Code Segment

Embedded Software Essentials

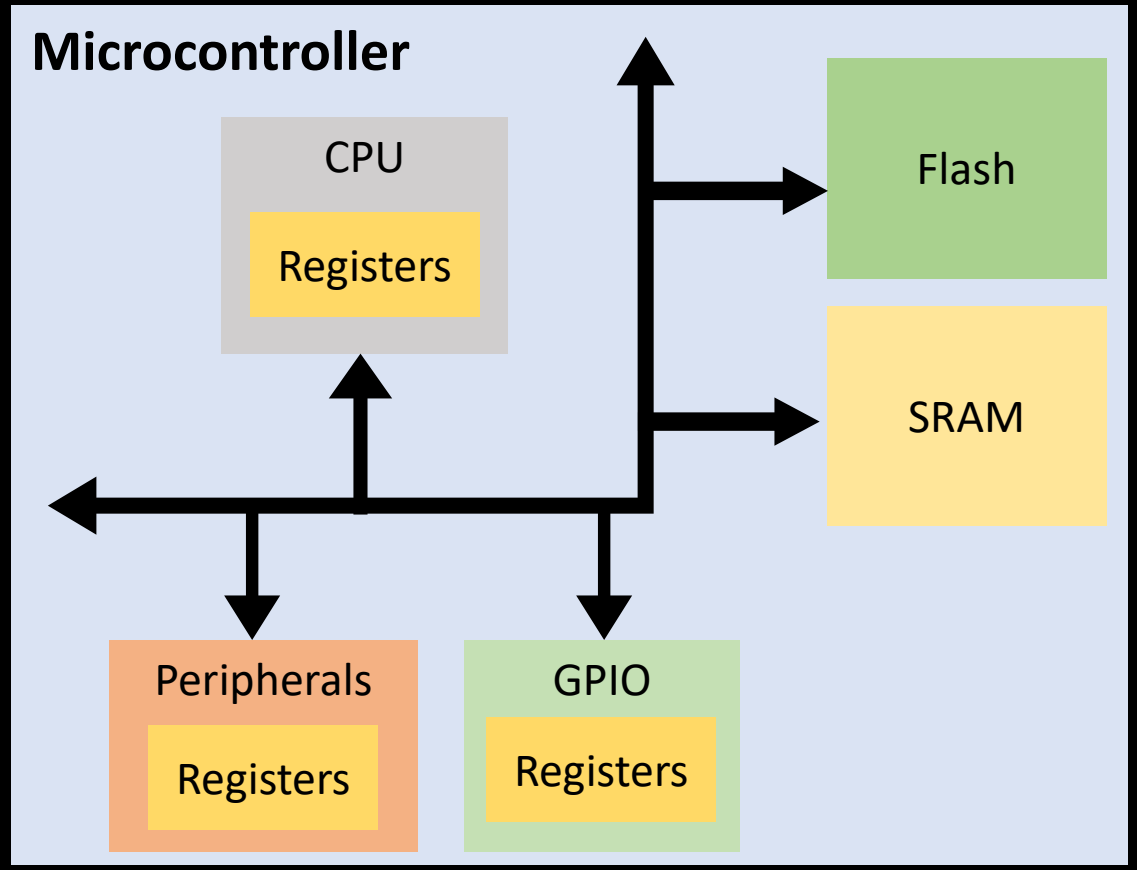
C1M3V8

Memory

- Three Main Types of Memory
  - Flash (non-volatile)
  - RAM (volatile)
  - Registers (volatile)

- Instructions are fetched from code memory, decoded and then executed in the Microcontroller CPU

