1.What are static variables?

2.What are volatile variables?

3.What do you mean by const keyword ?

4.What is interrupt latency?

5.How you can optimize it?

6.What is size of character, integer, integer pointer, character pointer?

7.What is NULL pointer and what is its use?

8.What is void pointer and what is its use?

9.What is ISR?

10.What is return type of ISR?

11.Can we use any function inside ISR?

12.Can we use printf inside ISR?

13.Can we put breakpoint inside ISR?

14.How to decide whether given processor is using little endian format or big endian format ?

15.What is Top half & bottom half of a kernel?

16.Difference between RISC and CISC processor.

17.What is RTOS?

18.What is the difference between hard real-time and soft real-time OS?

19.What type of scheduling is there in RTOS?

20.What is priority inversion?

21.What is priority inheritance?

22.How many types of IPC mechanism you know?

23.What is semaphore?

24.What is spin lock?

25.What is difference between binary semaphore and mutex?

26.What is virtual memory?

27.What is kernel paging?

28.Can structures be passed to the functions by value?

29.Why cannot arrays be passed by values to functions?

30.Advantages and disadvantages of using macro and inline functions?

31.What happens when recursion functions are declared inline?

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

33.Can you have constant volatile variable? Yes, you can have a volatile pointer?

34.++*ip increments what? it increments what ip points to

35.Operations involving unsigned and signed — unsigned will be converted to signed

36.malloc(sizeof(0)) will return — valid pointer

37.main() {fork();fork();fork();printf("hello world"); } — will print 8 times.

38.Array of pts to functions — void (*fptr[10])()

39.Which way of writing infinite loops is more efficient than others? there are 3ways.

40.Who to know whether system uses big endian or little endian format and how to convert among them?

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

42.How is generic list manipulation function written which accepts elements of any kind?

43.What is the difference between embedded systems and the system in which RTOS is running?

44.How can you define a structure with bit field members?

45.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?

46.Which parameters decide the size of data type for a processor ?

47.What is job of preprocessor, compiler, assembler and linker ?

48.What is the difference between static linking and dynamic linking ?

49.How to implement a WD timer in software ?

Notes

once again, null pointer is a value, while void pointer is a type. These concepts are totally different and non-comparable.

Variables of type int are one machine-type word in length

Variables of type char are 1 byte in length. They can be signed this is the default or unsigned

you CANNOT CALL AN ISR, BUT CAN CALL A FUNCTION WITHIN IT.

- A variable is **defined** when the compiler allocates the storage for the variable.
- A variable is **declared** when the compiler is informed that a variable exists (and this is its type); it does not allocate the storage for the variable at that point.

- **auto : It is a local variable known only to the function in which it is declared. Auto is the default storage class**

- **static : Local variable which exists and retains its value even after the control is transferred to the calling function.**
- **extern : Global variable known to all functions in the file**
- **register : Social variables which are stored in the register.**

---

the difference between the lvalue and rvalue in the compiler message

Expressions that refer to memory locations are called "l-value" expressions. An l-value represents a storage region's "locator" value, or a "left" value, implying that it can appear on the left of the equal sign (=). L-values are often identifiers.

Expressions referring to modifiable locations are called "modifiable l-values." A modifiable l-value cannot have an array type, an incomplete type, or a type with the const attribute. For structures and unions to be modifiable l-values, they must not have any members with the const attribute. The name of the identifier denotes a storage location, while the value of the variable is the value stored at that location.

An identifier is a modifiable l-value if it refers to a memory location and if its type is arithmetic, structure, union, or pointer. For example, if `ptr` is a pointer to a storage region, then `*ptr` is a modifiable l-value that designates the storage region to which `ptr` points.

Any of the following C expressions can be l-value expressions:

- An identifier of integral, floating, pointer, structure, or union type
- A subscript ([ ]) expression that does not evaluate to an array
- A member-selection expression (–> or .)
- A unary-indirection (*) expression that does not refer to an array
- An l-value expression in parentheses
- A const object (a nonmodifiable l-value)

The term "r-value" (reference value)is sometimes used to describe the value of an expression and to distinguish it from an l-value. All l-values are r-values but not all r-values are l-values.

---

standard ANSI C recognizes the following keywords

auto : Defines a local variable as having a local lifetime

break : Passes control out of the compound statement

const : Makes variable value or pointer parameter unmodifiable.

continue : Passes control to the begining of the loop.

enum : Defines a set of constants of type int.

extern : Indicates that an identifier is defined elsewhere.

register : Tells the compiler to store the variable being declared in a CPU register.

return : exits immediately from the currently executing function to the calling routine, optionally returning a value.

static : Preserves variable value to survive after its scope ends.

struct : Groups variables into a single record.

union : Groups variables which share the same storage space.

volatile : Indicates that a variable can be changed by a background routine.

- `int*` - pointer to int
- `int const *` - pointer to const int
- `int * const` - const pointer to int
- `int const * const` - const pointer to const int

Now the first const can be on either side of the type so:

- `const int *` == `int const *`
- `const int * const` == `int const * const`

If you want to go really crazy you can do things like this:

- `int **` - pointer to pointer to int
- `int ** const` - a const pointer to a pointer to an int
- `int * const *` - a pointer to a const pointer to an int
- `int const **` - a pointer to a pointer to a const int
- `int * const * const` - a const pointer to a const pointer to an int

OS

thread of execution is the smallest unit of processing that can be scheduled by an operating system. A thread is a lightweight process. The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is contained inside a process.

The **kernel** (from German *Kern*, nucleus, core) is the main component of most computer operating systems; it is a bridge between applications and the actual data processing done at the hardware level. The kernel's responsibilities include managing the system's resources (the communication between hardware and software components) Usually as a basic component of an operating system, a kernel can provide the lowest-level abstraction layer for the resources (especially processors and I/O devices) that application software must control to perform its function. It typically makes these facilities available to application processes through inter-process communication mechanisms and system calls.

an operation (or set of operations) is **atomic**, **linearizable**, **indivisible** or **uninterruptible** if it appears to the rest of the system to occur instantaneously. Atomicity is a guarantee of isolation from concurrent processes. Additionally, atomic operations commonly have a succeed-or-fail definition — they either successfully change the state of the system, or have no visible effect.

