

02 From Text to Machine

- **The Toolchain:** Source (.c) -> Compiler -> Object -> Linker -> Executable
- **Data Types:** What kind of data do we store in C?
- **Endianness:** How do we store multi-byte Data?
- **Imperative Basics:** Our first C program:
 #include, main(), variables
 while(), if-else, for ().
- **Intro to pointers:** A pointer is a variable. It stores a target address.
 - volatile pointers – used for hardware access or seeing internal CPU operations.
 - Accessing Memory-Mapped I/O through a pointer.
- **DRY and function calls:** Don't Repeat Yourself, refactor!
- **Operators & Expressions:** the Grammar of C
 1. The "Scratchpad" (Pure Calculations)
 2. The "Hardware Switches" (Bitwise Operations)
 3. The "Save Button" (Memory Modification)
 4. Operator Precedence for reference.
- **Principle of programming:** MVC (Model-View-Controller)

02 The Tool Chain

Program Execution Models

- **Interpreted:** An interpreter reads and executes source text line-by-line at runtime.
 - *Examples:* Python, JavaScript, BASIC.
- **Compiled:** A build chain processes source text into native CPU instructions before execution.
 - *Examples:* C, C++, Rust, Go, Swift.

The C Build Chain

- **Compilation:** The **Compiler** converts source code into a **relocatable object**.
 - Contains native CPU instructions.
 - Lacks absolute memory addresses (cannot run yet).
 - Each source file produces its own object file.
- **Linking:** The **Linker** chains your objects (and external libraries) together.
 - Maps code and data to specific memory addresses.
 - Outputs the final **executable** file.

02 The Tool Chain

An example from a real run (we will see this program in action soon):

black text = actual command blue text = comment

sdcc (Small Device C Compiler) is our *compiler* and *linker*. The **-something** or **--something** are called *flags*. It gives special instructions to *compiler* and *linker*. (No need to worry about them in this class).

```
sdcc -mz80 --std c99 --Werror -c blinking-led.c -o blinking-led.rel
```

“**sdcc**” is our *compiler*. We compile from **source (.c)** to **relocatable object (.rel)**

-mz80 = we will compile for Z80 **--std c99** = We'll use C99 standard

```
sdcc -mz80 --std c99 --Werror -c nmi_handler.c -o nmi_handler.rel
```

I compile another **.c** file. This will become your library.

```
[100] STARTUP_OBJ: startup.rel    ← I have another library here to start up the Z80.
```

```
[100] MEMMAP:      memmap.ld      ← This is the memory map information file.
```

```
sdcc -mz80 --no-std-crt0 startup.rel nmi_handler.rel blinking-led.rel
```

```
-Wl-u -Wl-f,memmap.ld -o run.ihx
```

Link by chaining (low to high address): **startup.rel nmi_handler.rel blinking-led.rel**

using **memmap.ld** for address map. Produce **run.ihx** as “*executable*.”

```
sdobjcopy -I ihex -O binary run.ihx run.bin
```

(Trivial) The ROM in SimulIDE needs **.bin** file, so we convert **.ihx** to **.bin**

memmap.ld

```
-b _CODE = 0x0100
-b _DATA = 0x4800
-g _STACK_TOP = 0xBFFF
```

explain!

02 The Tool Chain – memory map

Recall our memory map for our hardware:

Address	What's there?	
0x0000	RESET jumps here (Z80 magic)	} startup.rel sits here
0x0066	NMI_ jumps here (Z80 magic)	
(1) 0x0100	program starts here	} nmi-handler.rel blinking-led.rel sit here
0x3FFF	last address for program	
0x4000 ... 0x40FF	Your (student's) safe memory zone. Free to write to and read from these addresses without danger	
0x4100 ... 0x47FF	Instructor memory zone. Danger zone! You may corrupt the whole system writing to these addresses.	
(2) 0x4800 ...	program data space. contains all data types compiler produce automatically	} any memory for data that any .rel needs will be read and written to here.
(3) 0xBFFF	top of the stack (more later). it grows upwards.	
0xC000 ... 0xFFFF	I/O zone. this is not RAM. we use these	

memmap.ld

```

-b _CODE = 0x0100 (1)
-b _DATA = 0x4800 (2)
-g _STACK_TOP = 0xBFFF (3)

```

02 The Tool Chain – memory map

Zooming in the _DATA area:

Address	What's there?
0x0000	RESET jumps here (Z80 magic)
0x0066	NMI_ jumps here (Z80 magic)
0x0100 0x3FFF	program starts here last address for program
0x4000 ... 0x40FF	Your (student's) safe memory zone. Free to write to and read from these addresses without danger
0x4100 ... 0x47FF	Instructor memory zone. Danger zone! You may corrupt the whole system writing to these addresses.
0x4800 ...	program data space. contains all data types compiler produce automatically
0xBFFF	top of the stack (more later). it grows upwards.
0xC000 ... 0xFFFF	I/O zone. this is not RAM. we use these

Address	What's there?
0x4800	constant GLOBALS start first.
	uninitialized GLOBALS
	HEAP grows downward (address increasing) (later)
	↓
	(still free) memory
	↑
0xBFFF	STACK grows upward (address decreasing) (later)

What kind of data do we store in C?

type	meaning	sizeof (.) - value in number of bytes for C99			
		Z80	intel Core 64	Apple ARM64	RV64
_Bool	Boolean: 1 or 0	1	1	1	1
unsigned char	A character, which can also be treated as unsigned number from [0, 255]	1	1	1	1
signed char	A character, which can also be treated as a 2's complement number from [-128, 127]	1	1	1	1
unsigned int	An unsigned integer. Range depends on size. range is $[0, 2^{\text{bytes} \cdot 8 - 1}]$ for Z80, it is [0, 65535] or [0x0000, 0xFFFF]	2	4	4	4
int can also be written as signed int	A 2's complement signed integer. Range depends on size. range is $[-2^{\text{bytes} \cdot 8 - 1}, 2^{\text{bytes} \cdot 8 - 1} - 1]$. For Z80 it is [-32768, 32767] or [0x8000, 0x7FFF]	2	4	4	4
void *	a pointer to an address on the address bus. For Z80, address range is [0x0000, 0xFFFF] which fits in 2 bytes – much more later	2	8	8	8
double	Scientific number like $a \cdot 10^n$ like $-1.602 \cdot 10^{-19}$ Not supported on Z80	N/A	8	8	8

Problem with C “standard”

app

comp arch

logic

type	meaning	sizeof (.) - value in number of bytes for C99			
		Z80	intel Core 64	Apple ARM64	RV64
ugly _Bool	Boolean: 1 or 0	1	1	1	1
char	ambiguous sign of char On some compiler, char means unsigned char [0, 255]. On some compiler, char means signed char [-128, 127].	1	1	1	1
unsigned int	An unsigned integer. Range depends on size. range is $[0, 2^{\text{bytes} \cdot 8} - 1]$ for Z80, it is $[0, 65535]$ or $[0x0000, 0xFFFF]$	2	4	4	4
int (automatically signed)-can also be signed int	A 2's complement signed integer. Range depends on size. range is $[-2^{\text{bytes} \cdot 8 - 1}, 2^{\text{bytes} \cdot 8 - 1} - 1]$. For Z80 it is $[-32768, 32767]$ or $[0x8000, 0x7FFF]$	2	4	4	4
void *	a pointer to an address on the address bus. For Z80, address range is $[0x0000, 0xFFFF]$ which fits in 2 bytes – much more later	2	8	8	8
double	Scientific number like $a \cdot 10^n$ like $-1.602 \cdot 10^{-19}$ Not supported on Z80	N/A	8	8	8

int size varies depending on CPU
dangerous for hardware design

C99 includes a fix:

You can “fix Boolean,” “fix the size,” and “fix the sign” of data types in your C code by:

`#include <stdbool.h>` ← add **bool**, **true**, and **false** – **true** is always **1** and **false** is always **0**.

`#include <stdint.h>` ← add **uintN_t** and **intN_t**, where **N** can be 8, 16, 32, or 64. also adds **MIN_** and **MAX_**

After including the above 2 lines at the beginning of your C program, you will have access to:

type	meaning	sizeof (.) - value in number of bytes for C99			
		Z80	intel Core 64	Apple ARM64	RV64
bool	Boolean: true (1) or false (0)	1	1	1	1
char	Some people still use char . Don't compute using char -- (bad example: char3 = char1 + char2;)	1	1	1	1
uint8_t	an unsigned 8-bit (1 byte) number in [0, UINT8_MAX]	1	1	1	1
int8_t	a signed 8-bit (1 byte) number (2's complement) from [INT8_MIN, INT8_MAX]	1	1	1	1
uint16_t int16_t	an unsigned or signed integer. 16 bits. (2 bytes) uint16_t range is [0, UINT16_MAX] int16_t range is [INT16_MIN, INT16_MAX]	2	2	2	2
uint32_t int32_t	an unsigned or signed integer. 32 bits. (4 bytes) uint32_t range is [0, UINT32_MAX] int32_t range is [INT32_MIN, INT32_MAX]	4	4	4	4

How do we store multi-byte data?

Suppose we want to store an **int16_t** (integer) value 6699 (0x1A2B) in memory address 0x8000

int16_t requires 16 bits (2 bytes), so we need 2 addresses to store them (0x8000-0x8001).

Almost all modern CPUs store multi-byte data in **little-endian** (little end first) format.

address	data
0x8000	0x2B
0x8001	0x1A

← For 0x1A2B, 0x2B is smaller, so it goes into 0x8000.

The bigger 0x1A goes to the next address 0x8001.

For **uint32_t** or **int32_t** value 0x1A2B3C4D stored at 0x4008, it goes like this:

address	data
0x4008	0x4D
0x4009	0x3C
0x400A	0x2B
0x400B	0x1A

Z80, intel Core, ARM64, RISC V-64 are all little-endian CPUs (note on next slide for ARM64)



How do we store multi-byte data?

Some machines store data in **big-endian** format (rare). Big End first

They store 0x2C19A030 at address 0x8E94 like this:

address	data	
0x8E94	0x2C	← 0x8E is the biggest one
0x8E95	0x19	...
0x8E96	0xA0	
0x8E97	0x30	← 0x30 is the smallest one

Some modern CPUs (like ARM64), are **bi-endian** (can support both little endian / big endian), but they are modern, and 99% of the time running in little-endian mode.

A notable exception: internet traffic. On the internet, we send multi-byte data all the time. **We always send the most significant byte first** (somewhat like big-endian).


Suppose we want to send 0x1122334455667788 over the internet, we send:

first byte → 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88 ← last byte sent

Storing a long “string” of symbols

We store a lot of types of numbers in a computer.

- Computer addresses those data in bytes through the address.
- each text character (ASCII) is a byte, so storing them is easy.
- if we want to store a “string” of text, here’s what we need:
- Address of the first character of the string. Add another byte with value `0x00` (called NUL) at the end. Example: if we store “Dear” starting at address `0x0123`:

address	data in hex	ASCII symbol
...	??	
0x0122	??	
 0x0123	0x44	D
0x0124	0x65	e
0x0125	0x61	a
0x0126	0x72	r
0x0127	0x00	NUL
...	??	

Some interesting *non-printable* ASCII codes:

Standard:

`0x00` (NUL) – always designate the end of string

`0x00` (BEL) – plays a “ding” sound!

`0x10` (LF) – *linefeed* – move to a new line, keep x position

`0x13` (CR) – *carriage return* – move to leftmost character (and *sometimes* move to a new line as well)

It takes 5 bytes to store string of length 4.

It takes $(n+1)$ bytes to store string of length n

So, to refer to a string, we give its beginning address.

A *memory address* is also called a *pointer* in C.

intentionally left blank

Our first C program

Suppose we want a program that calculates -27×54

Before we commit to data size, let's analyze a bit:

- inputs are not bigger than 8-bit signed, and they are signed (-27) (54)
- answer is *bigger* than 8-bit signed (127), so 16-bit signed integer should be used
- and we can make both inputs and outputs **int16_t** (16-bit, 2-byte) signed numbers.

C99 Code	Explanation
#include <stdbool.h>	INCLUDE standard header (definitions) for Boolean (true/false) – name ends in .h
#include <stdint.h>	INCLUDE standard header for fixed size integers (uint8_t , int8_t , uint16_t , int16_t)
void main(void) {	START the main function (main is always C app starting point):
int16_t a;	DECLARE an int16_t variable a
a = -27;	ASSIGN a to value -27
int16_t b = 54;	DECLARE and ASSIGN int16_t variable b to 54
int16_t m;	DECLARE an int16_t variable m
m = a*b;	ASSIGN m to value a*b
} [00-201]	END function main

Our first C program

Let's *compile* and *link*. Works perfectly. We got *.bin* executable. We put *.bin* in SimulIDE Z80 ROM and run it.

It runs, but **we cannot see the output *m*. Why not?**

1. Because our Z80 system is *headless*; it has no input from / output to the outside world.
2. After it calculates -27×54 , we have no idea which address variable *m* is stored at. we don't know the address of *a* or *b* either. We don't know where the *linker* decides to store *a*, *b*, and *m*

→ We have a great solution for this. It is called a **pointer**. → **pointer** lets us specify the address of *m*.

C99 Code	Explanation
<code>#include <stdbool.h></code>	INCLUDE standard definitions for Boolean (true/false)
<code>#include <stdint.h></code>	INCLUDE standard definitions for fixed size integers (<code>uint8_t</code> , <code>int8_t</code> , <code>uint16_t</code> , <code>int16_t</code>)
<code>void main(void) {</code>	START the <code>main</code> function (<code>main</code> is always C app starting point):
<code>int16_t a;</code>	DECLARE an <code>int16_t</code> variable <code>a</code>
<code>a = -27;</code>	ASSIGN <code>a</code> to value <code>-27</code>
<code>int16_t b = 54;</code>	DECLARE and ASSIGN <code>int16_t</code> variable <code>b</code> to <code>54</code>
<code>int16_t m;</code>	DECLARE an <code>int16_t</code> variable <code>m</code>
<code>m = a*b;</code>	ASSIGN <code>m</code> to value <code>a*b</code>
<code>}</code>	END function <code>main</code>



Intro to pointers 1

HEADS UP! IMPORTANT

The next slides (*Intro to pointers 1.x*) are probably the most important slides in this class. Pay close attention. If you do not understand, ask for assistance from the teaching staff right away. Do not wait.



Intro to pointers 1.1

What is a Pointer? A pointer is one of the most important concepts in C. It is just another variable, but with a specific purpose.

- **Normal Variable** (like `uint8_t`, `int16_t`, etc.): Stores a value (e.g., 5, 255, -1).
- **Pointer Variable:** Stores a **memory address** (e.g., `0x4200`).