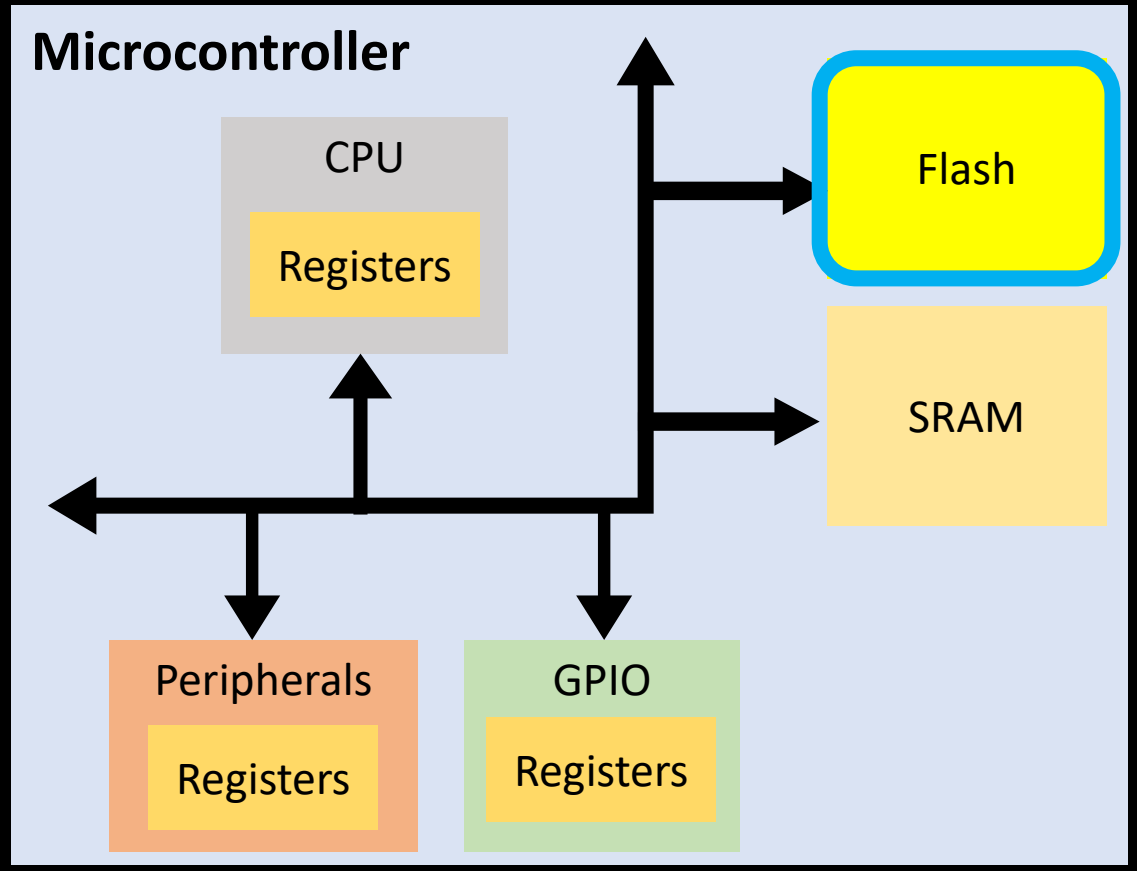
- Instructions utilize CPU registers



→ **Compiler manages Instruction register assignment**

Code Memory

- Runtime Read-Only Non-Volatile Memory (Flash)
- Stores instructions and some data
- Write/Erase requires extra credentials
  - Security and protection from Overwriting
- Latency and Durability Issues



Example Linker Script Contents

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

Physical Memory Regions

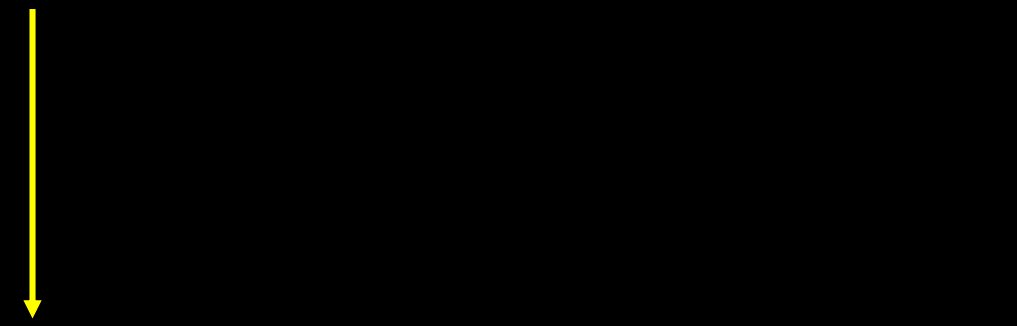
```
SECTIONS
{
    .intvecs :   > 0x00000000
    .text :   > MAIN
    .const :   > MAIN
    .cinit :   > MAIN
    .pinit :   > MAIN
    .data :   > DATA
    .bss :   > DATA
    .heap :   > DATA
    .stack :   > DATA (HIGH)
}
```