Answers

ask)1.What are static variables?

the external var have global scope across the entire program and a lifetime over the entire program run.
static, similarly provides a lifetime over the entire program, however; it provides a way to limit the scope of such variables
These variables are automatically initialized to zero upon memory allocation just as external variables are.
Static automatic variables continue to exist even after the block in which they are defined terminates. Thus, the value of a static variable in a function is retained between repeated function calls to the same function. The scope of static automatic variables is identical to that of automatic variables, i.e. it is local to the block in which it is defined; however, the storage allocated becomes permanent for the duration of the program. Static variables may be initialized in their declarations; however, the initializers must be constant expressions, and initialization is done only once at compile time when memory is allocated for the static variable

* the difference between external var and global var

When applied to the declaration of a variable, extern tells the compiler that said variable will be found in the symbol table of a different translation unit. This is used to expose variables in one implementation file to a different implementation file. The external declaration is not resolved by the compiler, but instead by the linker, meaning that extern is a legitimate mechanism to install modularity at the compiler level.

GLOBAL VARIABLE
Global variables are used extensively to pass information between sections of code that don't share a caller/callee relation like concurrent threads and signal handlers

What are the different storage classes in C?

U have given answer for storage specifiers, storage classes are only two types
1.auto
2.static
storage class specifiers
1.auto : reside the data is stack
2.static : resides the data in the memory and bss (**bss** is (**Block Started by Symbol**) used by many compilers and linkers for a part of the data segment containing statically-allocated variables represented solely by zero-valuedbits initially)
3.extern : resides the data in data memory
4.register : resides the data in the cpu register fast build but limited

---

2.What are volatile variables?

if a variable is declared as volatile ,optimization is not done regarding to that variable.If the value is modified many number of times, each time CPU goes and reads from main memory rather than cache memory.

The usage of this can be understood clearly only at kernel level but not at application level.

3.What do you mean by const keyword ?

The constant will be a read only data stored in Rom once it declared it will contain a rubbish it may or mayn't be modified it depends on the compiler so you have to define the constant variables not to declare them

When declaring a const variable, it is possible to put const either before or after the type
***the difference between const and #define**

#define simply replaces whatever you have defined by the text you want it to replace.
const variable's value cannot be manipulated during the course of the program.

#define is a text preprocessor command and like all text preprocessor commands (beginning with "#") are handled by textual substitution throughout the code before the compiler sees any of the code.

const is a compiler keyword that identifies a constant declaration. It is handled by the actual compiler

———

what are the c qualifiers

1) Volatile:
A variable should be declared volatile whenever its value could change unexpectedly. In practice, only three types of variables could change:

- **Memory-mapped peripheral registers**
- **Global variables modified by an interrupt service routine**
- **Global variables within a multi-threaded application**

2) Constant:
The addition of a 'const' qualifier indicates that the (relevant part of the) program may not modify the variable.

———

4.What is interrupt latency?

interrupt latency is the time between the generation of an interrupt by a device and the servicing of the device which generated the interrupt

here is usually a trade-off between interrupt latency, throughput (the average rate of successful message delivery over a communication channel), and processor utilization. Many of the techniques of CPU and OS design that improve interrupt latency will decrease throughput and increase processor utilization. Techniques that increase throughput may increase interrupt latency and increase processor utilization. Lastly, trying to reduce processor utilization may increase interrupt latency and decrease throughput.

5.How you can optimize it?

when the interrupt is caused the we have to save the registers of the cpu that used to handle the instruction too many registers lead to much more time to be taken in the saving process so we need the instruction to use few no of registers to reduce the context switching time and therefore reduce the latency but it is undesirable since it will slow the down most non-interrupt processing

factors affecting interrupt latency include:

Cycles needed to complete current CPU activities. To minimize those costs, microcontrollers tend to have short pipelines (often three instructions or less), small write buffers, and ensure that longer instructions are continuable or restartable. RISC design principles ensure that most instructions take the same number of cycles, helping avoid the need for most such continuation/restart logic.
The length of any critical section that needs to be interrupted

Interrupt nesting. Some microcontrollers allow higher priority interrupts to interrupt lower priority ones. This allows software to manage latency by giving time-critical interrupts higher priority (and thus lower and more predictable latency) than less-critical ones

Trigger rate. When interrupts occur back-to-back, microcontrollers may avoid an extra context save/restore cycle by a form of tail call optimization.

Reduce the time spent in the disabled region

reduce no of interrupts in the disable region to avoid creating race conditions with themselves

——

6.What is size of character, integer, integer pointer, character pointer?

Variables of type int are one machine-type word in length
Variables of type char are 1 byte in length. They can be signed this is the default or unsigned
*** Actually a pointer is just a address holder so its size is always that of an int data type,what ever may be the type of pointer.In a 16-bit compiler,its 2 bytes and in 32-bit compiler,its 4 bytes**

——

7.What is NULL pointer and what is its use?

A null pointer has a value reserved for indicating that the pointer does not refer to a valid object. it point to the address 0 which is reserved for the interrupt vector table so a null pointer does not point to a meaningful object, an attempt to dereference a null pointer usually causes a run-time error that 's why we initialize the pointer to null to make sure we will assign them before use them and if we don't it will gives us a run time error.

——

8.What is void pointer and what is its use?

Generic pointer can hold the address of any data type.

Void pointer is a specific pointer type - void * - a pointer that points to some data location in storage, which doesn't have any specific type.

———————————————————————————————————————

9.What is ISR? and the difference between ISR and subroutine??

routine executed in response to an interrupt is called as the interrupt service routine(ISR).
Subroutine is the part of a big program which perform some specific part related to the program.
Subroutine is called by the program itself. It helps to avoid complexity in the program.
ISR is the code executed in response to the interrupt which may occur during the execution of a program.
ISR may have nothing in common to the program which is being executed at the time of interrupt is received.
In the case of subroutine, only address of next field is stored in the stack automatically when it is called while in the case of ISR, along with the address of next field, content of status register and some other registers(if required) are also store

———————————————————————————————————————

10.What is return type of ISR? void

why can't we pass the arguments to an ISR and also why the ISR returns nothing.?