Pointers on the Z80:

Since the Z80 CPU has a 16-bit address bus, a pointer variable is always 2 bytes (16 bits) in size, *regardless of what it points to*. It needs this size to be able to hold any address from `0x0000` to `0xFFFF`.

The Importance of pointer Type

- Even though all pointers are 16-bit addresses on the Z80, **a pointer must always have a type**. The type tells the compiler how to read the data once it gets to that address.
 - Does it read 1 byte (`uint8_t`, `int8_t`, `bool`)?
 - Does it read 2 bytes (`int16_t`, `uint16_t`)?
 - Does it read 4 bytes (`int32_t`, `uint32_t`)?



Intro to pointers 1.2

Syntax and Declaration

- C uses the asterisk `*` to declare a pointer.

```
uint8_t *p1;
```

How to read this:

- **Don't say:** "`p1` stores `uint8_t` data." (WRONG)
- **Do say:** "`p1` stores an **address**. At that address, there is `uint8_t` data."

A helpful mental trick is to associate the asterisk with the type:

```
(uint8_t *)p1; // Read as: p1 is a "pointer-to-uint8_t"
```

Do not put in () in code. they just help you understand. Putting in () results in a compiler error.

- **Another Example:**

```
int16_t *p2; // p2 is (a 16-bit variable) holding an address.  
            // At that address, the computer expects to find an int16_t.
```



Intro to pointers 1.3

Common Pitfall: Multiple Declarations

Be careful when declaring multiple pointers on one line. The * attaches to the variable name, not the type.

If you want to declare two pointers to Booleans (b0 and b1):

```
// CORRECT
```

```
bool *b0, *b1;
```

```
// INCORRECT
```

```
bool* b0, b1;    // This declares 'b0' as a pointer-to-bool...  
                 // ...but 'b1' is just a regular bool!
```

```
// The incorrect code is equivalent to:
```

```
bool *b0;
```

```
bool b1;
```



Intro to pointers 1.4

The Address-of Operator (&)

Idea: "Where does this variable live?"

What it does: The **&** operator retrieves the memory address of a variable.

Analogy: If **var** is the *value* inside the mailbox, **&var** is the *address* of that mailbox.

The Type Change:

- If **x** is a **uint8_t** then **&x** is a **uint8_t *** (a pointer).

Read it as: "The address of..."

Example:

```
uint8_t count = 50;           // Value is 50
uint8_t *ptr = &count;        // ptr now holds the address of count
```

Important: You cannot manually invent safe addresses (usually), except in our class, you can safely use **0x4000-0x40FF** (see memory map). You ask the system for existing addresses using **&**.

We gave you **0x4000-0x40FF**, so you can play with pointers reading & writing to those RAM addresses. *Linker* will avoid the above address range. see *memmap.ld* from a previous slide.



Intro to pointers 1.5

The Dereference Operator (*) – Part I

- **Idea:** "Go to the address and see the content."
- **The Confusion:** In C, the `*` symbol has *two meanings* depending on context:
 1. **Declaration:** `uint8_t *p;` (Creates a pointer).
 2. **Dereference:** `val = *p;` (Follows the pointer).
- **What it does:** The `*` operator goes to the address stored in the pointer and accesses the data there.
- **Read it as:** "The value pointed to by..."

- **Visualizing the Link:** `ptr` is holding the address of `count`.
`ptr` → points to → `count`

Example from previous slide: if you start out with:

```
uint8_t count = 50;      // Value is 50
uint8_t *ptr  = &count;  // ptr has address of count
```

- **The "Remote Control" Effect:**
 - Since `ptr` points to `count`, using `*ptr` is like using a remote control to change `count`.
 - **Code:** `*ptr = 99;`
 - **Result:** The value of `count` changes to 99.
- **Key Concept:** You modified `count` without touching the variable `count` directly.



Intro to pointers 1.6

The Dereference Operator (*) – Part II

The scenario on the previous slide (remote control) usually happens between 2 distinct functions. C has a terminology for this:

1. The Default: Pass-by-Value (Safety)

- **How C works normally:** When the *Caller* (Function A) sends variables to the *Callee* (Function B), C makes a **copy** of that data.
- **The Benefit:** Function B works on a photocopy. It cannot accidentally corrupt Function A's original data.
- **The Problem:** Copying takes time and memory.

Z80 Context: If you have an array of 100 bytes, copying it fills up your stack and wastes CPU cycles.

2. The Solution: Pass-by-Reference (Efficiency)

- **How Pointers help:** Instead of copying the data, the *Caller* sends the address (*pointer*) of the data.
- **The Result:** Function B (*Callee*) goes to that address and works on the original data directly.
- **The Trade-off:**
 - **Pros:** Extremely fast. Uses only 2 bytes of stack space (the size of pointer), no matter how big the data is.
 - **Cons:** Function B can modify (or corrupt) Function A's data.

Pass-by-Value = email you a copy of a Word file ----- **Pass-by-Reference** = share Word file on OneDrive and let you edit



Intro to pointers 1.7 – pointer arithmetic

Side Note: `void *`

Definition: Address Known, Size Unknown.

Rule: Cannot be dereferenced or used in arithmetic without casting to a concrete type first.

Pointer Arithmetic & Data Types

The Concept: Pointer arithmetic is **not** integer addition. It is **address calculation** based on data type size. When you add `1` to a pointer, the compiler generates Z80 instructions to increase the address by `sizeof(type)`

Hardware Reality (Address Bus):

`uint8_t *ptr`: Step size is `1` byte. Address Bus changes: `0x4000` to `0x4001`.

`uint16_t *ptr`: Step size is `2` bytes. Address Bus changes: `0x4006` to `0x4008`.

Code Example (Student RAM `0x4000-0x40FF`):

```
uint8_t *p8  = (uint8_t *)0x4000U;
uint16_t *p16 = (uint16_t *)0x4006U;

// Arithmetic
p8  = p8 + 1;    // Address becomes 0x4001 (+1 byte)
p16 = p16 + 1;  // Address becomes 0x4008 (+2 bytes)
```



Intro to pointers 1.8 – pointer arithmetic

Pointer arithmetic in C99:

1. Convention: `base_address + offset`

Rule: Always write `ptr + offset`, never `offset + ptr`.

Why: C allows `3 + ptr`, but it confuses human. We always write `Base Address (Pointer) + Displacement (Integer)`.

Correct: `uint16_t *p = ...; p = p + 1;`

Avoid: `p = 1 + p;`

2. The Scaling Law.

Rule: Arithmetic operates on **Item Count**, not Byte Count.

Why: The compiler automatically multiplies the **integer offset** by `sizeof(type)`.

Hardware Reality: If you have `uint16_t *p` at `0x4000` and do `p + 1`, the CPU adds 2 to the address.

3. The Addition Trap: `ptr + ptr` is invalid in C.

4. The Void Prohibition

Rule: Never perform arithmetic on `void *`

Why: `void *` represents a **raw memory address** with **no defined size**. The compiler cannot calculate the scale factor.

Fix: You must cast a `void *` to a concrete type (usually `uint8_t *` for raw memory manipulation) before adding or subtracting.



Intro to pointers 1.9 – pointer arithmetic

Pointer arithmetic in C99:

5. The Subtraction Duality

Rule: Know the difference between `ptr - int`, and `ptr - ptr`.

Pointer - Integer: Returns a New Address.

`p - 1` moves the pointer back one element.

Pointer - Pointer: Returns a Long Integer (`ptrdiff_t`) – which is `int16_t` in z80.

`p2 - p1` returns the number of elements between them, not the number of bytes.

Formula: (Address2 - Address1) / sizeof(type)

Warning 1: you can only subtract pointers of the *same type*.

Warning 2: The 32KB Signed Limit

Because `ptrdiff_t` is a signed 16-bit integer (`int16_t`), the maximum positive distance it can represent is 32,767 (0x7FFF).

The Bug: If two pointers are more than 32K *elements* apart, the subtraction overflows.

Example:

```
uint8_t *p1 = (uint8_t *)0x0100;  
uint8_t *p2 = (uint8_t *)0xC200;  
ptrdiff_t d1 = p2 - p1;    // fails. ptrdiff_t is int16_t -> can't represent result
```

```
uint16_t *q1 = (uint16_t *)0x0200;  
uint16_t *q2 = (uint16_t *)0xD300;  
ptrdiff_t d2 = q2 - q1;    // works. (0xD300 - 0x0200)/2 = 0xD100/2 = 0x6880
```




Intro to pointers 1.10

The Golden Rule (Summary) Page 1 of 2

1. **&** (Ampersand) → Gets the address

- **Rule:** A pointer on Z80 is always 2 bytes (16 bits) but the *target size* varies.

```
uint8_t  v  = 5;    // Data size: 1 byte
uint8_t *p1 = &v;    // Ptr  size: 2 bytes (holds address of v)

uint32_t  w  = 17;   // Data size: 4 bytes
uint32_t *p2 = &w;   // Ptr  size: 2 bytes (holds address of w)
```

2. ***** (Asterisk) → Accesses the data

- **Rule:** Use ***** to Read or Write the target value.

```
uint8_t q = *p1;    // READ: q copies the value inside v
*p2 = 35;           // WRITE: w changes to 35
```

3. The Relationship: **&** and ***** are opposites:

- If **x** is a variable (uint8_t, uint16_t, etc.):
 ***(&x)** is logically the same as just **x**.
- If **y** is a pointer (uint8_t *, uint16_t *, etc.):
 &(*y) is logically the same as just **y**.



Intro to pointers 1.11

The Golden Rule (Summary) Page 2 of 2

app

comp arch

logic

4. SAFETY WARNINGS: Respect the Memory

4.1 The "Wild Pointer" (Uninitialized)

- **Error:** Using a pointer that points nowhere.

```
uint8_t *p;    // Address is RANDOM GARBAGE (example: 0xD2F1)
*p = 5;        // DANGER! You wrote 5 to a random address. (example: 0xD2F1)
```

4.2 The "Type Mismatch" (Overflow)

- **Error:** Pointing a **large** pointer at a **small** variable.
- **Why it breaks:** The pointer writes more bytes than the variable owns.

```
uint8_t  small = 0;           // Owns 1 byte of memory
uint16_t *p = (uint16_t*)&small; // Lie to compiler: Treat it as 2 bytes
*p = 0xBEAD;                  // WRITE: Fills 'small' (1 byte)
                               // ...AND overwrites the next byte in RAM!
```



Intro to pointers – Recap 1

type name	example variable	what is it?	size (bytes)	address of variable (example)	value stored inside	Can this value be used as an address?	How to access the target?
uint8_t	d	8-bit unsigned integer	1	0x4200	255 (0xFF)	no	N/A
bool	e	Boolean flag	1	0x4201	true (0x01)	no	N/A
int16_t	f	16-bit signed integer	2	0x4202	-1234 (0xFB2E)	no	N/A
uint32_t	g	32-bit unsigned integer	4	0x4204	409876 (0x00064114)	no	N/A
uint8_t *	p	pointer to uint8_t	2	0x4208	0x4200	yes (points to d)	*p
bool *	q	pointer to bool	2	0x420A	0x4201	yes (points to e)	*q
int16_t *	r	pointer to int16_t	2	0x420C	0x4202	yes (points to f)	*r
uint32_t *	s	pointer to uint32_t	2	0x420E	0x4204	yes (points to g)	*s

- **Why are pointers 2 bytes?** Because the Z80 CPU has a **16-bit Address Bus**. It takes 16 bits (2 bytes) to specify a location in memory (from 0x0000 to 0xFFFF).
- **Note on &:** The **address of variable** column is where the compiler decided to put **d, e, f, g, p, q, r, s**. You generally don't choose this; the compiler (SDCC) does.



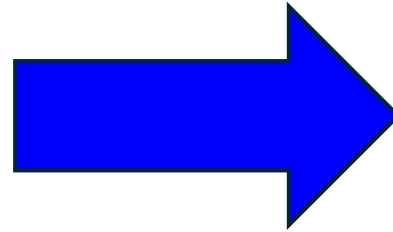
Intro to pointers – Recap 2

app

comp arch

logic

type name	variable	what is it?	size (bytes)	address of variable (example)	value stored inside
uint8_t	d	8-bit unsigned integer	1	0x4200	255 (0xFF)
bool	e	Boolean flag	1	0x4201	true (0x01)
int16_t	f	16-bit signed integer	2	0x4202	-1234 (0xFB2E)
uint32_t	g	32-bit unsigned integer	4	0x4204	409876 (0x00064114)
uint8_t *	p	pointer to uint8_t	2	0x4208	0x4200
bool *	q	pointer to bool	2	0x420A	0x4201
int16_t *	r	pointer to int16_t	2	0x420C	0x4202
uint32_t *	s	pointer to uint32_t	2	0x420E	0x4204



ADDRESS MAP

Address	Data	
0x4200	0xFF	uint8_t d ← (Value: 255)
0x4201	0x01	bool e ← (Value: true)
0x4202	0x2E	int16_t f ← (Value: -1234)
0x4203	0xFB	
0x4204	0x14	uint32_t g ← (Value: 409876)
0x4205	0x41	
0x4206	0x06	
0x4207	0x00	
0x4208	0x00	uint8_t *p — (Holds addr: 0x4200)
0x4209	0x42	
0x420A	0x01	bool *q — (Holds addr: 0x4201)
0x420B	0x42	
0x420C	0x02	int16_t *r — (Holds addr: 0x4202)
0x420D	0x42	
0x420E	0x04	uint32_t *s — (Holds addr: 0x4204)
0x420F	0x42	



Intro to pointers – Finale

Q: Is there a pointer to a pointer? A: YES.

```
uint8_t **v;
```

- **v** is a variable (2 bytes).
- It holds the address of a **second** variable (also 2 bytes).
- That second variable holds the address of the data (1 byte).

Q: Is there a pointer to a pointer to a pointer...? A: YES.

```
uint8_t *****w;
```

- There is no conceptual limit. You can chain addresses if you have memory to store them.

We will restrict ourselves to single pointers (*) in this class. *Why?* Multiple levels of indirection are used for complex dynamic data structures (like linked lists or OS kernels). For hardware control, one level of indirection (CPU → Hardware Register) is usually enough.

Before we fix our headless C program

Now we are in C99 land, and there's one clarification:

Hex Literals & The U Suffix (Part 1)

- **The "Signed" Default:** By default, C treats small hex *literals* as **signed int**. This can cause bugs during bitwise operations (like `>>`) due to sign extension.
- **The Safety Rule:** To force a constant to be treated as **unsigned int**, you must append the **U** suffix in your code.
- **Course Convention (Important):** To keep our slides clean and readable, I will omit the suffix in diagrams and bullet points.
 - **On Slides:** I will write `0x4000`
 - **In Code:** You must write `0x4000U`
- **Example:**

```
// DANGEROUS: Compiler may treat as signed int
```

```
uint32_t val = 0x80000000 >> 1;    // val may become 0xC0000000, a negative number
```

```
// SAFE: Explicitly unsigned
```

```
uint32_t val = 0x80000000U >> 1;    // val is guaranteed to be 0x40000000U
```



Before we fix our headless C program

Hex Literals & The U Suffix (Part 2)

The "Mixed Comparison" Trap

- **The Trap: Auto-Unsigned Promotion:**

In C99, a hex literal like `0xC000` automatically becomes **unsigned int** if it doesn't fit in a **signed int** (common on 16-bit systems).

