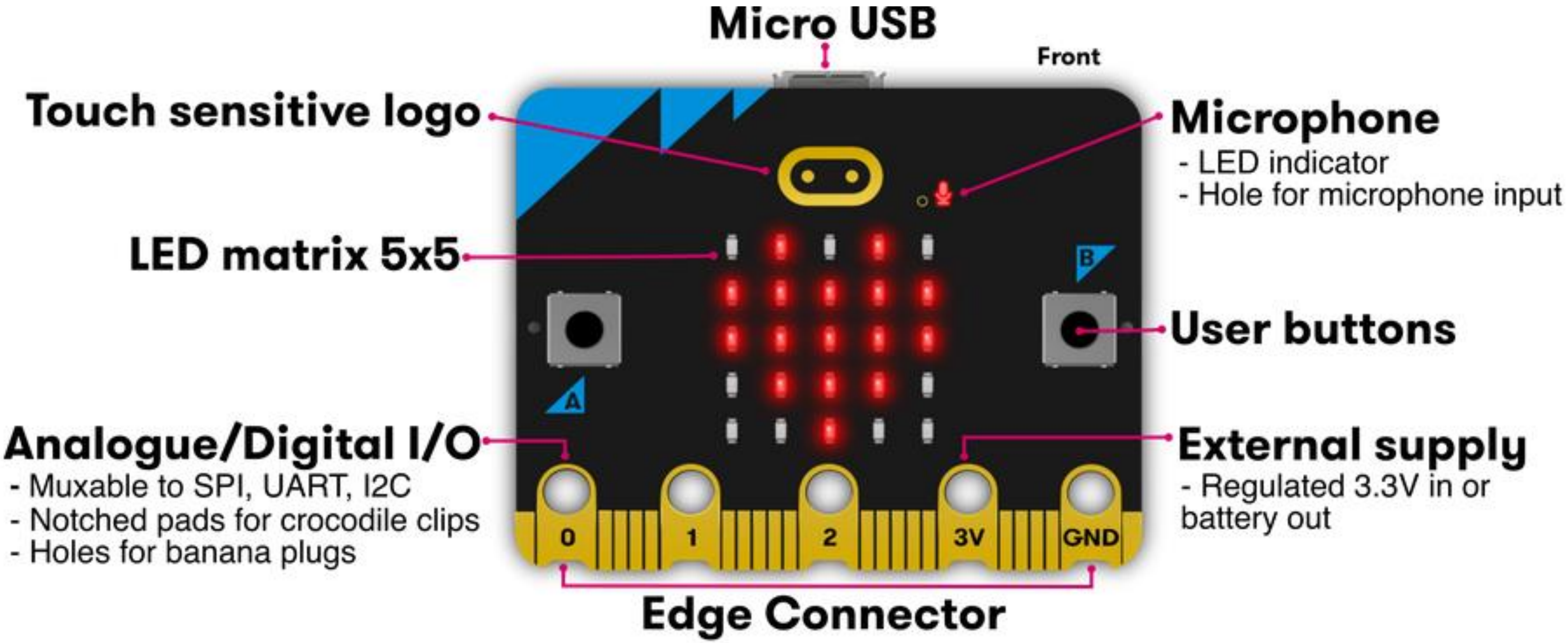
BBC Micro: Bit

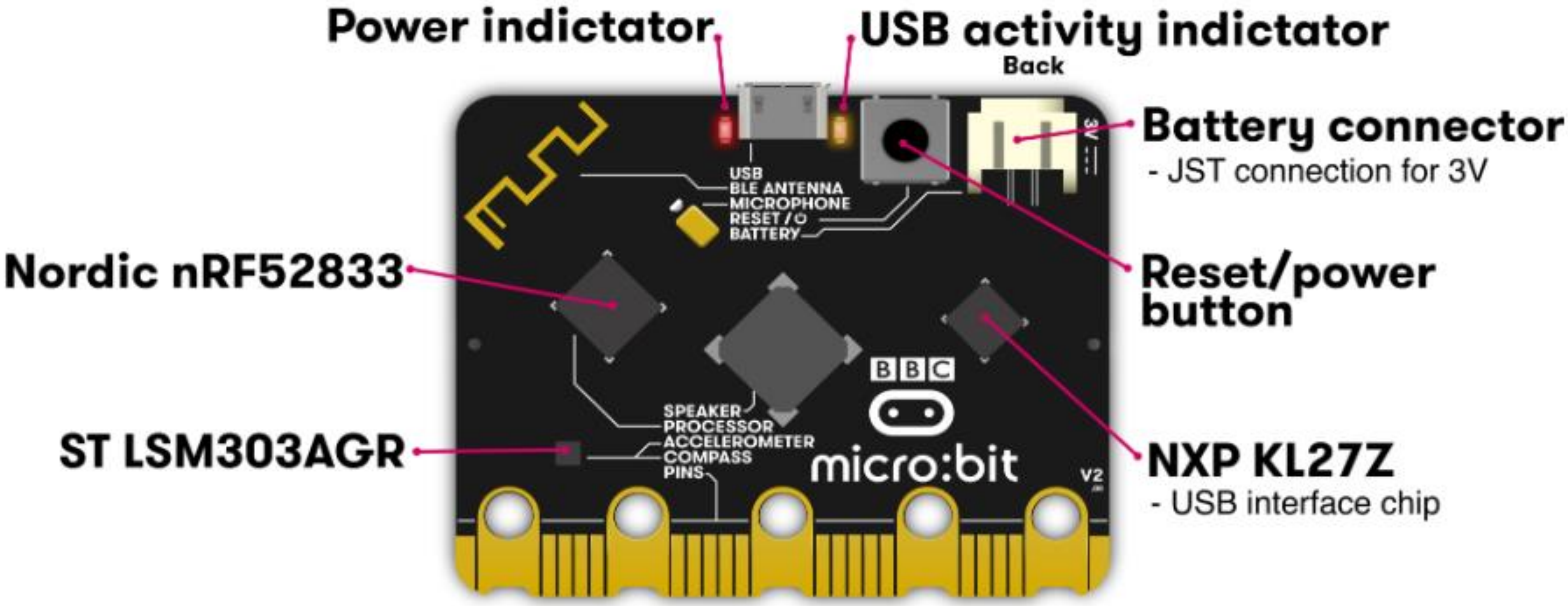
Introduction

Hardware

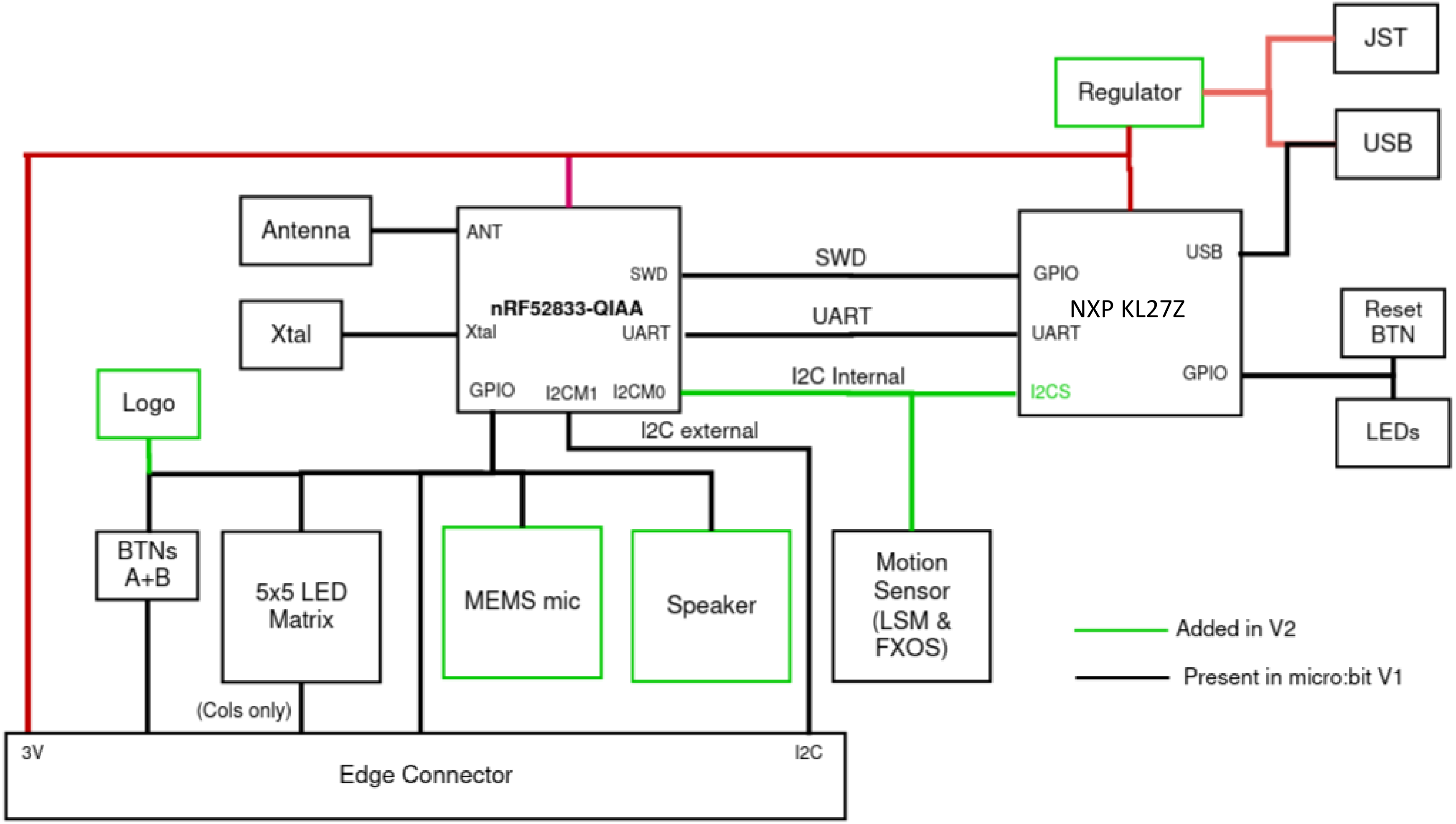


Micro USB
Front
Touch sensitive logo
Microphone
- LED indicator
- Hole for microphone input
LED matrix 5x5
B
User buttons
A
Analogue/Digital I/O
- Muxable to SPI, UART, I2C
- Notched pads for crocodile clips
- Holes for banana plugs
External supply
- Regulated 3.3V in or battery out
0
1
2
3V
GND
Edge Connector