An ISR is called when a hardware event occurs and is not called by code, so parameters cannot be passed to it. In addition, when you write an ISR, you know what conditions caused the interrupt, and therefore know what to do about it, without needing parameters to give additional information.

An ISR returns nothing, because there is no calling code to read the returned values - on return, program execution returns to the original instruction stream.

———————————————————————————————————————

11.Can we use any function inside ISR?

you should be aware of the stack of the ISR when you use a fun in the ISR the fun must be less than the stack of the ISR or else you will have a stack overflow problem

You can call a function from an isr if it is only the isr that is calling the function. Then it can be considered as part of the isr.

If some other part of the program is also calling the function, then it must be a re-entrant function. The isr might interrupt when the program is inside the function.

A re-entrant function is a function that does not use global or static variables and does not call other non re-entrant functions.

Some compilers will flag an error if an isr calls a function that is also called from another part of the program.Remember

Regardless of whether, or not, you call a function from your ISR (which is perfectly acceptable given the constraints others mentioned), the critical thing is the length of time the ISR service routine takes to execute. If the code takes too long to execute your system can become "interrupt bound". That is, it has no time to do anything other than service the interrupt. In the worse case, another interrupt occurs before the previous one's service routine has completed.

——

12.Can we use printf inside ISR?

Do not use printf/sprintf in an ISR

printf() should work inside the can ISR however this will introduce many areas for potential problems.

printf() is not reentrant so unless interrupts are disabled while calling it, it cannot be called from main code or from any other interrupt that does not have the same priority as the ISR use the printf() . In addition, it can never be called from the serial ISR. The library code for printf() is large (about 1kb) and slow to execute. This will dramatically affect the performance of your ISR and therefore your application. In addition it may adversely affect the behavior of your application.

——

13.Can we put breakpoint inside ISR?

Yes. We can use breakpoint inside the ISR.

——

14.How to decide whether given processor is using little endian format or big endian format ?

Endianness is the format to how multi-byte data is stor ed in computer memory. It describes the location of the most significant byte (MSB) and least significant byte (LSB) of an address in memory. Endianness is dictated by implemented, but rather the endian m odel of the CPU architecture dictates how the operating system is implemented

Software is sometimes designed with one specific Endian-architecture. the CPU architecture implementation of the system. The operating system does not dictate the endian modelSoftware is sometimes designed with one specific Endian-architecture in mind, limiting the portability of the code to other processor architectures. This type of implementation is considered to be *Endian-specific* .

Endian-neutral software can be developed, allowing the code to be ported easily between processors of different Endian-architectures

There are two main areas where Endianness must be considered. One area pertains to code portability. The second area pertains to sharing data between platforms.

Big-Endian stores the MSB at the lowest memory addres Little-Endian stores the LSB at the lowest memory address. in Big- endian test whether the number is positive or negative by looking at the byte at offset zero The numbers are also stored in the order in which they are printed out, so binary to decimal routines are particularly efficient.

 Most embedded communication processors and custom solu tions associated with the data plane are Big-Endian

The lowest memory address of multi-byte data is considered the starting address of the data

Note that some CPUs can be either big or little endian (Bi-Endian) by setting a processor register to the desired endian-architecture.

To differentiate between the big and the little endians write this simple code

```
int i = 1;                          // no to be stored in memory
char *pc = (char *)&i;
            // pointer to the 1st byte of the address that stored the num

if ( *pc == 1 ) //check if the 1st byte contain the num or something else here (0)
{
    /* little endian */  // if it contains the num then it's a little-endian
}
else
{
    /* Big endian */
}
```

————————————————————————————————————

ask)15.What is Top half & bottom half of a kernel?

both of them are approaches to handle the interrupts the bottom half of the kernel is simply a set of routines that are invoked to handle interrupts . servicing a remote invocation on the bottom half may be straightforward and fast but this approach has 3 drawbacks .

first it can only be used for the operation that can be executed safely in an interrupt handler. In most of OS the interrupt handlers are not allowed to block since this need a per- task context and interrupt handler can't relinquish the processor in order to wait for resources but it must run tell completion

second it may lead to indefinite blocking of the current task that was running before the interrupt occurs when a series of remote invocation requests arrives to the master the application task on the master kernel will be interrupted and must wait until all the requests are completely serviced . furthermore new requests may arriver during the old request are serviced this implies that the application task can suffer indefinitely long blocking

third since interrupt handling commonly has a higher priority than the application tasks high priority tasks may be interrupted by the remote requests initiated by low priority tasks

the top half allow the kernel service handler to execute in task contest this permits the requested operation to block during its execution note that the top half may be slower since it require a context switch out of the interrupt handler and may require a synchronization with the other tasks running in the kernel it also require more complicated implementation

since we have many system calls involve blocking so we rather prefer the top half than the bottom half but we can have a mix like in linux

One of the main problems with interrupt handling is how to perform lengthy tasks within a handler. Often a substantial amount of work must be done in response to a device interrupt, but interrupt handlers need to finish up quickly and not keep interrupts blocked for long. These two needs (work and speed) conflict with each other, leaving the driver writer in a bit of a bind.

Linux (along with many other systems) resolves this problem by splitting the interrupt handler into two halves. The so-called top half is the routine that actually responds to the interrupt—the one you register with request_irq. The bottom half is a routine that is scheduled by the top half to be executed later, at a safer time. The big difference between the top-half handler and the bottom half is that all interrupts are enabled during execution of the bottom half—that's why it runs at a safer time. In the typical scenario, the top half saves device data to a device-specific buffer, schedules its bottom half, and exits: this operation is very fast. The bottom half then performs whatever other work is required, such as awakening processes, starting up another I/O operation, and so on. This setup permits the top half to service a new interrupt while the bottom half is still working.

to implement bottom-half processing, Tasklets are often the preferred mechanism for bottom-half processing; they are very fast, but all tasklet code must be atomic. The alternative to tasklets is workqueues, which may have a higher latency but that are allowed to sleep

———

16.Difference between RISC and CISC processor.