Compiled Memory Sections

Example Linker Script Contents

Code Segment maps to Flash memory

- **Contains multiple sub-segments**



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

Compiled Memory Sections

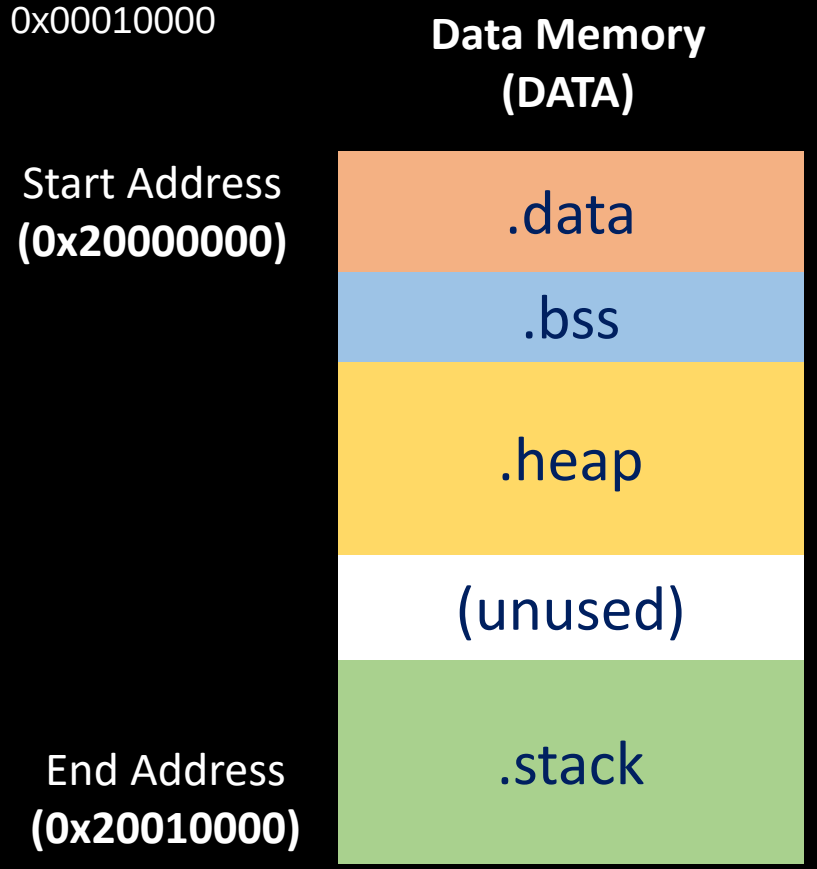
```
MEMORY
{
   MAIN  (RX) : origin = 0x00000000, length = 0x00040000
   DATA  (RW) : origin = 0x20000000, length = 0x00010000
}
```

