Embedded System programming

Sarath T.V.

Memory layout

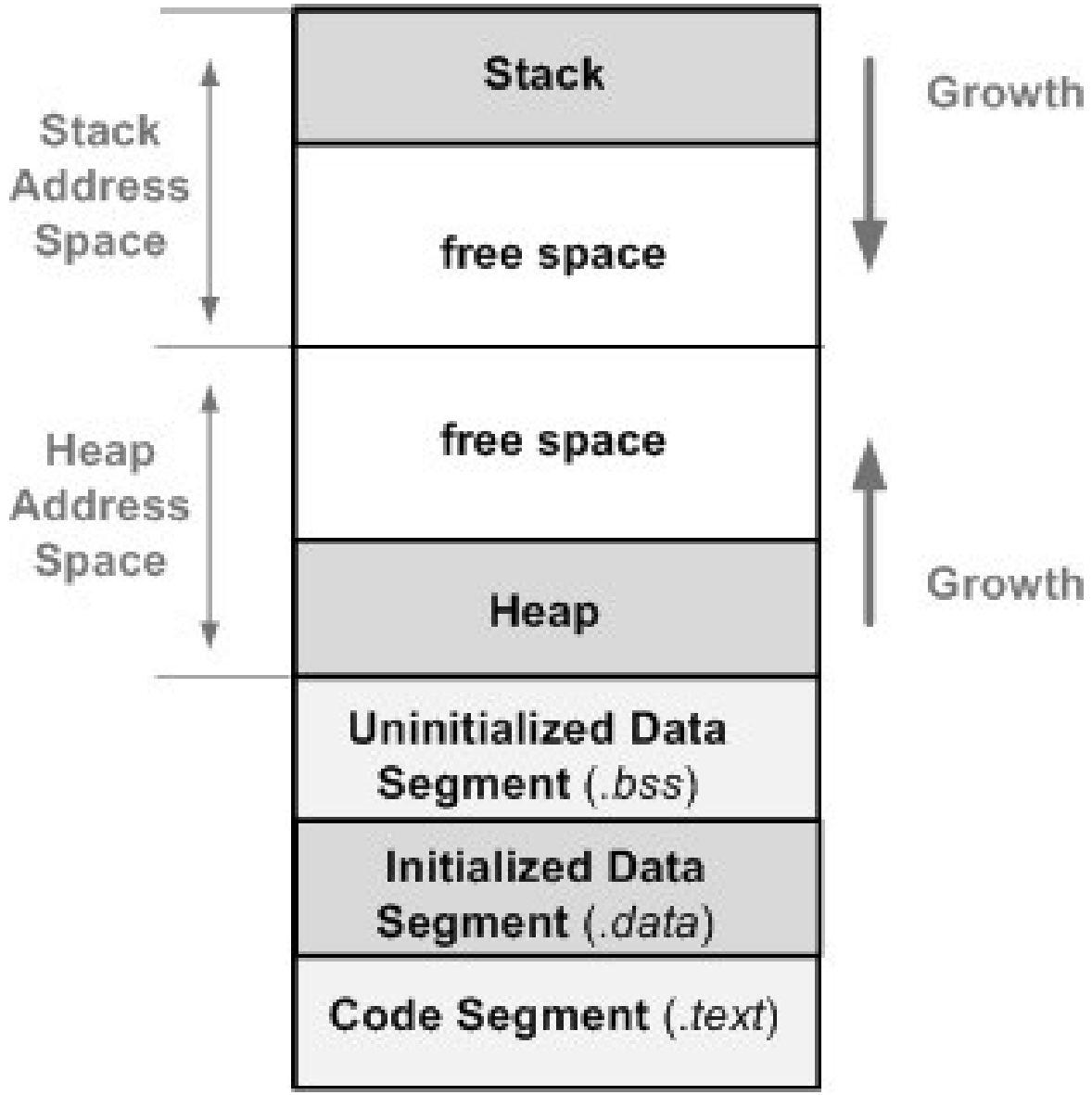
Where in memory are my variables stored in C?

For a running program both the machine instructions (program code) and data are stored in the same memory space.

The memory is logically divided into **text** and **data segments**.

Modern systems use a **single text segment** to store program instructions, but **more than one segment** for **data**, depending upon the storage class of the data being stored there.

1. Text or Code Segment
2. Initialized Data Segments
3. Uninitialized Data Segments
4. Stack Segment
5. Heap Segment


Stack
Stack Address Space
free space
Growth
free space
Heap Address Space
Heap
Growth
Uninitialized Data Segment (.bss)
Initialized Data Segment (.data)
Code Segment (.text)

Code Segment (.text)

This segment stores the executable program code (the machine instructions). Variables defined with the ***const*** type qualifier can also be placed in the code segment.

It is a **read-only segment** stored in **a non-volatile memory**. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.(**to avoid risk** of **getting overridden** by programming bugs like buffer-overflow, etc.)

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.

Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on.

The code segment does not contain program variables like local variable (also called as automatic variables in C), global variables, etc.

Based on the C implementation, the code segment can also contain **read-only string literals**. For example, **when you do printf("Hello, world")** then string "Hello, world" gets created in the code/text segment. You can verify this using **size** command.

