app
comp arch
logic

04 Interrupts and stack

- New app: dot runner.
- interrupts
- how to write interrupt handlers
  - method 1: static variables (inside function)
  - method 2: global variables
- using periodic interrupt signals to move "real time" forward.
- stack
- MVC details of the dot runner
- controlling simulation speed
- controlling "real-time" speed

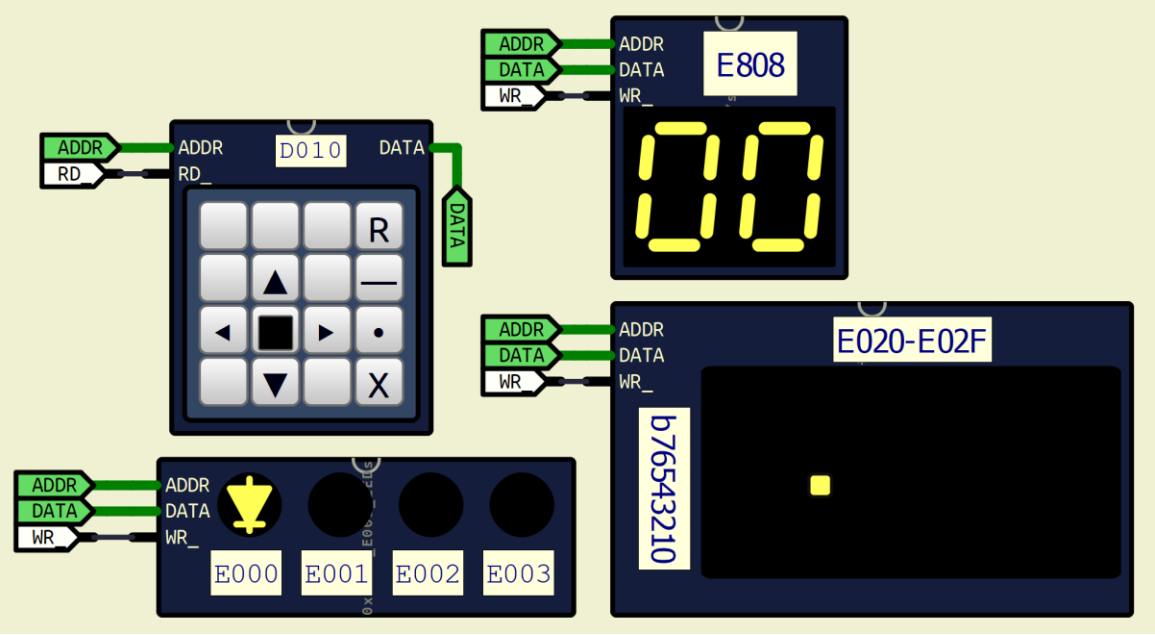
app
comp arch
logic

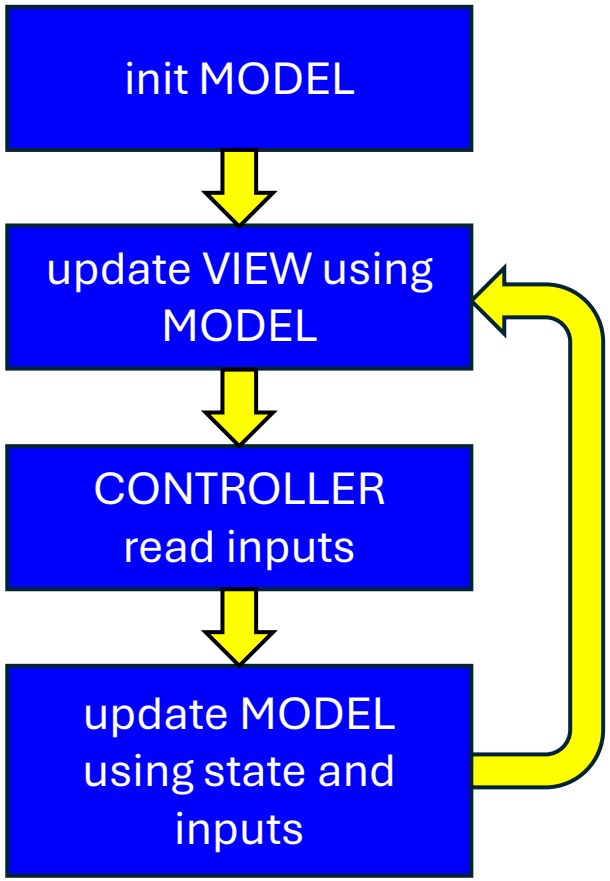
04 Interrupts and stack

How we code MVC:

The next problem is a dot runner:

- Canvas is black. Dot is yellow. Dot starts at row 4, column 7
- Pressing arrow buttons move the dot in that direction, and the dot will keep moving. But pressing ■ makes the dot stop moving.
- Running speed = 1/200mS = 5 movements/second
- If the dot hits boundary, it *wraps-around:*
  - Moving left to leftmost column → next position is rightmost column, and vice-versa.
  - Moving up to top row → next position is bottom row, and vice-versa.

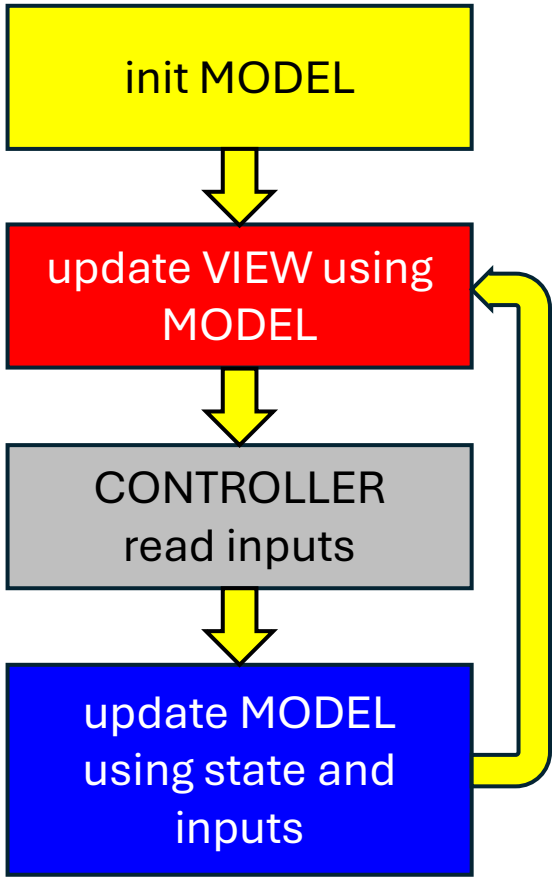


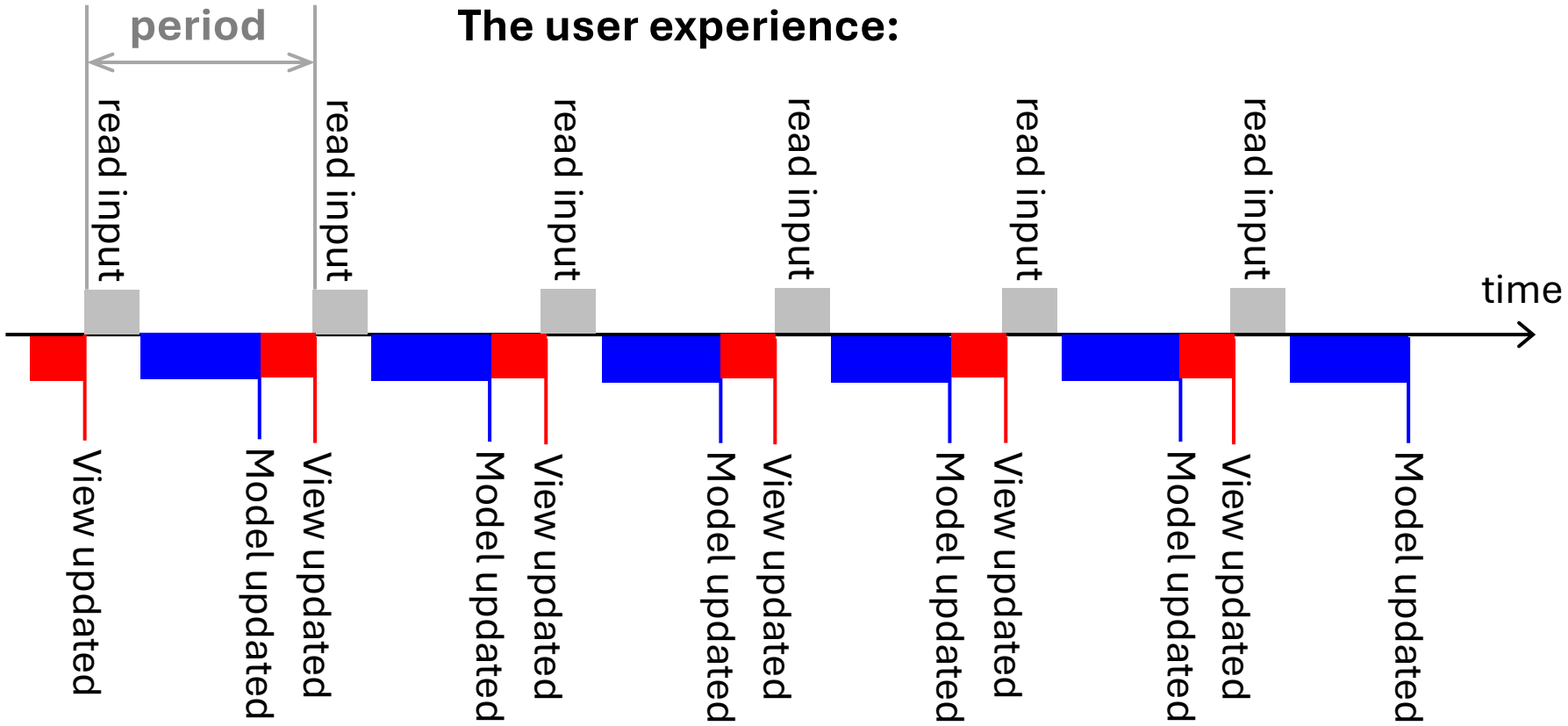


04 Interrupts and stack

There are 2 problems:

1. **Cannot fix period length:** changing system spec (clock freq, memory speed) changes period.
2. **Cannot curb period variations:** M, V, and C may take different times to compute, depending on the situation. Example: adding a new enemy, which doesn't happen every period.

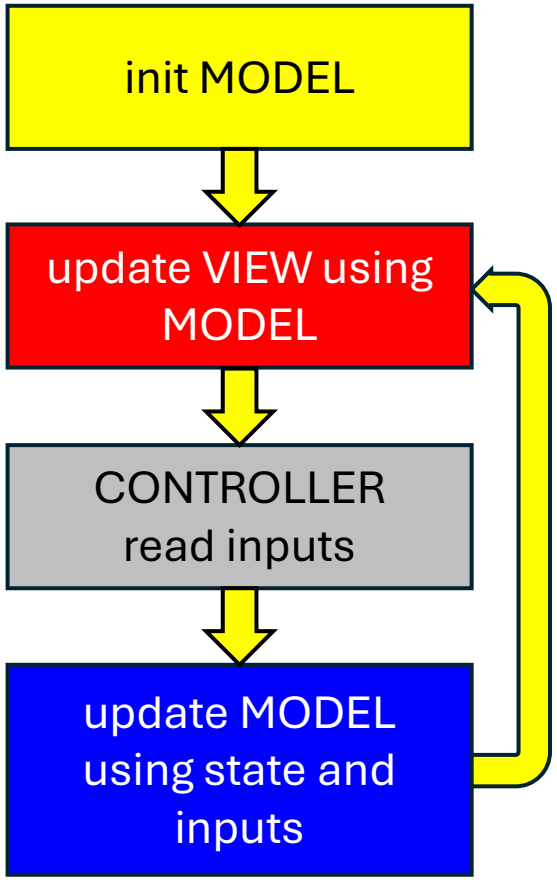




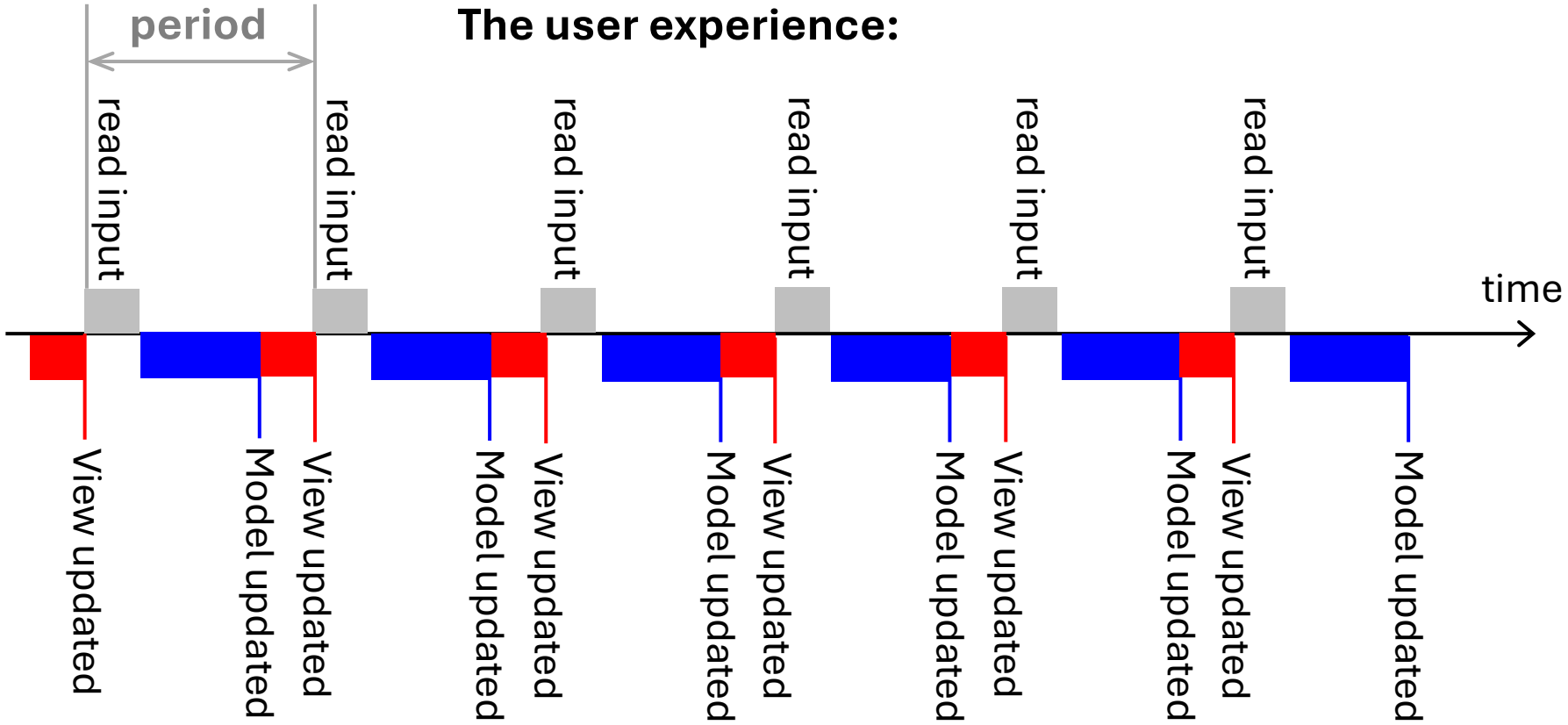
04 Interrupts and stack

The fix is backwards: We start out from period that we want:

