A 'C' Test: The 0x10 Best Questions for Would-be Embedded Programmers

Nigel Jones

Pencils up, everyone. Here's a test to identify potential embedded programmers or embedded programmers with potential

An obligatory and significant part of the recruitment process for embedded systems programmers seems to be the "C test." Over the years, I have had to both take and prepare such tests and, in doing so, have realized that these tests can be informative for both the interviewer and interviewee. Furthermore, when given outside the pressure of an interview situation, these tests can also be quite entertaining.

From the interviewee's perspective, you can learn a lot about the person who has written or administered the test. Is the test designed to show off the writer's knowledge of the minutiae of the ANSI standard rather than to test practical know-how? Does it test ludicrous knowledge, such as the ASCII values of certain characters? Are the questions heavily slanted towards your knowledge of system calls and memory allocation strategies, indicating that the writer may spend his time programming computers instead of embedded systems? If any of these are true, then I know I would seriously doubt whether I want the job in question.

From the interviewer's perspective, a test can reveal several things about the candidate. Primarily, you can determine the level of the candidate's knowledge of C. However, it's also interesting to see how the person responds to questions to which they don't know the answers. Do they make intelligent choices backed up with good intuition, or do they just guess? Are they defensive when they are stumped, or do they exhibit a real curiosity about the problem and see it as an opportunity to learn something? I find this information as useful as their raw performance on the test.

With these ideas in mind, I have attempted to construct a test that is heavily slanted towards the requirements of embedded systems. This is a lousy test to give to someone seeking a job writing compilers! The questions are almost all drawn from situations I have encountered over the years. Some of them are tough; however, they should all be informative.

This test may be given to a wide range of candidates. Most entry-level applicants will do poorly on this test, while seasoned veterans should do very well. Points are not assigned to each question, as this tends to arbitrarily weight certain questions. However, if you choose to adapt this test for your own uses, feel free to assign scores.

Preprocessor

1. *Using the #define statement, how would you declare a manifest constant that returns the number of seconds in a year? Disregard leap years in your answer.*

```
#define SECONDS_PER_YEAR
  (60 * 60 * 24 * 365)UL
```

I'm looking for several things here:

Basic knowledge of the #define syntax (for example, no semi-colon at the end, the need to parenthesize, and so on)

An understanding that the pre-processor will evaluate constant expressions for you. Thus, it is clearer, and penalty-free, to spell out how you are calculating the number of seconds in a year, rather than actually doing the calculation yourself

A realization that the expression will overflow an integer argument on a 16-bit machine-hence the need for the L, telling the compiler to treat the variable as a Long As a bonus, if you modified the expression with a UL (indicating unsigned long), then you are off to a great start. And remember, first impressions count!

2. *Write the "standard" MIN macro-that is, a macro that takes two arguments and returns the smaller of the two arguments.*

```
#define MIN(A,B)
  ((A) <=  (B) ? (A) : (B))
```

The purpose of this question is to test the following:

Basic knowledge of the #define directive as used in macros. This is important because until the inline operator becomes part of standard C, macros are the only portable way of generating inline code. Inline code is often necessary in embedded systems in order to achieve the required performance level

Knowledge of the ternary conditional operator. This operator exists in C because it allows the compiler to produce more optimal code than an if-then-else sequence. Given that performance is normally an issue in embedded systems, knowledge and use of this construct is important

Understanding of the need to very carefully parenthesize arguments to macros I also use this question to start a discussion on the side effects of macros, for example, what happens when you write code such as:

```
least = MIN(*p++, b);
```

3. *What is the purpose of the preprocessor directive #error?*

Either you know the answer to this, or you don't. If you don't, see Reference 1. This question is useful for differentiating between normal folks and the nerds. Only the nerds actually read the appendices of C textbooks to find out about such things. Of course, if you aren't looking for a nerd, the candidate better hope she doesn't know the answer.

Infinite loops

4. *Infinite loops often arise in embedded systems. How does you code an infinite loop in C?*

There are several solutions to this question. My preferred solution is:

```
while(1)
{
É
}
```

Many programmers seem to prefer:

```
for(;;)
{
É
}
```

This construct puzzles me because the syntax doesn't exactly spell out what's going on. Thus, if a candidate gives this as a solution, I'll use it as an opportunity to explore their rationale for doing so. If their answer is basically, "I was taught to do it this way and I haven't thought about it since," it tells me something (bad) about them.