CISC
Complexity found in hardware
Memory-to-memory : load and store functionality found in a single instruction Less lines of code needed to provide same functionality
instructions not always the same size
instructions are difficult to decode because instructions are not uniform
to make use of pipelining,instructions need to be broken down to smaller components at processor level
RISC
Complexity on software side
Register-to-Register : load and store are separate instructions
More instructions necessary to provide same functionality
all instructions of a uniform size
instructions are easier to decode because of how they are set up-ex: opcode will always be in the same place
capable of using pipelining by design
While the PC world is dominated by CISC processors, elsewhere mostly RISC processors are used.

———

17.What is RTOS?

A real-time operating system (RTOS) is an operating system that guarantees a certain capability within a specified time constraint

Features of RTOS

An RTOS must be designed in a way that it should strike a balance between supporting a rich feature set for development and deployment of real time applications and not compromising on the deadlines and predictability.

The following points describe the features of an RTOS (Note that this list is not exhaustive) :

§ Context switching latency should be short. This means that the time taken while saving the context of current task and then switching over to another task should be short.

§ The time taken between executing the last instruction of an interrupted task and executing the first instruction of interrupt handler should be predictable and short. This is also known as interrupt latency.

§ Similarly the time taken between executing the last instruction of the interrupt handler and executing the next task should also be short and predictable. This is also known as interrupt dispatch latency.

§ Reliable and time bound inter process mechanisms should be in place for processes to communicate with each other in a timely manner.

§ An RTOS should have support for multitasking and task preemption. Preemption means to switch from a currently executing task to a high priority task ready and waiting to be executed.

§ Real time Operating systems but support kernel preemption where-in a process in kernel can be preempted by some other process.

Some Misconceptions related to RTOS

§ *RTOS should be fast.* This is not true. An RTOS should have a deterministic behavior in terms of deadlines but its not true that the processing speed of an RTOS is fast. This ability of responsiveness of an RTOS does not mean that they are fast.
§ *All RTOS are same.* As already discussed we have three types of RTOS (Hard, firm and soft).
§ *RTOS cause considerable amount of CPU overhead.* Well, again this is not true. Only 1%-4% of CPU time is required by an RTOS

——

18.What is the difference between hard real-time and soft real-time OS?

1. **Hard RTOS** : These type of RTOS strictly adhere (commit) to the deadline associated with the tasks. Missing on a deadline can have catastrophic(disaster) affects. The air-bag example we discussed in the beginning of this article is example of a hard RTOS as missing a deadline there could cause a life.

2. **Firm RTOS** : These type of RTOS are also required to adhere to the deadlines because missing a deadline **may not** cause a catastrophic affect but could cause undesired affects, like a huge reduction in quality of a product which is highly undesired.

3. **Soft RTOS** : In these type of RTOS, missing a deadline is acceptable. For example On-line Databases.

——

19.What type of scheduling is there in RTOS?

RTOS uses priority-based preemptive scheduling, which allows high-priority threads to meet their deadlines consistently.

———

20.What is priority inversion?

priority inversion is a problematic scenario in scheduling when a higher priority task is indirectly preempted by a lower priority task effectively "inverting" the relative priorities of the two tasks.

This violates the priority model that high priority tasks can only be prevented from running by higher priority tasks and briefly by low priority tasks which will quickly complete their use of a resource shared by the high and low priority tasks.

———

21.What is priority inheritance?

Under the policy of priority inheritance, whenever a high priority task has to wait for some resource shared with an executing low priority task, the low priority task is temporarily assigned the priority of the highest waiting priority task for the duration of its own use of the shared resource, thus keeping medium priority tasks from pre-empting the (originally) low priority task, and thereby affecting the waiting high priority task as well. Once the resource is released, the low priority task continues at its original priority level.

———

22.How many types of IPC mechanism you know?

IPC is : Inter Process Communication is a set of methods for the exchange of data among multiple tasks(threads) in one or more processes

1) Message passing is the paradigm of communication where messages are sent from a sender to one or more recipients. Forms of messages include (remote) method invocation, signals, and data packets. When designing a message passing system several choices are made:

- Whether messages are transferred reliably
- Whether messages are guaranteed to be delivered in order
- Whether messages are passed one-to-one (unicast), one-to-many (multicast or broadcast), many-to-one (client–server), or many-to-many (AllToAll).
- Whether communication is synchronous or asynchronous.

2) Process synchronization refers to the idea that multiple processes are to join up or handshake at a certain point, so as to reach an agreement or commit to a certain sequence of action.

synchronization or serialization, strictly defined, is the application of particular mechanisms to ensure that two concurrently-executing processes do not execute specific portions of a program at the same time. If one process has begun to execute a serialized portion of the program, any other process trying to execute this portion must wait until the first process finishes. Synchronization is used to control access to state both in small-scale multiprocessing systems

3) shared memory is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies. Shared memory is an efficient means of passing data between programs. Depending on context, programs may run on a single processor or on multiple separate processors. Using memory for communication inside a single program

a method of inter-process communication (IPC), i.e. a way of exchanging data between programs running at the same time. One process will create an area in RAM which other processes can access,

Since both processes can access the shared memory area like regular working memory, this is a very fast way of communication . On the other hand, it is less powerful, as for example the communicating processes must be running on the same machine and care must be taken to avoid issues if processes sharing memory are running on separate CPUs and the underlying architecture is not cache coherent.

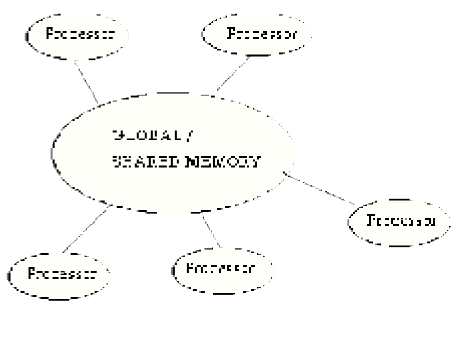
IPC socket (inter-process communication socket **which** is kind of shared memory technique) is a data communications endpoint for exchanging data between processes executing within the same host operating system.