- Changing system spec (clock freq, memory speed) should *not* change period.
- Different times to run controller, model, view should *not* change period.
- We need to give some "slack" to make sure worst case (slow CPU, long code) still works.
- "period" must be long enough for worst-case-scenario



The user experience:

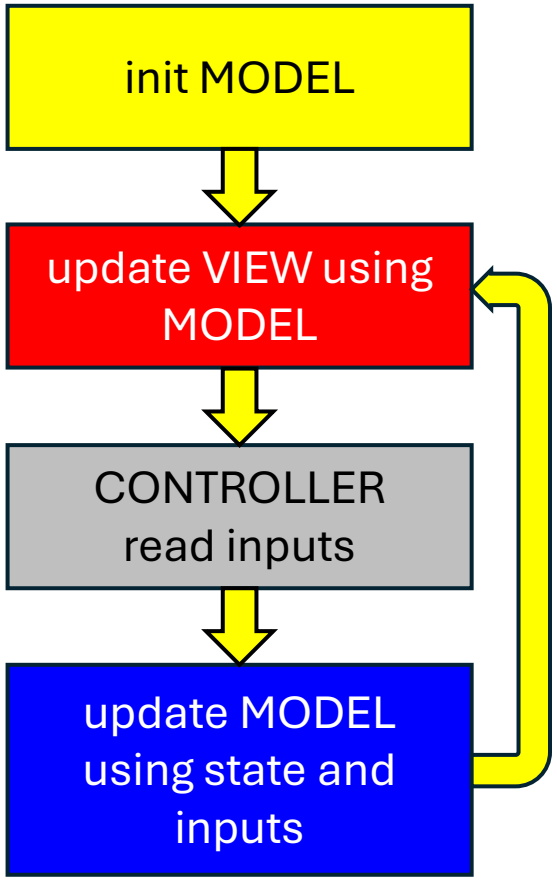


04 Interrupts and stack

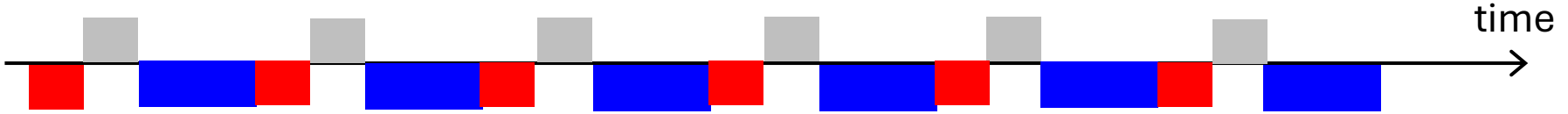
What "period" to use for interactive, constant time, program?

- **~25-30** times per second: synchronizes with TV (25-30 fps)
- **~60** times per second: high fluidity
- **>120** times per second: competitive gaming, e-sports
- **tradeoffs:** shorter period means more fluid UX/UI, but shorter per-cycle computation budget

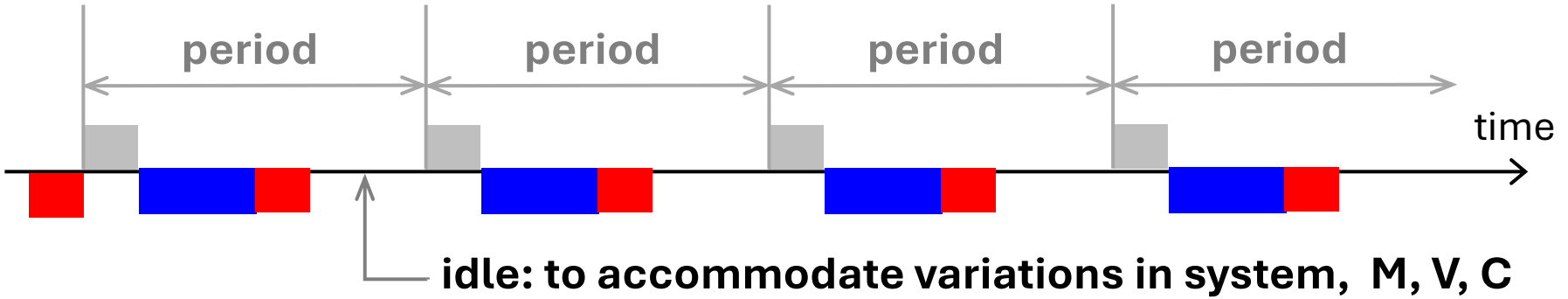
We also need to introduce "safety factor" (no-compute time) to accommodate MVC compute variations.



Before: Going as fast as we can



After: Running at pre-defined period, with some safety factor.



04 Interrupts and stack

app
comp arch
logic

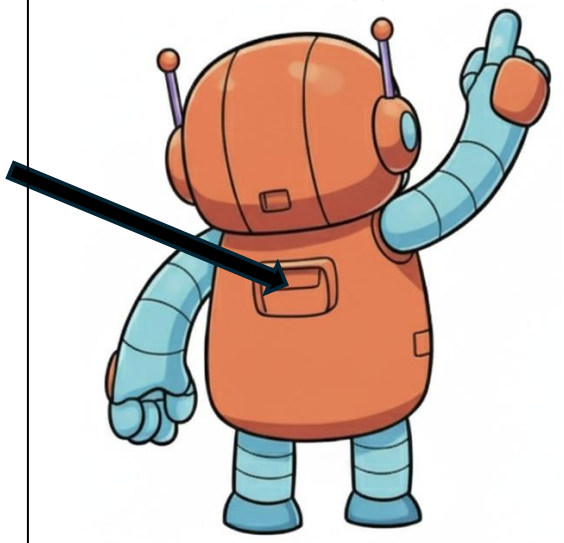
How do we enforce the desired "period"? Look at slide set 01:

There are **emergency** buttons on back of robot. Pushing them makes robot stop running programs.

These buttons are called *interrupts*. Your `main` (or anything called by `main`)will stop executing, and an *interrupt handler* will run instead. *interrupt handler* is just another function.

After *interrupt handler* finishes (return), `main` will resume.

Mr. Z80



conceptual

app
comp arch
logic

04 Interrupts and stack

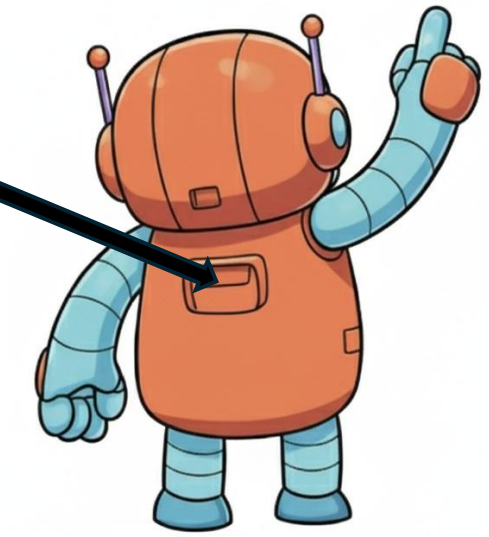
How do we enforce the desired "period"? Look at slide set 01:

There are **emergency** buttons on back of robot. Pushing them makes robot stop running programs.

These buttons are called *interrupts*. Your `main` (or anything called by `main`)will stop executing, and an *interrupt handler* will run instead. *interrupt handler* is just another function.

After *interrupt handler* finishes (return), `main` will resume.

Mr. Z80



There's a signal called **NMI**_ on the Z80 package.

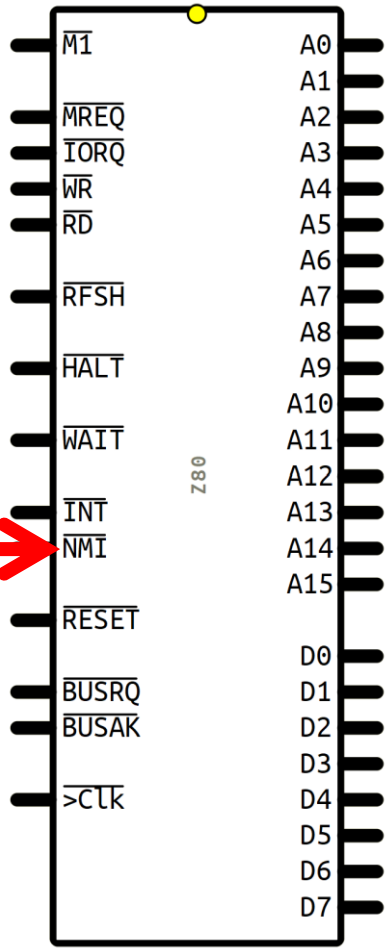
- NMI stands for *Non-Maskable-Interrupt*.
- *Non-Maskable* means software cannot ignore it.

NMI_ pin is currently hooked up to a pushbutton.

We will hook it up to a "period-generator" (a clock-generator running at low frequency).

Every time NMI_ has a *negative edge* (1→0), interrupt is triggered. An *interrupt handler* will run.

Z80



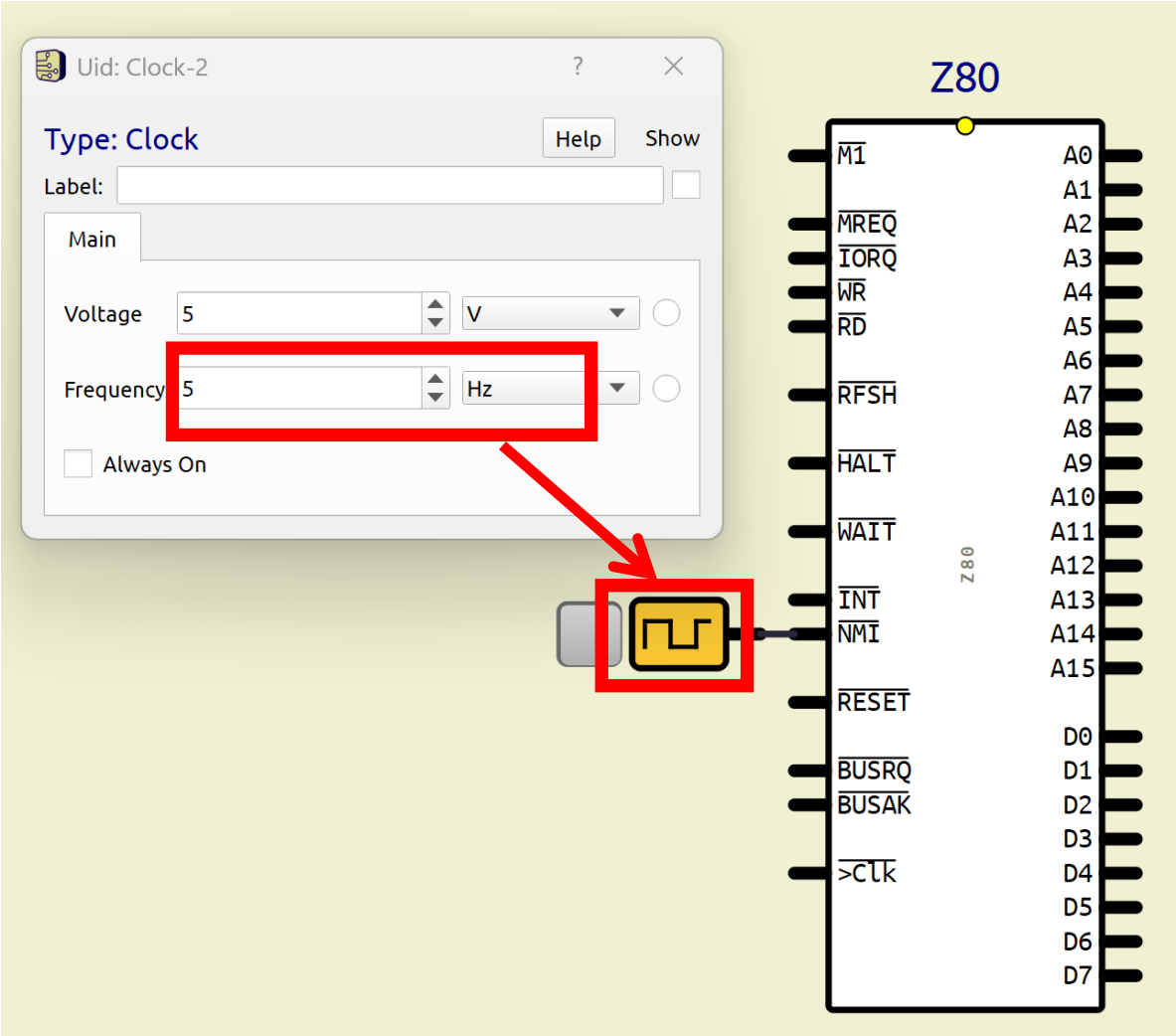
conceptual

actual

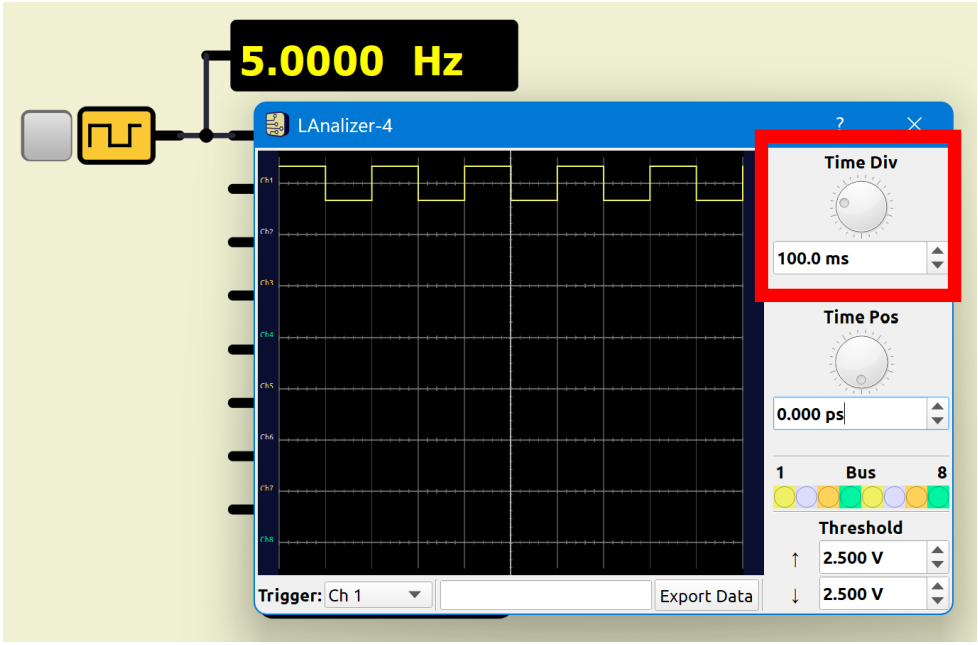
04 Interrupts and stack

app
comp arch
logic

How we enforce the desired "period":



