Embedded system Technical interview questions-Embedded C

1) What is the use of volatile keyword?

The C's volatile keyword is a qualifier that tells the compiler not to optimize when applied to a variable. By declaring a variable volatile, we can tell the compiler that the value of the variable may change any moment from outside of the scope of the program. A variable should be declared volatile whenever its value could change unexpectedly and beyond the comprehension of the compiler.
In those cases it is required not to optimize the code, doing so may lead to erroneous result and load the variable every time it is used in the program. Volatile keyword is useful for memory-mapped peripheral registers, global variables modified by interrupt service routine, global variables accessed by multiple tasks within a multi-threaded application.

2) Can a variable be both const and volatile?

The const keyword make sure that the value of the variable declared as const can't be changed. This statement holds true in the scope of the program. The value can still be changed by outside intervention. So, the use of const with volatile keyword makes perfect sense.

3) Can a pointer be volatile?

If we see the declaration volatile int *p, it means that the pointer itself is not volatile and points to an integer that is volatile. This is to inform the compiler that pointer p is pointing to an integer and the value of that integer may change unexpectedly even if there is no code indicating so in the program.

4) What is size of character, integer, integer pointer, character pointer?

- The sizeof character is 1 byte.
- Size of integer is 4 bytes.
- Size of integer pointer and character is 8 bytes on 64 bit machine and 4 bytes on 32 bit machine.

5) What is NULL pointer and what is its use?

The NULL is a macro defined in C. Null pointer actually means a pointer that does not point to any valid location. We define a pointer to be null when we want to make sure that the pointer does not point to any valid location and not to use that pointer to change anything. If we don't use null pointer, then we can't verify whether this pointer points to any valid location or not.

6) What is void pointer and what is its use?

The void pointer means that it points to a variable that can be of any type. Other pointers points to a specific type of variable while void pointer is a somewhat generic pointer and can be pointed to any data type, be it standard data type(int, char etc) or user define data type (structure, union etc.). We can pass any kind of pointer and reference it as a void pointer. But to dereference it, we have to type the void pointer to correct data type.

7) What is ISR?

An ISR(Interrupt Service Routine) is an interrupt handler, a callback subroutine which is called when a interrupt is encountered.

8) What is return type of ISR?

ISR does not return anything. An ISR returns nothing because there is no caller in the code to read the returned values.

9) What is interrupt latency?

Interrupt latency is the time required for an ISR responds to an interrupt.

10) How to reduce interrupt latency?

Interrupt latency can be minimized by writing short ISR routine and by not delaying interrupts for more time.

11) Can we use any function inside ISR?

We can use function inside ISR as long as that function is not invoked from other portion of the code.

12) Can we use printf inside ISR?

Printf function in ISR is not supported because printf function is not reentrant, thread safe and uses dynamic memory allocation which takes a lot of time and can affect the speed of an ISR up to a great extent.

13) Can we put breakpoint inside ISR?

Putting a break point inside ISR is not a good idea because debugging will take some time and a difference of half or more second will lead to different behavior of hardware. To debug ISR, definitive logs are better.

14) Can static variables be declared in a header file?

A static variable cannot be declared without defining it. A static variable can be defined in the header file. But doing so, the result will be having a private copy of that variable in each source file which includes the header file. So it will be wise not to declare a static variable in header file, unless you are dealing with a different scenario.

15) Is Count Down_to_Zero Loop better than Count_Up_Loops?

Count down to zero loops are better. Reason behind this is that at loop termination, comparison to zero can be optimized by the compiler. Most processors have instruction for comparing to zero. So they don't need to load the loop variable and the maximum value, subtract them and then compare to zero. That is why count down to zero loop is better.

16) What are inline functions?

The ARM compilers support inline functions with the keyword __inline. These functions have a small definition and the function body is substituted in each call to the inline function. The argument passing and stack maintenance is skipped and it results in faster code execution, but it increases code size, particularly if the inline function is large or one inline function is used often.

17) Can include files be nested?

Yes. Include files can be nested any number of times. But you have to make sure that you are not including the same file twice. There is no limit to how many header files that can be included. But the number can be compiler dependent, since including multiple header files may cause your computer to run out of stack memory.

18) What are the uses of the keyword static?

Static keyword can be used with variables as well as functions. A variable declared static will be of static storage class and within a function, it maintains its value between calls to that function. A variable declared as static within a file, scope of that variable will be within that file, but it can't be accessed by other files.

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

19) What are the uses of the keyword volatile?