Data Segments

- *Data segment* stores program data.
- This data could be in form of **initialized** or **uninitialized variables**, and it could be **local** or **global**.
- Data segment is further divided into four sub-data segments (initialized data segment, uninitialized or .bss data segment, stack, and heap) to store variables depending upon if they are local or global, and initialized or uninitialized.
  - Initialized Data Segments
  - Uninitialized Data Segments
  - Stack Segment
  - Heap Segment
- The data segment never lies between the heap and stack area

Uninitialized data segment

- This segment is also known as bss.
- This is the portion of memory which contains:
  - Uninitialized global variables
  - Uninitialized constant global variables.
  - Uninitialized local static variables.
- Any **global** or **static local variable** which is **not initialized** will be stored in the uninitialized data segment
- If you declare a **global variable** and **initialize** it as **0** or **NULL** then still it would go to uninitialized data segment or bss.

This segment stores all global and local variables that are initialized to zero or do not have explicit initialization in the source code.

The memory locations of **variables** stored in this segment are **initialized** to **zero before** the **execution** of the **program starts**.

Uninitialized data segment, often called the "**bss**" segment, named after an ancient assembler operator that stood for "**block started by symbol**."

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.

Initialized Data or 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.
- For instance the global string defined **by char s[] = "hello world"** in C and a C statement like **int debug=1** outside the main (i.e. global) would be stored in initialized read-write area.
- And a global C statement like **const char* string = "hello world"** makes the string literal "hello world" to be stored in initialized read-only area and the character pointer variable string in initialized read-write area.
- Ex: static int i = 10 will be stored in data segment and global int i = 10 will also be stored in data segment

- This segment stores all global variables (with storage class specifier *static* or *extern)* and local (defined inside a function) *static* variables that have defined initial values different than zero..
- Initialized data segment stores:
- Initialized global variables (including pointer variables)
- Initialized constant global variables.
- Initialized local static variables.
- For example: global variable int globalVar = 1; or static local variable static int localStatic = 1; will be stored in initialized data segment.
- The **size** of this **segment** is determined by the size of the values in the program's source code, and **does not** change at **run time**.

```
/*****Identify Memory location for these variable *****/
#include<stdio.h>
int variable_1=192;
 int variable_2;
int variable_3 = 0;

int main() {
static int variable_2=67;
/* ... Function code ... */

}
```

Ordered, on top of each other

No particular order

Stack

- The stack area traditionally **adjoined** the **heap** area and grew the opposite direction;
- when the stack pointer met the heap pointer, free memory is exhausted.
- Typically grow opposite directions.
- 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**.
- Each time a function is called, the address of where to **return** to and **certain** information about the **caller's environment**.
- The **newly** called **function** then **allocates room** on the stack for its **automatic** and **temporary variables**.

- Stack segment is used to store variables which are created inside functions (*function could be main function or user-defined function*), variable like

  Local variables of the function ***(including pointer variables)***

  Arguments passed to function

  Return address

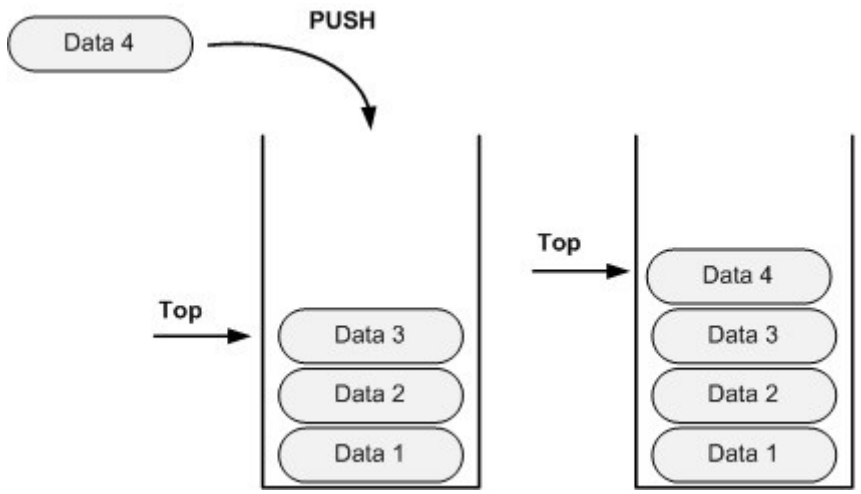
- Variables stored in the stack will be removed as soon as the function execution finishes.

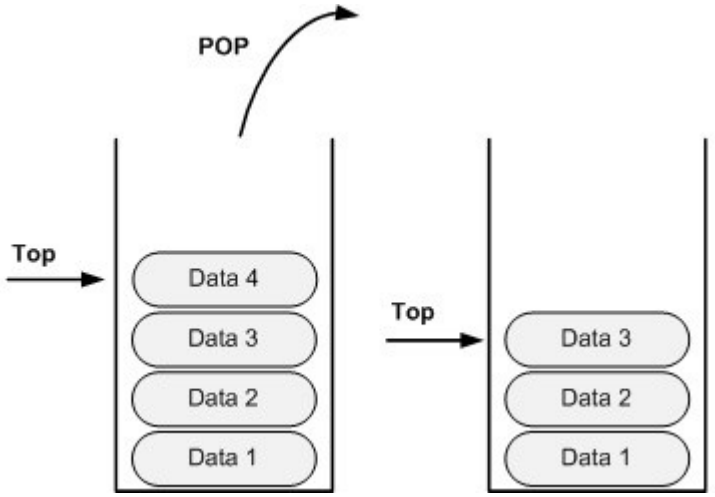
The Concept of Stack and Its Usage in Microprocessors

In general we can describe the ***stack*** as a temporary storage for data. The access to data in the stack is organized as **Last In First Out (LIFO)**, which means that the last data stored in the stack is the first than can be retrieved and the first data stored in the stack is the last to be retrieved. The operation of adding data to the stack is called **PUSH** and the operation of retrieving data is called **POP(PULL)**.