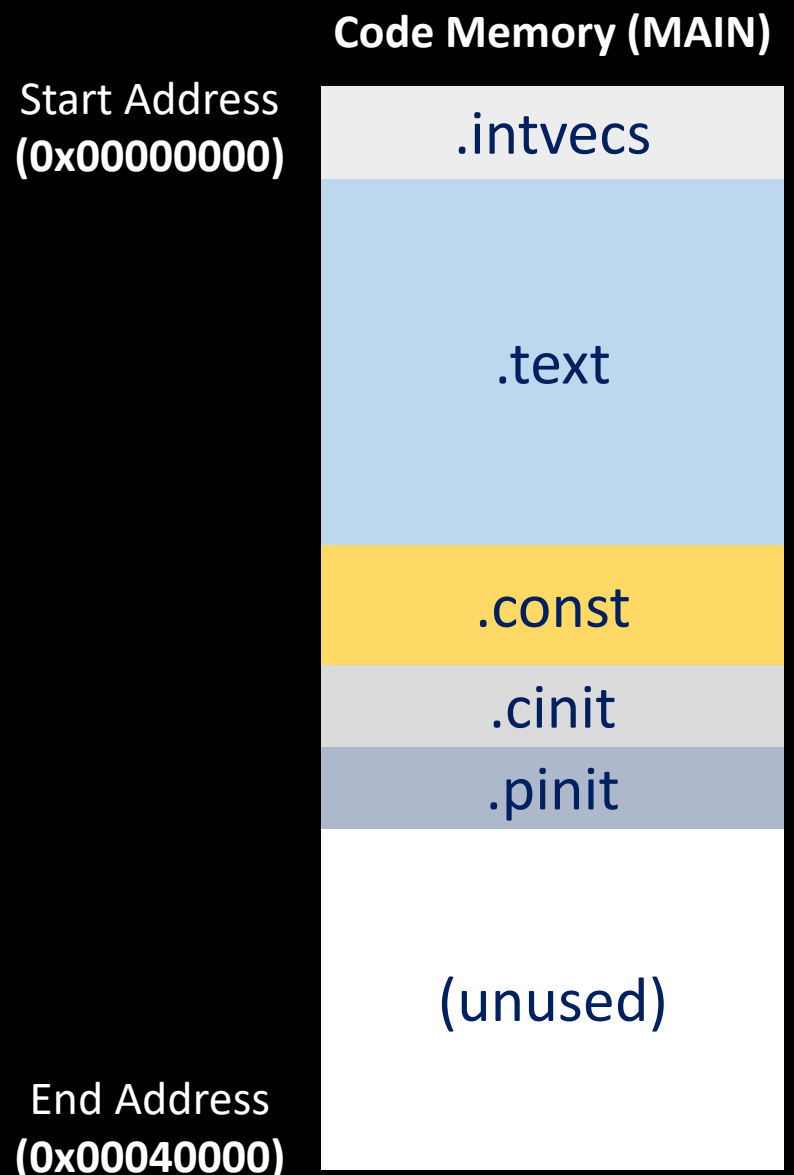
Physical Memory Regions

Memory Segments

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

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



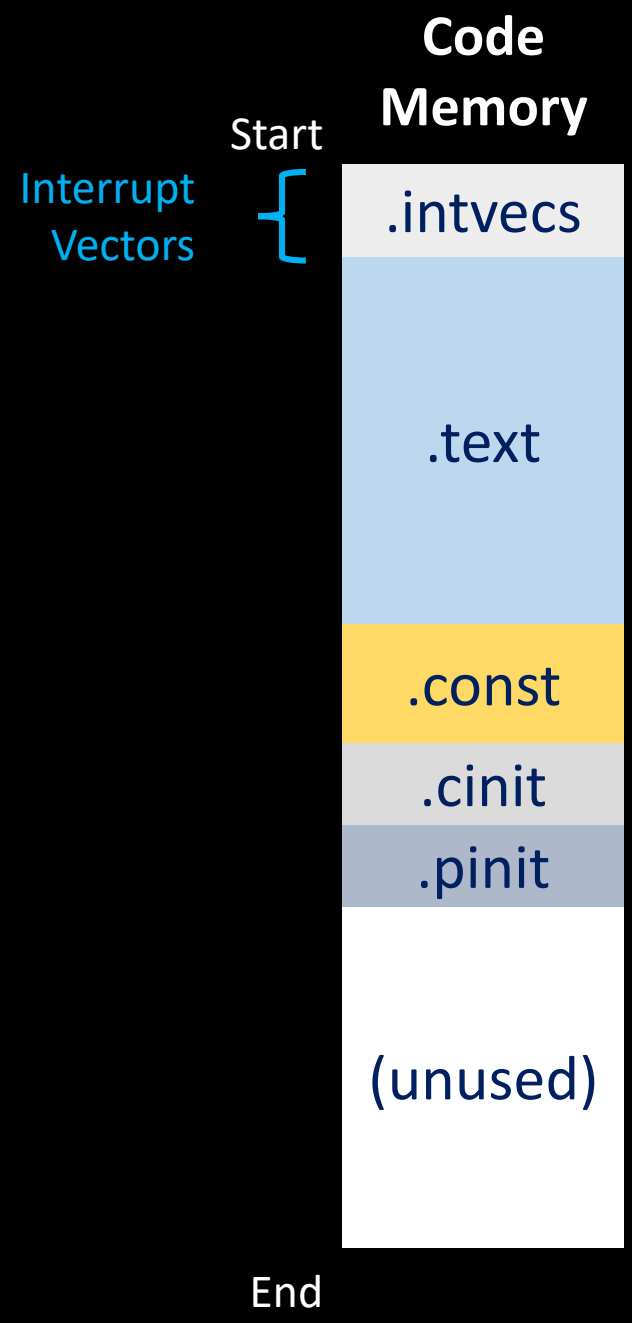


Vector Table

