Embedded C interview Questions for Embedded Systems Engineers

Q: Which is the best way to write Loops?
Q: Is Count Down_to_Zero Loop better than Count_Up_Loops?
A: Always, count Down to Zero loops are better.
This is because,at loop termination, comparison to Zero can be optimized by complier. (Eg using SUBS)
Else, At the end of the loop there is ADD and CMP.

Q: What is loop unrolling?
A: Small loops can be unrolled for higher performance, with the disadvantage of increased codesize. When a loop is unrolled, a loop counter needs to be updated less often and fewer branches are executed. If the loop iterates only a few times, it can be fully unrolled, so that the loop overhead completely disappears.
eg:

```
int countbit1(uint n)
{ int bits = 0;
while (n != 0)
{
if (n & 1) bits++;
n &gt;&gt;= 1;
}
return bits;
}

int countbit2(uint n)
{ int bits = 0;
while (n != 0)
{
if (n & 1) bits++;
if (n & 2) bits++;
if (n & 4) bits++;
if (n & 8) bits++;
n &gt;&gt;= 4;
}
return bits;
}
```

Q: How does, taking the address of local variable result in unoptimized code?
A: The most powerful optimization for compliler is register allocation. That is it operates the variable from register, than memory. Generally local variables are allocated in registers. However if we take the address of a local variable, compiler will not allocate the variable to register.

Q: How does global variables result in unoptimized code?
A: For the same reason as above, compiler will never put the global variable into register. So its bad.

Q: So how to overcome this problem?
A: When it is necessary to take the address of variables, (for example if they are passed as a reference parameter to a function). Make a copy of the variable, and pass the address of that copy.

Q: Which is better a char, short or int type for optimization?
A: Where possible, it is best to avoid using char and short as local variables. For the types char and short the compiler needs to reduce the size of the local variable to 8 or 16 bits after each assignment. This is called sign-extending for signed variables and zeroextending for unsigned variables. It is implemented by shifting the register left by 24 or 16 bits, followed by a signed or unsigned shift right by the same amount, taking two instructions (zero-extension of an unsigned char takes one instruction).
These shifts can be avoided by using int and unsigned int for local variables. This is particularly important for calculations which first load data into local variables and then process the data inside the local variables. Even if data is input and output as 8- or 16-bit quantities, it is worth considering processing them as 32-bit quantities.

Q: How to reduce function call overhead in ARM based systems?
A:
. Try to ensure that small functions take four or fewer arguments. These will not use the stack for argument passing. It will copied into registers.
· If a function needs more than four arguments, try to ensure that it does a significant amount of work, so that the cost of passing the stacked arguments is outweighed.
· Pass pointers to structures instead of passing the structure itself.
· Put related arguments in a structure, and pass a pointer to the structure to functions. This will reduce the number of parameters and increase readability.
· Minimize the number of long long parameters, as these take two argument words.
This also applies to doubles if software floating-point is enabled.
· Avoid functions with a parameter that is passed partially in a register and partially on the stack (split-argument). This is not handled efficiently by the current compilers: all register arguments are pushed on the stack.
· Avoid functions with a variable number of parameters. Varargs functions

Q: What is a pure function in ARM terminology?
A: Pure functions are those which return a result which depends only on their arguments.
They can be thought of as mathematical functions: they always return the same result if the arguments are the same. To tell the compiler that a function is pure, use the special declaration keyword __pure.

```
__pure int square(int x)
{ return x * x;
}
```

Compiler does optimization for pure functions. For example, the values which are allocated to memory can be safely cached in registers, instead of being written to memory before a call and reloaded afterwards.

Q: What are inline functions?
A: The ARM compilers support inline functions with the keyword __inline. This results in each call to an inline function being substituted by its body, instead of a normal call. This results in faster code, but it adversely affects code size, particularly if the inline function is large and used often.