Volatile keyword is used to prevent compiler to optimize a variable which can change unexpectedly beyond compiler's comprehension. Suppose, we have a variable which may be changed from scope out of the program, say by a signal, we do not want the compiler to optimize it. Rather than optimizing that variable, we want the compiler to load the variable every time it is encountered. If we declare a variable volatile, compiler will not cache it in its register.

20) What is Top half & bottom half of a kernel?

Sometimes to handle an interrupt, a substantial amount of work has to be done. But it conflicts with the speed need for an interrupt handler. To handle this situation, Linux splits the handler into two parts –*Top half and Bottom half*. The top half is the routine that actually responds to the interrupt. The bottom half on the other hand is a routine that is scheduled by the upper half to be executed later at a safer time.

All interrupts are enabled during execution of the bottom half. The top half saves the device data into the specific buffer, schedules bottom half and exits. The bottom half does the rest. This way the top half can service a new interrupt while the bottom half is working on the previous.

---

21) Difference between RISC and CISC processor.

RISC (Reduced Instruction Set Computer) could carry out a few sets of simple instructions simultaneously. Fewer transistors are used to manufacture RISC, which makes RISC cheaper. RISC has uniform instruction set and those instructions are also fewer in number. Due to the less number of instructions as well as instructions being simple, the RISC computers are faster. RISC emphasise on software rather than hardware. RISC can execute instructions in one machine cycle.

CISC (Complex Instruction Set Computer) is capable of executing multiple operations through a single instruction. CISC have rich and complex instruction set and more number of addressing modes. CISCemphasise on hardware rather that software, making it costlier than RISC. It has a small code size, high cycles per second and it is slower compared to RISC.

22) What is RTOS?

In an operating system, there is a module called the scheduler, which schedules different tasks and determines when a process will execute on the processor. This way, the multi-tasking is achieved. The scheduler in a Real Time Operating System (RTOS) is designed to provide a predictable execution pattern. In an embedded system, a certain event must be entertained in strictly defined time.

To meet real time requirements, the behaviour of the scheduler must be predictable. This type of OS which have a scheduler with predictable execution pattern is called Real Time OS(RTOS). The features of an RTOS are

- Context switching latency should be short.
- Interrupt latency should be short.
- Interrupt dispatch latency should be short.
- Reliable and time bound inter process mechanisms.
- Should support kernel preemption.

23) What is the difference between hard real-time and soft real-time OS?

A Hard real-time system strictly adheres to the deadline associated with the task. If the system fails to meet the deadline, even once, the system is considered to have failed. In case of a soft real-time system, missing a deadline is acceptable. In this type of system, a critical real-time task gets priority over other tasks and retains that priority until it completes.

24) What type of scheduling is there in RTOS?

RTOS uses pre-emptive scheduling. In pre-emptive scheduling, the higher priority task can interrupt a running process and the interrupted process will be resumed later.

25) What is priority inversion?

If two tasks share a resource, the one with higher priority will run first. However, if the lower-priority task is using the shared resource when the higher-priority task becomes ready, then the higher-priority task must wait for the lower-priority task to finish. In this scenario, even though the task has higher priority it needs to wait for the completion of the lower-priority task with the shared resource. This is called priority inversion.

26) What is priority inheritance?

Priority inheritance is a solution to the priority inversion problem. The process waiting for any resource which has a resource lock will have the maximum priority. This is priority inheritance. When one or more high priority jobs are blocked by a job, the original priority assignment is ignored and execution of critical section will be assigned to the job with the highest priority in this elevated scenario. The job returns to the original priority level soon after executing the critical section.

27) How many types of IPC mechanism you know?

Different types of IPC mechanism are -

- Pipes
- Named pipes or FIFO
- Semaphores
- Shared memory
- Message queue
- Socket

28) What is semaphore?

Semaphore is actually a variable or abstract data type which controls access to a common resource by multiple processes. Semaphores are of two types -

- Binary semaphore – It can have only two values (0 and 1). The semaphore value is set to 1 by the process in charge, when the resource is available.
- Counting semaphore – It can have value greater than one. It is used to control access to a pool of resources.

29) What is spin lock?

If a resource is locked, a thread that wants to access that resource may repetitively check whether the resource is available. During that time, the thread may loop and check the resource without doing any useful work. Suck a lock is termed as spin lock.

30) What is difference between binary semaphore and mutex?

The differences between binary semaphore and mutex are as follows -

- Mutual exclusion and synchronization can be used by binary semaphore while mutex is used only for mutual exclusion.
- A mutex can be released by the same thread which acquired it. Semaphore values can be changed by other thread also.
- From an ISR, a mutex can not be used.
- The advantage of semaphores is that, they can be used to synchronize two unrelated processes trying to access the same resource.
- Semaphores can act as mutex, but the opposite is not possible.

