LANGUAGES AND TOOL FOR EMBEDDED SYSTEM DESIGN

LUIS F. GONZALEZ PEREZ, PHD

TE3059 EMBEDDED SYSTEMS

COMPUTER SCIENCE DEPARTMENT

ITESM - GDA

OUTLINE

- Embedded system build tools
- C language in embedded system design
- Memory Types, Segments and Management (from a C language perspective)

OUTLINE

- **Embedded system build tools**
- C language in embedded system design
- Memory Types, Segments and Management (from a C language perspective)

THE BUILD PROCESS

- Embedded system programming is not substantially different from the programming you've done before
- The only thing that changes in embedded system programming is that the target hardware platform is unique.
- Unfortunately, that one difference leads to a lot of additional software complexity, and it is also the reason why the programmer must be more aware of the software build process than ever before, and more aware of the hardware platform his/her code has to be targeted to

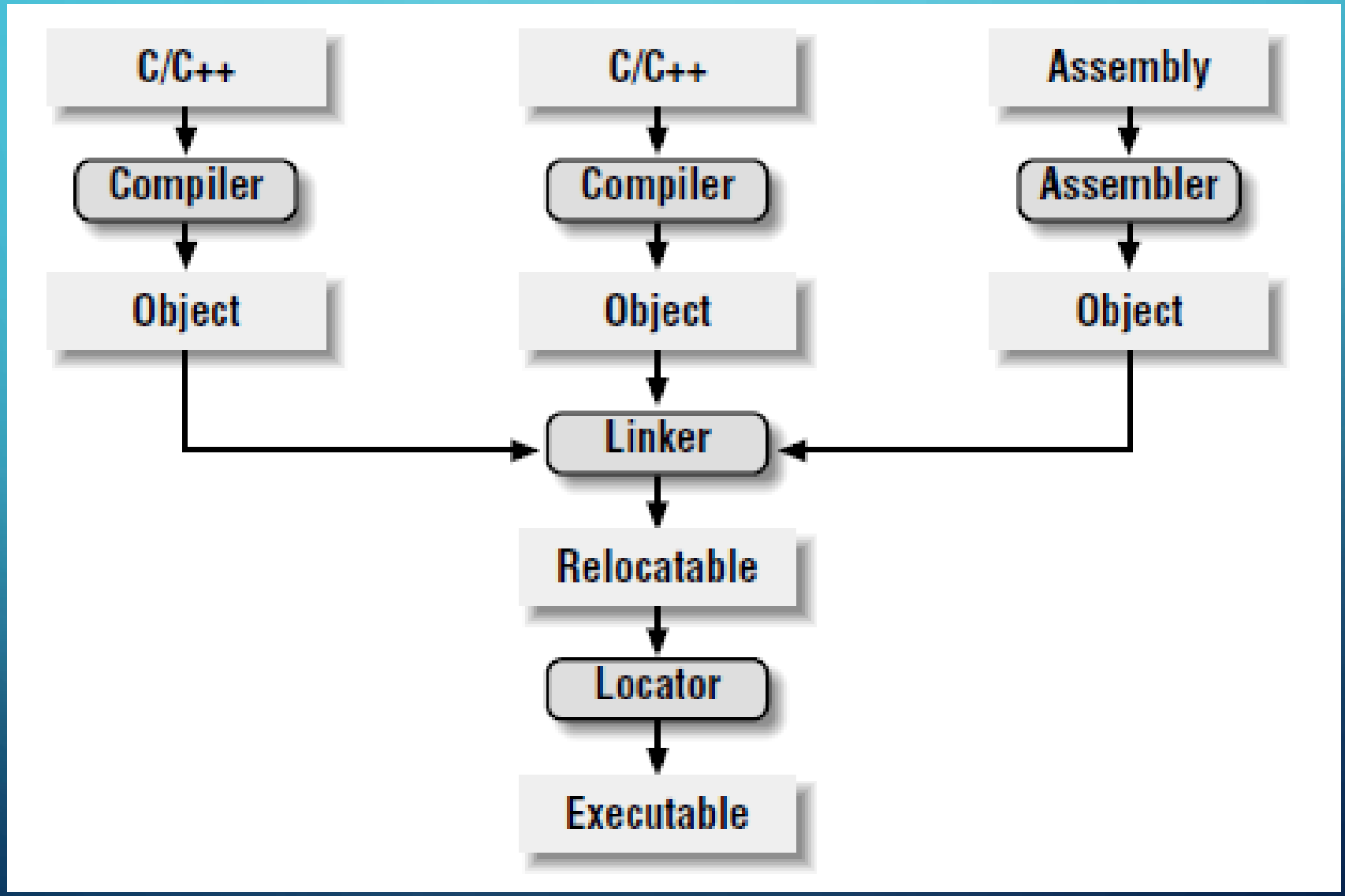
THE BUILD PROCESS

- Embedded software development tools can rarely make assumptions about the target platform (cause each one is unique)
- That is why the programmer must provide some of his own knowledge of the system to the tools by giving them more explicit instructions
- The process of converting the source code representation of an embedded software into an executable binary image involves **three distinct steps**

BUILD PROCESS STEPS

1. Each of the source files must be compiled to generate the assembly code
2. The assembly code is assembled into an **object file** or **machine code**
3. All of the object files that result from the second step must be linked together to produce a single object file, called the **relocatable program**
4. Physical memory addresses must be assigned to the relative offsets within the relocatable program. This step is called relocation. The result is a file that contains an **executable binary image** that is ready to be run on the embedded system

BUILD PROCESS STEPS



COMPILING

- A compiler's duty is to translate a program written in a human-readable language into an equivalent set of opcodes (assembly code) for a particular processor
- An assembler is also a compiler but one that performs a much simpler one-to-one translation from one line of mnemonics to the equivalent opcode
- Compilers/Assemblers are the first step of the embedded software build process

COMPILING

- The output of the compiler will be an object file, a specially formatted binary file that contains the set of instructions and data resulting from the language translation process
- The contents of an object file can be thought of as a very large data structure that is usually defined by a standard format like **Common Object File Format (COFF)** or **Extended Linker Format (ELF)**

COMPILING

- Most object files begin with a header that describes the sections that follow
- Each of these sections contain one or more blocks of code or data that originated within the original source file but have been regrouped by the compiler into related sections
- For example
  - All the code blocks are collected into a section called **text**
  - Initialized global variables into a section called **data**
  - Uninitialized global variables into a section called **bss**

COMPILING

- There is also a symbol table somewhere in the object file that contains the names and locations of all the variables and functions referenced within the source file
- Parts of this table may be incomplete because not all the variables and functions are defined in the same file. It is up to the linker to resolve such unresolved references of variables and functions that are defines in other files

LINKING

- All of the object files resulting from compiling must be combined in a special way before the program can be executed
- The job of the linker is to combine these object files and resolve all of the unresolved symbols
- The output of the linker is a new object file that contains all of the code and data from the input object files and is in the same object file format
- It merges the text, data and bss sections of the input files
- When the linker is finished executing, all of the machine language code from all of the input ovject files will be in the **text** section of the new file, and all of the initialized and unitialized variables will reside in the new **data** and **bss** sections, respectively.

LINKING

- After merging all of the code and data sections and resolving all of the symbol references, the linker produces a special “relocatable” copy of the program
- This means that the program is complete except for one little thing, no memory addresses have been assigned to the code and data sections yet
- If we weren’t working on an embedded system, we’d be finished building our software
- For embedded applications, we need an absolutely located binary image

LOCATING

- The locator is the tool that converts a relocatable program to executable binary image
- However, the programmer is responsible of doing most of the work in this step
- The programmer has to provide 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
- Then, it will produce an output file that contains the binary memory image that can be loaded into the target ROM

LOCATING

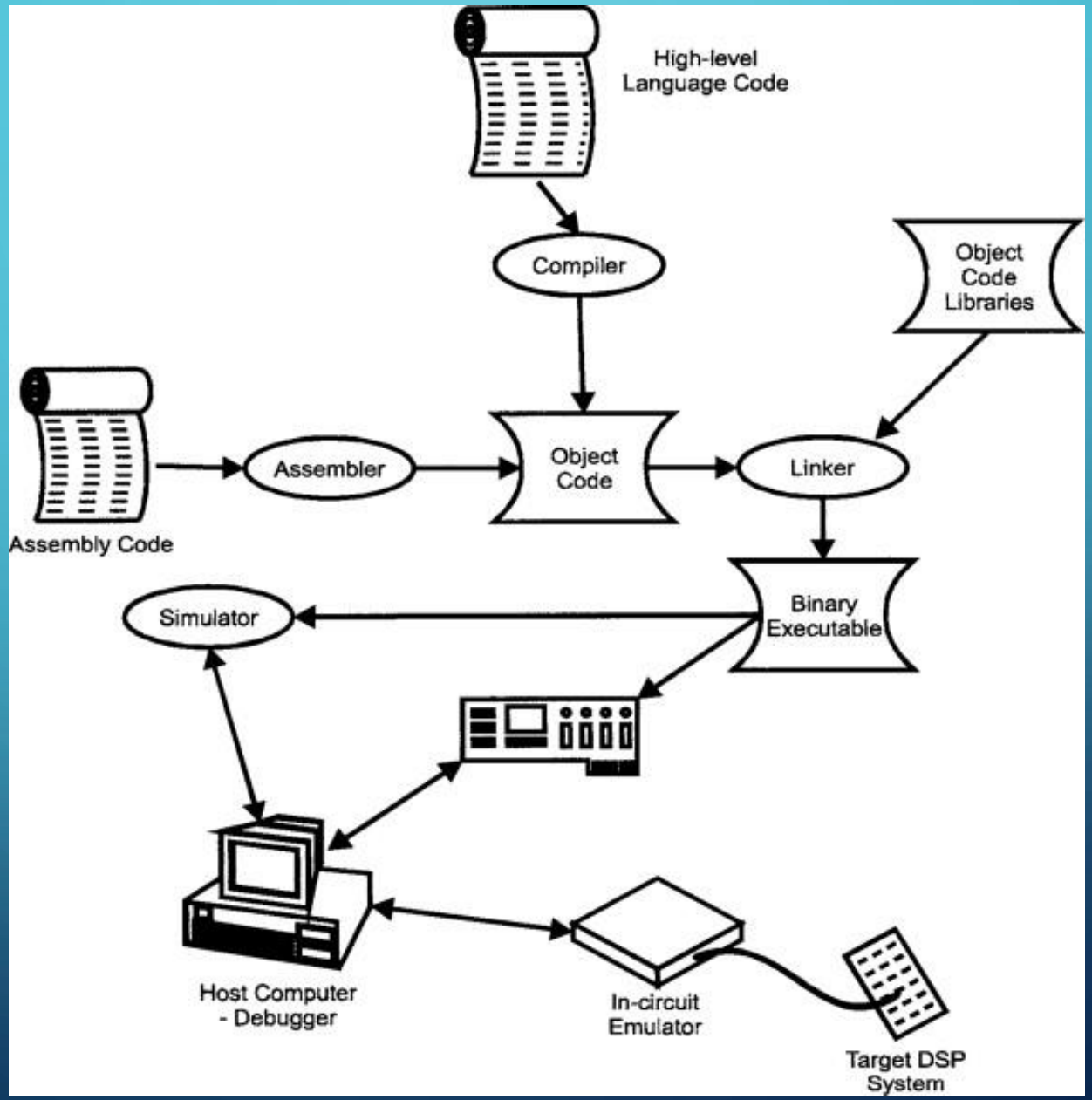
- In many cases, the locator is part of the linker
- The memory information required by the linker can be passed in the form of a linker script
- These scripts are used to control the exact order of the code and data sections within the relocatable program
- Moreover, it is also possible to use these scripts to **establish the location of each section in memory**