schematic



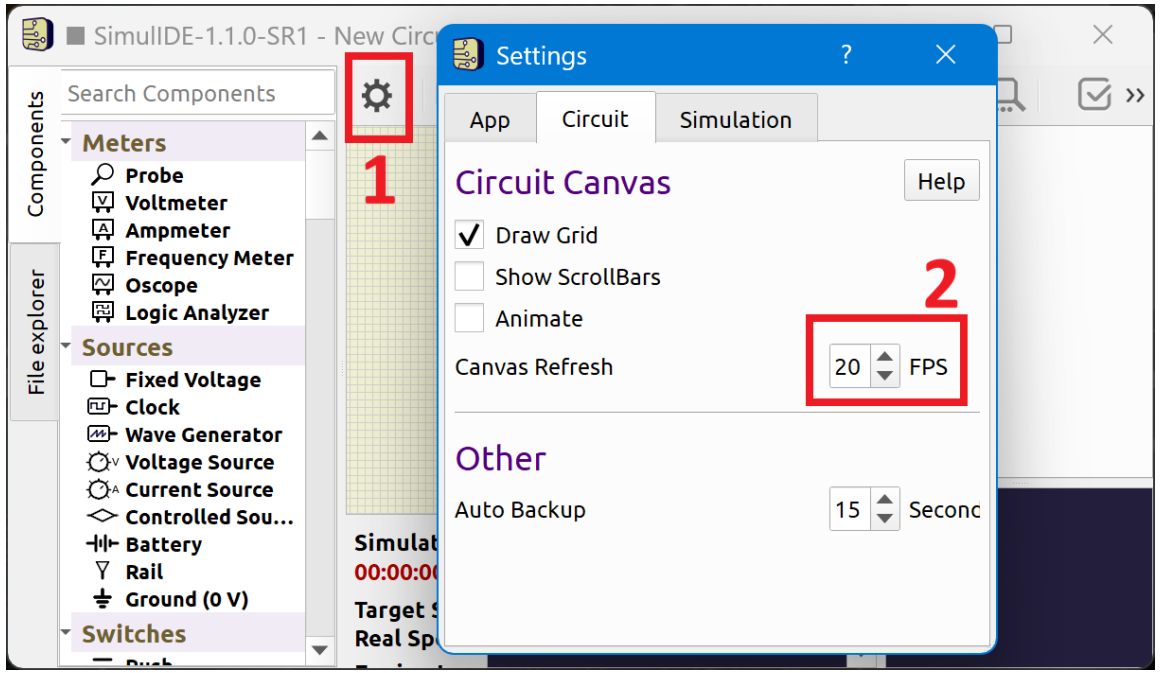
signal

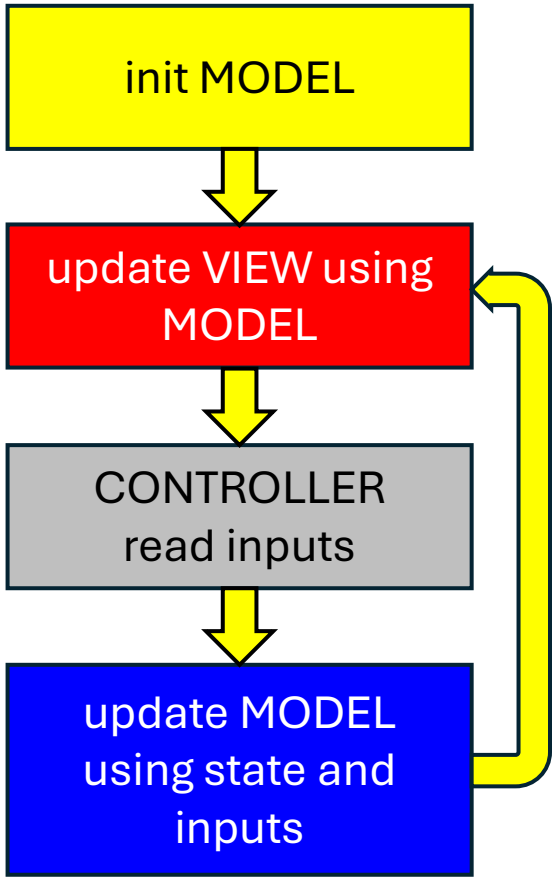
- distance between each gray vertical bar is 100mS.
- so, the period is 200mS → 5Hz
- *interrupt* is triggered at every negative edge (1 → 0)

04 Interrupts and stack

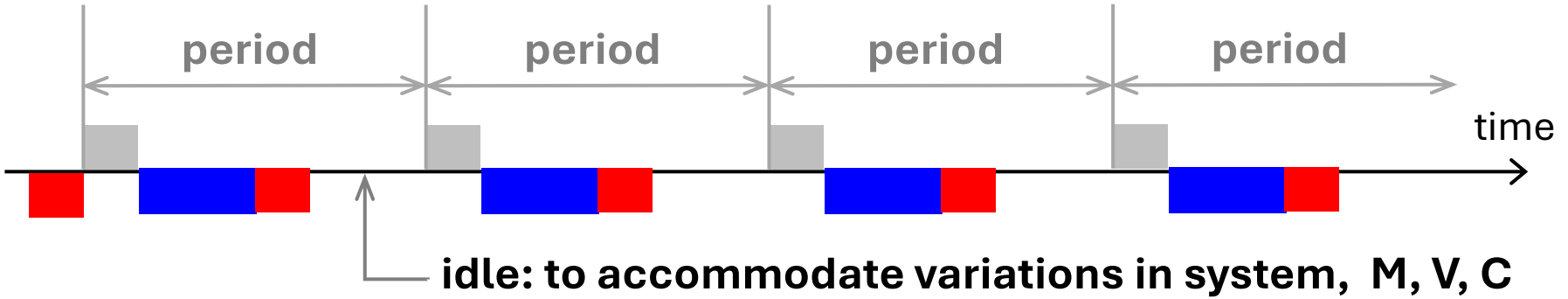
SimulIDE supports different FPS (frames per second) Canvas

- You can control the sim update rate here.
- Should be in sync with design period
  20FPS = 50ms per update → update view every 50·N ms like 50, 100, 150, 200, ...





After: Running at pre-defined period, with some safety factor.



app
comp arch
logic

04 Interrupts and stack

How do we "handle" the interrupt (1):

- through a type of function called an *interrupt handler*.
- **Golden Rule:** always make *interrupt handler* **takes as little time as possible**:
  - *interrupt* stops the CPU from executing the `main` thread.
  - Another *interrupt* while we're inside an *interrupt handler* can happen.
  - That 2nd interrupt could be using *the same interrupt handler* as the ongoing interrupt. This causes error that is very hard to debug. (cooking a second dish while stalling the unfinished first dish).
- because an *interrupt handler* starts externally, its function prototype takes no argument and return nothing:

```
void nmi_handler(void);
```

- our compiler looks at the **special keyword** `__critical __interrupt` to designate a function to be an interrupt handler.
- I chose the name `nmi_handler` to be our interrupt handler in out **crt0** (startup code).
- so, all in all, we declare the function prototype as:

```
void nmi_handler(void);
```

- and write the function body as:

```
void nmi_handler(void) __critical __interrupt {
    // code here
}
```

app
comp arch
logic

04 Interrupts and stack

How do we "handle" the interrupt (2):

- unlike `main`, `nmi_handler` is called repeatedly.
- remember that lifecycle of a variable starts and ends with the function execution.
- we need to store values over several `nmi_handler` calls.
- 2 ways, with different purposes:

Method 1: local static variables: prefix variable declarations with the keyword `static:`

Method 2: global variables: declare them outside function:

For our dot runner, we will use **method 1**.

app
comp arch
logic

04 Interrupts and stack

How do we "handle" the interrupt (3):

Method 1: local static variables: prefix variable declarations with the keyword `static:`

- `static` variables persist through many function calls. They are initialized *once only*.

```
void nmi_handler(void) __critical __interrupt {
    static uint8_t m = 0;   // will be 0 at the first call only
    // code
    m = new_m;              // next time nmi_handler is called, m will be new_m
}
```

- The keyword `static` can be used in other functions too. Note that the variable is private to the function where it is defined. no one else can touch it.:

```
uint16_t packet_id(void){
    static uint16_t count = 0;
    count = count + 1;
    return count;
}

uint16_t d = packet_id();  // first call,  d is 1
uint16_t e = packet_id();  // second call, e is 2
```

04 Interrupts and stack

How do we "handle" the interrupt (4):

Method 2: global variables: declare them outside function:

```
volatile uint8_t hello = 1;   // I will store value

void nmi_handler(void) __critical __interrupt {
    uint8_t local_hello = hello; // can read hello here
    hello = my_new_hello;        // can update hello here
    ...
}
```

- **Every function (including `main` and `nmi_handler`) can access this variable.**
  - You may or may not want this behavior. Use it to fit the code's purpose.
- The "shared" variable **must be** declared as `volatile`.
  - Both `main` and `nmi_handler` can access, and compiler doesn't know when interrupt will occur. `volatile` forces every read to `hello` from any function must be explicitly performed, just like IO addresses.

04 Interrupts and stack

📖 **How do we "handle" the interrupt (5):**

Method 1: Local Static Variables (`static`)
Best for: Internal logic, counters, and state machines inside the interrupt.

Pros:

- **Encapsulation (Safety):** Only `nmi_handler` can see or touch this variable. It is impossible for main or another function to accidentally corrupt your timer or counter.
- **Namespace Hygiene:** You can use the name count in five different functions without them conflicting.
- **Self-Contained:** All the logic for that specific task stays inside the function.

Cons:

- **Isolation:** `main` cannot read this variable. You cannot use this method if you need to display the value or use it to trigger game logic outside the interrupt.
- **Hidden Complexity:** For beginners, it can be confusing that a variable declared "inside" a function doesn't reset when the function ends.

04 Interrupts and stack

📖 **How do we "handle" the interrupt (6):**

Method 2: Global Variables (`volatile`)
Best for: Communication between the Interrupt and the Main Loop.

Pros:

- **The Bridge:** This is the only way to pass data out of an interrupt. Since nmi_handler has no return value, you must use a global variable if you want the main while(1) loop to react to an event (e.g., "Player pressed button").

Cons:

- **Safety Risk:** Any function in your program can modify this variable by mistake. It relies on the programmer's discipline to not touch it elsewhere.
- **The "Atomicity" Trap:** The Z80 is an 8-bit CPU. If your global variable is 16-bit (like `uint16_t score`), the Z80 takes two cycles to read it (low byte, then high byte). If an interrupt fires between those two cycles, `main` will read a corrupted "torn" value (half old, half new).

04 Interrupts and stack

The dot runner:

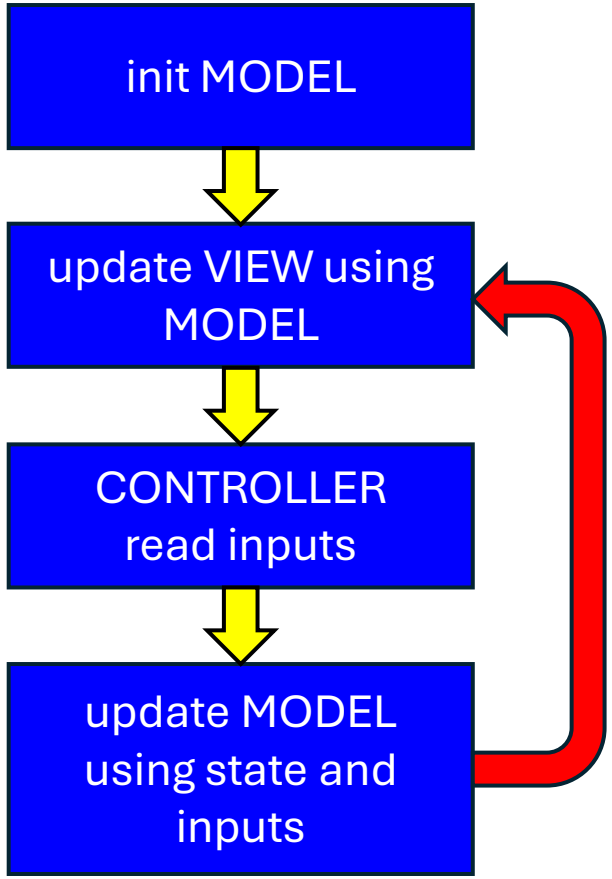
- Canvas is black. Dot is yellow. Dot starts at row 4, column 7
- Pressing arrow buttons lets dot run in that direction. Pressing ■ makes the dot stop moving.
- If the dot hits boundary, it *wraps-around*.

