# **Matrix row-wise and column wise**

1. Row-Major order
2. Column-Major order (vectorization)

## Reasons to use it: When you are traversing array you will have better performance, because accessing the element from contiguous order is usually faster than accessing element from 2D array.

**Example:**

1, 2, 3

4, 5, 6

**If you want to use row-major order or column-major order you must to know the exact width and height of array, because the length of array will be width \* height**

**int[][] my\_array = new int[rowsNumber \* columnsNumber]**

### **Row-Major Order**

*Element = currentRow \* columnsNumber + currentColumn*

for(int currRow = 0; currRow < rowsNumber; currRow++)

{

for(int currCol = 0 ; currCol < columnsNumber; currCol++)

printf(“%d , ”, my\_array[currRow \* columnsNumber + currCol]);

}

***The result is: 1, 2, 3, 4, 5, 6***

### **Column-Major order (vectorization)** *Element = currentColumn \* rowsNumber + currentRow*

for(int currRow = 0; currRow < rowsNumber; currRow++)

{

for(int currCol = 0; currCol < columnsNumber; currCol++)

printf(“%d, ” , my\_array[currCol \* rowsNumber + currRow])

}

***The result is: 1, 4, 2, 5, 3, 6***

**Matrix Multiplication column wise and row wise**

// Matrix A (m\*n)

// Matrix B (n\*k)

// Matrix C (m\*k)

for(i=0; i<m; i++)

{

for(j=0;j<k;j++)

{

c[i][j]=0;

for(l=0;l<n;l++)

c[i][j] += a[i][l] \* b[l][j];

}

}

In fact, there are three different dimensions involved in matrix multiplication when you multiply an MxL matrix A with an LxN matrix B to get an MxN matrix C.

L (k) | N (j)

|

| 5 2 1

L (k) |

| 6 3 7

|

-----------------+-------------

|

1 3 | 23 11 22

M (i) |

2 4 | 34 16 30

|

The letters in parentheses are the variables the code below uses to iterate over the respective dimension.

Now you can multiply your matrices in row-major format:

#define M 2

#define N 3

#define L 2

int A[M \* L] = {1, 3, 2, 4};

int B[L \* N] = {5, 2, 1, 6, 3, 7};

int res[M \* N];

int i, j, k;

for (i = 0; i < M; i++)

{

for (j = 0; j < N; j++)

{

res[j + i \* N] = 0;

for (k = 0; k < L; k++)

{

//res[currCol + currRow \* columnsNum] += A[currCol + currRow \* columnsNum] \* B[currCol + currRow \* columnsNum]

res[j + i \* N] += A[k + i \* L] \* B[j + k \* N];

}

}

}

for (i = 0; i < M \* N; i++)

printf("[%d] = %d\n", i, res[i]);

or in column-major format:

#define M 2

#define N 3

#define L 2

int A[M \* L] = {1, 2, 3, 4};

int B[L \* N] = {5, 6, 2, 3, 1, 7};

int res[M \* N];

int i, j, k;

for (i = 0; i < M; i++)

{

for (j = 0; j < N; j++)

{

res[j \* M + i] = 0;

for (k = 0; k < L; k++)

{

//res[currRow + currCol\* rowsNum] += A[currRow + currCol\* rowsNum] \* B[currRow + currCol \* rowsNum]

res[j \* M + i] += A[k \* M + i] \* B[j \* L + k];

}

}

}

for (i = 0; i < M \* N; i++)

printf("[%d] = %d\n", i, res[i]);

# **What does int argc, char \*argv[] mean?**

argv and argc are how command line arguments are passed to main() in C and C++.

argc will be the number of strings pointed to by argv. This will (in practice) be 1 plus the number of arguments.

Try the following program:

#include <iostream> int main(int argc, char\*\* argv)

{

cout << "Have " << argc << " arguments:" << endl;

for (int i = 0; i < argc; ++i)

{

cout << argv[i] << std::endl; }

}

Running it with ./test a1 b2 c3 will output

Have 4 arguments: ./test a1 b2 c3

# **Function pointers in C ??**

Let's start with a basic function which we will be *pointing to*:

int addInt(int n, int m)

{

return n+m;

}

First thing, let's define a pointer to a function which receives 2 ints and returns an int:

int (\*functionPtr)(int,int);

Now we can safely point to our function:

functionPtr = &addInt;

Now that we have a pointer to the function, let's use it:

int sum = (\*functionPtr)(2, 3); // sum == 5

Passing the pointer to another function is basically the same:

int add2to3(int (\*functionPtr)(int, int))

{

return (\*functionPtr)(2, 3);

}

**Function Pointers as Return Type ??**

We can use function pointers in return values as well (try to keep up, it gets messy):

// this is a function called functionFactory which receives parameter n // and returns a pointer to another function which receives two ints // and it returns another int

int (\*functionFactory(int n))(int, int)

{

printf("Got parameter %d", n);

int (\*functionPtr)(int,int) = &addInt;

return functionPtr;

}

But it's much nicer to use a typedef:

typedef int (\*myFuncDef)(int, int); // note that the typedef name is indeed myFuncDef

myFuncDef functionFactory(int n)

{

printf("Got parameter %d", n);

myFuncDef functionPtr = &addInt;

return functionPtr;

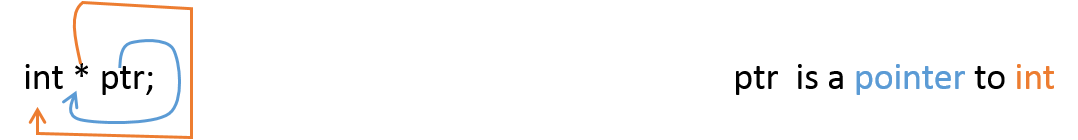
}

# **What is the difference between const int\*, const int \* const, and int const \*?**

Read it backwards (as driven by [Clockwise/Spiral Rule](http://c-faq.com/decl/spiral.anderson.html))...

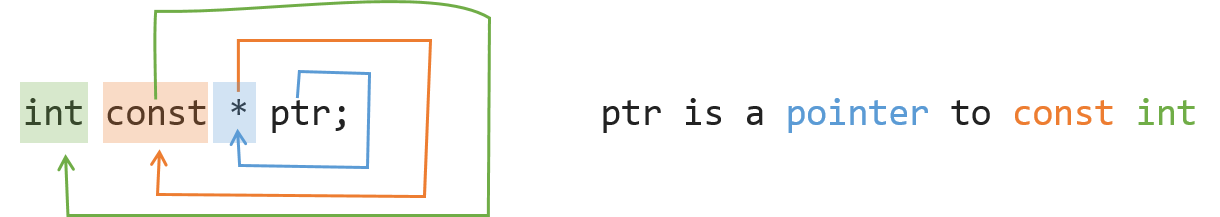
For those who don't know about Clockwise/Spiral Rule: Start from the name of the variable, move clockwisely (in this case, move backward) to the next pointer or type. Repeat until expression ends.

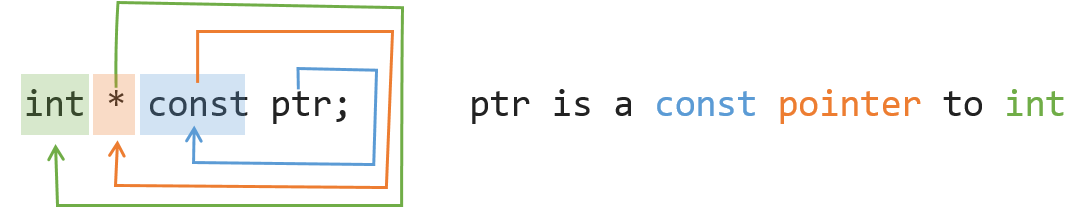
here is a demo:











* 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

**How to initialize array to same value ??**

Unless that value is 0 (in which case you can omit some part of the initializer and the corresponding elements will be initialized to 0), there's no easy way.

Don't overlook the obvious solution, though:

int myArray[10] = { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 };

Elements with missing values will be initialized to 0:

int myArray[10] = { 1, 2 }; // initialize to 1,2,0,0,0...

So this will initialize all elements to 0:

int myArray[10] = { 0 }; // all elements 0

In C++, an empty initialization list will also initialize every element to 0. This is [not allowed](https://stackoverflow.com/a/17589839) with C:

int myArray[10] = {}; // all elements 0 in C++

Remember that objects with static storage duration will initialize to 0 if no initializer is specified:

static int myArray[10]; // all elements 0

If your compiler is GCC you can use following syntax:

int array[1024] = {[0 ... 1023] = 5};

**Linked list vs Array**

Linked lists are preferable over arrays when:

a) you need constant-time insertions/deletions from the list (such as in real-time computing where time predictability is absolutely critical)

b) you don't know how many items will be in the list. With arrays, you may need to re-declare and copy memory if the array grows too big

c) you don't need random access to any elements

d) you want to be able to insert items in the middle of the list (such as a priority queue)

Arrays are preferable when:

a) you need indexed/random access to elements

b) you know the number of elements in the array ahead of time so that you can allocate the correct amount of memory for the array

c) you need speed when iterating through all the elements in sequence. You can use pointer math on the array to access each element, whereas you need to lookup the node based on the pointer for each element in linked list, which may result in page faults which may result in performance hits.

d) memory is a concern. Filled arrays take up less memory than linked lists. Each element in the array is just the data. Each linked list node requires the data as well as one (or more) pointers to the other elements in the linked list.

Array Lists (like those in .Net) give you the benefits of arrays, but dynamically allocate resources for you so that you don't need to worry too much about list size and you can delete items at any index without any effort or re-shuffling elements around. Performance-wise, arraylists are slower than raw arrays.

**How to create array of strings ??**

How to read C types

char (\*a[2])[14] = > a is an array of two pointers to an array of 14 chars

“Start at the name, read right when you can, left when you must": char (\*a[2])[14] - start at a, move right: "array of two", move left: "pointer to", bracket complete so read right: "array of forteen", read left: "char"... Put it together and we have "a is array of two pointers to arrays of forteen chars"

int N = 3;

char \*\*array = (char\*\*) malloc((N+1)\*sizeof(char\*));

array[0] = "fluffy";

array[1] = "small";

array[2] = "bunny";

array[3] = 0;

**Why use pointers to pointers ?**

If you want to have a list of characters (a word), you can use char \*word

If you want a list of words (a sentence), you can use char \*\*sentence

If you want a list of sentences (a monologue), you can use char \*\*\*monologue

If you want a list of monologues (a biography), you can use char \*\*\*\*biography

If you want a list of biographies (a bio-library), you can use char \*\*\*\*\*biolibrary

If you want a list of bio-libraries (a ??lol), you can use char \*\*\*\*\*\*lol

**Bus Error**

Bus errors are rare nowadays on x86 and occur when your processor cannot even attempt the memory access requested, typically:

* using a processor instruction with an address that does not satisfy its alignment requirements.

Segmentation faults occur when accessing memory which does not belong to your process, they are very common and are typically the result of:

* using a pointer to something that was deallocated.
* using an uninitialized hence bogus pointer.
* using a null pointer.
* overflowing a buffer.

PS: To be more precise this is not manipulating the pointer itself that will cause issues, it's accessing the memory it points to (dereferencing).

**Char \*s and char[]s ?**

What is the difference between these initializations?

char a[] = "string literal";

char \*p  = "string literal";

My program crashes if I try to assign a new value to p[i].

A: A string literal (the formal term for a double-quoted string in C source) can be used in two slightly different ways:

1. As the initializer for an array of char, as in the declaration of char a[] , it specifies the initial values of the characters in that array (and, if necessary, its size).
2. Anywhere else, it turns into an unnamed, static array of characters, and this unnamed array may be stored in read-only memory, and which therefore cannot necessarily be modified. In an expression context, the array is converted at once to a pointer, as usual (see section 6), so the second declaration initializes p to point to the unnamed array's first element.

Some compilers have a switch controlling whether string literals are writable or not (for compiling old code), and some may have options to cause string literals to be formally treated as arrays of const char (for better error catching).

**what happens free(ptr); ptr=NULL; free(ptr);  ?**

free(ptr); ptr = NULL;

See:

man free The free() function deallocates the memory allocation pointed to by ptr. If ptr is a NULL pointer, no operation is performed.

When you set the pointer to NULL after free() you can call free() on it again and no operation will be performed.

**Basic function pointer**

#include <stddef.h>

/\* Declare a simple function that takes one integer

argument and returns an integer. \*/

static int my\_function(int foo);

/\* Define a global variable named "my\_function\_pointer"

whose type is a pointer to a function that takes one

integer argument and returns an integer. Initialize

this to NULL. \*/

int (\*my\_function\_pointer)(int) = NULL;

int main(void) {

/\* Point the function pointer to the simple function \*/

my\_function\_pointer = my\_function;

/\* Invoke the function via the global function

pointer variable. \*/

return my\_function\_pointer(0);

}

/\* Define the simple "my\_function" function.\*/

static int my\_function(int foo) {

return foo;

}

**Array of pointers to functions - Function pointers**

#include <iostream>  
using namespace std;

int pass(int n)

{

return n;

}

int square(int n)

{

return n\*n;

}

int add(int n)

{

return n+n;

}

int main()

{ int num = 100;

cout << "a function pointer:" << endl;

int (\*pf)(int);

pf = pass;

cout << pf(num) << endl;

cout << "an array of function pointers:" << endl;

int (\*apf[3])(int); // Array of function pointers

apf[0] = pass; // Assigning the function pointer with the name of the function

apf[1] = square;

apf[2] = add;

cout << apf[0](num) << endl; // calling the function with function pointer

cout << apf[1](num) << endl;

cout << apf[2](num) << endl;

return 0;}

**What is functional states ? How to use function pointers to move across different functions or states ? Array of function pointers ??**

typedef enum {GEAR\_DOWN = 0, WTG\_FOR\_TKOFF, RAISING\_GEAR,

GEAR\_UP, LOWERING\_GEAR} State\_Type;

/\* This table contains a pointer to the function to call in each state.\*/

// Array of function pointers

void (\*state\_table[]) () = {GearDown, WtgForTakeoff, RaisingGear, GearUp, LoweringGear};

State\_Type curr\_state; // Enum variable

Main()

{

InitializeLdgGearSM();

/\* The heart of the state machine is this one loop.The function corresponding to the current state is called once per iteration. \*/

while (1)

{

state\_table[curr\_state]();

DecrementTimer();

/\* Do other functions, not related to this state machine.\*/

}

};

void InitializeLdgGearSM()

{

curr\_state = GEAR\_DOWN;

timer = 0.0;

/\* Stop all the hardware, turn off the lights, etc.\*/

}

void GearDown()

{

/\* Raise the gear upon command, but not if the airplane is

on the ground.\*/

if ((gear\_lever == UP) && (prev\_gear\_lever == DOWN) && (squat\_switch == UP))

timer = 2.0;

curr\_state = WTG\_FOR\_TKOFF;

prev\_gear\_lever = gear\_lever; /\* Store for edge detection.\*/

}

void RaisingGear()

{

/\* Once all 3 legs are up, go to the GEAR\_UP state.\*/

if((nosegear\_is\_up==MADE)&&(leftgear\_is\_up==MADE)&&(rtgear\_is\_up== MADE))

Curr\_state = GEAR\_UP;

/\* If the pilot changes his mind, start lowering the gear.\*/

if (gear\_lever == DOWN)

curr\_state = LOWERING\_GEAR;

}

void GearUp()

{

/\* If the pilot moves the lever to DOWN, lower the gear.\*/

if (gear\_lever == DOWN)

curr\_state = LOWERING\_GEAR;

}

void WtgForTakeoff()

{

/\* Once we’ve been airborne for 2 sec., start raising the gear.\*/

if (timer <=0.0)

curr\_state = RAISING\_GEAR;

/\*If we touch down again, or if the pilot changes his mind, start over.\*/

if ((squat\_switch ==DOWN) || (gear\_lever == DOWN))

{

timer = 2.0;

curr\_state = GEAR\_DOWN;

/\* Don’t want to require that he toggle the lever again this was just a bounce.\*/

prev\_gear\_lever = DOWN;

}

}

**Typedef Function pointer**

/\* Use typedef to declare the name "my\_function\_ptr\_t"

as an alias whose type is a pointer to a function that

takes one integer argument and returns an integer \*/

typedef int (\*my\_function\_ptr\_t)(int);

/\* Declare a simple function that takes one integer

argument and returns an integer. \*/

int my\_function(int foo);

int main(void) {

/\* Define a function pointer variable and point it to

the function. \*/

my\_function\_ptr\_t my\_function\_pointer = my\_function;

/\* Invoke the function via the function pointer. \*/

return my\_function\_pointer(0);

}

/\* Define the simple "my\_function" function.\*/

int my\_function(int foo)

{

return foo;

}

**Function pointer as Return value from function**

/\* Declare a function that takes an integer parameter and

returns a function pointer with type double(\*)(double).

This will act like an array indexing function. \*/

double (\*my\_function(int))(double);

/\* Declare a set of functions that take one double

argument and return a double. \*/

double mycbfunction1(double param);

double mycbfunction2(double param);

/\* Declare a default function which does nothing. \*/

double mycbthink(double param);

int main(void) {

/\* Call my\_function to retrieve a function pointer,

in this case, the one which squares our input. \*/

double (\*my\_function\_pointer)(double) = my\_function(0);

/\* Call the function pointer which should square the

input value of 5. \*/

return my\_function\_pointer(5);

}

/\* Define a function that returns a function pointer based

on an index. \*/

double (\*my\_function(int index))(double)

{

switch (index){

case 0: return mycbfunction1;

case 1: return mycbfunction2;

default: return mycbthink;

}

}

/\* Define a callback function that returns the square of

the provided parameter. \*/

double mycbfunction1(double param)

{

return param \* param;

}

/\* Define a callback function that returns half of the

provided parameter. \*/

double mycbfunction2(double param)

{

return param / 2;

}

/\* Define a callback function that returns the provided

parameter without any modifications. \*/

double mycbthink(double param)

{

return param;

}

#include <stdio.h>

// A normal function with an int parameter

// and void return type

void fun(int a)

{

    printf("Value of a is %d\n", a);

}

int main()

{

    // fun\_ptr is a pointer to function fun()

    void (\*fun\_ptr)(int) = &fun;

    /\* The above line is equivalent of following two

       void (\*fun\_ptr)(int);

       fun\_ptr = &fun;

    \*/

    // Invoking fun() using fun\_ptr

    (\*fun\_ptr)(10);

    return 0;

}

**Function Pointer at its best??**

Start with your declaration for f1:

int (\*f1)(float);

You want f2 to be a pointer to a function returning f1, so substitute f1 in the declaration

above with the declaration for f2:

int (\* f1 )(float);

|

+-----+-----+

| |

v v

int (\*(\*f2)(double))(float);

The declaration reads as

f2 -- f2

\*f2 -- is a pointer

(\*f2)( ) -- to a function

(\*f2)(double) -- taking a double parameter

\*(\*f2)(double) -- returning a pointer

(\*(\*f2)(double))( ) -- to a function

(\*(\*f2)(double))(float) -- taking a float parameter

int (\*(\*f2)(double))(float) -- returning int

You repeat the process for f3:

int (\*(\* f2 )(double))(float);

|

+---+----+

| |

v v

int (\*(\*(\*f3)(int))(double))(float);

which reads as

f3 -- f3

\*f3 -- is a pointer

(\*f3)( ) -- to a function

(\*f3)(int) -- taking an int parameter

\*(\*f3)(int) -- returning a pointer

(\*(\*f3)(int))( ) -- to a function

(\*(\*f3)(int))(double) -- taking a double parameter

\*(\*(\*f3)(int))(double) -- returning a pointer

(\*(\*(\*f3)(int))(double))( ) -- to a function

(\*(\*(\*f3)(int))(double))(float) -- taking a float parameter

int (\*(\*(\*f3)(int))(double))(float); -- returning int

**Object oriented programming in C??**

[**https://www.codementor.io/michaelsafyan/object-oriented-programming-in-c-du1081gw2**](https://www.codementor.io/michaelsafyan/object-oriented-programming-in-c-du1081gw2)

**Abstraction and Encapsulation ??**

**Encapsulation** means that the internal representation of an object is generally hidden from view outside of the object's definition.

Example

**Abstraction** is a mechanism which represent the essential features without including implementation details.

* **Every function is an**[**encapsulation**](https://en.wikipedia.org/wiki/Encapsulation_%28computer_programming%29); in pseudocode:

point x = { 1, 4 }

point y = { 23, 42 }

numeric d = distance(x, y)

Here, distance encapsulates the calculation of the (Euclidean) distance between two points in a plane: it hides implementation details. This is encapsulation, pure and simple.

* [**Abstraction**](https://en.wikipedia.org/wiki/Abstraction_%28software_engineering%29)**is the process of generalisation**: taking a concrete implementation and making it applicable to different, albeit somewhat related, types of data. The classical example of abstraction is C’s qsort function to sort data:

The thing about qsort is that it doesn't care about the data it sorts — in fact, it doesn’t knowwhat data it sorts. Rather, its input type is a typeless pointer (void\*) which is just C’s way of saying “I don't care about the type of data” (this is also called type erasure). The important point is that the implementation of qsort always stays the same, regardless of data type. The only thing that has to change is the compare function, which differs from data type to data type. qsort therefore expects the user to provide said compare function as a function argument.

Encapsulation and abstraction go hand in hand so much so that you could make the point that they are truly inseparable. For practical purposes, this is probably true; that said, here’s an encapsulation that’s not much of an abstraction:

class point {

numeric x

numeric y

}

We encapsulate the point’s coordinate, but we don’t materially abstract them away, beyond grouping them logically.

And here’s an example of abstraction that’s not encapsulation:

T pi<T> = 3.1415926535

Another explanation is

**Abstraction** is the concept of describing something in simpler terms, i.e abstracting away the details, in order to focus on what is important (This is also seen in abstract art, for example, where the artist focuses on the building blocks of images, such as colour or shapes). The same idea translates to OOP by using an inheritance hierarchy, where more abstract concepts are at the top and more concrete ideas, at the bottom, build upon their abstractions. At its most abstract level there is no implementation details at all and perhaps very few commonalities, which are added as the abstraction decreases.

As an example, at the top might be an interface with a single method, then the next level, provides several abstract classes, which may or may not fill in some of the details about the top level, but branches by adding their own abstract methods, then for each of these abstract classes are concrete classes providing implementations of all the remaining methods.

**Encapsulation** is a *technique*. It may or may not be for aiding in abstraction, but it is certainly about information hiding and/or organisation. It demands data and functions be grouped in some way - of course good OOP practice demands that they should be grouped by abstraction. However, there are other uses which just aid in maintainability etc.

<https://stackoverflow.com/questions/1025844/what-is-refactoring-and-what-is-only-modifying-code>

<https://stackoverflow.com/questions/2000464/loose-coupling-patterns-for-embedded-systems-programming>

<https://stackoverflow.com/questions/1403890/how-do-you-implement-a-class-in-c?rq=1>

Design patterns

Refactoring if the function is too long.

Opaque pointer in C

Design patterns in C

Adnan's classes | AdnanAziz.com 🡪 Elements of programing interview guy’s course material

Google Interview Questions

**Makefile Related Questions**

1. Explain the use of #error  
If compiler compiles this line then it shows a compiler fatal error: and stop further compilation of program.

#include<stdio.h>

#ifndef \_\_MATH\_H

#error First include then compile

#else

int main()

{

float a,b=25;

a=sqrt(b);

printf("%f",a);

return 0;

}

#endif

Output:compiler error --> Error directive :First include then compile  
  
2. Problems associated with dynamic memory allocation in typical embedded systems  
  
Following are the problems associated:

1.Sufficiency: how can we be sure that we have provided sufficient memory, so that a critical memory allocation will never be refused?

2.Garbage management: how can we be sure that memory that is no longer required is released to the memory manager at exactly the right time? If we release memory too early, we will have dangling pointers or double-free errors. If we release memory too late, we will have memory leaks. These kinds of problem plague development of large C/C++ programs.

3.Fragmentation: how can we avoid the situation in which we want to allocate N bytes of memory, but all the available memory is in fragments smaller than N, even though the total available is much larger than N? Memory fragmentation is a plague of long-running C/C++ systems that use dynamic memory extensively.

4.Timeliness: when we need to allocate memory, what is the upper bound on the time that the memory manager may take to service the request? Memory managers for C/C++ typically search freelists or more complex structures for fragments of sufficient size, therefore calls to alloc or new typically exhibit a variable and sometimes long latency.

Although the four issues of sufficiency, garbage management, fragmentation and timeliness are serious obstacles to using dynamic memory in C/C++ critical embedded systems, there are a few strategies that can mitigate them. Here are the ones I am aware of:

**2. Smart pointers in C++**

Consider the following simple C++ code with normal pointers.

|  |
| --- |
| MyClass \*ptr = new MyClass();  ptr->doSomething();  // we must do delete(ptr) to avoid memory leak |

Using [smart pointers](http://en.wikipedia.org/wiki/Smart_pointer), we can make pointers to work in way that we don’t need to explicitly call delete.

[Smart pointer](http://en.wikipedia.org/wiki/Smart_pointer) is a wrapper class over a pointer with operator like \* and -> overloaded. The objects of smart pointer class look like pointer, but can do many things that a normal pointer can’t like automatic destruction (yes, we don’t have to explicitly use delete), reference counting and more.  
The idea is to make a class with a pointer, destructor and [overloaded operators](http://geeksquiz.com/operator-overloading-c/) like \* and ->. Since destructor is automatically called when an object goes out of scope, the dynamically allocated memory would automatically deleted (or reference count can be decremented).

Consider the following simple smartPtr class.

|  |
| --- |
| #include<iostream>  using namespace std;    class SmartPtr  {     int \*ptr;  // Actual pointer  public:        // for use of explicit keyword     explicit SmartPtr(int \*p = NULL) { ptr = p; }       // Destructor     ~SmartPtr() { delete(ptr); }       // Overloading dereferencing operator     int &operator \*() {  return \*ptr; }  };    int main()  {      SmartPtr ptr(new int());      \*ptr = 20;      cout << \*ptr;      // We don't need to call delete ptr: when the object      // ptr goes out of scope, destructor for it is automatically      // called and destructor does delete ptr.      return 0;  } |

Output:

20

**Can we write one smart pointer class that works for all types?**  
Yes, we can use [templates](http://geeksquiz.com/templates-cpp/) to write a generic smart pointer class. Following C++ code demonstrates the same.

|  |
| --- |
| #include<iostream>  using namespace std;  template <class T>  class SmartPtr  {     T \*ptr;  // Actual pointer  public:  //Constructor        explicit SmartPtr(T \*p = NULL)  { ptr = p; }  //Destructor     ~SmartPtr()  { delete(ptr); }     // Overloading dereferncing operator     T & operator \* ()  {  return \*ptr; }     // Overloding arrow operator so that members of T can be accessed     // like a pointer (useful if T represents a class or struct or     // union type)     T \* operator -> () { return ptr; }  };  int main()  {      SmartPtr<int> ptr(new int());      \*ptr = 20;      cout << \*ptr;      return 0;  } |

Output:

20

Smart pointers are also useful in management of resources, such as file handles or network sockets.

**Refer USC slides pdf in the folder**

**3. What context does an interrupt handler run in**?

They run in a special context called interrupt context.

*When executing an interrupt handler or bottom half, the kernel is in interrupt context.* Recall that process context is the mode of operation the kernel is in while it is executing on behalf of a processor.

The *functions that sleep should not be called from interrupt handler***.**  
  
The processing of interrupts is split into two parts, or halves. The interrupt handler is the top half. It is run immediately upon receipt of the interrupt and performs only the work that is time critical, such as acknowledging receipt of the interrupt or resetting the hardware. Work that can be performed later is delayed until the bottom half. The bottom half runs in the future, with all interrupts enabled.   
  
Interrupt handlers are the responsibility of the driver managing the hardware. Each device has one associated driver and, if that device uses interrupts (and most do), then that driver registers one interrupt handler.

a. Drivers can register an interrupt handler and enable a given interrupt line for handling via the function.