LOCATING - EXAMPLE

- The embedded system has 512KB of RAM and ROM
- The script instructs the locator to locate the **data** and **bss** sections in RAM, starting at address 0x00000
- The **text** section will go in ROM, starting at address 0x8000
- The **initial values** of the variables in the data segment will be made part of the ROM image by the addition of the **>rom**

```
MEMORY
{
    ram : ORIGIN = 0x00000, LENGTH = 512K
    rom : ORIGIN = 0x80000, LENGTH = 512K
}

SECTIONS
{
    data ram :                          /* Initialized data.        */
    {
        _DataStart = . ;
        *(.data)
        _DataEnd   = . ;

    } >rom

    bss :                               /* Uninitialized data.      */
    {
        _BssStart = . ;
        *(.bss)
        _BssEnd   = . ;
    }

    _BottomOfHeap = . ;                 /* The heap starts here.    */
    _TopOfStack = 0x80000;              /* The stack ends here.     */

    text rom :                            /* The actual instructions.  */
    {
        *(.text)

    }
}
```

BUILD PROCESS SUMMARY



OUTLINE

- Embedded system build tools
- **C language in embedded system design**
- Memory Types, Segments and Management (from a C language perspective)

C LANGUAGE FOR EMBEDDED SYSTEMS. WHY?

- The C programming language is one of the few constants across any embedded system
- C has become the language of choice of embedded programmers
- Why?
  - It is small and fairly simple to learn
  - Compilers are available for almost every processor in use today
  - C has the benefit of being processor-independent, which allows programmers to focus on algorithms and applications, rather than on the details of a particular processor architecture
  - BUT MOST IMPORTANTLY, C is a very "low level" high-level language. It gives embedded programmers and extraordinary degree of direct hardware control without sacrificing the benefits of high-level languages

BASIC C PROGRAM ANATOMY

```
#include "STM32L1xx.h"      /* I/O port/register names/addresses for the STM32L1xx microcontrollers */

/* Global variables – accessible by all functions */
int count, bob;             //global (static) variables – placed in RAM

/* Function definitions*/
int function1(char x) {     //parameter x passed to the function, function returns an integer value
  int i,j;                  //local (automatic) variables – allocated to stack or registers
  -- instructions to implement the function
}

/* Main program */
void main(void) {
   unsigned char sw1;       //local (automatic) variable (stack or registers)
   int k;                   //local (automatic) variable (stack or registers)
  /* Initialization section */
   -- instructions to initialize  variables, I/O ports, devices, function registers
  /* Endless loop */
   while (1) {              //Can also use:  for(;;) {
    -- instructions to be repeated
   } /* repeat forever */
}
```

Declare local variables

Initialize variables/devices

Body of the program

VARIABLES DECLARATION

- A **variable** is an addressable storage location to data to be used by the program
- Each variable must be **declared** to indicate size and type of data to be stored, and name to be used to reference information
- Varible declaration formt

```
type-qualifier(s) type-modifier data-type variable-name = initial-value;
```

- Example

```
const unsigned char toto = 12;
long int tutu;
…
tutu = 400;
```

Memory space for variables can vary from registers, RAM or ROM, depending on the type-qualifier!!!

VARIABLES DECLARATION

- **Data types** can be:
  - Char
  - Integer
  - Floating point
  - Enumarated
  - Derived
  - Void

```
const unsigned char toto = 12;
long int tutu;
…
tutu = 400;
```

VARIABLES DECLARATION

- **Type modifiers** are used to change the size of these types or change the properties of the variables
- Type modifiers can be
  - Short
  - Long
  - Unsigned
  - signed

```
const unsigned char toto = 12;
long int tutu;
…
tutu = 400;
```

VARIABLES DECLARATION

- **Type qualifiers** are used to provide specific instructions to the compiler on how the variables must be managed
- Type qualifiers can be
  - const
  - volatile
  - static
  - register
  - extern
  - …

DATA TYPES EXAMPLE

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

TYPEDEF

- typedef is a keyword that defines a new type of variable created by the programmer
- These new variables are supposed to ease the readability of the code

```
typedef int new_int;

new_int result;
```

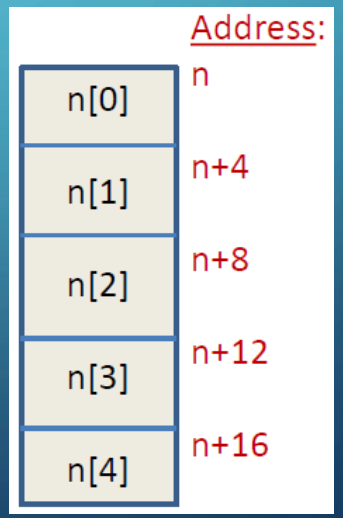
COMPLEX DATA TYPES

- Data types that are extremely useful in organizing an embedded program
- These are
  - Pointer (discussed later in this presentation)
  - Arrays
  - Enumerated types
  - Unions
  - Structures

ARRAYS

- An array is a set of data stored in contiguous block of memory, beginning at a named address
- Arrays can be multidimensional
  - An array is declared by its name and number of data elements, N
  - Elements are “indexed”, indices are [0.. N-1]

```
int n[5];
```



ENUMERATED TYPES

- Enumerated types are finite sets of named values defined by the programmer
- This allow the code to be more readable and maintainable
- Examples

```
enum
{
    state1,
    state2,
    state3
};
```

```
void function (void)
{
    static int st = state1;
    switch (st)
    {
        case state1:
            //code
        break;
        case state2:
            //code
        breadk;
        case state3:
            //code
        break;
    }
};
```

ENUMERATED TYPES

- Enumerated types combined with typedef serve to create enum types that should render the code more organized

```
Typedef enum
{
    GPS_OFF;
    GPS_ON;
} teGPS_ON_OFF;

void fnGPS (teGPS_ON_FF GpsStatus)
{
    if (GpsStatus == GPS_OFF)
    {
        //code to turn off the GPS
    }
    else if (GpsStatus == GPS_ON)
    {
        //code to turn the GPS on
    }
}

int main (void)
{
    fnGPS (GPS_OFF);
    while (1)
    {
    }
}
```

STRUCTURES

- Structures are used to group meaningful data of different types into a single variable name
- They are thoroughly used in embedded systems development to build understandable data structures
- Like arrays, these data are stored in contiguous memory locations
- Combined with the typedef keyword, the creation of a structure is more readable

STRUCTURE EXAMPLE 1

```
Struct the_time {
        unsigned int hours;
        unsigned int minutes;
        unsigned int seconds;
};
```

```
Int main (void)
{
        struct the_time my_time;

        my_time.hours = 7;
        my_time.minutes = 30;
        my_time.seconds = 55;

        while (1)
        {
        }
}
```

STRUCTURE EXAMPLE 2

```
typedef struct {
        unsigned int hours;
        unsigned int minutes;
        unsigned int seconds;
} the_time;
```

```
Int main (void)
{
        the_time my_time;
        my_time.hours = 7;
        my_time.minutes = 30;
        my_time.seconds = 55;

        while (1)
        {
        }
}
```

UNIONS

- Unions are similar to structures except that the elements of the union share the same memory location.
- It is useful when we don't want to waste memory or memory is scarce
- Unions can be thought of as a memory location used to store temporary variables of different types and sizes.
- The size of the union is the size of its biggest element

UNIONS EXAMPLE

```
union data32
{
        int DoubleWord;
        struct TwoWord
        {
                short Word_L;
                short Word_H;
        } Word;
        struct FourBytes
        {
                char byte_0;
                char byte_1;
                char byte_2;
                char byte_3;
        } Byte;
};
```

```
Int main (void)
{
        union data32 my_data32;
        my_data32.DoubleWord = 0x12345678;
        printf("%x\r\n", my_data32.DoubleWord);    //12345678
        printf("%x\r\n", my_data32.Word.Word_H);   //1234
        printf("%x\r\n", my_data32.Word.Word_L);   //5678
        printf("%x\r\n", my_data32.Byte.byte3);    //12
        printf("%x\r\n", my_data32.Byte.byte2);    //34
        printf("%x\r\n", my_data32.Byte.byte1);    //56
        printf("%x\r\n", my_data32.Byte.byte0);    //78
}
```

OPERATORS

- Data is manipulated using a variety of operators

Type	Operators
Logical	||, &&, !
Bitwise	<<, >>, |, &, ^, ~
Arithmetic	+, -, /, *, ++, --, %
Relational	<, <=, >, =>, ==, !=

LOGICAL OPERATORS

- Boolean conditions
  - False: any condition that is zero
  - True: any non-zero condition

`||`	logical OR
`&&`	logical AND
`!`	Logical NOT

Usage:

```
if (conditionX || condition Y)
if (condition && condition Y)
while (!condition)
```

BITWISE OPERATORS

- Operations performed on a bit-by-bit basis in a variable or variables
  - << left shift
  - >> right shift
  - | bitwise OR
  - & bitwise AND
  - ^ bitwise XOR
  - ~ one's complement

& (AND) | (OR) ^ (XOR) ~ (Complement)

```
C = A & B;          A   0 1 1 0 0 1 1 0
(AND)               B   1 0 1 1 0 0 1 1
                    C   0 0 1 0 0 0 1 0

C = A | B;          A   0 1 1 0 0 1 0 0
(OR)                B   0 0 0 1 0 0 0 0
                    C   0 1 1 1 0 1 0 0

C = A ^ B;          A   0 1 1 0 0 1 0 0
(XOR)               B   1 0 1 1 0 0 1 1
                    C   1 1 0 1 0 1 1 1

B = ~A;             A   0 1 1 0 0 1 0 0
(COMPLEMENT)        B   1 0 0 1 1 0 1 1
```

BIT SET, RESET, COMPLEMENT, TEST

- Use a "mask" to select bit(s) to be modified

```
C = A & 0xFE;     A     a b c d e f g h
                  0xFE  1 1 1 1 1 1 1 0   Clear selected bit of A
                  C     a b c d e f g 0

C = A & 0x01;     A     a b c d e f g h
                  0xFE  0 0 0 0 0 0 0 1   Clear all but the selected bit of A
                  C     0 0 0 0 0 0 0 h

C = A | 0x01;     A     a b c d e f g h
                  0x01  0 0 0 0 0 0 0 1   Set selected bit of A
                  C     a b c d e f g 1

C = A ^ 0x01;     A     a b c d e f g h
                  0x01  0 0 0 0 0 0 0 1   Complement selected bit of A
                  C     a b c d e f g h'
```

BIT OPERATIONS EXAMPLES

- Create a pulse on bit 0 of PORTA (assuming bit is initially 0)

```
PORTA = PORTA | 0x01;  //force bit 0 to 1
PORTA = PORTA & 0xFE;  //force bit 0 to 0

if ((PORTA & 0x80) != 0)
   toto();                  //call toto if bit 7 of PORTA is 1
c = PORTB & 0x04;           //mask all but bit 2 of PORTB
if ((PORTA & 0x01) == 0)    //test bit 0 of PORTA
   PORTA = c | 0x01;        //write c to PORTA with bit 0 set to 1
```

