Fun with 'Embedded' C

What to Expect?

- What is in Embedded?
- De-jargonified Pointers
- Hardware Programming
- Compiler Optimizations
- Register Programming Techniques
- Playful Bit Operations

What is in Embedded?

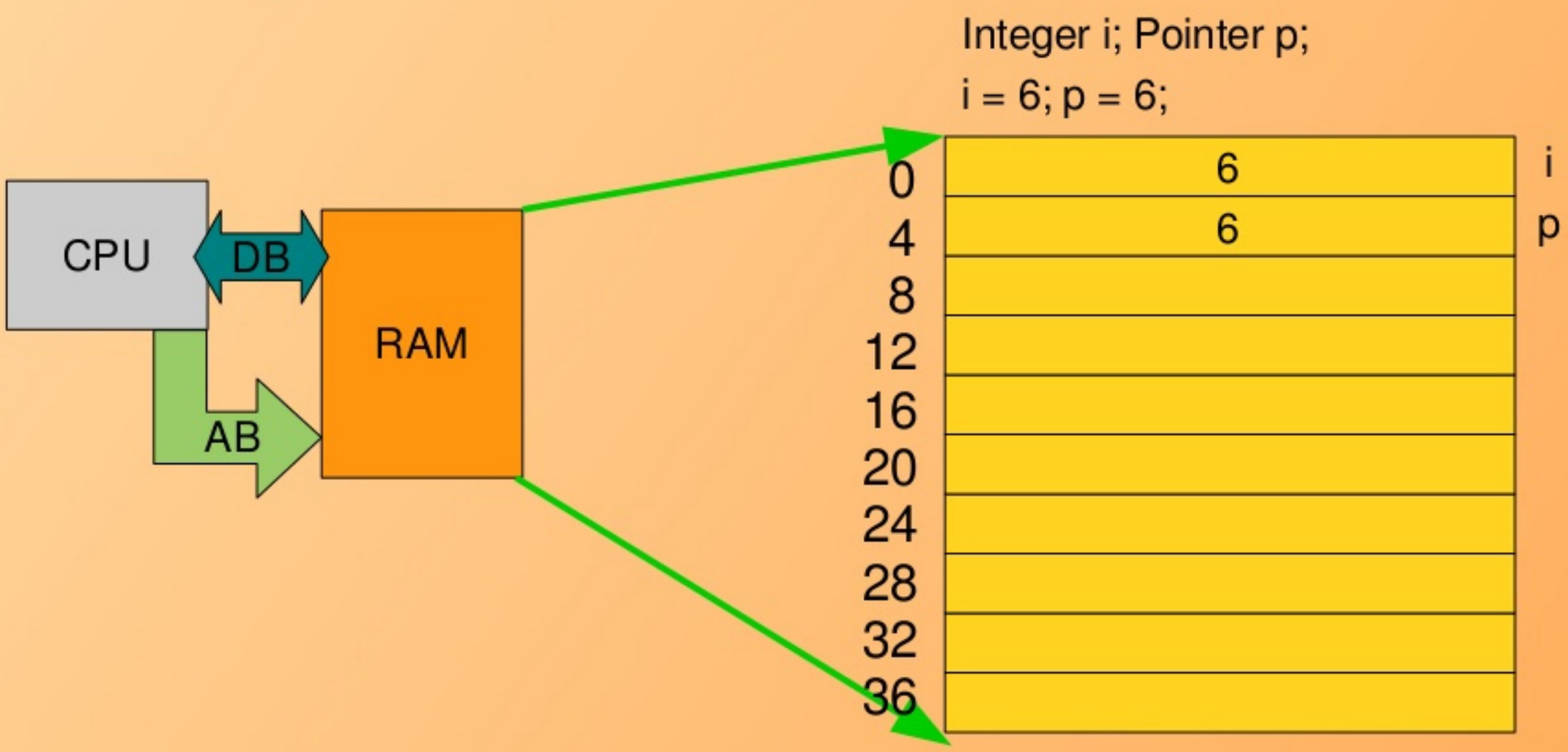
- Typically for a cross architecture
  - Needs cross compilation
  - Needing architecture specific options like -mcpu
- No-frills Programming
  - Typically no library code usage
  - No init setup code
- Have a specific / custom memory map
  - Needs specific code placement
- Programming a Bare Metal
  - No code support framework like stack, ...
  - No execution support framework like loader