/\* request\_irq: allocate a given interrupt line \*/

int request\_irq(unsigned int irq,

irqreturn\_t (\*handler)(int, void \*, struct pt\_regs \*),

unsigned long irqflags,

const char \*devname,

void \*dev\_id)

The first parameter, irq, specifies the interrupt number to allocate. For some devices, for example legacy PC devices such as the system timer or keyboard, this value is typically hard-coded. For most other devices, it is probed or otherwise determined programmatically and dynamically.  
  
IRQ\_FLAG:

SA\_SHIRQ This flag specifies **that the interrupt line can be shared among multiple interrupt handlers. Each handler registered on a given line must specify this flag;** otherwise, only one handler can exist per line.

SA\_INTERRUPT This flag specifies that the given interrupt handler is a fast interrupt handler. **Fast interrupt handlers run with all interrupts disabled on the local processor. This enables a fast handler to complete quickly, without possible interruption from other interrupts.** By default (without this flag), all interrupts are enabled except the interrupt lines of any running handlers, which are masked out on all processors.

The fifth parameter, dev\_id, is used primarily for shared interrupt lines. When an interrupt handler is freed (discussed later), dev\_id provides a unique cookie to allow the removal of only the desired interrupt handler from the interrupt line. Without this parameter, it would be impossible for the kernel to know which handler to remove on a given interrupt line. You can pass NULL here if the line is not shared, but you must pass a unique cookie if your interrupt line is shared

Note that request\_irq() can sleep and therefore cannot be called from interrupt context or other situations where code cannot block. It is a common mistake to call request\_irq() when it is unsafe to sleep. This is partly because of why request\_irq() can sleep: It is indeed unclear. On registration, an entry corresponding to the interrupt is created in /proc/irq.

b. Freeing an Interrupt Handler

When your driver unloads, you need to unregister your interrupt handler and potentially disable the interrupt line. To do this, call

void free\_irq(unsigned int irq, void \*dev\_id)

If the specified interrupt line is not shared, this function removes the handler and disables the line. If the interrupt line is shared, the handler identified via dev\_id is removed, but the interrupt line itself is disabled only when the last handler is removed. Now you can see why a unique dev\_id is important. With shared interrupt lines, a unique cookie is required to differentiate between the multiple handlers that can exist on a single line and allow free\_irq() to remove only the correct handler. In either case (shared or unshared), if dev\_id is non-NULL, it must match the desired handler.

A call to free\_irq() must be made from process context.

c. The following is a typical declaration of an interrupt handler:

static irqreturn\_t intr\_handler(int irq, void \*dev\_id, struct pt\_regs \*regs)  
Note that this declaration matches the prototype of the handler argument given to request\_irq(). The first parameter, irq, is the numeric value of the interrupt line the handler is servicing. This is not entirely useful today, except perhaps in printing log messages. Before the 2.0 kernel, there was not a dev\_id parameter and thus irq was used to differentiate between multiple devices using the same driver and therefore the same interrupt handler

The return value of an interrupt handler is the special type irqreturn\_t. An interrupt handler can return two special values, IRQ\_NONE or IRQ\_HANDLED. The former is returned when the interrupt handler detects an interrupt for which its device was not the originator. The latter is returned if the interrupt handler was correctly invoked, and its device did indeed cause the interrupt.   
  
Reentrancy and Interrupt Handlers

Interrupt handlers in Linux need not be reentrant. When a given interrupt handler is executing, the corresponding interrupt line is masked out on all processors, preventing another interrupt on the same line from being received. Normally all other interrupts are enabled, so other interrupts are serviced, but the current line is always disabled. Consequently, the same interrupt handler is never invoked concurrently to service a nested interrupt. This greatly simplifies writing your interrupt handler.  
  
Shared Handlers  
If any one device does not share fairly, none can share the line. When request\_irq() is called with SA\_SHIRQ specified, the call succeeds only if the interrupt line is currently not registered, or if all registered handlers on the line also specified SA\_SHIRQ. When the kernel receives an interrupt, it invokes sequentially each registered handler on the line. Therefore, it is important that the handler be capable of distinguishing whether it generated a given interrupt. The handler must quickly exit if its associated device did not generate the interrupt. This requires the hardware device to have a status register (or similar mechanism) that the handler can check. Most hardware does indeed have such a feature.

static irqreturn\_t rtc\_interrupt(int irq, void \*dev\_id, struct pt\_regs \*regs)

{

/\*

\* Can be an alarm interrupt, update complete interrupt,

\* or a periodic interrupt. We store the status in the

\* low byte and the number of interrupts received since

\* the last read in the remainder of rtc\_irq\_data.

\*/

spin\_lock (&rtc\_lock);

rtc\_irq\_data += 0x100;

rtc\_irq\_data &= ~0xff;

rtc\_irq\_data |= (CMOS\_READ(RTC\_INTR\_FLAGS) & 0xF0);

if (rtc\_status & RTC\_TIMER\_ON)

mod\_timer(&rtc\_irq\_timer, jiffies + HZ/rtc\_freq + 2\*HZ/100);

spin\_unlock (&rtc\_lock);

/\*

\* Now do the rest of the actions

\*/

spin\_lock(&rtc\_task\_lock);

if (rtc\_callback)

rtc\_callback->func(rtc\_callback->private\_data);

spin\_unlock(&rtc\_task\_lock);

wake\_up\_interruptible(&rtc\_wait);

kill\_fasync (&rtc\_async\_queue, SIGIO, POLL\_IN);

return IRQ\_HANDLED;

}

This function is invoked whenever the machine receives the RTC interrupt. First, note the spin lock calls: The first set ensures that rtc\_irq\_data is not accessed concurrently by another processor on an SMP machine, and the second set protects rtc\_callback from the same.

○ How would you get access to data you need in the interrupt handler (especially if multiple interrupts are serviced by the same handler)?

Interrupt context and process context

<http://www.linuxinternals.org/blog/2014/05/07/spinlock-implementation-in-linux-kernel/>  
  
Still searching for the actual question though ?

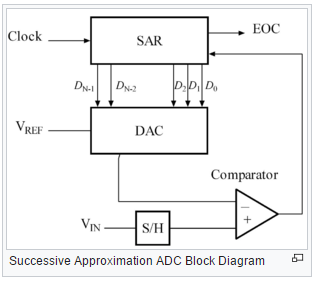
○ How would having a long-running interrupt handler adversely affect other code?

○ How would you minimize the amount of processing in an interrupt handler?

○ Peripherals often generate a very large number of interrupts. What techniques

could you use to reduce the number of interrupts seen by the processor?  
  
Apple Interview Questions Glassdoor

5. ADC? How much resolution? What interrupt to be used? Successive approximation?   
A **successive approximation ADC** is a type of [analog-to-digital converter](https://en.wikipedia.org/wiki/Analog-to-digital_converter) that converts a continuous [analog](https://en.wikipedia.org/wiki/Analog_signal) waveform into a discrete [digital](https://en.wikipedia.org/wiki/Digital_data) representation via a [binary search](https://en.wikipedia.org/wiki/Binary_search) through all possible [quantization](https://en.wikipedia.org/wiki/Quantization_(signal_processing)) levels before finally converging upon a digital output for each conversion.



**The successive approximation register is initialized so that the**[**most significant bit**](https://en.wikipedia.org/wiki/Most_significant_bit)**(MSB) is equal to a**[**digital**](https://en.wikipedia.org/wiki/Digital_data)**1. This code is fed into the DAC, which then supplies the analog equivalent of this digital code (*V*ref/2) into the comparator circuit for comparison with the sampled input voltage.** **If this analog voltage exceeds *V*in the comparator causes the SAR to reset this bit; otherwise, the bit is left a 1.**

Then the next bit is set to 1 and the same test is done, continuing this [binary search](https://en.wikipedia.org/wiki/Binary_search_algorithm) until every bit in the SAR has been tested. The resulting code is the digital approximation of the sampled input voltage and is finally output by the SAR at the end of the conversion (EOC).

10. RISC 5 stage Pipeline

Reduced Instruction Set Computer (RISC). In a RISC processor, the execution of a single instruction is split in different stages, which are chained together as a pipeline. Each instruction operates on a set of registers contained within the processor. Processor registers are used as operands or as targets for the processor instructions, and for control. For example, the ARM processor contains 17 registers: data register r0 to r14, a program counter register pc, and a processor status register cpsr. The Microblaze processor has 32 general purpose registers (r0 to r31) and up to 18 special-purpose registers (such as the program counter, the status register, and more). Each stage of a RISC pipeline takes one clock cycle to complete. A typical RISC pipeline has three or five stages, and Fig. 7.3 illustrates a five-stage pipeline. The five stages of the pipeline are called Instruction Fetch, Instruction Decode, Execute, Buffer, and Write-back. As an instruction is executed, each of the stages performs the following activities.

• *Instruction Fetch*: The processor retrieves the instruction addressed by the program counter register from the instruction memory.

• *Instruction Decode*: The processor examines the instruction opcode. For the case of a branch-instruction, the program counter will be updated. For the case of a compute-instruction, the processor will retrieve the processor data registers that are used as operands.

• *Execute*: The processor executes the computational part of the instruction on a datapath. In case the instruction will need to access data memory, the execute stage will prepare the address for the data memory.

• *Buffer*: In this stage, the processor may access the data memory, for reading or for writing. In case the instruction does not need to access data memory, the data will be forwarded to the next pipeline stage.

• *Write Back*: In the final stage of the pipeline, the processor registers are updated.

the five-stage RISC pipeline is able to accept a new instruction every clock cycle. Thus, the instruction throughput in a RISC processor may be as high as one instruction every clock cycle. Because of the pipelining, each instruction may take up to five clock cycles to complete. The instruction latency therefore can be up to five clock cycles. A RISC pipeline improves instruction throughput at the expense of instruction latency. However, the increased instruction latency of a RISC processor is usually not a problem because the clock frequency of a pipelined processor is higher than that of a non-pipelined processor.

• *Control hazards* are pipeline hazards caused by branches.

• *Data hazards* are pipeline hazards caused by unfulfilled data dependencies.

• *Structural hazards* are caused by resource conflicts and cache misses.

**Function call and Stack**

<https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/stack.html>

**Fixed-point**

A fixed point number has a specific number of bits (or digits) reserved for the integer part (the part to the left of the decimal point) and a specific number of bits reserved for the fractional part (the part to the right of the decimal point). No matter how large or small your number is, it will always use the same number of bits for each portion. For example, if your fixed point format was in decimal IIIII.FFFFF then the largest number you could represent would be 99999.99999 and the smallest would be 00000.00001. Every bit of code that processes such numbers has to have built-in knowledge of where the decimal point is.

**Floating Point**

A floating point number does not reserve a specific number of bits for the integer part or the fractional part. ***Instead it reserves a certain number of bits for the number (called the*mantissa*or*significand*) and a certain number of bits to say*where*within that number the decimal place sits*** (called the exponent). So a floating point number that took up 10 digits with 2 digits reserved for the exponent might represent a largest value of 9.9999999e+50 and a smallest value of 0.0000001e-49.

**Understanding power consumption**

Power is measured in watts and is proportional to the square of the current used in the system:

P = I2\*R

power = (current)2 \* resistance

Watts (W) = (Amps (A))2 \* Ohms (Ω)

Since your system is modeled as a resistor, if you focus on decreasing your current, you decrease your power consumption. Another way to look at power is as the product of voltage and current:

P = V \* I

power = voltage \* current

Watts (W) = Volts (V) \* Amps (A)

This is why your processor core may run at 1.8V even though other parts of your system run at 3 or 5V. Since the core takes a lot of current, the lower voltage means less power consumption. The two ways of looking at power are equivalent, according to the golden rule of electrical engineering, *Ohm's law*:

V = I \* R

voltage = current \* resistance

Volts (V) = Amps (A) \* Ohms (Ω)

**Reduce power consumption in embedded systems. ??**

Power is now the major constraint to using transistor; in the past, it was raw silicon area ν Techniques for reducing power: Turn off clock of idling FPU or cores, Dynamic Voltage-Frequency Scaling (DVFS) , Low activity periods operate at lower frequency and voltage , Low power state for DRAM, disks ν Must return to fully active mode to read/write ν Overclocking, turning off cores while keeping one in Turbo mode (for single threaded code)

Static power consumption ν Due to current leakage even when a transistor is off ν Leakage current increases in processor with smaller transistor sizes ν Powerstatic = Currentstatic x Voltage ν Scales with number of transistors ν To reduce: turn off the power supply to the inactive modules (power gating)

**Field Programmable Gate Array**

A *Field Programmable Gate Array* (FPGA) can be used to implement just about any hardware design. One use is to prototype a lump of hardware that will eventually find its way into an ASIC. However, there is nothing to say that the FPGA can’t remain in the final product, and it quite often does. Whether it does will depend on the relative weights of the development cost and production cost for a particular project, as well as the need to upgrade the hardware design after the product ships.

**What is the difference between a copy constructor and an overloaded assignment operator?**

**A:** A copy constructor constructs a new object by using the content of the argument object. An overloaded assignment operator assigns the contents of an existing object to another existing object of the same class.

**Copy Constructor**

**A:** Constructor which initializes the it's object member variables (by shallow copying) with another object of the same class. If you don't implement one in your class then compiler implements one for you. for example:

(a) Boo Obj1(10); // calling Boo constructor

(b) Boo Obj2(Obj1); // calling boo copy constructor

(c) Boo Obj2 = Obj1;// calling boo copy constructor

When are copy constructors called?

**A:** Copy constructors are called in following cases:

(a) when a function returns an object of that class by value

(b) when the object of that class is passed by value as an argument to a function

(c) when you construct an object based on another object of the same class

(d) When compiler generates a temporary object

**Copy Constructor For Linked List With Template**

//Definition of the node

template <class Type>

struct nodeType

{

Type info;

nodeType<Type> \*link;

};

**template <class Type> void linkedListType<Type>::copyList (const linkedListType<Type>& otherList)**

{

nodeType<Type> \*newNode; //pointer to create a node

nodeType<Type> \*current; //pointer to traverse the list

if (first != NULL) //if the list is nonempty, make it empty

destroyList();

if (otherList.first == NULL) //otherList is empty

{

first = NULL;

last = NULL;

count = 0;

}

else

{

current = otherList.first; //current points to the list to be copied

count = otherList.count;

//copy the first node

first = new nodeType<Type>; //create the node

first->info = current->info; //copy the info

first->link = NULL; //set the link field of the node to NULL

last = first; //make last point to the first node

current = current->link; //make current point to the next node

//copy the remaining list

while (current != NULL)

{

newNode = new nodeType<Type>; //create a node

newNode->info = current->info; //copy the info

newNode->link = NULL; //set the link of newNode to NULL

last->link = newNode; //attach newNode after last

last = newNode; //make last point to the actual last

current = current->link; //make current point to the

}//end while

}//end else

}//end copyList

**What is clock tree**

**What is IP core**

### **Soft cores**

IP cores are typically offered as **synthesizable**[**RTL**](https://en.wikipedia.org/wiki/Register-transfer_level). Synthesizable cores are **delivered in a**[**hardware description language**](https://en.wikipedia.org/wiki/Hardware_description_language)**such as**[**Verilog**](https://en.wikipedia.org/wiki/Verilog) or [VHSIC hardware description language](https://en.wikipedia.org/wiki/VHSIC_hardware_description_language) (VHDL). These are analogous to [high level languages](https://en.wikipedia.org/wiki/High-level_programming_language) such as C in the field of computer programming. **IP cores delivered to chip makers as RTL permit chip designers to modify designs (at the functional level),** though many IP vendors offer no warranty or support for modified designs.

IP cores are also sometimes offered as generic gate-level [netlists](https://en.wikipedia.org/wiki/Netlist). The netlist is a boolean-algebra representation of the IP's logical function implemented as generic [gates](https://en.wikipedia.org/wiki/Logic_gate) or [process](https://en.wikipedia.org/wiki/Semiconductor_device_fabrication) specific [standard cells](https://en.wikipedia.org/wiki/Standard_cell).

An IP core implemented as generic gates is portable to any process technology. A gate-level netlist is analogous to an assembly-code listing in the field of computer programming. A netlist gives the IP core vendor reasonable protection against reverse engineering.

Both netlist and synthesizable cores are called "soft cores", as both allow a synthesis, placement and route ([SPR](https://en.wikipedia.org/wiki/Integrated_circuit_design#Physical_design)) design flow.

### **Hard cores**

Such cores, whether analog or digital, are called "hard cores" (or hard macros), because the **core's application function cannot be meaningfully modified by chip designers**

**11.What is Volatile?**To declare a variable volatile, include the keyword volatile before or after the data type in the variable definition. For instance both of these declarations will declare foo to be a volatile integer:

volatile int foo;

int volatile foo;

Now, it turns out that pointers to volatile variables are very common, especially with memory-mapped I/O registers. Both of these declarations declare pReg to be a pointer to a volatile unsigned 8-bit integer:

volatile uint8\_t \* pReg;

uint8\_t volatile \* pReg;

Volatile pointers to non-volatile data are very rare (I think I've used them once), but I'd better go ahead and give you the syntax:

int \* volatile p;  
  
Finally, if you apply volatile to a struct or union, the entire contents of the struct/union are volatile. If you don't want this behavior, you can apply the volatile qualifier to the individual members of the struct/union.

Proper Use of C's volatile Keyword

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

1. Memory-mapped peripheral registers

2. Global variables modified by an interrupt service routine

3. Global variables accessed by multiple tasks within a multi-threaded application

Example:

**uint32\_t volatile \*pGpio0Set = (uint32\_t volatile \*)(0x40E00018);**

//Writing in the register

void gpioFunction(void)

{

\*pGpio0Set = 1; /\* First write. \*/

\*pGpio0Set = 2; /\* Second write. \*/

}

//Reading from register

Uint8\_t read (

{

Uint8\_t data = \*pGpioSet

return data;

}

**Constant Qualifier**

The qualifier const can be applied to the declaration of any variable to specify that its value will not be changed (Which depends upon where const variables are stored, we may change value of const variable by using pointer). The result is implementation-defined if an attempt is made to change a const.

**1) Pointer to variable.**

|  |
| --- |
| int \*ptr; |

We can change the value of ptr and we can also change the value of object ptr pointing to. Pointer and value pointed by pointer both are stored in read-write area.

**2) Pointer to constant.**  
Pointer to constant can be declared in following two ways.

|  |
| --- |
| const int \*ptr; |

or

|  |
| --- |
| int const \*ptr; |

We can change pointer to point to any other integer variable, but cannot change value of object (entity) pointed using pointer ptr. Pointer is stored in read-write area (stack in present case).

|  |
| --- |
| #include <stdio.h>  int main(void)  {      int i = 10;      int j = 20;      const int \*ptr = &i;    /\* ptr is pointer to constant \*/      printf("ptr: %d\n", \*ptr);      \*ptr = 100;        /\* error: object pointed cannot be modified using the pointer ptr \*/        ptr = &j;          /\* valid \*/      printf("ptr: %d\n", \*ptr);        return 0;  } |

Output:

error: assignment of read-only location ‘\*ptr’

**3) Constant pointer to variable.**

|  |
| --- |
| int \*const ptr; |

Above declaration is constant pointer to integer variable, means we can change value of object pointed by pointer, but cannot change the pointer to point another variable.

|  |
| --- |
| #include <stdio.h>    int main(void)  {     int i = 10;     int j = 20;     int \*const ptr = &i;    /\* constant pointer to integer \*/     printf("ptr: %d\n", \*ptr);     \*ptr = 100;    /\* valid \*/     printf("ptr: %d\n", \*ptr);     ptr = &j;        /\* error \*/     return 0;  } |

Output:

error: assignment of read-only variable ‘ptr’

**4) constant pointer to constant**

|  |
| --- |
| const int \*const ptr; |

Above declaration is constant pointer to constant variable which means we cannot change value pointed by pointer as well as we cannot point the pointer to other variable. Let us see with example.

|  |
| --- |
| #include <stdio.h>    int main(void)  {      int i = 10;      int j = 20;      const int \*const ptr = &i;        /\* constant pointer to constant integer \*/        printf("ptr: %d\n", \*ptr);        ptr = &j;            /\* error \*/      \*ptr = 100;        /\* error \*/        return 0;  } |

Output:

error: assignment of read-only variable ‘ptr’

error: assignment of read-only location ‘\*ptr’

**12.What are bitfields?**

A *bitfield* is a **field of one or more bits within a larger integer value**. Bitfields are useful for bit manipulations and are supported within a struct by C language compilers.

struct

{

uint8\_t bit0 : 1;

uint8\_t bit1 : 1;

uint8\_t bit2 : 1;

uint8\_t bit3 : 1;

uint8\_t nibble : 4;

} foo;

**Bits within a bitfield can be individually** set, tested, cleared, and toggled without affecting the state of the other bits outside the bitfield.

To test bits using the bitfield, use code such as the following:

if (foo.bit0)

{

/\* Do other stuff. \*/

}

Here’s how to test a wider field (such as two bits) using a bitfield:

if (foo.nibble == 0x03)

{

/\* Do other stuff. \*/

}

To set a bit using a bitfield, use this code:

foo.bit1 = 1;

And to toggle a bit using the bitfield, use this:

foo.bit3 = ~foo.bit3; /\* or !foo.bit3 \*/

There are some issues you must be aware of should you decide to use bitfields. Bitfields are not portable; some compilers start from the least significant bit, while others start from the most significant bit. In some cases, the compiler may require enclosing the bitfield within a union; doing this makes the bitfield code portable across ANSI C compilers.

union

{

uint8\_t byte;

struct

{

uint8\_t bit0 : 1;

uint8\_t bit1 : 1;

uint8\_t bit2 : 1;

uint8\_t bit3 : 1;

uint8\_t nibble : 4;

} bits;

} foo;

Instead of accessing only individual bits, the register can be written to as a whole.

For example, the bitfield union, along with bitmasks, can be useful when initializing a register, as shown here:

foo.byte = (TIMER\_COMPLETE | TIMER\_ENABLE);

while individual bits are still accessible, as shown here:

foo.bits.bit2 = 1;

**Struct Overlays**

In embedded systems featuring memory-mapped I/O devices, it is sometimes useful to overlay a C struct onto each peripheral’s control and status registers. Benefits of struct overlays are that you can read and write through a pointer to the struct, the register is described nicely by the struct, code can be kept clean, and the compiler does the address construction at compile time.

The following example code shows a struct overlay for a timer peripheral. If a peripheral’s registers do not align correctly, reserved members can be included in the struct.

*typedef struct*

*{*

*uint16\_t count; /\* Offset 0 \*/*

*uint16\_t maxCount; /\* Offset 2 \*/*

*uint16\_t \_reserved1; /\* Offset 4 \*/*

*uint16\_t control; /\* Offset 6 \*/*

*} volatile timer\_t;*

***timer\_t \*pTimer = (timer\_t \*)(0xABCD0123);***

Note that the individual fields of a struct, as well as the entire struct, can be declared volatile.

**Addressing method for peripherals**

#define \_\_I volatile /\*!< defines 'read only' permissions \*/

#else

#define \_\_I volatile const /\*!< defines 'read only' permissions \*/

#endif

#define \_\_O volatile /\*!< defines 'write only' permissions \*/

#define \_\_IO volatile /\*!< defines 'read / write' permissions \*/

typedef struct

{

union {

\_\_I uint8\_t RBR;

\_\_O uint8\_t THR;

\_\_IO uint8\_t DLL;

uint32\_t RESERVED0;

};

union {

\_\_IO uint8\_t DLM;

\_\_IO uint32\_t IER;

};

union {

\_\_I uint32\_t IIR;

\_\_O uint8\_t FCR;

};

\_\_IO uint8\_t LCR;

uint8\_t RESERVED1[7];

\_\_I uint8\_t LSR;

uint8\_t RESERVED2[7];

\_\_IO uint8\_t SCR;

uint8\_t RESERVED3[3];

\_\_IO uint32\_t ACR;

\_\_IO uint8\_t ICR;

uint8\_t RESERVED4[3];

\_\_IO uint8\_t FDR;

uint8\_t RESERVED5[7];

\_\_IO uint8\_t TER;

uint8\_t RESERVED6[39];

\_\_IO uint32\_t FIFOLVL;

} LPC\_UART0\_TypeDef;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Peripheral memory map \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Base addresses \*/

#define LPC\_FLASH\_BASE (0x00000000UL)

#define LPC\_RAM\_BASE (0x10000000UL)

#define LPC\_GPIO\_BASE (0x2009C000UL)

**#define LPC APB0\_BASE (0x40000000UL) \_\_\_**

#define LPC\_APB1\_BASE (0x40080000UL)

#define LPC\_AHB\_BASE (0x50000000UL)

#define LPC\_CM3\_BASE (0xE0000000UL)

/\* APB0 peripherals \*/

**#define LPC\_UART0\_BASE (LPC\_APB0\_BASE + 0x0C000)**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Peripheral declaration \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**#define LPC\_UART0 ((LPC\_UART0\_TypeDef \*) LPC\_UART0\_BASE )**

Each register can store 32 bit data and each register has 32 bit address. All the members of the structure are aligned on a word boundary.

**Writing value to any of the register of LPC\_UART0**

LPC\_UART0->LCR = 0x002;

**Reading the value from register**

uint32\_t data;

Data = LPC\_UART0->LCR;

**13.What is Enum?**

Enumeration (or enum) is a user defined data type in C. **It is mainly used to assign names to integral constants**, the names make a program easy to read and maintain.

Interesting facts about initialization of enum.

1. Two enum names can have same value.

enum State {Working = 1, Failed = 0, Freezed = 0};

2. If we do not explicitly assign values to enum names, the compiler by default assigns values starting from 0. For example, in the following C program, sunday gets value 0, monday gets 1, and so on.

enum day {sunday, monday, tuesday, wednesday, thursday, friday, saturday};

3. We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.

enum day {sunday = 1, monday, tuesday = 5, wednesday, thursday = 10, friday, saturday};

4. The value assigned to enum names must be some integral constant, i.e., the value must be in range from minimum possible integer value to maximum possible integer value.

5. All enum constants must be unique in their scope. For example, the following program fails in compilation.

enum state {working, failed};

enum result {failed, passed};

**What is a difference between structure and Union?**

A structure is a user-defined data type available in C that allows to combining data items of different kinds.

struct [structure name]

{

member definition;

member definition;

...

member definition;

};

[**union**](http://www.geeksforgeeks.org/union-c/)

A union is a special data type available in C that allows storing different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple purposes.

union [union name]

{

member definition;

member definition;

...

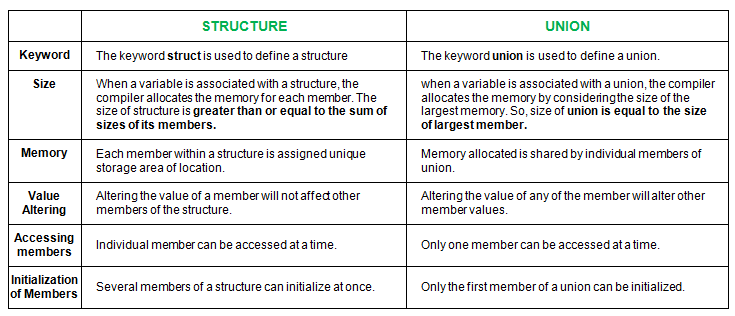
member definition;

};

**Similarities between Structure and Union**

1. Both are user-defined data types used to store data of different types as a single unit.
2. Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.
3. Both structures and unions support only assignment = and sizeof operators. The two structures or unions in the assignment must have the same members and member types.
4. A structure or a union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable as a corresponding parameter.
5. **‘.’** operator is used for accessing members.

**Differences**



**What is a difference between #define and constant**

**#define** is a [preprocessor directive](http://www.geeksforgeeks.org/interesting-facts-preprocessors-c/). Things defined by #define are replaced by the preprocessor before compilation begins.

**const** variables are actual variables like other normal variable.

The big advantage of const over #define is type checking. We can also have pointers to const variables, we can pass them around, typecast them and any other thing that can be done with a normal variable. One disadvantage that one could think of is extra space for variable which is immaterial due to optimizations done by compilers.

In general const is a better option if we have a choice. There are situations when #define cannot be replaced by const. For example, #define can take parameters (See [this](http://www.geeksforgeeks.org/interesting-facts-preprocessors-c/)for example). #define can also be used to replace some text in a program with another text.

**#define Macros**

**#pragma**

In the C and C++ programming languages, #**pragma once** is a non-standard but widely supported preprocessor directive **designed to cause the current source file to be included only once in a single compilation**.