The stack can be viewed as a **pile** of **plates**. The **bottom** plate is the **first data** pushed onto the stack and the **top plate** is the **last data** pushed.

When the **top plate** is removed (**pulled**), the **one below pops** up to become the **new top**. The top plate is pulled first and the **bottom plate** is **pulled last**. It is important to note that **only** the **top element** in the **stack can be accessed** (in the classic stack implementation).


Data 4
PUSH
Top
Data 3
Data 2
Data 1
Top
Data 4
Data 3
Data 2
Data 1


POP
Top
Data 4
Data 3
Data 2
Data 1
Top
Data 3
Data 2
Data 1

Stack Usage in Programming Languages

The vast majority of programming languages that implement functions use a ***stack***. Each function is given its own section of memory to operate in. This section is a part of the ***stack*** and is referred to as a ***stack frame***. It most commonly includes the following components:

The return address (when the function is complete it returns back to the function call)

Arguments passed to the function

Local variables of the function

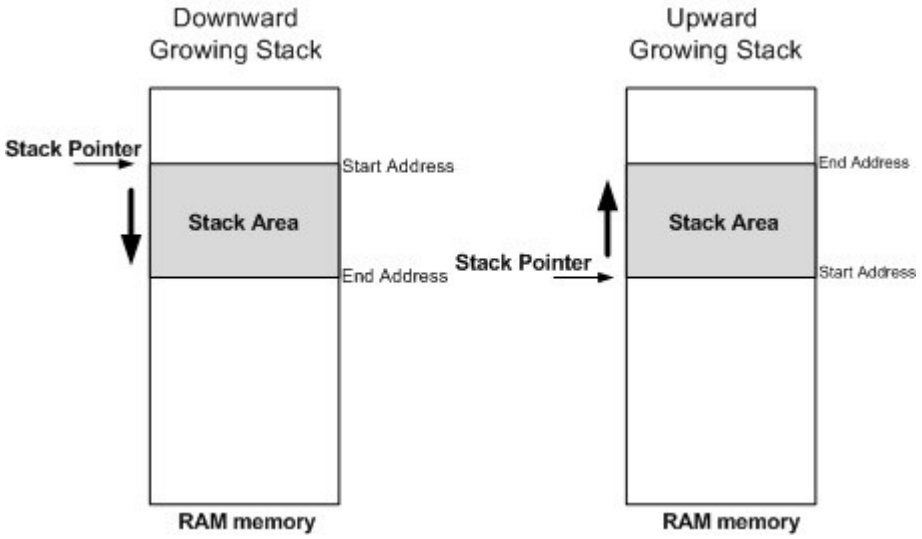
Saved copies of any registers modified by the function that have to be restored

When a program enters a function, space is allocated on top of the stack. When the program leaves the function, the space is freed. The life span of variables on the stack is limited to the duration of the function. Data that must survive across function calls is stored on the ***heap.***

Hardware Implementation of Stack

The common hardware implementation of **stack** consists of reserved contiguous region of memory with a stack pointer into that memory. The stack has a fixed location in memory at which it begins.

The ***stack pointer*** is a hardware register that points to the current extents of the stack. There are stacks that grow downwards and ones that grow upwards. The stack can technically be located anywhere in memory, depending on the system.



Types of Stack

Software to implement a stack - different implementation choices result different types of stacks.

There are two types of stack depending on how the stack grows.

Ascending Stack -grows upwards. Starts from a **low memory address** and, as items are pushed onto it, **progresses to higher memory addresses**.

In a push the **stack pointer is incremented**, i.e. the stack grows towards higher address.