31) What is virtual memory?

Virtual memory is a technique that allows processes to allocate memory in case of physical memory shortage using automatic storage allocation upon a request. The advantage of the virtual memory is that the program can have a larger memory than the physical memory. It allows large virtual memory to be provided when only a smaller physical memory is available. Virtual memory can be implemented using paging.

A paging system is quite similar to a paging system with swapping. When we want to execute a process, we swap it into memory. Here we use a lazy swapper called pager rather than swapping the entire process into memory. When a process is to be swapped in, the pager guesses which pages will be used based on some algorithm, before the process is swapped out again. Instead of

swapping whole process, the pager brings only the necessary pages into memory. By that way, it avoids reading in unnecessary memory pages, decreasing the swap time and the amount of physical memory.

32) What is kernel paging?

Paging is a memory management scheme by which computers can store and retrieve data from the secondary memory storage when needed in to primary memory. In this scheme, the operating system retrieves data from secondary storage in same-size blocks called pages. The paging scheme allows the physical address space of a process to be non continuous. Paging allows OS to use secondary storage for data that does not fit entirely into physical memory.

33) Can structures be passed to the functions by value?

Passing structure by its value to a function is possible, but not a good programming practice. First of all, if we pass the structure by value and the function changes some of those values, then the value change is not reflected in caller function. Also, if the structure is big, then passing the structure by value means copying the whole structure to the function argument stack which can slow the program by a significant amount.

34) Why cannot arrays be passed by values to functions?

In C, the array name itself represents the address of the first element. So, even if we pass the array name as argument, it will be passed as reference and not its address.

35) Advantages and disadvantages of using macro and inline functions?

The advantage of the macro and inline function is that the overhead for argument passing and stuff is reduced as the function are in-lined. The advantage of macro function is that we can write type insensitive functions. It is also the disadvantage of macro function as macro functions can't do validation check. The macro and inline function also increases the size of the executable.

36) What happens when recursive functions are declared inline?

Inlining an recursive function reduces the overhead of saving context on stack. But, inline is merely a suggestion to the compiler and it does not guarantee that a function will be inlined. Obviously, the compiler won't be able to inline a recursive function infinitely. It may not inline it at all or it may inline it, just a few levels deep.

37) #define cat(x,y) x##y concatenates x to y. But cat(cat(1,2),3) does not expand but gives preprocessor warning. Why?

The cat(x, y) expands to x##y. It just pastes x and y. But in case of cat(cat(1,2),3), it expands to cat(1,2)##3 instead of 1##2##3. That is why it is giving preprocessor warning.

38) ++*ip increments what?

It increments the value to which ip points to and not the address.

39) Declare a manifest constant that returns the number of seconds in a year using preprocessor? Disregard leap years in your answer.

The correct answer will be -

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

Do not forget to use UL, since the output will be very big integer.

40) Using the variable a, write down definitions for the following:

- An integer
- A pointer to an integer
- A pointer to a pointer to an integer
- An array of ten integers
- An array of ten pointers to integers
- A pointer to an array of ten integers
- A pointer to a function that takes an integer as an argument and returns an integer
- Pass an array of ten pointers to a function that take an integer argument and return an integer.

The correct answer is as follows -

- int a;
- int *a;
- int **a;
- int a[10];
- int *a[10];
- int (*a)[10];
- int (*a)(int);
- int (*a[10])(int);

If you have problem understanding these, please read pointer section in the provide tutorial thoroughly.

41) Consider the two statements below and point out which one is preferred and why?

```
#define B struct A *

typedef struct A * C;
```

The typedef is preferred. Both statements declare pointer to struct A to something else and in one glance both looks fine. But there is one issue with the define statement. Consider a situation where we want to declare p1 and p2 as pointer to struct A. We can do this by -

```
C p1, p2;
```

But doing - B p1, p2, it will be expanded to struct A * p1, p2. It means that p1 is a pointer to struct A but p2 is a variable of struct A and not a pointer.

42) What will be the output of the following code fragment?

```
char *ptr;

if ((ptr = (char *)malloc(0)) == NULL)

{

   puts("Got a null pointer");

}

else

{

   puts("Got a valid pointer");

}
```

The output will be “Got a valid pointer”. It is because malloc(0) returns a valid pointer, but it allocates size 0. So this pointer is of no use, but we can use this free pointer and the program will not crash.

43) What is purpose of keyword const?