## Example[[edit](https://en.wikipedia.org/w/index.php?title=Pragma_once&action=edit&section=1)]

**File "grandparent.h"**

#pragma once

**struct** foo

{

int member;

};

**File "parent.h"**

#include *"grandparent.h"*

**File "child.c"**

#include *"grandparent.h"*

#include *"parent.h"*

The most common alternative to #pragma once is to use #define to set an [include guard](https://en.wikipedia.org/wiki/Include_guard) macro, the name of which is picked by the programmer to be unique to that file. For example,

#ifndef GRANDPARENT\_H

#define GRANDPARENT\_H

... contents of grandparent.h

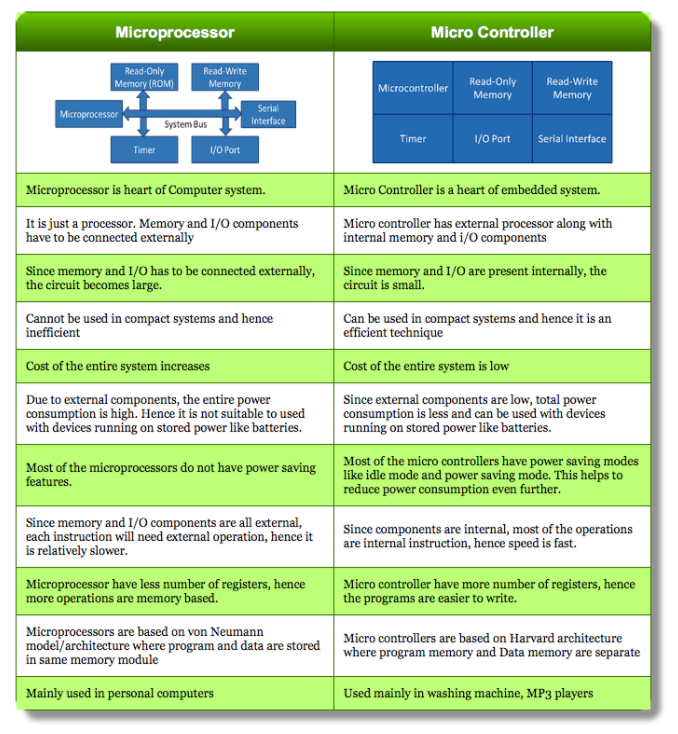
#endif */\* !GRANDPARENT\_H \*/*

16.**Hard real time systems and soft real time systems**

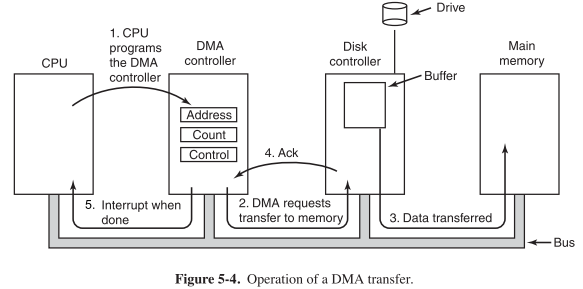
1> For a Hard real-time system, **if the system fails to meet the deadline even once** the system is considered to have Failed.

2> For a Soft real-time system, **even if the system fails to meet the deadline, possibly more than once** (i.e. for multiple requests), the system is not considered to have failed. But, in this case the results of the requests are not worthless value for a result after its deadline, is not zero, rather it degrades as time passes after the deadline. Eg.: Streaming audio-video

**18.Microprocessor Vs MicroController**



19.DMA



**Direct memory access** (**DMA**) is a feature of computer systems that allows certain hardware subsystems to access main system memory (RAM), independent of the central processing unit (CPU). **With DMA, the CPU first initiates the transfer,** then it does other operations while the transfer is in progress, and **it finally receive an interrupt from the DMA controller when the operation is done.**

This feature is useful at any time that the CPU cannot keep up with the rate of data transfer, or when the CPU needs to perform useful work while waiting for a relatively slow I/O data transfer. Many hardware systems use DMA, including disk drive controllers, graphics cards, network cards and sound cards. DMA is also used for intra-chip data transfer in multi-core processors. Computers that have DMA channels can transfer data to and from devices with much less CPU overhead than computers without DMA channels. Similarly, a processing element inside a multi-core processor can transfer data to and from its local memory without occupying its processor time, allowing computation and data transfer to proceed in parallel.

**Cache coherency and DMA**

*Cache incoherence due to DMA*

DMA can lead to cache coherency problems. Imagine a CPU equipped with a cache and an external memory that can be accessed directly by devices using DMA. When the CPU accesses location X in the memory, the current value will be stored in the cache. Subsequent operations on X will update the cached copy of X, but not the external memory version of X, assuming a write-back cache. **If the cache is not flushed to the memory before the next time a device tries to access X, the device will receive a stale value of X. Similarly, if the cached copy of X is not invalidated when a device writes a new value to the memory, then the CPU will operate on a stale value of X.**

This issue can be addressed in one of two ways in system design: Cache-coherent systems implement a method in hardware whereby external writes are signaled to the cache controller which then performs **a cache invalidation for DMA writes or cache flush for DMA reads.**

**Modes of operation DMA**

**2.1 Burst mode**

**An entire block of data is transferred in one contiguous sequence.** Once the DMA controller is granted access to the system bus by the CPU, **it transfers all bytes of data in the data block before releasing control of the system buses back to the CPU,** but renders the CPU inactive for relatively long periods of time. The mode is also called “Block Transfer Mode”. It is also used to stop unnecessary data.

**2.2 Cycle stealing mode**

The *cycle stealing mode* is used in systems in which the **CPU should not be disabled for the length of time needed for burst transfer modes.**

In the cycle stealing mode, the DMA controller obtains access to the system bus the same way as in burst mode, using **BR (Bus Request) and BG (Bus Grant) signals**, which are the two signals **controlling the interface between the CPU and the DMA controller.** However, in cycle stealing mode, **after one byte of data transfer, the control of the system bus is de-asserted to the CPU via BG. It is then continually requested again via BR, transferring one byte of data per request, until the entire block of data has been transferred.**

By continually obtaining and releasing the control of the system bus, **the DMA controller essentially interleaves instruction and data transfers. The CPU processes an instruction, then the DMA controller transfers one data value, and so on**. On the one hand, the data block is not transferred as quickly in cycle stealing mode as in burst mode, but on the other hand the CPU is not idled for as long as in burst mode.

**Cache Coherency:**

In [computer architecture](https://en.wikipedia.org/wiki/Computer_architecture), **cache coherence** **is the uniformity of shared resource data that ends up stored in multiple**[**local caches**](https://en.wikipedia.org/wiki/Cache_(computing))**.** When clients in a system maintain [caches](https://en.wikipedia.org/wiki/CPU_cache) of a common memory resource, problems may arise with incoherent data, which is particularly the case with [CPUs](https://en.wikipedia.org/wiki/Central_processing_unit) in a [multiprocessing](https://en.wikipedia.org/wiki/Multiprocessing) system.

**In a**[**shared memory**](https://en.wikipedia.org/wiki/Shared_memory_architecture)**multiprocessor system with a separate cache memory for each processor, it is possible to have many copies of shared data: one copy in the main memory and one in the local cache of each processor that requested it**. When one of the copies of data is changed, the other copies must reflect that change. Cache coherence is the discipline which ensures that the changes in the values of shared operands(data) are propagated throughout the system in a timely fashion.

The following are the requirements for cache coherence:

* **Write Propagation: Changes to the data in any cache must be propagated to other copies (of that cache line) in the peer caches.**
* **Transaction Serialization: Reads/Writes to a single memory location must be seen by all processors in the same order.**

Theoretically, coherence can be performed at the load/store [granularity](https://en.wikipedia.org/wiki/Granularity). However, in practice it is generally performed at the granularity of cache blocks

The two most common mechanisms of ensuring coherency are [*snooping*](https://en.wikipedia.org/wiki/Bus_sniffing) and [*directory-based*](https://en.wikipedia.org/wiki/Directory-based_cache_coherence), each having their own benefits and drawbacks.

Snooping based protocols tend to be faster, if enough [bandwidth](https://en.wikipedia.org/wiki/Memory_bandwidth) is available, since all transactions are a request/response seen by all processors. The drawback is that snooping isn't scalable. Every request must be broadcast to all nodes in a system, meaning that as the system gets larger, the size of the (logical or physical) bus and the bandwidth it provides must grow. Directories, on the other hand, tend to have longer latencies (with a 3 hop request/forward/respond) but use much less bandwidth since messages are point to point and not broadcast. For this reason, many of the larger systems (>64 processors) use this type of cache coherence.

### [**Snooping**](https://en.wikipedia.org/wiki/Bus_sniffing)

First introduced in 1983,[[7]](https://en.wikipedia.org/wiki/Cache_coherence#cite_note-7) snooping is a process where the individual caches monitor address lines for accesses to memory locations that they have cached.[[4]](https://en.wikipedia.org/wiki/Cache_coherence#cite_note-:3-4) The ***write invalidate protocols*** and ***write update protocols*** make use of this mechanism.

For the snooping mechanism, a snoop filter reduces the snooping traffic by maintaining a plurality of entries, each representing a cache line that may be owned by one or more nodes. When replacement of one of the entries is required, the snoop filter selects for replacement the entry representing the cache line or lines owned by the fewest nodes, as determined from a presence vector in each of the entries. A temporal or other type of algorithm is used to refine the selection if more than one cache line is owned by the fewest number of nodes.

### **Directory-based**

In a directory-based system, the data being shared is placed in a common directory that maintains the coherence between caches. The directory acts as a filter through which the processor must ask permission to load an entry from the primary memory to its cache. When an entry is changed, the directory either updates or invalidates the other caches with that entry.

[Distributed shared memory](https://en.wikipedia.org/wiki/Distributed_shared_memory) systems mimic these mechanisms in an attempt to maintain consistency between blocks of memory in loosely coupled systems.[[9]](https://en.wikipedia.org/wiki/Cache_coherence#cite_note-9)

## Coherence protocols[[edit](https://en.wikipedia.org/w/index.php?title=Cache_coherence&action=edit&section=6)]

Coherence Protocols apply cache coherence in multiprocessor systems. The intention is that two clients must never see different values of the same shared data.

The protocol must implement the basic requirements for coherence. It can be tailor made for the target system/application.

Protocols can also be classified as Snooping(Snoopy/Broadcast) or Directory based. Typically, early systems used directory based protocols where a directory would keep a track of the data being shared and the sharers. In Snoopy protocols , the transaction request. (read/write/upgrade) are sent out to all processors. All processors snoop the request and respond appropriately.

Write Propagation in Snoopy protocols can be implemented by either of the following :

* **Write Invalidate** : when a write operation is observed to a location that a cache has a copy of, the cache controller invalidates its own copy of the snooped memory location.[[4]](https://en.wikipedia.org/wiki/Cache_coherence#cite_note-:3-4)
* **Write Update** : when a write operation is observed to a location that a cache has a copy of, the cache controller updates its own copy of the snooped memory location with the new data.

If the protocol design states that whenever any copy of the shared data is changed, all the other copies must be "updated" to reflect the change, then it is a ***write update*** protocol. If the design states that on a write to a cached copy by any processor requires other processors to discard/invalidate their cached copies, then it is a ***write invalidate*** protocol.

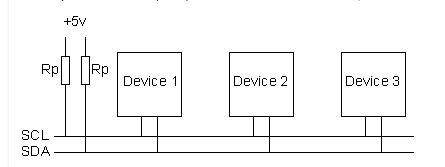
However, scalability is one shortcoming of broadcast protocols.

##### **Garbage collection**

Garbage collection is **a strategy for automatically detecting memory allocated to objects that are no longer usable in a program, and returning that allocated memory to a pool of free memory locations.** This method is in contrast to "manual" memory management where a programmer explicitly codes memory requests and memory releases in the program. **While automatic garbage has the advantages of reducing programmer workload and preventing certain kinds of memory allocation bugs**, garbage collection does require memory resources of its own, and can compete with the application program for processor time.

**Communication Protocols**

**I2C**



I2C requires a mere two wires, like **asynchronous serial**, but those two wires can support up to 127 slave devices

7 bit or 10 bit addressing mode (27 – 1).

Also, unlike SPI, **I2C can support a multi-master system**, allowing more than one master to communicate with all devices on the bus (although the master devices can’t talk to each other over the bus and must take turns using the bus lines).

Data rates fall between asynchronous serial and SPI; most I2C devices can communicate at **100kHz or 400kHz**.

There is some overhead with I2C**; for every 8 bits of data to be sent, one extra bit of meta data (the “ACK/NACK” bit, which we’ll discuss later) must be transmitted**.

This is just two wires, called **SCL(clock) and SDA (Data)**.

**SCL is used to synchronize all data transfers** over the I2C bus.

The **SCL & SDA lines are connected to all devices** on the I2C bus.

**Both SCL and SDA** lines are "**open drain" drivers**. What this means is that the chip can drive its output low, but it cannot drive it high. **For the line to be able to go high you must provide pull-up resistors to the 5v supply**. There should be a resistor from the SCL line to the 5v line and another from the SDA line to the 5v line. You **only need one set of pull-up resistors** for the whole I2C bus, not for each device,

The value of the resistors is not critical. I have seen anything from 1k8 (1800 ohms) to 47k (47000 ohms) used. 1k8, 4k7 and 10k are common values, **Using a smaller pull-up can achieve higher speeds, but then each device must have the capability of sinking that much more current**. For example, with a 5v BUS, and 1K pull-up, each device must be able to sink 5mA.

**What is the purpose of pull up resistor in i2C?**

**How capacitance and I2C are related?**

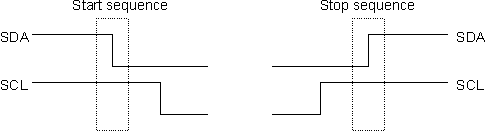
The **pull-up resistors along with the capacitance** of the cabling or bus **creates a charging RC time constant**. If using off board cabling or very long buses, **the total bus capacitance increases, which increases the rise time of the signal and reduces maximum operating frequency**.

**The total bus capacitance also increases with the number of devices connected to the bus.**

**Details**

The devices on the I2C bus are either masters or slaves. **The master is always the device that drives the SCL clock line.** The slaves are the devices that respond to the master. **A slave cannot initiate a transfer over the I2C bus, only a master can do that.** There can be, and usually are, multiple slaves on the I2C bus, however there is normally only one master. It is possible to have multiple masters, but it is unusual and not covered here.

**The I2C Physical Protocol**  
When the master wishes to talk to a slave it **begins by issuing a start sequence** on the I2C bus. The start sequence and stop sequence are special in that these are the only places where the SDA (data line) can change while the SCL (clock line) is high. **When data is being transferred, SDA must remain stable and not change whilst SCL is high.**



Data is transferred in sequences of 8 bits. The bits are placed on the SDA line starting with the MSB (Most Significant Bit).

**For every 8 bits** transferred, **the device receiving** the data **sends back an acknowledge** bit, so there are actually **9 SCL clock pulses to transfer each 8 bit byte of data.**

If the receiving device **sends back a low ACK bit, then it has received the data and is ready to accept another byte**.

If it sends back a **high ACK bit then it is indicating it cannot accept any further data** and the master should terminate the transfer by sending a stop sequence.

http://www.robot-electronics.co.uk/images/i2cc.GIF

**How fast?**  
**The standard clock (SCL) speed for I2C up to 100KHz.** Philips do define faster speeds: **Fast mode**, which is up to **400KHz** and **High Speed** mode which is up to **3.4MHz.**

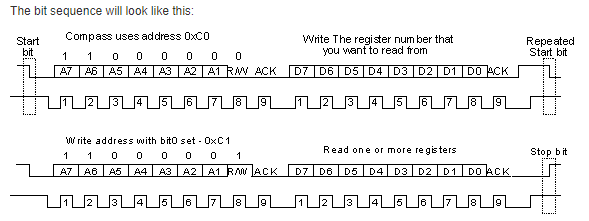
**I2C Device Addressing?**  
All I2C addresses are either 7 bits or 10 bits. The use of 10 bit addresses is rare and is not covered here.

**When sending out the 7bit address, we still always send 8 bits. The extra bit is used to inform the slave if the master is writing to it or reading from it**.

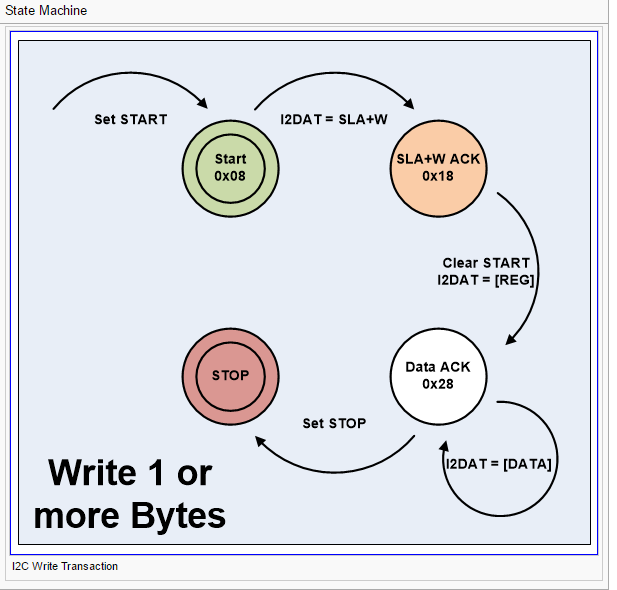
If the **bit is 0,** the master is **writing to the slave**.

If the **bit is 1,** the master is **reading from the slave.**

The **7bit address** is placed in the **upper 7 bits of the byte and the Read/Write (R/W) bit is in the LSB (Least Significant Bit). (MSB first and LSB last)**



So the read/write bit just makes it an odd/even address.



#### 

#### **Clock stretching using SCL**

One of the more significant features of the I²C protocol is **clock stretching**. An **addressed slave device** may **hold the clock line (SCL) low after receiving (or sending) a byte, indicating that it is not yet ready to process more data.** The master that is communicating with the slave may not finish the transmission of the current bit, but **must wait until the clock line actually goes high**. **If the slave is clock-stretching, the clock line will still be low.**

The master must wait until it observes the clock line going high, and an additional minimal time (4 μs for standard 100 kbit/s I²C) before pulling the clock low again.

#### **Arbitration using SDA (Important)**

**Every master monitors the bus for start and stop bits** and does not start a message while another master is keeping the bus busy. However, **two masters may start transmission at about the same time**; in this case, **arbitration occurs**. If one transmitter sets SDA to 1 (not driving a signal) and a second transmitter sets it to 0 (pull to ground), the result is that the line is low. The first transmitter then observes that the level of the line is different from that expected and concludes that another node is transmitting. The first node to notice such a difference is the one that loses arbitration: it stops driving SDA. If it is a master, it also stops driving SCL and waits for a STOP; then it may try to reissue its entire message. In the meantime, the other node has not noticed any difference between the expected and actual levels on SDA and therefore continues transmission. It can do so without problems because so far the signal has been exactly as it expected; no other transmitter has disturbed its message.

**If the two masters are sending a message to two different slaves, the one sending the lower slave address always "wins" arbitration in the address stage.** Since the two masters may send messages to the same slave address, and addresses sometimes refer to multiple slaves, arbitration must continue into the data stages.

**Bit Banging:**

**Bit banging** is a **technique for**[**serial communications**](https://en.wikipedia.org/wiki/Serial_communications)**using software instead of dedicated hardware**. Software directly **sets and**[**samples**](https://en.wikipedia.org/wiki/Sampling_(signal_processing))**the state of pins on the**[**microcontroller**](https://en.wikipedia.org/wiki/Microcontroller), and is responsible for all parameters of the signal: timing, levels, synchronization, etc.

Some embedded systems don’t have hardware dedicated to performing all of the interface functions of a serial interface. In this case, general-purpose I/O signals are connected to external devices, and it is up to the software to implement the communication protocol. *Bit banging* is a slang term for the process of transferring serial data under software control. Bit banging can be used for any serial interface, including I2C, SPI, and even UARTs. When implementing a serial interface via bit banging, the software controls all of the signals to operate the interface.

However, bit-banging carries a *software overhead* consuming CPU cycles that you could otherwise utilize for other purposes. This may have a noticeable effect on system responsiveness to other events, and in a *hard real-time system*, may significantly impact the system’s ability to meet real-time deadlines

**SPI**

SPI stands for **S**erial **P**eripheral **B**us. It is **a high-speed, full-duplex** bus that uses minimum of 3 wires to exchange data. The popularity of this bus rose when SD cards (and its variants ie: micro-sd) officially supported this bus according to the SD specifications. Furthermore, unlike UART in which you can only have one transmitter and a receiver, SPI bus can have one master and multiple slave devices.

### **SPI Bus Signals**

|  |  |
| --- | --- |
| * MOSI --> Master Out Slave In (driven by master) * MISO --> Master In Slave Out (driven by slave) * SCK  --> Clock (driven by master) * CS   --> Chip-select signal, one per slave | [http://www.socialledge.com/sjsu/images/9/98/Spi_tutorial_conns.jpg](http://www.socialledge.com/sjsu/index.php?title=File:Spi_tutorial_conns.jpg)  SPI BUS Connections |

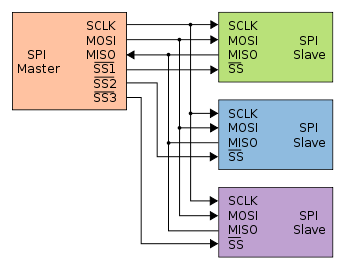
**The CS signal selects one slave, and the slave takes over the MISO pin. If a slave is not selected, then it shall leave the MISO pin in hi-z state.**

If multiple slaves have their CS signal asserted, they will try to take control of the MISO pin and damage their MISO pins.

For example, if one slave drives the signal high (connect to 3.3v) and the other drives it low (connect to ground), then short-circuit will occur damaging this pin.

**The SCK signal can reach speed of 24Mhz** and beyond, however, SD cards are usually limited to 24Mhz according to the specifications. Furthermore, any signal over 24Mhz on a PCB requires special design consideration to make sure it will not deteriorate, thus 24Mhz is the usual maximum. Furthermore, you need a CPU twice as fast as the speed you wish to run to support it. For example, to run at 24Mhz SPI, we need 48Mhz CPU or higher. Because each wire is driven directly (rather than open-collector), higher speeds can be attained compared to 400Khz I2C bus.

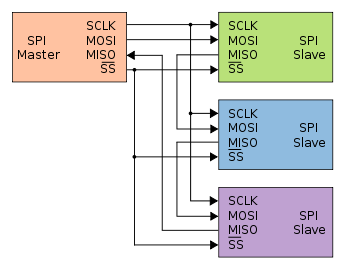
### Independent slave configuration

[](https://en.wikipedia.org/wiki/File:SPI_three_slaves.svg)

Typical SPI bus: master and three independent slaves

In the **independent slave configuration**, there is an **independent chip select line for each slave**. **A pull-up resistor between power source and chip select line is highly recommended for each independent device to reduce cross-talk between devices.** This is the way SPI is normally used. **Since the MISO pins of the slaves are connected together, they are required to be tri-state pins** (high, low or high-impedance).

### Daisy chain configuration

[](https://en.wikipedia.org/wiki/File:SPI_three_slaves_daisy_chained.svg)

Daisy-chained SPI bus: master and cooperative slaves

Some products that implement SPI may be connected in a [daisy chain](https://en.wikipedia.org/wiki/Daisy_chain_(electrical_engineering)) configuration, **the first slave output being connected to the second slave input, etc. The SPI port of each slave is designed to send out during the second group of clock pulses an exact copy of the data it received during the first group of clock pulses**.

The whole chain acts as a communication [shift register](https://en.wikipedia.org/wiki/Shift_register); daisy chaining is often done with shift registers to provide a bank of inputs or outputs through SPI. Such a feature only requires a single SS line from the master, rather than a separate SS line for each slave.[[4]](https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus#cite_note-4)

Applications that require a daisy chain configuration include [SGPIO](https://en.wikipedia.org/wiki/SGPIO) and [JTAG](https://en.wikipedia.org/wiki/JTAG)

### **Advantages**

* **Full duplex communication in the default version of this protocol**.
* [Push-pull drivers](https://en.wikipedia.org/wiki/Push-pull_output) (as opposed to open drain) provide good signal integrity and high speed
* **Higher**[**throughput**](https://en.wikipedia.org/wiki/Throughput)**than**[**I²C**](https://en.wikipedia.org/wiki/I%C2%B2C)**or**[**SMBus**](https://en.wikipedia.org/wiki/System_Management_Bus)
* Complete protocol flexibility for the bits transferred
  + Not limited to 8-bit words
  + Arbitrary choice of message size, content, and purpose
* Extremely simple hardware interfacing
  + **Typically lower power requirements than**[**I²C**](https://en.wikipedia.org/wiki/I%C2%B2C)**or SMBus due to less circuitry** (including pull up resistors)
  + No arbitration or associated failure modes
  + Slaves use the master's clock and do not need precision oscillators
  + Slaves do not need a unique [address](https://en.wikipedia.org/wiki/Address_space) — unlike [I²C](https://en.wikipedia.org/wiki/I%C2%B2C) or [GPIB](https://en.wikipedia.org/wiki/GPIB) or [SCSI](https://en.wikipedia.org/wiki/SCSI)
  + Transceivers are not needed
* Uses only four pins on IC packages, and wires in board layouts or connectors, much fewer than parallel interfaces
* At most one unique bus signal per device (chip select); all others are shared
* Signals are unidirectional allowing for easy [Galvanic isolation](https://en.wikipedia.org/wiki/Galvanic_isolation)
* Not limited to any maximum clock speed, enabling potentially high speed
* **Simple software implementation**

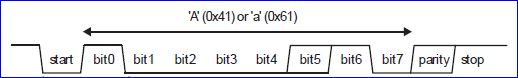
### **Disadvantages**

* Requires more pins on IC packages than [I²C](https://en.wikipedia.org/wiki/I%C2%B2C), even in the [*three-wire*](https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus#Three-wire_serial_buses) variant
* No in-band addressing; out-of-band chip select signals are required on shared buses
* **No hardware**[**flow control**](https://en.wikipedia.org/wiki/Flow_control_(data))**by the slave (**but the master can delay the next clock edge to slow the transfer rate)
* **No hardware slave acknowledgment** (the master could be transmitting to nowhere and not know it)
* Typically supports only one master device (depends on device's hardware implementation)
* No error-checking protocol is defined
* Without a formal standard, validating conformance is not possible
* Only handles short distances compared to [RS-232](https://en.wikipedia.org/wiki/RS-232), [RS-485](https://en.wikipedia.org/wiki/RS-485), or [CAN-bus](https://en.wikipedia.org/wiki/CAN-bus). (its distance can be extended with use of transceivers like [RS-422](https://en.wikipedia.org/wiki/RS-422))
* Many existing variations, making it difficult to find development tools like host adapters that support those variations
* SPI does not support [hot swapping](https://en.wikipedia.org/wiki/Hot_swapping) (dynamically adding nodes).
* Interrupts must either be implemented with out-of-band signals or be faked by using periodic polling similarly to USB 1.1 and 2.0
* Some variants like [Multi I/O SPI](https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus#Multi_I.2FO_SPI) and [three-wire serial buses](https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus#Three-wire_serial_buses) defined below are half-duplex.

**UART**

**UART** stands for **U**niversal **A**synchronous **R**eceiver **T**ransmitter. There is one wire for transmitting data, and one wire to receive data. A common parameter is the baud rate known as "bps" which stands for **b**its **p**er **s**econd. If a transmitter is configured with 9600bps, then the receiver must be listening on the other end at the same speed.