Hardware


Power indicator
USB activity indicator
Back
Battery connector
- JST connection for 3V
Nordic nRF52833
Reset/power button
ST LSM303AGR
NXP KL27Z
- USB interface chip
USB
BLE ANTENNA
MICROPHONE
RESET / ⏻
BATTERY
SPEAKER
PROCESSOR
ACCELEROMETER
COMPASS
PINS
3V
BBC
micro:bit
V2

Hardware


JST
Regulator
USB
Antenna
ANT
SWD
nRF52833-QIAA
Xtal
UART
GPIO
I2CM1
I2CM0
Xtal
Logo
SWD
UART
I2C Internal
I2C external
GPIO
USB
NXP KL27Z
UART
I2CS
GPIO
Reset
BTN
LEDs
BTNs
A+B
5x5 LED
Matrix
MEMS mic
Speaker
Motion
Sensor
(LSM &
FXOS)
Added in V2
Present in micro:bit V1
(Cols only)
3V
I2C
Edge Connector

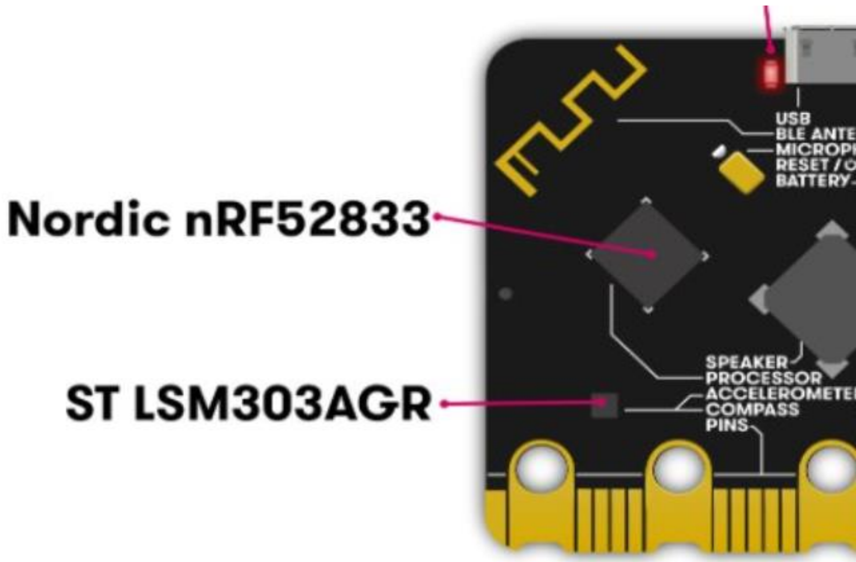
MCUs

There are 2 MCUs (microcontroller units) on board.

- Nordic nRF52833   -> main MCU , where all the user instructions run.
- NXP KL27Z   -> interface MCU, which basically helps in flashing the code in the RAM and controls all interactions with USB, including the debugger

Nordic nRF52

The nRF52833 application processor is where user programs run.



Flash ROM : 512KB

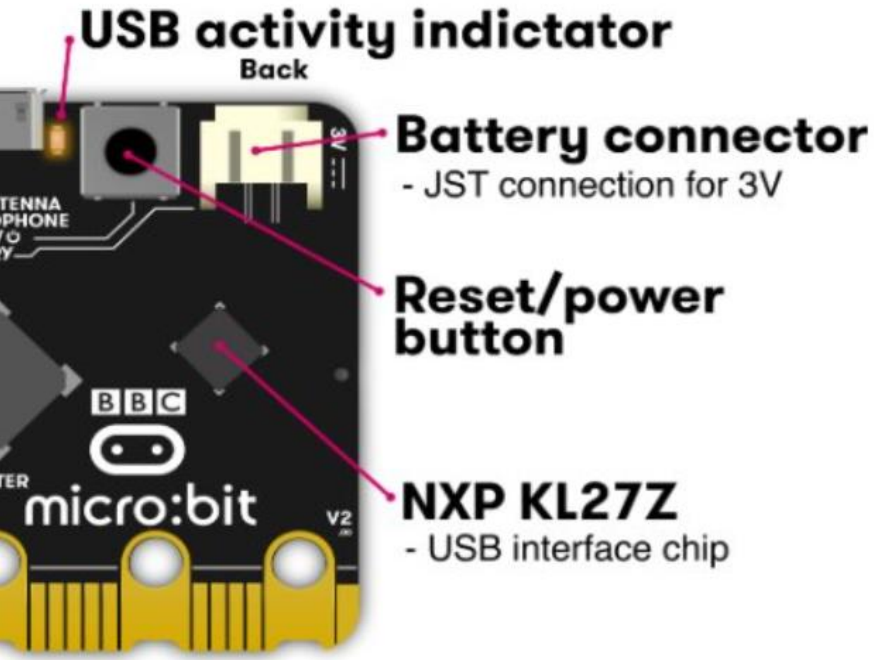
RAM : 128KB

Speed : 64MHz

ISA : ARM-Cortex-V7 instructions

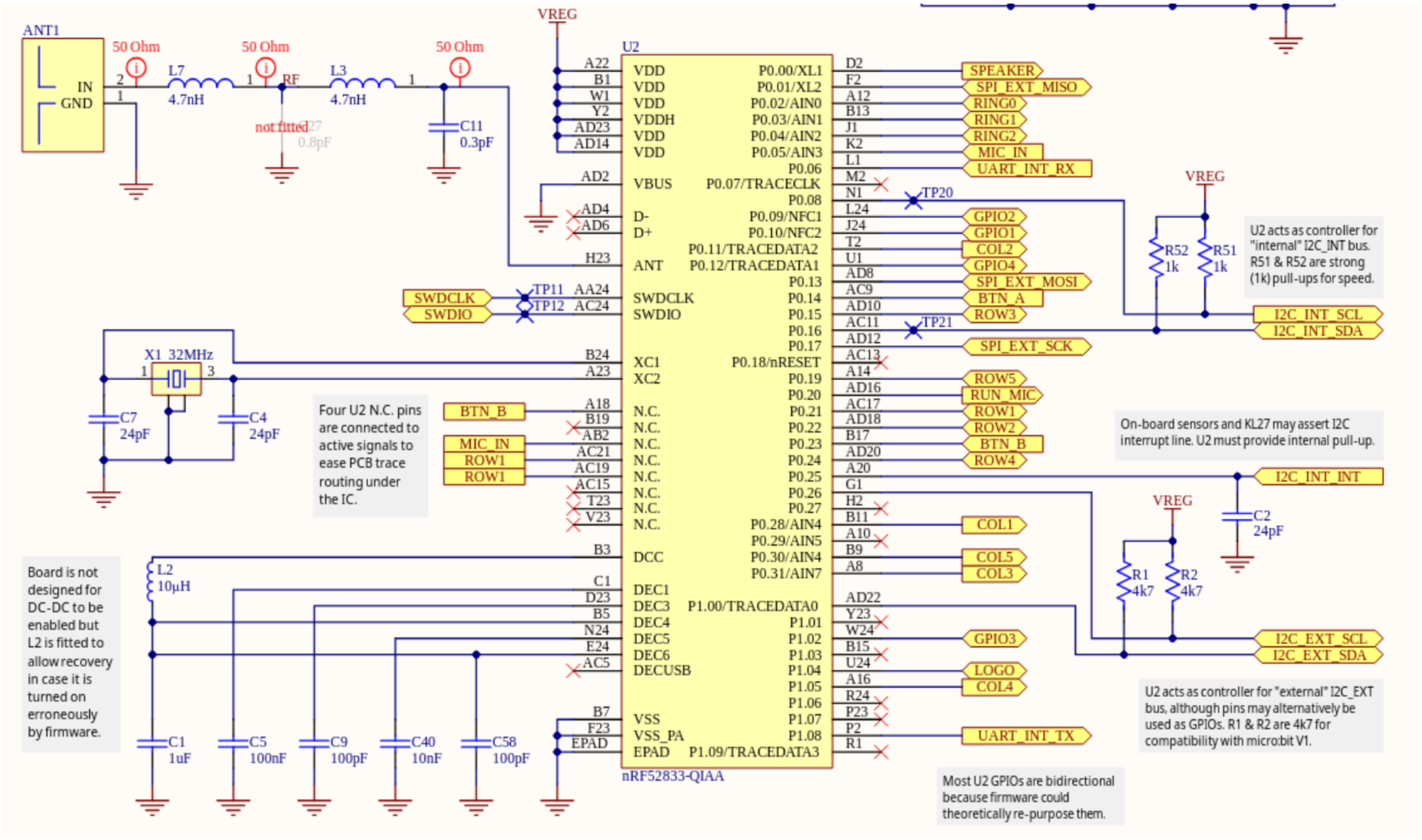
A link to all important instructions : [ARMv7-cheat-sheet.pdf]()