- **The Bug: Signed vs. Unsigned:** If you compare a *signed variable* (like an error code or offset) against an *unsigned hex literal* (like `0xA321U`), the *signed variable* is implicitly *promoted* to a large unsigned number.
- **The Fix:** Always use the **U** suffix for addresses/masks, and explicit casts for comparisons.

Example: `int16_t offset = -1; // Error condition`

```
// BUG: 0xC000 is unsigned (on 16-bit).
```

```
// 'offset' promotes to 0xFFFF. Result is TRUE!
```

```
if (offset > 0xC000) {  
    jump_to_app();    // Oops, jumps to crash  
}
```

```
// SAFE: Force valid comparison
```

```
if (offset > 0xC000U) ... // Still risky, better to cast variable
```

★ volatile Pointers

app

comp arch

logic

The Problem: Compiler Optimization vs. Hardware Reality

Compiler's Job: It optimizes code for speed. It assumes memory values only change if *the code itself* changes them.

The Trap: To save time, the compiler may read a memory address **once**, cache the value in a CPU register and reuse that register value instead of reading from RAM again.

Hardware Reality: Device registers (status flags, I/O ports) change **externally** (by the hardware, not your code). If the compiler caches the old value, your code misses the hardware update.

Conclusion: ALWAYS declare pointers to IO addresses as **volatile**. (Prefix pointer declaration or casting with **volatile**)

Example: Polling a Status Register (Address 0xD200)

✗ WITHOUT **volatile** (The Bug)

The compiler sees that ***ptr** is not modified *inside* the **while** loop. It moves the memory read outside to optimize.

```
uint8_t *ptr = (uint8_t *)0xD200U;

// COMPILED LOGIC:
// 1 Read 0xD200 into Register in CPU (A).
// 2 Is Register A == 0?
// 3 If yes, jump back to self, CPU NEVER reads 0xD200 again
while (*ptr == 0) {
    // Waiting for hardware...
}
```

✓ WITH **volatile** (The Fix)

The **volatile** keyword means: *"This address may change unexpectedly. Never cache it. Always read from actual memory."*

```
// Correctly defined pointer
volatile uint8_t *ptr = (volatile uint8_t *)0xD200U;

// COMPILED LOGIC:
// 1. Read 0xD200 into Register A.
// 2. Is Register A == 0?
// 3. If yes, jump back to Step 1 (Re-read memory).
while (*ptr == 0) {
    // Successfully detects changes at 0xD200
}
```




Let's write better C

But, before that, a condensed *coding guideline* (full version is on class repo at **docs-students/**):

[Rules 1.1, 1.2, 3.2] - Always define macros to access hardware this way:

```
#define SOME_IO_ADDRESS ((volatile type*)0xC300U) // see more about type below
```

[Rule 2.1] - Always use <stdint.h> for type above:

- **Never** use standard `int`, `short`, or `long`.
- **Addresses:** Always use `uint16_t` (Z80 address bus is 16-bit).
- **Unsigned Data:** Use `uint8_t` for 8-bit values and `uint16_t` for 16-bit values.
- **Signed Data:** Use `int8_t` or `int16_t` **only** when dealing with signed data (2's complement number), whether for calculation or reading/writing from/to IO address.

[Rule 2.2] - Always Use `bool` from <stdbool.h> for logic.

- Use `true` and `false` for flags and logic

[Rule 5.2] - Infinite Main Loop:

- Always declare the `main` function like this: `void main(void)`
- Embedded programs never "exit". `main()` shall end with `while(true) {}`.

[Rule 5.3] - Keep Functions Short: Limit functions to ~30 lines where possible.

Let's write prettier C

A condensed *style guide* (full version is on class repo at **docs-students/**):

[Rule 5] - Naming Conventions

- **Variables & Functions:** Use **snake_case** (lowercase with underscores).

```
uint8_t sensor_value;
void read_data(void);
```

- **Constants & Macros:** Use **UPPER_CASE**.

```
#define LED0 ((volatile uint8_t *) 0xDEA0U)
#define MAX_BUFFER 255
const uint8_t LED_PIN_INDEX = 1U;
```

- **Pointer Spacing:** Always attach the asterisk (*****) to the **variable name**, not the type.

```
uint8_t *data_ptr; // good example
uint8_t* data_ptr; // bad example
```

There is an *exception* to ***name** above, but it's for later.



[Rule 3] - Formatting & Indentation

- **Indentation:** Use **4 Spaces**. Do not use tab.
- **Braces:** Use "One True Brace Style" (opening brace on the same line). **Always** use braces, even for single statement.
- **One Statement Per Line:** Never combine logic or block formatting onto a single line.

Why: This improves readability and ensures step-through debuggers can stop at every specific action.

Bad (Hard to read/debug):

```
if (ready) { go = true; }
```

Good:

```
if (ready) {
    go = true;
}
```

- **Closing Braces Alignment:** Closing braces (**}**) must vertically align with the line that opens the brace (**{**)

Why: This maintains clear visual block structure and prevents the "**dangling else**" error where the else accidentally attaches to the wrong **if** statement.

Example: See *style guide* document for example.

Let's go back to fix our headless C program

OLD headless code –
can't see results

```
#include <stdbool.h>
#include <stdint.h>

void main(void) {
    int16_t a;
    a = -27;
    int16_t b = 54;
    int16_t m;

    m = a * b;
}
```

NEW code - we can now see results
at 0x4000

```
#include <stdbool.h>
#include <stdint.h>

#define M_ADDR ((int16_t *)0x4000U)

void main (void) {
    int16_t a;
    a = -27;
    int16_t b = 54;
    int16_t *p = M_ADDR;

    // multiply. yeah!
    *p = a * b;
}
```

explanation of the difference

why don't we need **volatile**?

0x4000 is our magic address for multiplication result. (we chose it).

declare **int16_t** pointer **p**. set value to 0x4000U, so we can:

***p = a * b; // write to M_ADDR**

int16_t foo;
foo = *p; // read from M_ADDR

Let's compile and link and run our executable in the simulator

Let's go back to fix our headless C program

code

```
#include <stdbool.h>
#include <stdint.h>

#define M_ADDR ((int16_t *)0x4000U)

void main (void) {
    int16_t a;
    a = -27;
    int16_t b = 54;
    int16_t *p = M_ADDR;

    *p = a * b;
}
```

[03-202]

simulation result - looking inside RAM at 0x4000

RAM: RAM: 0x4000-0xBFFF

	00	01	02	03	04	05	06	07	08	09	0A	0B
0x0000	4E	FA	00	00	00	00	00	00	00	00	00	00
0x0010	00	00	00	00	00	00	00	00	00	00	00	00
0x0020	00	00	00	00	00	00	00	00	00	00	00	00
0x0030	00	00	00	00	00	00	00	00	00	00	00	00
0x0040	00	00	00	00	00	00	00	00	00	00	00	00
0x0050	00	00	00	00	00	00	00	00	00	00	00	00
0x0060	00	00	00	00	00	00	00	00	00	00	00	00
0x0070	00	00	00	00	00	00	00	00	00	00	00	00
0x0080	00	00	00	00	00	00	00	00	00	00	00	00

RAM: 0x4000-0xBFFF

- **int16_t** is a 16-bit (2 bytes) signed integer, so it takes 2 bytes.
- we write **int16_t** to address **0x4000**, so the 2-byte data occupies **0x4000** and **0x4001**
- remember little endian, so **0x4E** is lower byte and **0xFA** is higher byte → data written is **0xFA4E**
- convert **0xFA4E** (which is in 2's complement) to base 10, we have **0xFA4E = -1458**, which is **-27 x 54**

You may not appreciate it yet, but we have achieved a very major milestone – we can read output!

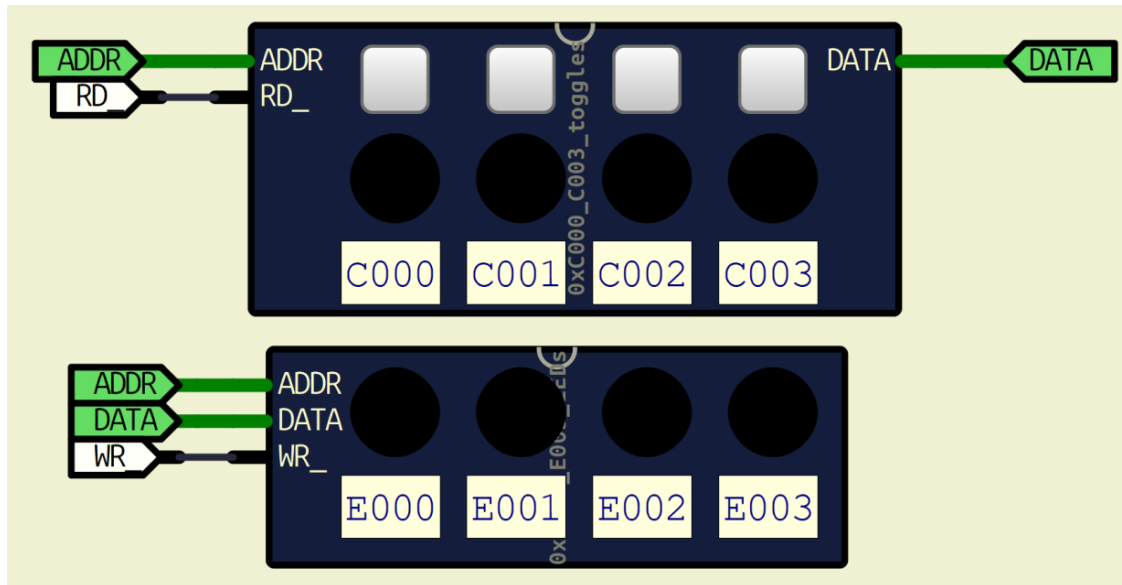
intentionally left blank

02 From Text to Machine

Let's build a blinking light.

we have 4 lights at 0xE000-E003

- writing true to the above address turns light on
- writing false turns light off.



C99 Code [04_203]

```
#include <stdbool.h>
#include <stdint.h>

#define LED0 ((volatile bool *)0xE000U)

void main(void) {

    volatile bool *led;
    led = LED0;

    while (true) {
        // toggle light
        *led = true;
        *led = false;
    }
}
```

02 From Text to Machine

New C knowledge: MACRO DEFINITION (aka #define)

MACRO DEFINITION of LED0: Find & Replace every "LED0" with
"((volatile bool *)0xE000U)"

New C knowledge: while () statement:

```
while (expression) {  
    // code1  
}  
// code2
```

1. *expression* is evaluated.
2. If it is **true**, *// code1* is executed until reaching "}"
3. go back to 1. (evaluate *expression* again)
4. *// code2* will only execute after *expression* turns **false**.

C99 Code [04_203]

```
#include <stdbool.h>  
#include <stdint.h>
```

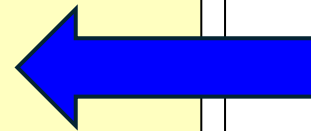
```
#define LED0 ((volatile bool *)0xE000U)
```

```
void main(void) {
```

```
    volatile bool *led;  
    led = LED0;
```

```
    while (true) {  
        // toggle light  
        *led = true;  
        *led = false;  
    }
```

```
}
```



02 From Text to Machine

Summarizing the ingredients in a C program:

A. Source text: (removed/processed before parsing)

A1: *Preprocessor Directives:* `#include` and `#define`

A2: *Comments:* for human reading only: **2 styles:**

`/* --- from slash-star to star-slash is comment --- */`

`// from double slashes to end of this line is comment`

B. The actual C Language: (parsed by the compiler)

B1: *Declarations/Definitions:* the "What"

introduces names of variables and functions, as well as declaring their properties (type, return type, etc.)

B2: *Statements:* the "Flow"

control flow: `while`, `if-else`, `for`, and grouped actions
`{ ... }`

B3: *Expressions:* "Math/Action" - calculations & *side effects*

`led = LED0;`

`*led = true;`

`m = a * b;`

`true` (condition for while)

C99 Code [04_203]

```
#include <stdbool.h>
```

```
#include <stdint.h>
```

```
#define LED0 ((volatile bool *)0xE000U)
```

```
void main(void) {
```

```
    volatile bool *led;
```

```
    led = LED0;
```

```
    while (true) {
```

```
        // toggle light
```

```
        *led = true;
```

```
        *led = false;
```

```
    }
```

```
}
```


02 From Text to Machine

app

comp arch

logic

Summarizing the ingredients in a C program:

A. Source text: (removed/processed before parsing)

A1: *Preprocessor Directives:* `#include` and `#define`

A2: *Comments:* for human reading only: **2 styles:**

`/* --- from slash-star to star-slash is comment --- */`

`// from double slashes to end of this line is comment`

B. The actual C Language: (parsed by the compiler)

B1: *Declarations/Definitions:* the "What"

introduces names of variables and functions, as well as declaring their properties (type, return type, etc.)

B2: *Statements:* the "Flow"

control flow: `while`, `if-else`, `for`, and grouped actions
`{ ... }`

B3: *Expressions:* "Math/Action" - calculations & side effects

`led = LED0;`

`*led = true;`

`m = a * b;`

`true` (condition for while)

C99 Code [04_203]

`#include <stdbool.h>` A1

`#include <stdint.h>` A1

`#define LED0 ((volatile bool *)0xE000U)`

↖ A1

↖ B1

↖ B3

↙ B1

`void main(void) {`

B1 `volatile bool *led;`

B3 `led = LED0;`

↙ B3

B2→ `while (true) {`

A2 `// toggle light`

B3 `*led = true;`

B3 `*led = false;`

`}`

B2

`}`



02 From Text to Machine

app

comp arch

logic

More C terminologies (CHECK THAT YOU UNDERSTAND EVERY LINE):

```
#include <stdbool.h>           // filename.h is called a header file and we #include it
                                // < ... > designates a standard library header -- used system-wide

#include "my_project.h"        // "filename.h" is a user-defined header - for our project only

#define CHAR_A (0x77U)         // MACRO, also called #define -- find & replace
#define LED0 ((volatile bool *)0xE000U)
#define DISPLAY_BASE ((volatile uint8_t *)0xE100U)

void main(void)               // void means "nothing." In this case, the main function takes void (nothing) and
                                // (nothing) as arguments and returns void (nothing)

int16_t sum(uint8_t a, int8_t b) // function sum takes 2 arguments. a: 8-bit unsigned int,
                                // and b: 8-bit signed int. sum returns 16-bit signed int.

bool c, d, e; uint8_t p, q;    // declare variables c, d, e as Boolean type. p, q as 8-bit unsigned int
c = true; p = 3; q = -4;       // = is called assignment. variable on LHS of = is updated.

uint8_t r = 14; int16_t s = 2770; // variables can be declared and assigned in one statement

volatile uint8_t *y;           // pointers need to be declared. A pointer variable stores the target's address
y = DISPLAY_BASE;              // if y is a pointer, assign it ONLY WITH pointer of the same type

q = *y;                        // READ: Copy the value from y's target address into q.
*y = 4U;                       // WRITE: Store the value 4U into y's target address.
```

let's see the **[04_203]** blinking light already!