UART is a serial communication, so bits must travel on a single wire. If you wish to send a **char** over UART, the char is enclosed within a **start** (logic 0) and a **stop** bit (logic 1), so to send 8-bits of **char** data, it would require 2-bit overhead; this 10-bit of information is called a **UART frame**. Let's take a look at how the character 'A' is sent over UART. In ASCII table, the character 'A' has the value of 65, which in binary is: 0100-0001. If you inform your UART hardware that you wish to send this data at 9600bps, here is how the frame would appear on an oscilloscope:

[](http://www.socialledge.com/sjsu/index.php?title=File:Uart_tutorial_frame.jpg)

UART Frame of 'A'

**UART Receiver**

**All operations of the UART hardware are controlled by a clock signal which runs at a multiple of the data rate**, typically **8 times the bit rate**. **The receiver tests the state of the incoming signal on each clock pulse, looking for the beginning of the start bit. If the apparent start bit lasts at least one-half of the bit time, it is valid and signals the start of a new character.** If not, it is considered a spurious pulse and is ignored. After waiting a further bit time, the state of the line is again sampled and the resulting level clocked into a shift register. After the required number of bit periods for the character length (5 to 8 bits, typically) have elapsed, the contents of the shift register are made available (in parallel fashion) to the receiving system. The UART will set a flag indicating new data is available, and may also generate a processor [interrupt](https://en.wikipedia.org/wiki/Interrupt) to request that the host processor transfers the received data.

A UART usually contains the following components:

* a clock generator, usually a multiple of the bit rate to allow sampling in the middle of a bit period.
* input and output [shift registers](https://en.wikipedia.org/wiki/Shift_register)
* transmit/receive control
* read/write control logic
* transmit/receive buffers (optional)
* system data bus buffer (optional)
* First-in, first-out ([FIFO](https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics))) buffer memory (optional)
* Signals needed by a third party DMA controller (optional)
* Integrated bus mastering DMA controller (optional)

## Special transceiver conditions

### Overrun error

An "overrun error" occurs when the receiver cannot process the character that just came in before the next one arrives. Various devices have different amounts of buffer space to hold received characters. The CPU or DMA controller must service the UART in order to remove characters from the input buffer. If the CPU or DMA controller does not service the UART quickly enough and the buffer becomes full, an Overrun Error will occur, and incoming characters will be lost.

### Underrun error

An "underrun error" occurs when the UART transmitter has completed sending a character and the transmit buffer is empty. In asynchronous modes this is treated as an indication that no data remains to be transmitted, rather than an error, since additional stop bits can be appended. This error indication is commonly found in USARTs, since an underrun is more serious in synchronous systems.

### Framing error

A "framing error" occurs when the designated "start" and "stop" bits are not found. As the "start" bit is used to identify the beginning of an incoming character, it acts as a reference for the remaining bits. If the data line is not in the expected state (hi/lo) when the "stop" bit is expected, a *Framing Error* will occur.

### Parity error

A [Parity Error](https://en.wikipedia.org/wiki/Parity_bit) occurs when the [parity](https://en.wikipedia.org/wiki/Parity_(mathematics)) of the number of 1 bits disagrees with that specified by the parity bit. Use of a parity bit is optional, so this error will only occur if parity-checking has been enabled.

### Break condition

A "break condition" occurs when the receiver input is at the "space" (logic low, i.e., '0') level for longer than some duration of time, typically, for more than a character time. This is not necessarily an error, but appears to the receiver as a character of all zero bits with a framing error. The term "break" derives from [currentloop](https://en.wikipedia.org/wiki/Current_loop) signaling, which was the traditional signaling used for [teletypewriters](https://en.wikipedia.org/wiki/Teletypewriter). The "spacing" condition of a current loop line is indicated by no current flowing, and a very long period of no current flowing is often caused by a break or other fault in the line.

Some equipment will deliberately transmit the "space" level for longer than a character as an attention signal. When signaling rates are mismatched, no meaningful characters can be sent, but a long "break" signal can be a useful way to get the attention of a mismatched receiver to do something (such as resetting itself). [Unix-like](https://en.wikipedia.org/wiki/Unix-like) systems can use the long "break" level as a request to change the signaling rate, to support dial-in access at multiple signaling

I2C Vs SPI Vs. UART

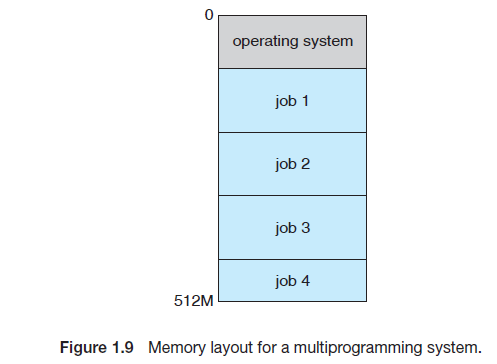
<http://www.rfwireless-world.com/Terminology/UART-vs-SPI-vs-I2C.html>

|  |  |  |  |
| --- | --- | --- | --- |
| **Features** | **UART** | **SPI** | **I2C** |
| Full Form | **Universal Asynchronous Receiver/Transmitter** | Serial Peripheral Interface | Inter-Integrated Circuit |
| Pin Designations | TxD: Transmit Data  RxD: Receive Data | SCLK: Serial Clock  MOSI: Master Output, Slave Input  MISO: Master Input, Slave Output  SS: Slave Select | SDA: Serial Data  SCL: Serial Clock |
| Data rate | **As this is is asynchronous communication**, data rate between two devices wanting to communicate should be set to equal value. Maximum data rate supported is about 230 Kbps to 460kbps. | Maximum data rate limit is not specified in SPI interface. Usually supports about 10 Mbps to 20 Mbps | I2C supports 100 kbps, 400 kbps, 3.4 Mbps. Some variants also supports 10 Kbps and 1 Mbps. |
| Distance | Lower about 50 feet | highest | Higher |
| Type of communication | **Asynchronous** | **Synchronous** | **Synchronous** |
| Number of masters | Not Application | One | One or more than One |
| Clock | No Common Clock signal is used. Both the devices will use there independent clocks. | There is one common serial clock signal between master and slave devices. | There is common clock signal between multiple masters and multiple slaves. |
| Hardware complexity | lesser | less | more |
| Protocol | For 8 bits of data one start bit and one stop bit is used. | Each company or manufacturers have got their own specific protocols to communicate with peripherals. Hence one needs to read datasheet to know read/write protocol for SPI communication to be established. For example we would like SPI communication between microcontroller and EPROM. Here one need to go through read/write operational diagram in the EPROM data sheet. | It uses start and stop bits. It uses ACK bit for each 8 bits of data which indicates whether data has been received or not. Figure depicts the data communication protocol. |
| Software addressing | As this is one to one connection between two devices, addressing is not needed. | Slave select lines are used to address any particular slave connected with the master. There will be 'n' slave select lines on master device for 'n' slaves. | There will be multiple slaves and multiple masters and all masters can communicate with all the slaves. Upto 27 slave devices can be connected/addressed in the I2C interface circuit. |
| Advantages | • **It is simple communication and most popular which is available due to UART support in almost all the devices with 9 pin connector**. It is also referred as RS232 interface. | •It is simple protocol and hence so not require processing overheads.   •Supports full duplex communication.   •Due to separate use of CS lines, same kind of multiple chips can be used in the circuit design.   •SPI uses push-pull and hence higher data rates and longer ranges are possible.   •SPI uses less power compare to I2C | •Due to open collector design, limited slew rates can be achieved.   •More than one masters can be used in the electronic circuit design.   •**Needs fewer i.e. only 2 wires for communication.**   •**I2C addressing is simple which does not require any CS lines used in SPI and it is easy to add extra devices on the bus.**   •It uses open collector bus concept. Hence there is bus voltage flexibity on the interface bus.   •**Uses flow control.** |
| Disadvantages | • **They are suitable for communication between only two devices.**  • It supports fixed data rate agreed upon between devices initially before communication otherwise data will be garbled. | • As number of slave increases, number of CS lines increases, this results in hardware complexity as number of pins required will increase.   **• To add a device in SPI requires one to add extra CS line and changes in software for particular device addressing is concerned**.   **•Master and slave relationship can not be changed as usually done in I2C interface.**   •**No flow control available in SPI.** | •**Increases complexity of the circuit when number of slaves and masters increases.**   •**I2C interface is half duplex.**   •**Requires software stack to control the protocol and hence it needs some processing overheads on microcontroller/microprocessor**. |

**20. Multiprogramming, multiprocessing, multithreading and multitasking (4 Multis)**

**Multiprogramming**

One of the most important aspects of operating systems is the ability to multiprogram. A single program cannot, in general, keep either the CPU or the I/O devices busy at all times. Single users frequently have multiple programs running. **Multiprogramming** increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute. The idea is as follows: The operating system keeps several jobs in memory simultaneously (Figure 1.9). Since, in general, main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the **job pool**. This pool consists of all processes residing on disk awaiting allocation of main memory.



The set of jobs in memory can be a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation,

to complete. In a non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and executes, another job. When *that* job needs to wait, the CPU is switched to *another* job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle.

The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

**Process**  
The system call **fork() is used to create new processes**. It does not take any arguments and **returns a process ID**.

If **fork() returns a negative value**, the **creation of a child process was unsuccessful**. A call to **fork() returns a zero to the newly created child process and the same call to fork() returns a positive value (the process ID of the child process) to the parent.**

The returned **process ID is of type pid\_t** defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use **function getpid() to retrieve the process ID assigned to this process**.

**Copy-on-Write**

Traditionally, **upon fork(), all resources owned by the parent are duplicated and the copy is given to the child**.

In Linux, **fork() is implemented through the use of *copy-on-write* pages**. Copy-on-write (or *COW*) is a technique **to delay or altogether prevent copying of the data**. **Rather than duplicate the process address space, the parent and the child can share a single copy. The data, however, is marked in such a way that if it is written to, a duplicate is made and each process receives a unique copy. Consequently, the duplication of resources occurs only when they are written; until then, they are shared read-only**. This technique delays the copying of each page in the address space until it is actually written to.

In the case that the pages are never written—for example, **if exec() is called immediately after fork()—they never need to be copied. The only overhead incurred by fork() is the duplication of the parent’s page tables and the creation of a unique process descriptor for the child.**

In the common case that a process executes a new executable image immediately after forking, this optimization prevents the wasted copying of large amounts of data (with the address space, easily tens of megabytes). This is an important optimization because the Unix philosophy encourages quick process execution.  
  
**What is a Zombie process?**

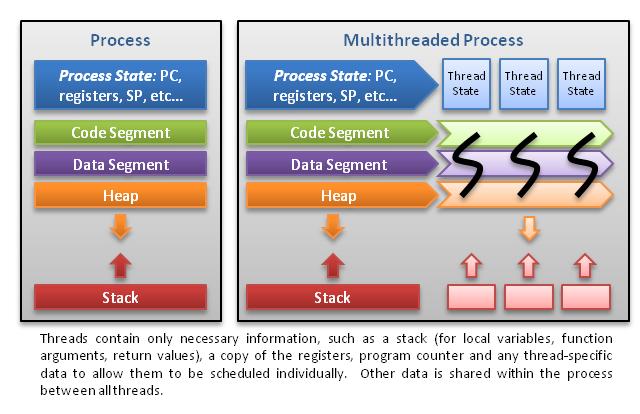
When using fork() to create child processes it is important to keep track of these processes. For instance, **when a child process terminates, an association with the parent survives until the parent either terminates normally or calls wait(). The child process entry in the process table is not freed up immediately. Although it is no longer active, the child process is still in the system. The child process is at that point referred to as a zombie process.**

**Note, if the parent terminates abnormally then the child process gets the process with PID 1, (init) as parent**. (such a **child process is often referred to as an orphan**). The child process is now a zombie. It is no longer running, it's original parent process is gone, and it has been inherited by init. It will remain in the process table as a zombie until the next time the table is processed. If the process table is long this may take a while. till init cleans them up. As a general rule, program wisely and try to avoid zombie processes. When zobbies accumulate they eat up valuable resources.

The waitpid() system call is another call that can be used to wait for child processes. This system call however can be used to wait for a specific process to terminate.  
  
**So whats the difference between fork() and vfork()?**

The system call vfork(), is a low overhead version of fork(), as fork() involves copying the entire address space of the process and is therefore quite expensive. The basic difference between the two is that when a new process is created with vfork(), the parent process is temporarily suspended, and the child process might borrow the parent's address space. This strange state of affairs continues until the child process either exits, or calls execve(), at which point the parent process continues. This means that the child process of a vfork() must be careful to avoid unexpectedly modifying variables of the parent process. In particular, the child process must not return from the function containing the vfork() call, and it must not call exit() (if it needs to exit, it should use \_exit(); actually, this is also true for the child of a normal fork()). However, since vfork() was created, the implementation of fork() has improved , most notably with the introduction of `copy-on-write', where the copying of the process address space is transparently faked by allowing both processes to refer to the same physical memory until either of them modify it. This largely removes the justification for vfork(); indeed, a large proportion of systems now lack the original functionality of vfork() completely. For compatibility, though, there may still be a vfork() call present, that simply calls fork() without attempting to emulate all of the vfork() semantics.

**Multithreading**



**A traditional (or heavyweight) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time**.

Example:

A Web browser might have **one thread display images or text while another thread retrieves data from the network**, for example. **A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background**.

A single application may be required to perform several similar tasks. For example, a Web server accepts client requests for Web pages, images, sound, and so forth. A busy Web server may have several

(perhaps thousands of) clients concurrently accessing it. If the Web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced. One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request.

If the new process will perform the same tasks as the existing process, why incur all that overhead? It is generally more efficient to use one process that contains multiple threads. If the Web-server process is multithreaded, the server will create a separate thread that listens for client requests. When a request is made, rather than creating another process, the server will create a new thread to service the request and resume listening for additional requests.

**Benefits of threads**

The benefits of multithreaded programming can be broken down into four major categories:

**1. Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. For instance, a multithreaded Web browser could allow user interaction in one thread while an image was being loaded in another thread.

**2. Resource sharing**. Processes may only share resources through techniques such as shared memory or message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process towhich they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

**3. Economy**. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general it is much more time consuming to create and manage processes than threads. In Solaris, for example, creating a process is about thirty times slower than creating a thread, and context switching is about five times slower.

**4. Scalability.** The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. A single-threaded process can only run on one

processor, regardless how many are available. Multithreading on a multi- CPU machine increases parallelism. We explore this issue further in the following section.

***Refer Multithreading document***

**Debugging multithreaded application**

* + 1. Using gdb
* To list all the active threads use command,

***“info threads”***

* In non multithreading applications there is only one thread i.e. main function. Therefore, to check the stack trace we use command ***“bt”***. But in multithreading applications, as there are many threads and each thread has its own stack. But ***“bt”*** command will display the stack trace of current active thread only. Now what if want to inspect stack trace of all the threads at same point.

***“ thread apply all bt ”***

It will display the stack trace of all the active threads.

* The command ***“thread apply “*** applies the specified command to specified thread ID. Also, you can specify “all” instead of thread ID to apply the command to all threads.
* To display the stack trace of current thread only use command ***“bt”***
* To switch between different threads to inspect their stack trace while debugging use following

***“thread <thread-specific-number>”***

2. Putting printf statements

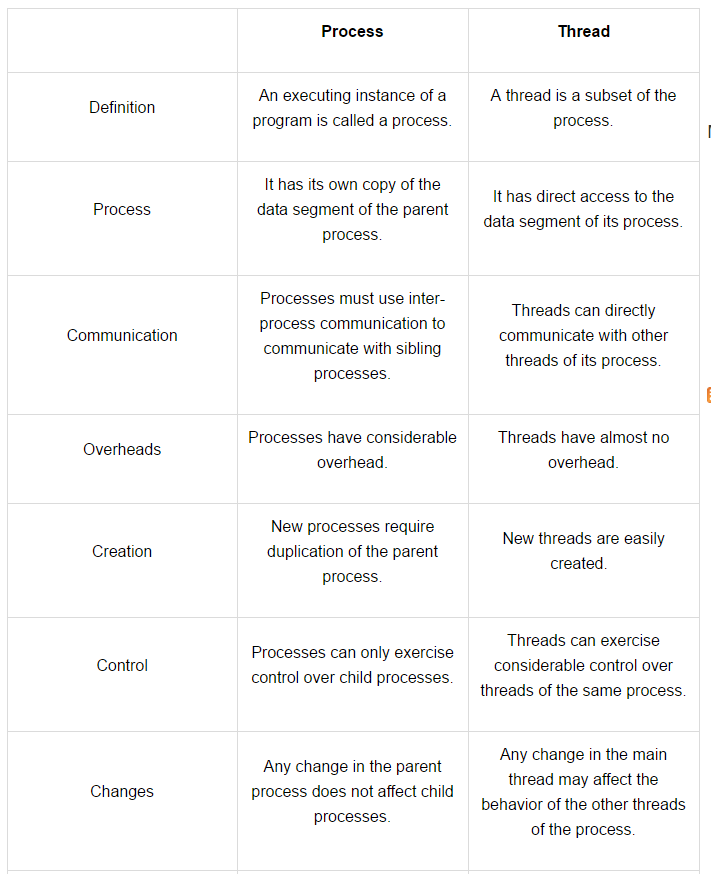
**Nice Value in Linux**In Linux we can set guidelines for the CPU to follow when it is looking at all the tasks it has to do. These guidelines are called ***niceness*** or ***nice value***. **The Linux niceness scale goes from -20 to 19.** The lower the number the more priority that task gets. If the niceness value is high number like 19 the task will be set to the lowest priority and the CPU will process it whenever it gets a chance. **The default nice value is zero.  
Setting Priority on Existing Processes**

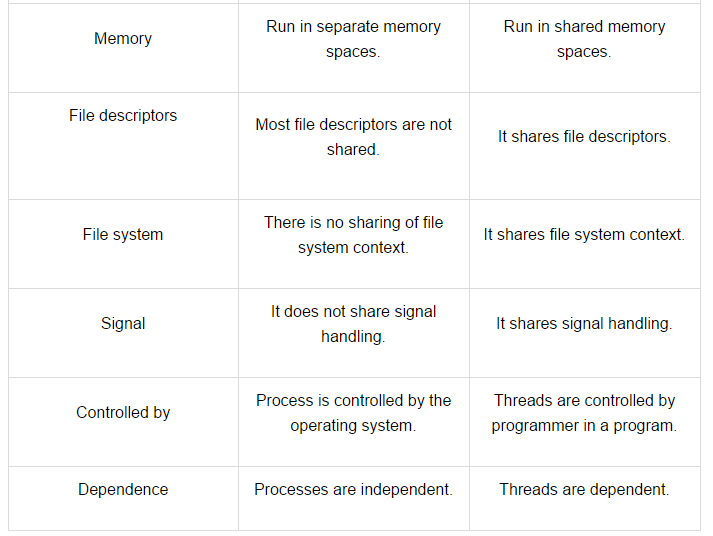
* renice 10 -p 24447

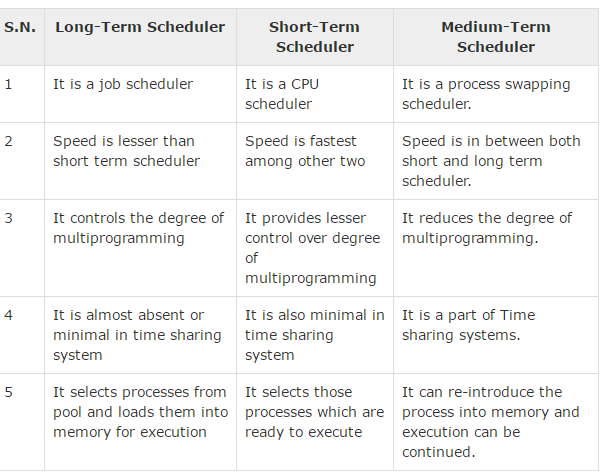
**How to debug kernel panics**

**What is Kconfig**

**What is make menuConfig?**

**Process and Threads**  




**Schedulers:**  


**IOCTL**  
  
ioctl function is useful when one is implementing a device driver to set the configuration on the device.

int ioctl(int fd, int request, ...)

1. fd is file descriptor, the one returned by open
2. request is request code. e.g GETFONT will get current font from printer, SETFONT will set font on a printer.
3. third argument is void \*. Depending on second argument, the third may or may not be present. e.g. if second argument is SETFONT, third argument may give font name as ARIAL.

So now int request is not just a macro, one is required to generate request code to be used by user application and device driver module to determine which configuration on device must be played with. One sends a request code using ioctl from user application and then uses the request code in device driver module to determine which action to perform.

A request code has 4 main parts

1. A Magic number - 8 bits

2. A sequence number - 8 bits

3. Argument type (typically 14 bits), if any.

4. Direction of data transfer (2 bits).

If request code is SETFONT to set font on a printer, the direction for data transfer will be from user application to device driver module. User sends font name Arial to printer. If request code is GETFONT, direction is from printer to user application.

To generate request code Linux provide some predefined function like macros.

1.\_IO(MAGIC, SEQ\_NO) both are 8 bits, 0 to 255, e.g. let us say we want to pause printer. This does not require adata transfer. So we would generate request code as below

#define PRIN\_MAGIC 'P'

#define NUM 0

#define PAUSE\_PRIN \_\_IO(PRIN\_MAGIC, NUM)

Now use ioctl as

ret\_val = ioctl(fd, PAUSE\_PRIN);

Corresponding system call in driver module will receive the code and pause the printer.

1. \_\_IOW(MAGIC, SEQ\_NO, TYPE) MAGIC and SEQ\_NO are same as above, but third part gives the type of next argument, recall the third argument of ioctl is void \*. W in \_\_IOW indicates that direction of data is from user application to driver module. Let us take an example, Suppose one is telling printer to set font to Arial.
2. #define PRIN\_MAGIC 'S'
3. #define SEQ\_NO 1
4. #define SETFONT \_\_IOW(PRIN\_MAGIC, SEQ\_NO, unsigned long)

further,

char \*font = "Arial";

ret\_val = ioctl(fd, SETFONT, font);

Now font is a pointer, which means it is an address best represented as unsigned long, hence the third part of \_IOW mentions type as such. Also, this address of font is passed to corresponding system call implemented in device driver module as unsigned long and we need to cast it to proper type before using it. Kernel space can access user space and hence this works. other two function like macros are \_\_IOR(MAGIC, SEQ\_NO, TYPE) and \_\_IORW(MAGIC, SEQ\_NO, TYPE) where direction of data flow will be from kernel space to user space and both ways respectively.

# **The Kernel Configuration and Build Process**

http://www.linuxjournal.com/article/6568

**CopyToUser**

**CopyFromUser**

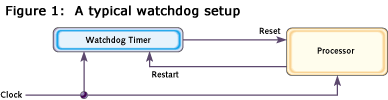
**26.Watchdog timer?**

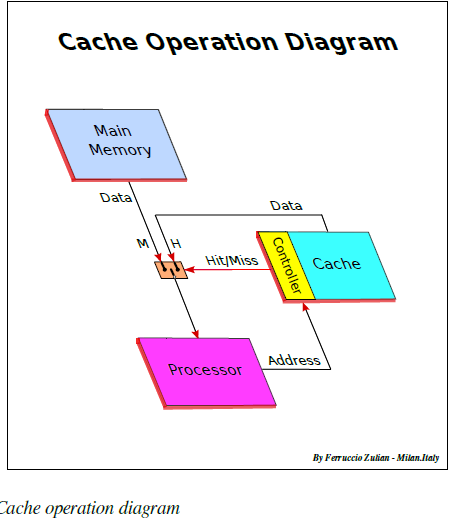
A watchdog timer is **a piece of hardware** that can be used to **automatically detect software anomalies and reset the processor if any occur.**

A watchdog timer is **based on a counter** that **counts down from some initial value to zero**. The **embedded software** selects the counter's initial value and **periodically restarts it**.

If the **counter ever reaches zero before the software restarts** it, the **software is presumed to be malfunctioning** and the processor's reset signal is asserted. **The processor (and the embedded software it's running) will be restarted** as if a human operator had cycled the power.

Figure 1 shows a typical arrangement. As shown, the watchdog timer is a chip external to the processor. However, it could also be included within the same chip as the CPU. This is done in many microcontrollers. In either case, the output from the watchdog timer is tied directly to the processor's reset signal.



**27. Cache (L1, L2, L3)**   


There are three basic structures and two (plus one) types

of caches:

***\_* Fully Associative cache**  
**Any memory block can be stored in any cache location.** It is called “fully associative” because each data stored in cache is associated to its full address.

***\_* Direct Mapped cache**  
**In direct mapped (or single set-associative cache) any memory block can be stored in one specific cache entry only**. The entry where to store the block is *direct* *derived* from the memory address (so that the name “Direct-Mapped”).

***\_* Set Associative cache**

Types:

*\_* Instruction cache

*\_* Data cache  
  
**Write policy**

The cache’s write policy determines how it handles writes to memory locations that are currently being held in cache. Generally, only the Data Cache is involved, because usually instructions are not self-modifying, and in the case of code self-modifying, the software may force the CPU to store this code only in MM without involving the cache

There are two basic policy types:

*\_* Write Through

*\_* Write-Back (or Copy Back)

**Write through**

*\_* Data is written at the same time both to cache and to MM, or to cache and then to memory

**Write-back (or copy back)**

Data is updated only in cache. The data is “written back” to MM when needed, for instance on cache line replacement (overwrite) or when required by other caches. This reduces bus and memory traffic because the next cache line update is taken only in cache without involving the memory. The bit "**D**" or "**M**" – (Dirty or Modified) is set on in cache *Directory*.

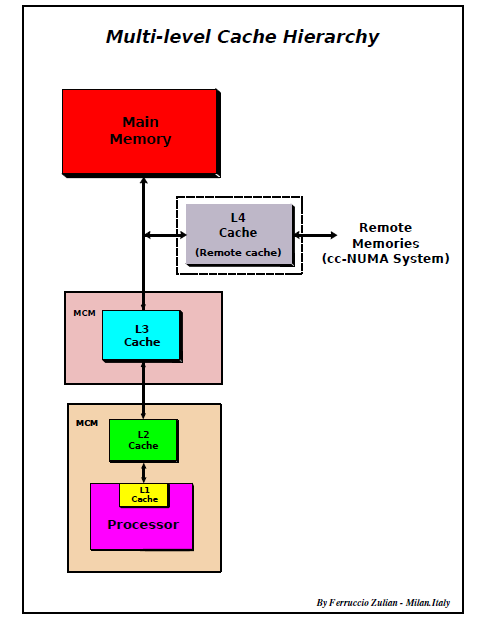
In case of miss on write there are two different solutions:

**Write allocate**

***Write allocate on miss*** called also ***Fetch on- write*** or ***RWITM*** (**R**ead **W**ith **I**ntent **T**o**M**odify) or ***Read for Write.*** *\_* On a write miss, the cache line is first read fromMain Memory or from a cache in case of *Cache Intervention*,then the data is written in cache with thenew data – cache line updating – (cache line partialwrite: byte, halfword, word or doubleword – 8, 16,32 or 64 bit, according to intrinsic parallelism of theprocessing unit

**Write no-allocate** ( or no-Write allocate)

The data is directly written in MM bypassing the cache *\_ Write allocate* is usually associated with *write-back*. Write *no-allocate* is usually associated with *writethrough*.



**27b. Cache Coherency**   
  
In systems as SMP – Symmetric Multiprocessor System, multi-core and NUMA system, where a dedicated cache

for each *processor*, *core* or *node* is used, a consistency problem may occur when a same data is stored in more than one cache. This problem arises when a data is modified in one cache. This problem can be solved in two ways:

1. Invalidate all the copies on other caches

(broadcast-invalidate)

2. Update all the copies on other caches (writebroadcasting),

while the memory may be updated

(write through) or not updated (writeback).

**28. Compiling, linking and loading**   
The compilation of a C++ program involves three steps:

1. Preprocessing: the **preprocessor takes a C++ source code file and deals with the #includes, #defines and other preprocessor directives**. The output of this step is a "pure" C++ file without pre-processor directives.
2. Compilation: the compiler takes the **pre-processor's output and** **produces an object file** from it.
3. Linking: the linker takes **the object files produced by the compiler and produces either a library or an executable file.**

# **Preprocessing**

The preprocessor handles the preprocessor directives, like #include and #define. It is agnostic of the syntax of C++, which is why it must be used with care. It works on one C++ source file at a time by replacing #include directives with the content of the respective files (which is usually just declarations), doing replacement of macros (#define), and selecting different portions of text depending on #if, #ifdef and #ifndef directives.

.