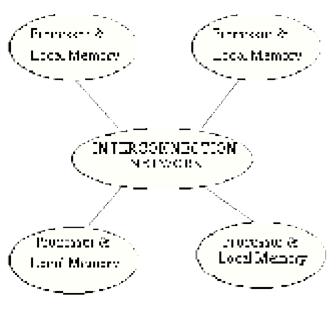
4) remote procedure call (**RPC**) is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space(commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction.

OR

1. Using a SHARED MEMORY and SHARED VARIABLES

This consists of a global address space which is accessible by all N processors. A processor can communicate to another by writing into the global memory where the second processor can read it.





Shared memory solves the inter processes communication problem but introduces the problem of simultaneous accessing of the same location in the memory.

INTERCONNECTION NETWORK and MESSAGE PASSING

Here each processor has its own private (local) memory and there is no global, shared memory. Therefore the processors need to be connected in some way to allow communication of data to take place

———

23.What is semaphore?

semaphores are a technique for coordinating or synchronizing activities in which multiple processes compete for the same operating system resources. A semaphore is a value in a designated place in operating system (or kernel) storage that each process can check and then change. Depending on the value that is found, the process can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphores can be binary (0 or 1) or can have additional values. Typically, a process using semaphores checks the value and then, if it using the resource, changes the value to reflect this so that subsequent semaphore users will know to wait.

Semaphores are commonly use for two purposes: to share a common memory space and to share access to files. Semaphores are one of the techniques for interprocess communication (IPC)
The C programming language provides a set of interfaces or "functions" for managing semaphores.

Semaphore is a location in memory whose value can be tested and set by more than one process.The test and set operation is, so far as each process is concerned uninterruptible or atomic to avoid a dead lock ; once started nothing can stop it. The result of the test and set operation is the addition of the current value of the semaphore and the set value, which can be positive or negative. Depending on the result of the test and set operation one process may have to sleep until the semphore's value is changed by another process. Semaphores can be used to implement critical regions, areas of critical code that only one process at a time should be executing.

———

24.What is spin lock?

a **spinlock** is a lock where the process simply waits in a loop ("spins") repeatedly checking until the lock becomes available. Since the process remains active but isn't performing a useful task, the use of such a lock is a kind of busy waiting. Once acquired, spinlocks will usually be held until they are explicitly released, although in some implementations they may be automatically released if the thread being waited on (that which holds the lock) blocks, or "goes to sleep

Spinning can also be used to generate an arbitrary time delay

———

25.What is difference between binary semaphore and mutex?

Mutexes are typically used to serialise access to a section of re-entrant code that cannot be executed concurrently by more than one process. A mutex object only allows one process into a controlled section, forcing other processes which attempt to gain access to that section to wait until the first process has exited from that section

(A mutex is really a semaphore with value 1.)

A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore).

——

26.What is virtual memory?

Virtual memory is a feature of an operating system that enables a process to use a memory (RAM) address space that is independent of other processes running in the same system, and use a space that is larger than the actual amount of RAM present, temporarily relegating some contents from RAM to a disk, with little or no overhead.

Virtual memory enables each process to act as if it has the whole memory space to itself, since the addresses that it uses to reference memory are translated by the virtual memory mechanism into different addresses in physical memory. This allows different processes to use the same memory addresses - the memory manager will translate references to the same memory address by two different processes into different physical addresses. One process generally has no way of accessing the memory of another process. A process may use an address space larger than the available physical memory, and each reference to an address will be translated into an existing physical address. The bound on the amount of memory that a process may actually address is the size of the swap area, which may be smaller than the addressable space.

——

27.What is kernel paging?

paging is one of the memory-management schemes by which a computer can store and retrieve data from secondary storage for use in main memory. , the operating system retrieves data from secondary storage in same-size blocks called *pages*. The main advantage of paging over memory segmentation is that it allows the physical address space of a process to be noncontiguous

Kernel paging works the same way as any other paging. Code and data (of the OS) that is frequently accessed will be kept in RAM while the remainder will remain on disk - where it belongs. The system will not remove any portion of the kernel from RAM unless it has found a better use for it

in the context of the kernel "Nonpaged" means code and data that can never be paged out under any circumstances. "Paged" means code and data that CAN be paged out if necessary

When code is paged out it is not normally copied to the pagefile. This is not necessary as it can simply be reloaded from the original files. This applies to most code, not just the kerenl

——

28.Can structures be passed to the functions by value?

Structures can be passed to the functions by value. But it has to be copied to another location. Hence wastage of memory since the structure has more than one variable (members) which means it reserve a lot of memory area

Yes structures can be passed to functions by value. Though passing by value has two disadvatanges :

1) The chages by the calling function are not reflected

2) Its slower than the pass by refrence funcion call.

——

29.Why cannot arrays be passed by values to functions?

Arrays can't be passed by values. Because , the array name is evaluated to be a pointer to the first element of the array. and a called function uses this pointer (passed as an argument) to indirectly access the elements of the array

——

ask)30.Advantages and disadvantages of using macro and inline functions?

Advantage: Macros and Inline functions are efficient than calling a normal function. The times spend in calling the function is saved in case of macros and inline functions as these are included directly into the code.

Disadvantage: Macros and inline functions increased the size of executable code.

Difference in inline functions and macro
1) Macro is expanded by preprocessor and inline function are expanded by compiler.
// 2) Expressions passed as arguments to inline functions are evaluated once. In some cases, expressions passed as arguments to macros can be evaluated more than once.

More over inline functions are used to overcome the overhead of function calls. Macros are used to maintain the readability and easy maintenance of the code.

inline processed during compilation .compiler optimizes a inline function either as a function if the performance overhead in calling a function is less compared to macro implementation, OR, it is implemented as Macro itself. This optimization makes inline more preferable than macro

In MACROS the code is literally copied into the location it was called from. So if the user passes a "double" instead of an "int" then problems could occur.

In INLINE functions, if this scenario happens the compiler will complain about incompatible types.

In MACRO --> you cannot make the macro return something which is not the result of the last expression invoked inside it.