SHIFT OPERATORS

- Shift operators

  x >> y (right shift operand x by y bit positions)

  x << y (left shift operand x by y bit positions)

- Vacated bits are filled with zeroes
- Fast way to multiply/divide by powers of 2

```
B = A << 3;                A   1 0 1 0 1 1 0 1
(Left shift 3 bits)        B   0 1 1 0 1 0 0 0

B = A >> 2;                A   1 0 1 1 0 1 0 1
(Right shift 2 bits)       B   0 0 1 0 1 1 0 1

B = '1';                   B = 0 0 1 1 0 0 0 1 (ASCII 0x31)
C = '5';                   C = 0 0 1 1 0 1 0 1 (ASCII 0x35)
D = (B << 4) | (C & 0x0F);
       (B << 4)        = 0 0 0 1 0 0 0 0
     (C & 0x0F)        = 0 0 0 0 0 1 0 1
             D         = 0 0 0 1 0 1 0 1 (Packed BCD 0x15)
```

ARITHMETIC OPERATORS

+	addition
-	Subtraction
*	Multiply
/	divide
++	increment
--	decrement
%	modulus

- Examples

```
toto = A + B;
toto = A - B;
toto = A % 2;
A++;
A--;
```

RELATIONAL OPERATORS

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

- Examples

```
if (A <= B)
if (A == B)
while (A != 0)
while (A < 10)
```

PROGRAM FLOW CONTROL

```
if (condition) {
        // code
}
if(condition) {
        // code
}
else {
        //code
}
```

```
if (condition1) {
        // code
}
else
if(condition2) {
        // code
}
else {
        //code
}
```

```
switch (expression) {
        case const-exp1:
                // code
                break;
        case const-exp2:
                // code
                break;
        …
        default:
                //code
                break;
}
```

LOOPS

- Allows a block of code to be repeated based on a condition
- Three possibilities: for, while, do .. while

```
for ( variable-initialize; condition; variable expression){
    // code
}
```

```
while (condition) {
    //code
}
```

```
Do {
    //code
} while (condition);
```

FUNCTIONS

- Small blocks of idea used to organize a software idea that takes a list of paramentes and returns a piece of data
- They use the same types discussed before
- Help to manage program complexity
  - Smaller tasks are easier to design and debug
  - Offer reusability
- A function is called by another code to perform a task
  - The function may return a result to the caller
  - One or more arguments may be passed to the function/procedure

FUNCTION DECLARATION AND DEFINITION

- Functions require a declaration or prototype and a definition
- Prototype include a function type, function name and a parameter list

```
function-type  function-name  (param1-type param1, param2-type param2, …)
```

- Function definition includes the contents or definition of what the functions has to do

```
function-type  function-name  (param1-type param1, param2-type param2, …) {
    //code that function has to execute
}
```

Example declaration:

```
int main()
int toto (int k, int n);
int *bar (float data);
Void tutu( int *ptr);
```

Example definition:

```
int toto (int k, int n)
{
    int j;
    j = n+k-5;
    return (j);
}
```

FUNCTION ARGUMENTS

- Calling code can pass arguments to a function in two ways
  - By **value:** a variable or constant value is passed to the function. Function can use, but not modify the value
  - By **reference**: the address of the variable is passed. Function can both read and update the variable
- Values or addresses are typically passed to the function by pushing them onto the system stack

PASS BY VALUE EXAMPLE

```
// function to calculate the square of a number
int square (int x)
{
    int y;
    y = x*x;      //using the passed value
    return (y);   //return the result
}

Void main()
{
    int k,n;
    n = 5;
    k = square(n);
    n = square(5);
}
```

PASS BY REFERENCE EXAMPLE

```
// function to calculate the square of a number
void square (int x, int *y)
{
    *y = x*x;     //write the result to location whose address is y
}

void main()
{
    int k,n;
    n = 5;
    square(n, &k);   //compute square of n and put the result in k
    square(5, &n);   //compute square of 5 and put the result in n
}
```

POINTERS

- Special data types that hold address information
- There are three operators related to pointers

```
inf toto = 0x34;
```

- Declaration

```
int *ptr;
```

- Address-of operator

```
ptr = &toto;
```

- Dereference operator

```
*ptr = 0x52;
```

```
=> toto =  0x52;
```

POINTERS

```
int toto = 0x34;
int *ptr;
ptr = &toto;
*ptr = 0x52;
```

Memory	
Address	Data
0x000	0x0000BAE7
0x004	0xFF123496
0x108	0xFEBA1235

Memory	
Address	Data
0x000 (toto)	0x00000034
0x004	0xFF123496
0x108	0xFEBA1235

toto declaration
Int toto = 0x34;

Memory	
Address	Data
0x000 (toto)	0x00000034
0x004 (ptr)	0xFF123496
0x108	0xFEBA1235

Pointer definition
int *ptr;

POINTERS

```
int toto = 0x34;
int *ptr;
ptr = &toto;
*ptr = 0x52;
```

Memory	
Address	Data
0x000	0x00000034
0x004	0x00000000
0x108	0xFEBA1235

pointer variable holds the address of toto
ptr = &toto;

Memory	
Address	Data
0x000 (toto)	0x00000052
0x004	0x00000000
0x108	0xFEBA1235

Pointer dereferencing and Assigning a new value, changes totooto declaration
*ptr = 0x52;

OUTLINE

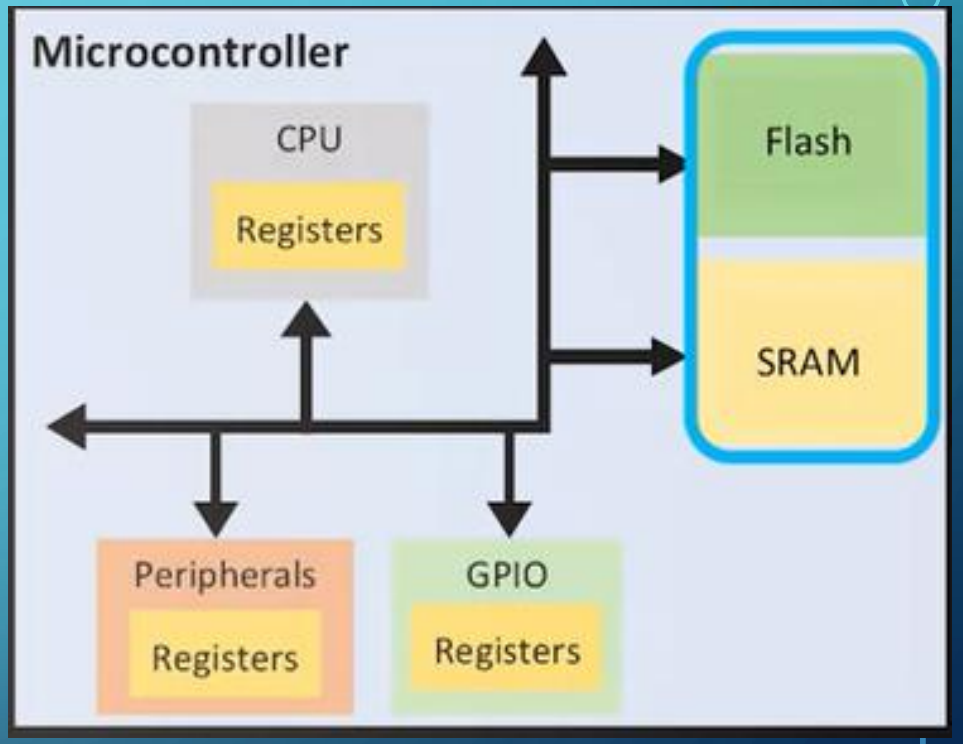
- Embedded system build tools
- C language in embedded system design
- **Memory Types, Segments and Management (from a C language perspective)**

MEMORY ORGANIZATION

- Memory is a key component of embedded systems and, unfortunately, one of the major limiting resources in embedded systems
- Embedded systems tend to have small amounts of memory as compared to desktops or data centers
- Hence, it is important to utilize and organize memory very efficiently in embedded systems

MEMORY ORGANIZATION

- Three types of storage needed for a program
  - Code memory (usually stored in Flash memory)
  - Data memory (usuallt stored in RAM
  - Runtime state program (register memory)
    - CPU registers
    - Peripheral registers



MEMORY SEGMENTS

- Register memory differs from SRAM and Flash as it will always have a set interface not dependent on applications
- Applications are free to use these memories in whatever capacity they need to accomplish their tasks
- This means that programs need to know some details about these memories before they can be installed and executed
- The linker accomplishes this by taking a high level program and separating it out into multiple components and these blocks are then installed on the physical memory

MEMORY SEGMENTS

- Much of the memory is in a physically different part of the chip or platform but access the same way
- This is referred to as an **address space**
- One oway to organize the memory in an embedded system is to use a **memory map**

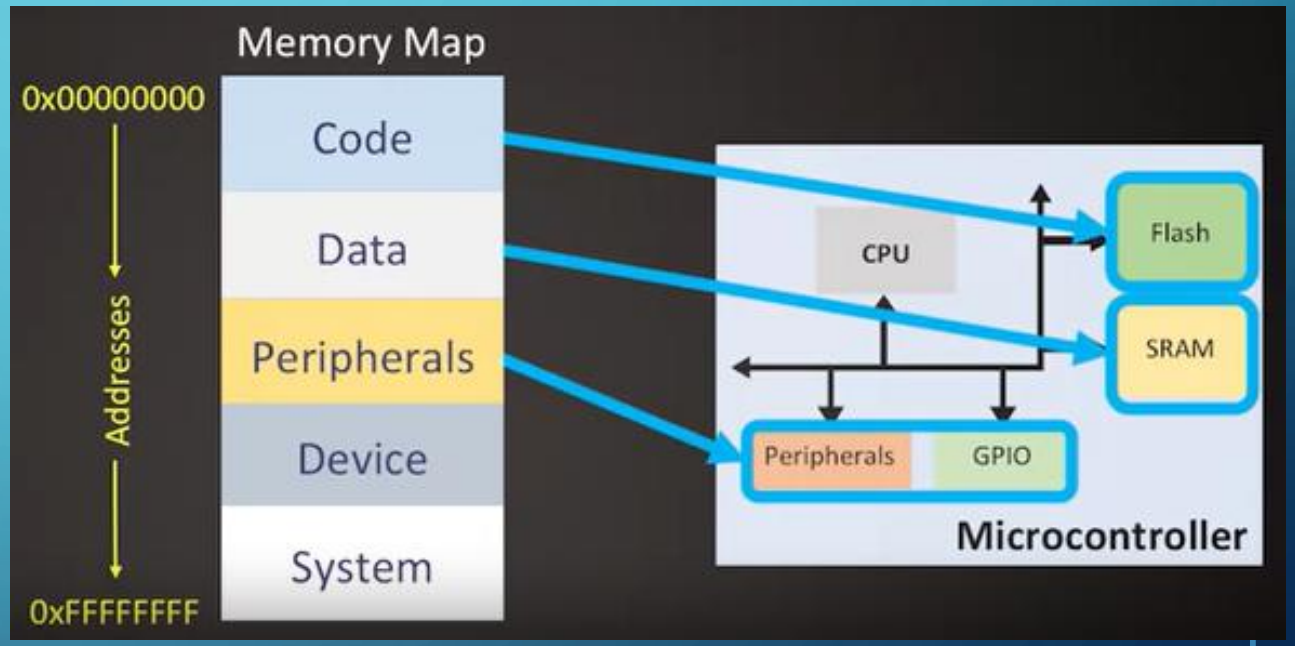
MEMORY SEGMENTS

- A memory map provides us a way to map different segments and sub-segments to the target memory
- There are mapped by assigning certain addresses in and **address space**
- This way, an embedded system can use a unified access method to memory via software and addresses