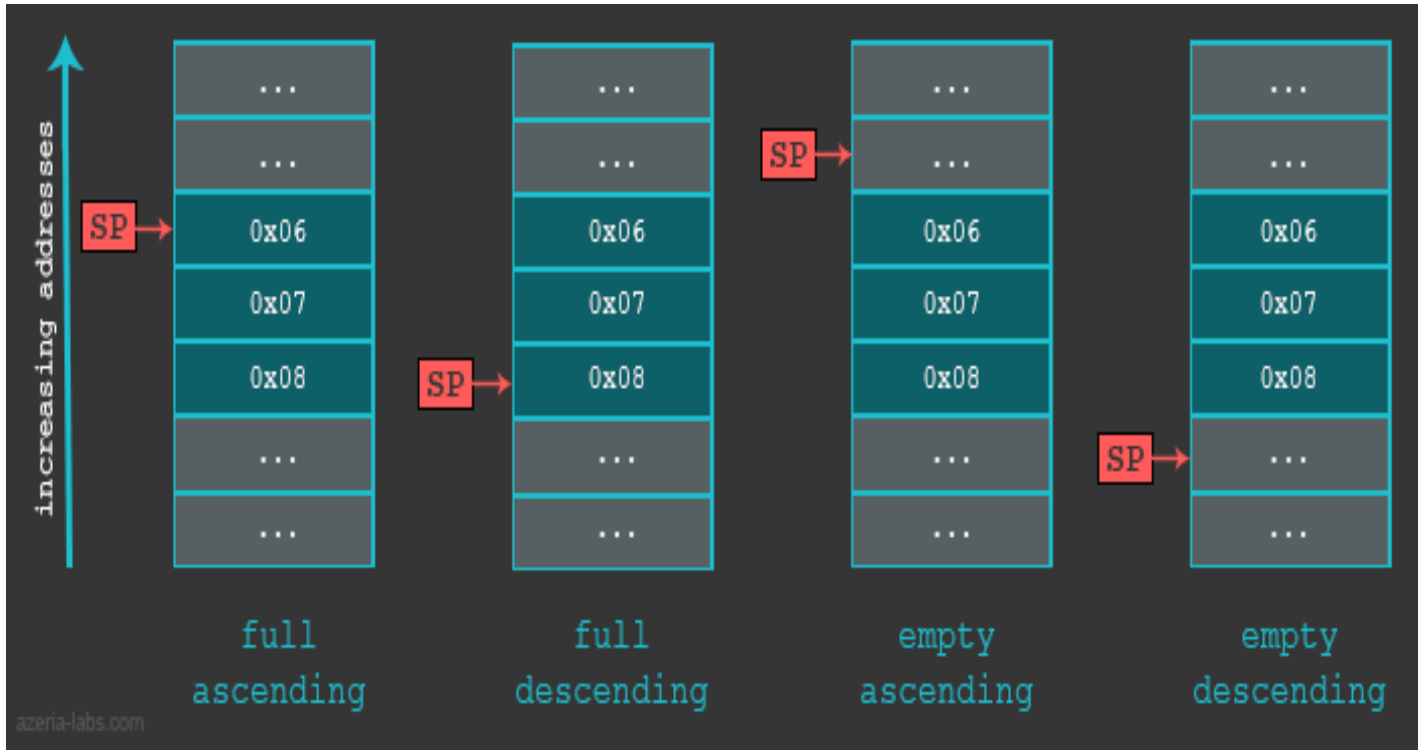
Descending Stack -grows downwards. It starts from a **high memory address**, and as items are pushed onto it, **progresses** to **lower memory addresses**. In a push the **stack pointer is decremented**, i.e. the stack grows towards lower address.

There are two types of stack depending on what the stack pointer points to.

Empty stack -Stack pointer **points** to the **location** in **which** the **next item** will be **stored**. A **push** will **store** the **value**, and **increment** the **stack pointer**. The stack pointers points to the next free (empty) location on the stack,

Full stack - Stack **pointer points** to the **location** in which the **last item** was **stored**. A **push** will **increment** the **stack pointer** and **store** the value. the stack pointer points to the topmost item in the stack, i.e. the location of the last item to be pushed onto the stack.

Four different stacks are possible - *full-ascending, full-descending, empty-ascending, empty-descending*.


increasing addresses
SP
0x06
0x07
0x08
full
ascending
full
descending
empty
ascending
empty
descending
azeria-labs.com

Applications

The simplest application of a stack is **to reverse a word**. You push a given word to stack - letter by letter - and then pop letters from the stack.

Another application is an "**undo" mechanism** in text editors; this operation is accomplished by keeping all text changes in a stack.

Backtracking. This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or **maze** - how do you find a way from an entrance to an exit?

Once you reach a **dead end**, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.

Implementation of Stack Data Structure

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size.

API s for stack operations

Push: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.

Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

Peek or Top: Returns top element of stack.

isEmpty: Returns true if stack is empty, else false.

Pop and Top/peek !!!!

Top is a state **variable** for your stack, which is stored in a regular array. The variable top references the top of the stack by storing an array index.

The first operation, **pop**, uses the **decrement operator** to **change** the **variable top**, and hence the state of the stack:

--top is equivalent to top = top - 1.

The value is still in the array, but since the variable top now references a different index, this value is effectively removed: the top of the stack is now a different element. Now if you call a push, this popped value will be overwritten.

The second operation peek **doesn't change** the **value** of the **variable top**, it only uses it to **return** the **value** at the **top** of the **stack**. The variable **top still references** the **same value** as the stack's top, and so the stack is unchanged.

Heap

- Heap is the segment where **dynamic memory allocation** usually takes place.
- The heap area begins at the end of the BSS segment.
- The Heap area is managed by **malloc**, **realloc**, and **free**.
- The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

The Concept of Heap and Its Usage in Embedded Systems

- The heap is a segment of the system memory (RAM) that provides dynamic memory allocation.
- Dynamic allocation usually requires **two basic steps**:
- ***Creating the dynamic space***
- ***Creating a pointer holding the address for the newly created space*** *(new variable names can't be created while the program is running, so pointers are needed)*
- There are several functions (part of *stdlib* library) for dynamic memory allocation and management:

malloc() and calloc( ) – reserve space

realloc() – move a reserved block of memory to another allocation of different dimensions

free() – release allocated space

Heap Usage

Here are some of the most common cases when the heap is used:

- When we need a **data** that must **live** after the **function returns**. Once data is stored on the heap during **function execution** it is **not affected** when the **function ends**
- When we **need a lot of memory**
- When we don't know exactly **how much data** we will **need to store** during the execution of the program

Heap Usage Pitfalls

Using **dynamic memory allocation** brings a certain **degree** of **complexity** and if the end application does **not explicitly requires** it, the usage of heap should be **avoided** (especially in small embedded systems).