NXP KL27Z

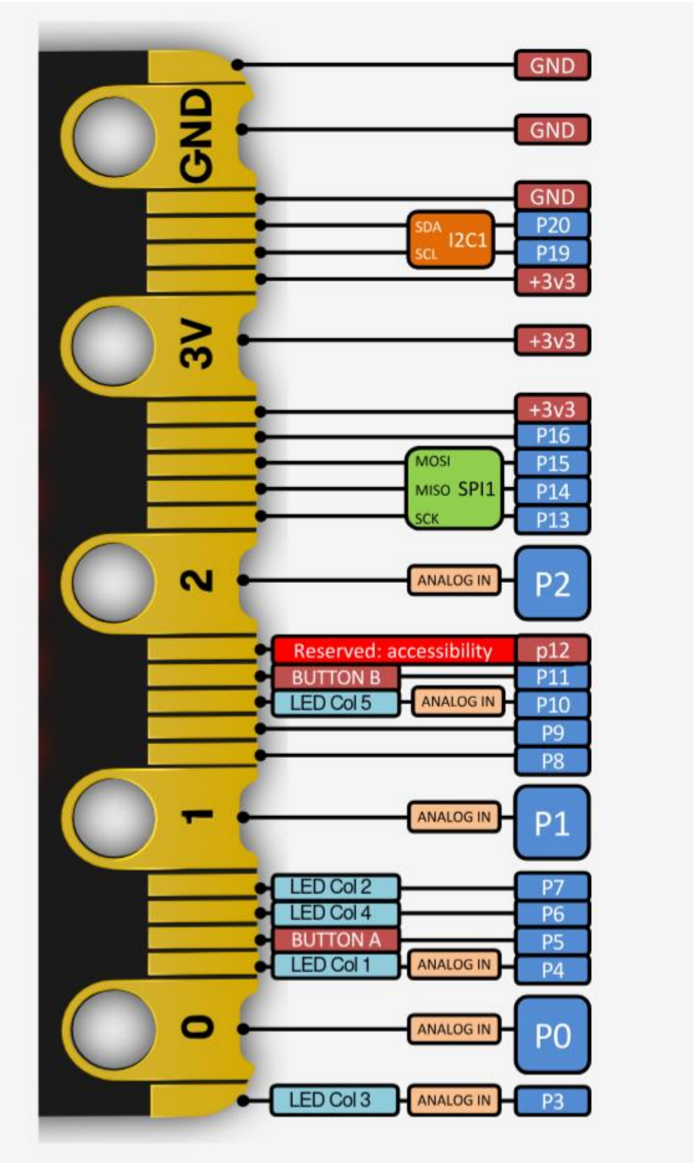


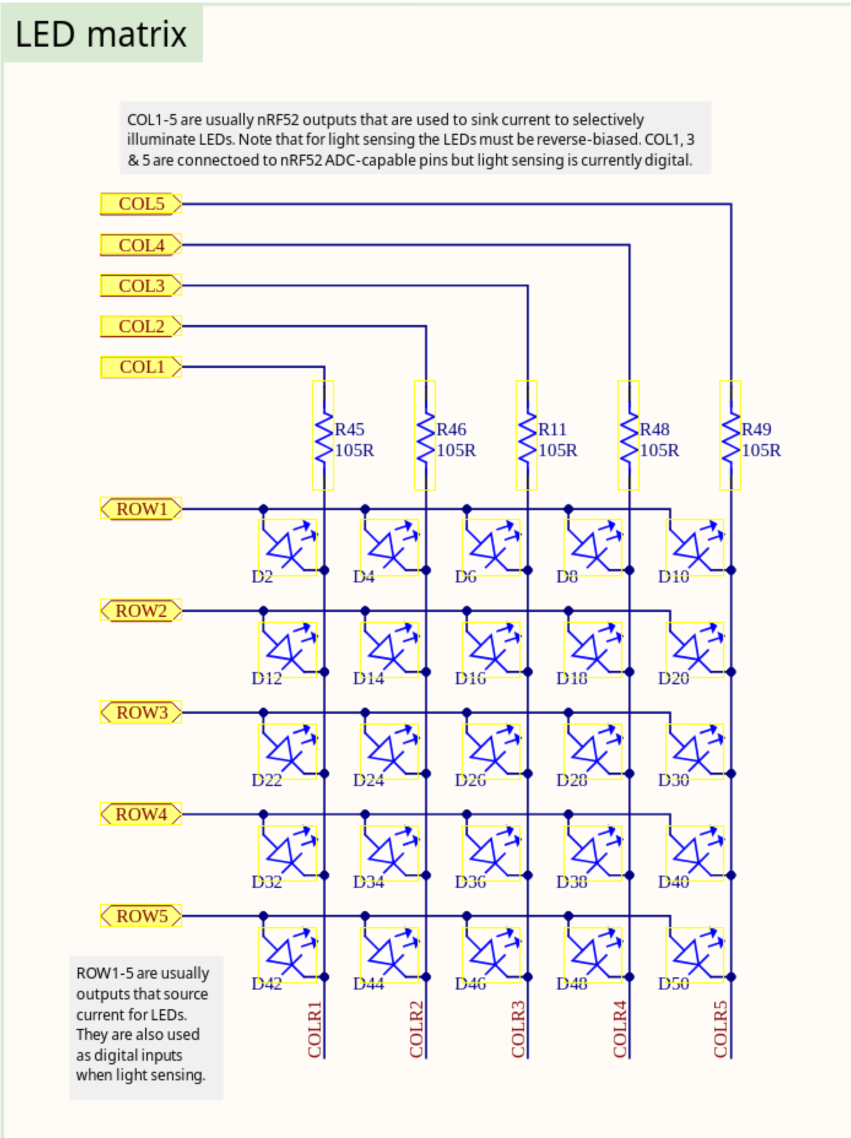
- The interface chip handles the USB connection
- And is used for flashing new code
- Sending and receiving serial data back and forth to your main computer.
- Helps in Real-Time Debugging

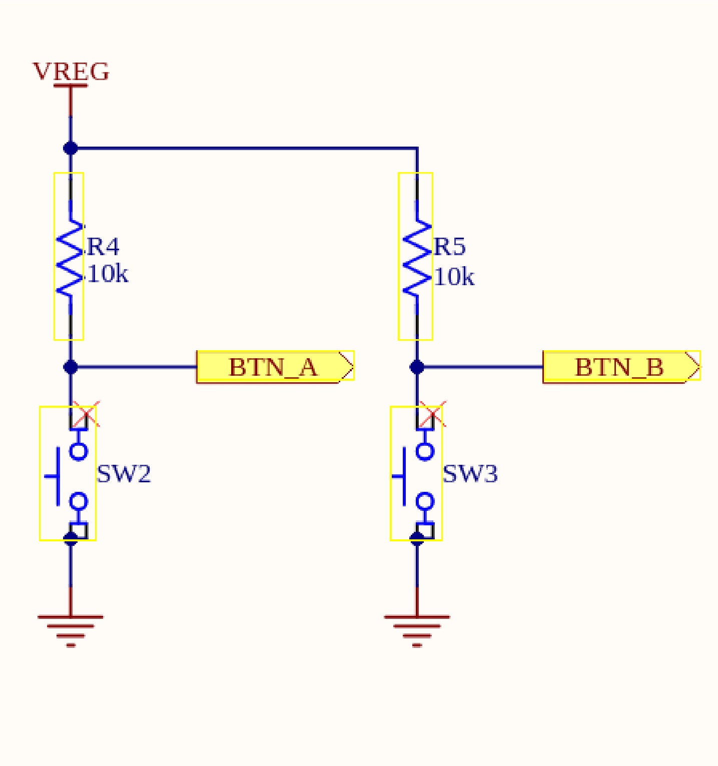
Pin configuration of our Micro:Bit



Pin configuration of our Micro:Bit







NOTE:
The buttons are **active low** and **passive high**

Pin configuration of our Micro:Bit

GPIO on nRF52833	Allocation	Interface (KL27 / nRF52)	Edge Connector name
P0.00	SPEAKER	KL27_DAC / IF_SPEAKER	
P1.05	COL4	N	P6
P0.02	RING0	N	P0
P0.03	RING1	N	P1
P0.04	RING2	N	P2
P0.05	MIC_IN	N	
P0.06	UART_INT_RX	PTA18 / P0.03	
P1.08	UART_INT_TX	PTA19 / P0.02	
P0.08	I2C_INT_SCL	PTC1 / P0.29	
P0.10	GPIO1	N	P8
P0.09	GPIO2	N	P9
P0.11	COL2	N	P7
P1.02	GPIO3	N	P16
P0.19	ROW5	N	
P0.14	BTN_A	N	P5
P0.23	BTN_B	N	P11
P1.04	FACE_TOUCH	N	
P0.16	I2C_INT_SDA	PTC2 / P0.28	
P0.17	SCK_EXTERNAL	N	P13
P0.01	MISO_EXTERNAL	N	P14
P0.13	MOSI_EXTERNAL	N	P15
P0.20	RUN_MIC	N	
P0.21	ROW1	N	
P0.22	ROW2	N	
P0.15	ROW3	N	
P0.24	ROW4	N	
P0.25	COMBINED_SENSOR_INT	PTA1 / P0.09	
P0.26	I2C_EXT_SCL	N	P19
P1.00	I2C_EXT_SDA	N	P20
P0.12	GPIO4	N	P12
P0.28	COL1	N	P4
P0.31	COL3	N	P3
P0.30	COL5	N	P10

Memory Map

- The nRF52833 contains 512 kB of flash memory and 128 kB of RAM that can be used for code and data storage.
- The CPU and peripherals with EasyDMA can access memory
- EasyDMA is a module implemented by some peripherals to gain direct access to Data RAM.
- EasyDMA is not able to access flash
- A peripheral can implement multiple EasyDMA instances to provide dedicated channels. For example, for reading and writing of data between the peripheral and RAM.