Memory region	Description	Access via	Address range
Code	Normally flash SRAM or ROM	ICode and Dcode bus	0x00000000-0x1FFFFFFF
SRAM	On-chip SRAM, with bit-banding feature	System bus	0x20000000-0x3FFFFFFF
Peripheral	Normal peripherals, with bit-banding feature	System bus	0x40000000-0x5FFFFFFF
External RAM	External memory	System bus	0x60000000-0x9FFFFFFF
External device	External peripherals or shared memory	System bus	0xA0000000-0xDFFFFFFF
Private peripheral bus	System devices, see Table 2-3 on page 2-25	System bus	0xE0000000-0xE00FFFFF
Vendor specific	-	-	0xE0100000-0xFFFFFFFF

MEMORY SEGMENTS

- With a unified address space, the addresses of memory segments may be kept in system across different chips in a family but do not guarantee these memories maintain the same address
- Luckily, the segments and the use of an address space to interact with these components allow the compiler to represent these in a unified mechanism
- The compiler does not need to know anything related to the platform, just the addresses that it needs to read and write to access these peripherals



DATA MEMORY

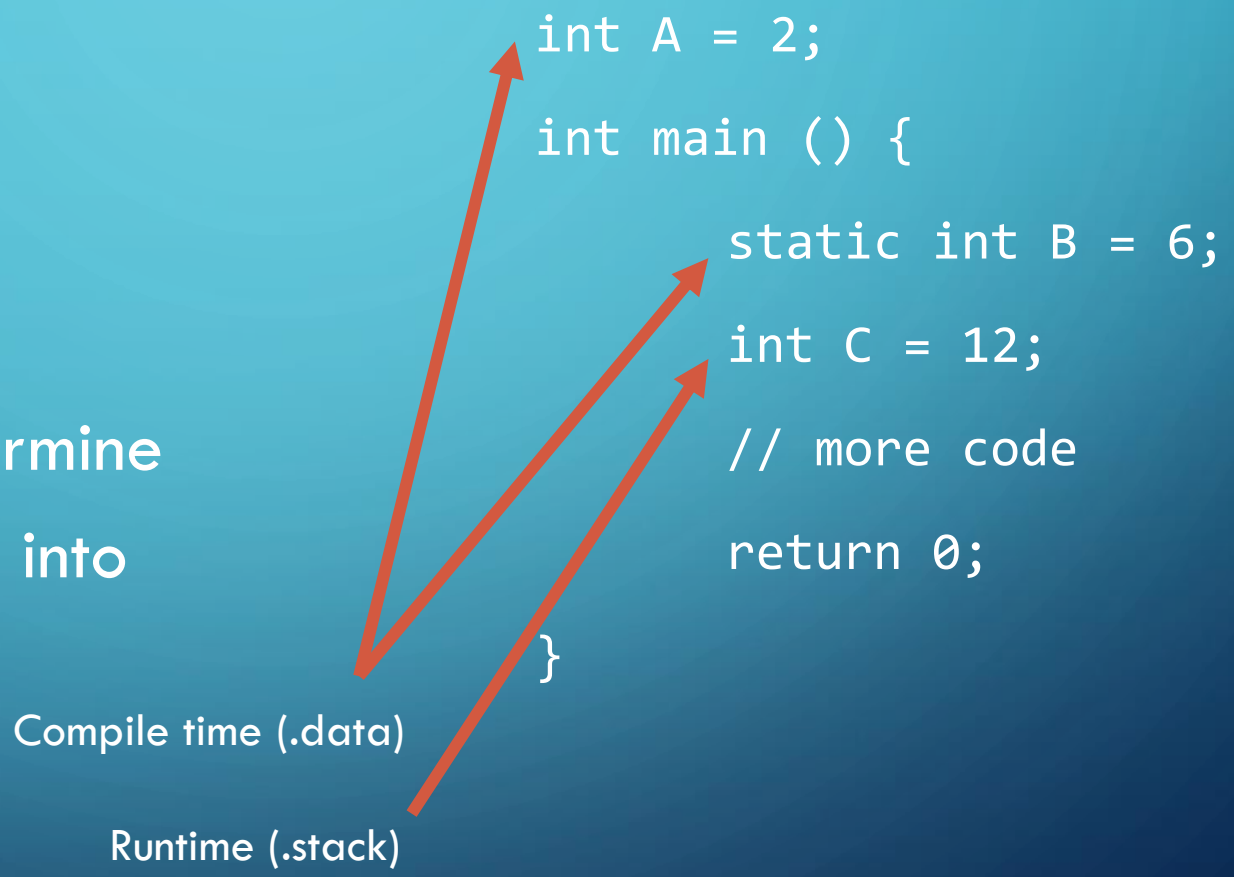
- Data memory stores our program's operands
- Data memory can have many different forms for data address information, configuration information or just raw application data
- What is data memory:
  - Pointers
  - Structures
  - Arrays
  - Single variables
  - and other type of data

DATA MEMORY

- The contents of data memory is constantly changing throughout the execution of a program
- This is done by loading data from data memory into CPU registers, operating on it and storing the result back into memory
- Because of this, instead of seeing data as a bunch of variables, we can see it as allocated data
- **IMPORTANT:** not all data we create needs to be in the form of a variable

DATA MEMORY

- Data can be allocated at compile time and runtime
- Data can have different characteristics which determine how they will be mapped into the **Data Segment**



DATA MEMORY

- The **Data Segment** is a method of organizing data memory at compile time in order to map addresses into physical memory
- Four main sub-segments
  - Stack
  - Heap
  - Data
  - BSS (Block Started by Symbol)
- Each segment size depends on how the program is written

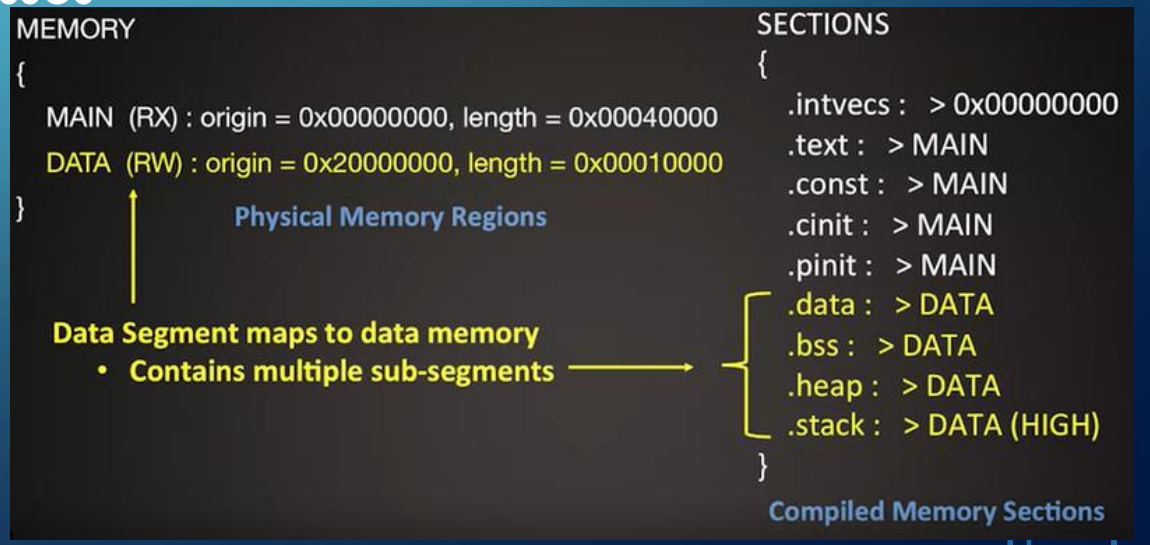
DATA SEGMENT

- **Stack** stores temporary data like local variables
- Heap stores dynamic information at runtime
- Data stores non-zero initialized global and static data
- BSS stores zero initialized and uninitialized global and static data

```
Int A_bss;
Int B_bss = 0;
Int C_data = 1;
Voit toto (int D_stack){
        int E_stack
        int F_stack
        static int G_bss;
        static int H_bss = 0;
        static int I_data = 1;

        //more code
        return;
}
```

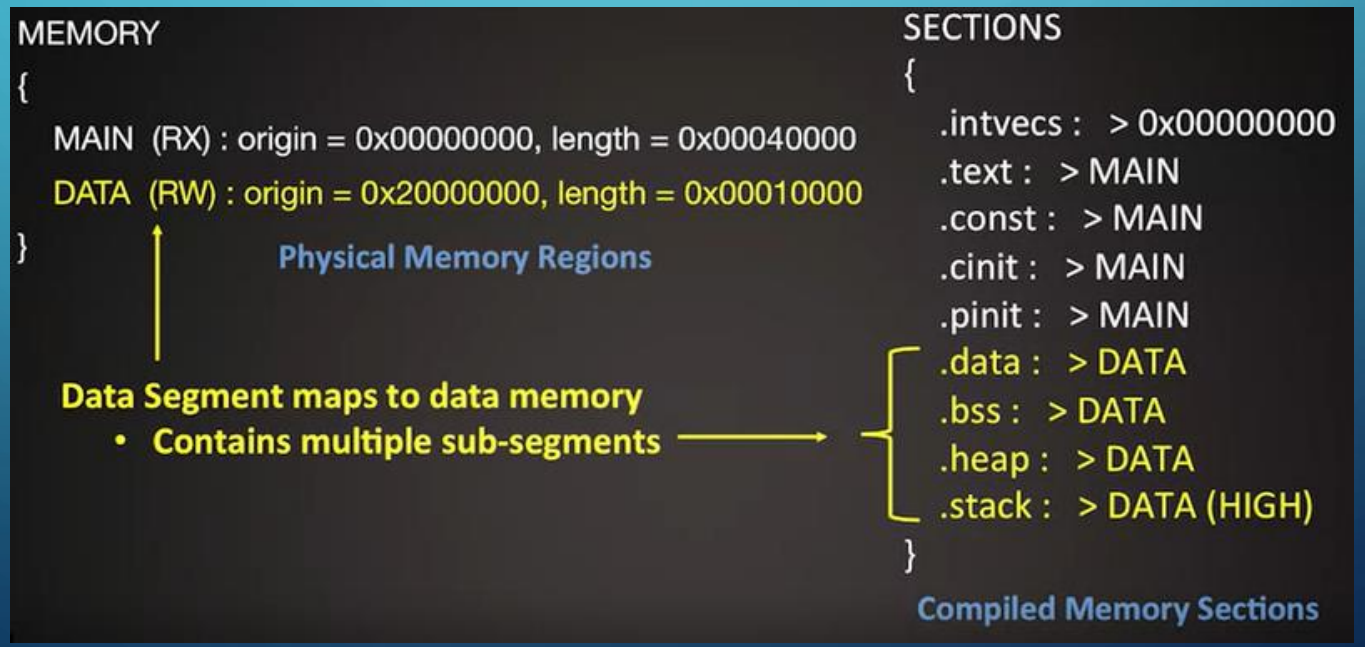
IMPORTANCE OF DATA SEGMENTS

- Code memory contains the operations to be performed on data
- Those operations need to know exact locations of where the data they will use is located
- The linker provides segment/sub-segment mapping to physical memories
- Carefull management of the code and data at compile time will substitute the loading and storing of data into the CPU using addresses and address offsets to unique locations in memory