There are a lot of specifics that a person should be familiar with before deciding on using the heap.

Below are listed some **common pitfalls**, when the heap integrity is compromised:

Overwritten heap data

Allocation failures, when a too big buffer is requested to be allocated.

Heap is responsible for memory leaks

The C malloc() and free() function can take a long time to execute, which is in conflict with the real-time constraints of an embedded system

Heap memory management requires **additional memory** as the **algorithms** need to store some form of header information for each allocated block

Dynamic Memory Allocation

A program can at any time request that the operating system allocate additional memory.

The memory is allocated from a data structure known as a heap, which facilitates keeping track of which portions of memory are in use by which application.

Memory allocation occurs via an **operating system call** (such as malloc in C).

When the program **no longer needs** access to memory that has been so allocated**, it deallocates** the memory (by calling free in C).

A garbage collector is a **task** that **runs** either periodically or when memory gets tight that analyzes the data structures that a program has **allocated and automatically frees any portions of memory that are no longer referenced within the program**.

calloc() & malloc()

Allocate memory dynamically.

Memory is allocated during runtime(execution of the program) from heap segment.

- **Initialization**: malloc() **allocates memory block** of given **size** (in **bytes**) and returns a **pointer** to the **beginning** of the **block**.

```
void * malloc( size_t size );
```

  - malloc() **doesn't initialize** the allocated memory.
  - access the content of memory block -**garbage values**.

- *calloc*() **allocates** the **memory** and also **initializes** the allocates memory **block** to **zero**.

  access the content of these blocks then we'll get 0.

- **Number of arguments:** Unlike malloc(), calloc() takes two arguments:
  1) Number of blocks to be allocated.
  2) Size of each block.

```
void * calloc( size_t num, size_t size );
```

- **Return Value:** After successful allocation in malloc() and calloc(), a **pointer** to the block of memory is returned otherwise **NULL** value is returned which indicates the failure of allocation

```
#include <stdio.h>
#include <stdlib.h>
 int main() {
   int *arr;
// malloc() allocate the memory for 5  integers containing garbage values
 arr = (int *)malloc(5 * sizeof(int)); // 5*4bytes


// Deallocates memory previously allocated by malloc() function
   free( arr );
// calloc() allocate the memory for 5 integers and
   // set 0 to all of them
   arr = (int *)calloc(5, sizeof(int));


   // Deallocates memory previously allocated by calloc() function
   free(arr);


   return(0);
}
```

Realloc

If suppose we allocated more or less memory than required, then we can change the size of the previously allocated memory space using realloc.

void *realloc(pointer, new-size);

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
char *p1;
int m1, m2;
m1 = 10; m2 = 30;
p1 = (char*)malloc(m1);
strcpy(p1, “Embedded");
p1 = (char*)realloc(p1, m2);
strcat(p1, “System Programming"); printf("%s\n", p1);
 return 0;
}
```

When to use the Heap?

If you need to allocate a **large block of memory** (e.g. a large array, or a big struct), and you need to **keep** that **variable** around a **long time** (like a global), then you should allocate it on the **heap**. If you are dealing with relatively **small variables** that only need to **persist** as **long** as the **function** using them is **alive**, then you should use the **stack**, it's **easier** and **faster**.

If you need variables like arrays and structs that can change size dynamically (e.g. arrays that can grow or shrink as needed) then you will likely need to allocate them on the heap, and use dynamic memory allocation functions like malloc(), calloc(), realloc() and free() to manage that memory "by hand"

Stack vs Heap Pros and Cons

Stack

•very fast access
•don't have to explicitly de-allocate variables
•space is managed efficiently by CPU, memory will not become fragmented
•local variables only
•limit on stack size
•variables cannot be resized

Heap

•variables can be accessed globally
•no limit on memory size
•(relatively) slower access
•no guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed
•you must manage memory (you're in charge of allocating and freeing variables)
•variables can be resized using realloc()

Key Differences Between Stack and Heap Allocations

- In a stack, the allocation and deallocation is automatically done by whereas, in heap, it needs to be done by the programmer manually.
- Handling of Heap frame is costlier than handling of stack frame.
- Memory shortage problem is more likely to happen in stack whereas the main issue in heap memory is fragmentation.
- Stack frame access is easier than the heap frame as the stack have small region of memory and is cache friendly, but in case of heap frames which are dispersed throughout the memory so it cause more cache misses.
- Stack is not flexible, the memory size allotted cannot be changed whereas a heap is flexible, and the allotted memory can be altered.
- Accessing time of heap takes is more than a stack.

Memory Leak and Fragmentation

Memory leak: With or without garbage collection, it is possible for a program to inadvertently **accumulate memory** that is **never freed**.

```c
/* Function with memory leak */
#include <stdlib.h>

void f()
{
    int *ptr = (int *) malloc(sizeof(int));

    /* Do some work */

    return; /* Return without freeing ptr*/
}
```

To avoid memory leaks, memory allocated on heap should always be freed when no longer needed.