DRY principle and function calls

- Update Z80 CPU clock frequency to 200kHz (first physical version runs @2.5MHz, but our simulator can't)
- [\[05_203\]](#) Blinking light now is half-lit. Why? Because at 8MHz, light is turned on/off too quickly.

Solution: Let's make Z-80 busy doing "nothing" for a while:

- Regular technique (in C): let's make CPU count aimlessly. (🚀 There's a more "power-efficient" way.)
- Today's C compiler is smart. It optimizes for running speed. → Detects useless code and remove it.
- So, we use a **volatile** pointer to store counter variable (**c**) in a "safe zone" of real memory address.
- Start counter at 0 and keep counting until we do it 1,000 times.
- Memory map defines 0x4000–0x40FF to be "memory safe zone" (safe from *linker*, not from yourself!)
- I pick address 0x40F0 to be our *magic* address for **c**

```
#define COUNTER ((volatile uint16_t *)0x40F0U)

...
uint16_t *c = COUNTER;
*c = 0;
while (*c < 1000) {
    *c = *c + 1;
}
```

Key points:

1. Why **u** (unsigned)?
2. Why **16**? We have 8, 16, 32 available.
3. From 1, what addresses are used?
4. Are you sure ***c = *c + 1;** is performed *exactly* 1,000 times? Could it be 999? 1,001?

DRY principle and function calls

```
#include <stdbool.h>
#include <stdint.h>

#define LED0 ((volatile bool *)0xE000U)

void main(void) {

    volatile bool *led;
    led = LED0;

    while (true) {
        *led = true;
        *led = false;
    }
}
```

```
#include <stdbool.h>
#include <stdint.h> [05-204]

#define LED0 ((volatile bool *)0xE000U)
#define COUNTER ((volatile uint16_t *)0x40F0U)

void main(void) {

    volatile uint16_t *c = COUNTER;
    volatile bool *led = LED0;

    while (true) {
        *led = true;
        *c = 0; // intentional delay
        while (*c < 1000) {
            *c = *c + 1;
        }
        *led = false;
        *c = 0; // intentional delay
        while (*c < 1000) {
            *c = *c + 1;
        }
    }
}
```

DRY principle and function calls

```
#include <stdbool.h>
#include <stdint.h>

#define LED0 ((volatile bool *)0xE000U)
#define COUNTER ((volatile uint16_t *)0x40F0U)

void main(void) {

    volatile uint16_t *c = COUNTER;
    volatile bool *led = LED0;

    while (true) {
        *led = true;
        *c = 0; // intentional delay
        while (*c < 1000) {
            *c = *c + 1;
        }
        *led = false;
        *c = 0; // intentional delay
        while (*c < 1000) {
            *c = *c + 1;
        }
    }
}
```

[05-204]

2 possible improvements:

1st Programs *should not repeat* what can be written once.
This is called DRY principle. (Do not Repeat Yourself)

So, we should *refactor* delay code. *Refactor* means:

- improve internal structure, readability, and efficiency of our code
- while maintaining the same functionality:

We will *refactor* by *extracting function* from the blue background code.

This function will execute the intentional delay.

A *function* in C is

- a reusable block of code that takes arguments (optional) and performs a specific task, potentially returning a result.

DRY principle and function calls

A *function* in C is

- a reusable block of code that takes arguments (optional) and performs a specific task, potentially returning a result.

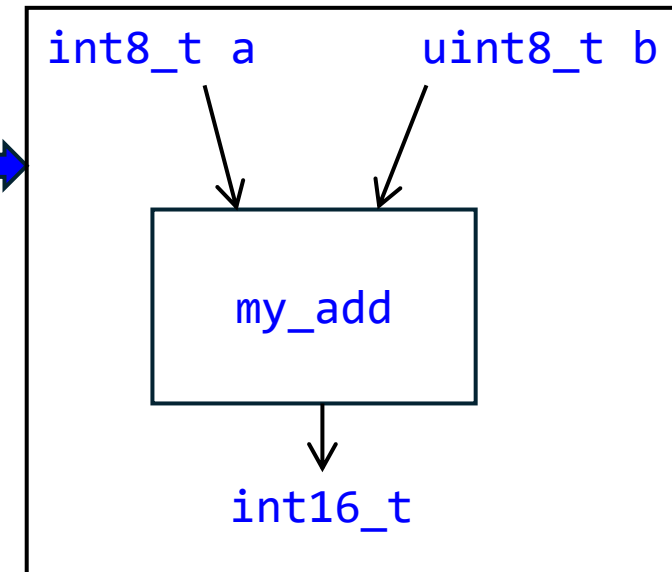
Let's declare a function. There's also a definition (aka function prototype), which will come later:

Function declaration must have 4 things:

1. function name
2. function arguments (list of inputs and their types into the function) – use `void` if you don't have any inputs
3. function return type (output type) – use `void` if you don't want any output.
--- {1, 2, 3} is called a **function prototype** – it tells the world (outside of function) how to use the function ---
4. function body (how it transforms inputs to output).

Example (written for ease of explanation, not following coding style:

```
int16_t my_add(int8_t a, uint8_t b) {  
    return a + b;  
}
```



How to read a function prototype: `my_add` is a function that takes 2 arguments:

- `a` of type `int8_t`, and `b` of type `uint8_t`
- This function *returns* data type `int16_t`
- (now you have to dig inside `{ }`) – it calculates its return value by doing `a+b`

DRY principle and function calls

Now let's apply it to our `delay`: with the 2nd function refactor improvement: *parameterization* (*parametrization*).

So, We will make `delay` take an argument, called `n` of type `uint16_t` (why?) and will run while loop `n` times.

1. function name
2. function arguments (list of inputs and their types into the function) – use `void` if you don't have any inputs
3. function return type (output type) – use `void` if you don't want any output.
4. function body (how it transforms inputs to output).

Declaration of `delay`:

```
void delay(void) {  
    volatile uint16_t *c = COUNTER;  
    *c = 0;  
    while (*c < 1000) {  
        *c = *c + 1;  
    }  
    return;  
}
```

How to read: `delay` is a *function* that takes no arguments (void):

- This function *returns* nothing (void)
- (now you have to dig inside { }) – it sets `*c = 0`; increments it one at a time until `*c` reaches 4000, then returns to the caller. It does not give any return value;

DRY principle and function calls

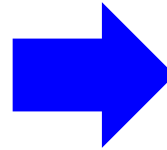
```
#include <stdbool.h>
#include <stdint.h>

#define LED0 ((volatile bool *)0xE000U)
#define COUNTER ((volatile uint16_t *)0x40F0U)

void main(void) {

    volatile uint16_t *c = COUNTER;
    volatile bool *led = LED0;

    while (true) {
        *led = true;
        *c = 0; // intentional delay
        while (*c < 1000) {
            *c = *c + 1;
        }
        *led = false;
        *c = 0; // intentional delay
        while (*c < 1000) {
            *c = *c + 1;
        }
    }
}
```



```
#include <stdbool.h> [05-205]
#include <stdint.h>

#define LED0 ((volatile bool *)0xE000U)
#define COUNTER ((volatile uint16_t *)0x40F0U)

void delay(uint16_t n) {
    volatile uint16_t *c = COUNTER;
    *c = 0;
    while (*c < n) {
        *c = *c + 1;
    }
}

void main(void) {

    volatile bool *led = LED0;

    while (true) {
        *led = true;
        delay(1000); // function call
        *led = false;
        delay(1000); // function call
    }
}
```

Function arguments in C are "passed by value"

- *pass by value* means copy the arguments, send copy of arguments to the function. Do not send origin. Analogy: send a copy of the file to your friend. your friend can modify it, but original file is with you.
- This contrasts with *pass by reference*. This sends the origin of the data. Analogy: send link to your file. Your friend edits *your* file.
- *pass by value* is safe, but inconvenient. *pass by reference* is dangerous, because there can be *side-effects* to the arguments.
- Having *side-effects* means modifications to data can occur.

pass by value (C99 behavior)

```
// your function
uint16_t increment_by_2(uint16_t a) {
    a = a + 2; // modify a
    return a;
}
```

```
// caller
uint8_t f = 16;
uint8_t g;
g = increment_by_2(f) + 1;
// g becomes 19. f stays 16.
```

pass by reference (NOT AVAILABLE in C)

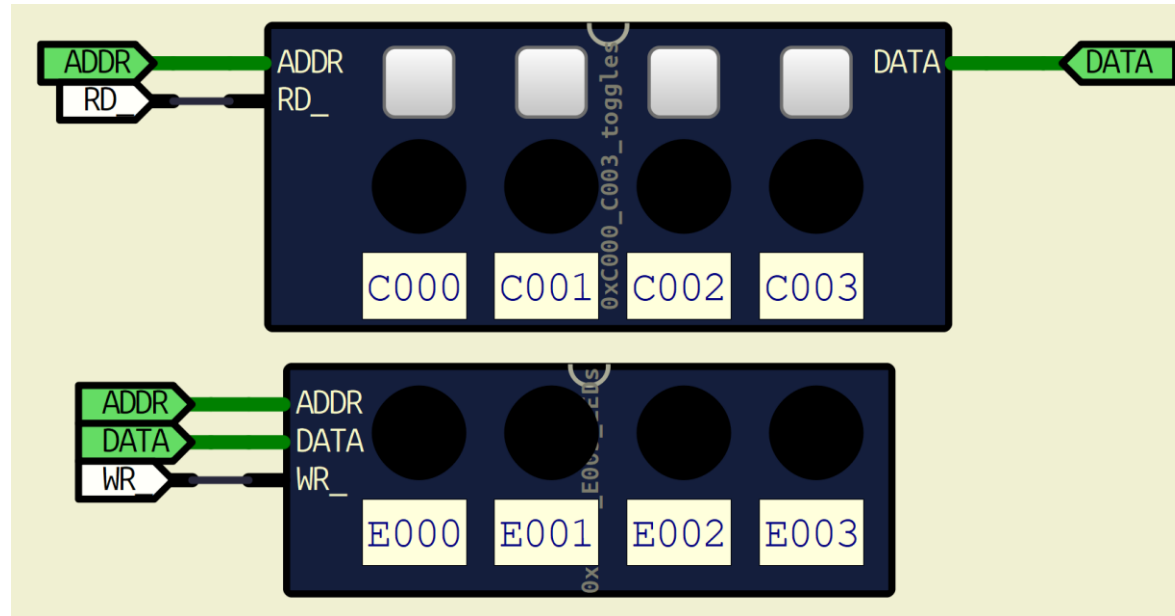
```
// your function
uint16_t increment_by_2(uint16_t a) {
    a = a + 2; // modify a
    return a;
}
```

```
// caller
uint8_t f = 16;
uint8_t g;
g = increment_by_2(f) + 1;
// g becomes 19. f is modified to 18.
```

let's see the **[05_205]** blinking light!

Combine Read and Write to IO

We have a blinking light. We were *writing* to IO device (LED).
Now, let's try *reading* from an IO device (switch).



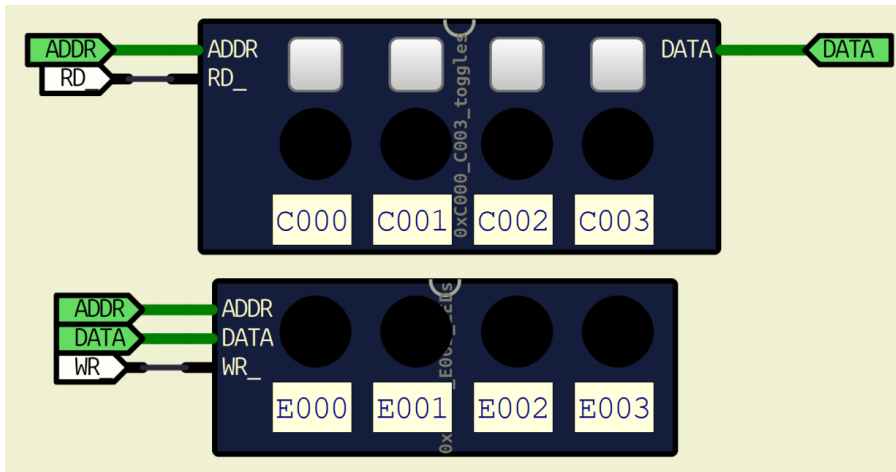
0xC000-3 contains four *toggle* switches. Reading from them yields 0 (status light off) or 1 (light on).
Pushing the switch above the light change its state.