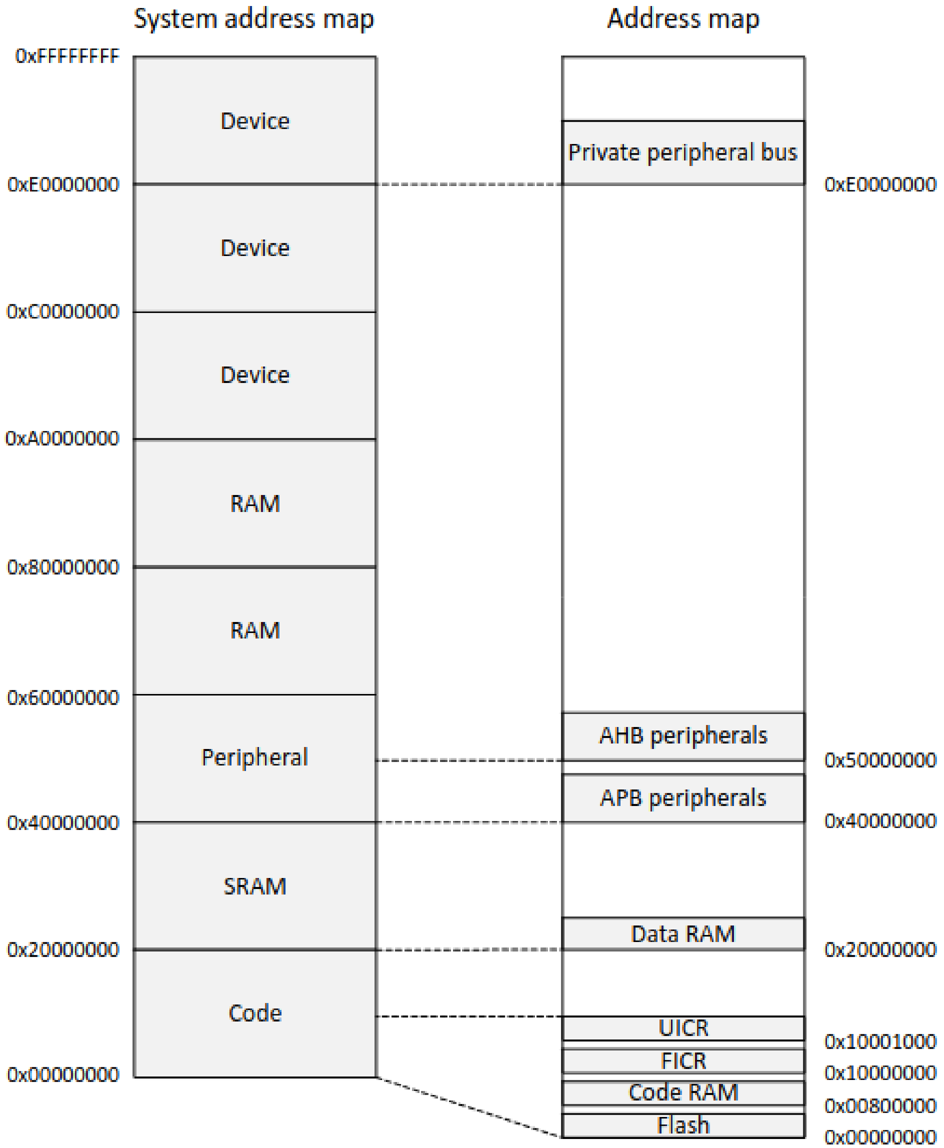
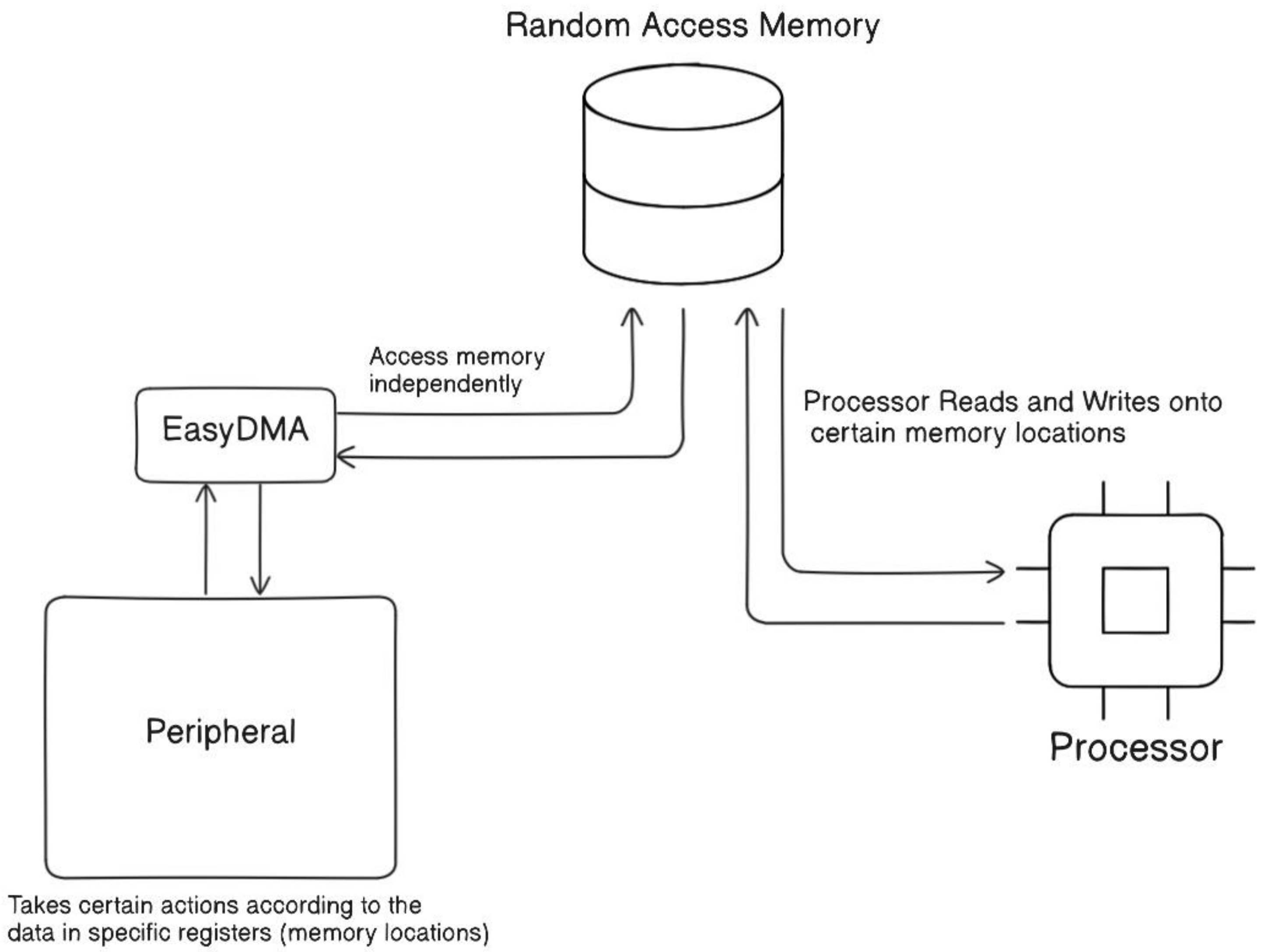