```
/* Function without memory leak */
#include <stdlib.h>;

void f()
{
   int *ptr = (int *) malloc(sizeof(int));

   /* Do some work */

   free(ptr);
   return;
}
```

Memory Leak

```
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. There is no way to access (or free it) now, as there are no ways to get to it anymore.

Memory fragmentation

Memory fragmentation: This occurs when a program **chaotically allocates** and **deallocates memory** in **varying sizes**.

A fragmented memory has allocated and **free memory chunks interspersed**, and often the **free memory chunks become too small to use**.

Defragmentation is required. Defragmentation and garbage collection are both very problematic for real-time systems.

For this example, it is assumed there is a 10K heap. First, an area of 3K is requested, thus:

```
#define K (1024)

char *p1;

p1 = malloc(3*K);
```

Then, a further 4K is requested:

```
p2 = malloc(4*K);
```

3K of memory is now free.

Some time later, the first memory allocation, pointed to by p1, is de-allocated:

```
free(p1);
```

This leaves 6K of memory free in two 3K chunks. A further request for a 4K allocation is issued:

```
p1 = malloc(4*K);
```

This results in a failure – NULL is returned into p1 – because, even though 6K of memory is available, there is not a 4K contiguous block available. This is memory fragmentation.

Suppose for a simple toy example that you have ten bytes of memory:

```
|   |   |   |   |   |   |   |   |   |   |
  0   1   2   3   4   5   6   7   8   9
```

Now let's allocate three three-byte blocks, name A, B, and C:

```
| A | A | A | B | B | B | C | C | C |   |
  0   1   2   3   4   5   6   7   8   9
```

Now deallocate block B:

```
| A | A | A |   |   |   | C | C | C |   |
  0   1   2   3   4   5   6   7   8   9
```

Variables/automatic variables ---> stack section

Dynamically allocated variables ---> heap section

Initialized global variables -> data section

Uninitialized global variables -> data section (bss)

Static variables -> data section

String constants -> text section/code section

Text code -> text section/code section

Registers -> CPU registers

Tools

Cppcheck

Valgrind

splint

RATS

SMATCH

Array

An array is collection of items stored at continuous memory locations. The idea is to declare multiple items of same type together.

In C, declare an array by specifying its type and size or by initializing it or by both.

```
// Array declaration by specifying size
int arr[10];
// Array declaration by initializing elements
int arr[] = {10, 20, 30, 40}
// Array declaration by specifying size and initializing  elements
int arr[6] = {10, 20, 30, 40}
```

Array elements are accessed by using an integer index. Array index starts with 0 and goes till size of array minus 1.

Multidimensional Arrays

Multi-dimensional arrays are declared by **providing more** than **one set** of **square** [ ] brackets after the **variable name** in the declaration statement.

For two dimensional arrays, the **first dimension** is commonly considered to be the **number** of **rows**, and the **second dimension** the number of columns.

```
int a[ 2 ][ 3 ] = { { 5, 6, 7 }, { 10, 20, 30 } };
```

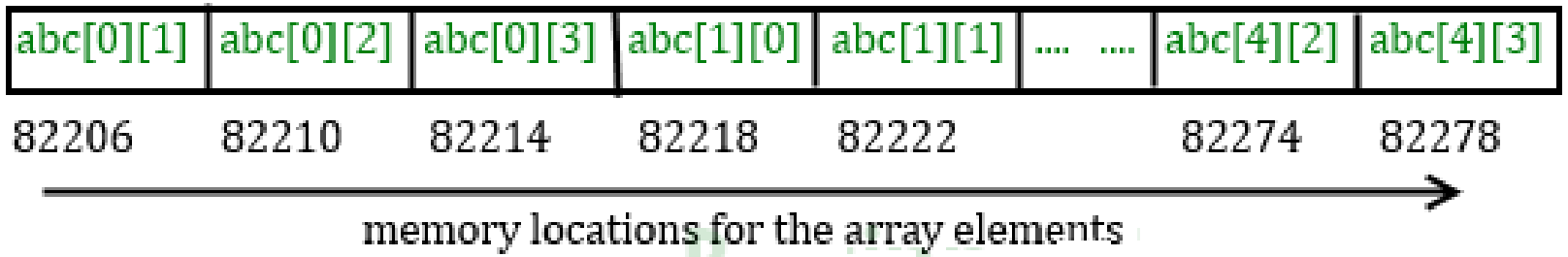
	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

2D array conceptual memory representation

abc[0][0]	abc[0][1]	abc[0][2]	abc[0][3]
abc[1][0]	abc[1][1]	abc[1][2]	abc[1][3]
abc[2][0]	abc[2][1]	abc[2][2]	abc[2][3]
abc[3][0]	abc[3][1]	abc[3][2]	abc[3][3]
abc[4][0]	abc[4][1]	abc[4][2]	abc[4][3]

Here my array is abc[5][4], which can be conceptually viewed as a matrix of 5 rows and 4 columns. Point to note here is that subscript starts with zero, which means abc[0][0] would be the first element of the array.

Actual representation of this array in memory



Array is of integer type so each element would use 4 bytes that s the reason there is a difference of 4 in element's addresses.