- **Interrupt**: Asynchronous events with an associated software routine

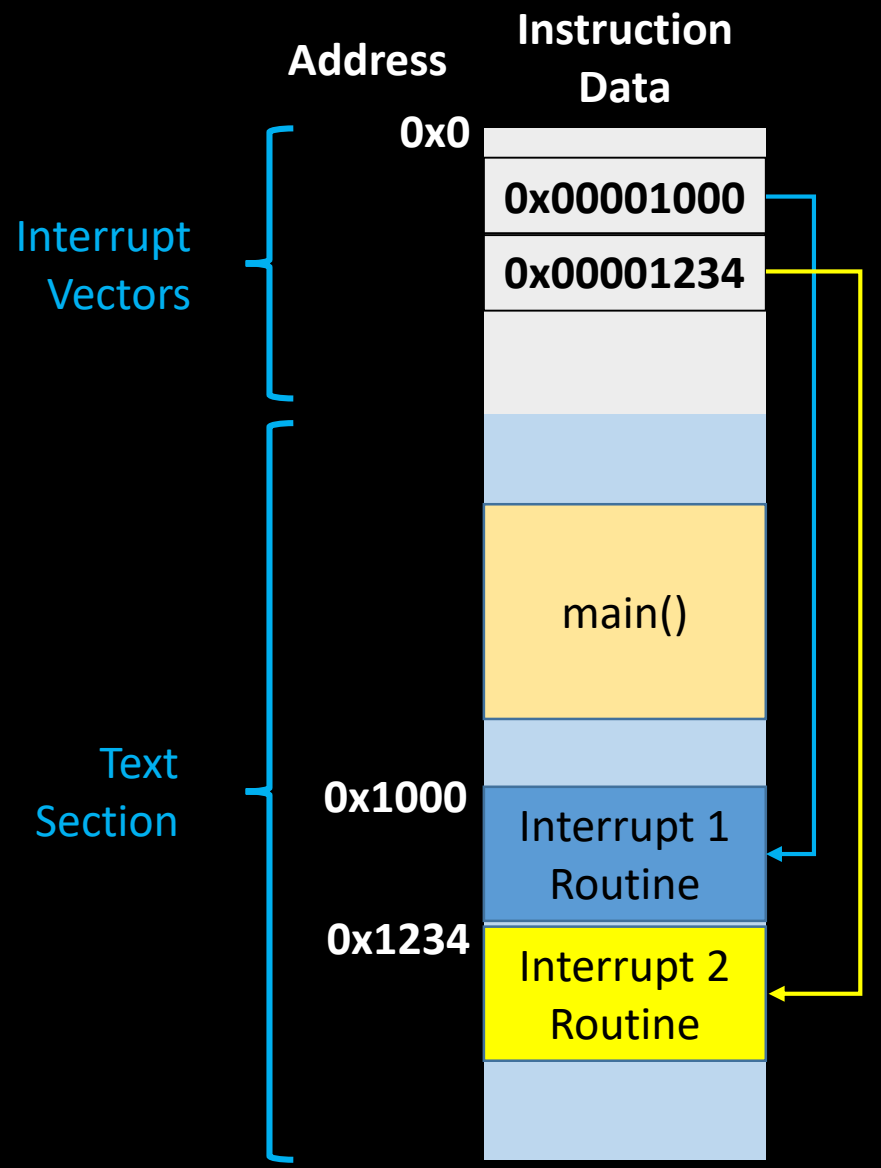
- Vector Tables contains an platform defined list of function address
  - Used to "jump" into a routine

- Typically first part of code memory (address zero)



