpectedly long delay between update events.
By buffering we ensure that the new value is only set when the count value is at 0, hence ensuring that a match will occur before having to wrap around.

Question 1 (again...) {{7}}

a) C hides many of the architectural details and prevents the programmer having to know the assembly language. This is good as it prevents the programmer having to spend time becoming familiar with these details. {{2}}

b) Code written for one processor can be recompiled for a completely different processor with minimal or no changes to the code. This allows code reuse which saves the programmer time. {{2}}

c) Takes C code and converts it to assembly instructions which carry out the C code. {{1}}

d) They allow read/write access to arbitrary memory addresses. {{1}}

e) We need to be able to access arbitrary memory addresses in order to interface with our peripherals. {{1}}

Question 2 DON'T MARK

a) Assembly is better suited to some tasks such as defining the vector table. Also, it is necessary to initialise the memory to a state which the C code expects.

b) The C code expects the static variables (global variables) to have been initialised to their reset values.

c) Copying the initialisation values from flash to the memory addresses of the static variables (aka: the data segment)

d) This gives our data explicit and guaranteed sizes.

Question 3 {{7}}

a) Treat (typecast) the number as a pointer to an unsigned 16-bit number. {{1}}
b) Dereference. Access the data pointed to by the pointer. {{1}}
c) Or-equals. Set the data to the current data OR'd with the number 0xAABB {{1}}
d) 0x40001010: 0xBB 0x40001011: 0xAA {{4 x 1 = 4}}

Question 4 {{2}}
*(uint32_t *)0x48001000 = *(uint32_t *)0x20000010;

Question 5 DON'T MARK
a) Variables whose lifetime is the entire extent of the program and have a defined initial value. Often global variables. Allocated at the start of RAM in the data segment

b) Variables defined inside a function whose lifetime is the lifetime of the function. Allocated on the stack.

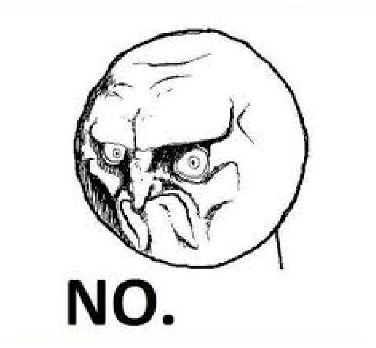
c) By using variables which are deallocated when no longer needed we are able to make better (more efficient) use of available RAM.

d) As no automatic initialisation is done for automatic variables they will take on whatever garbage happened to be in the memory addresses which are allocated to them.
Static variables are initialised to 0 or some custom value by the startup code before the C code is run.

Bonus: {{3}}
Data is the segment which holds static variables which have had a non-zero initialisation value specified.
BSS is the segment which contains static variables which have either explicitly been initialised to 0 or have not had an initialisation value specified meaning they default to 0.

By putting all variables which must be initialised to 0 in one place (the BSS segment), and knowing that all data in BSS must be initialised to 0 means that we simply have to store the start and end addresses of this block and can initialise everything between those addresses to 0, rather than having to store the individual initialisation values for all of the variables in flash memory to be copied to RAM.
(damn, that's a long sentence… Somebody neaten that up please...)

Tighe:



Tighe Barris, master of the pictorial response to any situation. - TJM