Let's design this:

Loop forever:

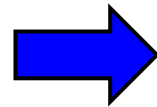
If 0xC000 is 1 (light on) let 0xE000 be off. Do the same for 0xC001-3

Combine Read and Write to IO



Loop forever:

If 0xC000 is 1 (light on), 0xE000 is off.
do the same for 0xC001-3



```
#include <stdbool.h>
#include <stdint.h>

#define LED0      ((volatile bool *)0xE000U)
#define TOGGLE0   ((volatile bool *)0xC000U)

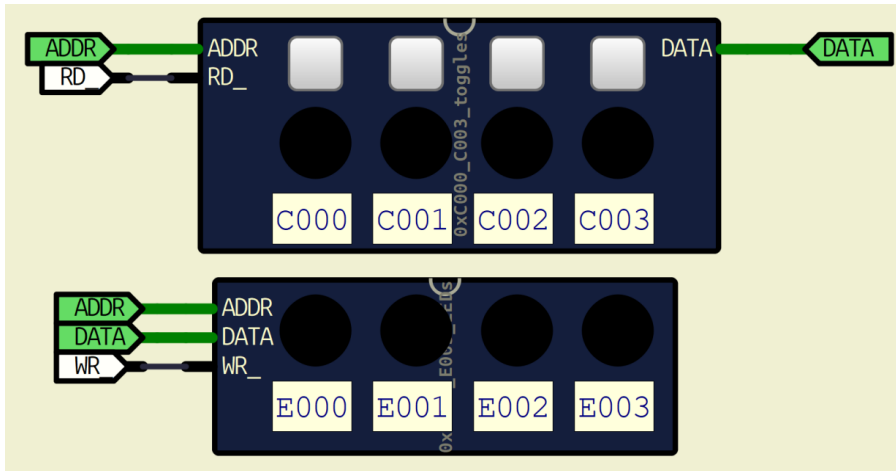
void main(void) {

    volatile bool *led0      = LED0;
    volatile bool *toggle0   = TOGGLE0;
    bool state0;

    while (true) {
        state0 = *toggle0; // read from
                           // 0xC000

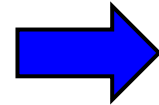
        if (state0) {
            *led0 = false;
        } else {
            *led0 = true;
        }
    }
}
```

Combine Read and Write to IO



Loop forever:

If 0xC000 is 1 (light on), 0xE000 is off.
do the same for 0xC001-3



```
// 05-206
#include <stdbool.h>
#include <stdint.h>

#define LED0      ((volatile bool *)0xE000U)
#define TOGGLE0   ((volatile bool *)0xC000U)

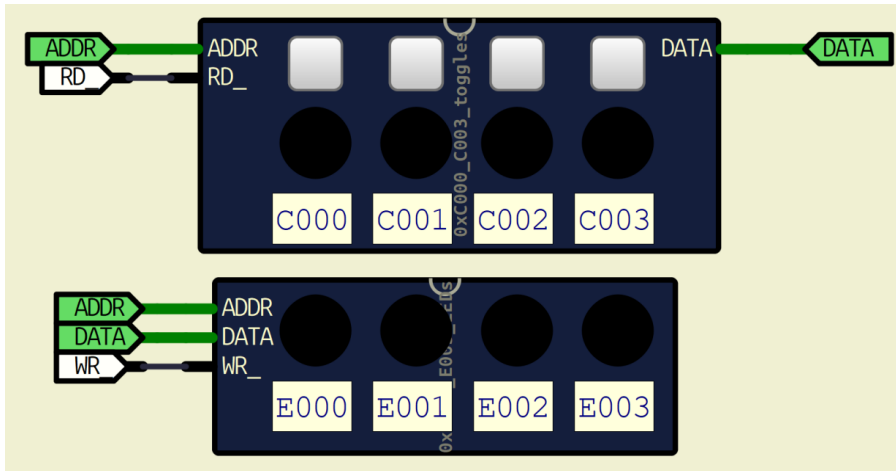
void main(void) {

    volatile bool *led0      = LED0;
    volatile bool *toggle0 = TOGGLE0;
    bool state0; // don't need it anymore

    while (true) {

        if (*toggle0) {
            *led0 = false;
        } else {
            *led0 = true;
        }
    }
}
```

Combine Read and Write to IO



Loop forever:

If 0xC000 is 1 (light on), 0xE000 is off.
do the same for 0xC001-3

Incomplete: we haven't touched 0xC001-3 and 0xE001-3 at all. We'll come back to it shortly.

// 05-206

```
#include <stdbool.h>
#include <stdint.h>
```

```
#define LED0      ((volatile bool *)0xE000U)
#define TOGGLE0   ((volatile bool *)0xC000U)
```

```
void main(void) {
```

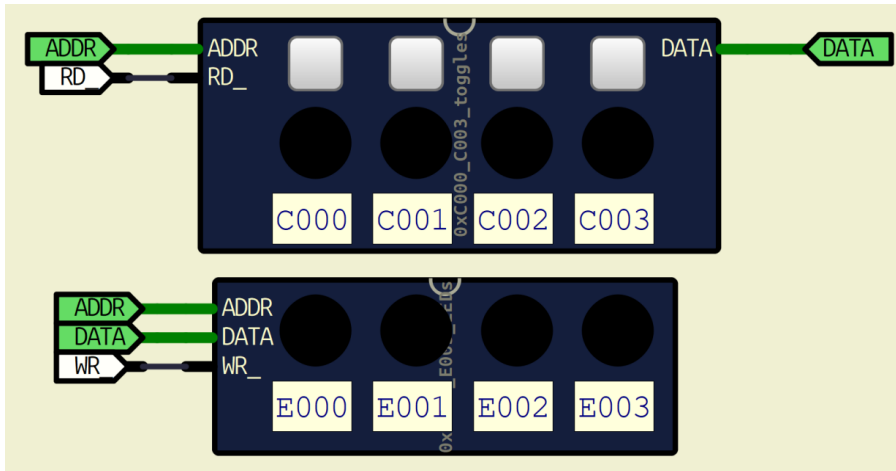
```
    volatile bool *led0      = LED0;
    volatile bool *toggle0   = TOGGLE0;
    bool state0; // don't need it anymore
```

```
    while (true) {
```

```
        if (*toggle0) {
            *led0 = false;
        } else {
            *led0 = true;
        }
```

```
    }
```

Combine Read and Write to IO



if-else is a type of *flow-control statement*:

```
if (expression) {
    // this will run if expression is "true"
} else {
    // this will run otherwise
}

----- else part is optional, so, you can have: -----
if (expression) {
    // this will run if expression is "true"
}
```

// 05-206

```
#include <stdbool.h>
#include <stdint.h>
```

```
#define LED0      ((volatile bool *)0xE000U)
#define TOGGLE0   ((volatile bool *)0xC000U)
```

```
void main(void) {
```

```
    volatile bool *led0      = LED0;
    volatile bool *toggle0 = TOGGLE0;
```

```
    while (true) {
```

```
        if (*toggle0) {
            *led0 = false;
        } else {
            *led0 = true;
        }
    }
```

```
}
```


Combine Read and Write to IO

To read 0xC000-3 and toggle 0xE000-3:

How do we apply DRY principle to do it?

Start from led (address 0xE000)

Start from switch (address 0xC000)

Do it 4 times, with increasing address:

```
if (*toggle) {
    *led = false;
} else {
    *led = true;
}
```

Let's apply what we know: while ()

```
led = LED0;
toggle = TOGGLE0;
while (toggle < 0xC004U) {
    if (*toggle) {
        *led = false;
    } else {
        *led = true;
    }
    led = led + 1;
    toggle = toggle + 1;
}
```

```
// 05-206
#include <stdbool.h>
#include <stdint.h>

#define LED0      ((volatile bool *)0xE000U)
#define TOGGLE0   ((volatile bool *)0xC000U)

void main(void) {

    volatile bool *led0      = LED0;
    volatile bool *toggle0   = TOGGLE0;

    while (true) {

        if (*toggle0) {
            *led0 = false;
        } else {
            *led0 = true;
        }

    }
}
```

Combine Read and Write to IO

To read 0xC000-3 and toggle 0xE000-3:

```
led = LED0;
toggle = TOGGLE0;
while (toggle < 0xC004U) {
    if (*toggle) {
        *led = false;
    } else {
        *led = true;
    }
    led = led + 1; // a bit DRY here:
    toggle = toggle + 1; // led & toggle
} // move together
```

```
led = LED0;
toggle = TOGGLE0;
uint8_t i = 0; // extract movement to index
while (i < 4) { // while() runs 4 times
    if (*(toggle + i)) {
        *(led + i) = false;
    } else {
        *(led + i) = true;
    }
    i = i + 1; // move to the next address
}
```

```
// 05-206
#include <stdbool.h>
#include <stdint.h>

#define LED0      ((volatile bool *)0xE000U)
#define TOGGLE0   ((volatile bool *)0xC000U)

void main(void) {

    volatile bool *led0      = LED0;
    volatile bool *toggle0 = TOGGLE0;

    while (true) {

        if (*toggle0) {
            *led0 = false;
        } else {
            *led0 = true;
        }
    }
}
```

Combine Read and Write to IO

To read 0xC000-3 and toggle 0xE000-3:

// 05-206 (OLD)

```
#include <stdbool.h>
#include <stdint.h>

#define LED0      ((volatile bool *)0xE000U)
#define TOGGLE0   ((volatile bool *)0xC000U)

void main(void) {

    volatile bool *led0      = LED0;
    volatile bool *toggle0 = TOGGLE0;

    while (true) {

        if (*toggle0) {
            *led0 = false;
        } else {
            *led0 = true;
        }
    }
}
```

// NEW

```
#include <stdbool.h>
#include <stdint.h>

#define LED0      ((volatile bool *)0xE000U)
#define TOGGLE0   ((volatile bool *)0xC000U)

void main(void) {
    volatile bool *led      = LED0;
    volatile bool *toggle = TOGGLE0;

    while (true) {

        uint8_t i = 0;
        while (i < 4) {
            if (*(toggle + i)) {
                *(led + i) = false;
            } else {
                *(led + i) = true;
            }
            i = i + 1;
        }
    }
}
```

Combine Read and Write to IO

To read 0xC000-3 and toggle 0xE000-3:

Now, look at:

```
uint8_t i = 0;
while (i < 4) {
    if (*(toggle + i)) {
        *(led + i) = false;
    } else {
        *(led + i) = true;
    }
    i = i + 1;
}
```

while and its code inside is called a **while** loop.

C provides statement that is **99% equivalent** to:

```
F1;
while (F2) {
    // code...
    F3;
}
```

With a statement called **for**. So, instead of writing the above **while**, we write:

```
for (F1; F2; F3) {
    // code...
}
```

Apply this to our code, we got:

```
for (uint8_t i = 0; i < 4; i = i + 1) {
    if (*(toggle + i)) {
        *(led + i) = false;
    } else {
        *(led + i) = true;
    }
}
```

This statement is casually called a **for** loop:

F1 is called **initialization** – sets up the condition before looping
F2 is called **condition** – checks whether loop should continue
F3 is called **increment** (bad name) – update loop variable.

Combine Read and Write to IO

To read 0xC000-3 and toggle 0xE000-3:

```
// 05-207
#include <stdbool.h>
#include <stdint.h>

#define LED0      ((volatile bool *)0xE000U)
#define TOGGLE0 ((volatile bool *)0xC000U)

void main(void) {
    volatile bool *led      = LED0;
    volatile bool *toggle = TOGGLE0;

    while (true) {

        for (uint8_t i = 0; i < 4; i = i + 1) {
            if (*(toggle + i)) {
                *(led + i) = false;
            } else {
                *(led + i) = true;
            }
        }
    }
}
```

let's see the **[05_207]** toggle switches to lights



Difference between while and for

The 1% difference between **for** and **while**:

C provides a *jump statement* called **continue**.

continue means "skip the rest of current iteration (loop)"

Example: You want to write the following:

```

0x4000 with 0x00
0x4001 with 0x01
0x4002 with 0x02
0x4003 with 0x03
0x4004 with 0x04
0x4005 with 0x05
    skip 0x4006
0x4007 with 0x07
0x4008 with 0x08
0x4009 with 0x09

```

The difference:

- **continue** in **while** - *skip everything* - check **condition** - if **condition** is true, start the loop again.
- **continue** in **for** - *skip everything* - **do the increment** - check **condition** - if **condition** is true, start loop again

common code:

```

#define A0 ((volatile uint8_t *)0x4000U)
uint8_t *a = A0;

```

using while (FAILURE):

```

uint8_t i = 0;
while (i < 10) {
    if (i == 6) { // if i equals 6
        continue; // skip the rest
    }
    *(a + i) = i;
    i = i + 1; // incrementing i is
              // skipped as well
}

```

using for (SUCCESS): increment is always done regardless of "continue"

```

for (uint8_t i = 0; i < 10; i=i+1) {
    if (i == 6) { // if i equals 6
        continue; // skip the rest
    }
    *(a + i) = i;
}

```



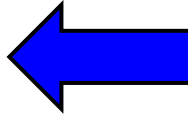
Difference between while and for

common code:

```
#define A0 ((volatile uint8_t *)0x4000U)
uint8_t *a = A0;
```

using **while (FAILURE)**:

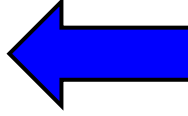
```
uint8_t i = 0;
while (i < 10) {
    if (i == 6) { // if i equals 6
        continue; // skip the rest
    }
    *(a + i) = i;
    i = i + 1; // incrementing i is
              // skipped as well
}
```



This code will get stuck when **i** gets to 6 and will never leave the while loop. because **i** is never updated when **i** is 6. continue makes it skip line that increments i

using **for (SUCCESS)**: increment is always done regardless of "continue"

```
for (uint8_t i = 0; i < 10; i=i+1) {
    if (i == 6) { // if i equals 6
        continue; // skip the rest
    }
    *(a + i) = i;
}
```



When **i** gets to 6, continue is triggered. after continue, assignment **i = i + 1** is performed, so i becomes 7 before running the left loop.



Difference between while and for

common code:

```
#define A0 ((volatile uint8_t *)0x4000U)
uint8_t *a = A0;
```

using while (FIXED):

```
uint8_t i = 0;
while (i < 10) {
    if (i == 6) { // if i equals 6
        i = i + 1; // increment i
        continue; // then restart loop
    }
    *(a + i) = i;
    i = i + 1; // incrementing i is
              // skipped as well
}
```

Can we “fix” the while version?

Yes, easily.

Increment `i` before `continue`, but it isn’t DRY.
there are **2** occurrences of `i = i + 1;`

Does this mean `for` is always better than `while`?

No. It just means that you should use the statement that fits the purpose of your code.

For this particular case, `for` happens to solve this problem more elegantly than `while`.

Can I see all this in action? `while` really got stuck?

Yes, look at [\[05-208\]](#)

I run `for` first, writing to `0x4000-0x4009`.

I run “fixed while” writing to `0x4010-0x4019`

I run “broken while” writing to `0x4020-0x4029`.

You will see that `0x4026-0x4029` aren’t written.



There are 2 "versions" of `while` loop

1. check the condition first before executing the code inside:

```
while (condition) {  
    // code may not run at all  
}
```

2. run the code inside first, and check the condition whether the loop should continue:

```
do {  
    // code will run 1+ times  
} while (condition);
```

Which one to use? Depends. See examples:

Example 1: check if some device is busy, if it is, our code cannot continue:

```
while (*DEVICE_BUSY) {  
    // no need to run if device is not busy  
}
```

Example 2: waiting for a keypad input, keypad input will be 0 if no key has been pressed.

```
do {  
    key = *KEYPAD;  
} while (key > 0); // 0 = no keypress yet
```

If we were to do example 2 using method 1, it gets ugly:

```
key = 0; // unnecessary assignment to enter loop  
while (key > 0) {  
    key = *KEYPAD;  
}
```



Cascaded `if-else`

A customer at **BITKA Exchange** can be of class *platinum*, *gold*, *silver*, or *regular*, depending on the number of bitcoins `b` they have:

- A *platinum* customer has ≥ 20 bitcoins
- A *gold* customer has ≥ 10 bitcoins
- A *silver* customer has ≥ 5 bitcoins
- Otherwise, they are a *regular* customer

```
if (b >= 20) {
    class = PLATINUM;
} else {
    if (b >= 10) {
        class = GOLD;
    } else {
        if (b >= 5) {
            class = SILVER;
        } else {
            class = REGULAR;
        }
    }
}
```

The code on the left is ugly and is very hard to read. So, we use a structure called *cascaded if-else*.