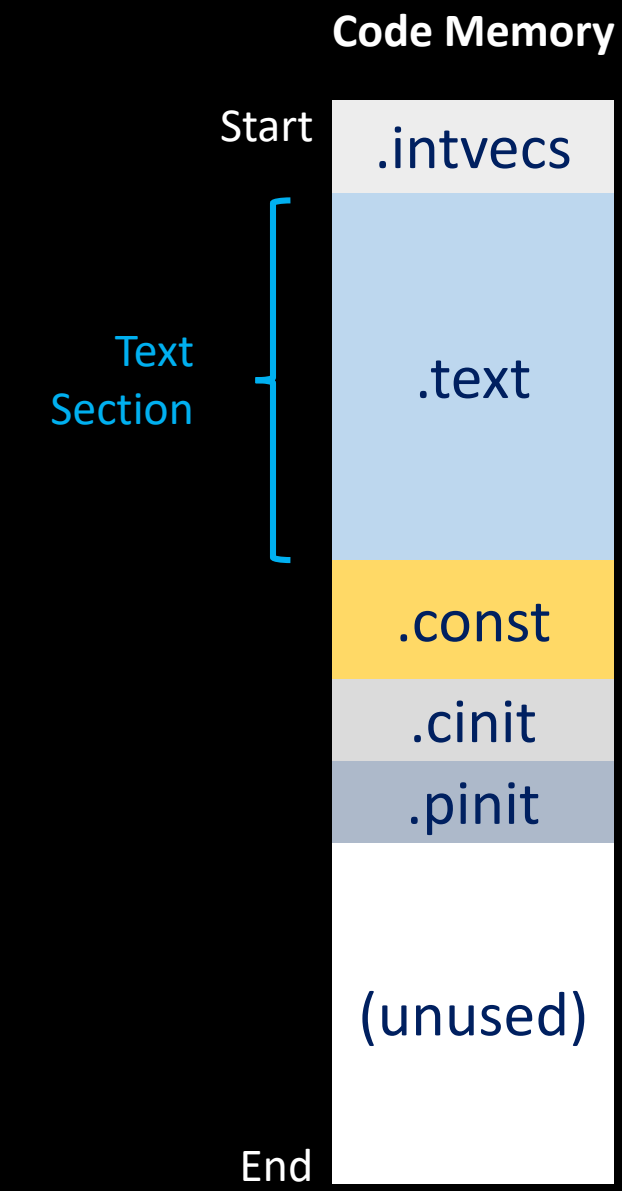
Vector Table

- **Interrupt**: Asynchronous events with an associated software routine
- Vector Tables contains an platform defined list of function address
  - Used to "jump" into a routine
- Typically first part of code memory (address zero)



Text Segment

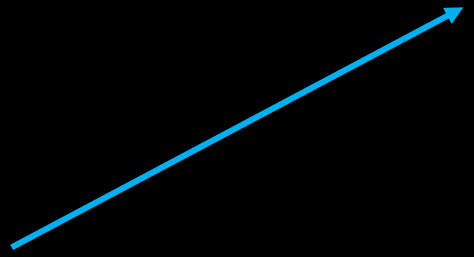
- Text Section contains all of your written software
  - Main
  - User defined Functions
  - Interrupt Routines
  - Standard Library Code
- Usually the largest segment in code
  - Size Depends on software implementation

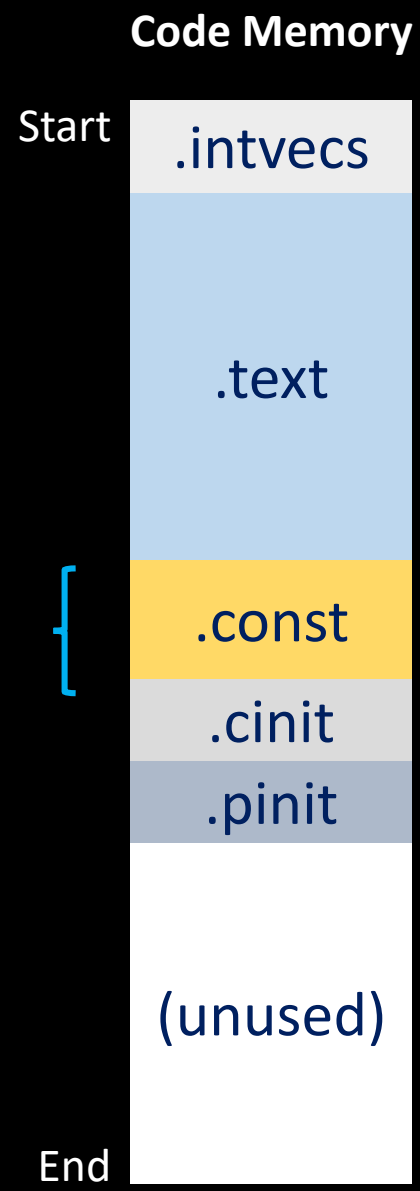


Read-Only/Const Segment

- Read-Only (RO) or Const contains constant variable defined data
  - Very hard to overwrite at runtime
- Usually the largest segment in code
  - Size Depends on software implementation