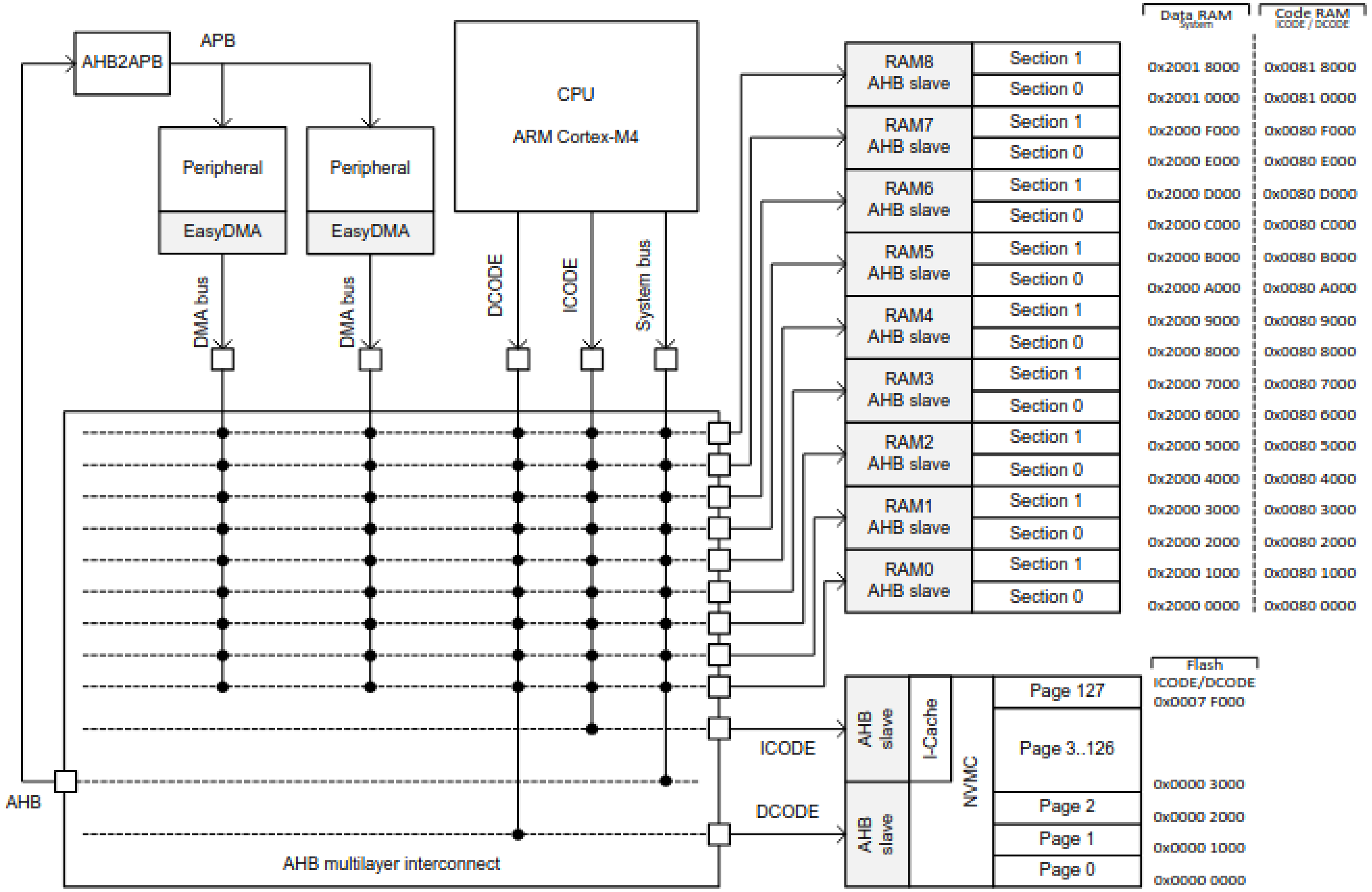
Figure 3: Memory map

EasyDMA



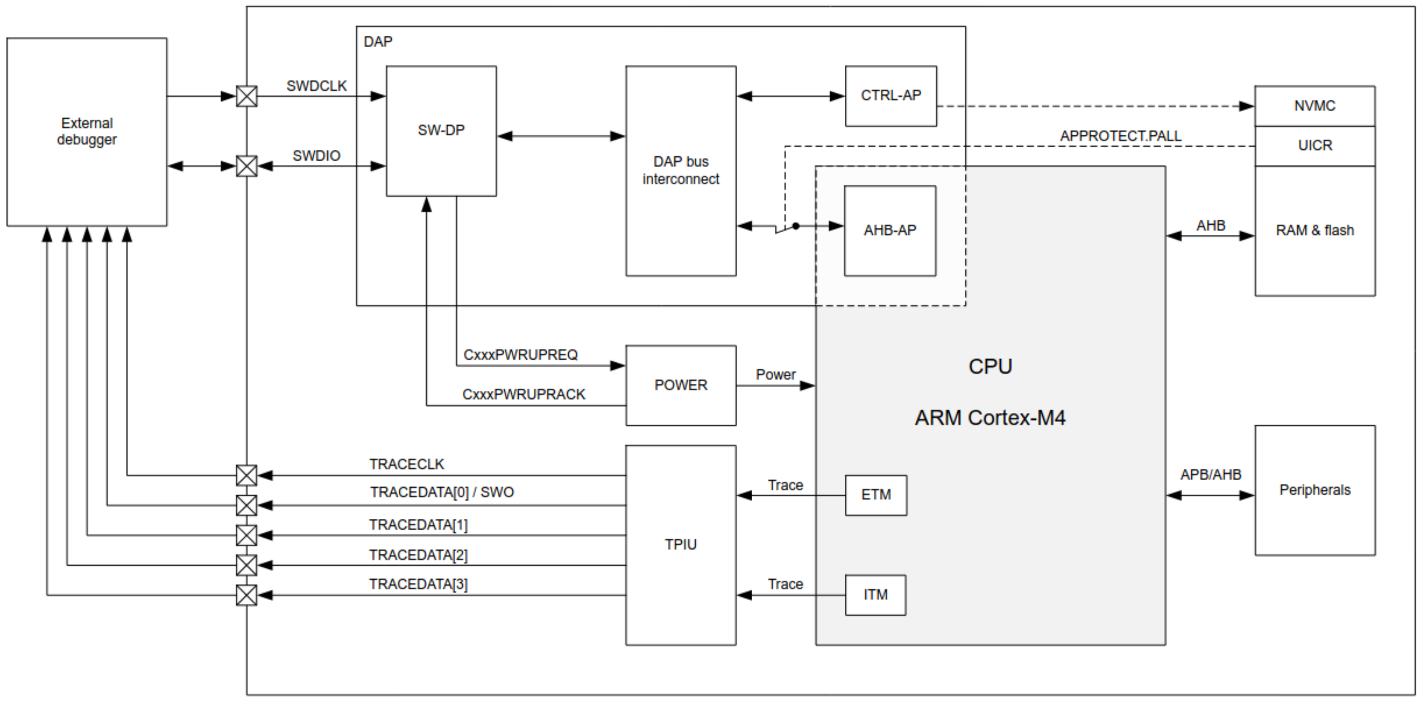
Random Access Memory
Access memory
independently
EasyDMA
Processor Reads and Writes onto
certain memory locations
Peripheral
Processor
Takes certain actions according to the
data in specific registers (memory locations)

Some Extra Details (optional)



Real-time debug

- The nRF52833 supports real-time debugging.
- Real-time debugging allows interrupts to execute to completion in real time when breakpoints are set in thread mode or lower priority interrupts.
- We will use this feature through our VS code extension



Peripherals

The nRF52833 is loaded with peripherals like:

- Comparators
- **GPIOs   (general purpose input output pins)**
- **GPIOTEs (GPIO tasks and events)**
- **PWM (pulse width modulation)**
- Radio (2.4 GHz)
- RNG (Random Number Generator)
- RTC (Real Time Counter)
- SPI (Serial Peripheral Interface)
- Timers
- **I2C communication module**
- **EasyDMA**

*The peripherals in bold are important ones and relatively easy to work with

Peripheral interface

- Peripherals are controlled by the CPU by writing to configuration registers and task registers.
- Peripheral events are indicated to the CPU by event registers and interrupts if they are configured for a given event.
- Every peripheral is assigned a fixed block of 0x1000 bytes of address space, which is equal to 1024 x 32 bit registers.
- Most peripherals feature an ENABLE register.
- The peripheral must be enabled before tasks and events can be used.
- <u>**Peripherals access the memory using EasyDMA (easy direct memory access)**</u>

ID	Base address	Peripheral	Instance	Description
0	0x40000000	CLOCK	CLOCK	Clock control
0	0x40000000	POWER	POWER	Power control
0	0x50000000	GPIO	GPIO	General purpose input and output
0	0x50000000	GPIO	P0	General purpose input and output, port 0
0	0x50000300	GPIO	P1	General purpose input and output, port 1
1	0x40001000	RADIO	RADIO	2.4 GHz radio
2	0x40002000	UART	UART0	Universal asynchronous receiver/transmitter
2	0x40002000	UARTE	UARTE0	Universal asynchronous receiver/transmitter with EasyDMA, unit 0
3	0x40003000	SPI	SPI0	SPI master 0
3	0x40003000	SPIM	SPIM0	SPI master 0
3	0x40003000	SPIS	SPIS0	SPI slave 0
3	0x40003000	TWI	TWI0	Two-wire interface master 0
3	0x40003000	TWIM	TWIM0	Two-wire interface master 0
6	0x40006000	GPIOTE	GPIOTE	GPIO tasks and events
7	0x40007000	SAADC	SAADC	Analog to digital converter
8	0x40008000	TIMER	TIMER0	Timer 0
9	0x40009000	TIMER	TIMER1	Timer 1
10	0x4000A000	TIMER	TIMER2	Timer 2
11	0x4000B000	RTC	RTC0	Real-time counter 0
12	0x4000C000	TEMP	TEMP	Temperature sensor
13	0x4000D000	RNG	RNG	Random number generator
14	0x4000E000	ECB	ECB	AES electronic code book (ECB) mode block encryption
15	0x4000F000	AAR	AAR	Accelerated address resolver

Tasks and Events

- Tasks are used to trigger actions in a peripheral.
- A peripheral can implement multiple tasks with each task having a separate register in that peripheral's task register group.
- A task is triggered when firmware writes 1 to the task register, or when the peripheral itself or another peripheral toggles the corresponding task signal.

- Events are used to notify peripherals and the CPU about events that have happened
- For example, a state change in a peripheral.
- A peripheral may generate multiple events with each event having a separate register in that peripheral’s event register group.
- An event register is only cleared when firmware writes 0 to it.

Example : Dummy Peripheral

Lets, say our dummy peripheral has the base address of 0x30004000 and there are several registers that control this peripheral . One of them is shown below:

Register	Offset	Description
DUMMY	0x514	Example of a register controlling a dummy feature

Real address of this register in memory will be: Base + Offset = 0x30004514

Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID					D	D	D	D						C	C	C								B							A	A
Reset 0x00050002	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**

ID	Acce	Field	Value ID	Value	Description	
A	RW	FIELD_A			Example of a read-write field with several enumerated values	
			Disabled	0	The example feature is disabled	
			NormalMode	1	The example feature is enabled in normal mode	
			ExtendedMode	2	The example feature is enabled along with extra functionality	
B	RW	FIELD_B			Example of a deprecated read-write field	Deprecated
			Disabled	0	The override feature is disabled	
			Enabled	1	The override feature is enabled	
C	RW	FIELD_C			Example of a read-write field with a valid range of values	
			ValidRange	[2..7]	Example of allowed values for this field	
D	RW	FIELD_D			Example of a read-write field with no restriction on the values	

As you can see there are some special bits that hold some important meaning.

