Embedded C Interview Questions

Part 2

1. **What kind of loop is better - Count up from zero or Count Down to zero?**

   Loops that involve count down to zero are better than count-up loops. This is because the compiler can optimize the comparison to zero at the time of loop termination. The processors need not have to load both the loop variable and the maximum value for comparison due to the optimization. Hence, count down to 0 loops are always better.

2. **What do you understand by a null pointer in Embedded C?**

   A null pointer is a pointer that does not point to any valid memory location. It is defined to ensure that the pointer should not be used to modify anything as it is invalid. If no address is assigned to the pointer, it is set to `NULL`.

   Syntax:

   ```
   data_type *pointer_name = NULL;
   ```

   One of the uses of the null pointer is that once the memory allocated to a pointer is freed up, we will be using NULL to assign to the pointer so that it does not point to any garbage locations.

3. **Following are some incomplete declarations, what do each of them mean?**

   ```
   1. const int x;
   2. int const x;
   3. const  int  *x;
   4. int * const x;
   5. int  const * x const;
   ```

   The first two declaration points 1 and 2 mean the same. It means that the variable `x` is a read-only constant integer.

   The third declaration represents that the variable `a` is a pointer to a constant integer. The integer value can't be modified but the pointer can be modified to point to other locations.

The fourth declaration means that the variable `x` is a constant pointer to an integer value. It means that the integer value can be changed, but the pointer can't be made to point to anything else.

The last declaration means that the variable x is a constant pointer to a constant integer which means that neither the pointer can point to a different location nor the integer value can be modified.

4. **Why is the statement ++i faster than i+1?**

   ++i instruction uses single machine instruction like INR (Increment Register) to perform the increment.

   For the instruction i+1, it requires to load the value of the variable i and then perform the INR operation on it. Due to the additional load, ++i is faster than the i+1 instruction.

5. **What are the reasons for segmentation fault in Embedded C? How do you avoid these errors?**

   Following are the reasons for the segmentation fault to occur:

   While trying to dereference NULL pointers.

   While trying to write or update the read-only memory or non-existent memory not accessible by the program such as code segment, kernel structures, etc.

   While trying to dereference an uninitialized pointer that might have been pointing to invalid memory.

   While trying to dereference a pointer that was recently freed using the free function.

   While accessing the array beyond the boundary.

   Some of the ways where we can avoid Segmentation fault are:

   Initializing Pointer Properly: Assign addresses to the pointers properly. For instance:

We can also assign the address of the matrix, vectors or using functions like calloc, malloc etc.

Only important thing is to assign value to the pointer before accessing it.

```
int varName;
int *p = &varName;
```

Minimize using pointers: Most of the functions in Embedded C such as scanf, require that address should be sent as a parameter to them. In cases like these, as best practices, we declare a variable and send the address of that variable to that function as shown below:

```
int x;
scanf("%d",&x);
```

In the same way, while sending the address of variables to custom-defined functions, we can use the & parameter instead of using pointer variables to access the address.

```
int x = 1;
x = customFunction(&x);
```

Troubleshooting: Make sure that every component of the program like pointers, array subscripts, & operator, * operator, array accessing, etc as they can be likely candidates for segmentation error. Debug the statements line by line to identify the line that causes the error and investigate them.

6. **Is it recommended to use printf() inside ISR?**

   printf() is a non-reentrant and thread-safe function which is why it is not recommended to call inside the ISR.

7. **Is it possible to pass a parameter to ISR or return a value from it?**

   An ISR by nature does not allow anything to pass nor does it return anything. This is because ISR is a routine called whenever hardware or software events occur and is not in control of the code.

8. **What Is a Virtual Memory in Embedded C and how can it be implemented?**

   Virtual memory is a means of allocating memory to the processes if there is a shortage of physical memory by using an automatic allocation of storage. The

main advantage of using virtual memory is that it is possible to have larger virtual memory than physical memory. It can be implemented by using the technique of paging.

Paging works as follows:

Whenever a process needs to be executed, it would be swapped into the memory by using a lazy swapper called a pager.

The pager tries to guess which page needs to get access to the memory based on a predefined algorithm and swaps that process. This ensures that the whole process is not swapped into the memory, but only the necessary parts of the process are swapped utilizing pages.

This decreases the time taken to swap and unnecessary reading of memory pages and reduces the physical memory required.