A third solution is to use a **goto** :

```
Loop:
...
goto Loop;
```

Candidates who propose this are either assembly language programmers (which is probably good), or else they are closet BASIC/FORTRAN programmers looking to get into a new field.

Data declarations

5. *Using the variable a, give definitions for the following:*

a) An integer
b) A pointer to an integer
c) A pointer to a pointer to an integer
d) An array of 10 integers
e) An array of 10 pointers to integers
f) A pointer to an array of 10 integers
g) A pointer to a function that takes an integer as an argument and returns an integer
h) An array of ten pointers to functions that take an integer argument and return an integer

The answers are:

a) **int a;** // An integer
b) **int *a;** // A pointer to an integer
c) **int **a;** // A pointer to a pointer to an integer
d) **int a[10];** // An array of 10 integers
e) **int *a[10];** // An array of 10 pointers to integers
f) **int (*a)[10];** // A pointer to an array of 10 integers
g) **int (*a)(int);** // A pointer to a function a that takes an integer argument and returns an integer
h) **int (*a[10])(int);** // An array of 10 pointers to functions that take an integer argument and return an integer

People often claim that a couple of these are the sorts of thing that one looks up in textbooks-and I agree. While writing this article, I consulted textbooks to ensure the syntax was correct. However, I expect to be asked this question (or something close to it) when I'm being interviewed. Consequently, I make sure I know the answers, at least for the few hours of the interview. Candidates who don't know all the answers (or at least most of them) are simply unprepared for the interview. If they can't be prepared for the interview, what will they be prepared for?

Static

6. *What are the uses of the keyword static?*

This simple question is rarely answered completely. Static has three distinct uses in C:

A variable declared static within the body of a function maintains its value between function invocations

A variable declared static within a module, (but outside the body of a function) is accessible by all functions within that module. It is not accessible by functions within any other module. That is, it is a localized global

Functions declared static within a module may only be called by other functions within that module. That is, the scope of the function is localized to the module within which it is declared

Most candidates get the first part correct. A reasonable number get the second part correct, while a pitiful number understand the third answer. This is a serious weakness in a candidate, since he obviously doesn't understand the importance and benefits of localizing the scope of both data and code.

Const

7. *What does the keyword const mean?*

As soon as the interviewee says "const means constant," I know I'm dealing with an amateur. Dan Saks has exhaustively covered **const** in the last year, such that every reader of ESP should be extremely familiar with what const can and cannot do for you. If you haven't been reading that column, suffice it to say that **const** means "read-only." Although this answer doesn't really do the subject justice, I'd accept it as a correct answer. (If you want the detailed answer, read Saks' columns-carefully!)

If the candidate gets the answer correct, I'll ask him these supplemental questions:

What do the following declarations mean?

const int a;
int const a;
const int *a;
int * const a;
int const * a const;

The first two mean the same thing, namely **a** is a const (read-only) integer. The third means **a** is a pointer to a const integer (that is, the integer isn't modifiable, but the pointer is). The fourth declares **a** to be a const pointer to an integer (that is, the integer pointed to by a is modifiable, but the pointer is not). The final declaration declares **a** to be a const pointer to a const integer (that is, neither the integer pointed to by a, nor the pointer itself may be modified). If the candidate correctly answers these questions, I'll be impressed. Incidentally, you might wonder why I put so much emphasis on const, since it is easy to write a correctly functioning program without ever using it. I have several reasons:

The use of **const** conveys some very useful information to someone reading your code. In effect, declaring a parameter **const** tells the user about its intended usage. If you spend a lot of time cleaning up the mess left by other people, you'll quickly learn to appreciate this extra piece of information. (Of course, programmers who use **const** , rarely leave a mess for others to clean up.)

const has the potential for generating tighter code by giving the optimizer some additional information

Code that uses **const** liberally is inherently protected by the compiler against inadvertent coding constructs that result in parameters being changed that should not be. In short, they tend to have fewer bugs

Volatile

8. *What does the keyword volatile mean? Give three different examples of its use.*

A **volatile** variable is one that can change unexpectedly. Consequently, the compiler can make no assumptions about the value of the variable. In particular, the optimizer must be careful to reload the variable every time it is used instead of holding a copy in a register.

Examples of **volatile** variables are:

Hardware registers in peripherals (for example, status registers)
Non-automatic variables referenced within an interrupt service routine
Variables shared by multiple tasks in a multi-threaded application

Candidates who don't know the answer to this question aren't hired. I consider this the most fundamental question that distinguishes between a C programmer and an embedded systems programmer. Embedded folks deal with hardware, interrupts, RTOSes, and the like. All of these require volatile variables. Failure to understand the concept of **volatile** will lead to disaster.