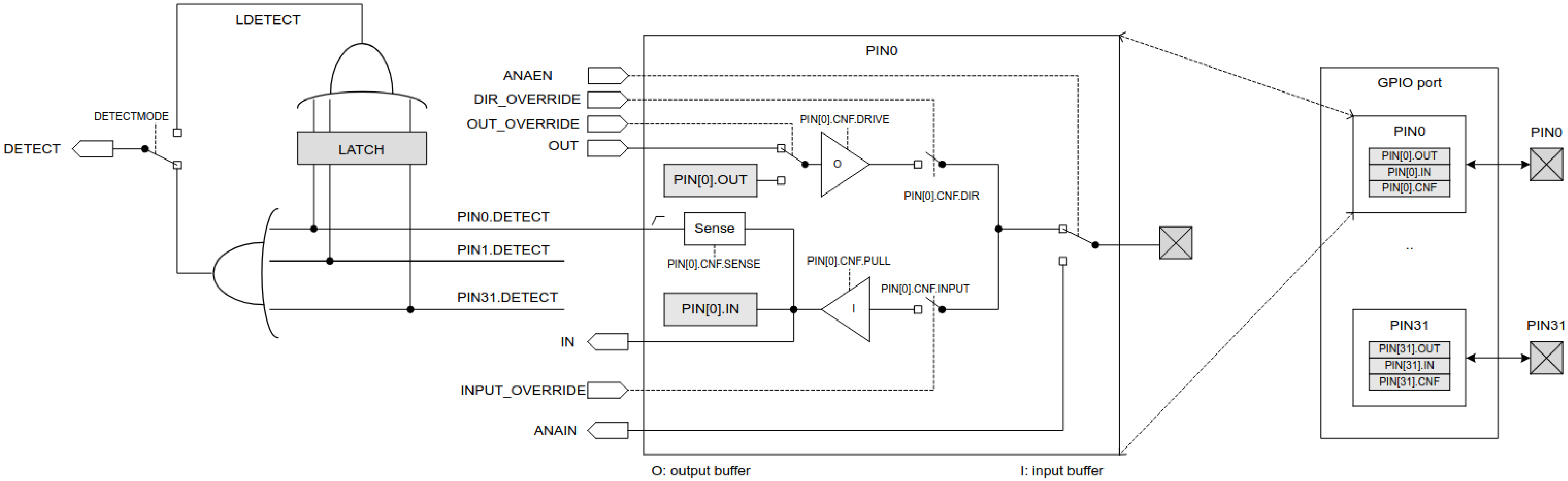
We can configure our peripherals using such registers.

The peripheral will continuously access this register using easyDMA and make the necessary changes as soon as the CPU makes changes in the content of the register

Controlling GPIOS

- The general purpose input/output pins (GPIOs) are grouped 2 Ports with each port having up to 32 GPIOs

• The GPIO port peripheral implements up to 32 pins, PIN0 through PIN31. Each of these pins can be individually configured in the PIN_CNF[n] registers (n=0..31)



Controlling GPIOS

- The general purpose input/output pins (GPIOs) are grouped 2 Ports with each port having up to 32 GPIOs

- The GPIO port peripheral implements up to 32 pins, PIN0 through PIN31. Each of these pins can be individually configured in the PIN_CNF[n] registers (n=0..31)

6.8.2 Registers

Base address	Peripheral	Instance	Description	Configuration	
0x50000000	GPIO	GPIO	General purpose input and output		Deprecated
0x50000000	GPIO	P0	General purpose input and output, port 0	P0.00 to P0.31 implemented	
0x50000300	GPIO	P1	General purpose input and output, port 1	P1.00 to P1.09 implemented	

Controlling GPIOS

- To start using GPIO pins , we have to first configure them using the CNF Registers.

6.8.2.10 PIN_CNF[n] (n=0..31)

Address offset: 0x700 + (n × 0x4)

Configuration of GPIO pins

Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID															E	E						D	D	D					C	C	B	A
Reset 0x00000002	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**

ID	Acce	Field	Value ID	Value	Description
A	RW	DIR			Pin direction. Same physical register as DIR register
			Input	0	Configure pin as an input pin
			Output	1	Configure pin as an output pin
B	RW	INPUT			Connect or disconnect input buffer
			Connect	0	Connect input buffer

Controlling GPIOS

- To start using GPIO pins , we have to first configure them using the CNF Registers.

6.8.2.10 PIN_CNF[n] (n=0..31)

Address offset: 0x700 + (n × 0x4)

Configuration of GPIO pins

Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ID															E	E
Reset 0x00000002	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Bit number	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID						D	D	D					C	C	B	A
Reset 0x00000002	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**

ID	Acce	Field	Value ID	Value	Description
A	RW	DIR			Pin direction. Same physical register as DIR register
			Input	0	Configure pin as an input pin
			Output	1	Configure pin as an output pin
B	RW	INPUT			Connect or disconnect input buffer
			Connect	0	Connect input buffer
C	RW	PULL			Pull configuration
			Disabled	0	No pull
			Pulldown	1	Pull down on pin
			Pullup	3	Pull up on pin

We can configure PIN 13 of port P0 to be an output pin with pulldown register by writing :
0b0101 to the register at address
= Base + offset
= 0x50000000 + 0x700 + (0x4)*13

Controlling GPIOS

C	RW	PULL			Pull configuration
			Disabled	0	No pull
			Pulldown	1	Pull down on pin
			Pullup	3	Pull up on pin
D	RW	DRIVE			Drive configuration
			S0S1	0	Standard '0', standard '1'
			H0S1	1	High drive '0', standard '1'
			S0H1	2	Standard '0', high drive '1'
			H0H1	3	High drive '0', high 'drive '1''
			D0S1	4	Disconnect '0' standard '1' (normally used for wired-or connections)
			D0H1	5	Disconnect '0', high drive '1' (normally used for wired-or connections)
			S0D1	6	Standard '0'. disconnect '1' (normally used for wired-and connections)
			H0D1	7	High drive '0', disconnect '1' (normally used for wired-and connections)
E	RW	SENSE			Pin sensing mechanism
			Disabled	0	Disabled
			High	2	Sense for high level
			Low	3	Sense for low level

- For our purpose option B , D and E are not much useful

Controlling GPIOS

We have now successfully configured PIN : P0.13
As an output pin with pulldown register.

Now to set the pin HIGH , we need to write to address = Base + offset = 0x50000000 + **0x508**
Data to write : **0b000000000000000000100000000000000**
As soon as we write this data onto the memory location , the peripheral will access this location using EasyDMA and set the pin HIGH

6.8.2.2 OUTSET

Address offset: 0x508

Set individual bits in GPIO port

Read: reads value of OUT register.

Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID	f	e	d	c	b	a	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A
Reset 0x00000000	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

ID	Acce	Field	Value ID	Value	Description
A-f	RW	PIN[i] (i=0..31)			Pin i
			Low	0	Read: pin driver is low
			High	1	Read: pin driver is high
			Set	1	Write: writing a '1' sets the pin high; writing a '0' has no effect

Controlling GPIOS

We have now successfully configured PIN : P0.13
As an output pin with pulldown register.

Now to set the pin **LOW**, we need to write to address = Base + offset = 0x50000000 + **0x50C**
Data to write **: 0b00000000000000000100000000000000**
As soon as we write this data onto the memory location , the peripheral will access this location using EasyDMA and set the pin **LOW**

6.8.2.3 OUTCLR

Address offset: 0x50C

Clear individual bits in GPIO port

Read: reads value of OUT register.

Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID	f	e	d	c	b	a	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A
Reset 0x00000000	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

ID	Acce	Field	Value ID	Value	Description
A-f	RW	PIN[i] (i=0..31)			Pin i
			Low	0	Read: pin driver is low
			High	1	Read: pin driver is high
			Clear	1	Write: writing a '1' sets the pin low; writing a '0' has no effect

Controlling GPIOS

We have now successfully configured PIN : P0.13
As an output pin with pulldown register.

Now to set the pin **LOW**, we need to write to address = Base + offset = 0x50000000 + **0x50C**
Data to write **: 0b00000000000000000010000000000000**
As soon as we write this data onto the memory location , the peripheral will access this location using EasyDMA and set the pin **LOW**

6.8.2.3 OUTCLR

Address offset: 0x50C

Clear individual bits in GPIO port

Read: reads value of OUT register.

Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID	f	e	d	c	b	a	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A
Reset 0x00000000	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

ID	Acce	Field	Value ID	Value	Description
A-f	RW	PIN[i] (i=0..31)			Pin i
			Low	0	Read: pin driver is low
			High	1	Read: pin driver is high
			Clear	1	Write: writing a '1' sets the pin low; writing a '0' has no effect

Controlling GPIOS

- So far, we have only seen how to control the pins as output.
- To take input (maybe from the onboard buttons) we need to configure those pins (as specified in pinout) as output pins.
- After that we can simply look at the memory location of = Base addr of port + 0x510
- Each of the 32 bits will indicate the logic level at those pins at that moment.
- For example , if Bit-9 == 1 then it means pin9 of that port is high .