In INLINE --> you can return any value by using keyword return();

In MACROS -- > debugging is tough (because they refer to the expanded code, rather than the code the programmer typed.)

INLINe --> debugging is easy

*** The basic idea is to save time at a cost in space. Inline functions are a lot like a placeholder Once you define an inline function, using the 'inline' keyword, whenever you call that function the compiler will replace the function call with the actual code from the function.**

How does this make the program go faster? Simple, function calls are simply more time consuming than writing all of the code without functions. To go through your program and replace a function you have used 100 times with the code from the function would be time consuming not too bright. Of course, by using the inline function to replace the function calls with code you will also greatly increase the size of your program.

*** A macro is a way to automate a task that you perform repeatedly or on a regular basis. It is a series of commands and actions that can be stored and run whenever you need to perform the task. You can record or build a macro, and then play the macro to automatically repeat the series of commands or actions. it is a kind of text manipulation**

A macro is a set of commands that can be played back at will to perform a given task. These tasks can be something simple from inserting your name and address into a word processor(it is a kind of text manipulation) to something more complex such as launching a program, copying data from it, activating another program, pasting the data into it and repeating this several times. Tasks performed by macros are typically repetitive in nature allowing significant savings in time by executing the macro instead of manually repeating the commands.

Preprocessor macros are just substitution patterns applied to your code. They can be used almost anywhere in your code because they are replaced with their expansions before any compilation starts.

Inline functions are actual functions whose body is directly injected into their call site. They can only be used where a function call is appropriate.

Now, as far as using macros vs. inline functions in a function-like context, be advised that:

- **Macros are not type safe, and can be expanded regardless of whether they are syntatically correct - the compile phase will report errors resulting from macro expansion problems.**
- **Macros can be used in context where you don't expect, resulting in problems**
- **Macros are more flexible, in that they can expand other macros - whereas inline functions don't necessarily do this.**
- **Macros can result in side effects because of their expansion, since the input expressions are copied wherever they appear in the pattern.**
- **Inline function are not always guaranteed to be inlined - some compilers only do this in release builds, or when they are specifically configured to do so. Also, in some cases inlining may not be possible.**

- **Inline functions can provide scope for variables (particularly static ones), preprocessor macros can only do this in code blocks {...}, and static variables will not behave exactly the same way.**

——

31.What happens when recursion functions are declared inline?

Recursion (computer science) a procedure or subroutine, implemented in a programming language, whose implementation references itself
programming languages support recursion by allowing a function to call itself within the program text

Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem.

"The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions" like the factorial function you may use it to get the factorial of any no if you do it with a recursive function

An inline function replaces the call to the function by the body of the function, thus reducing the overhead of saving the context in stack. This is good for functions which are small in size and called occasionally. A recursive function calls an instance of itself and thus can be a deeply nested. Different compilers handle this differently. Some will inline it up to a certain depth and then call a non-inlined instance for further recursion; others will not inline the function at all and generate a normal function call.

——

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

in this case the cat(x,y) is the macro which is defined by using the preprocessor directive , this will be substituted only at the place where it is called in this example it happens like this cat(1,2)##3 which will once again become 1##2##3
here if we use ## in between we can join or concatenat only two variables that why it gives a preprocessor warning
it is just a text manipulation if it has a match between the text as it was defined then it will replace it but if the text wasn't as predefined then we will have a preprocessing error

——

33.Can you have constant volatile variable? Yes, you can have a volatile pointer?

Constants can be defined by placing the keyword const in front of any variable declaration. If the keyword volatile is placed after const, then this allows external routines to modify the variable (such as hardware devices). This also forces the compiler to retrieve the value of the variable each time it is referenced rather than possibly optimizing it in a register.

Example: An input-only buffer for an external device could be declared as const volatile (or volatile const, order is not important) to make sure the compiler knows that the variable should not be changed (because it is input-only) and that its value may be altered by processes other than the current program

——

34.++*ip increments what?

it will increment the value of ip .reason is ,operator hireachary pointer will bind first to variable ip so expression will become like this ++(value of ip)

——

35.Operations involving unsigned and signed ?

unsigned will be implicitly promoted to signed

note

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

Here output would give you "> 6". The reason for this is that expressions involving signed and unsigned types have all operands
promoted to unsigned types
in the above condition (a+b > 6) condition returns -14 i.e. a non zero value so he checking with nonzero value that's why it prints ">6".
so in this case it condition result defaultly treats as unsinged. until if you typecast it to signed

——

36.malloc(sizeof(0)) will return — valid pointer

**sizeof(0) returns the size of integer whose value is 0. Hence malloc
would allocate sizeof(int) bytes and return a valid pointer.
malloc always returns (void *) (void pointer).which we have to explicitly Typecast it
It is equivalent to free() and it returns valid pointer.
code to check it**

```
int *p;
p = (int*)malloc(sizeof(0));
if ( p == NULL )
printf("Invalid Pointer");
else
printf("Valid Pointer");
```

——

37.main() {fork();fork();fork();printf("hello world"); } — will print 8 times.

The `fork()' used to create a new process from an existing process. The new process is called the child process, and the existing process is called the parent. We can tell which is which by checking the return value from `fork()'. The parent gets the child's pid returned to him, but the child gets 0 returned to him.

§ If fork() returns a negative value, it indicates that the creation of the process was unsuccessful.
§ fork() returns a zero to the newly created child process.
§ fork() returns a positive value, the process ID of the child process, to the parent

Basically, the fork() call, inside a process, creates an exact copy of that process somewhere else in the memory (meaning it'll copy variable values, etc...), and runs the copy from the point the call was made (for the assembly kids : it means that the relative value of the next instruction pointer is also copied)

in this example the program will find the 1st fork()which will make a copy of the program which contain another fork() that will make another copy of the program then we find the last fork that is going to make the last copy of the program but those forks (system calls) were nested then the explanation will be the last fork() copy the printf("hello world") now we have two lines of printf("hello world") these is going to be nested in the second fork() which will make a copy from the point the call was made which means a copy of the two lines of printf("hello world") now we have four lines of printf("hello world") the second fork will be nested to the first fork () that will make a copy from the point the call was made which means a copy of the four lines of printf("hello world") thus we have 8 lines of printf("hello world") to be executed in the main