IMPORTANCE OF DATA SEGMENTS

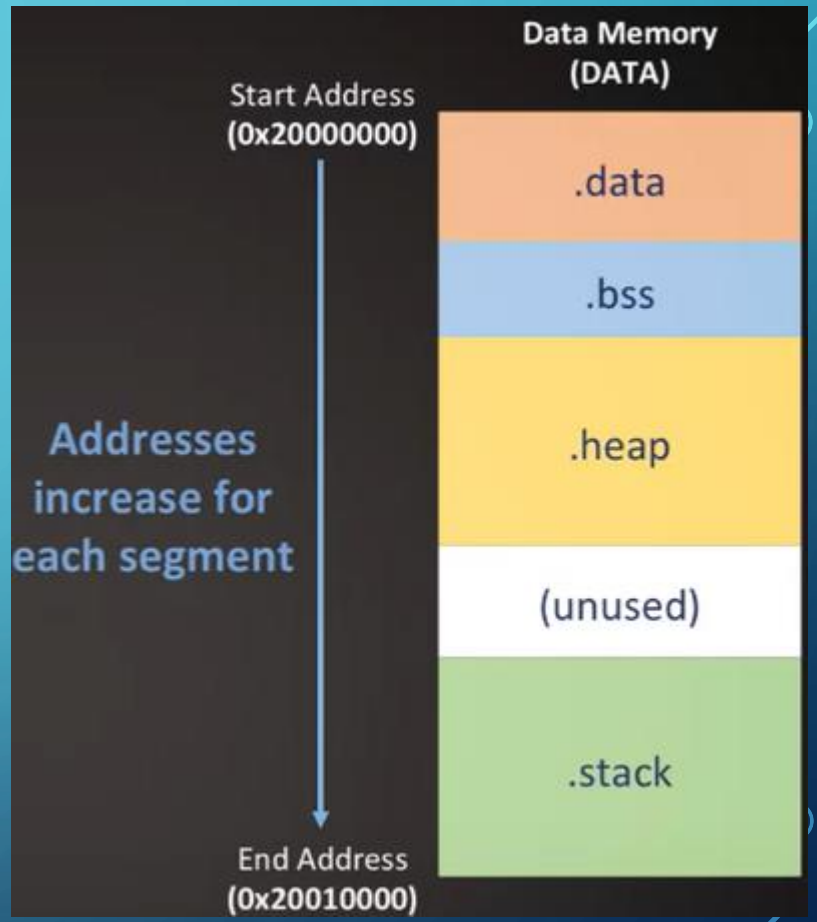
- Without a careful management of these locations and memory, the programmer could easily corrupt the data by mis-addressing a variable or writing outside the bounds of the segment space
- The linker file also specifies the order in which the sub-segments will be placed



DATA SEGMENTS

- Data segments are allocated contiguously except for the stack

Remember why?



ALLOCATED DATA CHARACTERISTICS

- Allocated data can vary from the following aspects
  - Size
  - Access
  - Scope
  - Location
  - Creation time
  - lifetime

Data allocation is not limited to static allocation at compile time but also dynamic allocation at runtime

ALLOCATED DATA CHARACTERISTICS

- In C, this characteristics are managed by utilizing
  - Variable types
  - Type qualifiers
  - Type modifiers
  - Storage classes
  - Compiler attributes
  - Specialized functions

Data allocation is not limited to static allocation at compile time but also dynamic allocation at runtime

VARIABLE LIFETIME AND SCOPE

- Data memory can exist for different duration in time
  - **Lifetime of function or block**
    - allocated at runtime
    - destroyed when the function returns
    - local scope (only accessible during the function)
    - Memory they occupy is reused by other functions
  - Lifetime of program
  - Longer than a function/block, less than a program

```
int A;
int main () {
        int B = 6;
        int C = 12;
        int *C_p = &C;
        A = C + *C_p;
        return 0;
}
```

VARIABLE LIFETIME AND SCOPE

- Data memory can exist for different duration in time
  - **Lifetime of function or block**
  - **Lifetime of program**
    - Exist for the entire duration of the program
    - Allocated at compile time
    - Never destroyed
    - Occupy memory that can never be reused
    - Global scope (accessible by other parts of the program)
  - Longer than a function/block, less than a program

```
int A;
int main () {
        int B = 6;
        int C = 12;
        int *C_p = &C;
        A = C + *C_p;
        return 0;
}
```

VARIABLE LIFETIME AND SCOPE

- Data memory can exist for different duration in time
  - **Lifetime of function or block**
  - **Lifetime of program**
  - Longer than a function/block, less than a program
    - Dynamic allocation: data allocated at runtime and managed by the programmer

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

DATA ALLOCATION AND SPECIAL KEYWORDS

- Data allocation characteristics can be specified by the way data is declared
- C has special keywords and functions that allow us to control this allocation
- These keywords affect data allocation
  - Variable types
  - Type qualifiers
  - Type modifiers
  - Storage classes

C KEYWORDS

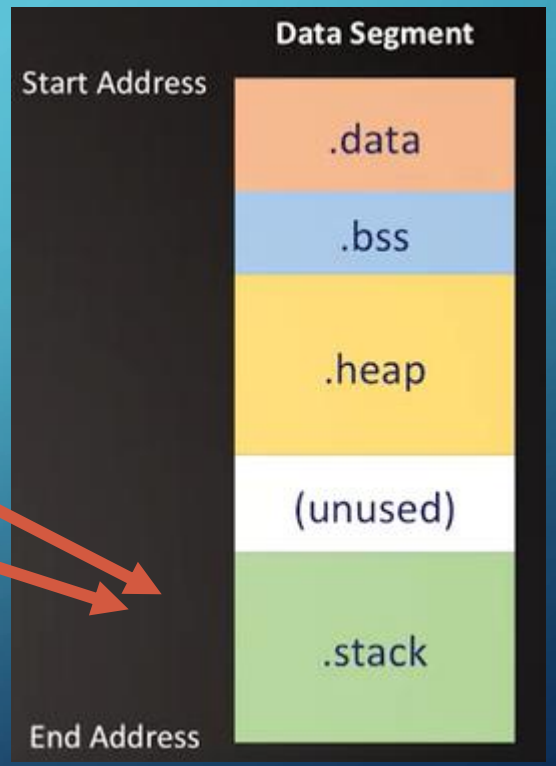
- Variable types directly affect the size of the allocated data (char, int, long, etc.)
- Type qualifiers
  - Const type qualifier allocates data as constant and will be mapped to ROM
- Type modifiers modify the size and sign of the data type (unsigned, signed, short, long, etc.)
- Storage classes specify lifetime and scope of a data type
  - Auto
  - Static
  - Extern
  - Register

```
Auto int A;
Static int A;
Extern int A
Register int A
```

C KEYWORDS

- Auto keyword: automatically allocates/deallocates data on the stack
  - It has a lifetime of a function or block

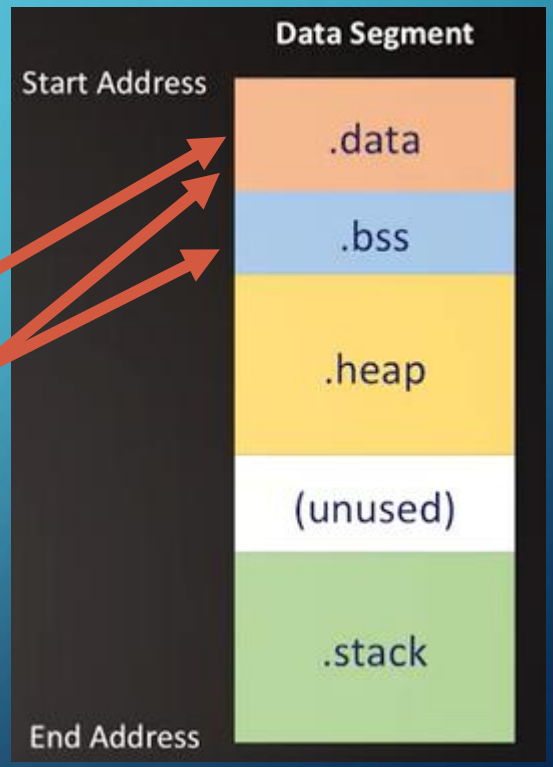
```
void toto() {
        auto int A;
        int B;
        // more code
        return;
}
```



C KEYWORDS

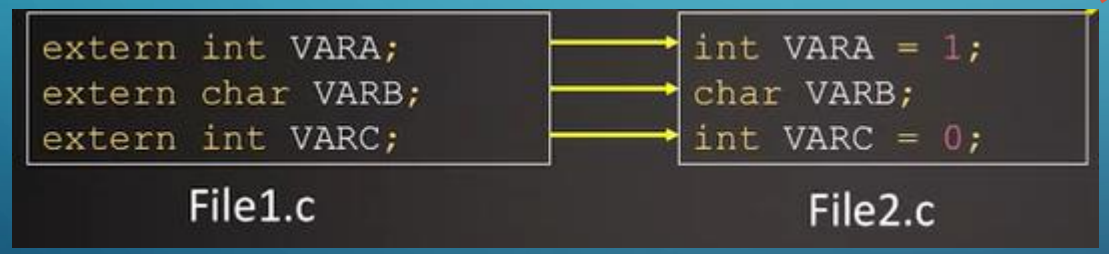
- static keyword: data will persist in memory till the end of the program
- When declared in a function, the access to that data is only available within the function
- It has a lifetime of the program but local scope

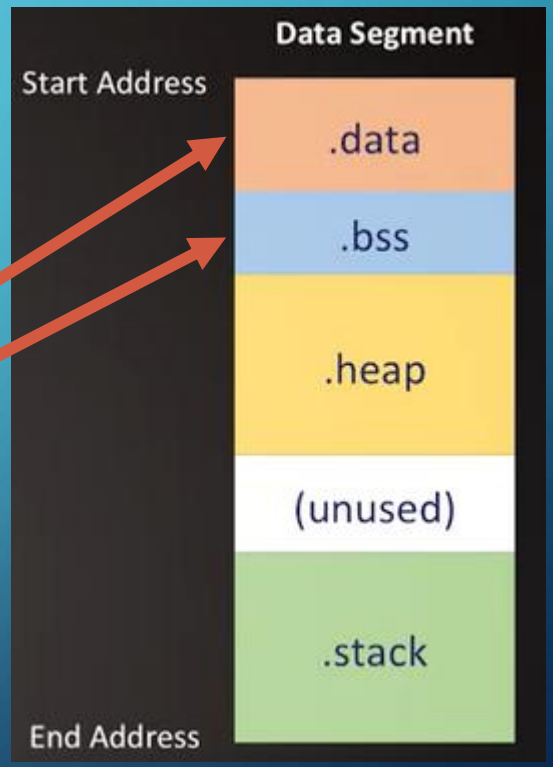
```
static int A = 1;
static char B;
static int C = 0;
```



