Preparation Tips – Embedded SW

What are we looking for?

We are looking for individuals, who are passionate about electronics, love to tackle challenging problems and build solutions that have an impact and can improve human lives though technology.

1. **Basics, basics, basics!** Strong basics and fundamentals are what allow engineers to tackle new problems effectively
2. **Perseverance** – people who quickly give up on a problem without trying or just saying “I don’t know” without taking time to think, are pretty much immediately ruled out.
3. **Sharpness** – being able to identify patterns, and making logical leaps that can circumvent multiple steps
4. **Thoroughness** – after making such a leap, being able to go back and justify the answer

Topics – for both written test and interview

- Basic C programming: Pointers, algorithms, stacks/queues
  - NCG:
    - Data structures: upto basics of graphs. BST, Hash, LL, etc. with corresponding algos.
  - Interns:
    - Data structures: upto LL etc. with corresponding algos.
- Embedded C
  - NCG & intern
    - Struct size calculation.
    - Bit manipulation.
    - Memory layout of C program.
    - Static & volatile keyword.
    - Compilation stages.
- Operating systems
  - NCG:
    - Scheduling algorithm
    - Multi-process synchronization techniques.
    - Virtual & physical memory.
  - Inters
    - Multi-process synchronization techniques.
- Computer Arch.
  - NCG
    - Caches.
    - DMA.
    - Organization.
    - Pipeline.

Topics – for both written test and interview

- Microcontroller
  - NCG & Intern
    - 8051 Assembly
    - Interrupts and their processing
    - DMA, timers, memories, ADC, DAC, GPIO & SPI.

References

- C Programming Language – by Brian W. Kernighan and Dennis M. Ritchie
- Embedded Systems: An Integrated Approach – by Layla b Das
- Operating Systems – by Andrew S Tanenbaum
- Computer Organization & Architecture – by William Stallings

Sample Problem 1.1

For a deeply embedded processor such that it:

- do not has any cache
- Has three memories such that in terms of speed they are:

  L1 RAM > L2 RAM > FLASH
- On startup it zero-initialize its .bss section (a place where all un-initialize variables are placed)

identify which implementations would be the slowest and fastest?

Program Section Name	Target Memory
.data	L2 RAM
.bss	L1 RAM
.stack	L1 RAM
.heap	L2 RAM
.code	FLASH
.rodata	FLASH

Implementation 1

```
int global_sum = 0;
compute_sum(range)
{
   for value in range {
      global_sum = global_sum + value
   }
}
```

Implementation 2

```
int global_sum = 0;
compute_sum(range)
{
   int local_sum = 0;
   for value in range {
      local_sum = local_sum + value
   }
   global_sum = local_sum;
}
```

Implementation 3

```
int global_sum = 0;
compute_sum(range)
{
   static int local_sum = 0;
   for value in range {
      local_sum = local_sum + value
   }
   global_sum = local_sum;
}
```

Solution 1.1

Here, we have to remember the following

Program Section Name	Contains
.data	All initialized global sections and initialized local static
.bss	All uninitialized global sections and uninitialized local static
.stack	Call stack
.heap	Dynamic allocation
.text	Code
.rodata	Read only data

Therefore, implementation 3 would be slowest, since, this implementation access variables stored in .data section a greater number of times.
Implementation 2 is the fastest because, even without any compiler optimization, local_sum would be stored in .stack which is present in L1RAM.

Sample Problem 1.2

Identify the error in the implementation:

```
int length;
void foo(int sg)
{
   int reverse_sg;
   while(sg>0)
   {
    length = length + 1;
    reverse_sg = reverse_sg * 10 + (sg%10);
    sg = sg / 10;
   }
}
```

Solution 1.2

- Here the problem is with file reverse_sg = reverse_sg * 10 + (sg%10); because *reverse_sg* is a local variable which could be stored in either stack or in CPU register bank and in either case, it will have garbage value, however, *length* variable will never be a problem because program on startup is zero-initializing the bss section.

Sample Problem 2

Consider a high performance application which is running on CortexA72. It has many different functions

1. it has a function that loops over every element of a linear data-structure starting from a beginning to end. To reduce the run time of this function, which linear data structure should it loop onto?
2. It has a function that does computation of a 2d matrix. Should that matrix be computed in column-major fashion or row-major fashion in order to reduce run-time? Why?
3. In a linked-list, how would you detect a loop?

Solution 2

1. Array because it has good special locality.
2. Row-major because it has good special locality.
3. Floyd's cycle-finding algorithm is used for this.

Sample Problem 3

An Industrial memory access controller access a memory bank which is 16K-bytes in size and is organized as 16K × 8. During testing, it was found that

1. At normal temperature, it works as per its design i.e. it reads and writes 4 bytes at a time.
2. at elevated temperature of 105 deg C, it writes 4 bytes correctly but reads 4 bytes with Least-Significant-Byte as garbage.

Write functions in C that read 4 bytes of data at a time at elevated temperature.

Solution 3

- This is to test bit manipulation.

```
// function to read bytes at normal temperature.
uint32_t read4bytes(uint32_t address)
{
        return MemoryController_HAL_get4bytes(address);
}

// function to read bytes at elevated temperature.
uint32_t read4bytes_ex(uint32_t address)
{
       uint32_t hi = read4bytes(address);
       uint32_t lo = read4bytes(address + 1);
       uint32_t res = (hi & ((uint32_t)(~0xff))) | ((lo >> 8) & ((uint32_t)(0xff)));
       return res;
}
```

Thank you!

All the best ☺