——

38.Array of pts to functions — void (*fptr[10])()

it's possible but you have to have functions with the same return walue and the same arguments type like

```
int sum(int a, int b);
int subtract(int a, int b);
int mul(int a, int b);
int div(int a, int b);
```

hence you can have an array of pointers used to point those function if you want and it will be declared as `int (*p[4]) (int x, int y);`

you can set the pointer as

```
 p[0] = sum; /* address of sum() */
p[1] = subtract; /* address of subtract() */
p[2] = mul; /* address of mul() */
p[3] = div; /* address of div() */
```

if the functions were void you can have void (*ptr[4])(intx,inty);

——

39.Which way of writing infinite loops is more efficient than others?

there are 3ways

- while(1){} or while(1);
- for(;;) {} for(;;);
- Loop:
  goto Loop;

——

40.Who to know whether system uses big endian or little endian format and how to convert among them?
convert among them using arrays

——

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

If you declare an identifier, you can use it freely throughout the program. It's called as forward declaration.
If you don't declare it and using it it is called as Forward Reference, but it'll be ending with errors except the following conditions.
1.GOTO statement usage --The label you are using is declared after the usage.
2.structure,Enum,unions are used before declaration.

```
1) goto test;           /*  Forward reference to statement label --
                          legal  */
test:
 if (a > 0 ) b = TRUE;

2) struct s
  { struct t *pt };    /*  Forward reference to structure t         */
.                      /* (Note that the reference is preceded      */
.                      /* by the  struct keyword to resolve         */
.                      /* potential ambiguity)                      */
struct t
   { struct s *ps };
```

The pointer is declared before the structure it points to is defined.
The compiler can get away with this because the pointer stores an address, and you don't need to know what is at that address to reserve the memory for the pointer

——

ask)42.How is generic list manipulation function written which accepts elements of any kind?

———

43.What is the difference between embedded systems and the system in which RTOS is running?

1.normal Emabedded systems are not Real Time System
2.Systems with RTOS are realtime systems.

Embedded system can include RTOS and cannot include also. it depends on the requirement. if the system needs to serve only event sequentially, there is no need of RTOS. If the system demands the parallel execution of events then we need RTOS.

RTOS or real time operating system is designed for embedded applications. In a multitasking system, which handles critical applications operating systems must be
1.deterministic in memory allocation,
2.should allow cpu time to different threads, task, process,
3.kernel must be non-preemptive which means context switch must happen only after the end of task execution. etc

———

44.How can you define a structure with bit field members?

struct-declarator:
declarator

type-specifier declarator opt : constant-expression

The constant-expression specifies the width of the field in bits. The type-specifier for the declarator must be unsigned int, signed int, or int, and the constant-expression must be a nonnegative integer value. If the value is zero, the declaration has no declarator. Arrays of bit fields, pointers to bit fields, and functions returning bit fields are not allowed. The optional declarator names the bit field. Bit fields can only be declared as part of a structure. The address-of operator (&) cannot be applied to bit-field components.

Unnamed bit fields cannot be referenced, and their contents at run time are unpredictable. They can be used as "dummy" fields, for alignment purposes. An unnamed bit field whose width is specified as 0 guarantees that storage for the member following it in the struct-declaration-list begins on an int boundary.

Bit fields must also be long enough to contain the bit pattern
Bit fields defined as int are treated as signed. A Microsoft extension to the ANSI C standard allows char and long types (both signed and unsigned) for bit fields. Unnamed bit fields with base type long, short, or char (signed or unsigned) force alignment to a boundary appropriate to the base type.

Some bit field members are stored left to right others are stored right to left in memory.

Bit fields are allocated within an integer from least-significant to most-significant bit

In all implementations, the default integer type for a bit field is **`unsigned`**.

The maximum bit field length is 64 bits. For portability, do not use bit fields greater than 32 bits in size.

he following restrictions apply to bit fields. You cannot:

- Define an array of bit fields
- Take the address of a bit field
- Have a pointer to a bit field

- Have a reference to a bit field

```
struct mybitfields
{
    unsigned short a : 4;
    unsigned short b : 5;
    unsigned short c : 7;
} test;

int main( void );
{
    test.a = 2;
    test.b = 31;
    test.c = 0;
}
the memory will be 0000000 11111 0010

example
    struct on_off {
                    unsigned light : 1;
                    unsigned toaster : 1;
                    int count;           /* 4 bytes */
                    unsigned ac : 4;
                    unsigned : 4;
                    unsigned clock : 1;
                    unsigned : 0;
                    unsigned flag : 1;
                   } kitchen ;
example
```

The structure kitchen contains eight members totalling 16 bytes. The following table describes the storage that each member occupies:

Member Name	Storage Occupied
Light	1 bit
Toaster	1 bit
(padding -- 30 bits)	To the next int boundary
Count	The size of an int (4 bytes)
Ac	4 bits
(unnamed field)	4 bits
Clock	1 bit
(padding -- 23 bits)	To the next int boundary (unnamed field)
Flag	1 bit
(padding -- 31 bits)	To the next int boundary

we start to reserve the space for the bits in a integer boundary then when we have a zero bit which means the next bit reservation will start in a new intger boundry and padding the old one's boundary when we have a integer as a structure member then it will start from anew boundary and padding the previous one's boundary

———

45.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?

60 = 00111100
for both little endian and big endian, the answer will be same.
otherwise chose some other number e.g 204 and the answer will depend on what your cpu's architecture is.

```
#include<stdio.h>
int onoff(char, int);
int main()
{
char a = 60;
int i;
for (i=0;i<=7;i++)
printf("%d", onoff(a,i));
return 0;
}

int onoff(char byte, int field)
{
byte = byte >> field;
byte = byte & 0x01;
return(byte);
}
```

——

46.Which parameters decide the size of data type for a processor ?

The maximum width or the maximum amounts of bits a processor can deal with at the same time at a given time is specific to the particular processor you're using and would be decided by the width of it's data bus and also the width of all the data lines inside the CPU.