# **Compilation**

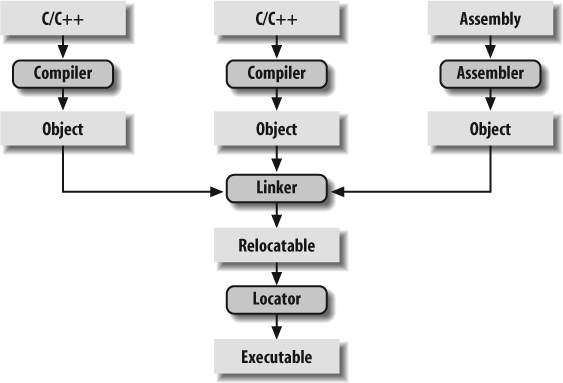
**The compilation step is performed on each output of the preprocessor**. The compiler parses the pure C++ source code (now without any preprocessor directives) and **converts it into assembly code**. **Then invokes underlying back-end(assembler in toolchain) that assembles that code into machine code producing actual binary file in some format(ELF, COFF, a.out, ...)**. This object file contains the compiled code (in binary form) of the symbols defined in the input. Symbols in object files are referred to by name.

# **Linking**

The **linker produces the final compilation output from the object files the compiler produced**. This output can be **either a shared (or dynamic) library or an executable**.

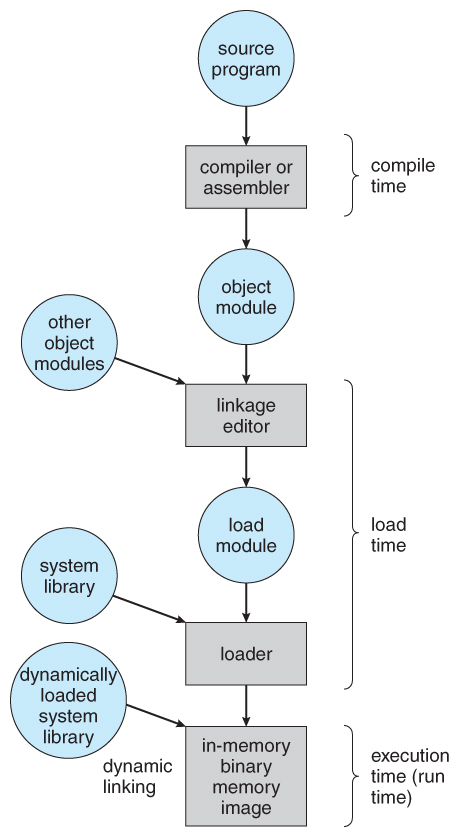
**It links all the object files by replacing the references to undefined symbols with the correct addresses. Each of these symbols can be defined in other object files or in libraries. If they are defined in libraries other than the standard library, you need to tell the linker about them.**

At this stage the **most common errors are missing definitions or duplicate definitions**. The former means that **either the definitions don't exist (i.e. they are not written), or that the object files or libraries where they reside were not given to the linker**. The latter is obvious: the same symbol was defined in two different object files or libraries.



#### **Address Binding**

* User **programs typically refer to memory addresses with symbolic names** such as "i", "count", and "averageTemperature". These **symbolic names must be mapped or *bound* to physical memory addresses**, which typically occurs in several stages:
  + **Compile Time** - **If it is known at compile time where a program will reside in physical memory, then*absolute code* can be generated by the compiler, containing actual physical addresses. However if the load address changes at some later time, then the program will have to be recompiled**. DOS .COM programs use compile time binding.
  + **Load Time** - **If the location at which a program will be loaded is not known at compile time, then the compiler must generate *relocatable code*, which references addresses relative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.**
  + **Execution Time** - **If a program can be moved around in memory during the course of its execution, then binding must be delayed until execution time.** This requires special hardware, and is the method implemented by most modern OSes.
* Figure 8.3 shows the various stages of the binding processes and the units involved in each stage:



#### 8.1.4 Dynamic Loading

* **Rather than loading an entire program into memory at once, dynamic loading loads up each routine as it is called.** The advantage is that unused routines need never be loaded, reducing total memory usage and generating faster program startup times. The downside is the added complexity and overhead of checking to see if a routine is loaded every time it is called and then loading it up if it is not already loaded.

#### 8.1.5 Dynamic Linking and Shared Libraries

* With ***static linking*** library **modules get fully included in executable modules, wasting both disk space and main memory usage**, because **every program that included a certain routine from the library would have to have their own copy of that routine linked into their executable code**.
* With ***dynamic linking***, however, **only a stub is linked into the executable module**, containing references to the actual library module linked in at run time. **This method saves disk space, because the library routines do not need to be fully included in the executable modules, only the stubs**.
  + **When a program uses a routine from a standard library and the routine changes, then the program must be re-built ( re-linked ) in order to incorporate the changes. However if DLLs are used, then as long as the stub doesn't change, the program can be updated merely by loading new versions of the DLLs onto the system.** In practice, **the first time a program calls a DLL routine, the stub will recognize the fact and will replace itself with the actual routine from the DLL library. Further calls to the same routine will access the routine directly and not incur the overhead of the stub access**.
* Swapping is typically not supported on mobile platforms, for several reasons:
  + Mobile devices typically use flash memory in place of more spacious hard drives for persistent storage, so there is not as much space available.
  + Flash memory can only be written to a limited number of times before it becomes unreliable.
  + The bandwidth to flash memory is also lower.
* Apple's IOS asks applications to voluntarily free up memory
  + Read-only data, e.g. code, is simply removed, and reloaded later if needed.
  + Modified data, e.g. the stack, is never removed, but . . .
  + Apps that fail to free up sufficient memory can be removed by the OS
* Android follows a similar strategy.
  + Prior to terminating a process, Android writes its ***application state*** to flash memory for quick restarting.

**29.** **.a and .so libraries**

Linking is the process of bringing external programs together required by the one we write for its successful execution. Static and dynamic linking are two processes of collecting and combining multiple object files in order to create a single executable.

Linking can be performed at both compile time, when the source code is translated into machine code; and load time, when the program is loaded into memory by the loader, and even at run time, by application programs. And, it is performed by programs called linkers. Linkers are also called link editors. Linking is performed as the last step in compiling a program.

After linking, for execution the combined program must be moved into memory. In doing so, there must be addresses assigned to the data and instructions for execution purposes. The above process can be summarized as program life cycle (*write -> compile -> link -> load -> execute*).  
  
**Static Linking and Static Libraries** is the result of the linker **making copy of all used library functions to the executable file.** Static Linking creates larger binary files, and need more space on disk and main memory. Examples of static libraries (libraries which are statically linked) **are .a files in Linux and .libfiles in Windows.**  
  
**Dynamic linking and Dynamic Libraries** Dynamic Linking **doesn’t require the code to be copied**, it is done by **just placing name of the library in the binary file.** **The actual linking happens when the program is run, when both the binary file and the library are in memory.** Examples of Dynamic libraries (libraries which are linked at run-time) **are .so in Linux and .dll in Windows.**

**31.Which one is better C or C++ for embedded systems (C vs C++)**  
In addition, there has been much debate within the embedded software community about the appropriateness of C++. It is widely believed that C++ programs produce larger executables that run more slowly than programs written entirely in C. However, C++ has many benefits for programmers.

This does not include expensive features such as multiple inheritance, virtual base classes, runtime type identification, exception handling and some of the newest additions such as templates, namespaces, and new-style casts. All these negatively impact code size, and exceptions and runtime type identification also increase execution time.

What’s left is a simpler version of C++ that is still object-oriented and a superset of C, but has significantly less runtime overhead and smaller runtime libraries. In fact, the earliest C++ compilers used a technology called C-front to turn C++ programs into C, which was then fed into a standard C compiler. That this is even possible demonstrates that many of the syntactical differences between the languages have little or no runtime cost.

For example, the definition of a class is completely benign. The list of public and private member data and functions is not much different from a struct and a list of function prototypes. However, the C++ compiler is able to use the public and private keywords to determine which method calls and data accesses are allowed and prohibited. Because this determination is made at compile time, there is no penalty paid at runtime. Thus, the use of classes alone affects neither the code size nor efficiency of your programs.

Default parameter values are also penalty-free. The compiler simply inserts code to pass the default value whenever the function is called without an argument in that position. Similarly, function name overloading involves only a compile-time code modification. Functions with the same names but different parameters are each assigned unique names during the compilation process. The compiler alters the function name each time it appears in your program, and the linker matches them up appropriately.

Operator overloading is another feature that might be used in embedded systems. Whenever the compiler sees such an operator, it simply replaces it with the appropriate function call.

Constructors and destructors have a slight penalty. These special methods are guaranteed to be called each time an object of the type is created or goes out of scope, respectively. However, this small amount of overhead is a reasonable price to pay for fewer bugs. Constructors eliminate an entire class of C programming errors having to do with uninitialized data structures. This feature has also proved useful for hiding the awkward initialization sequences associated with some classes.

Virtual functions also have a reasonable cost/benefit ratio. Without going into too much detail about what virtual functions are, let’s just say that polymorphism would be impossible without them. And without polymorphism, C++ would not be a true object-oriented language. The only significant cost of virtual functions is one additional memory lookup before a virtual function can be called. Ordinary function and method calls are not affected.

**Hardware breakpoint and software breakpoint**

You can go through [GDB internals](https://sourceware.org/gdb/wiki/Internals/Breakpoint%20Handling), its very well explains the HW and SW breakpoints.

HW breakpoints are something that require support from MCU. The ARM controllers have special registers where you can write some address space, whenever PC (program counter) == sp register CPU halts. Jtag is usually required to write into those special registers.

**Watchpoints** are a case where hardware handling is much faster:

watch var

rwatch var

awatch var

When you enter those commands on GDB 7.7 x86-64 it says:

Hardware watchpoint 2: var

This hardware capability for x86 is mentioned at: <http://en.wikipedia.org/wiki/X86_debug_register>

It is likely possible because of the existing paging circuit, which manages every memory access.

**Stack is growing upwards or downwards?**

void checkStack(int\*);

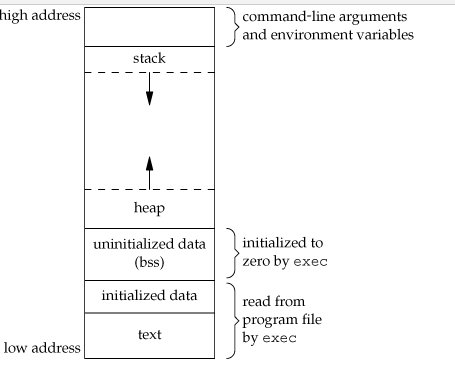
int main()   
{   
 int i;  // address 1000  
 checkStack(&i);   
 return 0;   
}   
void checkStack(int \*j)  // address 1000 passed  
{   
 int k;   
 if(&k > j)

printf("Stack is growing UP");   
 else

printf("Stack is growing DOWN");   
}

**38.Memory Layout of program, Storage classes and their mapping.**

<https://www.tecmint.com/12-top-command-examples-in-linux/>



**1. Text Segment:**  
A text segment , **also known as a code segment or simply as text**, is one of the sections of a program in an object file or in memory, **which contains executable instructions.**

As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.

Also, **the text segment is often read-only,** to prevent a program from accidentally modifying its instructions.

**2. Initialized Data Segment:**  
Initialized data segment, usually called simply the Data Segment. A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Note that, data segment is not read-only, since the values of the variables can be altered at run time.

This segment can be further classified into initialized read-only area and initialized read-write area.

**3. Uninitialized Data Segment:**  
Uninitialized data segment, often called the “bss” segment”  
Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

For instance a variable declared static int i; would be contained in the BSS segment.  
For instance a global variable declared int j; would be contained in the BSS segment.

**4. Stack:**  
The stack area traditionally adjoined the heap area and grew the opposite direction;

The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”; A stack frame consists at minimum of a return address.

Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.

**5. Heap:**  
Heap is the segment where dynamic memory allocation usually takes place.

The heap area begins at the end of the BSS segment and grows to larger addresses from there.The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single “heap area” is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process’ virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

**Matrix Multiplication**

**//For Square Matrix**

for (i = 0; i < N; i++)

{

   for (j = 0; j < N; j++)

   {

       res[i][j] = 0;

       for (k = 0; k < N; k++)

       res[i][j] += mat1[i][k]\*mat2[k][j];

   }

}

**//For Rectangular Matrix**

int res[m1][n2];

for (i = 0; i < m1; i++) //Row of the first matrix

{

    for (j = 0; j < n2; j++) //column of the second matrix

{

       res[i][j] = 0;

         for (x = 0; x < m2; x++) //row of the second matrix

{

             res[i][j] += mat1[i][k] \* mat2[k][j];

}

}

}

**Storage Class**:

Auto

Register

Static

Extern

Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block. Also, a normal global variable can be made extern as well by placing the ‘extern’ keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

<https://stackoverflow.com/questions/1433204/how-do-i-use-extern-to-share-variables-between-source-files>

**Extern “C” .. Not Understood at all … ☹**

extern "C" makes a function-name in C++ have 'C' linkage (compiler does not mangle the name) so that client C code can link to (i.e use) your function using a 'C' compatible header file that contains just the declaration of your function. Your function definition is contained in a binary format (that was compiled by your C++ compiler) that the client 'C' linker will then link to using the 'C' name.

When you state that a function has extern "C" linkage in C++, the C++ compiler does not add argument/parameter type information to the name used for linkage.

Just so you know, you can specify "C" linkage to each individual declaration/definition explicitly or use a block to group a sequence of declarations/definitions to have a certain linkage:

extern "C" void foo(int);

extern "C"

{

void g(char);

int i;

}

If you care about the technicalities, they are listed in section 7.5 of the C++03 standard, here is a brief summary (with emphasis on extern "C"):

* extern "C" is a linkage-specification
* Every compiler is required to provide "C" linkage
* a linkage specification shall occur only in namespace scope
* ~~all function types, function names and variable names have a language linkage~~[**See Richard's Comment:**](https://stackoverflow.com/questions/1041866/in-c-source-what-is-the-effect-of-extern-c#comment20842899_1041880) Only function names and variable names with external linkage have a language linkage
* two function types with distinct language linkages are distinct types even if otherwise identical
* linkage specs nest, inner one determines the final linkage
* extern "C" is ignored for class members
* at most one function with a particular name can have "C" linkage (regardless of namespace)
* ~~extern "C" forces a function to have external linkage (cannot make it static)~~**See Richard's comment:**'static' inside 'extern "C"' is valid; an entity so declared has internal linkage, and so does not have a language linkage
* Linkage from C++ to objects defined in other languages and to objects defined in C++ from other languages is implementation-defined and language-dependent. Only where the object layout strategies of two language implementations are similar enough can such linkage be achieved

**38. Static in C and C++**

**Static in C**  
Static variables have **a property of preserving their value even after they are out of their scope**. Hence, static variables preserve their previous value in their previous scope and are not initialized again in the new scope.

Syntax:

static data\_type var\_name = var\_value;

Following are some interesting facts about static variables in C.

**1)** A **static int variable remains in memory while the program is running**. A normal or auto variable is destroyed when a function call where the variable was declared is over.

For example, we can use static int to count number of times a function is called, but an auto variable can’t be sued for this purpose.

For example below program prints “1 2”

|  |
| --- |
| #include<stdio.h>  int fun()  {    static int count = 0;    count++;    return count;  }    int main()  {    printf("%d ", fun());    printf("%d ", fun());    return 0;  } |

**2)** Static variables are allocated memory in **data segment**, not stack segment.  
**3) Static variables** (like global variables) are **initialized as 0** **if not initialized explicitly**.  
**4)** In C, static variables can only be initialized using constant literals.

Static in Multithreading

**The following C function is intended to be used to allocate unique identifiers (UIDs) to its callers:**

get\_uid()

{

static int i = 0;

return i++;

}

**Explain in what way get\_uid() might work incorrectly in an environment where it is being called by multiple threads. Using a specific example scenario, give specific detail on why and how such incorrect behaviour might occur.**

Your assumption (threads have their own copy) is not correct. **The main problem with code is when multiple threads call that function get\_uid(), there's a possible race condition as to which threads increments i and gets the ID which may not be unique.**

# **Static functions in C**

In C, functions are global by default. **The “static” keyword before a function name** makes it static

|  |
| --- |
| static int fun(void){    printf("I am a static function ");  } |

Unlike global functions in C, **access to static functions is restricted to the file where they are declared**. Therefore**, when we want to restrict access to functions, we make them static**. Another reason for making functions static can be reuse of the same function name in other files.

Now, if we compile the above code with command “gcc file2.c file1.c”, we get the error “undefined reference to `fun1’” . This is because fun1() is declared staticin file1.c and cannot be used in file2.c.

**Static in C++**  
**Static Data Members:**

A **static member is shared by all objects of the class**.   
All **static data is initialized to zero when the first object is created, if no other initialization is present**.

We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator **::** to identify which class it belongs to.  
  
In C++, **a static member function of a class cannot be virtual**.

Also, **static member function cannot be const and volatile**.

|  |  |
| --- | --- |
| 3  4  5  6  7  8  9  10  11  12  13  14 | class Something  {  private:  **static int value;**  public:      static int getValue() {return value;} // static member function  };  **int Something::value = 1; // initializer**  int main()  {      std::cout << **Something::getValue()** << '\n';  } |

**Static Member Functions:**

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using **only the class name and the scope resolution operator ::** A static member **function can only access static data member, other static member functions and any other functions from outside the class**.

Static member functions have a class scope and they do not have access to the **this** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

**int\* arr[8]; // An array of int pointers.**

**int (\*arr)[8]; // A pointer to an array of integers**

**An Array of Pointers**An array can contain char pointers.  
char \*suit[4] = {"Hearts", "Diamonds", "Clubs", Spades"};

Note that even though the **suit** array has fixed size of 4, it can have arbitrary size of character arrays.

**null pointer**

NULL is defined as **(void \*)0**.

A null pointer is a value that any pointer may take to represent that it is pointing to "nowhere".

A null pointer refers to the value stored in the pointer itself.

**void pointer**

A void pointer is a **special type of pointer that can point to anywhere without a specific type**.

A void pointer refers to the type of data it points to.

**Important code snippets on pointers, string and character arrays**

**Typedef Vs Macro**

#include <stdio.h>

typedef char\* ptr;

#define PTR char\*

int main()

{

    ptr a, b, c;

    PTR x, y, z;

    printf("sizeof a:%u\n" ,sizeof(a) );

    printf("sizeof b:%u\n" ,sizeof(b) );

    printf("sizeof c:%u\n" ,sizeof(c) );

    printf("sizeof x:%u\n" ,sizeof(x) );

    printf("sizeof y:%u\n" ,sizeof(y) );

    printf("sizeof z:%u\n" ,sizeof(z) );

    return 0;

}

From the output of the above program size of “a” which is a pointer is 8 (on a machine where pointers are stored using 8 bytes). In the above program, when the comes to

typedef char\* ptr;

ptr a, b, c;

the second line effectively becomes

char\* a, b, c;

This declares a, b, c as char\*.

In contrast, define is like this:

#define PTR char\*

PTR x, y, z;

The second line efficiently becomes

char\* x,y,z;

This makes x and y, z different as x is pointer-to-a char where as y,z are char variables. When we declare macros with pointers while defining if while defining if declare more than one identifier then actual definition is given to first identifier and for the rest non pointer definition is given. In the above case x will be declared as char\*, so its size is the size of pointer where as y and z will be declared as char so, there size will be 1 byte.

**Segmentation fault**:  
On systems using **hardware**[**memory segmentation**](https://en.wikipedia.org/wiki/Memory_segmentation)**to provide**[**virtual memory**](https://en.wikipedia.org/wiki/Virtual_memory)**, a segmentation fault occurs when the hardware detects an attempt to refer to a non-existent segment, or to refer to a location outside the bounds of a segment, or to refer to a location in a fashion not allowed by the permissions granted for that segment**On systems using **only**[**paging**](https://en.wikipedia.org/wiki/Paging)**, an**[**invalid page fault**](https://en.wikipedia.org/wiki/Invalid_page_fault)**generally leads to a segmentation fault, and segmentation faults and page faults are both faults raised by the**[**virtual memory**](https://en.wikipedia.org/wiki/Virtual_memory)**management system.**   
Segmentation faults can also occur independently of page faults: illegal access to a valid page is a segmentation fault, but not an invalid page fault, and segmentation faults can occur in the middle of a page (hence no page fault), for example in a buffer overflow that stays within a page but illegally overwrites memory.

The following are some typical causes of a segmentation fault:

* **Dereferencing**[**null pointers**](https://en.wikipedia.org/wiki/Null_pointer)**– this is special-cased by memory management hardware**
* **Attempting to access a nonexistent memory address**
* **Attempting to access memory the program does not have rights to**
* **Attempting to write read-only memory**
* **Dereferencing or assigning to an uninitialized pointer (**[**wild pointer**](https://en.wikipedia.org/wiki/Wild_pointer)**, which points to a random memory address)**
* **Dereferencing or assigning to a freed pointer (**[**dangling pointer**](https://en.wikipedia.org/wiki/Dangling_pointer)**, which points to memory that has been freed/deallocated/deleted)**
* **A**[**buffer overflow**](https://en.wikipedia.org/wiki/Buffer_overflow)
* **A**[**stack overflow**](https://en.wikipedia.org/wiki/Stack_overflow)
* Attempting to execute a program that does not compile correctly. (Some compilers will output an [executable file](https://en.wikipedia.org/wiki/Executable_file) despite the presence of compile-time errors.)

37.**Error handling in the system, Core dumps**, etc

A core dump (in Unix parlance), memory dump, or system dump[1] consists of the recorded state of the working memory of a computer program at a specific time, generally when the program has crashed or otherwise terminated abnormally.

Core dumps are often used to assist in diagnosing and debugging errors in computer programs.

A core dump represents the complete contents of the dumped regions of the address space of the dumped process. Depending on the operating system, the dump may contain few or no data structures to aid interpretation of the memory regions. In these systems, successful interpretation requires that the program or user trying to interpret the dump understands the structure of the program's memory use.

A debugger can use a symbol table, if one exists, to help the programmer interpret dumps, identifying variables symbolically and displaying source code; if the symbol table is not available, less interpretation of the dump is possible, but there might still be enough possible to determine the cause of the problem. There are also special-purpose tools called dump analyzers to analyze dumps. One popular tool, available on many operating systems, is the GNU binutils' objdump.

**38. Dangling Pointer and Memory Leak**A [**dangling pointer**](http://en.wikipedia.org/wiki/Dangling_pointer) **points to memory that has already been freed. The storage is no longer allocated**. Trying to access it might cause a Segmentation fault.  
Common way to end up with a dangling pointer:

char\* func()

{

char str[10];

strcpy(str,"Hello!");

return(str);

}

//returned pointer points to str which has gone out of scope.

You are returning an address which was a local variable, which would have gone out of scope by the time control was returned to the calling function. **(Undefined behaviour)**

Another common dangling pointer example is an access of a memory location via pointer, after free has been **explicitly** called on that memory.

int \*c = malloc(sizeof(int));

free(c);

\*c = 3; //writing to freed location!

A [**memory leak**](http://www.ibm.com/developerworks/aix/library/au-toughgame/) is **memory which hasn't been freed, there is no way to access** (or free it) now, as there are no ways to get to it anymore.

(E.g. a pointer which **was** the only reference to a memory location **dynamically allocated** (and not freed) which points somewhere else now.)

void func(){

char \*ch;

ch = (char\*) malloc(10);

}

//ch not valid outside, no way to access malloc-ed memory

Char-ptr ch is a local variable that goes out of scope at the end of the function, leaking the dynamically allocated **10 bytes**.

**Memory Leak in C++**

<https://stackoverflow.com/questions/6261201/how-to-find-memory-leak-in-a-c-code-project>

**35**.**What is stack overflow** ?

**Stack Overflow**:

In software, **a stack overflow occurs if the call stack pointer exceeds the stack bound**. The call stack may consist of a limited amount of address space, often determined at the start of the program. The size of the call stack depends on many factors, including the programming language, machine architecture, multi-threading, and amount of available memory. **When a program attempts to use more space than is available on the call stack (that is, when it attempts to access memory beyond the call stack's bounds,** which is essentially a buffer overflow), the stack is said to overflow, typically resulting in a program crash.  
  
The **most common cause of stack overflow is excessively deep or infinite recursion**, in which a function calls itself so many times that the space needed to store the variables and information associated with each call is more than can fit on the stack  
int foo()

{

return foo();

}

The function foo, when it is invoked, continues to invoke itself, allocating additional space on the stack each time, until the stack overflows resulting in a segmentation fault.  
  
36.**Buffer overflows and impacts/problems from that**  
  
**Buffer Overflow**:

A buffer overflow condition exists **when a program attempts to put more data in a buffer than it can hold** or when a program attempts to put data in a memory area past a buffer. In this case, a buffer is a sequential section of memory allocated to contain anything from a character string to an array of integers. **Writing outside the bounds of a block of allocated memory can corrupt data, crash the program, or cause the execution of malicious code**.

Buffer overflows are not easy to discover and even when one is discovered, it is generally extremely difficult to exploit. Nevertheless, attackers have managed to identify buffer overflows in a staggering array of products and components.

**In a classic buffer overflow exploit, the attacker sends data to a program, which it stores in an undersized stack buffer. The result is that information on the call stack is overwritten, including the function's return pointer. The data sets the value of the return pointer so that when the function returns, it transfers control to malicious code contained in the attacker's data.**

3**8. Friend class and function in C++**

Friend Class A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example a LinkedList class may be allowed to access private members of Node.

class Node{

private:

int key;

Node \*next;

/\* Other members of Node Class \*/

friend class LinkedList; // Now class LinkedList can

};

Friend Function Like friend class, **a friend function can be given special grant to access private and protected members.** A friend function can be:

a) A method of another class

b) A global function

class Node{

private:

int key;

Node \*next;

/\* Other members of Node Class \*/

friend int LinkedList::search(); // Only search() of linkedList

// can access internal members

};

Following are some important points about friend functions and classes:

1) Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.

**2) Friendship is not mutual. If a class A is friend of B, then B doesn’t become friend of A automatically.**

**3) Friendship is not inherited** (See this for more details)

4) The concept of friends is not there in Java.

**References in C++**

When a variable is declared as reference**, it becomes an alternative name for an existing variable**.

A variable **can be declared as reference by putting ‘&’ in the declaration**.

#include<iostream>

using namespace std;

int main()

{

int x = 10;

// ref is a reference to x.

int& ref = x;

// Value of x is now changed to 20

ref = 20;

cout << "x = " << x << endl;

// Value of x is now changed to 30

x = 30;

cout << "ref = " << ref << endl;

return 0;

}

**References Vs. Pointers**  
  
Consider the following code:

Pointers References

int i; int i;

int \*pi = &i; int &ri = i;

In both cases the situation is as follows:

Both pi and ri contain addresses that point to the location of i, but the difference lies in

the appearance between references and pointers when they are used in expressions. In

order to assign a value of 4 to i in both cases, we write:

\*pi = 4; ri = 4;

Note the, when using pointers, the address must be dereferenced using the \*, whereas, when using references, the address is dereferenced without using any operators at all!

The main effect of this is that the address can directly be manipulated if it is a pointer. We can do things such as:

pi++;

to increment to the next address. This is not possible using references. Therefore, to summarize, a pointer can point to many different objects during its lifetime, a reference can refer to only one object during its lifetime.  
  
References are less powerful than pointers

**1) Once a reference is created, it cannot be later made to reference another object; it cannot be reseated. This is often done with pointers.**

2) References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.

3) A reference must be initialized when declared. There is no such restriction with pointers

Due to the above limitations, references in C++ cannot be used for implementing data structures like Linked List, Tree, etc. In Java, references don’t have above restrictions, and can be used to implement all data structures. References being more powerful in Java, is the main reason Java doesn’t need pointers.

References are safer and easier to use:

1) Safer: Since references must be initialized, wild references like wild pointers are unlikely to exist. It is still possible to have references that don’t refer to a valid location.

2) Easier to use: References don’t need dereferencing operator to access the value. They can be used like normal variables. ‘&’ operator is needed only at the time of declaration. Also, members of an object reference can be accessed with dot operator (‘.’), unlike pointers where arrow operator (->) is needed to access members.

Together with the above reasons, there are few places like copy constructor argument where pointer cannot be used. Reference must be used pass the argument in copy constructor. Similarly references must be used for overloading some operators like ++.  
  
**What is function overloading?**  
  
Function overloading is a feature in C++ **where two or more functions can have the same name but different parameters.**

Function overloading can be considered as an example of static polymorphism feature in C++.