On the (dubious) assumption that the interviewee gets this question correct, I like to probe a little deeper to see if they really understand the full significance of **volatile** . In particular, I'll ask them the following additional questions:

Can a parameter be both const and **volatile** ? Explain.
Can a pointer be **volatile** ? Explain.
What's wrong with the following function?:

```
int square(volatile int *ptr)
{
        return *ptr * *ptr;
}
```

The answers are as follows:

Yes. An example is a read-only status register. It is volatile because it can change unexpectedly. It is const because the program should not attempt to modify it

Yes, although this is not very common. An example is when an interrupt service routine modifies a pointer to a buffer

This one is wicked. The intent of the code is to return the square of the value pointed to by ***ptr** . However, since ***ptr** points to a volatile parameter, the compiler will generate code that looks something like this:

```
int square(volatile int *ptr)
{
  int a,b;
  a = *ptr;
  b = *ptr;
  return a * b;
}
```

Because it's possible for the value of ***ptr** to change unexpectedly, it is possible for **a** and **b** to be different. Consequently, this code could return a number that is not a square! The correct way to code this is:

```
long square(volatile int *ptr)
{
  int a;
  a = *ptr;
  return a * a;
}
```

Bit manipulation

9. *Embedded systems always require the user to manipulate bits in registers or variables. Given an integer variable a, write two code fragments. The first should set bit 3 of a. The second should clear bit 3 of a. In both cases, the remaining bits should be unmodified.*

These are the three basic responses to this question:

No idea. The interviewee cannot have done any embedded systems work

Use bit fields. Bit fields are right up there with trigraphs as the most brain-dead portion of C. Bit fields are inherently non-portable across compilers, and as such guarantee that your code is not reusable. I recently had the misfortune to look at a driver written by Infineon for one of their more complex communications chips. It used bit fields and was completely useless because my compiler implemented the bit fields the other way around. The moral: never let a non-embedded person anywhere near a real piece of hardware!

Use **#defines** and bit masks. This is a highly portable method and is the one that should be used. My optimal solution to this problem would be:

```
#define BIT3  (0x1 << 3)
static int a;


void set_bit3(void) {
        a |= BIT3;
}
void clear_bit3(void) {
        a &= ~BIT3;
}
```

Some people prefer to define a mask together with manifest constants for the set and clear values. This is also acceptable. The element that I'm looking for is the use of manifest constants, together with the |= and &= ~ constructs

Accessing fixed memory locations

10. *Embedded systems are often characterized by requiring the programmer to access a specific memory location. On a certain project it is required to set an integer variable at the absolute address 0x67a9 to the value 0xaa55. The compiler is a pure ANSI compiler. Write code to accomplish this task.*

This problem tests whether you know that it is legal to typecast an integer to a pointer in order to access an absolute location. The exact syntax varies depending upon one's style. However, I would typically be looking for something like this:

```
int *ptr;
ptr = (int *)0x67a9;
*ptr = 0xaa55;
```

A more obscure approach is:

```
*(int * const)(0x67a9) = 0xaa55;
```

Even if your taste runs more to the second solution, I suggest the first solution when you are in an interview situation.

Interrupts

11. *Interrupts are an important part of embedded systems. Consequently, many compiler vendors offer an extension to standard C to support interrupts. Typically, this new keyword is __interrupt. The following code uses __interrupt to define an interrupt service routine (ISR). Comment on the code.*

```
__interrupt double compute_area
  (double
 radius)
{
  double area = PI * radius *
    radius;
   printf("\nArea = %f", area);
   return area;
}
```

This function has so much wrong with it, it's hard to know where to start:

ISRs cannot return a value. If you don't understand this, you aren't hired

ISRs cannot be passed parameters. See the first item for your employment prospects if you missed this

On many processors/compilers, floating-point operations are not necessarily re-entrant. In some cases one needs to stack additional registers. In other cases, one simply cannot do floating point in an ISR. Furthermore, given that a general rule of thumb is that ISRs should be short and sweet, one wonders about the wisdom of doing floating-point math here

In a vein similar to the third point, printf() often has problems with reentrancy and performance. If you missed points three and four, I wouldn't be too hard on you. Needless to say, if you got these last two points, your employment prospects are looking better and better

Code examples

12. *What does the following code output and why?*

```
void foo(void)
{
  unsigned int a = 6;
  int b = -20;
    (a+b > 6) ? puts("> 6") : puts("<= 6");
}
```

This question tests whether you understand the integer promotion rules in C-an area that I find is very poorly understood by many developers. Anyway, the answer is that this outputs "> 6." The reason for this is that expressions involving signed and unsigned types have all operands promoted to unsigned types. Thus Ð20 becomes a very large positive integer and the expression evaluates to greater than 6. This is a very important point in embedded systems where unsigned data types should be used frequently (see Reference 2). If you get this one wrong, you are perilously close to not getting the job.