The addresses are generally represented in hex. This diagram shows them in integer just to show you that the elements are stored in contiguos locations, so that you can understand that the address difference between each element is equal to the size of one element(int size 4). For better understanding see the program below.

Actual memory representation of a 2D array

- A **2D array** is **stored** in the computer's memory **one row following another**.
- The **address of the first byte of memory** is considered as the memory **location** of the **entire 2D array**.
- Knowing the address of the first byte of memory, the compiler can easily compute to find the memory location of any other elements in the 2D array provided the number of columns in the array is known.
- If each data value of the array requires **B bytes** of memory, and if the array has **C columns**, then the memory location of an element such as score[m][n] is

  (m*c+n)*B

  from the address of the first byte.
- Note that **to find the memory location of any element**, **there is no need to know the total number of rows in the array,** i.e. **the size of the first dimension**. Of course the size of the first dimension is needed to prevent reading or storing data that is out of bounds.
- Again one should not think of a 2D array as just an array with two indexes. You should think of it as an **array of arrays.**

Higher dimensional arrays should be similarly **interpreted**. For example a **3D array should be thought of as an array of arrays of arrays**. To find the **memory location of any element in the array relative to the address of the first byte**, the **sizes of all dimensions other than the first must be known.**

Knowledge of how multidimensional arrays are stored in memory helps one understand how they can be initialized, and how they can be passed as function arguments.

ADC Configuration

Points to remember

- Configuration registers
- Result registers
- Logic for combining two result register values
- Resolution
- Converting digital values to Analog value.

********** Start Thinking about functions for ADC peripheral**********

Polling and Interrupts

Polling

- Continuously monitors the status for an event.
- The microcontroller must "access by itself" and “ask” for the information it needs for processing.
- Waste of time.
- External events are random in nature.

Interrupts

- Asynchronous signal indicating an event which needs the processors attention immediately regardless of the instruction it executes currently.
- Interrupt is “requesting" the processor to stop performing the current program and to “make time” to execute a special code.
- Upon receiving an interrupt signal, the microcontroller interrupts whatever it is doing and serves the interrupt.
- The program which is associated with the interrupt is called the **interrupt service routine (ISR) or interrupt handler.**

Sources of interrupt

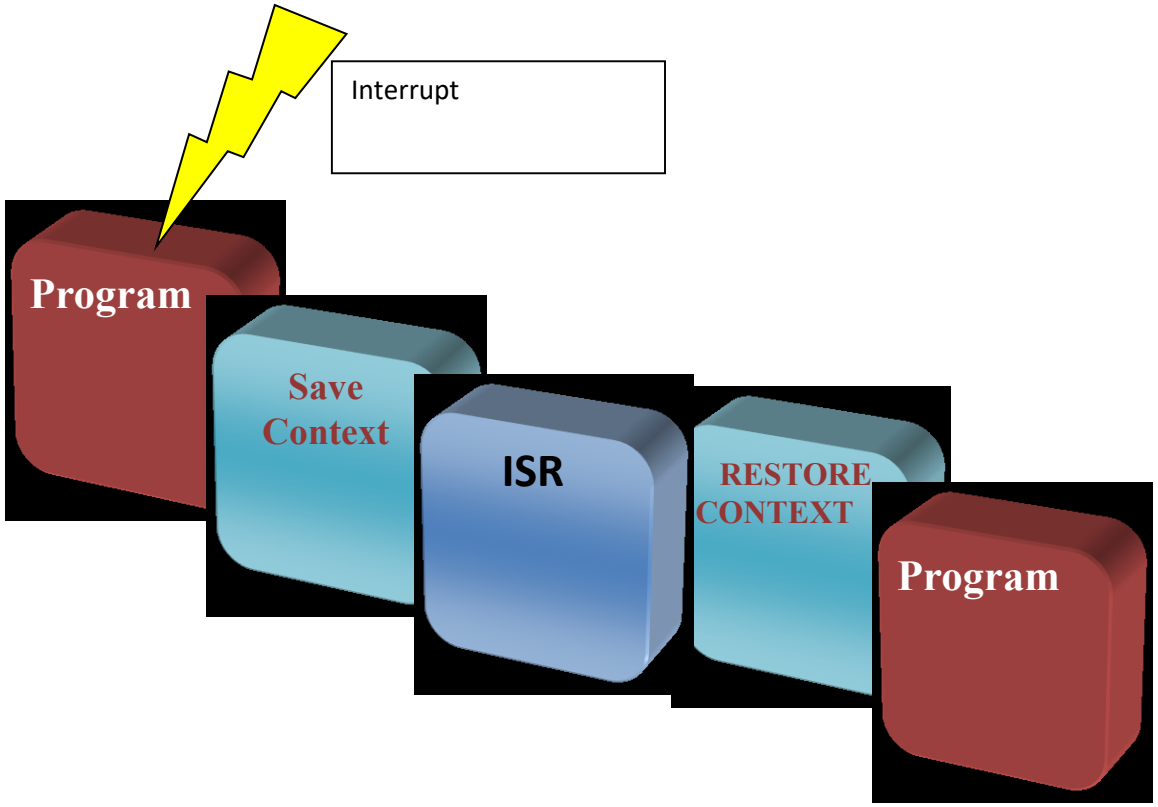
Hardware.