6.8.2.4 IN

Address offset: 0x510

Read GPIO port

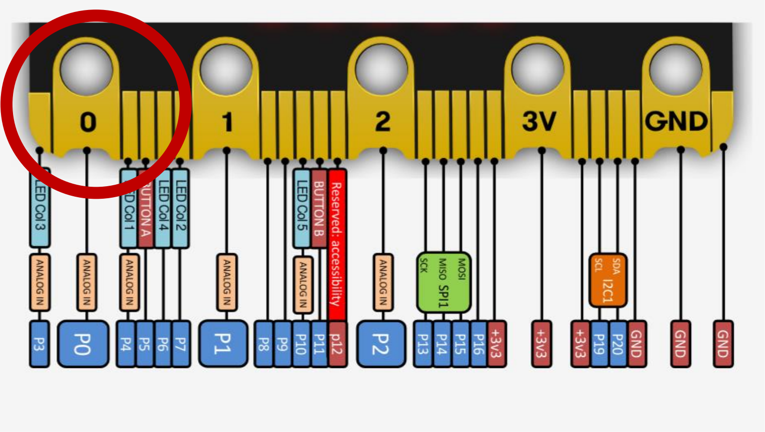
Bit number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID	f	e	d	c	b	a	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A
Reset 0x00000000	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

ID	Acce	Field	Value ID	Value	Description
A-f	R	PIN[i] (i=0..31)			Pin i
			Low	0	Pin input is low
			High	1	Pin input is high

Code for a simple Blink LED program

```
.syntax unified
.section .text
.global main

main:

LDR R0, =0x50000000       @ Load GPIO Port 0 base address
LDR R1, =0b100               @ Bit mask for Pin 2 (0b000000100)


@ Configure P0.02 as OUTPUT (PIN_CNF[2] = 0x700 + 2*0x4)
LDR R2, =0x708            @ Base offset of PIN_CNF
MOV R3, #1                @ Set direction to OUTPUT (1)
STR R3, [R0, R2]          @ Store configuration to PIN_CNF[0]
```

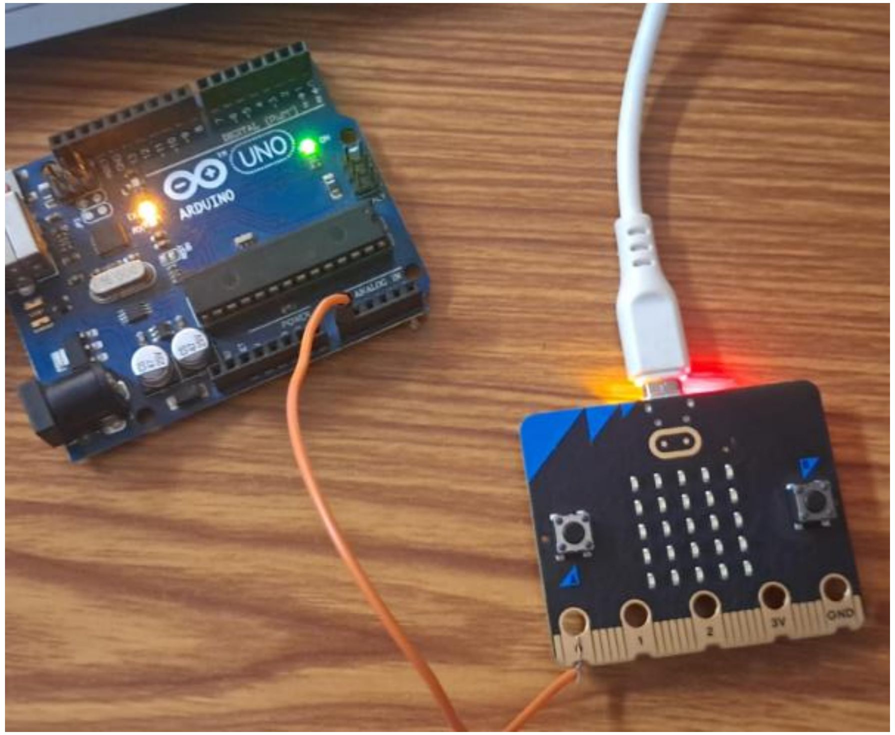
```
loop:

    @ Turn ON (Set P0.02 HIGH)
    LDR R2, =0x508
    STR R1, [R0, R2]

    @ Simple delay loop
    LDR R5, =5000000
    delay_on:
        SUB R5, R5, #1
        CMP R5, #0
        BGT delay_on

    @ Turn OFF (Set P0.02 LOW)
    LDR R2, =0x50C
    STR R1, [R0, R2]

    @ Simple delay loop
    LDR R5, =5000000
    delay_off:
        SUB R5, R5, #1
        CMP R5, #0
        BGT delay_off

B loop
```



GPIO P0.02 is connected to RING-0 of the microbit

Code for a simple Blink LED program

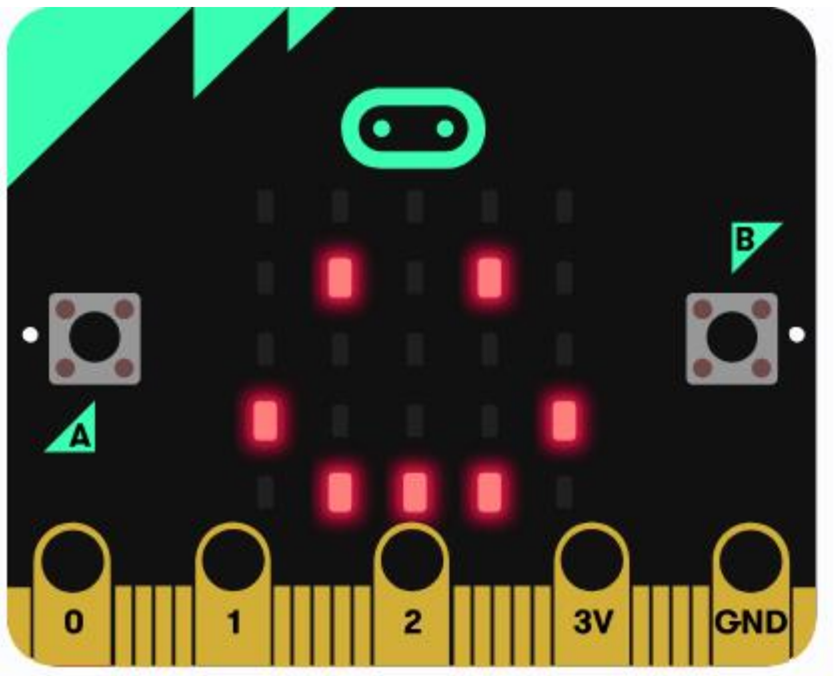
I plotted the pin output using another microcontroller (Arduino) , and here is the result.

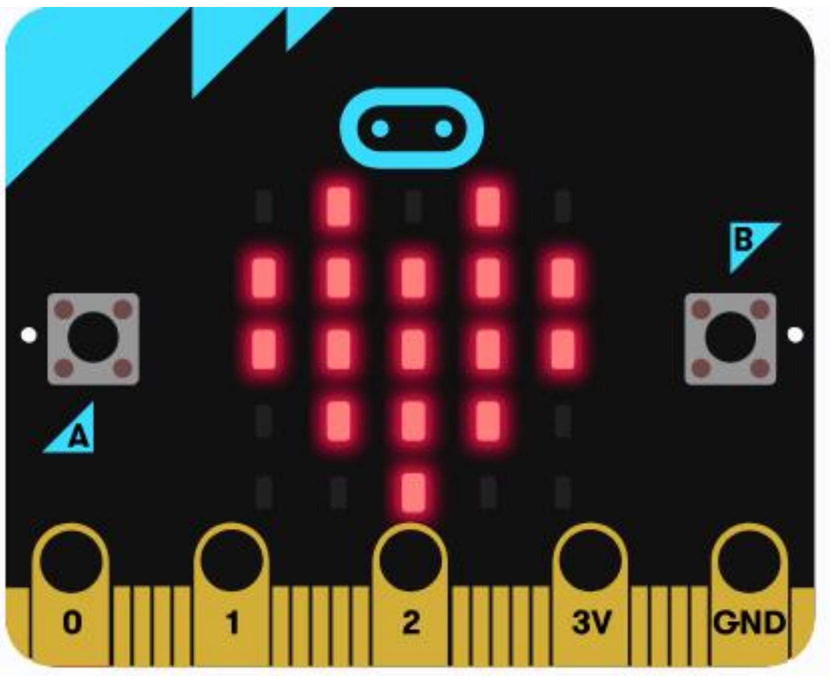


*I only connected the Pin and I was able to measure the voltage because the GND was internally connected through my laptop