We allow `else` to be followed immediately by `if`.

The left side re-written in *cascaded if-else*:

```
if (b >= 20) {
    class = PLATINUM;
} else if (b >= 10) {
    class = GOLD;
} else if (b >= 5) {
    class = SILVER;
} else {
    class = REGULAR;
}
```

Cascaded if-else is considered a good coding style.

intentionally left blank

Our Binary Support in BareMetal-C class

Recall that **C99 Standard** only supports decimal, octal, and hexadecimal

- Any number that begins with 0 is an octal (base-8) number
Example: $v = 0123 = 0123_8 = 1 \times 8^2 + 2 \times 8 + 3 = 64 + 16 + 3 = 83_{10}$
- Any number that begins with [1-9] is a decimal (base-10) number
Example: $v = 83 = 83_{10}$
- Any number that begins with 0x is a hexadecimal (hex = base-16) number
Example: $v = 0x53 = 5 \times 16 + 3 = 80 + 3 = 83_{10}$

We built a MACRO for binary. It's *non-standard*, and only supports 8-bit:

```
#include "baremetal_binary.h" // note that we use " . " instead of < . >

uint8_t v = B8(01010011); // v will be 010100112 = 0x53 = 83
uint8_t s = B8(11000001); // s will be 110000012 = 0xC1 = 193

// You cannot put variables or expressions in B8, only 0's & 1's

uint8_t t = B8(v - s); // silently broken, returns garbage
uint8_t t = B8(00000001+00100000); // silently broken, returns garbage
```

Our Delay Support in BareMetal-C class

We've also built a *library function* for `delay` we used earlier, called `baremetal_delay`

```
#include "baremetal_delay.h" // this lets code use baremetal_delay() function
```

Remember *function prototype* earlier? A *function prototype* has 3 things:

1. function name
2. number of arguments and type of each argument.
3. the function return type

This is the *function prototype* of `baremetal_delay`:

```
void baremetal_delay(uint16_t n);
```

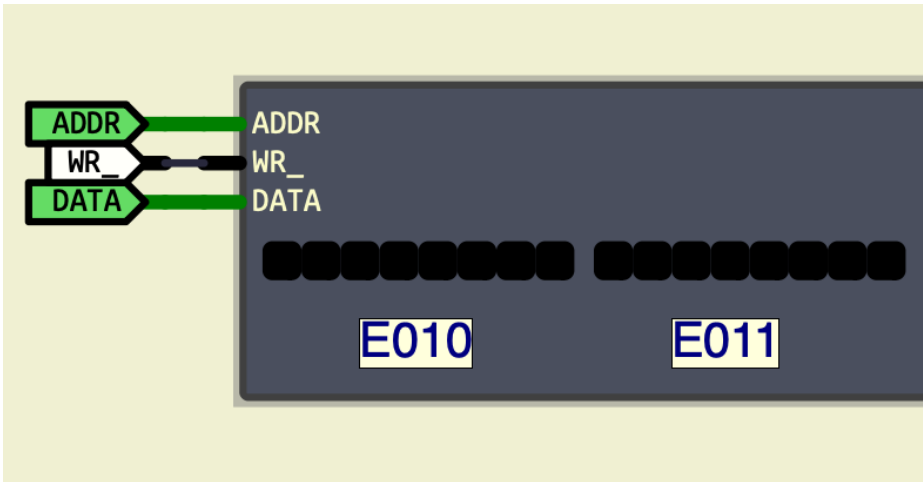
1. The function name is `baremetal_delay`
2. It (the function) takes one argument, `n`, of type 16-bit unsigned integer (`uint16_t n`).
3. It *returns* nothing. Nothingness in C is called `void`.
It tells you nothing about what `baremetal_delay` is doing.

The function argument names (`n`) is optional in C, but it is a good practice to include them to prevent confusion.

Our *coding guidelines* say you must include them. See it for more details.

A *library* is a collection of pre-written code that performs specific tasks. It typically consists of *header files* (which provide *function prototypes*, so you know how to use the functions) and *source files* (which contain the actual logic that gets compiled into your program).

Our Binary Support in BareMetal-C class



Let's see a demo:

we've added new hardware.

address: 0xE010-0xE017

for each address: 8 lights.

MSB is on the left ...

LSB is on the right

```
// 06-209
#include <stdlib.h>
#include <stdbool.h>
#include "baremetal_binary.h" // for B8()
#include "baremetal_delay.h" // for baremetal_delay

#define LED_STRIP ((volatile uint8_t *)0xE010U)

void main(void) {
    volatile uint8_t *s = LED_STRIP;

    while (true) {
        *s = B8(10101111); // X.X.XXXX
        baremetal_delay(1000);
        *s = B8(11110101); // XXXX.X.X
        baremetal_delay(1000);
        *s = B8(01010101); // .X.X.X.X
        baremetal_delay(1000);
    }
}
```

LET'S SEE IT RUN!

let's see [06_209]

Operators and Expressions

1. The Building Blocks: code is just a set of sentences.

- **Operands** are like *Nouns* – they are the **data**.
Examples: `5`, `temperature`, `count`, `0xE000U`
- **Operators** are like *Verbs* – they tell the CPU to **do** something to the data.
Examples: `+` (add), `-` (subtract), `*` (multiply), `==` (equality test)
- **Expressions** (phrases) - A *meaningful* combination of *Nouns* and *Verbs* that *reduces to a single value*. The value that an expression reduces to is called its *return value*. An expression *always returns a single value*.

<code>x + 5</code>	returns the value <code>x+5</code>
<code>(a+b)*c+d</code>	returns the value that has math notation: $(a + b) \cdot c + d$
<code>a > b</code>	returns <code>1</code> if <code>a > b</code> , and <code>0</code> otherwise
<code>count == 7</code>	rerurns <code>1</code> if <code>count</code> is equal to 7, and <code>0</code> otherwise

This is a quirk in C. Logical operators gives `int 1` for true and `int 0` for false.
C used to have only `int`. It's not until C99 that we have `bool`, `true` (1), and `false` (0).

Operators and Expressions

2. The "Scratchpad" (Pure Calculation)

Remember the CPU has an ALU (Arithmetic and Logic Unit). ALU is like a calculator.

Concept: The CPU "computes" these numbers but doesn't write anything down yet.

The Math (Arithmetic) Operators: $+$ $-$ $*$ $/$ $\%$

$5 + 3$ returns 8

$8 - 2$ returns 6

$7 * 4$ returns 28

$9 / 2$ returns 4 \leftarrow C returns integer division

$8 \% 3$ returns 2 – The $\%$ is called a *remainder*, *modulo (mod)* operator

of course, $()$ applies. When in doubt, use them! $()$

$(6*2)-(7\%2)$ returns 11 \leftarrow check!

Operators and Expressions

2. The "Scratchpad" (Pure Calculation)

Comparison Operators (The Question Mark): These operators *ask a question*.

The Result: The *answer* is always 1 (true) or 0 (false).

The List: `==` `!=` `<` `>` `<=` `>=`

`a == b` returns `1` if `a` is equal to `b`, `0` otherwise.

`c != d` returns `1` if `c` is not equal to `d`, `0` otherwise.

`e < f` returns `1` if `e` is less than `f`, `0` otherwise.

`g > h` returns `1` if `g` is greater than `h`, `0` otherwise.

`i <= j` returns `1` if `i` is less than or equal to `h`, `0` otherwise.

`k >= m` returns `1` if `k` is greater than or equal to `m`, `0` otherwise.

`==` and `!=` are called *equality operators*.

`<`, `>`, `<=`, and `>=` are called *relational operators*

Operators and Expressions

2. The "Scratchpad" (Pure Calculation)

Logical Operators (Decision Connectors)

The Tools: `&&` (AND) `||` (OR) `!` (NOT).

The Golden Rule: These only value(s) are Zero or Non-Zero. *Any non-zero number is considered true.*

<code>(a == b) && (c == d)</code>	returns 1 if (a equals b) AND (c equals d) , 0 otherwise.
<code>(e > f) (g < h)</code>	returns 1 if (e > f) OR (g < h) , 0 otherwise.
<code>!(n < 3)</code>	returns 1 if n is not less than 3 , 0 otherwise.
<code>7 && (-1)</code>	returns 1 ($7 \neq 0 \rightarrow \text{true}$) AND ($-1 \neq 0 \rightarrow \text{true}$).
<code>!0</code>	returns 1 , because $!0 \rightarrow !(false) \rightarrow 0$.
<code>!(1+2)</code>	returns 0 , because $!3 \rightarrow !(true) \rightarrow 0$.
<code>!!4</code>	returns 1 , because $!(!4) \rightarrow !(0) \rightarrow 1$.

`!` is affectionately known as the "bang!" operator

`!!` is also known as "bang-bang." *We use it to convert any non-zero number to 1 – why? Later.*

Operators and Expressions

2. The "Scratchpad" (Pure Calculation)

The *Ternary Operator* (The Conditional Calculator)

The Symbol: ? :

The Concept: An "inline `if` statement" that **returns a value**.

Structure: (Question) ? (Value if True) : (Value if False)

Examples: `(temp > 50) ? 255 : 0` // if temp > 50, returns 255, else, returns 0
 `foo ? 4 : 2` // if foo is non-zero, returns 4, else, returns 2

Why use it? It calculates a result instantly, unlike `if` which just directs traffic.

Be careful of redundancy: these 2 lines are equal:

<code>(val > 10)? 1 : 0</code>	// redundant
<code>(val > 10)</code>	// clearer

Operators and Expressions

3. The hardware "switches" (Pure Calculation)

Bytes (8-bit) vs. Switches (false is 0 / true is any other value) – the *bitwise operator*

Concept: For a byte (8-bit), `uint8_t` value 65 is 01000001 in binary (0x41 hex).

An Example: We have 8 relays (relay7 ... relay0) in a factory, and each relay can be ON (1) or OFF (0) to turn on/off 8 different lights. And the state of the relay and the control of the relay is given to us in only 1 byte (8 bits). This is enough. Let's call relay with `r`, so we have `r7...r0`.

So, suppose the hardware is hooked up to the CPU at memory address 0xFFF0:

At 0xFFF0: MSB → r7 r6 r5 r4 r3 r2 r1 r0 ← LSB

What may we want to do?

1. Check status of a particular relay whether it is ON (1) or OFF (0)
2. Turn any relay ON (1)
3. Turn any relay OFF (0)
4. Toggle any relay from ON to OFF, and OFF to ON

Operators and Expressions

3. The hardware "switches" (Pure Calculation)

The *bitwise operator*

Let's backtrack to our logic part, with symbols: \sim (NOT), $\&$ (AND), \wedge (XOR), \mid (OR).
We'll do it for just 1 bit.

The theorems are true for any value of x :

Theorem				
x	$\&$	0	$=$	0
x	$\&$	1	$=$	x
x	\mid	0	$=$	x
x	\mid	1	$=$	1
x	\wedge	0	$=$	x
x	\wedge	1	$=$	$\sim x$

← How to prove these?

Very easy. We have LHS, RHS, and x , which can only be 0 or 1.

For each theorem:

1. Set $x = 0$. Compare LHS and RHS. Equal?
2. Set $x = 1$. Compare LHS and RHS. Equal?
3. If $LHS = RHS$ for both values of x , then theorem is true.

Operators and Expressions

3. The hardware "switches" (Pure Calculation)

The *bitwise operator*

The application of the theorems:

Theorem	Application
$x \ \& \ 0 \ = \ 0$	Use & to write 0, or to block x from showing up (always shows 0): $x \ \& \ 0 \ =$ always returns 0 $x \ \& \ 1 \ =$ lets the value of x pass through
$x \ \& \ 1 \ = \ x$	
$x \ \ 0 \ = \ x$	Use to write 1: [uncommon: block x from showing up (always shows 1)] $x \ \ 0 \ =$ lets the value of x pass through $x \ \ 1 \ =$ always returns 1
$x \ \ 1 \ = \ 1$	
$x \ ^ \ 0 \ = \ x$	Use ^ to toggle x: we don't need to know the value of x to toggle it. $x \ ^ \ 0 \ =$ always returns x $x \ ^ \ 1 \ =$ always returns $\sim x$
$x \ ^ \ 1 \ = \ \sim x$	

Operators and Expressions

3. The hardware "switches" (Pure Calculation)

The *bitwise operator* – all **blue** symbols are bits. – C code is in **red**

remember the relays? RELAY_ADDR (0xFFF0): r7 r6 r5 r4 r3 r2 r1 r0

Example 1: Is relay 3 on?

relay:	r7	r6	r5	r4	r3	r2	r1	r0
&	0	0	0	0	1	0	0	0
result:	0	0	0	0	r3	0	0	0

C code

```
#define RELAY_ADDR ((volatile uint8_t *)0xFFF0U)
relay = *RELAY_ADDR;
r3status = relay & B8(00001000);
```

Example 2: Turn on relay 1

relay:	r7	r6	r5	r4	r3	r2	r1	r0
	0	0	0	0	0	0	1	0
result:	r7	r6	r5	r4	r3	r2	1	r0

```
relay = *RELAY_ADDR;
newrelay = relay | B8(00000010);
*RELAY_ADDR = newrelay;
```

Example 3: Toggle relays 2 & 6

relay:	r7	r6	r5	r4	r3	r2	r1	r0
^	0	1	0	0	0	1	0	0
result:	r7	r6'	r5	r4	r3	r2'	r1	r0

```
relay = *RELAY_ADDR;
newrelay = relay ^ B8(01000100);
*RELAY_ADDR = newrelay;
```

Example 4: Turn off relay 0

HOW? →

```
*RELAY_ADDR = *RELAY_ADDR & B8(11111110);
```

Example 5: Toggle every relay:

HOW? →

```
*RELAY_ADDR = ~(*RELAY_ADDR);
```

Operators and Expressions

3. The hardware "switches" (Pure Calculation)

More bitwise operator – the bitwise *shift operators*: \ll \gg

- Left shift: $m \ll n$
 - look at m as a bit-vector
 - shift m left by n positions, shift in 0's
 - discard bits that are shifted out
 - = $m \cdot 2^n$ if no "1" is shifted out
- Right shift: $m \gg n$
 - shift m right by n positions by:
 - arithmetic shift right (ASR)*:
 - extend MSB's
 - logical shift right (LSR)*:
 - 0's going into MSB's.
 - = $m / 2^n$ if no bits are "lost"

example: m is <code>uint8_t</code> (works the same way for other sizes)		
m	00010110	22
$m \ll 1$	00101100	44
$m \ll 3$	10110000	176
$m \ll 6$	10000000	N/A
m	11110110	246
$m \gg 1$	01111011	123
$m \gg 3$	00011110	30 (246/8)
ASR: $m \gg 3$	00011110	ASR = LSR for <code>uintN_t</code>
LSR: $m \gg 3$	00011110	

undefined behavior when ($n < 0$) or ($n > \text{number of bits in } m$)

Operators and Expressions

3. The hardware "switches" (Pure Calculation)

More bitwise operator – the bitwise *shift operators*: \ll \gg

- Left shift: $m \ll n$
 - look at m as a bit-vector
 - shift m left by n positions, shift in 0's
 - discard bits that are shifted out
 - = $m \cdot 2^n$ if no "1" is shifted out
- Right shift: $m \gg n$
 - shift m right by n positions by:
 - arithmetic shift right (ASR)*:
 - extend MSB's
 - logical shift right (LSR)*:
 - 0's going into MSB's.
 - = $m / 2^n$ if no bits are "lost"

example: m is <code>int8_t</code> (works the same way for other sizes)		
m	00010110	22
$m \ll 1$	00101100	44
$m \ll 3$	10110000	176
$m \ll 6$	10000000	N/A
m	11110110	-10
ASR: $m \gg 1$	11111011	-5
LSR: $m \gg 1$	01111011	N/A
ASR: $m \gg 3$	11111110	N/A
LSR: $m \gg 3$	00011110	N/A

Whether LSR or ASR is happening to *signed int* (e.g., `uint8_t`) is undefined by C99 standard!