13. *Comment on the following code fragment.*

```
unsigned int zero = 0;
unsigned int compzero = 0xFFFF;
/*1's complement of zero */
```

On machines where an int is not 16 bits, this will be incorrect. It should be coded:

```
unsigned int compzero = ~0;
```

This question really gets to whether the candidate understands the importance of word length on a computer. In my experience, good embedded programmers are critically aware of the underlying hardware and its limitations, whereas computer programmers tend to dismiss the hardware as a necessary annoyance.

By this stage, candidates are either completely demoralized-or they're on a roll and having a good time. If it's obvious that the candidate isn't very good, then the test is terminated at this point. However, if the candidate is doing well, then I throw in these supplemental questions. These questions are hard, and I expect that only the very best candidates will do well on them. In posing these questions, I'm looking more at the way the candidate tackles the problems, rather than the answers. Anyway, have fun...

Dynamic memory allocation

14. *Although not as common as in non-embedded computers, embedded systems do still dynamically allocate memory from the heap. What are the problems with dynamic memory allocation in embedded systems?*

Here, I expect the user to mention memory fragmentation, problems with garbage collection, variable execution time, and so on. This topic has been covered extensively in *ESP* , mainly by P.J. Plauger. His explanations are far more insightful than anything I could offer here, so go and read those back issues! Having lulled the candidate into a sense of false security, I then offer up this tidbit:

What does the following code fragment output and why?

```
char *ptr;
if ((ptr = (char *)malloc(0)) ==
   NULL)
else
  puts("Got a null pointer");
  puts("Got a valid pointer");
```

This is a fun question. I stumbled across this only recently when a colleague of mine inadvertently passed a value of 0 to **malloc** and got back a valid pointer! That is, the

above code will output "Got a valid pointer." I use this to start a discussion on whether the interviewee thinks this is the correct thing for the library routine to do. Getting the right answer here is not nearly as important as the way you approach the problem and the rationale for your decision.

Typedef

15. **Typedef** *is frequently used in C to declare synonyms for pre-existing data types. It is also possible to use the preprocessor to do something similar. For instance, consider the following code fragment:*

```
#define dPS  struct s *
typedef  struct s * tPS;
```

The intent in both cases is to define **dPS** *and* **tPS** *to be pointers to structure s. Which method, if any, is preferred and why?*

This is a very subtle question, and anyone who gets it right (for the right reason) is to be congratulated or condemned ("get a life" springs to mind). The answer is the **typedef** is preferred. Consider the declarations:

```
dPS p1,p2;
tPS p3,p4;
```

The first expands to:

```
struct s * p1, p2;
```

which defines **p1** to be a pointer to the structure and **p2** to be an actual structure, which is probably not what you wanted. The second example correctly defines **p3** and **p4** to be pointers.

Obscure syntax

16. *C allows some appalling constructs. Is this construct legal, and if so what does this code do?*

```
int a = 5, b = 7, c;
c = a+++b;
```

This question is intended to be a lighthearted end to the quiz, as, believe it or not, this is perfectly legal syntax. The question is how does the compiler treat it? Those poor compiler writers actually debated this issue, and came up with the "maximum munch" rule, which stipulates that the compiler should bite off as big (and legal) a chunk as it can. Hence, this code is treated as:

c = a++ + b;

Thus, after this code is executed, a = 6, b = 7, and c = 12.
If you knew the answer, or guessed correctly, well done. If you didn't know the answer then I wouldn't consider this to be a problem. I find the greatest benefit of this question is that it is good for stimulating questions on coding styles, the value of code reviews, and the benefits of using lint.

Well folks, there you have it. That was my version of the C test. I hope you had as much fun taking it as I had writing it. If you think the test is a good test, then by all means use it in your recruitment. Who knows, I may get lucky in a year or two and end up being on the receiving end of my own work.

Nigel Jones is a consultant living in Maryland. When not underwater, he can be found slaving away on a diverse range of embedded projects. He enjoys hearing from readers and can be reached at NAJones@compuserve.com .