System Organization:

- Timing is controlled by *interrupt signal*.
- All MVC reside in the *interrupt handler*.
- `main` is empty, containing only `while(true)`.

Software:

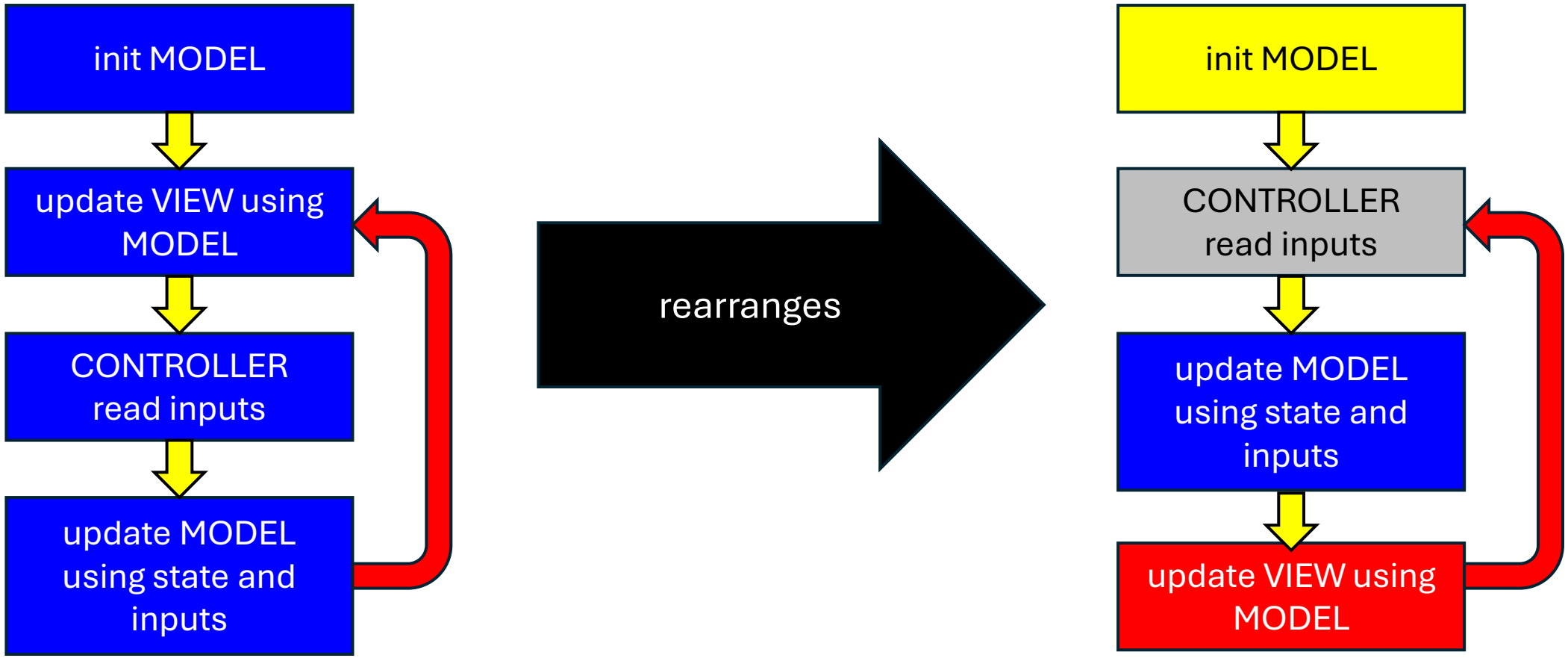
- All the boxes to the right sit inside nmi_handler.
- `nmi_handler` must be able to initialize model
- …and run the loop
- The loop arrow (red arrow) is done by repeatedly calling `nmi_handler`
- We'll arrange the loop structure a bit, but the flow remains the same.



04 Interrupts and stack

The dot runner:

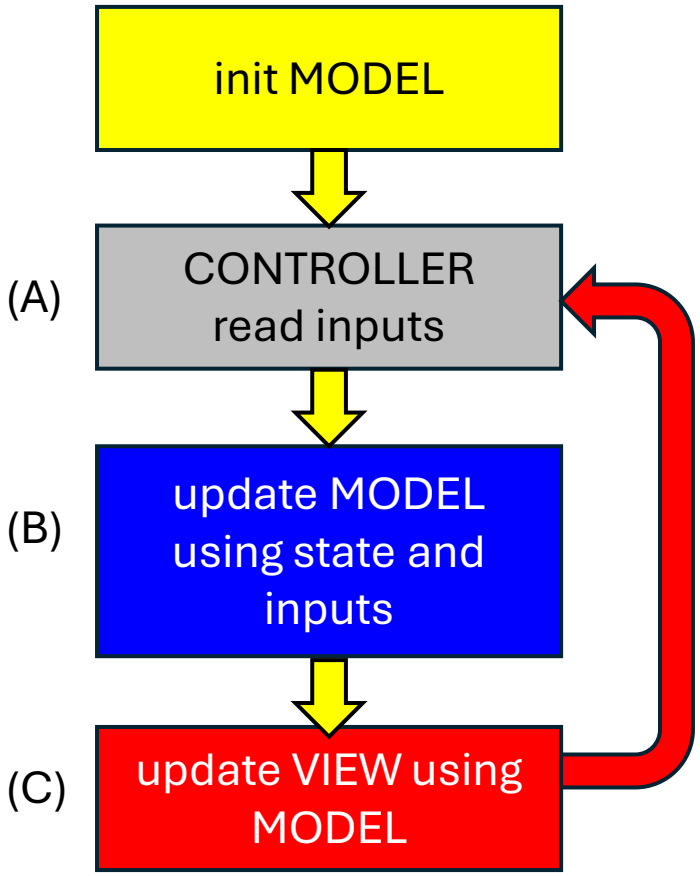
- Canvas is black. Dot is yellow. Dot starts at row 4, column 7
- Pressing arrow buttons lets dot run in that direction. Pressing ■ makes the dot stop moving.
- If the dot hits boundary, it *wraps-around*.



04 Interrupts and stack

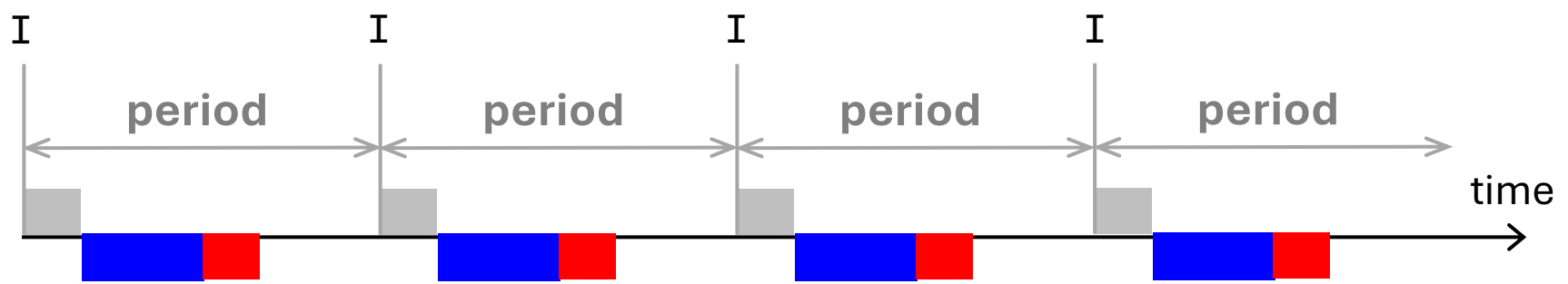
The dot runner:

- Canvas is black. Dot is yellow. Dot starts at row 4, column 7
- Pressing arrow buttons lets dot run in that direction. Pressing ■ makes the dot stop moving.
- If the dot hits boundary, it *wraps-around*.



Why do it this way?

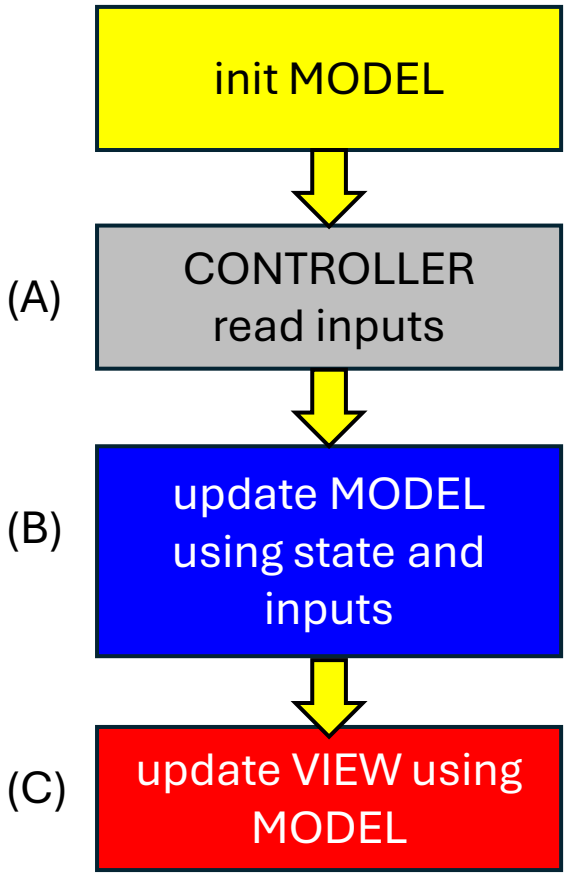
- interrupt handler starts at every vertical line `I`.
- in an interrupt handler, we want to read input first and update view last.
- So, it runs (A) (B) (C) then exits the interrupt handler.
- Doing (C) (A) (B) can cause input (A) to delay by one interrupt period before display (C) → UX suffers from "lag."
- (A) (B) (C) **minimizes delay from input** (A) **to view update** (C)



04 Interrupts and stack

The dot runner:

- Canvas is black. Dot is yellow. Dot starts at row 4, column 7
- Pressing arrow buttons lets dot run in that direction. Pressing ■ makes the dot stop moving.
- If the dot hits boundary, it *wraps-around*.



Let's take a look at `main`:

```
void main(void){
    while (true) {
    }
}
```

`main` is empty! All logic is moved to the interrupt handler.

04 Interrupts and stack

Detour: Stack (1):

- A *stack* is a type of *data structure*. It is also called a LIFO (Last-In, First-Out)
- Usually, a stack starts out *empty*.
- When you want to store some data in it, you *push* the data *into* the stack.
- When you want to retrieve data from it, you *pop* the data *from* the stack.
- The best analogy is a "stack of plates"
- Number of items stored in a stack is called the *stack depth*.

Z80 and his Stack

A `uint8_t` Stack (each "plate" is a `uint8_t` data)

- It starts out empty (no plate)
- You *push* the data in, one at a time, like this: (`push_stack` isn't a real function)

```
push_stack(0x12);  // plate written 0x12 is on the floor
```

- Repeat it a couple of times:

```
push_stack(0x4A);  // put plate with 0x4A on top of stack
push_stack(0xAB);  // put plate with 0xAB on top of stack
```

- Now, stack will look like this:

```
0xAB   <--  This is called the top of the stack
0x4A
0x12   <--  This is called the bottom or the base of the stack
```

stack *depth* is 3

- First `pop_stack` returns the data from top (`0xAB`). Returned data is discarded.
- Next `pop_stack` returns the `0x4A`. Next pop returns `0x12`. Now stack is *empty*.

app
comp arch
logic

04 Interrupts and stack

Detour: Stack (2):

- *Stack* is so important that the CPU has a dedicated *pointer* for it, called the *stack pointer* (SP).
- SP initializes to a value specified by the linker.
  - In most hardware, it is set to the highest address of usable RAM. (for our hardware, it is `0xBFFF`)
- Here's how it is done on our Z80:
  - The Stack Pointer (SP) initializes to the **top of usable RAM + 1**. For us, that is `0xC000`.
  - This initial value is, unfortunately confusingly, called the *base* or the *bottom* of stack. (Think stack of plates)
  - A **Push** operation (putting data on stack):
    - **Decrements** SP.
    - Writes data to the new address.
  - A **Pop** operation (removing data):
    - Reads data from current SP address.
    - **Increments** SP.
- Because address gets smaller as we add data (`0xC000` → `0xBFFF` → `0xBFFE`), we say the **Stack Grows Downward**.
- Data that typically get pushed onto the stack by compiler are called **frame** or **stack frame**.
- A **frame** is generated for each function call. It contains the function's arguments, return address, and local variables.

But... 

From our memory map

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

↑ bottom of stack

04 Interrupts and stack

Detour: Stack (3):

🚀 **But... that's the theory.**

Theory vs. Z80/SDCC Reality:

The Theory (Standard C): Every function call creates a **Stack Frame** containing its arguments and local variables.

The Z80 "Plot Twist": Accessing data on the stack via the Frame Pointer (`IX`) is **slow** (19 cycles).

The Solution (SDCC): By default, our compiler **cheats**. It moves local variables to hidden, fixed memory addresses (called **Static Overlay**) to speed up the code.

The Result: The Hardware Stack is mostly used just for **Return Addresses**.

Note: The "Recursion Trap"
By default, SDCC places local variables at fixed memory addresses (Static Overlay) to save time. This breaks recursion because the inner function call overwrites the outer function's variables.