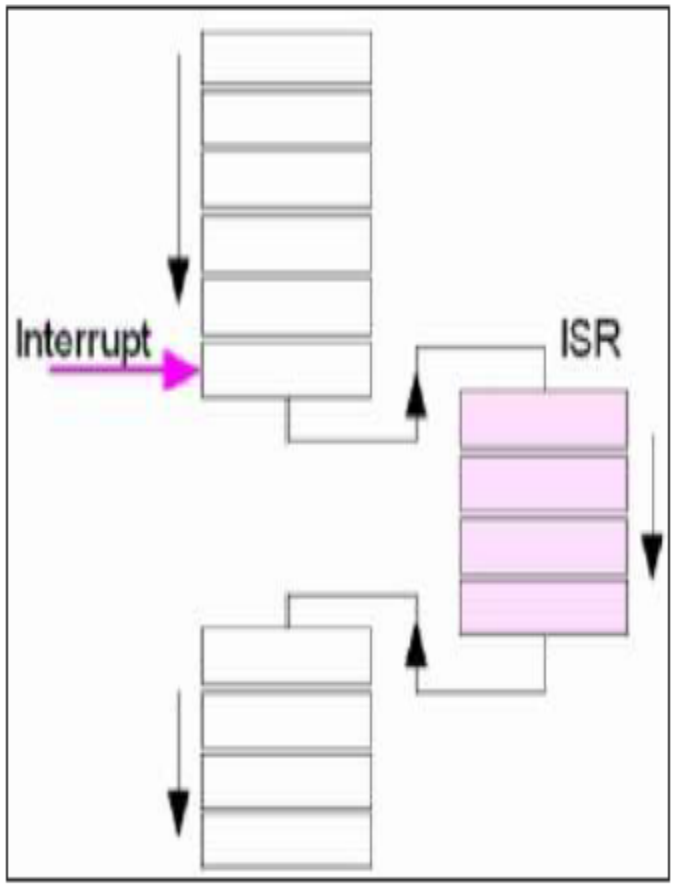
External hardware devices. Common example is pressing on the key on the keyboard, which causes to the keyboard to send interrupt to the microcontroller to read the information of the pressed key.

Software.

The processor or the peripherals can produce an interrupt like timer overflow.

Interrupt Handling & Context switching





Difference Between Polling and Interrupt

Background of Polling and Interrupt

In the first case, the processor checks at regular time intervals if a device needs an action. In case of an interrupt there is a mechanism by which the processor allows the external device (e.g. keyboard, sound card, etc.) to attract the processor's attention.

Mechanism of Polling and Interrupt

Interrupts are specially organized mechanisms for communication of peripheral devices. The devices notify the CPU if an action is needed. Polling is protocol – the CPU asks the devices regularly if an action is required.

Servicing of Polling and Interrupt

In polling the microcontroller services the device needing attention, and after that moves to the next device for monitoring. In case of interruption, when a signal for interruption is received, the CPU stops with the current activity and services the device. The services or the interruption processed is named interrupt service routine (ISR) or interrupt handler.

CPU

In the polling process, the CPU is on hold and checks if any device needs a service. This unnecessarily wastes time. In case of interruption process, on the other hand, the CPU is disturbed only if needed.

Appearance of Polling and Interrupt

The devices can be polled only at the regular interval when they are checked. Interruption can happen in any given time.

Advantages of Polling and Interrupt

Some of the advantages of polling are the relatively simple program, transmission reliability that takes place at maximum speed, i.e. as soon as the I/O device is ready and the no need of additional access chips. Interruption is beneficial because it can serves multiple devices, it is more flexible and efficient.

Disadvantages of Polling and Interrupt

Disadvantages of polling are the standby time of some devices that is shorter than the response time and then another method of transmission should be applied, as well as that the CPU consumes unnecessary time to check devices that have not searched for data transfer. Disadvantages of interrupts are the requirement for more complex hardware/software and loss of time until the CPU establishes which units request for interruption.

Criterion	Polling	Interrupt
Background	Checking at regular intervals	Processor is called if needed
Mechanism	Protocol	Mechanism
Servicing	CPU	Interrupt handler
CPU	On hold	Called if needed
Appearance	On regular interval	Anytime
Advantages	Simple program, transmission reliability, no need for additional chips	Serves multiple devices, flexible, efficient
Disadvantages	Standby time, time waste	More complex, time consuming

BASIS FOR COMPARISON	INTERRUPT	POLLING
Basic	Device notify CPU that it needs CPU attention.	CPU constantly checks device status whether it needs CPU's attention.
Mechanism	An interrupt is a hardware mechanism.	Polling is a Protocol.
Servicing	Interrupt handler services the Device.	CPU services the device.
Indication	Interrupt-request line indicates that device needs servicing.	Comand-ready bit indicates the device needs servicing.
CPU	CPU is disturbed only when a device needs servicing, which saves CPU cycles.	CPU has to wait and check whether a device needs servicing which wastes lots of CPU cycles.
Occurrence	An interrupt can occur at any time.	CPU polls the devices at regular interval.
Efficiency	Interrupt becomes inefficient when devices keep on interrupting the CPU repeatedly.	Polling becomes inefficient when CPU rarely finds a device ready for service.
Example	Let the bell ring then open the door to check who has come.	Constantly keep on opening the door to check whether anybody has come.