C KEYWORDS

- extern keyword: declares a global reference defined in another file to be visible by current file
- Can be data or bss
- Initial definition must be a global variable





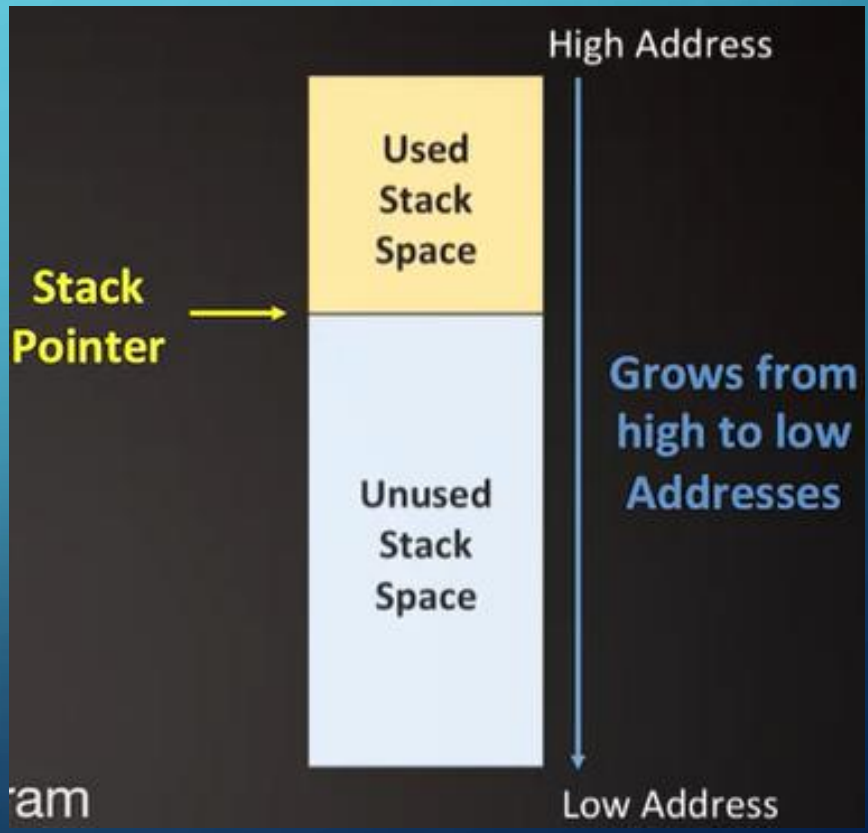
C KEYWORDS

- register keyword requests the compiler to allocate the data in the CPU registers
  - Used for repeated variable usage with high speed
  - Not commonly used
  - Not recommended, specially in ARM architectures cause ARM compilers allocate local variables and function parameters directly in the registers to reduce overhead

THE STACK

- As opposed to the Data and BSS segments that have data allocated at compile time, the Stack region gets **reserved at compile time** but the **data** actually gets **allocated at runtime**.
- The stack is a dynamic are of memory that implements a last-in-first-out queue, like a stack of plates in a cafeteria
- The only valid operations are to add of remove a plate from the top of the stack

- The runtime keeps a pointer, often in a register called **sp (stack pointer)** that indicates the current top of the stack

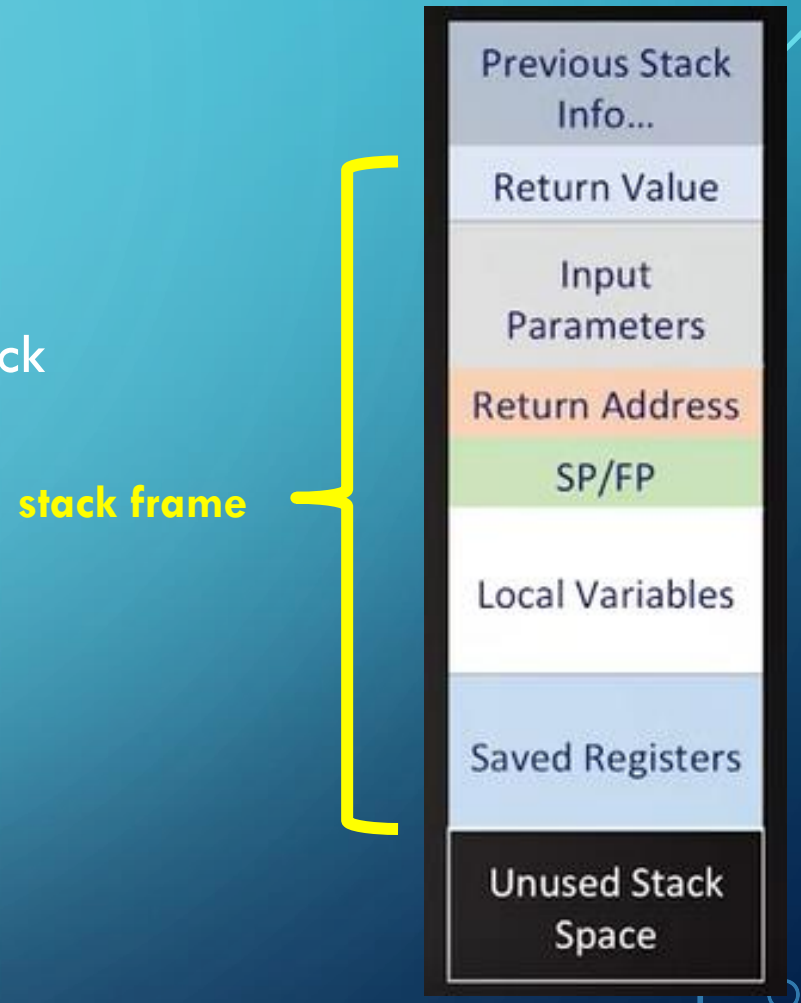


THE STACK

- The stack has three major uses:
  - Provides the storage area for local variables declared inside functions => automatic variables
  - Stores the "housekeeping" information involved when a function call is made. This housekeeping information is known as **"stack frame"** or "procedure activation record". It includes the address from which the function was called (where to jump back to when the called function is finished), any parameters that won't fit into registers and saved values of registers
  - It also works as a scratch-pad area, i.e., everytime the program needs some temporary storage, it uses the stack for this purpose

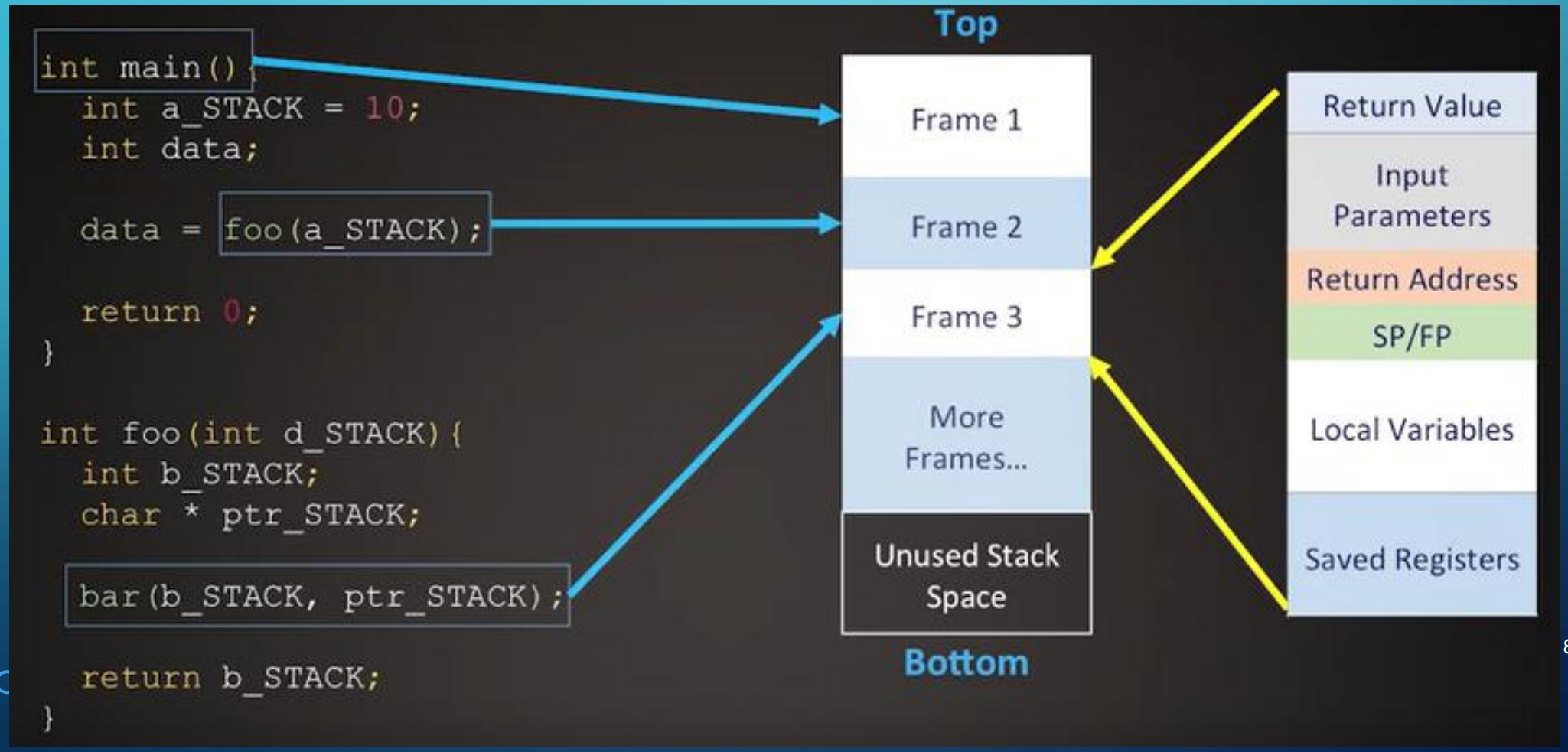
STACK FRAME

- Typical stack implementations store local variables, input parameters and return data on the stack
- In addition, other architecture specific items are put on the stack
  - Copy of used registers
  - Return addresses
  - Previous stack pointers
  - Previous special function register information
  - Information of the state of the CPU before an interrupt is serviced
- The combination of these routine specific data and CPU state information is called stack frame