To force "Real" stack frames (necessary for recursion), you must mark the function with the `__reentrant` keyword.

```
// usage of __reentrant forces 'n' onto the stack
uint16_t factorial(uint16_t n) __reentrant {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

04 Interrupts and stack

app
comp arch
logic

Detour: Stack (4):

The Stack Frame (Activation Record)

Definition: A temporary "workspace" created on the stack for each function call. It contains the function's arguments, return address, and local variables.

Navigation: Because the **Stack Pointer (SP)** moves constantly as data is pushed/popped, we use a fixed anchor called the **Frame Pointer (FP)** to reliably locate variables relative to the start of the function.

Implementation: While the SP is a specific hardware component, the FP is often a **role** assigned to a general-purpose register by the compiler.

The Anatomy of a Stack Frame: Visualizing the Frame and its Frame Pointer (FP) Read address from bottom to top

Address	Component	Purpose
FP - n	Local Variables	The scratchpad for variables declared *inside* the function (e.g., `int16_t i`, `uint8_t buffer[10]`).
FP	Saved Frame Pointer	A backup of the *previous function's* reference point. This is the "breadcrumb" (trace) that lets us return to the caller's context.
FP + 2	Return Address	The "Bookmark" telling the CPU exactly where to go back in the code once this function finishes.
FP + n	Arguments	Variables passed *into* the function (e.g., `uint8_t a`, `uint8_t b` from calling function `add(a, b)`). The caller places these here before jumping.

04 Interrupts and stack

app
comp arch
logic

Detour: Stack (5):

Address	Component	Purpose
FP - n	Local Variables	The scratchpad for variables declared *inside* the function (e.g., `int16_t i`, `uint8_t buffer[10]`).
FP	Saved Frame Pointer	A backup of the *previous function's* reference point. This is the "breadcrumb" (traces) that lets us return to the caller's context.
FP + 2	Return Address	The "Bookmark" telling the CPU exactly where to go back in the code once this function finishes.
FP + n	Arguments	Variables passed *into* the function (e.g., `uint8_t a`, `uint8_t b` from calling function `add(a, b)`). The caller places these here before jumping.

Why do we need a "Saved Frame Pointer" inside a frame?

- Think of the **Frame Pointer (FP)** as a fixed "anchor" for a function's variables. It stays still so the code can easily find "Argument 1" or "Local Variable X" relative to it, even as the **Stack Pointer (SP)** moves up and down.

The Problem: When Function A calls Function B, Function B needs its *own* anchor. If B just overwrites the current FP, Function A loses track of its variables.

The Solution: Before creating a new anchor, Function B **saves** Function A's anchor onto the stack.

1. **Enter B:** Save A's FP → Create B's FP.
2. **Exit B:** Restore A's FP from the stack → Return.

This creates a chain of "breadcrumbs" (traces) that allows the CPU to safely unwind back through thousands of nested function calls.

04 Interrupts and stack

app
comp arch
logic

Detour: Stack (6):

Do we really need frames and frame pointers?

The Modern Twist: "Frame Pointer Omission" (FPO)

In highly optimized modern code (release builds), compilers often **delete the Frame Pointer entirely**.

Why? To free up the frame pointer **role** register, so it can be used for general math or variables.

How do they track the stack? The compiler generates a secret side-table (DWARF debug info) that tells the debugger: *"At instruction 104, the variable X is at SP + 16."*

The Trade-off:

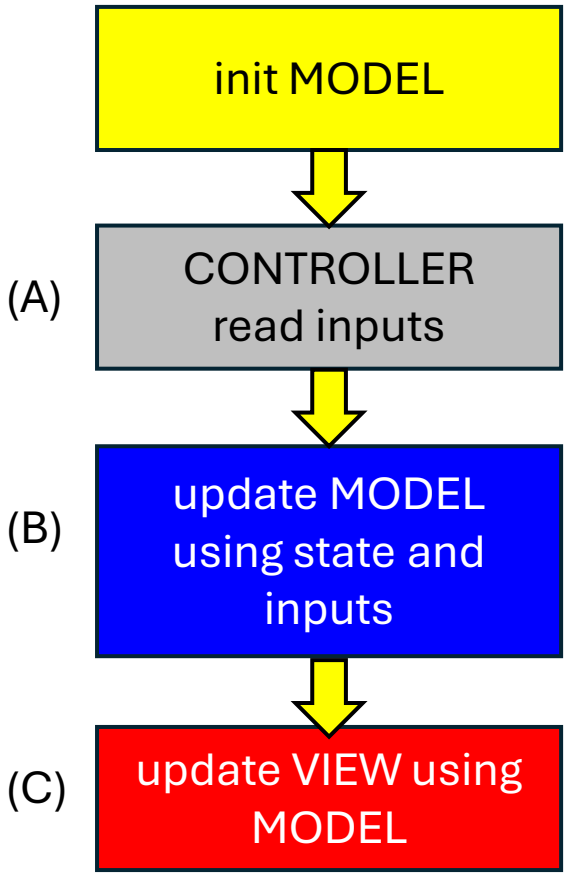
Pros: Code runs slightly faster (one more register available).

Cons: Debugging and "Stack Traces" (finding out who called who) become much slower and harder without that distinct "chain" of pointers.

04 Interrupts and stack

The dot runner:

- Canvas is black. Dot is yellow. Dot starts at row 4, column 7
- Pressing arrow buttons lets dot run in that direction. Pressing ■ makes the dot stop moving.
- If the dot hits boundary, it *wraps-around*.



Let's take a look at `main`:

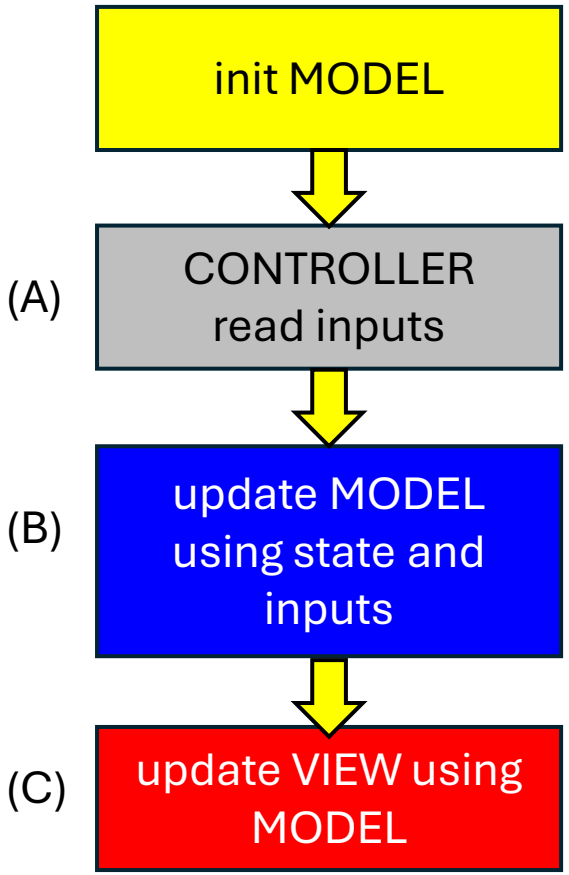
```
void main(void){
    while (true) {
    }
}
```

`main` is empty! All logic is moved to the interrupt handler.

04 Interrupts and stack

Getting back to the dot runner:

- Canvas is black. Dot is yellow. Dot starts at row 4, column 7
- Pressing arrow buttons lets dot run in that direction. Pressing ■ makes the dot stop moving.
- If the dot hits boundary, it *wraps-around*.



Let's take a look at *skeleton* of `nmi_handler`, which implements all boxes:

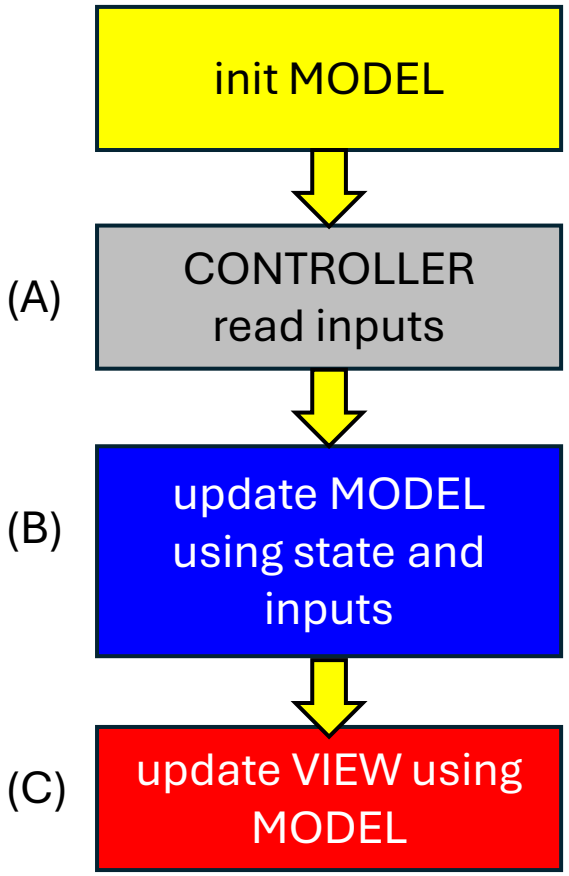
```
void nmi_handler(void) __critical __interrupt {
    // flags to run yellow box only once, and prevents NMI over NMI
    static bool busy = false;
    static bool initialized = false;
    // (code) declare all model and controller variables here
    if (busy) return; // prevent interrupt over interrupt
    busy = true;      // now we start working on the boxes
    if (!initialized) {
        // (code) do a model_init
        initialized = true;
        busy = false;
        return;
    }
    // CONTROLLER read and MODEL update
    // (code) controller_read
    // (code) update MODEL
    // (code) update VIEW
    busy = false;
    return;
}
```

```
// (code) ...

will contain C code
```

04 Interrupts and stack

Getting back to the dot runner:

- Canvas is black. Dot is yellow. Dot starts at row 4, column 7
- Pressing arrow buttons lets dot run in that direction. Pressing ■ makes the dot stop moving.
- If the dot hits boundary, it *wraps-around*.




```
void nmi_handler(void) __critical __interrupt {
    // flags to run yellow box only once, and prevents NMI over NMI
1   static bool busy = false;
    static bool initialized = false;
    // (code) declare all model and controller variables here
2   if (busy) return; // prevent interrupt over interrupt

3   busy = true;       // now we start working on the boxes
    if (!initialized) {
        // (code) do a model_init
        initialized = true;
4       busy = false;
        return;
    }
    // CONTROLLER read and MODEL update
    // (code) controller_read
    // (code) update MODEL
    // (code) update VIEW
4   busy = false;
    return;
}
```

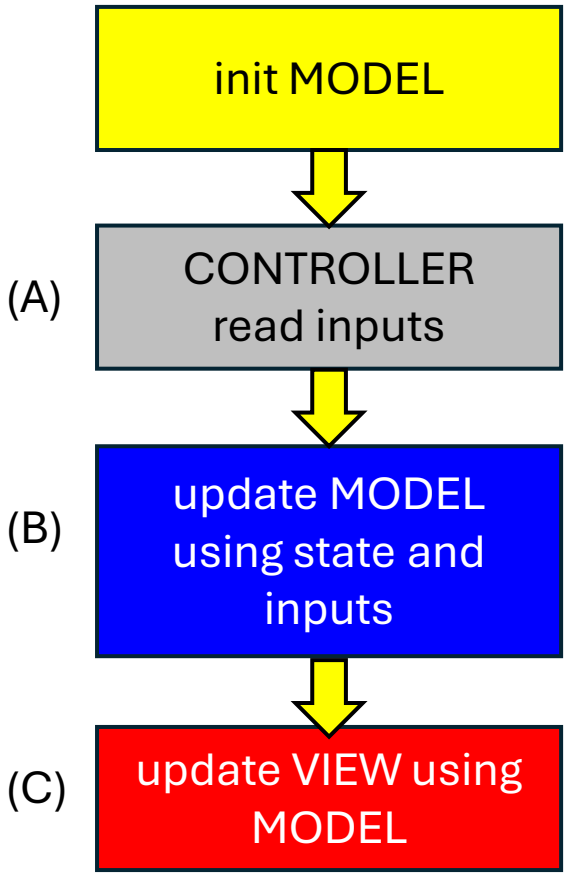
`busy` **prevents** *double interrupt execution* (interrupt while we're still in `nmi_handler`):

1. `busy` **must be** `static` and initially clear (set to `false`)
2. `busy` is checked before we do anything at all
3. `busy` is set before we start working on the boxes
4. `busy` is cleared before we return

04 Interrupts and stack

Getting back to the dot runner:

- Canvas is black. Dot is yellow. Dot starts at row 4, column 7
- Pressing arrow buttons lets dot run in that direction. Pressing ■ makes the dot stop moving.
- If the dot hits boundary, it *wraps-around*.




```
void nmi_handler(void) __critical __interrupt {
    // flags to run yellow box only once, and prevents NMI over NMI
    static bool busy = false;
1   static bool initialized = false;
    // (code) declare all model and controller variables here
    if (busy) return; // prevent interrupt over interrupt