The const keyword when used in c means that the value of the variable will not be changed. But the value of the variable can be changed using a pointer. The const identifier can be used like this -

```
const int a; or int const a;
```

Both means the same and it indicates that a is an constant integer. But if we declare something like this -

```
const int *p
```

then it does not mean that the pointer is constant but rather it is pointing to an constant integer. The declaration of an const pointer to a non-constant integer will look like this -

```
int * cont p;
```

44) 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 means that a is a constant integer.
- The third declaration means that a is a pointer to a constant integer.
- The fourth means that a is a constant pointer to a non-constant integer.
- The fifth means that a is a constant pointer to a constant integer.

45) How to decide whether given processor is using little endian format or big endian format ?

The following program can find out the endianness of the processor.

```
#include<stdio.h>

main ()

{

 union Test

 {

    unsigned int i;

    unsigned char c[2];

 };

 union Test a = {300};

 if((a.c [0] == 1) &&  (a.c [1] == 44))

 {

    printf ("BIG ENDIAN\n");

 }

 else

 {

    printf ("LITTLE ENDIAN\n");

 }

}
```

46) What is the concatenation operator?

The Concatenation operator (##) in macro is used to concatenate two arguments. Literally, we can say that the arguments are concatenated, but actually their value are not concatenated. Think it this way, if we pass A and B to a macro which uses ## to concatenate those two, then the result will be AB. Consider the example to clear the confusion-

```
#define SOME_MACRO(a, b) a##b

main()

{

  int var = 15;

  printf("%d", SOME_MACRO(v, ar));

}
```

Output of the above program will be 15.

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

There are several ways to code an infinite loop -

```
while(1)

{ }

or,

for(;;)

{ }

or,

Loop:

goto Loop
```

But many programmers prefer the first solution as it is easy to read and self-explanatory, unlike the second or the last one.

48) Guess the output:

```
main()

{

 fork();
```

```
 fork();

 fork();

 printf("hello world\n");

}
```

It will print “hello world' 8 times. The main() will print one time and creates 3 children, let us say Child_1, Child_2, Child_3. All of them printed once. The Child_3 will not create any child. Child2 will create one child and that child will print once. Child_1 will create two children, say Child_4 and Child_5 and each of them will print once. Child_4 will again create another child and that child will print one time. A total of eight times the printf statement will be executed.

49) What is forward reference w.r.t. pointers in c?

Forward Referencing with respect to pointers is used when a pointer is declared and compiler reserves the memory for the pointer, but the variable or data type is not defined to which the pointer points to. For example

```
struct A *p;

struct A

{

 // members

};
```

50) How is generic list manipulation function written which accepts elements of any kind?

It can be achieved using void pointer. A list may be expressed by a structure as shown below

```
typedef struct

{

 node *next;

 /* data part */

 .....

}node;
```

Assuming that the generic list may be like this

```
typedef struct

{
```

```
 node *next;

 void *data;

}node;
```

This way, the generic manipulation function can work on this type of structures.

51) How can you define a structure with bit field members?

Bit field members can be declared as shown below

```
struct A

{

 char c1 : 3;

 char c2 : 4;

 char c3 : 1;

};
```

Here c1, c2 and c3 are members of a structure with width 3, 4, and 1 bit respectively. The ':' indicates that they are bit fields and the following numbers indicates the width in bits.

52) How do you write a function which takes 2 arguments - a byte and a field in the byte and returns the value of the field in that byte?

The function will look like this -

```
int GetFieldValue(int byte, int field )

{

  byte = byte >> field;

  byte = byte & 0x01;

  return byte;

}
```

The byte is right shifted exactly n times where n is same as the field value. That way, our intended value ends up in the 0th bit position. "Bitwise And" with 1 can get the intended value. The function then returns the intended value.

53) Which parameters decide the size of data type for a processor ?

Actually, compiler is the one responsible for size of the data type. But it is true as long as OS allows that. If it is not allowable by OS, OS can force the size.

54) What is job of preprocessor, compiler, assembler and linker ?

The preprocessor commands are processed and expanded by the preprocessor before actual compilation. After preprocessing, the compiler takes the output of the preprocessor and the source code, and generates assembly code. Once compiler completes its work, the assembler takes the assembly code and produces an assembly listing with offsets and generate object files.

The linker combines object files or libraries and produces a single executable file. It also resolves references to external symbols, assigns final addresses to functions and variables, and revises code and data to reflect new addresses.

55) What is the difference between static linking and dynamic linking ?

In static linking, all the library modules used in the program are placed in the final executable file making it larger in size. This is done by the linker. If the modules used in the program are modified after linking, then re-compilation is needed. The advantage of static linking is that the modules are present in an executable file. We don't want to worry about compatibility issues.

In case of dynamic linking, only the names of the module used are present in the executable file and the actual linking is done at run time when the program and the library modules both are present in the memory. That is why, the executables are smaller in size. Modification of the library modules used does not force re-compilation. But dynamic linking may face compatibility issues with the library modules used.