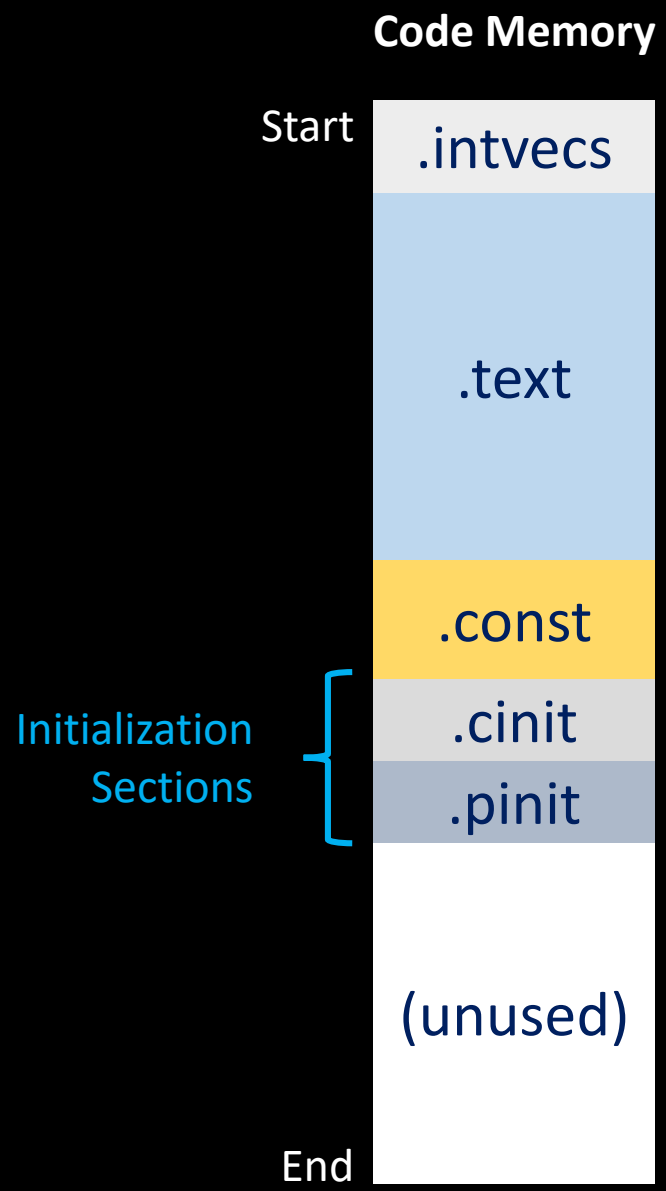
```
const char VARA = 'a';
const int VARB = 1;
```





Initialization Segments

- Code used to initialize software or data
  - Either startup or object instantiation
- Initial values are stored in Code Memory
  - Non-Zero Initialized Global & Static data (.data) and variable initial values
- Routines to set these initial values
  - .data – Initilalized at startup from code memory
  - .bss – Initialized to zero at startup