Operators and Expressions

3. The hardware "switches" (Pure Calculation)

More bitwise operator – the bitwise *shift operators*: << >>

whether LSR or ASR is happening to signed int is undefined by C99 standard!

- This does not affect left shift.
- You cannot rely on right shift operator in C to do ASR or LSR on signed int
- 2 possible fixes:

1. Before shifting, keep MSB, and use & or | to guarantee ASR or LSR as needed.

Extra: Write these functions: `int8_t asr(int8_t m, uint8_t n);` and `int8_t lsr(int8_t m, uint8_t n);`

2. Never right shift (or left shift) signed integer. only << or >> on unsigned int

This is what we will do in this class.

Our coding guidelines say: "Use U for Bit-level Data"

Examples:

good: `1U << 4` // add U to make a literal unsigned

bad: `1 << 4`

Operators and Expressions

4. The "Save Button" (Memory Modification – also called "side-effect")

So far, we've been calculating, and now we will commit the Result

Concept: The CPU takes the answer from the scratchpad and writes it into RAM.

The Operator: `=` (Assignment).

Crucial Distinction:

`==` is a **Question** ("Are they equal?").

`=` is a **Command** ("Make it equal!").

Assignment is an Expression too!

Example: `x = 10` is an expression.

1. Write `10` to memory (The Side Effect).
2. The expression evaluates to `10` (The Result)

Why this matters: This is why `a = b = c = 10` works \rightarrow `a = (b = (c = 10))`

Operators and Expressions

4. The "Save Button" (Memory Modification – also called "side-effect")

So far, we've been calculating, and now we will commit the Result

Concept: The CPU takes the answer from the scratchpad and writes it into RAM.

The Operator: `=` (Assignment).

Crucial Distinction:

`==` is a **Question** ("Are they equal?").

`=` is a **Command** ("Make it equal!").

Assignment is an Expression too!

Example: `x = 10` is an expression.

1. Write `10` to memory (The Side Effect).
2. The expression evaluates to `10` (The Result)

Why this matters: This is why `a = b = c = 10` works \rightarrow `a = (b = (c = 10))`

Operators and Expressions

4. The "Save Button" (Memory Modification – also called "side-effect")

Lazy Operators: += -= *= /= %= &= |= ^= <<= >>=

Concept: Assign a new value by modifying it from the old value.

a += 5	means	a = a + 5;
a -= 2	means	a = a - 2;
a &= B8(10000000)	means	a = a & B8(10000000);
a ^= B8(11111111)	means	a = a ^ B8(11111111);
a <<= 3	means	a = a << 3;

Do not use *lazy operators* in this class. It makes the code harder to read.

Operators and Expressions

4. The "Save Button" (Memory Modification – also called "side-effect")

Increment/Decrement operators: ++ --

Concept: Increment or decrement a modifiable *lvalue* (*locator-value*: variable representing a memory location).

There are 2 variants: *pre*-increment/decrement and *post*-increment/decrement.

Example: suppose *x* is 5

<i>x</i> ++	returns 5, and <i>x</i> becomes 6 (<i>x</i> is incremented by 1 <i>post-evaluation</i>)
++ <i>x</i>	returns 6, and <i>x</i> becomes 6 (<i>x</i> is incremented by 1 <i>pre-evaluation</i>)
<i>x</i> --	returns 5, and <i>x</i> becomes 4 (<i>x</i> is decremented by 1 <i>post-evaluation</i>)
-- <i>x</i>	returns 4, and <i>x</i> becomes 4 (<i>x</i> is decremented by 1 <i>pre-evaluation</i>)
(<i>x</i> + <i>y</i>)++	invalid. The result of (<i>x</i> + <i>y</i>) is a temporary value (<i>rvalue</i>), not a modifiable <i>lvalue</i> (variable with a memory address)
<i>x</i> = <i>x</i> ++	undefined behavior (UB) - modifying an object <i>x</i> <i>more than once</i>

Do not use *increment or decrement operators* in this class. Be explicit.

Good Example: *i* = *i* + 1

Bad Example: *i*++

Operators and Expressions

5. Operator precedence:

Compared to mathematics, C has many more operators. They are not evaluated equally.

Golden Rule: When unsure, use () – guarantees that stuff inside is evaluated first.

Example in math: $6/2 + 4*0$ evaluates to 3

Example in C: `temp > 50 * 7 - 23 & 0x80 || far` evaluates to ???

The C99 standard (and our documents) tells you exactly what it will evaluate to. But better use () to make them easier to read.

Bad example: `temp > 50 * 7 - 23 & 0x80 || far` // what???

"Better" example: `((temp > 50) * 7 - (23 & 0x80)) || far` // better, but still bad
// but order is clear

See [BareMetal-C/docs-students/C99-operator-precedence.md](#) for the complete table.



Operators and Expressions

Summary:

Expressions in C *always returns a single value*.

Examples: `5 + 4` returns `9`
`7 > 3` returns `1`
`1U << 3` returns `8`
`x = 6` returns `6`

Some expressions just calculate.

Some expressions "update the memory" – aka side-effects.

The only way we will *update the memory* in this class is using `=`

Do not use lazy operators (`+=` `-=` ...)

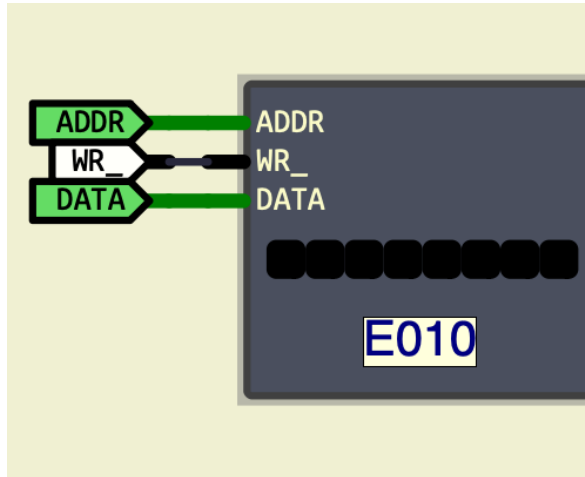
Do not use increment/decrement operators (`++` `--`)

Example: `x = x + 1;`
`larger = (a > b)? a : b;`

intentionally left blank

Principle of programming: MVC

Putting it all together: a read-to-clear (read2clear, rd2clr) switch:

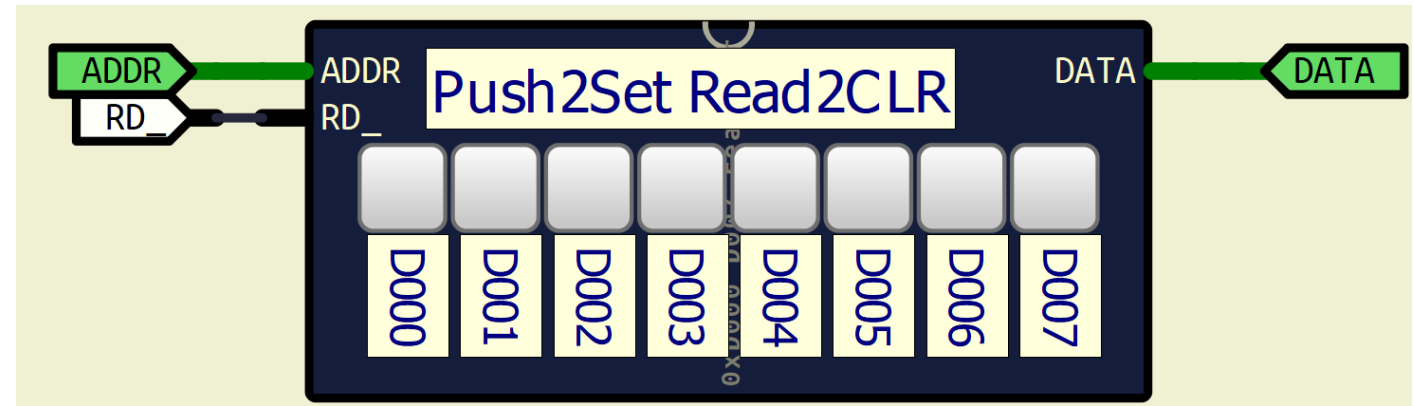


Existing Hardware:

A *write-only* LED-strip. You *cannot* read from **E010** to see light status.

Writing `uint8_t` data to **E010** lets "binary" data show up on 8 LEDs. The left LED is MSB. The right is LSB. So writing **0x86 = B8(10000110)**, to **E010**, we will have lights like this:

X-----XX-



New Hardware (Read to Clear switches:

A *read-only* set of buttons, mapped to address **0xD000-D007**. The buttons are *read-to-clear*, you can read it only once.

Example: For **D004** button:

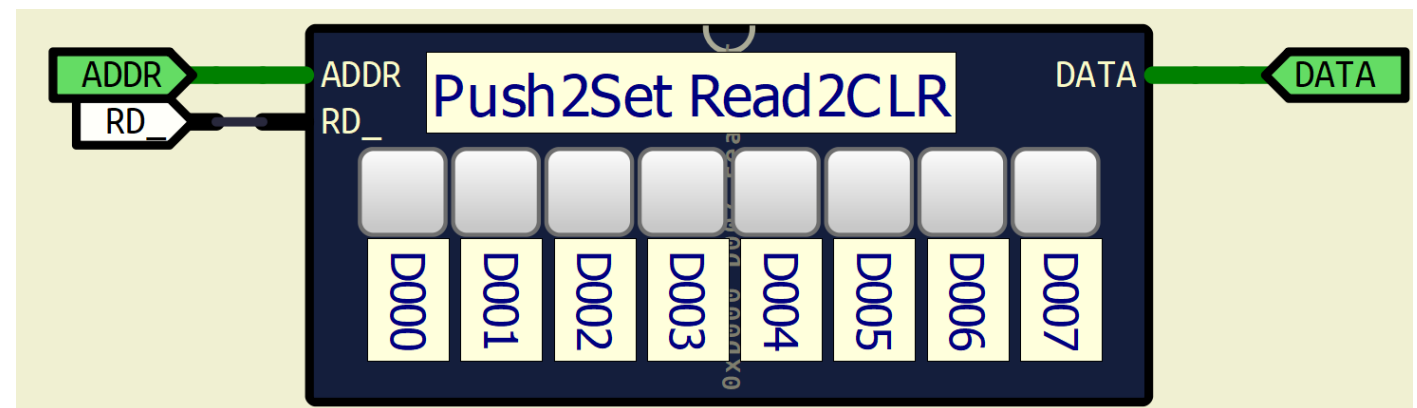
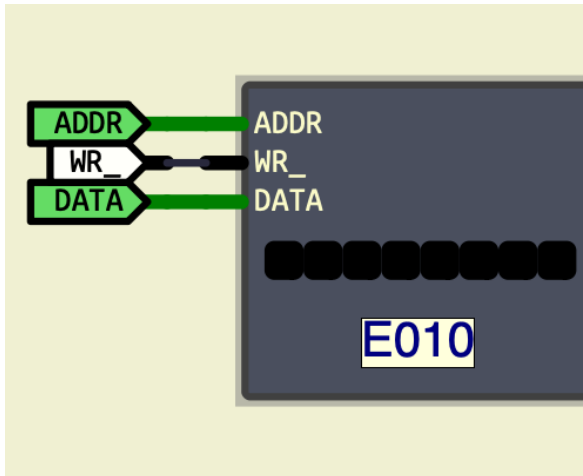
- Repeated presses to **D004** before reading it is similar to one press.
- The first read to **D004** after the button is pressed, the LSB of **D004** will be **1**. Other bits are *undefined*.
- Subsequent reads (without more button press) to **D004** yields LSB of **0**. Other bits are *undefined*.

Principle of programming: MVC

Building a pushbutton to light toggle:

We will write code that does the following:

- When program starts, all lights at **0xE010** are *on*.
- When you press any button in **0xD000-0xD007**, it will *toggle* the light *state* at the corresponding location. "*toggle*" means switch from on to off and vice-versa.
 - Press **D000** will toggle the leftmost light. Press **D007** will toggle the rightmost light.
 - The same thing for **D001** to **D006**. They run from left to right.



Principle of programming: MVC

MVC (Model-View-Controller) is a conceptual framework to organize code.

The Golden Rule: NEVER mix internal "logic" with "I/O". Separate them.

The Three Pillars:

1. **MODEL (The Truth):**

- The "State" of the system
- Pure C data structures (structs, enums) and logic.
- **Rule:** The Model **never** accesses hardware addresses directly.

2. **VIEW (The Output):**

- How the Model is presented to the world.
- Reads the Model *to* Writes to I/O (Memory Mapped Output).
- *Examples:* Turning on LEDs, drawing to screen, sending UART characters.

3. **CONTROLLER (The Input):**

- How the user influences the Model.
- Reads I/O (Memory Mapped Input) *to* Updates the Model.
- *Examples:* Reading button states, polling sensors.