56) What is the purpose of the preprocessor directive #error?

Preprocessor error is used to throw a error message during compile time. We can check the sanity of the make file and using debug options given below

```
#ifndef DEBUG

#ifndef RELEASE

#error Include DEBUG or RELEASE in the makefile

#endif

#endif
```

57) 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 can be achieved by the following code fragment:

```
int *ptr;

ptr = (int *)0x67a9;

*ptr = 0xaa55;
```

58) Significance of watchdog timer in Embedded Systems.

The watchdog timer is a timing device with a predefined time interval. During that interval, some event may occur or else the device generates a time out signal. It is used to reset to the original state whenever some inappropriate events take place which can result in system malfunction. It is usually operated by counter devices.

59) Why ++n executes faster than n+1?

The expression ++n requires a single machine instruction such as INR to carry out the increment operation. In case of n+1, apart from INR, other instructions are required to load the value of n. That is why ++n is faster.

60) What is wild pointer?

A pointer that is not initialized to any valid address or NULL is considered as wild pointer. Consider the following code fragment -

```
int *p;

*p = 20;
```

Here p is not initialized to any valid address and still we are trying to access the address. The p will get any garbage location and the next statement will corrupt that memory location.

61) What is dangling pointer?

If a pointer is de-allocated or freed and the pointer is not assigned to NULL, then it may still contain that address and accessing the pointer means that we are trying to access that location and it will give an error. This type of pointer is called dangling pointer.

62) Write down the equivalent pointer expression for referring the same element a[i][j][k][l] ?

We know that a[i] can be written as *(a+i). Same way, the array elements can be written like pointer expression as follows -

```
a[i][j] == *(*(a+i)+j)

a[i][j][k] == *(*(*(a+i)+j)+k)

a[i][j][k][l] == *(*(*(*(a+i)+j)+k)+l)
```

63) Which bit wise operator is suitable for checking whether a particular bit is on or off?

"Bitwise And" (&) is used to check if any particular bit is set or not. To check whether 5'th bit is set we can write like this

```
bit = (byte >> 4) & 0x01;
```

Here, shifting byte by 4 position means taking 5'th bit to first position and "Bitwise And" will get the value in 0 or 1.

64) When should we use register modifier?

The register modifier is used when a variable is expected to be heavily used and keeping it in the CPU's registers will make the access faster.

65) Why doesn't the following statement work?

```
char str[ ] = "Hello" ;

strcat ( str, '!' ) ;
```

The string function strcat( ) concatenates two strings. But here the second argument is '!', a character and that is the reason why the code doesn't work. To make it work, the code should be changed like this:

```
strcat ( str, "!" ) ;
```

66) Predict the output or error(s) for the following program:

```
void main()

{

   int const * p = 5;

   printf("%d",++(*p));

}
```

The above program will result in compilation error stating "Cannot modify a constant value". Here p is a pointer to a constant integer. But in the next statement, we are trying to modify the value of that constant integer. It is not permissible in C and that is why it will give a compilation error.

67)Guess the output:

```
#include<stdio.h>


main()

{

 unsigned int a = 2;

 int b = -10;

 (a + b > 0)?puts("greater than 0"):puts("less than 1");

}
```

Output - greater than 0

If you have guessed the answer wrong, then here is the explanation for you. The a + b is -8, if we do the math. But here the addition is between different integral types - one is unsigned int and another is int. So, all the operands in this addition are promoted to unsigned integer type and b turns to a positive number and eventually a big one. The outcome of the result is obviously greater than 0 and hence, this is the output.

68) Write a code fragment to set and clear only bit 3 of an integer.

```
#define BIT(n) (0x1 << n)

int a;

void SetBit3()

{

   a |= BIT(3);

}



void ClearBit3()

{

   a &= ~BIT(3);

}
```

69) What is wrong with this code?

```
int square(volatile int *p)

{

   return *p * *p;

}
```

The intention of the above code is to return the square of the integer pointed by the pointer p. Since it is volatile, the value of the integer may have changed suddenly and will result in something else which will looks like the result of the multiplication of two different integers. To work as expected, the code needs to be modified like this.

```
int square(volatile int *p)

{

   int a = *p;
```

```
    return a*a;

}
```

70) Is the code fragment given below is correct? If so what is the output?

```
int i = 2, j = 3, res;

res = i+++j;
```

The above code is correct, but little bit confusing. It is better not to follow this type of coding style. The compiler will interpret above statement as “res = i++ + j”. So the res will get value 5 and i after this will be 3.