    busy = true;       // now we start working on the boxes
2   if (!initialized) {
        // (code) do a model_init
3       initialized = true;
        busy = false;
        return;
    }
    // CONTROLLER read and MODEL update
    // (code) controller_read
    // (code) update MODEL
    // (code) update VIEW
    busy = false;
    return;
}
```

`initialzed` is a flag that implements a *one-shot* (code that executes only once):

1. `initialized` **must be** `static` and initially clear (set to `false`)
2. if `initialized` is not set, yellow box has not run. run it.
3. after `model_init` is complete, set `initialized` right away.

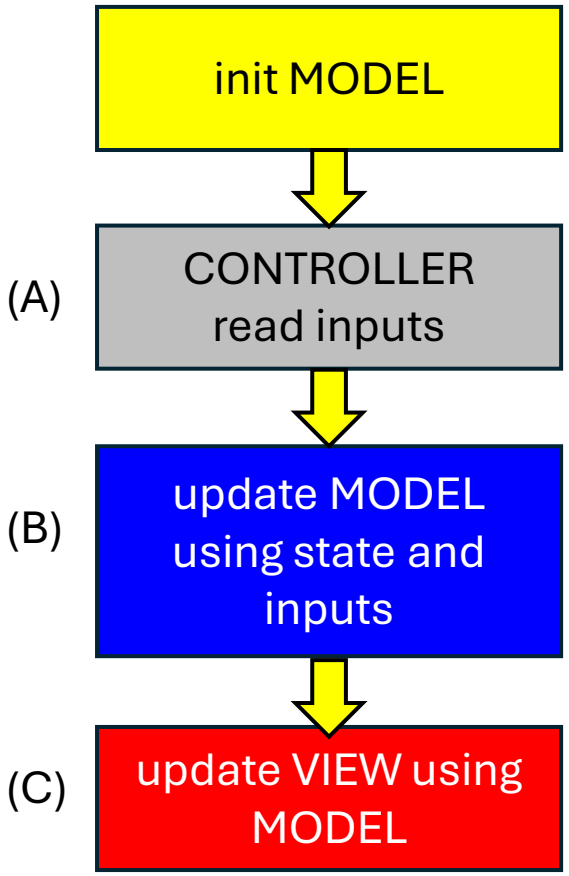
Step 3 ensures that `model_init` will run only once *(one-shot)*.

app
comp arch
logic

04 Interrupts and stack

Getting back to the dot runner:

- Canvas is black. Dot is yellow. Dot starts at row 4, column 7
- Pressing arrow buttons lets dot run in that direction. Pressing ■ makes the dot stop moving.
- If the dot hits boundary, it *wraps-around*.



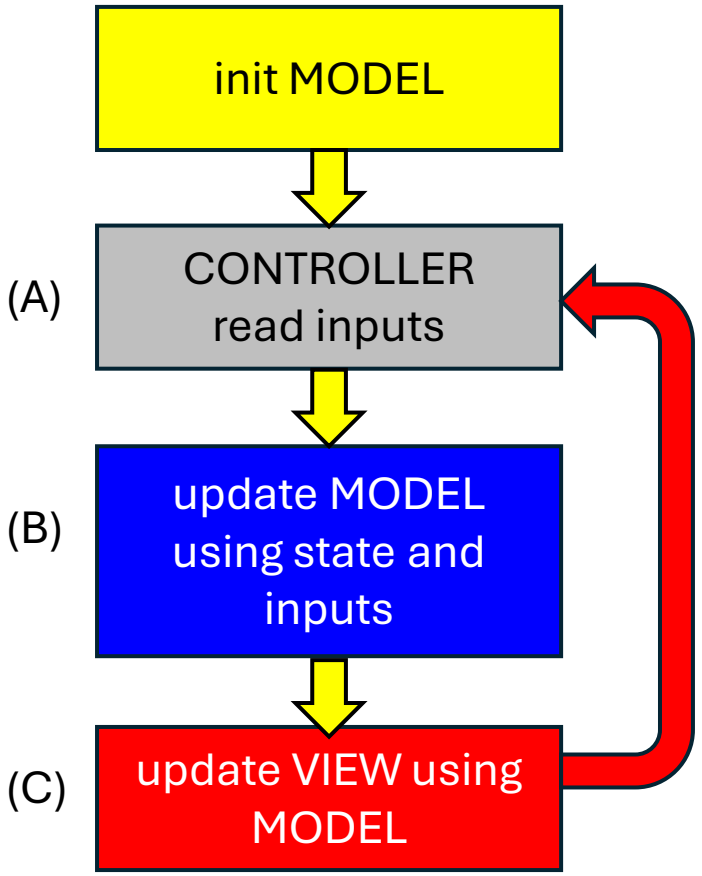

```
void nmi_handler(void) __critical __interrupt {
    // flags to run yellow box only once, and prevents NMI over NMI
    static bool busy = false;
    static bool initialized = false;
    // (code) declare all model and controller variables here
    if (busy) return; // prevent interrupt over interrupt
    busy = true;      // now we start working on the boxes

    if (!initialized) {
        // (code) do a model_init
        initialized = true;
        busy = false;
        return;
    }
    // CONTROLLER read and MODEL update
A   // (code) controller_read
B   // (code) update MODEL
C   // (code) update VIEW
    busy = false;
    return;
}
```

This shows where all the boxes are

04 Interrupts and stack

Getting back to the dot runner:

- Canvas is black. Dot is yellow. Dot starts at row 4, column 7
- Pressing arrow buttons lets dot run in that direction. Pressing ■ makes the dot stop moving.
- If the dot hits boundary, it *wraps-around*.




```
void nmi_handler(void) __critical __interrupt {
    // flags to run yellow box only once, and prevents NMI over NMI
    static bool busy = false;
    static bool initialized = false;
    // (code) declare all model and controller variables here
    if (busy) return; // prevent interrupt over interrupt
    busy = true;      // now we start working on the boxes

    if (!initialized) {
        // (code) do a model_init
        initialized = true;
        busy = false;
        return;
    }
    // CONTROLLER read and MODEL update
    // (code) controller_read
    // (code) update MODEL
    // (code) update VIEW
    busy = false;
    return;
}
```

The red arrow is not in code.

It is implemented by calling `nmi_handler` repeatedly.

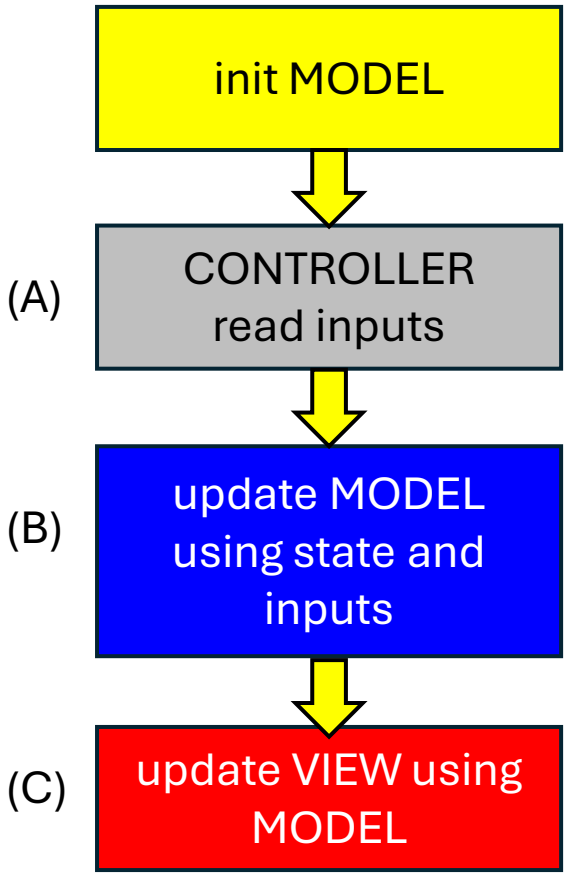
Through `initialized`:

- init MODEL is *one-shot*.
- The following runs in sequence:
  - CONTROLLER read
  - update MODEL
  - update VIEW

04 Interrupts and stack

Getting back to the dot runner:

- Canvas is black. Dot is yellow. Dot starts at row 4, column 7
- Pressing arrow buttons lets dot run in that direction. Pressing ■ makes the dot stop moving.
- If the dot hits boundary, it *wraps-around*.



Constant for ALL MVC:

```
#define DISP_WIDTH (16)  // LED matrix width. Used by MODEL and VIEW
```

Let's look at MODEL *states* for the dot runner:

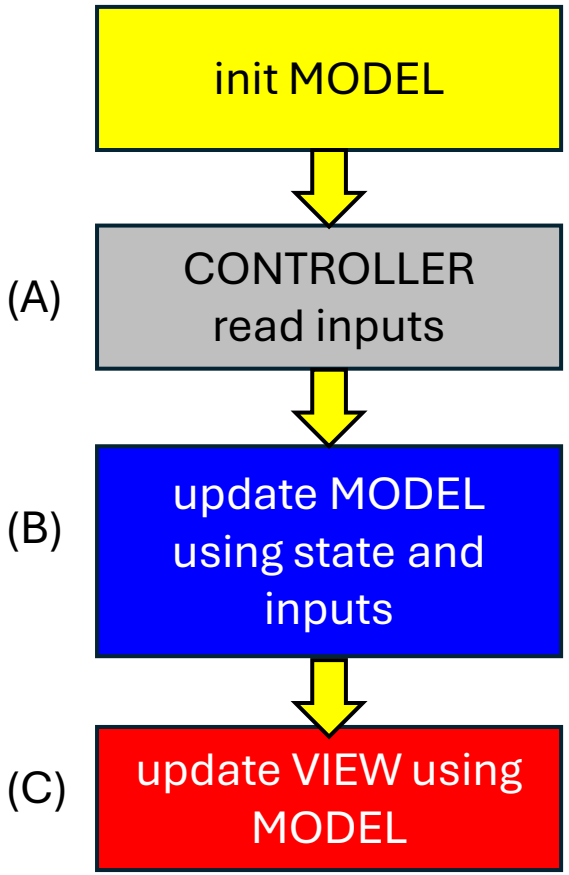
```
typedef enum {    // MODEL-CONTROLLER message
    NONE, STOP, LEFT, RIGHT, UP, DOWN   // why do we need both
} command;                              // NONE and STOP ?

typedef struct {  // MODEL states
    uint8_t row, col;    // current row and column
    command cmd;         // current command
} model_t;               // why don't we need matrix[16] like painter?
```

Let's look at init MODEL:

```
void model_init(model_t *mp){
    mp->row = 4;     // dot starts at row 4
    mp->col = 7;     //                col 7
    mp->cmd = STOP;  // dot starts not running
}
```

Let's look at update MODEL:

- Canvas is black. Dot is yellow.
- Dot starts at row 4, column 7
- Arrow buttons for directions.
- Pressing ■ to stop moving.
- Dot's movement *wraps-around*.

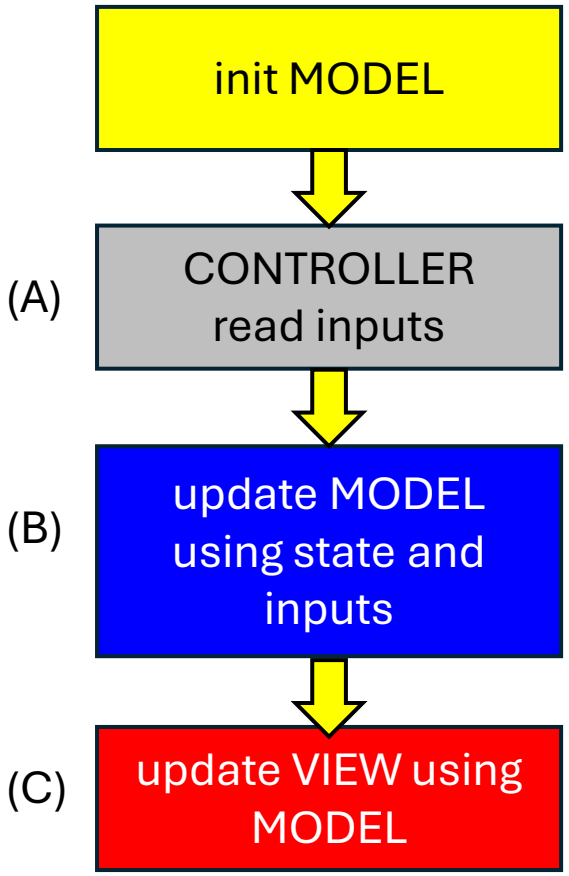



```
void model_update(model_t *mp, command c){
    // static avoids stack tax
    static uint8_t new_row, new_col;
    static command new_cmd;
    new_cmd = (c == NONE) ? mp->cmd : c;   // Why?
    new_row = mp->row;                     // unless update is needed
    new_col = mp->col;                     // keep same row and col
    switch (new_cmd){
        case STOP:  break;
        case LEFT:  new_col = (mp->col == 0)? DISP_WIDTH-1 : mp->col-1;
                    break;
        case RIGHT: new_col = (mp->col >= DISP_WIDTH-1)? 0 : mp->col+1;
                    break;
        case DOWN:  new_row = (mp->row == 7)? 0 : mp->row + 1;
                    break;
        case UP:    new_row = (mp->row == 0)? 7 : mp->row - 1;
                    break;
        default:    break;
    }
    // update state
    mp->cmd = new_cmd;
    mp->row = new_row;
    mp->col = new_col;
}
```

04 Interrupts and stack

Getting back to the dot runner:

- Canvas is black. Dot is yellow. Dot starts at row 4, column 7
- Pressing arrow buttons lets dot run in that direction. Pressing ■ makes the dot stop moving.
- If the dot hits boundary, it *wraps-around*.

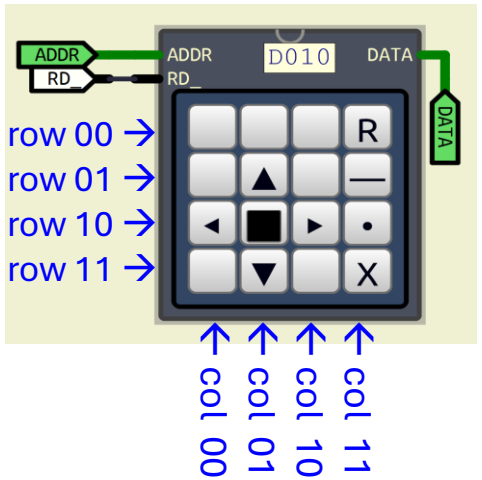


Let's look at addresses for VIEW:

```
#define MATRIX_BASE ((volatile uint8_t * const)0xE020U) // LED matrix
#define HEX_DISP ((volatile uint8_t * const)0xE808U)    // Row and Col
```

Let's look at address for CONTROL:

```
#define KEYPAD ((volatile const uint8_t * const)0xD010U) // read to clear
```



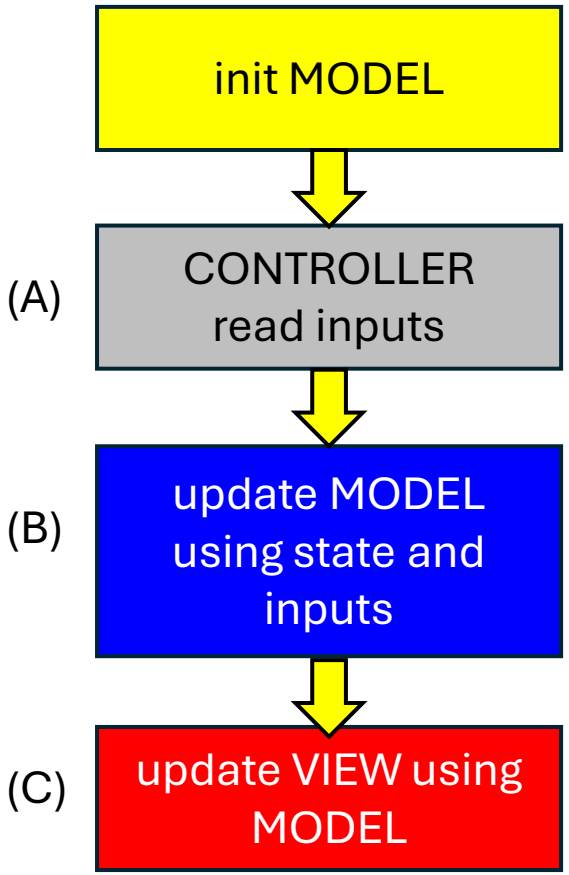
0xD010 bit field definition

Bit	7	6	5	4	3	2	1	0
Field	valid	0	R1	R0	0	0	C1	C0

```
UP ARROW is B8(10010001)
LEFT ARROW is B8(10100000)
STOP is B8(10100001)
RIGHT ARROW is B8(10100010)
DOWN ARROW is B8(10110001)
```

Let's look at controller read:

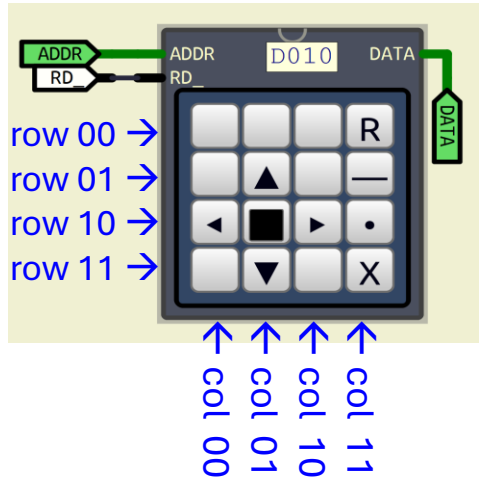
- Canvas is black. Dot is yellow.
- Dot starts at row 4, column 7
- Arrow buttons for directions.
- Pressing ■ to stop moving.
- Dot's movement *wraps-around*.