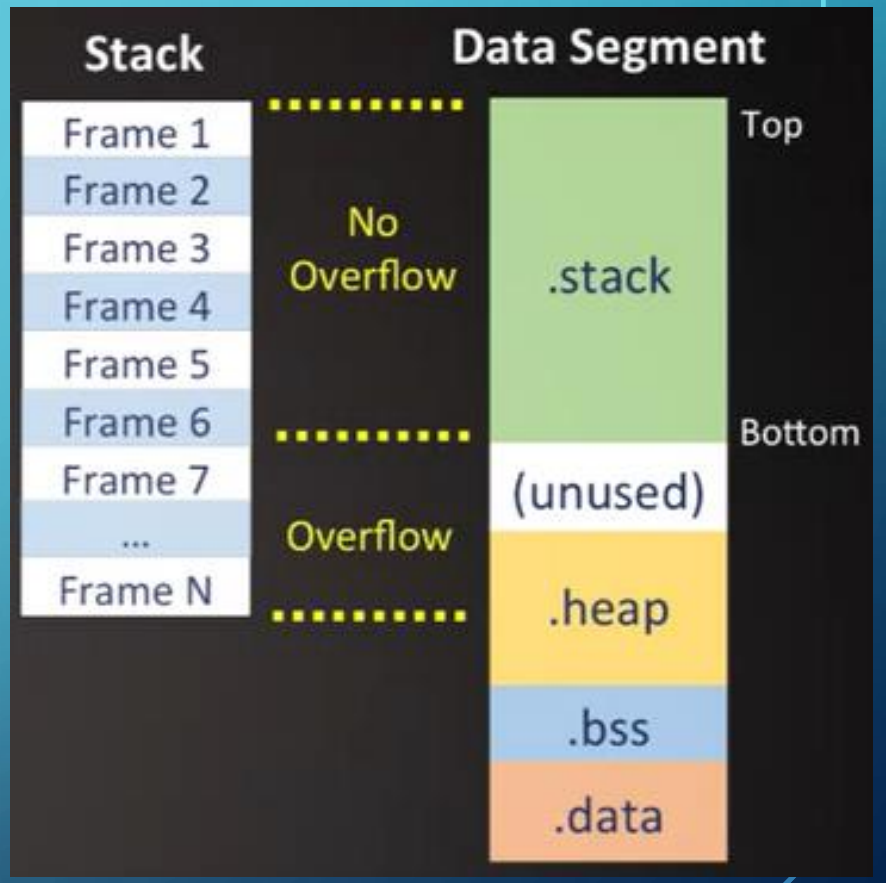
STACK FRAME

- When more and more sub-segments are nested, new frames are added to the stack



STACK OVERFLOW

- Stack is reserved to be a specifi size
- Frame sizes are affected by
  - Number of local variables
  - Number of input parameters
  - Size/type of local variables
  - Size/type of input parameters
  - Size/type of return data
  - Number of nested subroutine calls
  - Interrups/nested interrupts



There is a potential danger in overflowing the stack

ARM REGISTERS AND THE STACK

- We need to understand what an architecture's calling convention is,
- in order to understand how large a stack frame will be given with a function
- Let’s take a look at ARM…
- In arm, general purpose registers are used to store much of this information

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

ARM REGISTERS AND THE STACK

- r0 to r3 are used to hold arguments that are passed into a function
- r4 to r11 are used to store local variables
- r13 holds the current stack pointer of the n location of the used stakck memory
- r14 is the link register. It holds the return address for a function to return to. When a function is done, the return data is put in the r0 register

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

THE STACK - SUMMARY

- The stack is a vital memory segment for software developers.
- Stack is automatically used by the compiler, which in turn utilizes architecture-specific processes and instructions to call and return from a routine.
- All of these operations get compiled into the function call.
- There's still overhead, in order to call and return from a function, and the memory region of the stack occupies part of data memory.
- This is reserved at compile time, allocated at run time, and the operations to interact with this region are introduced at compilation.
- However, the memory itself is reused throughout the program as different functions can get called to allocate and de-allocate data in this region.

THE HEAP

- Dynamic memory
- Like the stack, the heap is reserved in compile time but data is allocated/deallocated at runtime
- The difference is this is done manually by the programmer with the use of functions that manage the region
- The lifetime of heap data will exist for as long as the software designer has not deallocated the data.
- Unlike our data types which specify a size, allocation in the heap can vary with each call

```
void toto(){
    char *ptr_heap;
    ptr_heap = (char*) malloc(8);
    // code
    free ((void*)ptr_heap);
}
```

THE HEAP

- The heap is a space of dynamic memory that is reserved in the data segment.
- The heap is useful because it allows us to dynamically change the size of allocated data.
- Unlike other subsegments, static allocation reserves a set number of bytes. With the heap, we do not have this limitation.
- With each call to reserve memory, we can specify any number of bytes to reserve.
- Additionally, you can resize a previous allocation.
- Much like the stack, allocations and de-allocations in the heap mean the compiler has to add extra instructions into the program for data management in this region.
- However, this is not done automatically, we have to use heap functions to directly reserve and free data in the heap by calling special functions in the software.

HEAP FUNCTIONS

- Heap can be used with the help of four functions
  - void *malloc (size_t size)
  - void *calloc (size_t nitems, size_t item_size)
  - void * realloc (void *ptr, size_t size)
  - void free (void *ptr)

CODE MEMORY

- Just like the data segment, the code segment can be broken down into many different sub-segments and characteristics.
- A third type of memory, register memory, is utilized by a program to run assembly instructions and to interact with the microcontroller.
- A program's executions is a series of instructions that is retrieved from our code memory.
- The CPU registers are utilized in nearly every assembly instruction that gets executed.
- In order for this interaction, the assembly instructions need to know precise details about what's registers and what memory locations each instruction is programmed to use.
- This information is encoded directly into the instructions.

CODE MEMORY

- Stores instructions and some data

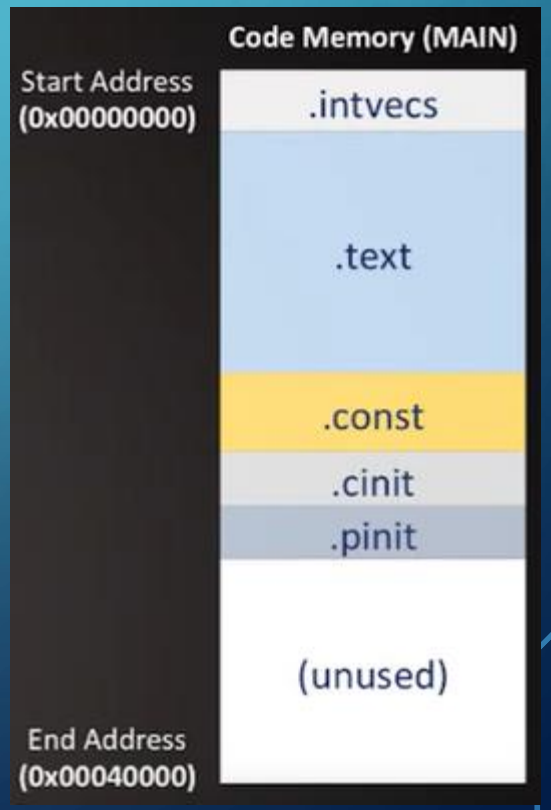
- **Contains multiple sub-segments**

```
MEMORY
{
   MAIN  (RX) : origin = 0x00000000, length = 0x00040000
   DATA  (RW) : origin = 0x20000000, length = 0x00010000
}
```

Physical Memory Regions

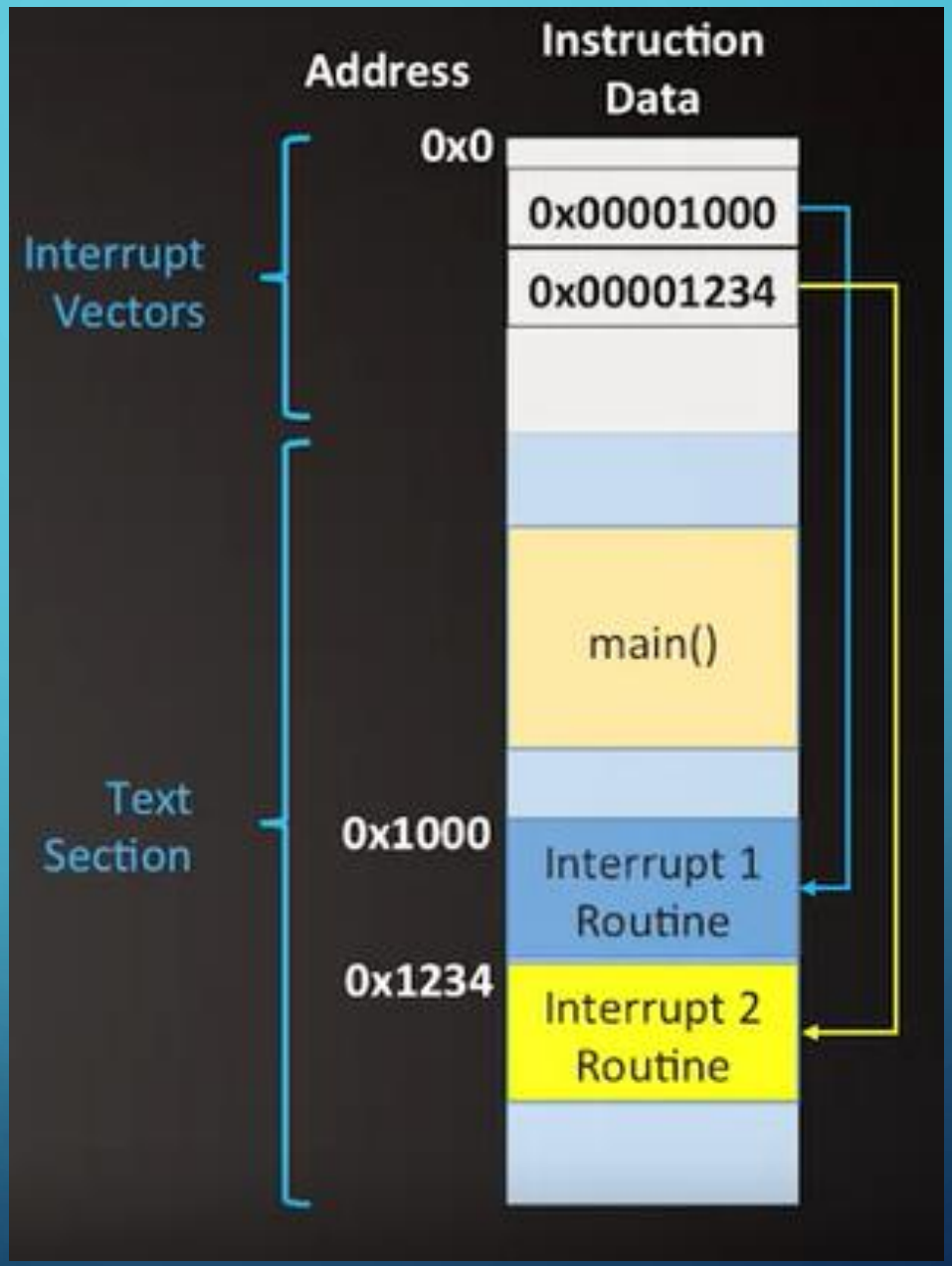
```
SECTIONS
{
   .intvecs :   > 0x00000000
   .text :   > MAIN
   .const :   > MAIN
   .cinit :   > MAIN
   .pinit :   > MAIN
   .data :   > DATA
   .bss :   > DATA
   .heap :   > DATA
   .stack :   > DATA (HIGH)
}
```