Following is a simple C++ example to demonstrate function overloading.

#include <iostream>

using namespace std;

void print(int i) {

cout << " Here is int " << i << endl;

}

void print(double f) {

cout << " Here is float " << f << endl;

}

void print(char\* c) {

cout << " Here is char\* " << c << endl;

}

int main() {

print(10);

print(10.10);

print("ten");

return 0;

}

**39.Difference between library call and a system call**  
<http://www.thegeekstuff.com/2012/07/system-calls-library-functions/?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+TheGeekStuff+(The+Geek+Stuff)>

Differences between a system and library call:

* **A library function is linked to the user program and executes in user space while a system call is not linked to a user program and executes in kernel space.**
* A library function execution time is counted in user level time while a system call execution time is counted as a part of system time.

User Space:

#include <time.h>

clock\_t start, end;

double cpu\_time\_used;

start = clock();

... /\* Do the work. \*/

end = clock();

cpu\_time\_used = ((double) (end - start)) / CLOCKS\_PER\_SEC;

Kernel Space:

static \_\_inline\_\_ unsigned long long rdtsc(void)

{

unsigned hi, lo;

\_\_asm\_\_ \_\_volatile\_\_ ("rdtsc" : "=a"(lo), "=d"(hi));

return ((unsigned long long) lo) | ((unsigned long long) hi)<<32;

}

* Library functions can be debugged easily using a debugger while System calls cannot be debugged as they are executed by the kernel.

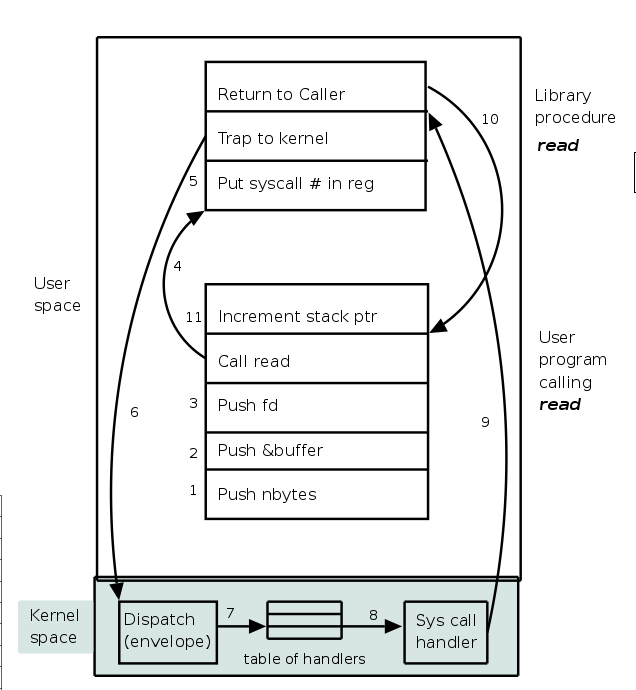
The following actions occur when the user executes the (Unix) system call

count = read(fd,buffer,nbytes)

which reads up to nbytes from the file described by fd into buffer. The actual number of bytes read is returned (it might be less than nbytes if, for example, an eof was encountered).

1. Push third parameter on to the stack.
2. Push second parameter on to the stack.
3. Push first parameter on to the stack.
4. Call the library routine, which involves pushing the return address on to the stack and jumping to the routine.
5. Machine/OS dependent actions. One is to put the system call number for read in a well defined place, e.g., a specific register. This requires assembly language.
6. Trap to the kernel. This enters the operating system proper and shifts the computer to privileged mode. Assembly language is again used.
7. The envelope uses the system call number to access a table of pointers to find the handler for this system call.
8. The read system call handler processes the request (see below).
9. Some magic instruction returns to user mode and jumps to the location right after the trap.
10. The library routine returns (there is more; e.g., the count must be returned).
11. The stack is popped (ending the function invocation of read).

A major complication is that the system call handler may block. Indeed, the read system call handler is likely to block.



40.Write a program to implement **memcpy() / memmov()** on your own –

memmove() offers guaranteed behavior if the source and destination arguments overlap. memcpy() makes no such guarantee, and may therefore be more efficient to implement. It's always safer to use memmove().   
*void \*mymemmove(void \*dest, const void \*src, size\_t size)*

*{*

char \*d =(char \*)dest;   
char \*s =(char \*)src;   
  
if(s == d)   
return dest;   
  
if(s < d)

{   
 //copy from back   
 s=s+size-1;   
 d=d+size-1;   
 while(size--)   
 {   
 \*d-- = \*s--;   
 }   
}   
else

{   
 //copy from front   
 while(size--)   
 \*d++=\*s++;   
}   
return dest;

*}*

*void \*mymemcpy(void \*to, const void \*from, size\_t size)*

*{*

*while (size-- != 0){*

*\*p1++ = \*p2++; }*

*return (to);*

*}*

43.Implement **printf** in C

*#include<stdio.h>*

*#include<stdarg.h>*

*main()*

*{*

*void myprintf(char \*,...);*

*char \* convert(unsigned int, int);*

*int i=65;*

*char str[]="This is my string";*

*myprintf("\nMessage = %s%d%x",str,i,i);*

*}*

*void myprintf(char \* frmt,...){*

*char \*p;*

*int i;*

*unsigned u;*

*char \*s;*

*va\_list argp;*

*va\_start(argp, fmt);*

*p=fmt;*

*for(p=fmt; \*p!='\0';p++)*

*{*

*if(\*p=='%')*

*{*

*putchar(\*p);*

*continue;*

*}*

*p++;*

*switch(\*p)*

*{*

*case 'c' : i=va\_arg(argp,int);*

*putchar(i);*

*break;*

*case 'd' : i=va\_arg(argp,int);*

*if(i<0){*

*i=-i;*

*putchar('-'); }*

*puts(convert(i,10));*

*break;*

*case 'o': i=va\_arg(argp,unsigned int);*

*puts(convert(i,8));*

*break;*

*case 's': s=va\_arg(argp,char \*);*

*puts(s);*

*break;*

*case 'u': u=va\_arg(argp,argp, unsigned int);*

*puts(convert(u,10));*

*break;*

*case 'x': u=va\_arg(argp,argp, unsigned int);*

*puts(convert(u,16));*

*break;*

*case '%': putchar('%');break;*

*}*

*}*

*va\_end(argp);*

*}*

*char \*convert(unsigned int, int)*

*{*

*static char buf[33];*

*char \*ptr;*

*ptr=&buf[sizeof(buff)-1];*

*\*ptr='\0';*

*do*

*{*

*\*--ptr="0123456789abcdef"[num%base];*

*num/=base;*

*}while(num!=0);*

*return(ptr);*

*}*

**44.Write your own sizeof()**

Here is an implementation.

***#define my\_sizeof(var) (char \*)(&var+1)-(char\*)(&var)***

**49.Questions on Call back functions**

**50.New and delete in C++**

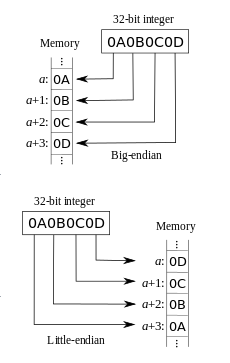
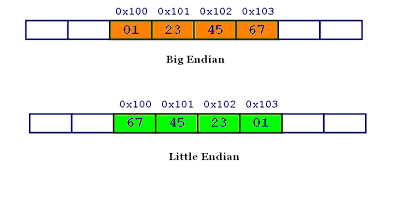
Dynamic memory is allocated using operator new. new is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets []. It returns a pointer to the beginning of the new block of memory allocated. Its syntax is:   
**int \*a = new int; //for single  
int \*b = new int[5]; //for array**

In most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of operator delete, whose syntax is:

**delete a; // for deleting single element  
delete[] b; //for deleting and array**

55.**Little Endian and Big Endian**  
Little and big endian are two ways of storing multibyte data-types (int, float, etc). In little endian machines, last byte of binary representation of the multibyte data-type is stored first. On the other hand, in big endian machines, first byte of binary representation of the multibyte data-type is stored first.

A variable x withvalue 0x01234567 will be stored as following.

[](http://4.bp.blogspot.com/_IEmaCFe3y9g/SO3GGEF4UkI/AAAAAAAAAAc/z7waF2Lwg0s/s1600-h/lb.GIF)

#include <stdio.h>

int main() {

unsigned int i = 1;

**char \*c = (char\*)&i**;

**if (\*c)**

**printf("Little endian");**

**else**

**printf("Big endian");**

getchar();

return 0;

}

# **this pointer in C++**

The ‘this’ pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. ‘this’ pointer is a constant pointer that holds the memory address of the current object. ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name).

For a class X, the type of this pointer is ‘X\* const’. Also, if a member function of X is declared as const, then the type of this pointer is ‘const X \*const’

Following are the situations where ‘this’ pointer is used:

**1) When local variable’s name is same as class member’s name**

|  |
| --- |
| #include<iostream>  using namespace std;  /\* local variable is same as a member's name \*/  class Test{  private:     int x;  public:     void setX (int x)     {         // The 'this' pointer is used to retrieve the object's x         // hidden by the local variable 'x'         this->x = x;     }     void print() { cout << "x = " << x << endl; }  };  int main()  {     Test obj;     int x = 20;     obj.setX(x);     obj.print();     return 0;} |

Output:

x = 20

For constructors, [initializer list](http://www.geeksforgeeks.org/archives/13797) can also be used when parameter name is same as member’s name.  
  
**2) To return reference to the calling object**

|  |
| --- |
| /\* Reference to the calling object can be returned \*/  Test& Test::func ()  {     // Some processing     return \*this;  } |

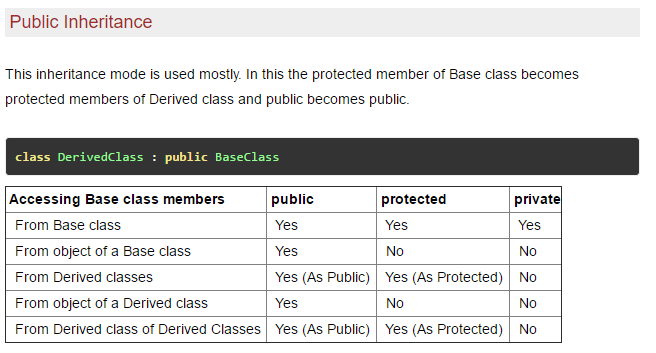
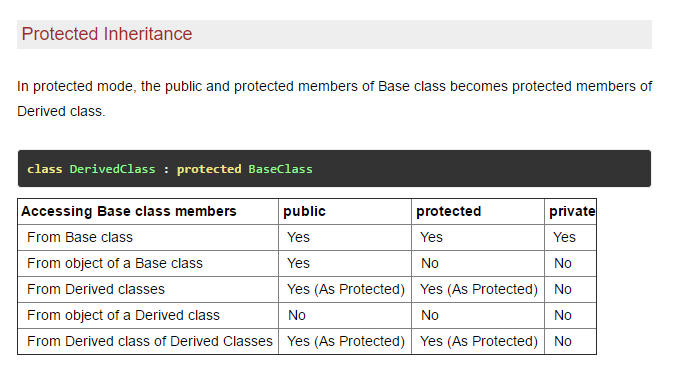
When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

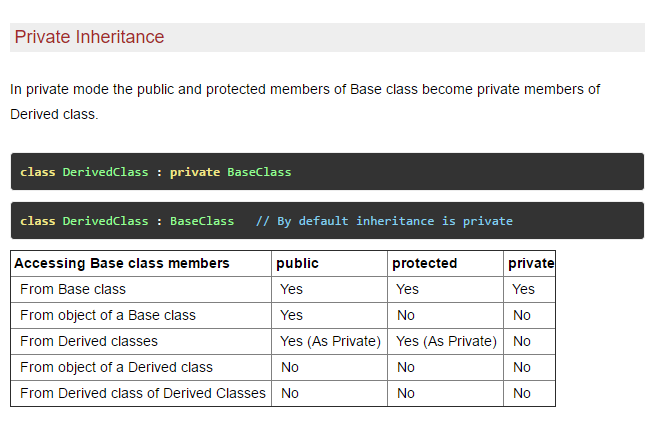
|  |
| --- |
| #include<iostream>  using namespace std;  class Test  {  private:    int x;    int y;  public:    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }    Test &setX(int a) { x = a; return \*this; }    Test &setY(int b) { y = b; return \*this; }    void print() { cout << "x = " << x << " y = " << y << endl; }  };  int main()  {    Test obj1(5, 5);    // Chained function calls.  All calls modify the same object    // as the same object is returned by reference    obj1.setX(10).setY(20);     obj1.print();    return 0;  } |

Output:

x = 10 y = 20

**Inheritance, call order for constructor**



# **Order of Constructor/ Destructor Call in C++**

**Prerequisite:** [Constructors](http://www.geeksforgeeks.org/constructors-c/)  
Whenever we create an object of a class, the default constructor of that class is invoked automatically to initialize the members of the class.

If we inherit a class from another class and create an object of the derived class, the order of invocation is that the base class’s default constructor will be invoked first and then the derived class’s default constructor will be invoked.

**Why the base class’s constructor is called on creating an object of derived class?**

The data members and member functions of base class comes automatically in derived class based on the access specifier but the definition of these members exists in base class only. So when we create an object of derived class, all of the members of derived class must be initialized but the inherited members in derived class can only be initialized by the base class’s constructor as the definition of these members exists in base class only. This is why the constructor of **base class is called first to initialize all the inherited members.**

|  |
| --- |
| #include <iostream>  using namespace std;  // base class  class Parent  {      public:      // base class constructor      Parent()      {          cout << "Inside base class" << endl;      }  };    // sub class  class Child : public Parent  {      public:      //sub class constructor      Child()      {          cout << "Inside sub class" << endl;      }  };    // main function  int main() {      // creating object of sub class      Child obj;      return 0;  } |

Output:

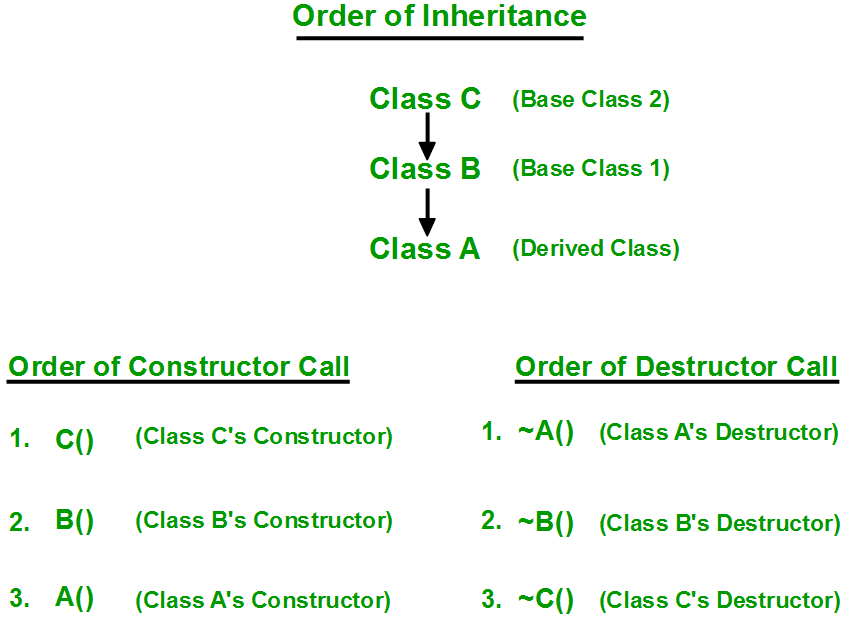
Inside base class

Inside sub class

**Order of constructor call for Multiple Inheritance**

For multiple inheritance order of constructor call is, the base class’s constructors are called in the order of inheritance and then the derived class’s constructor.

**Order of constructor and Destructor call for a given order of Inheritance**



**How to call the parameterized constructor of base class in derived class constructor?**

To call the parameterized constructor of base class when derived class’s parameterized constructor is called, you have to explicitly specify the base class’s parameterized constructor in derived class as shown in below program:

|  |
| --- |
| #include <iostream>  using namespace std;  // base class  class Parent  {      public:      // base class's parameterized constructor  **Parent(int i)**      {  int x =i;          cout << "Inside base class's parameterized constructor" << endl;      }  };  // sub class  class Child : public Parent  {      public:    **// sub class's parameterized constructor**  **Child(int j): Parent(j)**      {          cout << "Inside sub class's parameterized constructor" << endl;      }  };  // main function  int main() {        // creating object of class Child  **Child obj1(10);**      return 0;  } |

Output:

Inside base class's parameterized constructor

Inside sub class's parameterized constructor

**Important Points**:

* **Whenever the derived class’s default constructor is called, the base class’s default constructor is called automatically.**
* **To call the parameterized constructor of base class inside the parameterized constructor of sub class, we have to mention it explicitly.**
* **The parameterized constructor of base class cannot be called in default constructor of sub class, it should be called in the parameterized constructor of sub class.**

# **Destructors in C++**

**What is destructor?**  
Destructor is a member function which destructs or deletes an object.

**When is destructor called?**  
A destructor function is called automatically when the object goes out of scope:  
(1) the function ends  
(2) the program ends  
(3) a block containing local variables ends  
(4) a delete operator is called

**How destructors are different from a normal member function?**  
Destructors have same name as the class preceded by a tilde (~)  
Destructors don’t take any argument and don’t return anything

It is always in the public portion of the code

|  |
| --- |
| class String  {  private:      char \*s;      int size;  public:      String(char \*); // constructor      ~String();      // destructor  };    String::String(char \*c)  {      size = strlen(c);  **s = new char[size+1];**      strcpy(s,c);  }    String::~String()  {  **delete []s;**  } |

**When do we need to write a user-defined destructor?**  
If we do not write our own destructor in class, compiler creates a default destructor for us. **The default destructor works fine unless we have dynamically allocated memory or pointer in class.** When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

**Can a destructor be virtual?**  
Yes, In fact, it is always a good idea to make destructors virtual in base class when we have a virtual function.

**Static and Dynamic Polymorphism**

C++ polymorphism means that a call to a member function will cause a different function to be executed

#include <iostream>

using namespace std;

class Shape {

protected:

int width, height;

public:

Shape( int a = 0, int b = 0) {

width = a;

height = b;

}

int area() {

cout << "Parent class area :" <<endl;

return 0;

}

};

class Rectangle: public Shape {

public:

Rectangle( int a = 0, int b = 0):Shape(a, b) { }

int area () {

cout << "Rectangle class area :" <<endl;

return (width \* height);

}

};

class Triangle: public Shape{

public:

Triangle( int a = 0, int b = 0):Shape(a, b) { }

int area () {

cout << "Triangle class area :" <<endl;

return (width \* height / 2);

}

};

// Main function for the program

int main( ) {

Shape \*shape;

Rectangle rec(10,7);

Triangle tri(10,5);

// store the address of Rectangle

shape = &rec;

// call rectangle area.

shape->area();

// store the address of Triangle

shape = &tri;

// call triangle area.

shape->area();

return 0;

}

When the above code is compiled and executed, it produces the following result:

Parent class area

Parent class area

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.

class Shape {

protected:

int width, height;

public:

Shape( int a = 0, int b = 0) {

width = a;

height = b;

}

virtual int area() {

cout << "Parent class area :" <<endl;

return 0;

}

};

It produces the following result:

Rectangle class area

Triangle class area

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in \*shape the respective area() function is called. As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

## Pure Virtual Functions

It's possible that you'd want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following:

class Shape {

protected:

int width, height;

public:

Shape( int a = 0, int b = 0) {

width = a;

height = b;

}

// pure virtual function

virtual int area() = 0;

};

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

**Late binding (Runtime Polymorphism)?**

Late binding is also known as Dynamic Binding or Runtime Binding.

Sometimes compiler can’t know which function will be called till program is executed (runtime). This is known as late binding.

Binding is the process which is used by the compiler to convert identifiers (such as variable and function names) into machine language addresses.

It is a mechanism in which the method called by an object gets associated by name in runtime. Late binding happens when virtual keyword is used in member function declaration.

**Mechanism of Late Binding? (VTABLE)**

C++ implementation of virtual functions uses a special form of late binding known as virtual table (VTable). When a class declares a virtual member function, most of the compilers add a hidden member variable that represents a pointer to Virtual Method Table (VMT or VTable). We will call this pointer as vptr. This table represents an array of pointers to virtual functions. At compile-time, there is no information about which function will be called. At runtime, pointers from Virtual Method Table will point to right functions.

Look on the following example:

class A

{

public:

virtual void function1() {};

virtual void function2() {};

};

class B : public A

{

public:

virtual void function1() {};

};

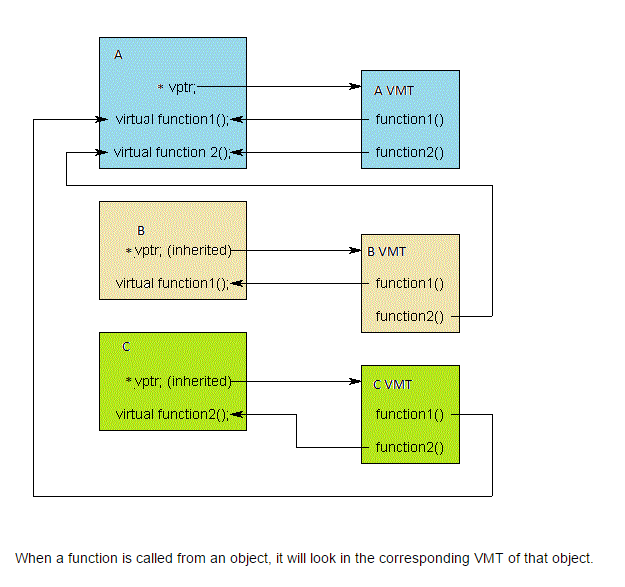
class C : public A

{

public:

virtual void function2() {};

};

When an object of any class is created, it has its own pointer to VMT:  
  


**Virtual Destructor?**

class A

{

public:

~A()

{

cout << "Base class destructor" << endl;

}

};

class B : public A

{

public:

~B()

{

cout << "B class destructor" << endl;

}

};

int main()

{

A\* a = new B; //It is only for runtime polymorphism

delete a;

}

**As you can see, pointer "a" points to an object of type B. When "a" is deleted, only the destructor of base class will be called. This means, that the object of a derived class will not be destroyed properly.**

This problem is easily resolved with Virtual Destructor.

class A

{

public:

virtual ~A() // virtual keyword added before the base class destructor

{

cout << "Base class destructor" << endl;

}

};

class B : public A

{

public:

~B()

{

cout << "Derived class destructor" << endl;

}

};

int main()

{

A\* a = new B;

delete a;

}

**When base class destructor is virtual, the derived class destructor is called first and after this, base class destructor is called:**

**Derived class destructor**

**Base class destructor**

**Abstract Class ?**

An Abstract class is a class that has **at least one pure virtual function**. A pure virtual function is a function without definition. To declare **pure virtual function** use the following syntax:

*virtual return-type function-name(parameter-list) = 0;*

For example, we can modify above baseClass to make it abstract class:

class baseClass

{

public:

baseClass(int val) :someValue(val)

{ }

//pure virtual function

**virtual void info() = 0;**

protected:

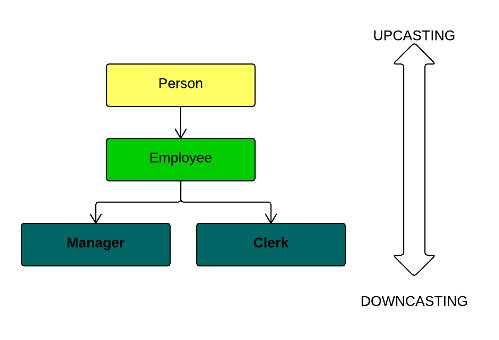
int someValue;

};

Now, info() is a pure virtual function. In this case when you use a pure virtual function then you must override it in the derived classes. You cannot create an object of the abstract class, but you can still use pointers of a base class to point an objects of the derived class.

**Upcasing And Downcasting**

Upcasting and downcasting are an important part of C++. Upcasting and downcasting gives a possibility to build complicated programs with a simple syntax. It can be achieved by using Polymorphism.

C++ allows that a derived class pointer (or reference) to be treated as base class pointer. This is upcasting.  
  
Downcasting is an opposite process, which consists in converting base class pointer (or reference) to derived class pointer. Upcasting and downcasting should not be understood as a simple casting of different data types. It can lead to a great confusion.  
In this topic, we will use the following hierarchy of classes:  
  
Upcasting is a process of treating a pointer or a reference of derived class object as a base class pointer. You do not need to upcast manually. You just need to assign derived class pointer (or reference) to base class pointer:

//pointer to base class object

Employee\* emp;

//object of derived class

Manager m1("Steve", "Kent", 3000, 0.2);

//implicit upcasting

emp = &m1;

When you use upcasting, the object is not changing. Nevertheless, when you upcast an object, you will be able to access only member functions and data members that are defined in the base class:

//It's ok

emp->FirstName;

emp->salary;

//Fails because upcasting is used

emp->getComm();

One of the biggest advantage of upcasting is the capability of writing generic functions for all the classes that are derived from the same base class. Look on example:

**void** congratulate(Employee\* emp)

{

cout << "Happy Birthday!!!" << endl;

emp->show();

emp->addBonus(200);

};

This function will work with all the classes that are derived from the Employee class. When you call it with objects of type Manager and Person, they will be automatically upcasted to Employee class:

//automatic upcasting

congratulate(&c1);

congratulate(&m1);

Try to run this program:  
Happy Birthday!!!  
First Name: Kevin Last Name: Jones  
Happy Birthday!!!  
First Name: Steve Last: Name Kent  
  
Downcasting is an opposite process for upcasting. It converts base class pointer to derived class pointer. Downcasting must be done manually. It means that you have to specify explicit type cast. Downcasting is not safe as upcasting. You know that a derived class object can be always treated as base class object. However, the opposite is not right. For example, a Manager is always a Person; But a Person is not always a Manager. It could be a Clerk too. You have to use an explicit cast for downcasting:

//pointer to base class object

Employee\* emp;

//object of derived class

Manager m1("Steve", "Kent", 3000, 0.2);

//implicit upcasting

emp = &m1;

//explicit downcasting from Employee to Manager

Manager\* m2 = (Manager\*)(emp);

This code compiles and runs without any problem, because emp points to an object of Manager class.  
What will happen, if we try to downcast a base class pointer that is pointing to an object of base class and not to an object of derived class? Try to compile and run this code:

Employee e1("Peter", "Green", 1400);

//try to cast an employee to Manager

Manager\* m3 = (Manager\*)(&e1);

cout << m3->getComm() << endl;

e1 object is not an object of Manager class. It does not contain any information about commission. That why such an operation can produce unexpected results.

**Stack overflow and underflow**

First, we will consider the situation where the allocated stack area is placed at the beginning of RAM. For example, assume we allocate **4096 bytes for the stack from 0x2000.0000 to 0x2000.0FFF**, see the left side of Figure 2.6. The SP is initialized to 0x2000.1000, and the stack is considered empty. **If the SP becomes less than 0x2000.0000 a stack overflow has occurred.** The stack overflow will cause a bus fault because there is nothing at address 0x1FFF.FFFC. **If the software tries to read from or write to any location greater than or equal to 0x2000.1000 then a stack underflow has occurred**. At this point the stack and global variables may exist at overlapping addresses. Stack underflow is a very difficult bug to recognize, because the first consequence will be unexplained changes to data stored in global variables.

### **Startup code**

One of the things that traditional software development tools do automatically is insert startup code: a small block of assembly language code that prepares the way for the execution of software written in a high-level language. Each high-level language has its own set of expectations about the runtime environment. For example, programs written in C use a stack. Space for the stack has to be allocated before software written in C can be properly executed. That is just one of the responsibilities assigned to startup code for C programs.

Most cross-compilers for embedded systems include an assembly language file called startup.asm, crt0.s (short for C runtime), or something similar. The location and contents of this file are usually described in the documentation supplied with the compiler.

Startup code for C programs usually consists of the following series of actions:

1. Disable all interrupts.
2. Copy any initialized data from ROM to RAM.
3. Zero the uninitialized data area.
4. Allocate space for and initialize the stack.
5. Initialize the processor’s stack pointer.
6. Call main.