```
command controller_read(void){
    static uint8_t key;  // avoid stack tax
    static command c;    // avoid stack tax again
    key = *KEYPAD;
    if (! (key & B8(10000000)) ){
        return NONE;    // no new keypress
    }
    // valid bit is set, check key
    switch (key & B8(01111111)){
        case B8(00010001):  c = UP;
                            break;
        case B8(00100000):  c = LEFT;
                            break;
        case B8(00100001):  c = STOP;
                            break;
        case B8(00100010):  c = RIGHT;
                            break;
        case B8(00110001):  c = DOWN;
                            break;
        default:            c = NONE;
                            break;
    }
    return c;
}
```

0xD010 bit field definition

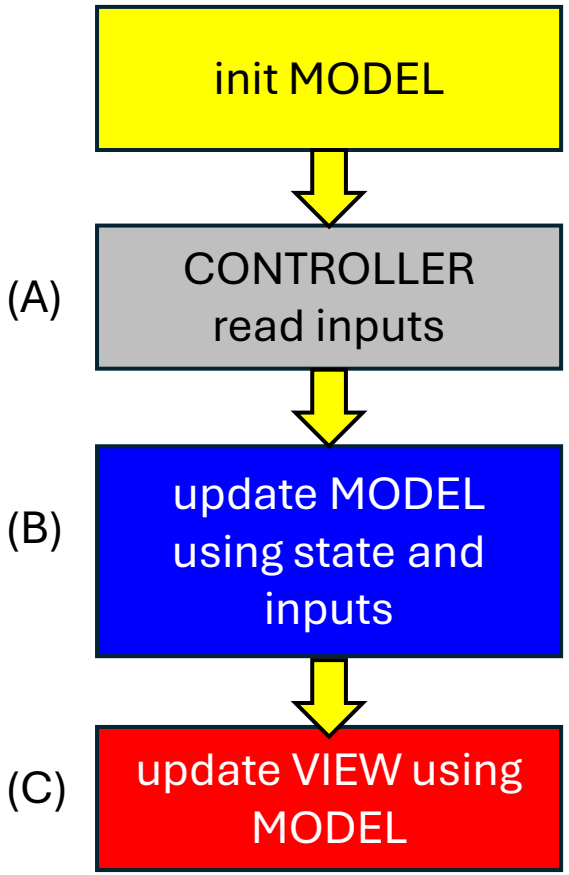
Bit	7	6	5	4	3	2	1	0
Field	valid	0	R1	R0	0	0	C1	C0




```
   UP ARROW is B8(10010001)
 LEFT ARROW is B8(10100000)
      STOP is B8(10100001)
RIGHT ARROW is B8(10100010)
 DOWN ARROW is B8(10110001)
```

Let's look at CONTROLLER read:

- Canvas is black. Dot is yellow.
- Dot starts at row 4, column 7
- Arrow buttons for directions.
- Pressing ■ to stop moving.
- Dot's movement *wraps-around*.

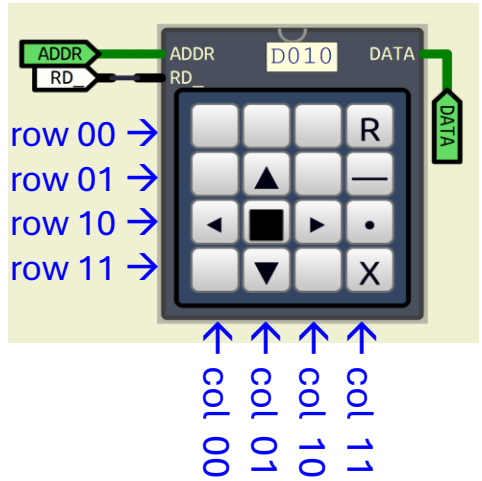



```
command controller_read(void){
    static uint8_t key;  // avoid stack tax
    static command c;    // avoid stack tax again
    key = *KEYPAD;
    if (! (key & B8(10000000)) ){
        return NONE;     // no new keypress
    }
    // valid bit is set, check key
    switch (key & B8(01111111)){
        case B8(00010001):  c = UP;
                            break;
        case B8(00100000):  c = LEFT;
                            break;
        case B8(00100001):  c = STOP;
                            break;
        case B8(00100010):  c = RIGHT;
                            break;
        case B8(00110001):  c = DOWN;
                            break;
        default:            c = NONE;
                            break;
    }
    return c;
}
```

Why is "valid" in code 0, but hardware is 1?

0xD010 bit field definition

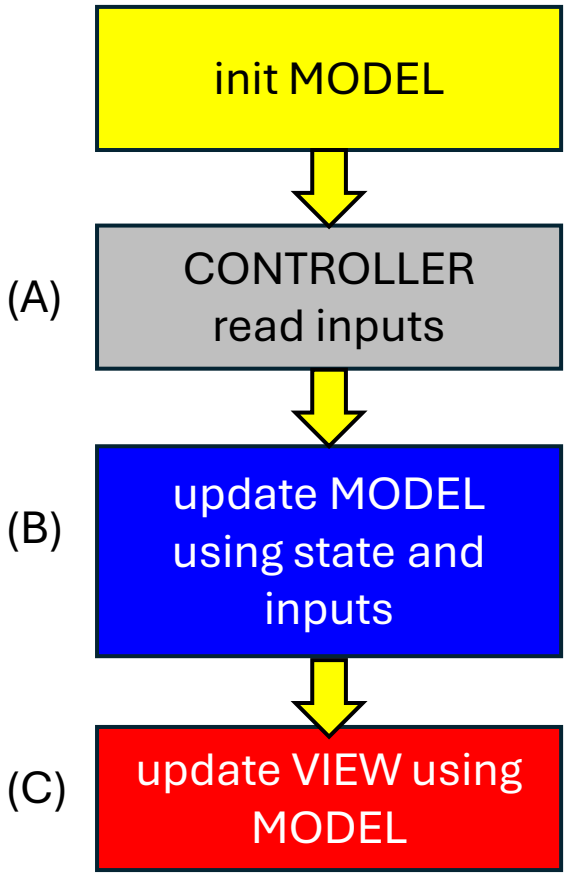
Bit	7	6	5	4	3	2	1	0
Field	valid	0	R1	R0	0	0	C1	C0




```
   UP ARROW is B8(10010001)
 LEFT ARROW is B8(10100000)
 STOP is B8(10100001)
RIGHT ARROW is B8(10100010)
 DOWN ARROW is B8(10110001)
```

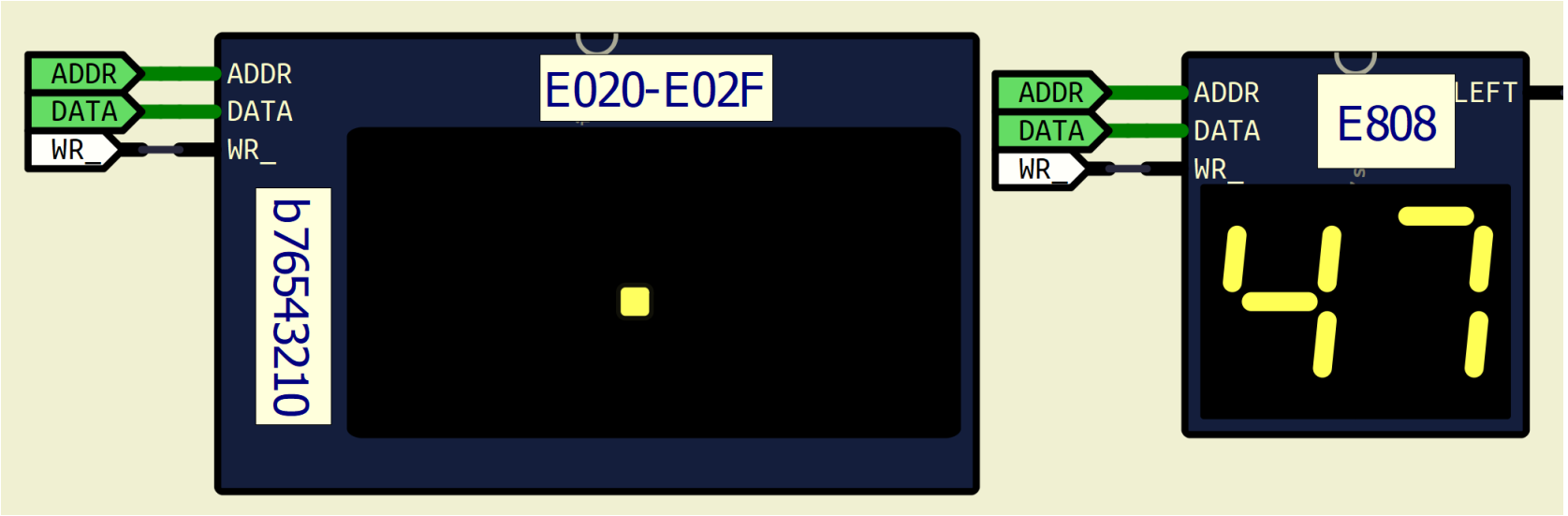
04 Interrupts and stack

- Canvas is black. Dot is yellow.
- Dot starts at row 4, column 7
- Arrow buttons for directions.
- Pressing ■ to stop moving.
- Dot's movement *wraps-around*.



Let's look at update VIEW:

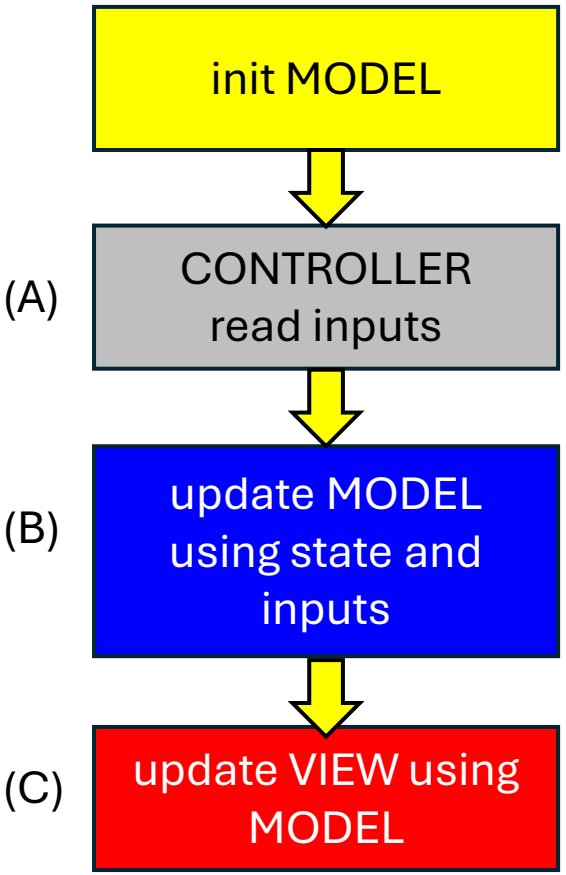
```
#define MATRIX_BASE ((volatile uint8_t * const)0xE020U)
#define HEX_DISP ((volatile uint8_t * const)0xE808U)

void view_update(model_t *mp){
    for(uint8_t i = 0; i < DISP_WIDTH; i = i + 1){
        *(MATRIX_BASE + i) = (mp->col == i)? 1U << (7 - mp->row) : 0;
    }
    *HEX_DISP = (mp->row << 4) | mp->col;
}
```



04 Interrupts and stack

app
comp arch
logic

- Canvas is black. Dot is yellow.
- Dot starts at row 4, column 7
- Arrow buttons for directions.
- Pressing ■ to stop moving.
- Dot's movement *wraps-around*.

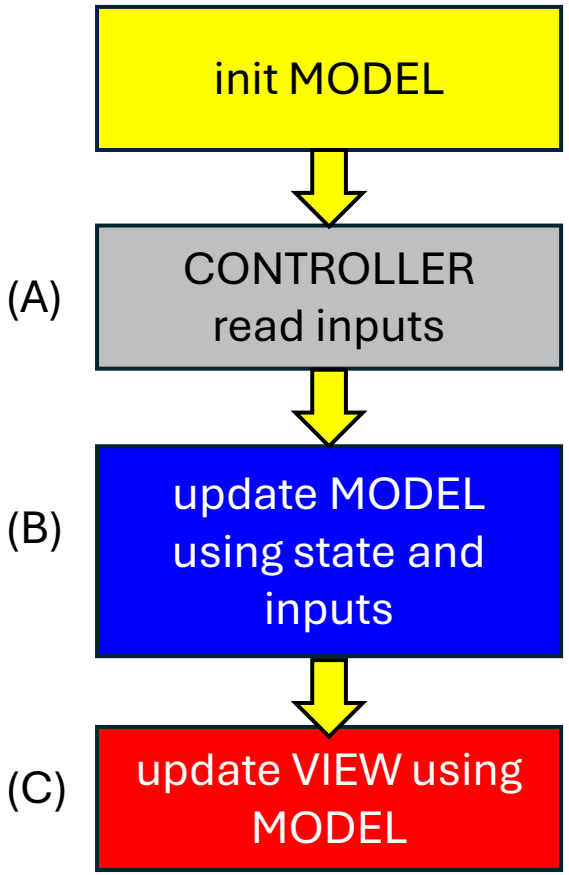


Let's look at the whole `nmi_handler` (effectively replaces `main`)**:**