Principle of programming: MVC

MVC (Model-View-Controller) Example - Tank Simulator

1. **THE MODEL (Global Variables)** The "truth" about the tank's location.

```
// Lives in RAM (0x4200)
uint8_t tank_x = 10;
uint8_t tank_y = 10;
```

2. **THE CONTROLLER (Input)** Checks hardware keys to change the variables.

```
void input_tank(void) {
    volatile uint8_t *KEYPAD = (volatile uint8_t *)0xC000U;
    if (*KEYPAD == 1) { // 'Right' key
        tank_x = tank_x + 1;
    }
}
```

3. **THE VIEW (Output)** Sends current variables to the screen.

```
void render_tank(void) {
    volatile uint8_t *SCREEN_X = (volatile uint8_t *)0xC010U;
    *SCREEN_X = tank_x; // Copy Model to Hardware
}
```

Principle of programming: MVC

MVC (Model-View-Controller) Example - Tank Simulator

4. THE GAME LOOP - The `main()` function just coordinates the parts.

```
void main(void) {  
    // 1. Setup  
    tank_x = 0;  
  
    // 2. The Loop  
    while(true) {  
        input_tank();    // Controller: Update x based on key  
        // ... game logic (e.g., check collisions) ...  
        render_tank();   // View: Draw new x to screen  
    }  
}
```

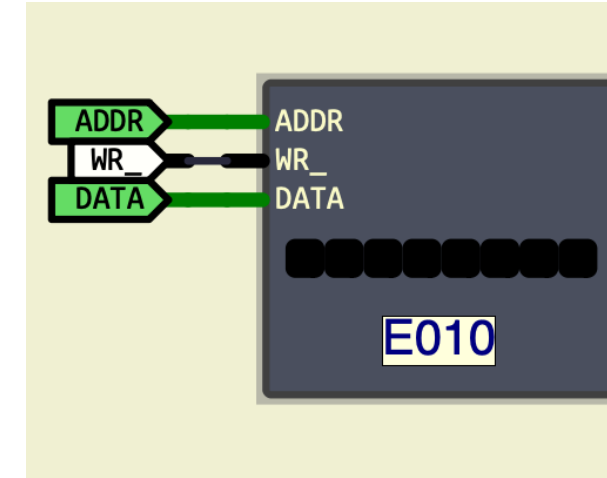
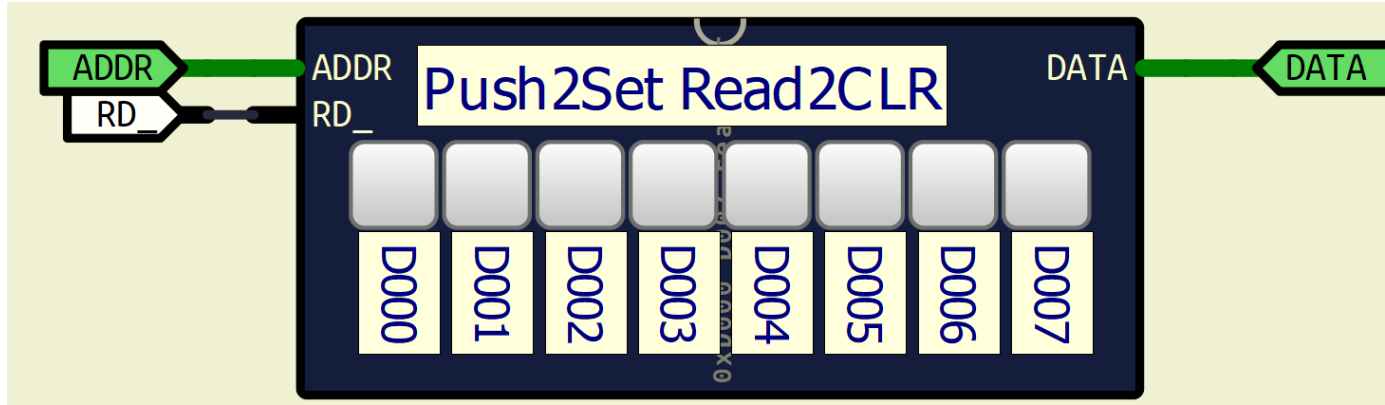
Why is MVC good?

- **Separation of Concerns:** Logic is Protected: The model is separated from hardware addresses
- **Easier Maintenance:** Changing hardware later, you only change `render_tank()`.
- **Easier Debugging:**
 - Tank moves to the wrong spot? Check the **Model**.
 - Tank is in the right spot, but screen is blank? Check the **View**.

Principle of programming: MVC

Building a pushbutton to light toggle:

- Pressing `0xD000-0xD007` toggles the corresponding light at `0xE010`.



MODEL: need one `uint8_t` data to store "state" of whether each bit is 1 or 0

VIEW: read the `state` and write it to `0xE010`

CONTROLLER: read through `0xD000 – 0xD007`, toggling corresponding bit of state if pushbutton was pressed



Principle of programming: MVC

app

comp arch

logic

Quick info about "State":

State is the "memory" of a system. It represents the specific condition of a system at a distinct moment in time.

- **Completeness:** It contains all necessary information to determine how the system behaves next. You do not need to know the entire history of the system, only its current state.
- **Transformation:** Logic determines how the system moves from the **Current State** to the **Next State** based on Input.

Key Rule: If you turn the power off and on, "**State**" is what you must reload to continue exactly where you left off.

Examples:

- **Vending Machine:** The logical state is "20 Baht inserted." It doesn't matter *how* (5+5+5+5 vs. 10+10B coins); the state is just the total.
- **Game Save File:** A snapshot containing location, HP, and inventory.
- **Bank Account Balance:** 1500 THB – doesn't matter whether it's 1000+500 or 2000-500.
- **Board Game:** A photo of a Go board + information on which stone was last played.



Principle of programming: MVC

Quick info about "State":

For software, **state** is the set of values stored in memory that defines exactly what the system is doing right now.

- **Variables = State:** In C, your global variables, static variables, and register values collectively form the program's state.
- **State Machine Logic:** Systems are defined by how they transition:

$$\text{Next State} = f(\text{Current State}, \text{Input})$$

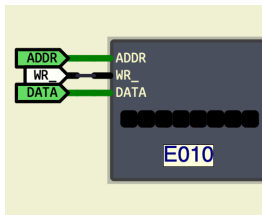
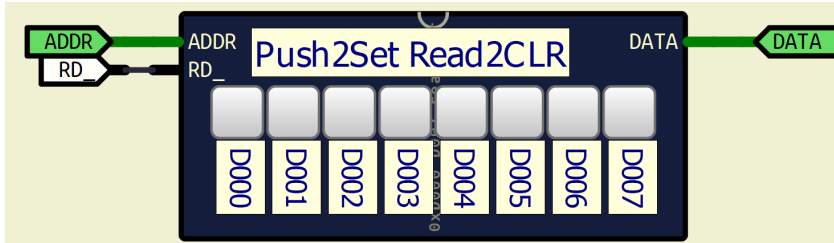
Examples:

- **Skytrain Kiosk:** `credit = 20`. If `Input = 10`, then `credit = 30`.
- **Traffic Light:** State is RED. Timer expires (Input) → State becomes GREEN
- **CPU:** The value of all the relevant registers in the CPU and the contents of the memory. (Think save game file)

Principle of programming: MVC

Building a pushbutton to light toggle:

- Pressing `0xD000-0xD007` toggles the corresponding light at `0xE010`.



MODEL: need one `uint8_t` data to store "state" of whether each bit is 1 or 0 – receives info from CONTROLLER and updates itself.

VIEW: read the `state` and write it to `0xE010`
VIEW can *only read* from MODEL

CONTROLLER: read through `0xD000 – 0xD007`, toggling corresponding bit of state if pushbutton was pressed – CONTROLLER sends info to the MODEL

```
// 07-210
#include <stdint.h>
#include <stdbool.h>
#include "baremetal_binary.h"

uint8_t state = B8(00000000); // MODEL and initializer

#define PUSHBUTTON_BASE ((volatile uint8_t *)0xD000U) // CONTROLLER
#define LED_STRIP ((volatile uint8_t *)0xE010U) // VIEW

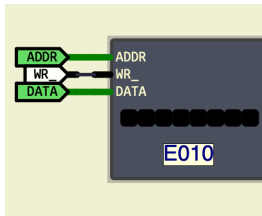
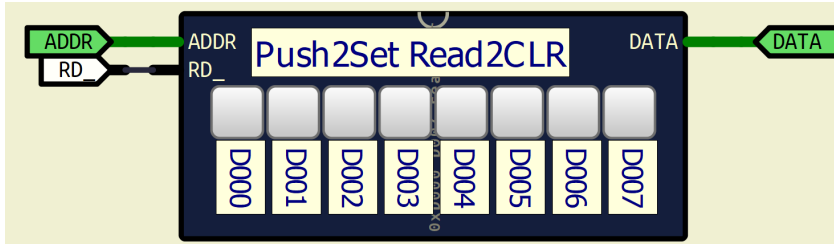
void main(void){
    uint8_t bit_mask;

    while (true) {
        // CONTROLLER, clear bit mask
        bit_mask = B8(00000000);
        for (uint8_t i = 0; i < 8; i = i + 1) {
            if (*(PUSHBUTTON_BASE + i)) {
                bit_mask = bit_mask | ( 1U << (7-i) ); // update
            }
        }
        // UPDATE MODEL
        state = state ^ bit_mask;
        // VIEW
        *LED_STRIP = state;
    }
}
```

Principle of programming: MVC

Building a pushbutton to light toggle:

- Pressing `0xD000-0xD007` toggles the corresponding light at `0xE010`.



MODEL: need one `uint8_t` data to store "state" of whether each bit is 1 or 0 – receives info from CONTROLLER and updates itself.

VIEW: read the `state` and write it to `0xE010`
VIEW can *only read* from MODEL

CONTROLLER: read through `0xD000 – 0xD007`, toggling corresponding bit of state if pushbutton was pressed – CONTROLLER sends info to the MODEL

```
// 07-210
#include <stdint.h>
#include <stdbool.h>
#include "baremetal_binary.h"

uint8_t state = B8(00000000); // MODEL and initializer

#define PUSHBUTTON_BASE ((volatile uint8_t *)0xD000U) // CONTROLLER
#define LED_STRIP ((volatile uint8_t *)0xE010U) // VIEW

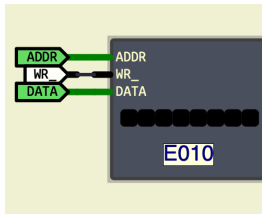
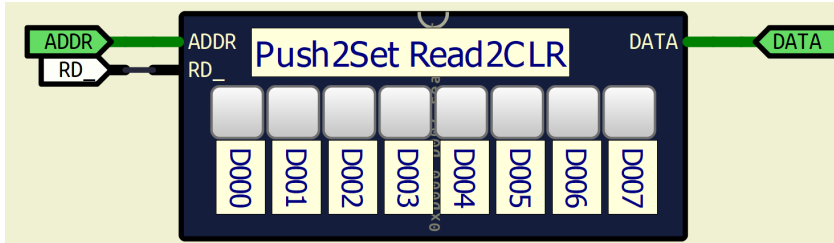
void main(void){
    uint8_t bit_mask;

    while (true) {
        // CONTROLLER, clear bit mask
        bit_mask = B8(00000000);
        for (uint8_t i = 0; i < 8; i = i + 1) {
            if (*(PUSHBUTTON_BASE + i)) {
                bit_mask = bit_mask | ( 1U << (7-i) ); // update
            }
        }
        // UPDATE MODEL
        state = state ^ bit_mask;
        // VIEW
        *LED_STRIP = state;
    }
}
```

Principle of programming: MVC

Building a pushbutton to light toggle:

- Pressing `0xD000-0xD007` toggles the corresponding light at `0xE010`.



MODEL: need one `uint8_t` data to store "state" of whether each bit is 1 or 0 – receives info from CONTROLLER and updates itself.

VIEW: read the `state` and write it to `0xE010`
VIEW can *only read* from MODEL

CONTROLLER: read through `0xD000 – 0xD007`, toggling corresponding bit of state if pushbutton was pressed – CONTROLLER sends info to the MODEL (`bit_mask`)

```
// 07-210
#include <stdint.h>
#include <stdbool.h>
#include "baremetal_binary.h"

uint8_t state = B8(00000000); // MODEL and initializer

#define PUSHBUTTON_BASE ((volatile uint8_t *)0xD000U) // CONTROLLER
#define LED_STRIP ((volatile uint8_t *)0xE010U) // VIEW

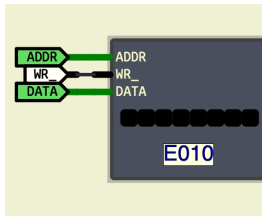
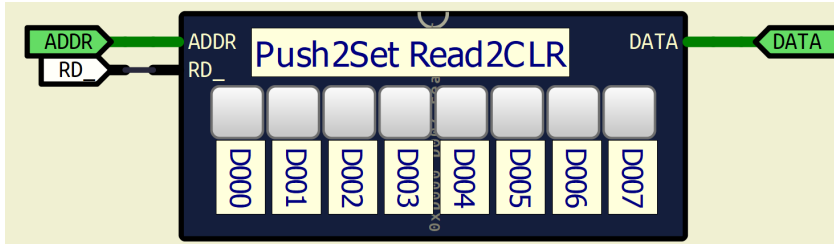
void main(void){
    uint8_t bit_mask;

    while (true) {
        // CONTROLLER, clear bit mask
        bit_mask = B8(00000000);
        for (uint8_t i = 0; i < 8; i = i + 1) {
            if (*(PUSHBUTTON_BASE + i)) {
                bit_mask = bit_mask | ( 1U << (7-i) ); // update
            }
        }
        // UPDATE MODEL
        state = state ^ bit_mask; // bit_mask sent to MODEL
        // VIEW
        *LED_STRIP = state;
    }
}
```

Principle of programming: MVC

Building a pushbutton to light toggle:

- Pressing `0xD000-0xD007` toggles the corresponding light at `0xE010`.



MODEL: need one `uint8_t` data to store "state" of whether each bit is 1 or 0 – receives info from CONTROLLER and updates itself.

VIEW: read the `state` and write it to `0xE010`
VIEW can *only read* from MODEL

CONTROLLER: read through `0xD000 – 0xD007`, toggling corresponding bit of state if pushbutton was pressed – CONTROLLER sends info to the MODEL

```
// 07-210
#include <stdint.h>
#include <stdbool.h>
#include "baremetal_binary.h"

uint8_t state = B8(00000000); // MODEL and initializer

#define PUSHBUTTON_BASE ((volatile uint8_t *)0xD000U) // CONTROLLER
#define LED_STRIP ((volatile uint8_t *)0xE010U) // VIEW

void main(void){
    uint8_t bit_mask;

    while (true) {
        // CONTROLLER, clear bit mask
        bit_mask = B8(00000000);
        for (uint8_t i = 0; i < 8; i = i + 1) {
            if (*(PUSHBUTTON_BASE + i)) {
                bit_mask = bit_mask | ( 1U << (7-i) ); // update
            }
        }
        // UPDATE MODEL
        state = state ^ bit_mask;
        // VIEW
        *LED_STRIP = state;
    }
}
```

let's see **[07_210]**