Essentially a CPU can operate on a certain amount of high's and low's (in terms of voltage) at any particular time, so I guess if you've got a 32bit CPU, it will have 32 physical lines in parallel on which it operates.

——

47.What is job of preprocessor, compiler, assembler and linker ?

The C PreProcessor cpp expands all those macros definitions and include statements (and anything else that starts with a #) and passes the result to the actual compiler. The preprocessor is not so interesting because it just replaces some short cuts you used in your code with more code. The output of cpp is just C code; if you didn't have any preprocessor statements in your file, you wouldn't need to run cpp. The preprocessor does not require any knowledge about the target architecture.

Compiler: This software, converts the code written in high-level language into object file. Compiler converts all the files of a given project at once "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language

Assembler: While compiler processes high level languages, assembler has the capability of processing only the low level assembly language. This assembly language is extremely core (microprocessor/platform) specific.

Linker: Linker uses the object files created by the compiler or the assembler and then uses the predefined library objects to create an executable. This executable is the one which is seen by the common user or microcontroller if the program has to run on a chip.

Interpreter: This is a software tool which interprets the user written code line by line, unlike compiler, which processes everything at once. In this case a single line is executed at a time. It is time consuming and not used in real-time applications

The machine code file is called an object file An object file is a binary representation of your program. The compiler or The assembler gives a memory location to each variable and instruction; we will see later that these memory locations are actually represented symbolically or via offsets. It also make a lists of all the unresolved references that presumably will be defined in other object file or libraries, e.g. printf. A typical object file contains the program text (instructions) and data (constants and strings), information about instructions and data that depend on absolute addresses, a symbol table of unresolved references, and possibly some debugging information

Since an object file will be linked with other object files and libraries to produce a program, the complier or the assembler cannot assign absolute memory locations to all the instructions and data in a file. Rather, it writes some notes in the object file about how it assumed things were layed out. It is the job of the linker to use these notes to assign absolute memory locations to everything and resolve any unresolved references

——

48.What is the difference between static linking and dynamic linking ?

in static linking, functions and variables which are defined in external library files are linked inside your executable. That means that the code is actually linked against your code when compiling/linking.

With dynamic linking external functions that you use in your software are not linked against your executable. Instead they reside in a external library files which are only referenced by your software. Ie: the compiler/linker instructs the software on where to find the used functions.

On windows platforms you can even explicitly load DLL files at run time and hook up the functions contained in the DLL(dynamic linking library).

DLL is an executable file and it is a shared library for functions.
It differs from static linking in that the executable module to include only the information needed at run time to locate the executable code for the functions.
In static linking, the linker gets all of the referenced functions from the static link library and places it with your code into your executable.

DLLs save memory, reduce swapping, save disk space, upgrade easier, provide after-market support, provide a mechanism to extend the MFC library classes, support Multilanguage programs, and ease the creation of international versions.

in programming static linking means that linker embeds any used libraries into your program binary, but if you use dynamic linking you will get only calls embedded and your program will use dynamic linked libraries in separate files. In Windows programming dynamic linked libraries usually have .dll extension. Using dynamic linking reduces compilation and linking time

———

49.How to implement a WD timer in software ?

I think we will have to do a loop on the first line in main beside the normal while (1) the code will be

```
int main()

{

          while(1)

          {

                    init();

                    while(1)

                    {//normal functionality

                    if (WD timer ==1) //this is a flag raised by the timer specified for the watch dog

                    // WD reset = feed value;

                              WD reset--;

                      If(WD rest ==0) // these check if the WD wasn't fed

                    break; //to out the inner loop  and got to the initialization loop if we didn't make the
//big while the break will turn the program off without turning it on again with the big while the
//system will go to the initialization and rest itself without the need to turn it off



                    }

          }

}
```

———

Notes

The va_arg() macros are used to pass a variable number of arguments to a function.

———

Q Can a global variable be hidden by a local variable with block scope?

A Yes. If a local variable shares the same name with a global variable, the global

variable can be hidden by the local variable for the scope of the block within which the local variable is defined with block scope. However, outside the block, the local variable cannot be seen, but the global variable becomes visible again.

Q Why do you need the static specifier?

A In many cases, the value of a variable is needed, even if the scope of the block, in which the variable is declared, has exited. By default, a variable with block scope has a temporary memory storage—that is, the lifetime of the variable starts when the block is executed and the variable is declared, and ends when the execution is finished. Therefore, to declare a variable with permanent duration, you have to use the static specifier to indicate to the compiler that the memory location of the variable and the value stored in the memory location should be retained after the execution of the block.

Q Does using the register specifier guarantee to improve the performance of a program?

A Not really. Declaring a variable with the register specifier only suggests to the compiler that the variable be stored in a register. But there is no guarantee that the variable will be stored in a register. The compiler can ignore the request based on the availability of registers or other restrictions.

Q When you declare a variable with the extern specifier, do you define the variable or allude to a global variable elsewhere?

A When a variable is declared with the extern specifier, the compiler considers the declaration of the variable as an allusion rather than a definition. The compiler will therefore look somewhere else to find a global variable to which the variable with extern alludes.

———————————————————————————————————————

Can static variables be declared in a header file?

You can't declare a static variable without defining it as well (this is because the storage class modifiers static and extern are mutually exclusive). A static variable can be defined in a header file

———————————————————————————————————————

When does the compiler not implicitly generate the address of the first element of an array?

When array name appears in an expression such as:

1.array as an operand of the sizeof operator.
2.array as an operand of & operator.
3.array as a string literal initializer for a character array.
Then the compiler not implicitly generate the address of the first element of an array

——

What is static memory allocation and dynamic memory allocation?

Static memory allocation: The compiler allocates the required memory space for a declared variable. By using the address of operator, the reserved address is obtained and this address may be assigned to a pointer variable. Since most of the declared variable have static memory, this way of assigning pointer value to a pointer variable is known as static memory allocation. memory is assigned during compilation time. Dynamic memory allocation: It uses functions such as malloc( ) or calloc( ) to get memory dynamically .If these functions are used to get memory dynamically and the values returned by these functions are assigned to pointer variables, such assignments are known as dynamic memory allocation. memory is assigned during run time.

——