Compiled Memory Sections



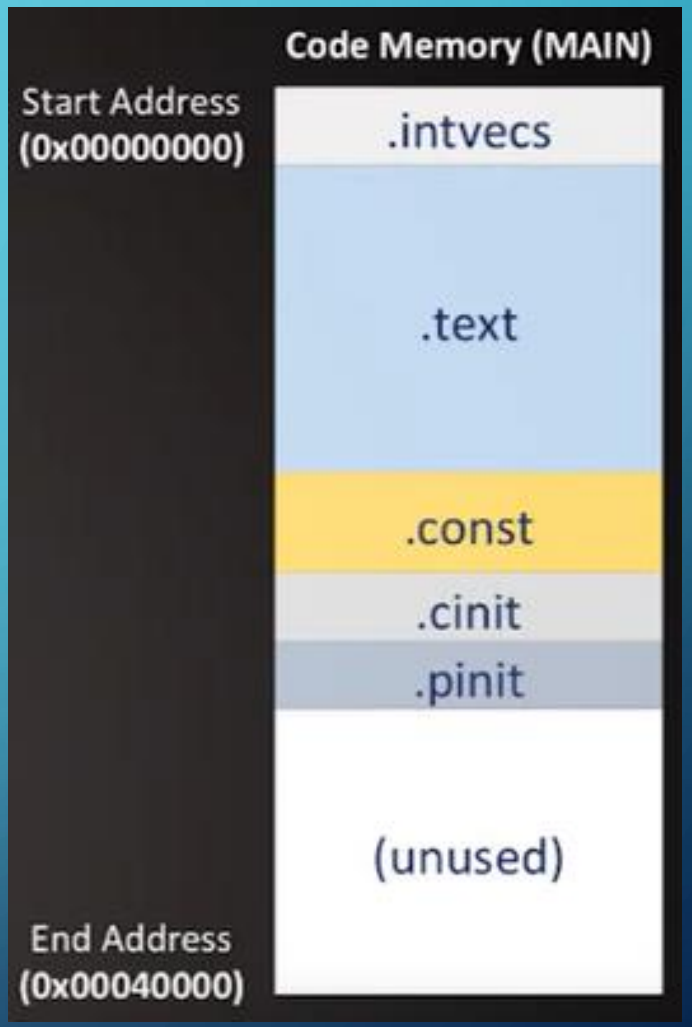
VECTOR TABLE

- Interrup vector table



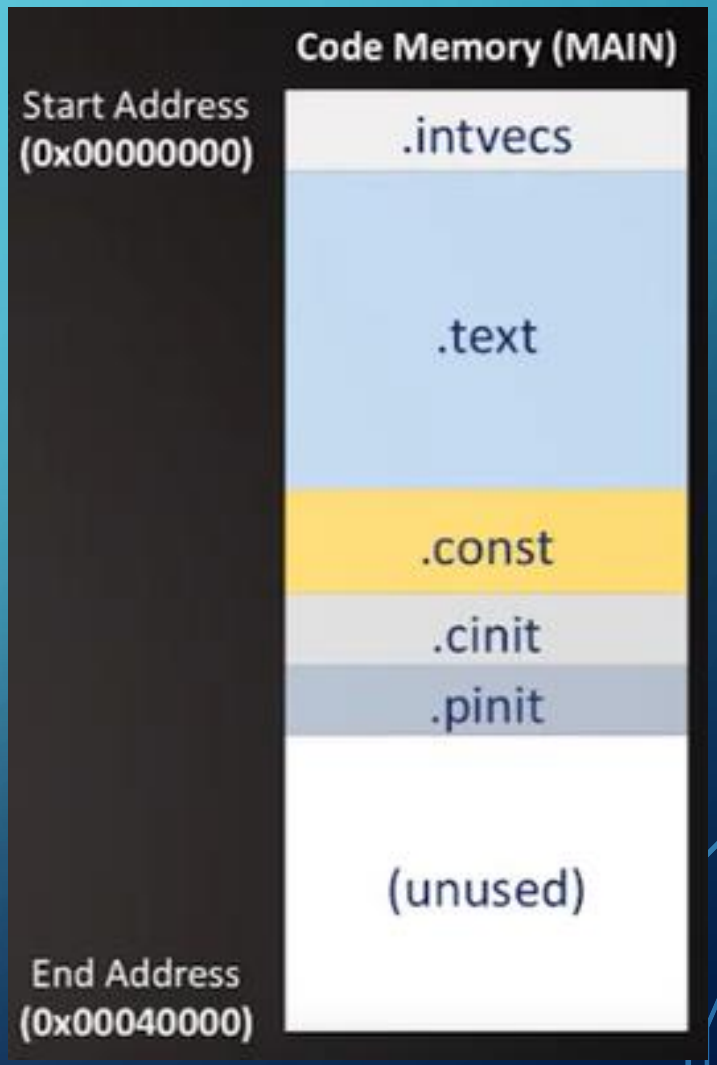
TEXT SEGMENT

- Contains all the written software
  - Main
  - User defined functions
  - Interrupt routines
  - Standard library code
- Size depends on the software implementation



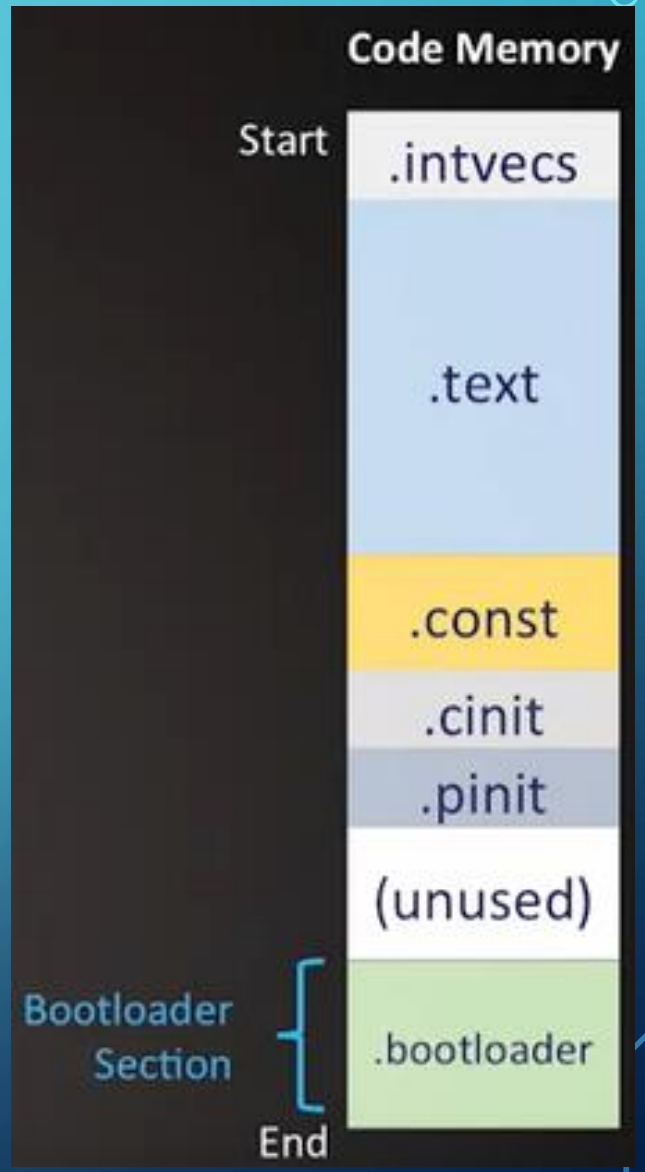
INITIALIZATION SEGMENTS

- Code used to initialize software or data
- Initial values of variables are stored in code memory and loaded into data memory at startup



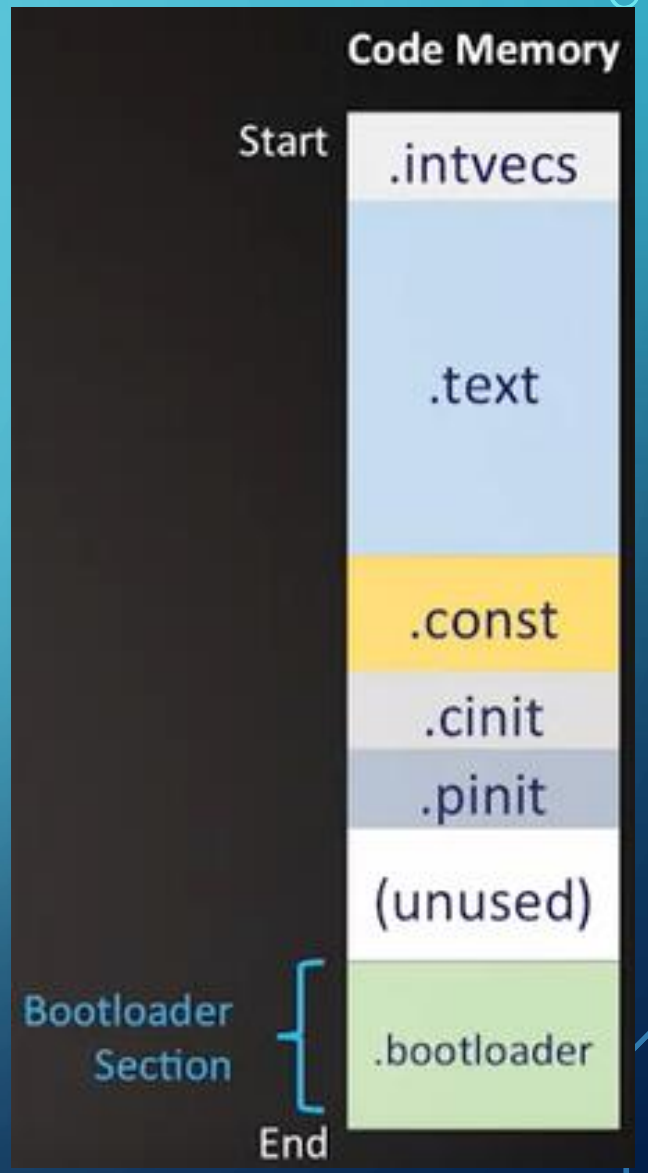
BOOTLOADER

- Embedded systems need a method of installing code
- Using external program loaders that connect directly to a processor are expensive and require extra hardware.
- On board processors are used to program the main processor through a cheaper much more standard interface like USB.
- These extra on-board processors are referred to as external bootloader or flash
- Small block of code that is installed in code memory that is run at startup
- It is used to check for a new install or run existing build



BOOTLOADER

- Alternatively we can reserve a small segment of code memory to act as installer -> bootloader
- Small block of code that is installed in code memory that is run at startup
- It is used to check for a new install or run existing build
- At reboot, the bootloader will search for a signal in one of the communication interfaces signalling a program install
- If the booloader sees this signal, it overwrites the current flash memory with the new program
- Otherwise, it will call the currently loaded program and begin program execution



CONCLUSIONS

- Developing C-code for embedded systems is much like develoing C-code for other types of platforms (desktop, servers, etc.)
- However, embedded systems engineers must be aware of the following aspects when developing a C-program for embedded systems
  - Resources, specially memory, are scares.
  - Compilation/assembly/linking is platform-dependent
  - Memory resources are different from an embedded platform to the other
- Embedded software engineers must have a precise understanding on how the memory (data and program) is organized, physically and logically

CONCLUSIONS

- In this class we learned the following topics
  - Embedded software build process
  - The use of C in the development of embedded systems
  - Memory in embedded systems from a C language standpoint

- These topics are quite important for the correct implementation of embedded software in embedded platforms.