Typically, the startup code will also include a few instructions after the call to main. These instructions will be executed only in the event that the high-level language program exits (i.e., the call to main returns). Depending on the nature of the embedded system, you might want to use these instructions to halt the processor, reset the entire system, or transfer control to a debugging tool.

Because the startup code is often not inserted automatically, the programmer must usually assemble it himself and include the resulting object file among the list of input files to the linker. He might even need to give the linker a special command-line option to prevent it from inserting the usual startup code. Working startup code for a variety of target processors can be found in a GNU package called libgloss .

## Locating

The tool that performs the conversion from relocatable program to executable binary image is called a locator. It takes responsibility for the easiest step of the build process. In fact, you have to do most of the work in this step yourself, by providing information about the memory on the target board as input to the locator. The locator uses this information to assign physical memory addresses to each of the code and data sections within the relocatable program. It then produces an output file that contains a binary memory image that can be loaded into the target.

Whether you are writing software for a general-purpose computer or an embedded system, at some point the sections of your relocatable program must be assigned actual addresses.

In some cases, there is a separate development tool, called a locator, to assign addresses. However, in the case of the GNU tools, this feature is built into the linker (ld). **The memory information required by the GNU linker can be passed to it in the form of a linker script. Such scripts are sometimes used to control the exact order of the code and data sections within the relocatable program.** But here, we want to do more than just control the order; we also want to establish the physical location of each section in memory.

Below is linker script file

ENTRY (main)

MEMORY

{

ram : ORIGIN = 0x00400000, LENGTH = 64M

rom : ORIGIN = 0x60000000, LENGTH = 16M

}

SECTIONS

{

data : /\* Initialized data. \*/

{

\_DataStart = . ;

\*(.data)

\_DataEnd = . ;

} >ram

bss : /\* Uninitialized data. \*/

{

\_BssStart = . ;

\*(.bss)

\_BssEnd = . ;

} >ram

text : /\* The actual instructions. \*/

{

\*(.text)

} >ram

}

This script informs the GNU linker’s built-in locator about the memory on the target board, which contains 64 MB of RAM and 16 MB of flash ROM. [[*3*](https://www.safaribooksonline.com/library/view/programming-embedded-systems/0596009836/ch04.html#ftn.ID-d1766e2359)**] The linker script file instructs the GNU linker to locate the data, bss, and textsections in RAM starting at address 0x00400000**. The first executable instruction is designated with the ENTRY command, which appears on the first line of the preceding example. In this case, the entry point is the function main.

Names in the linker command file that begin with an underscore (e.g.,\_DataStart) can be referenced similarly to ordinary variables from within your source code. The linker will use these symbols to resolve references in the input object files. So, for example, there might be a part of the embedded software (usually within the startup code) that copies the initial values of the initialized variables from ROM to the data section in RAM. The start and stop addresses for this operation can be established symbolically by referring to the addresses as \_DataStart and \_DataEnd.

A linker script can also use various commands to direct the linker to perform other operations.

The output of this final step of the build process is a binary image containing physical addresses for the specific embedded system. This executable binary image can be downloaded to the embedded system or programmed into a memory chip. You’ll see how to download and execute such memory images in the next chapter.

# **A Quick Look at Makefiles**

Makefiles might be a bit of a pain to set up, but they can be a great timesaver and a very powerful tool when building project files over and over (and over) again. Having a sample available can reduce the pain of setting up a makefile.

The basic layout for a makefile build rule is:

target: prerequisite

command

The target is what is going to be built, the prerequisite is a file that must exist before the target can be created, and the command is a shell command used to create the target. There can be multiple prerequisites on the target line (separated by white space) and/or multiple command lines. But be sure to put a tab, not spaces, at the beginning of every line containing a command.

Here’s a makefile for building our Blinking LED program:

XCC = arm-elf-gcc

LD = arm-elf-ld

CFLAGS = -g -c -Wall \\

-I../include

LDFLAGS = -Map blink.map -T viperlite.ld -N

all: blink.exe

led.o: led.c led.h

$(XCC) $(CFLAGS) led.c

blink.o: blink.c led.h

$(XCC) $(CFLAGS) blink.c

blink.exe: blink.o led.o viperlite.ld

$(LD) $(LDFLAGS) -o $@ led.o blink.o

clean:

-rm -f blink.exe \*.o blink.map

The first four statements in this makefile contain variables for use in the makefile. The variable names are on the left side of the equal sign. In this makefile, the respective variables do the following:

XCC

Defines the compiler executable program

LD

Defines the linker executable program

CFLAGS

Defines the flags for the compiler

LDFLAGS

Defines the flags for the linker

To execute the makefile’s build instructions, simply change to the directory that contains the makefile and enter the command:

# **make**

The make utility will search the current directory for a file named makefile. If your makefile has a different name, you can specify that on the command line following the -f option.

With the previous command, the make utility will make the first target it finds. You can also specify targets on the command line for the make utility. For example, because all is the default target in the preceding makefile, you can just as easily use the following command:

# **make all**

A target called clean is typically included in a makefile, with commands for removing old object files and executables, in order to allow you to create a fresh build. The command line for executing the clean target is:

# **make clean**

Keep in mind that we’ve presented a very basic example of the make utility and makefiles for a very basic project. The make utility contains very powerful tools within its advanced features that can benefit you when executing large and more complex projects.

**Oscilloscope or logic analyzer?**  
When given the choice between using a scope or a logic analyzer, many engineers will choose an oscilloscope. However, scopes have limited usefulness in some applications. Depending on what the user is trying to accomplish, a logic analyzer may yield more useful information.

***When to use a scope:***

* When it is required to observe small voltage excursions on the signal.
* When high time-interval accuracy is required.

***When to use a logic analyzer:***

* When one wishes to observe many signals at the same time.
* When it's necessary to look at signals in the system the same way hardware does.
* When it's required to trigger on a pattern of highs and lows on several lines and see the results.

A logic analyzer reacts the same way as a logic [circuit](http://www.pldesignline.com/encyclopedia/defineterm.jhtml?term=circuit&x=&y=) does when a single threshold is crossed by a signal in the system. It recognizes the signal to be either low or high. It can also trigger on patterns of highs and lows presented on these signals.

In general, you should use a logic analyzer when it's necessary to look at more lines than can be monitored on an oscilloscope. Logic analyzers are particularly useful for looking at time relationships or data on a [bus](http://www.pldesignline.com/encyclopedia/defineterm.jhtml?term=bus&x=&y=) – for example, a microprocessor address, data, or control bus. They can [decode](http://www.pldesignline.com/encyclopedia/defineterm.jhtml?term=decode&x=&y=) the information on microprocessor buses and present it in a meaningful form.

**Organization of a Stack Frame**

A stack frame consists of several elements, including:

*Return address*

The address in the program where the function is to return upon completion

*Storage for local data*

Memory allocated for local variables

*Storage for parameters*

Memory allocated for the function’s parameters

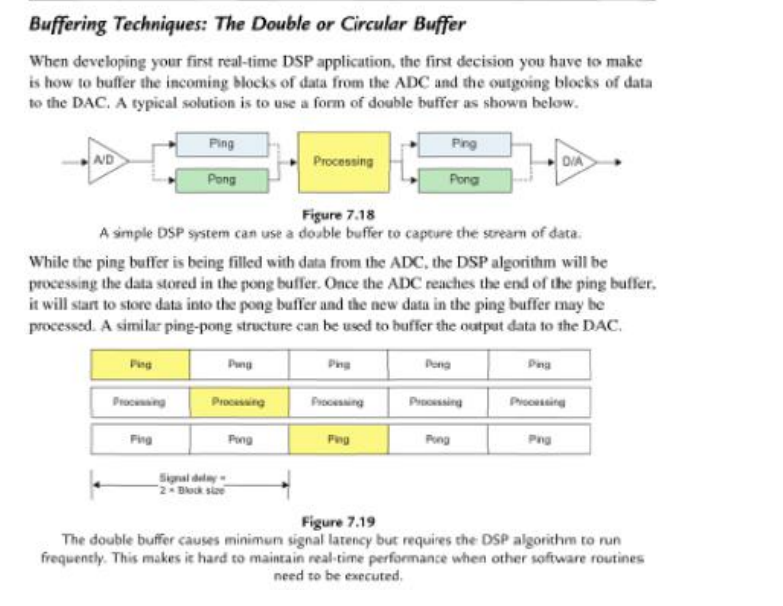
*Stack and base pointers*

Pointers used by the runtime system to manage the stack

The typical C programmer will not be concerned about the stack and base pointers used in support of a stack frame. However, understanding what they are and how they are used provides a more in-depth understanding of the program stack. A stack pointer usually points to the top of the stack. A stack base pointer (frame pointer)

is often present and points to an address within the stack frame, such as the return address. This pointer assists in accessing the stack frame’s elements. Neither of these pointers are C pointers. They are addresses used by the runtime system to manage the program stack. If the runtime system is implemented in C, then these pointers may be real C pointers.

**Ping Pong Buffer**



**Code Refactoring**

Code refactoring is the process of changing a computer program's internal structure without modifying its external functional behavior or existing functionality, in order to improve internal non-functional properties of the software, for example to improve code readability, to simplify code structure, to change code to adhere to a given programming paradigm, to improve maintainability, to improve performance, or to improve extensibility.

There are two general categories of benefits to the activity of refactoring.

1. Maintainability. It is easier to fix bugs because the source code is easy to read and the intent of its author is easy to grasp.[[4]](https://en.wikipedia.org/wiki/Code_refactoring#cite_note-martin-4) This might be achieved by reducing large monolithic routines into a set of individually concise, well-named, single-purpose methods. It might be achieved by moving a method to a more appropriate class, or by removing misleading comments.
2. Extensibility. It is easier to extend the capabilities of the application if it uses recognizable [design patterns](https://en.wikipedia.org/wiki/Design_patterns), and it provides some flexibility where none before may have existed.

List of refactoring techniques:

1. Techniques that allow for more [abstraction](https://en.wikipedia.org/wiki/Abstraction_(computer_science))
   1. [Encapsulate field](https://en.wikipedia.org/wiki/Field_encapsulation) – force code to access the field with getter and setter methods
   2. [Generalize type](https://en.wikipedia.org/wiki/Type_generalization) – create more general types to allow for more code sharing
   3. Replace type-checking code with state/strategy[[6]](https://en.wikipedia.org/wiki/Code_refactoring#cite_note-6)
   4. Replace conditional with [polymorphism](https://en.wikipedia.org/wiki/Polymorphism_(computer_science)) [[7]](https://en.wikipedia.org/wiki/Code_refactoring#cite_note-7)
2. Techniques for breaking code apart into more logical pieces
   1. Componentization breaks code down into reusable semantic units that present clear, well-defined, simple-to-use interfaces.
   2. [Extract class](https://en.wikipedia.org/wiki/Extract_class) moves part of the code from an existing class into a new class.
   3. Extract method, to turn part of a larger [method](https://en.wikipedia.org/wiki/Method_(computer_science)) into a new method. By breaking down code in smaller pieces, it is more easily understandable. This is also applicable to [functions](https://en.wikipedia.org/wiki/Function_(programming)).
3. Techniques for improving names and location of code
   1. Move method or move field – move to a more appropriate [class](https://en.wikipedia.org/wiki/Class_(computer_science)) or source file
   2. [Rename method](https://en.wikipedia.org/wiki/Rename_method) or rename field – changing the name into a new one that better reveals its purpose
   3. Pull up – in [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming) (OOP), move to a [superclass](https://en.wikipedia.org/wiki/Superclass_(computer_science))
   4. Push down – in OOP, move to a [subclass](https://en.wikipedia.org/wiki/Subclass_(computer_science))

**What is the use of const pointer ?**

const is a tool which you should use in pursuit of a very important C++ concept:

Find bugs at compile-time, rather than run-time, by getting the compiler to enforce what you mean.

Even though it doesn't change the functionality, adding const generates a compiler error when you're doing things you didn't mean to do. Imagine the following typo:

void foo(int\* ptr) { ptr = 0;// oops, I meant \*ptr = 0 }

If you use int\* const, this would generate a compiler error because you're changing the value to ptr. Adding restrictions via syntax is a good thing in general. Just don't take it too far -- the example you gave is a case where most people don't bother using const.

How to convert string const char \*

If you just want to pass a [std::string](http://en.cppreference.com/w/cpp/string/basic_string) to a function that needs const char\* you can use

std::string str; const char \* c = str.c\_str();

If you want to get a writable copy, like char \*, you can do that with this:

std::string str; char \* writable = new char[str.size() + 1];

std::copy(str.begin(), str.end(), writable);

writable[str.size()] = '\0'; // don't forget the terminating 0 // don't forget to free the string after finished using it

delete[] writable;

### std::vector

This is the standard way (does not require any external library). You use [std::vector](http://en.cppreference.com/w/cpp/container/vector), which completely manages the memory for you.

std::string str; std::vector<char> writable(str.begin(), str.end()); writable.push\_back('\0');

# **Pointer expressions: \*ptr++, \*++ptr and ++\*pt**

Here's a detailed explanation which I hope will be helpful. Let's begin with your program, as it's the simplest to explain.

int main() { const char \*p = "Hello"; while(\*p++) printf("%c",\*p); return 0; }

The first statement:

const char\* p = "Hello";

declares p as a pointer to char. When we say "pointer to a char", what does that mean? It means that the value of p is the address of a char; p tells us where in memory there is some space set aside to hold a char.

The statement also initializes p to point to the first character in the string literal "Hello". For the sake of this exercise, it's important to understand p as pointing not to the entire string, but only to the first character, 'H'. After all, p is a pointer to one char, not to the entire string. The value of p is the address of the 'H' in "Hello".

Then you set up a loop:

while (\*p++)

What does the loop condition \*p++ mean? Three things are at work here that make this puzzling (at least until familiarity sets in):

1. The precedence of the two operators, postfix ++ and indirection \*
2. The value of a postfix increment expression
3. The side effect of a postfix increment expression

1. Precedence. A quick glance at the precedence table for operators will tell you that postfix increment has a higher precedence (16) than dereference / indirection (15). This means that the complex expression \*p++ is going to be grouped as: \*(p++). That is to say, the \* part will be applied to the value of the p++ part. So let's take the p++ part first.

2. Postfix expression value. The value of p++ is the value of p *before the increment*. If you have:

int i = 7; printf ("%d\n", i++); printf ("%d\n", i);

the output will be:

7 8

because i++ evaluates to i before the increment. Similarly p++ is going to evaluate to the current value of p. As we know, the current value of p is the address of 'H'.

So now the p++ part of \*p++ has been evaluated; it's the current value of p. Then the \* part happens. \*(current value of p) means: access the value at the address held by p. We know that the value at that address is 'H'. So the expression \*p++ evaluates to 'H'.

Now hold on a minute, you're saying. If \*p++ evaluates to 'H', why doesn't that 'H' print in the above code? That's where *side effects* come in.

3. Postfix expression side effects. The postfix ++ has the *value* of the current operand, but it has the *side effect* of incrementing that operand. Huh? Take a look at that int code again:

int i = 7; printf ("%d\n", i++); printf ("%d\n", i);

As noted earlier, the output will be:

7 8

When i++ is evaluated in the first printf(), it evaluates to 7. But the C standard guarantees that at some point before the second printf() begins executing, the *side effect* of the ++ operator will have taken place. That is to say, before the second printf() happens, i will have been incremented as a result of the ++ operator in the first printf(). This, by the way, is one of the few guarantees the standard gives about the timing of side effects.

In your code, then, when the expression \*p++is evaluated, it evaluates to 'H'. But by the time you get to this:

printf ("%c", \*p)

that pesky side-effect has occurred. p has been incremented. Whoa! It no longer points to 'H', but to one character past 'H': to the 'e', in other words. That explains your cockneyfied output:

ello

Hence the chorus of helpful (and accurate) suggestions in the other answers: to print the Received Pronunciation "Hello" and not its cockney counterpart, you need something like

while (\*p) printf ("%c", \*p++);

So much for that. What about the rest? You ask about the meanings of these:

\*ptr++ \*++ptr ++\*ptr

We just talked about the first, so let's look at the second: \*++ptr.

We saw in our earlier explanation that postfix increment p++ has a certain *precedence*, a *value*, and a *side effect*. The prefix increment ++p has the same *side effect* as its postfix counterpart: it increments its operand by 1. However, it has a different *precedence* and a different *value*.

The prefix increment has lower precedence than the postfix; it has precedence 15. In other words, it has the same precedence as the dereference / indirection operator \*. In an expression like

\*++ptr

what matters is not precedence: the two operators are identical in precedence. So *associativity* kicks in. The prefix increment and the indirection operator have right-left associativity. Because of that associativity, the operand ptr is going to be grouped with the rightmost operator ++ before the operator more to the left, \*. In other words, the expression is going to be grouped \*(++ptr). So, as with \*ptr++ but for a different reason, here too the \* part is going to be applied to the value of the ++ptr part.

So what is that value? The value of the prefix increment expression is the value of the operand *after the increment*. This makes it a very different beast from the postfix increment operator. Let's say you have:

int i = 7; printf ("%d\n", ++i); printf ("%d\n", i);

The output will be:

8 8

... different from what we saw with the postfix operator. Similarly, if you have:

const char\* p = "Hello"; printf ("%c ", \*p); // note space in format string printf ("%c ", \*++p); // value of ++p is p after the increment printf ("%c ", \*p++); // value of p++ is p before the increment printf ("%c ", \*p); // value of p has been incremented as a side effect of p++

the output will be:

H e e l // good dog

Do you see why?

Now we get to the third expression you asked about, ++\*ptr. That's the trickiest of the lot, actually. Both operators have the same precedence, and right-left associativity. This means the expression will be grouped ++(\*ptr). The ++ part will be applied to the value of the \*ptr part.

So if we have:

char q[] = "Hello"; char\* p = q; printf ("%c", ++\*p);

the surprisingly egotistical output is going to be:

I

What?! Okay, so the \*p part is going to evaluate to 'H'. Then the ++ comes into play, at which point, it's going to be applied to the 'H', not to the pointer at all! What happens when you add 1 to 'H'? You get 1 plus the ASCII value of 'H', 72; you get 73. Represent that as a char, and you get the char with the ASCII value of 73: 'I'.

That takes care of the three expressions you asked about in your question. Here is another, mentioned in the first comment to your question:

(\*ptr)++

That one is interesting too. If you have:

char q[] = "Hello"; char\* p = q; printf ("%c", (\*p)++); printf ("%c\n", \*p);

it will give you this enthusiastic output:

HI

What's going on? Again, it's a matter of *precedence*, *expression value*, and *side effects*. Because of the parentheses, the \*p part is treated as a primary expression. Primary expressions trump everything else; they get evaluated first. And \*p, as you know, evaluates to 'H'. The rest of the expression, the ++ part, is applied to that value. So, in this case, (\*p)++ becomes 'H'++.

What is the value of 'H'++? If you said 'I', you've forgotten (already!) our discussion of value vs. side effect with postfix increment. Remember, 'H'++ evaluates to the *current value of* 'H'. So that first printf() is going to print 'H'. Then, as a *side effect*, that 'H' is going to be incremented to 'I'. The second printf() prints that 'I'. And you have your cheery greeting.

All right, but in those last two cases, why do I need

char q[] = "Hello"; char\* p = q;

Why can't I just have something like

/\*const\*/ char\* p = "Hello"; printf ("%c", ++\*p); // attempting to change string literal!

Because "Hello" is a string literal. If you try ++\*p, you're trying to change the 'H' in the string to 'I', making the whole string "Iello". In C, string literals are read-only; attempting to modify them invokes undefined behavior. "Iello" is undefined in English as well, but that's just coincidence.

Conversely, you can't have

char p[] = "Hello"; printf ("%c", \*++p); // attempting to modify value of array identifier!

Why not? Because in this instance, p is an array. An array is not a modifiable l-value; you can't change where p points by pre- or post- increment or decrement, because the name of the array works as though it's a constant pointer. (That's not what it actually is; that's just a convenient way to look at it.)

To sum up, here are the three things you asked about:

\*ptr++ // effectively dereferences the pointer, then increments the pointer \*++ptr // effectively increments the pointer, then dereferences the pointer ++\*ptr // effectively dereferences the pointer, then increments dereferenced value

And here's a fourth, every bit as much fun as the other three:

(\*ptr)++ // effectively forces a dereference, then increments dereferenced value

The first and second will crash if ptr is actually an array identifier. The third and fourth will crash if ptr points to a string literal.

There you have it. I hope it's all crystal now. You've been a great audience, and I'll be here all week.

**Cache**

## Spatial locality: If one item is referenced, the likelihood of other address close by will be referenced soon

## Temporal locality: One item that is referenced at one point in time it tends to be referenced soon again

**Two matrices can be stored in either row major or column major order in contiguous memory. Does the time complexity of computing their multiplication vary depending on the storage scheme? That is, I want to know whether it will work faster if stored in row major or column major order.**

Transposition of *one* matrix source has been used to accelerate matrix multiply, but matrix multiply is the classic use case for blocking to maximize reuse within a level of the memory hierarchy

No, the complexity remains the same.

The main difference between row-major and column-major order is memory access patterns - for example if you iterate by column then row-major order would be jumping around memory, which is bad for CPUs because they cannot read-ahead/cache memory.

This is because transferring data from RAM to the CPU on modern processors is relatively very slow, compared with the speed of the processors. Thus most of the time would be spent on waiting for the memory to arrive.

To solve this problem, memory accesses are answered in in bunches, with a lot of extra data coming to the CPU's caches. This makes it important to use "good" memory access patterns, so that the machine can try and predict what you will use next, and therefore cache it, and thus eliminate the transfer-wait-times as much as possible. But if you jump around memory instead of going in order, the machine will potentially have to wait for each access.

The wait time is bounded by a constant however, so the complexity remains the same. And of course, doing it in the correct order does not decrease the complexity

**#define vs inline method**

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.

**ARM Processor**

The ARM processor uses an approach known as a reduced instruction set. This approach provides many advantages for embedded systems requiring performance yet small power consumption.

The RISC, Reduced Instruction Set Computer was developed by the British company ARM Holdings - ARM standing for Advanced RISC Machine.

The core IP is normally provided and embedded in chips manufactured by others and in this way ARM itself does not manufacture any chips of its own.

## ARM basics

The key element of an ARM RISC processor that the reduced instruction set means that the processor can run on using fewer transistors and hence reduce current consumption. The current consumption of any processor is a key attribute for many portable applications because it directly reflects into battery life.

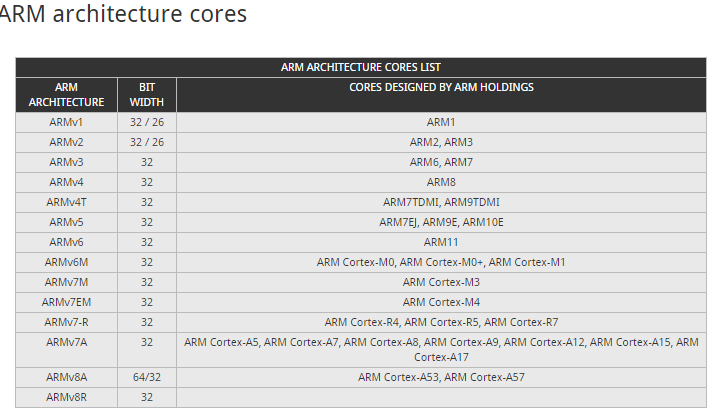
Accordingly, the ARM processor is well known for its low energy consumption and use in may portable devices.

Current versions use 32-bit instructions with 32-bit addressed 1 byte wide memory which is effectively reduced to just over 24 bit addressing due to 4 byte alignment, with some addressing reserved in byte wise allocation for Memory Mapped I/O, but accommodates 16-bit instructions for economy and can also handle Java byte codes which use 32-bit addresses.

**ARM Architecture**

The ARM processor architecture forms the basis of all ARM processors

Although the ARM architecture has evolved over time, it still adopts the same basic processor architecture adopted by the early ARM devices.



The ARM architecture has evolved over time. ARM architecture, ARMv7, defines the architecture for the first of the Cortex series of cores, for which there are three architecture "profiles".

* ***A-profile:***   Application profile relates to Cortex-A series.
* ***R-profile:***   Real-time profile relates to Cortex-R series.
* ***M-profile:***   Microcontroller profile relates to Cortex-M series.

The ARM architecture supports implementations across a wide range of performance points, establishing it as the leading architecture in many market segments. The ARM architecture supports a very broad range of performance points leading to very small implementations of ARM processors, and very efficient implementations of advanced designs using state of the art micro-architecture techniques. Implementation size, performance, and low power consumption are key attributes of the ARM architecture.

ARM developed architecture extensions to provide support for Java acceleration (Jazelle), security (TrustZone), SIMD, and Advanced SIMD (NEON) technologies. The ARMv8-architecture adds a Cryptographic extension as an optional feature.

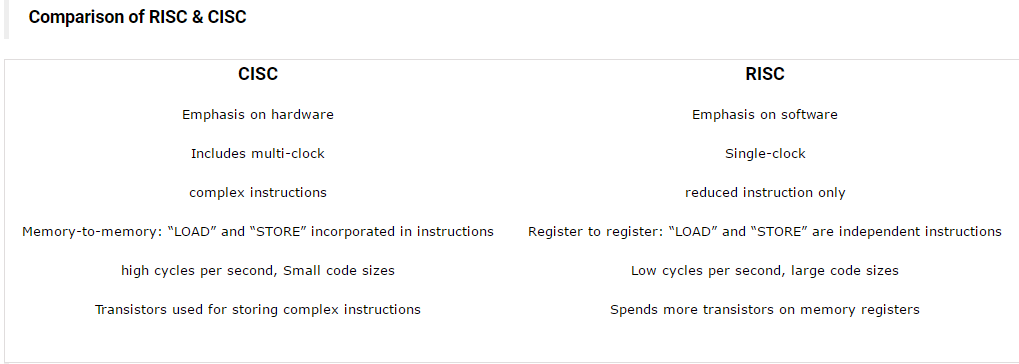
The ARM architecture is similar to a Reduced Instruction Set Computer (RISC) architecture, as it incorporates these typical RISC architecture features:

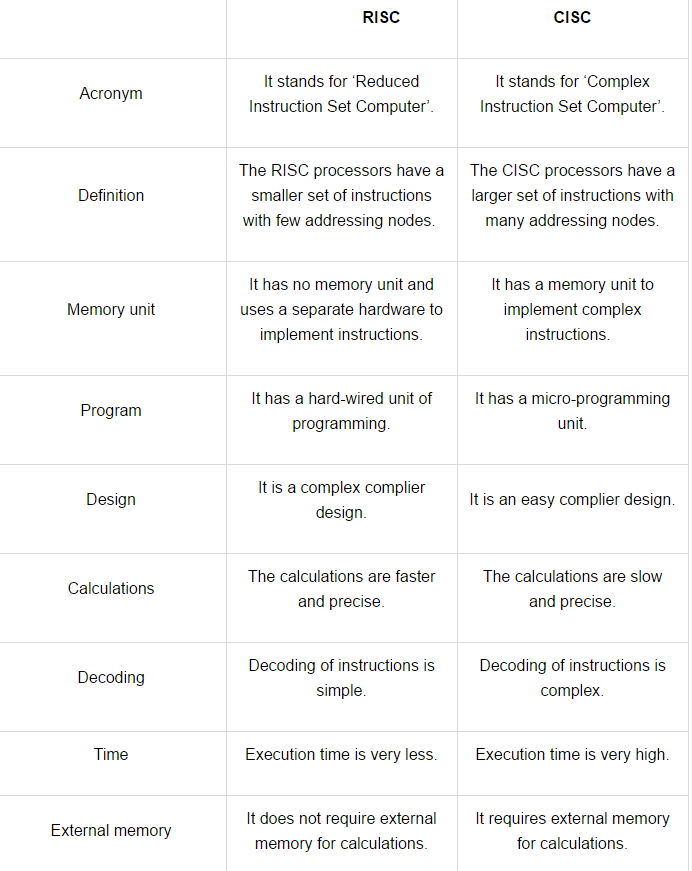
A uniform register file load/store architecture, where data processing operates only on register contents, not directly on memory contents.