Pointers: De-jargonification

through 7 rules

Rule #0: Foundation

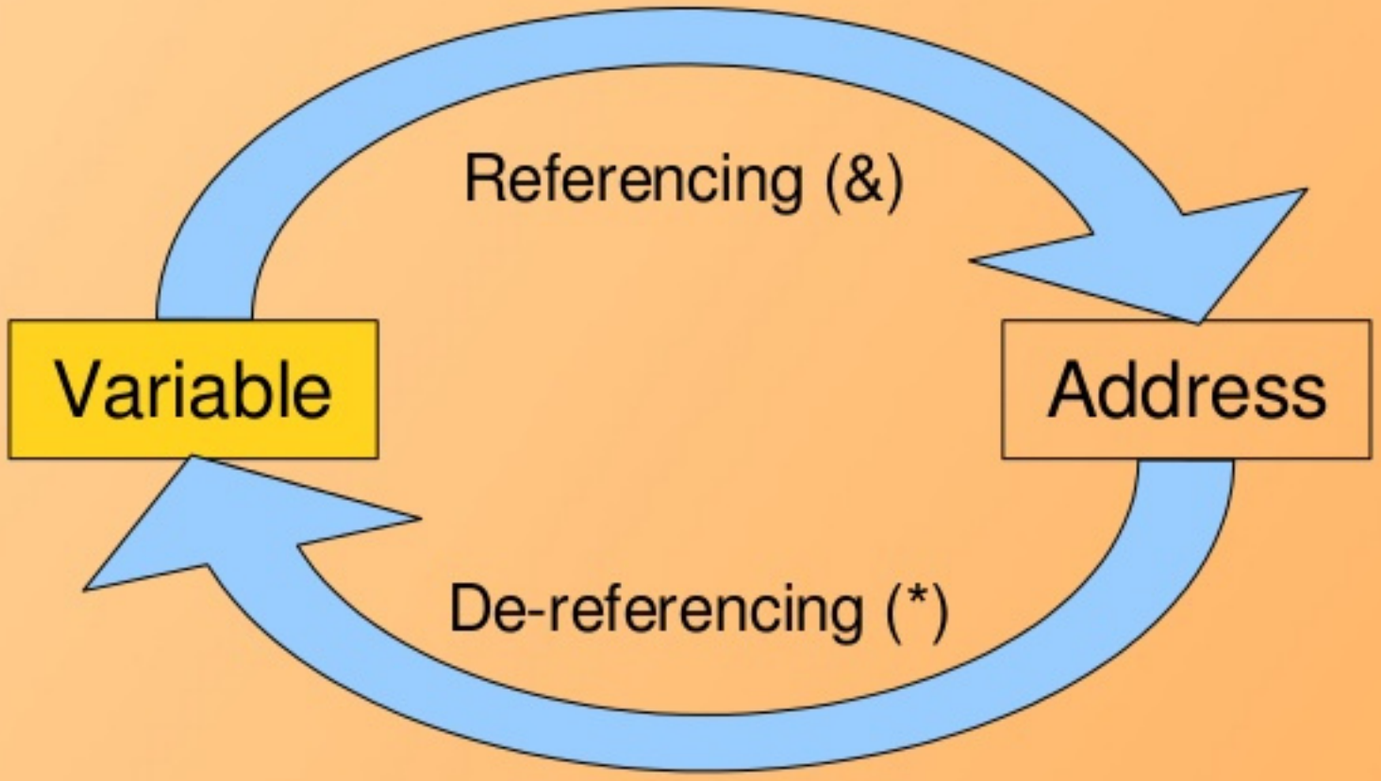
- Memory Locations & its Address



Rule #1: Pointer as an Integer

- "Pointer is an Integer"
- Exceptions:
  - May not be of same size
  - Rule #2

Rule #2: Pointer not an Integer



Rule #3: Pointer Type

- Why do we need types attached to pointers?
  - Only for 'dereferencing'
- “Pointer of type t = t Pointer = (t *)”
  - It is a variable
    - Which contains an address
    - Which when dereferenced returns a variable of type t
    - Starting from that address
- Defining a Pointer, indirectly

Rule #4: Pointer Value

- “Pointer pointing to a Variable = Pointer contains the Address of the Variable”
- “Pointing means Containing Address”

Rule #5: NULL Pointer

- Need for Pointing to 'Nothing'
- Evolution of NULL, typically 0
- “Pointer value of NULL = Null Addr = Null Pointer = Pointing to Nothing”

Array Interpretations

- Original Big Variable
  - Consisting of Smaller Variables
  - Of Same Type
  - Placed consecutively
- Constant Pointer to the 1st Small Variable
  - In the Big Variable

Rule #6: Array vs Pointer

- arr + i = &arr[i]
- Value(arr + i) = Value(arr) + i * sizeof(*arr)
- “Value(p + i) = Value(p) + i * sizeof(*p)”
- Corollaries:
  - p + i = &p[i]
  - *(p + i) = p[i]
  - sizeof(void) = 1

Rule #7: Allocation Types

- "Static Allocation vs Dynamic Allocation"
  - Named vs Unnamed Allocation
  - Managed by Compiler vs User
  - Done internally by Compiler vs Using malloc/free
- Dynamic corresponding of a 1-D Static Array
  - Can be treated same once allocated
  - Except their sizes

2-D Arrays

- Each Dimension could be
  - Static, or
  - Dynamic
- Various Forms for 2-D Arrays (2x2 = 4)
  - Both Static (Rectangular) - arr[r][c]
  - First Static, Second Dynamic - *arr[r]
  - First Dynamic, Second Static - (*arr)[c]
  - Both Dynamic - **arr
- 2-D Arrays using a Single Level Pointer

Hardware Programming

Compiler Optimizations

- Using -O0, -O1, -O2, -O3, -Os, -Ofast, -Og
- May eliminate seemingly redundant code
  - But important from embedded C perspective
  - Examples
    - Seemingly meaningless reads/writes
    - NOP loop for delay
    - Functions not called from C code
- Ways to avoid
  - Use -O0 or no optimization
  - Use volatile for hardware mapped variables
  - Use __attribute__((optimize("O0"))) for specific functions
  - Use asmlinkage for functions called from assembly

Register Programming Techniques

- Direct using the (Bus) Address
- Indirect through some Direct Register
- Multiplexed using some Config Registers / Bits
  - Example: UART Registers, ...
- Clear On Set
  - Example: Status Registers, ...
- Protected Access using Lock / Unlock Registers
  - Example: MAC Id Registers, ...

Bit Operations

- Using the C operators &, |, ^, ~, <<, >>
- Assignment equivalents of those
- Clearing using &, ~
- Setting using |
- Toggling using ^
- Shifting, Multiplication using <<
- Shifting, Division using >>

What all have we learnt?

- Specifics of Embedded C
  - Architecture Specifics, Linker Scripts, Bare Metal
- Pointers Simplified
  - 7 Rules, Arrays
- Hardware Programming
  - Compiler Optimizations
  - Register Programming Techniques
  - Playful Bit Operations

Any Queries?