```
void nmi_handler(void) __critical __interrupt {
    // we declare them all as static to avoid stack allocation
    static bool busy = false;
    static bool initialized = false;
    static model_t m;
    static model_t *mp = &m;
    static command c;

    if (busy) return; // we cannot afford to do a second interrupt

    busy = true;
    if (!initialized) {
        model_init(mp);
        initialized = true;
        busy = false;
        return;
    }
    // CONTROLLER read and MODEL update
    c = controller_read();
    model_update(mp, c);
    view_update(mp);
    busy = false;
    return;
}
```

04 Interrupts and stack

- Canvas is black. Dot is yellow.
- Dot starts at row 4, column 7
- Arrow buttons for directions.
- Pressing ■ to stop moving.
- Dot's movement *wraps-around*.



One more addition: adding an "activity LED" to `nmi_handler`:

- We want to see when `nmi_handler` is "`busy`".
- We add a light as a *side-channel* VIEW. ("side-channel" – not in the spec)
- Here is what we will do:

```
// NMI handler monitor
#define IN_NMI_LED ((volatile bool *)0xE000U)
```

- In `nmi_handler`, we will replace:

```
busy = true;
```

with

```
busy = true;
*IN_NMI_LED = true;
```

- In `nmi_handler`, we will replace:

```
busy = false;
```

with

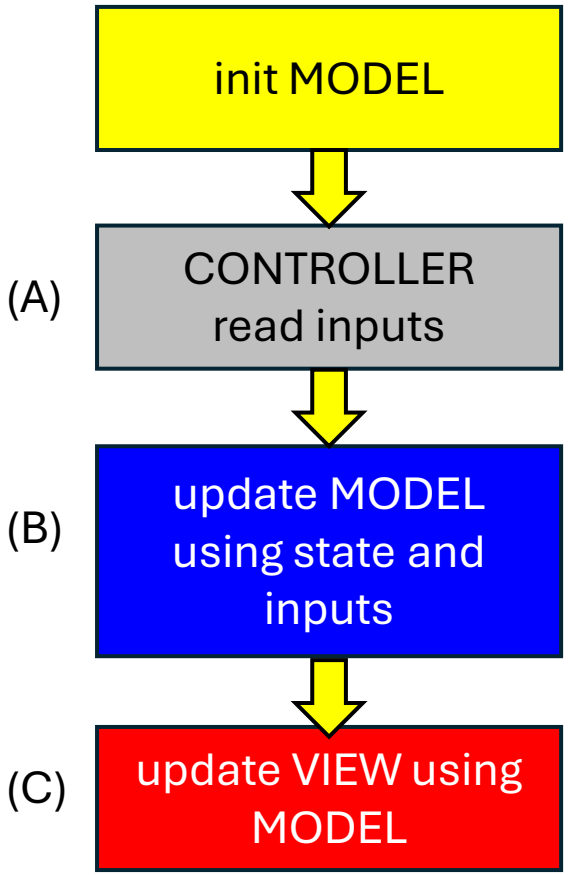
```
busy = false;
*IN_NMI_LED = false;
```

The effect: when `nmi_handler` is running, the light at `0xE000` will be on.

- So, we will know how much time (percentage) is spent running `nmi_handler`.

04 Interrupts and stack

- Canvas is black. Dot is yellow.
- Dot starts at row 4, column 7
- Arrow buttons for directions.
- Pressing ■ to stop moving.
- Dot's movement *wraps-around*.

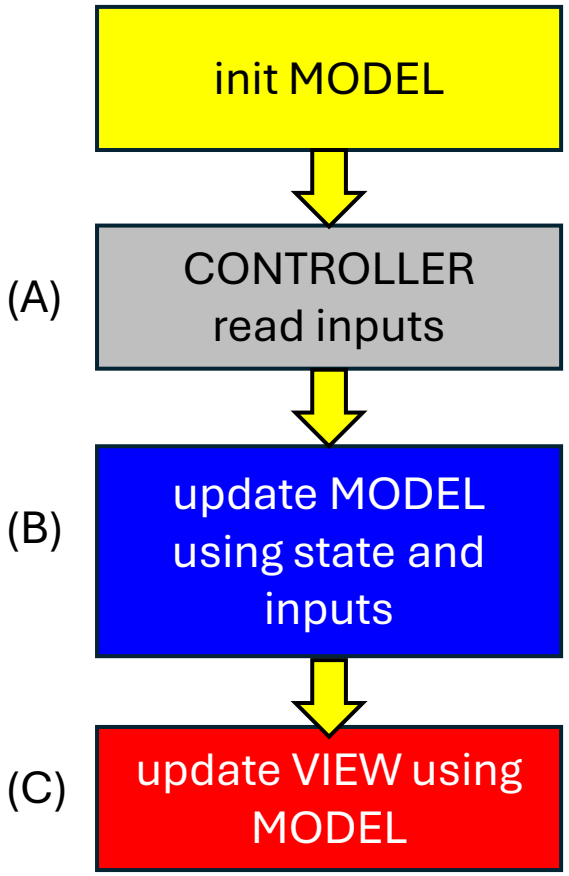


Let's see it running: [13-401]

Let's also play with NMI_ input signal frequency.

04 Interrupts and stack

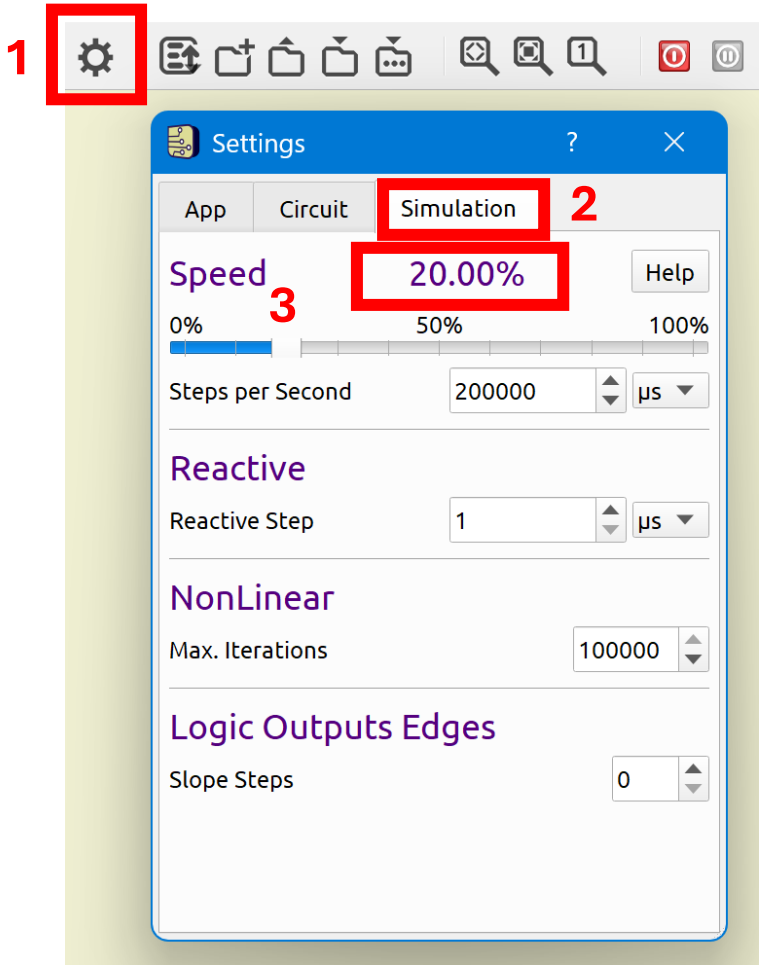
- Canvas is black. Dot is yellow.
- Dot starts at row 4, column 7
- Arrow buttons for directions.
- Pressing ■ to stop moving.
- Dot's movement *wraps-around*.



Seeing [13-401] running.

`IN_NMI_LED` **is blinking too quickly.**

We can "slow down time" to see it in simulIDE (not possible in real life):



Slowing down Simulation:

1. Click "Settings"
2. Click "Simulation" tab
3. Set Speed to about 10%-20%
4. Click "Run" again.

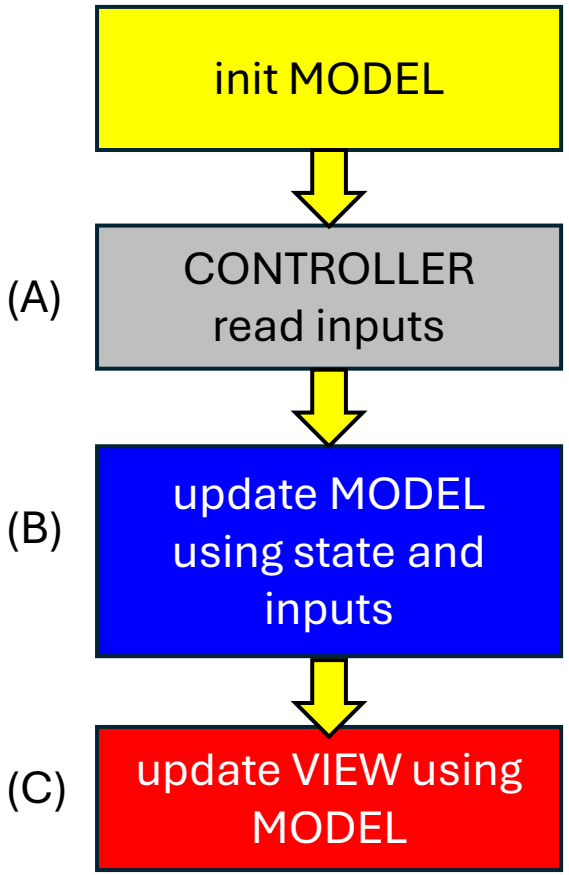
Now simulation runs at 10%-20% of the "real-time" speed.

You'll see that the `IN_NMI_LED` light is mostly off.

→ That means most of the time, the CPU is running `main` and not `nmi_handler`.

04 Interrupts and stack

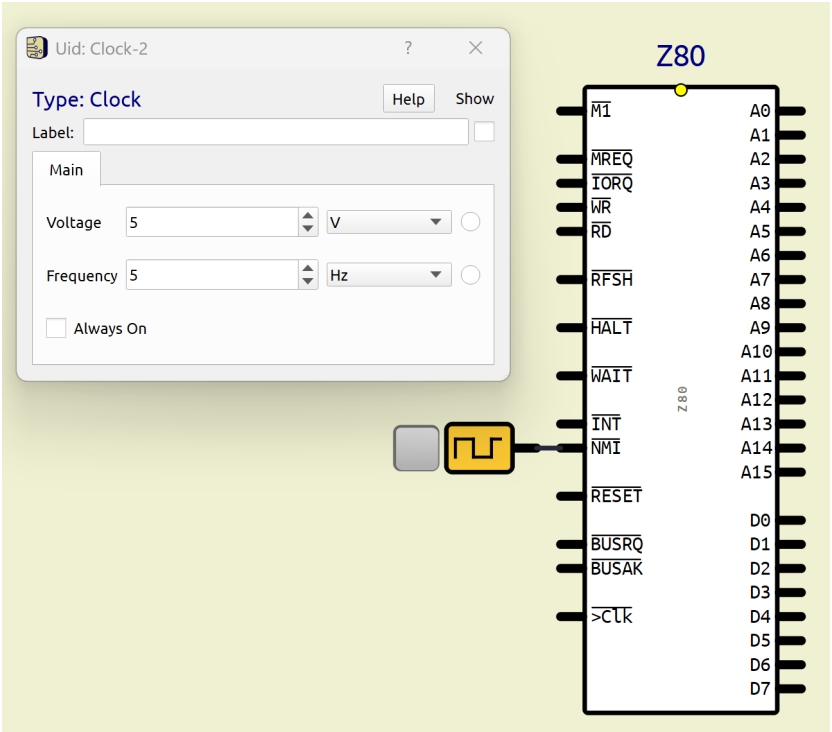
- Canvas is black. Dot is yellow.
- Dot starts at row 4, column 7
- Arrow buttons for directions.
- Pressing ■ to stop moving.
- Dot's movement *wraps-around*.



There are 2 parameters here:

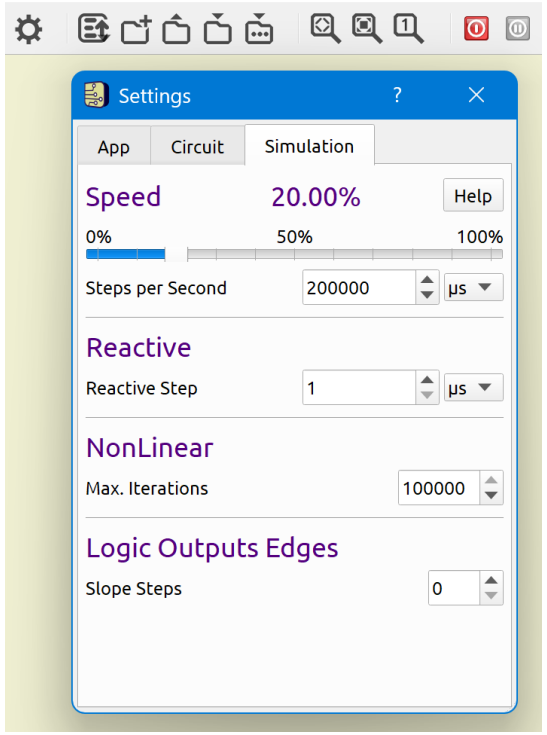
1. **interrupt period:** This control the real-time speed.

- This shows up in REAL hardware.
- Control this by changing the frequency of input to NMI_ of the CPU.



2. **simulation speed:** This control the speed of simulation.

- Not related to hardware.
- Is like "playback speed" on YouTube.



app
comp arch
logic