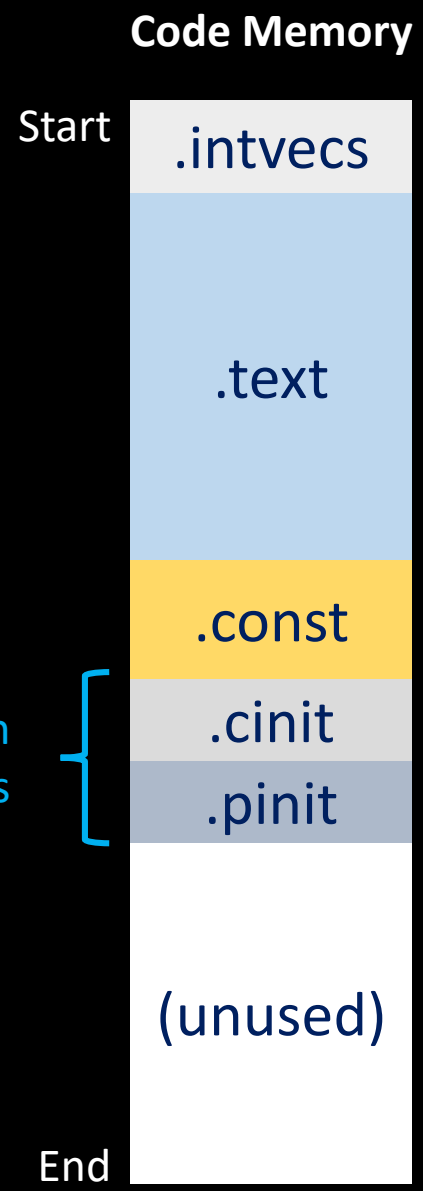


Initialization Segments

- Code used to initialize software or data
  - Either startup or object instantiation

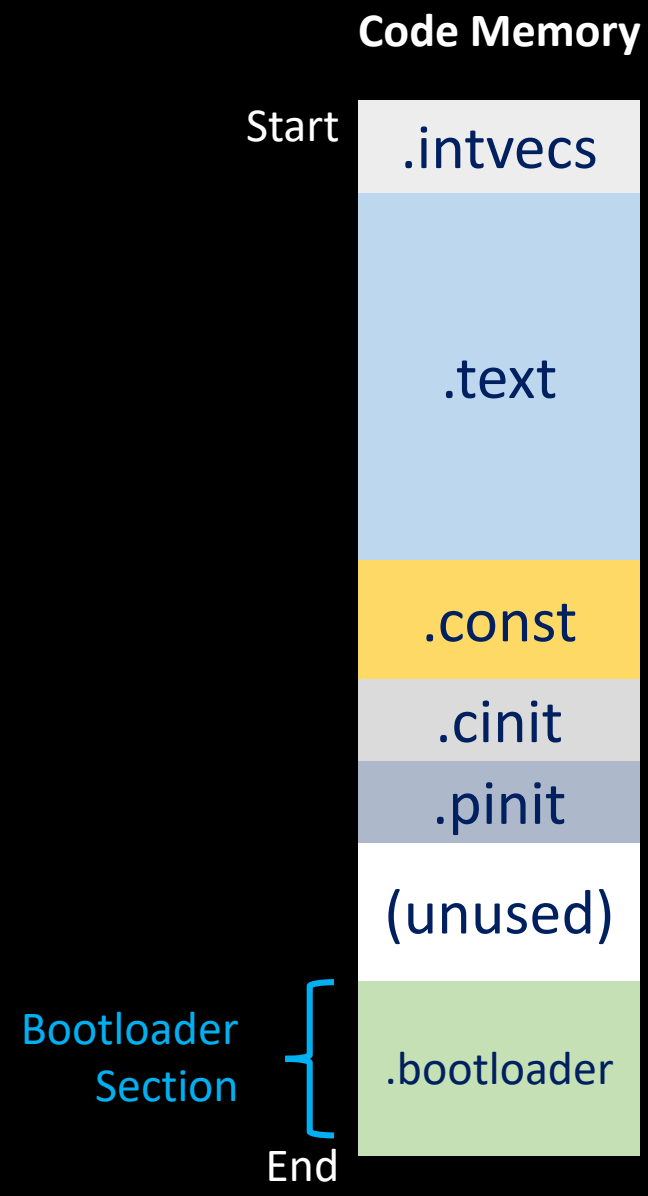
- Names depend on compiler, architecture or c-standard
  - .cinit
  - .pinit
  - .init/.fini
  - .init_array
  - ctors/.dtors

Mixture of Initialization functions and initialization Data



Bootloader

- Small block of code that is installed in code memory that is run at startup
- Allows you to check for a new install or run existing build
- Helps with reducing hardware costs



Bootloader

- On-chip software to install build using platform interfaces
  - Put in a safe/protected space so you do not overwrite it!
- Create a Linker Script section for the bootloader code
- Exclude writing boot section for every install

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

```
MEMORY
{
   MAIN  (RX) : origin = 0x00000000, length = 0x0003FC00
   BOOT (RX) : origin = 0x0003FC00, length = 0x00000400
   DATA  (RW) : origin = 0x20000000, length = 0x00010000
}
```

Standard Libraries

- Linker by defult links with c-standard libraries specific for you architecture
- Can remove the dependency on using these libraries (ex. --nostdlib)
- Requires to implement all libraries and supporting software yourself