9. **What is the issue with the following piece of code?**

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

From the code given, it appears that the function intends to return the square of the values pointed by the pointer p. But, since we have the pointer point to a volatile integer, the compiler generates code as below:

```
int square ( volatile int *p){
    int x , y;
    x = *p ;
    y = *p ;
    return x * y ;}
```

Since the pointer can be changed to point to other locations, it might be possible that the values of the x and y would be different which might not even result in the square of the numbers. Hence, the correct way for achieving the square of the number is by coding as below:

```
long square (volatile int *p ){
    int x ;
    x = *p ;
    return x*x;}
```

10. The following piece of code uses __interrupt keyword to define an ISR. Comment on the correctness of the code.

```
__interrupt double calculate_circle_area (double radius){
    double circle_area  = PI * radius * radius;
    printf ( 'Area = %f ' , circle_area);
    return circle_area;}
```

Following things are wrong with the given piece of code:

ISRs are not supposed to return any value. The given code returns a value of datatype double.

It is not possible to pass parameters to ISRs. Here, we are passing a parameter to the ISR which is wrong.

It is not advisable to have printf inside the ISR as they are non-reentrant and thereby it impacts the performance.

11. What is the result of the below code?

```
void demo(void){
    unsigned int x = 10 ;
    int y = -40;
    if(x+y > 10) {printf("Greater than 10");}
    else {printf("Less than or equals 10");}}
```

In Embedded C, we need to know a fact that when expressions are having signed and unsigned operand types, then every operand will be promoted to an unsigned type. Herem the -40 will be promoted to unsigned type thereby making it a very large value when compared to 10. Hence, we will get the statement “Greater than 10” printed on the console.

12. What are the reasons for Interrupt Latency and how to reduce it?

Following are the various causes of Interrupt Latency:

Hardware: Whenever an interrupt occurs, the signal has to be synchronized with the CPU clock cycles. Depending on the hardware of the processor and the logic of synchronization, it can take up to 3 CPU cycles before the interrupt signal has reached the processor for processing.

Pipeline: Most of the modern CPUs have instructions pipelined. Execution happens when the instruction has reached the last stage of the pipeline. Once

the execution of an instruction is done, it would require some extra CPU cycles to refill the pipeline with instructions. This contributes to the latency.

Interrupt latency can be reduced by ensuring that the ISR routines are short. When a lower priority interrupt gets triggered while a higher priority interrupt is getting executed, then the lower priority interrupt would get delayed resulting in increased latency. In such cases, having smaller ISR routines for lower priority interrupts would help to reduce the delay.

Also, better scheduling and synchronization algorithms in the processor CPU would help minimize the ISR latency.

13. Is it possible to protect a character pointer from accidentally pointing it to a different address?

It can be done by defining it as a constant character pointer. const protects it from modifications.

14. What do you understand by Wild Pointer? How is it different from Dangling Pointer?

A pointer is said to be a wild pointer if it has not been initialized to NULL or a valid memory address. Consider the following declaration:

```
int *ptr;
*ptr = 20;
```

Here the pointer ptr is not initialized and in the next step, we are trying to assign a valid value to it. If the ptr has a garbage location address, then that would corrupt the upcoming instructions too.

If we are trying to de-allocate this pointer and free it as well using the free function, and again if we are not assigning the pointer as NULL or any valid address, then again chances are that the pointer would still be pointing to the garbage location and accessing from that would lead to errors. These pointers are called dangling pointers.

15. What are the differences between the following 2 statements #include "..." and #include <...>?

Both declarations specify for the files to be included in the current source file. The difference is in how and where the preprocessor looks for including the

files. For `#include "..."`, the preprocessor just searches for the file in the current directory as where the source file is present and if not found, it proceeds to search in the standard directories specified by the compiler. Whereas for the `#include <...>` declaration, the preprocessor looks for the files in the compiler designated directories where the standard library files usually reside.

16. When does a memory leak occur? What are the ways of avoiding it?

Memory leak is a phenomenon that occurs when the developers create objects or make use of memory to help memory and then forget to free the memory before the completion of the program. This results in reduced system performance due to the reduced memory availability and if this continues, at one point, the application can crash. These are serious issues for applications involving servers, daemons, etc that should ideally never terminate.

Example of Memory Leak:

```
#include <stdlib.h>

void memLeakDemo(){
   int *p = (int *) malloc(sizeof(int));

   /* Some set of statements */

   return; /* Return from the function without freeing the pointer p*/}
```

In this example, we have created pointer p inside the function and we have not freed the pointer before the completion of the function. This causes pointer p to remain in the memory. Imagine 100s of pointers like these. The memory will be occupied unnecessarily and hence resulting in a memory leak.

We can avoid memory leaks by always freeing the objects and pointers when no longer required. The above example can be modified as:

```
#include <stdlib.h>;

void memLeakFix(){
   int *p = (int *) malloc(sizeof(int));

   /* Some set of statements */

   free(p); // Free method to free the memory allocated to the pointer p
   return;}
```