Simple addressing modes, with all load/store addresses determined from register contents and instruction fields only.

Enhancements to a basic RISC architecture enable ARM processors to achieve a good balance of high performance, small code size, low power consumption and small silicon area.

**RISC Vs. CISC**





**Computer Booting**

When the computer is switched on, it’s of no use because the data stored in the memory(RAM) is garbage and there is no Operating System running. The first thing motherboard does is to initialize its own firmware and get the CPU running. Some of the CPU registers including Instruction Pointer (EIP) have predefined values. In x86 systems the initial value of the EIP is 0xfffffff0 and the instruction stored at this memory location is executed. The instruction is JMP (JUMP) to a Read Only Memory (ROM) which contains the BIOS and its code starts executing.

**Functions of BIOS**  
**POST** (Power On Self Test) to ensure that the various components present in the system are functioning properly. If video card is missing or not functioning properly, motherboard emit beeps since error cannot be displayed. Beeps are emitted according to Beep Codes of the motherboard and it varies from one motherboard to other. A comprehensive list of beep codes can be found here. If the computer passes the video card test, manufacturer logo is printed on the screen.

It initializes the various hardware devices. It is an important process so as to ensure that all the devices operate smoothly without any conflicts. BIOSes following ACPI create tables describing the devices in the computer.

It looks for an Operating System to load. Typically, the BIOS will search it in Hard Drives, CD-ROMs, floppy disks etc. The actual search order can be configured by the user by changing Boot Order in BIOS settings. If BIOS cannot find a bootable operating system it displays an error message “Non-System Disk Error”.

Generally, the operating system is present in the hard disk. We confine our discussion to how operating system boots from the hard disk.

**Master Boot Record**  
The first sector of the hard disk is called Master Boot Record (MBR). The structure of MBR is operating system independent. It is of 512 bytes and it has mainly two components. The first 446 bytes contain a special program called Bootstrap Loader. The next 64 bytes contains a partition table. A partition table stores all the information about the partitions in a hard disk and file system types (a file system describes how data will be stored and retrieved from the partition). A partition table is required to boot up the operating system. The last two bytes of MBR contains a magic number AA55. It is used to classify whether the MBR is valid or not. An invalid magic number indicates that the MBR is corrupt and machine will not be able to boot.

**Bootstrap Loader**  
Bootstrap loader or the boot loader contains the code to load an operating system. Earlier Linux distributions used LILO (LInux Loader) bootloader. Today, most of the distributions use GRUB (GRand Unified Bootloader) which has many advantages over LILO. BIOS loads the bootstrap loader into the memory (RAM) and starts executing the code.

Boot loader of traditional operating systems like Windows 98 used to identify an active partition in the hard disk by looking at the active flag of partition table and loading its boot sector into the memory. Boot sector is the first sector of each partition in contrast to MBR which is the first sector of the hard disk. The boot sector is of 512 bytes in memory and contains code to boot an operating system in that partition. However boot loaders like GRUB and LILO are more robust and boot process is not so straight forward.

Booting an operating system with GRUB is a two stage process: stage 1 and stage 2. In some cases an intermediate stage 1.5 may also be used to load stage 2 from an appropriate file system. Stage 1 is the boot loader code itself and its task is to only call the stage 2 which contains the main code. This is done because of the tiny size of the stage 1 (512 bytes). GRUB stage 2 loads the Linux Kernel and initramfs into the memory.

Kernel is the core component of an operating system. It has complete control of all the things happening in a system. It is the first part of the operating system to load into the memory and remains there throughout the session.

To access a file system it must be first mounted. When kernel is loaded into the memory none of the file system is mounted and hence initial RAM based file system (initramfs) is required by kernel to execute programs even before the root file system is mounted. Kernel executes a init (initialization) program which has pid=1. It is a daemon process and continues to run until the computer is shut down. It also load the modules and drivers required to mount the root file system. Linux stores information about the major file systems in a file /etc/fstab

**init**  
init is the last step of the kernel boot sequence. It looks for the file /etc/inittab to see if there is an entry for initdefault. It is used to determine initial run-level of the system. A run-level is used to decide the initial state of the operating system.  
Some of the run levels are:

**Level**

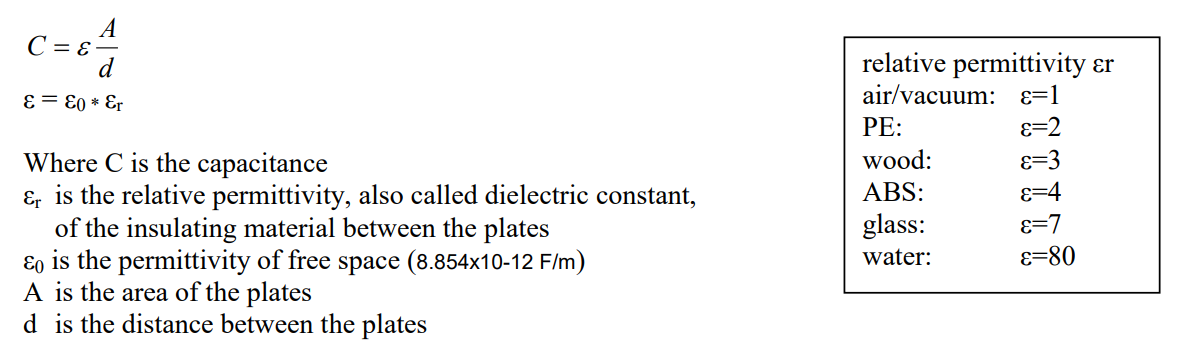
0 –> System Halt  
1 –> Single user mode  
3 –> Full multiuser mode with network  
5 –> Full multiuser mode with network and X display manager  
6 –> Reboot  
The above design of init is called SysV- pronounced as System five. Several other implementations of init have been written now. Some of the popular implementations are systemd and upstart. Upstart is being used by ubuntu since 2006. More details of the upstart can be found here.

The next step of init is to start up various daemons that support networking and other services. X server daemon is one of the most important daemon. It manages display, keyboard, and mouse. When X server daemon is started you see a Graphical Interface and a login screen is displayed.

**Capacitative Touch**

Basic Principle The simplest form of a capacitor consists of two conductors, e.g. two metal plates, separated by an insulator.

The following formula shows the parameters which influence capacitance:



The larger the area of the plates, the larger is the capacitance. The smaller the distance between the two plates, the higher is the capacitance. The insulating material determines the dielectric constant. Table 1 list some common materials and their permittivity. The electrode of a touch sensor represents one plate of such a capacitor. The corresponding 2nd plate is represented by the environment of the sensor electrode (to form a parasitic capacitor C0) and another conductive object, like a human finger for example (to form touch capacitor CT). This capacitor, i.e. the sensor electrode, is connected to a measurement circuit. The capacitance of the sensor pad is measured periodically. If a conductive object approaches or touches the electrode, relative permittivity the measured capacitance will increase. This change is detected by the measurement circuit and converted into a trigger signal.

Considering the formula above, one can see that a bigger pad and a thinner overlaying cover material, leads to a bigger touch capacitance CT and as a result, a bigger capacitance difference between touched and untouched sensor pad. In other words, the size of the electrode and the covering material influence the sensitivity of the sensor.

**Self Capacitance**

The electronics measure the current on each electrode to ground and therefore is called “self-capacitance”.

There are two options for how the system can detect touch — multi-pad construction or rows and columns.

In a row-and-column construction, each row and column is an electrode and therefore is individually addressed by the controller. Even though the intersection of a row and column represents a unique coordinate pair, the electronics are not able to measure each individual intersection as they can only measure each electrode.

Ghost points are the result of imaginary or false row and column intersections in locations other than the touch location.

**Number of Touch Points (one, two, multiple)**

Projected capacitive technology in varying configurations can support single, dual and multi-touch. Self-capacitance solutions are limited to single and dual touch where gesturing can be achieved, but ghosting effects can occur. Mutual capacitance is capable of full multi-touch interactivity since each node intersection is individually addressable.

Consider the best new touchscreen phones, in which the user interface is a large capacitive sensing screen that differentiates a flick from a tap and tracks the motion of your finger but doesn't track your ear.

Self capacitance works OK for single-touch systems, but with multi-touch systems there is no way to resolve the positional ambiguity that results from more than one simultaneous touch on different parts of the screen.

For example, if a user touches on the capacitive grid at locations X1, Y1 and X2, Y2, the energized lines simply tell the chip that X1, X2, Y1, Y2 lines have all been touched. It doesn't know the combination thereof. It could be that the chip reports X1, Y2 and X2, Y1 were the touch locations. This problem is known as ghosting.

In contrast, mutual capacitance measurement uses an orthogonal matrix of transmit and receive electrodes arranged as an array of multiple smaller touch nodes created by the geometry of the electrode structure.

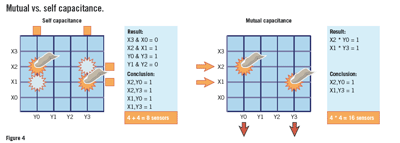
In a mutual capacitance based system, each touch is uniquely detected as an xy coordinate pair, whereas in a self capacitance system, the detection of X and Y coordinates of a touch is independent.

If two touches are present in a mutual capacitance system, this would be detected as (X1,Y1) and (X2,Y2), whereas in a self-capacitance system it would be detected as (X1,X2,Y1,Y2), leaving two potential combinations of coordinates. The self-capacitance ghosting problem is exponential and becomes impossible to solve as you transition to three or more touches.

A mutual capacitive array is interpreted as a complete touch surface that maintains the ability to resolve multiple touch points within each individual "small" screen. Because the capacitive coupling at each point in the matrix can be measured independently, it means that there is no ambiguity in the reported coordinates for multiple touches.

It is then technically possible to have unlimited touch recognition.

**Figure 4** compares mutual vs self capacitance.



The simplest and most common gestures are touch (and double touch), drag, flick,.

A single touch, analogous to a click event with a mouse, is defined by the amount of time a touch is active and the amount of movement during the touch. Typical values might be that the touch-down and touch-up events must be less than a half second apart, and the finger cannot move by more than five pixels.

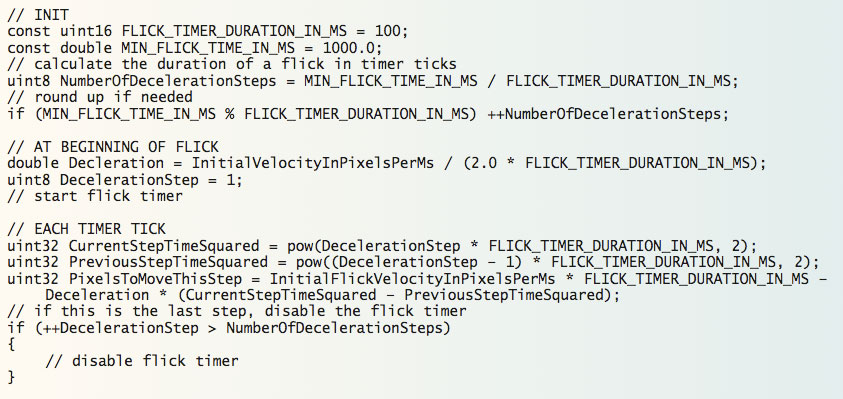
A **double touch** is a simple extension of the single touch where the second touch must occur within a certain amount of time after the first touch, and the second touch must also follow the same timing and positional requirements as the first touch. Keep in mind that if you are implementing both a single touch and a double touch, the single touch will need an additional timeout to ensure that the user isn't executing a double touch.

A **flick** is similar to a drag but with a different purpose. A drag event begins when the finger touches the panel and ends when the finger is removed. A flick can continue to generate events after the finger is removed. This can be used to implement the kinds of fast scrolling features common on many cell phones where a list continues to scroll even after the finger is lifted. A flick can be implemented in several ways, with the responsibilities divided between the gesture-recognition layer and the application layer. Before we discuss the different ways to implement flick gestures, let's first focus on how to define a flick.

A *flick* is generally a fast swipe of the finger across the surface of the touch panel in a single direction. The actual point locations during the flick do not typically matter to the application. The relevant parameters are velocity and direction. To identify a flick, the gesture-recognition layer first needs to determine the velocity of the finger movement. This can be as simple as determining the amount of time between the finger-down report and the finger-up report divided by the distance traveled. However, this can slow the response time since the velocity is not determined until after the gesture has finished.

This flick algorithm is based on the standard formula for calculating distance traveled based on velocity and acceleration: *d* = *vt* + ½*at*2. The velocity is the initial velocity of the flick gesture. The acceleration and time are tuned to meet the application’s requirements. In this case we use a constant time and determine the rate of deceleration required to complete the decay in that time. A timer is used to generate each of the flick deceleration events. The timer should fire every 10 to 200 milliseconds depending on processor utilization and the required smoothness of the flick. On each timer tick we need to move a certain number of pixels, which decreases with each step.The number of pixels to move on each step is calculated by taking the delta between the total distance of the previous step and the total distance of the next step:

Delta = (*vtn*+1 +½*atn*+12) – (*vtn* + ½*atn*2)

After simplifying and putting in terms of code variables, we get:   
  
PixelsToMoveThisStep = InitialFlickVelocityInPixelsPerMs \* StepTimeInMs   
– Deceleration\*(CurrentStepTimeInMs2 – PreviousStepTimeInMs2)  
  
The flick processing is divided among three areas of the code:  
  
[](http://m.eet.com/media/1121088/1211esdocularl01.jpg)

For greater responsiveness, track the velocity as the gesture is occurring. If the velocity is above a certain threshold, a flick has occurred. The direction of the flick can also be determined from the touch-down and touch-up positions using the arc tangent function

(refer to the sidebar on rotation angles for details). However, this simplistic approach can result in false flicks. The user may draw a very quick circle, which meets the velocity requirements but most certainly should not be interpreted as a flick. To prevent this type of false report, the gesture-recognition engine should determine if all the reported points are in a relatively straight line.

If the algorithm is reporting the velocity in real time, it must include some kind of prequalification of the gesture direction before it starts reporting flick events to the application. The gesture-recognition engine also needs to decide how to report the direction of the flick. This is driven by the user-interface requirements and the amount of decoupling desired between the gesture-recognition layer and the application. In the most generic version, a flick might report the direction as an angle. Or it could report it in a more application-friendly way such as up, right, left, or down.

|  |
| --- |
| **Rotation angle**  The rotation angle is based on the position of the two fingers defining the rotation. The coordinates of the two touches can be used in conjunction with the arctangent function to return an angle in radians:  AngleInRadians = atan((SecondTouch.Y - FirstTouch.Y) / (SecondTouch.X - FirstTouch.X));   The angle can be easily converted to degrees if needed:  AngleInDegrees = AngleInRadians \* 180.0 / 3.14159;   There are several issues with the atan function as used above. First, if the two X coordinates are the same then a divide-by-zero error occurs. Second, the atan function cannot determine which quadrant the angle is in and so returns a value between +π/2 (+90º) and -π/2 (-90º). It's better to use the atan2 function which handles both of these problems and returns a value between π(+180º) and -π(-180º):  AngleInRadians = atan2(SecondTouch.Y - FirstTouch.Y, SecondTouch.X - FirstTouch.X);   The use of floating-point libraries can be avoided by creating a table of angles based on the ratio of Y and X. This has the added advantage of avoiding the radians-to-degrees conversion. The Y value will need to be multiplied by some factor so that the resulting Y-to-X ratio can be expressed as an integer. The size of the table and the Y multiplier used depends on the required resolution of the angle. A four-entry table is enough to return up, right, down, or left. A table with 360 entries can return the exact angle in whole degrees. Just be sure that the Y multiplier used to calculate the ratio is large enough to give the desired resolution.  The rotation angle is typically implemented as an offset. Wherever the user first places their fingers becomes the reference angle. As the fingers rotate around the panel, the actual rotation angle is the difference between the reference angle and the current angle. |

Another typical feature of a flick is *inertia*. Inertia provides a natural end to a flick operation by, for example, slowing down the scrolling of a list until it comes to a stop. Inertia can be implemented at the application level or the gesture-recognition level. At the gesture level, inertia can be implemented as a series of events with decreasing velocity. This could be implemented with specific inertia events, but it's typically easier to reuse the flick event for the same purpose.

The initial flick event includes the direction and velocity of the actual flick gesture. A timer in the gesture-recognition engine then continues to generate flick events using the original direction and a decreasing velocity. The decreasing velocity is generated by an exponential decay function that is tuned to create whatever deceleration profile works best for the application. You can even dynamically control the deceleration profile based on the application state

**Audio Jack Detection**

Advanced accessory jack detection circuitry which determines insertion, removal, as well as type detection of accessories with a 3.5-mm headset jack.

When the insertion of a jack is detected, the accessory type detection algorithm runs until two consecutive type detections produce the same result. In general, the type detection algorithm is not run again after this point of time. Hence, on-the-fly change of accessory type is not detected.

*Shutdown Mode:*

Shutdown mode enables lowest power consumption from device. During this mode, accessory insertion, removal and type detection are not supported, but as soon as system comes out of shutdown, detection will work fine.

*Sleep Mode:*

System in Sleep Mode This mode is enabled by programming SLEEP, register 0x1D (bit-6) to 1. During this mode, accessory insertion, removal, type and single-button press/release detection are supported.

*Accessory Not Inserted in Sleep Mode:*

If accessory was previously inserted and is then removed, interrupt gets generated. Upon interrupt following sequence can be used to determine accessory removal and take appropriate action:

• Read control registers 0x00 to 0x02

• If JKIN, register 0x00, bit-7 = 0, then infer that accessory has been removed. Program SLEEP, register 0x1D, bit-6 = 1.

*Accessory Inserted in Sleep Mode:*

If the accessory was previously not inserted and is then inserted, interrupt gets generated.

Upon interrupt following sequence can be used to determine accessory insertion and take appropriate action:

• Read control registers 0x00 to 0x02

• If JKIN, register 0x00, bit 7 = 1, then infer that accessory has been inserted.

• Read control register 0x19 to determine type of accessory When the system wakes up and programs the device out of SLEEP mode, appropriate blocks will automatically turn on based on the type of accessory.

Button Detection during Sleep Mode:

During Sleep mode, single-button press/release detection is supported. For a button press, the system can use this event to wake up the system, and then program the device (details in Button Detection) to detect the second button press correctly. Upon interrupt, the following sequence needs to be followed for button detection:

• Read control registers 0x00 to 0x02

• If MCSW, register 0x00, bit 1 = 1, then infer that button has been pressed/released.

As long as button is pressed, MCSW continues to remain set.

*System in Wake-Up Mode:*

This mode is enabled by programming SLEEP, register 0x1D (bit-6) to 0.

*Accessory Not Inserted*:

If accessory was previously inserted and is then removed, interrupt gets generated. Upon interrupt following sequence can be used to determine accessory removal and take appropriate action:

• Read control registers 0x00 to 0x02

• If JKIN, register 0x00, bit 7 = 0, then infer that accessory has been removed. Program SLEEP, register 0x1D, bit-6 = 1.

*Accessory Inserted*:

If accessory was previously not inserted and is then inserted, an interrupt is generated. Upon an interrupt, the following sequence can be used to determine that an accessory insertion has occurred and take appropriate action:

• Read control registers 0x00 to 0x02 • If JKIN, register 0x00, bit-7 = 1, then infer that accessory has been inserted.

• Read control register 0x19 (bit 6-0, STATE) to determine type of accessory.

**What OS was used in Audience for the ASIC?**

FreeRTOS. (Preemptive priority based scheduler)

**Explain the basic program flow of the audio?**

There were several modules

1. Control Manager
2. Routing
3. Voice Q - > PDM mic
4. BootLoader (ROM)
5. Audio Jack Detection
6. Basic communication drivers

**PDM:**

PDM stands for pulse density modulation. However, it is really better summarized as “oversampled 1-bit audio

Most current digital audio systems use multi-bit PCM (pulse code modulation) to represent the signal. PCM has the advantage of being easy to manipulate. This allows signal processing operations to be performed on the audio stream, such as mixing, filtering, and equalization. PDM, which uses only one bit to convey audio, is simpler in concept and execution than PCM. It has become popular as a way to deliver audio from microphones to the signal processor in mobile telephones. PDM is ideally suited for this task because it brings the benefits of digital, such as low noise and freedom from interfering signals, at low cost.

PCM (Pulse Code Modulation): a system for representing a sampled signal as a series of multi-bit words. This is the technology used in audio CDs.

PDM (Pulse Density Modulation): a system for representing a sampled signal as a stream of single bits.

Oversampling:

The noise incurred by reducing the word length is substantial. In audio, however, mid- and high-frequencies are very important, and very audible. It is simply not possible to achieve acceptable results if the word length is reduced to one bit, even with noise shaping. The resulting high-pass noise is clearly audible. The answer is to use a higher sampling rate. This increases the bandwidth of the system, creating new spectrum above the audible range. Noise shaping can then be used to push noise into that spectrum.

A higher sampling rate can be realized in two ways:

• By using a higher sampling rate in the first place. This is the method used in PDM microphones, where the typical sampling rate is 3 MHz.

• By interpolating an existing signal that has been sampled at a low rate. This is the method used in many DACs, where a typical incoming sample rate is 48 kHz. It is also used in systems which represent audio internally as PCM, but transmit audio to external devices in PDM form

PDM Microphones:

A PDM microphone, also called a digital microphone, consists of the following parts:

• A microphone element. Typically this is an electret capsule.

• An analog preamplifier.

• A PDM modulator.

• Interface logic.

The analog signal from the microphone element is first amplified, and then sampled at a high rate and quantized in the PDM modulator. The modulator combines the operations of quantization and noise shaping; the output is a single bit at the high sampling rate. The noise shaping ensures that the noise in the audio band is relatively low, while the noise above the audio band is relatively high. The interface logic is responsible for accepting a master clock and transmitting the sampled bitstream. The device to which the microphone connects provides the master clock to the PDM microphone. The clock rate defines the sampling rate of the system, as well as the rate at which bits are transmitted on the data line. Although there is no defined standard, typically the oversampling ratio is 64. So to achieve a bandwidth of 24 kHz (comparable to a PCM system sampled at 48 kHz), a master clock frequency of 3.072 MHz is needed. The one-bit data is asserted on the data line on either the rising or falling edge of the master clock. Most PDM microphones support stereo operation, in which one microphone asserts the data line on the rising edge of the master clock, while a second microphone asserts on the falling edge. On the non-asserted edge, the data output has a high impedance. The data lines from the two microphones can then simply be connected together. The PDM receiver is responsible for separating the two bitstreams.

DACs and PCM-to-PDM converters

In some commercial DACs, and in systems which convert PCM to PDM, the procedure is slightly different from PDM microphones. The signal has already been sampled at a low rate, and is in PCM form. To achieve the high sampling rate needed for noise shaping to be effective, the signal must first be interpolated. Its wordlength is then reduced to one bit in a noise shaper. Interpolation is a digital filtering operation in which extra samples are generated in between the existing samples to increase the effective sampling rate. For PDM applications, the oversampling ratio is typically 64; that is, 63 new samples are generated for each input sample.

PDM Modulators

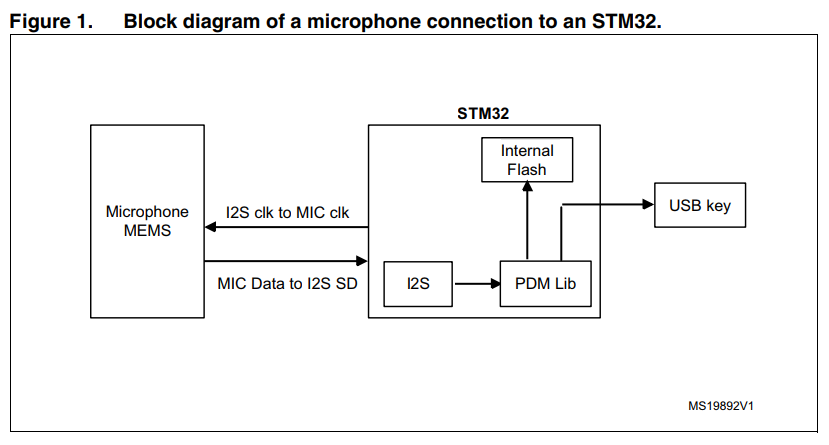
The PDM modulator (in PDM microphones) or the noise shaper (in PCM-to-PDM converters) is responsible for producing a one-bit signal which has very low noise in the passband. The complexity of the modulator is expressed by its order. The order of a modulator is equal to the number of integrators (accumulating nodes) it contains; in general, the higher the order, the more aggressively the noise is shaped from the passband to the stopband, and the better the noise performance. However, higher order modulators are more complex to design and manufacture; they are more likely to become unstable under certain operating conditions; and their maximum input level before overload is lower. While there is no industry standard, typical modulators in PDM microphones are fourth order. This offers a good compromise between noise performance and complexity.

Conclusion

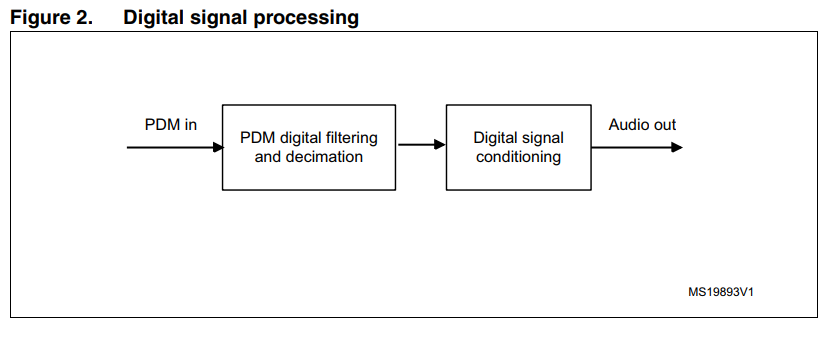
PDM is a cost-effective way of conveying audio digitally, in mono or stereo, over a clock/data pair. Despite the inherent limitations of a one-bit representation, it is possible to achieve extremely high audio performance with careful design. The APx PDM Interface option generates and analyzes PDM signals natively, greatly simplifying the design and troubleshooting of all aspects of the PDM signal chain.

Pulse density modulation, or PDM, is a form of modulation used to represent an analog signal in the digital domain. In a PDM signal, specific amplitude values are not encoded into pulses as they would be in PCM. Instead it is the relative density of the pulses that corresponds to the analog signal's amplitude. To get the framed data from the PDM bit stream, decimation filters are usually used. The first stage of decimation is used to reduce the sampling frequency, followed by a high pass filter to remove the signal DC offset.

The MP45DT02 MEMS microphone outputs a PDM signal, which is a high frequency (1 to 3.25 MHz) stream of 1-bit digital samples. This output is acquired in blocks of 8 samples by using a synchronous serial port (SPI or I2S) of the STM32 microcontroller. The microphone's PDM output is synchronous with its input clock; therefore an STM32 SPI/ I2S peripheral generates a clock signal for the microphone.



The data coming from the microphone is sent to the decimation process, which consists of two parts: a decimation filter converting 1-bit PDM data to PCM data, followed by two individually configurable IIR filters (low pass and high pass). The reconstructed audio is in 16-bit pulse-code modulation (PCM) format. After the conversion, it produces raw data that can be handled depending on the application implementation (stored as wave/compressed data in a mass storage media, transferred to an external audio codec DAC through I2S peripheral...).



PDM digital filtering and decimation The PDM signal from the microphone is filtered and decimated in order to obtain a sound signal at the required frequency and resolution. The frequency of the PDM data output from the microphone (which is the clock input to the microphone) must be a multiple of the final audio output needed from the system. For example, to perform a decimation of 80, for the output rate of 30 kHz, we need to provide a clock frequency 2.4MHz to the microphone. The output of the filter pipeline is a 16-bit value, we consider [-32768, 32767] as the output range for a unitary gain (0 dB). 4.2 Digital signal conditioning The digital audio signal resulting from the previous filter pipeline is further processed for proper signal conditioning. The first stage is a high pass filter designed mainly to remove the signal DC offset. It has been implemented via an IIR filter with a cut-off frequency below the audible frequency range in order to preserve signal quality. The second stage is a low pass filter implemented using an IIR filter. Both filters can be enabled/disabled and configured (cut-off frequencies) by using the filter initialization function.Gain can be controlled by an external integer variable (MicGain) as shown in the following equation: G = MicGain/64.