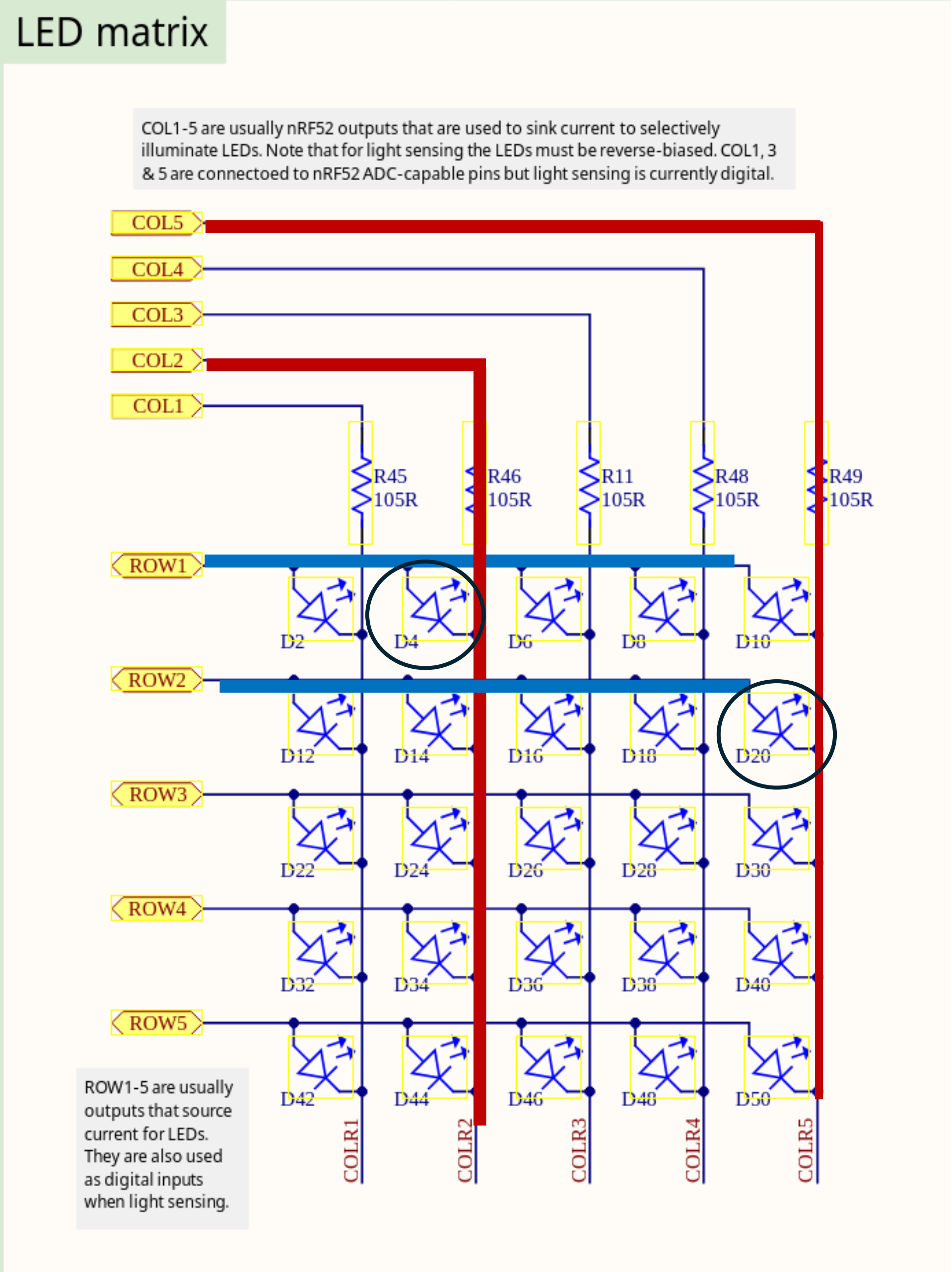
Controlling the LED MATRIX

The display is a 5x5 array of LEDs.



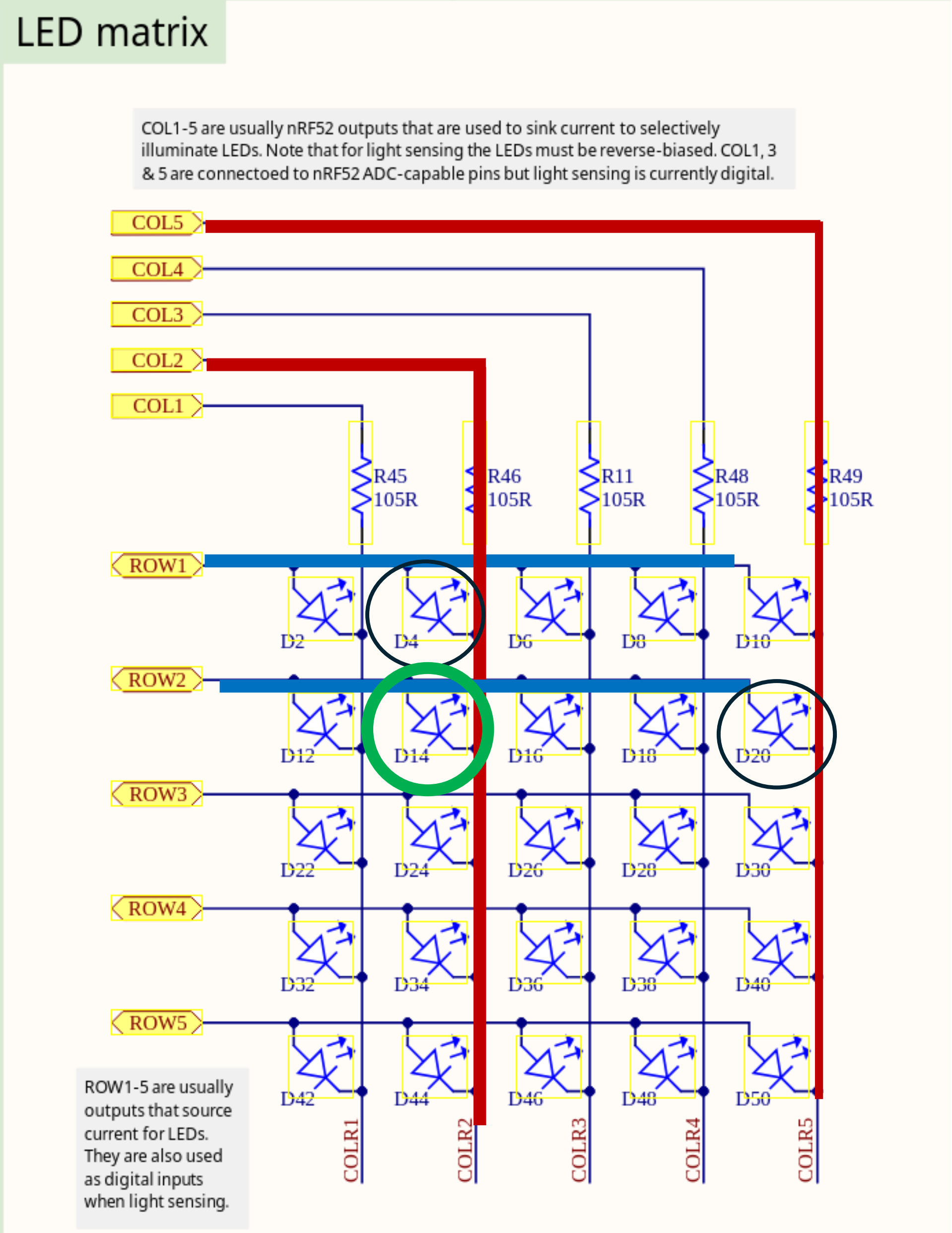


Controlling the LED MATRIX



- The 25 LEDs are wired in a row-column configuration to reduce the number of required control pins.
- To turn on D4 led , Row1 GPIO must be HIGH and Col2 GPIO should be set to LOW
- Now if I want to simultaneously turn on D20 , I will set Row2 to be LOW and COL5 to be HIGH
- **Can you find the problem with this ???**

Controlling the LED MATRIX



- The 25 LEDs are wired in a row-column configuration to reduce the number of required control pins.
- To turn on D4 led , Row1 GPIO must be HIGH and Col2 GPIO should be set to LOW
- Now if I want to simultaneously turn on D20 , I will set Row2 to be LOW and COL5 to be HIGH
- **Can you find the problem with this ???**
- **D14 will also get turned on !**

Controlling the LED MATRIX

- You cannot turn on two LEDs in different rows and columns at the same time because they will create an unintended path due to shared rows and columns. This is called **ghosting** or **cross-activation**.

- To display multiple LEDs in different rows and columns, we will use a technique called **multiplexing** or **frame refreshing.**

- A full LED pattern is divided into separate row-wise frames.

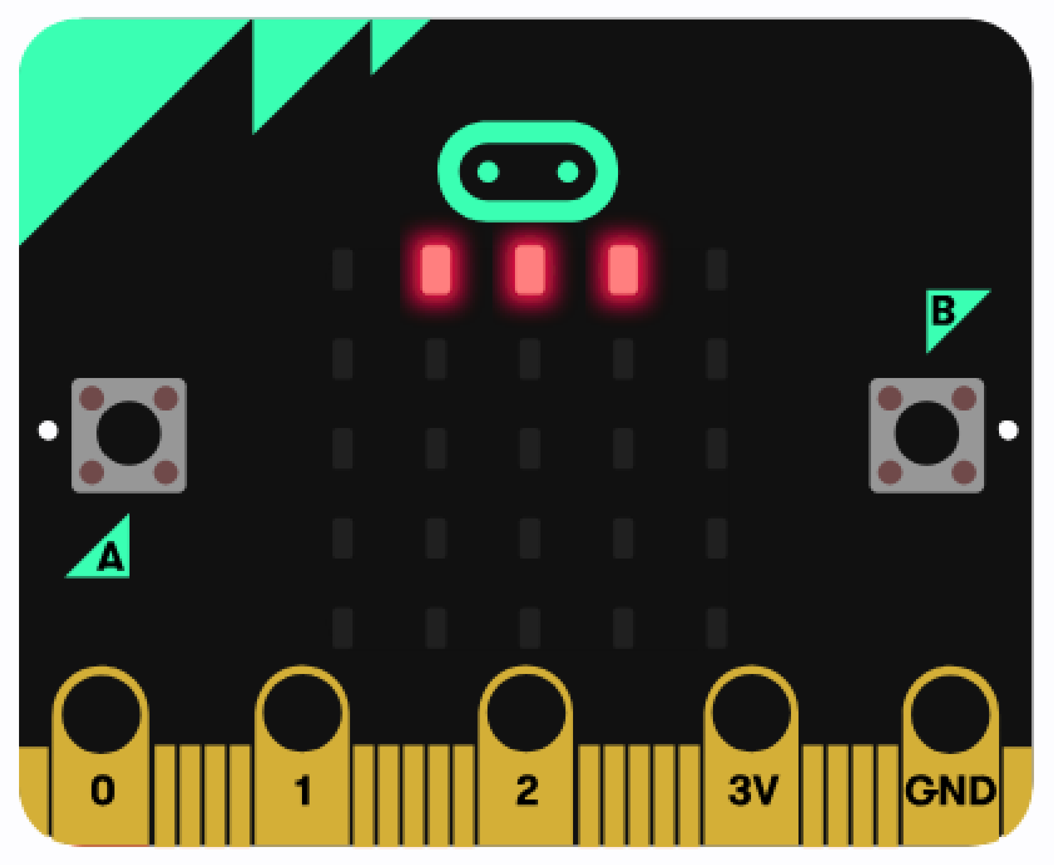
- Each frame represents one active row at a time.

- After a very short delay , we will deactivate the row and activate the next one.

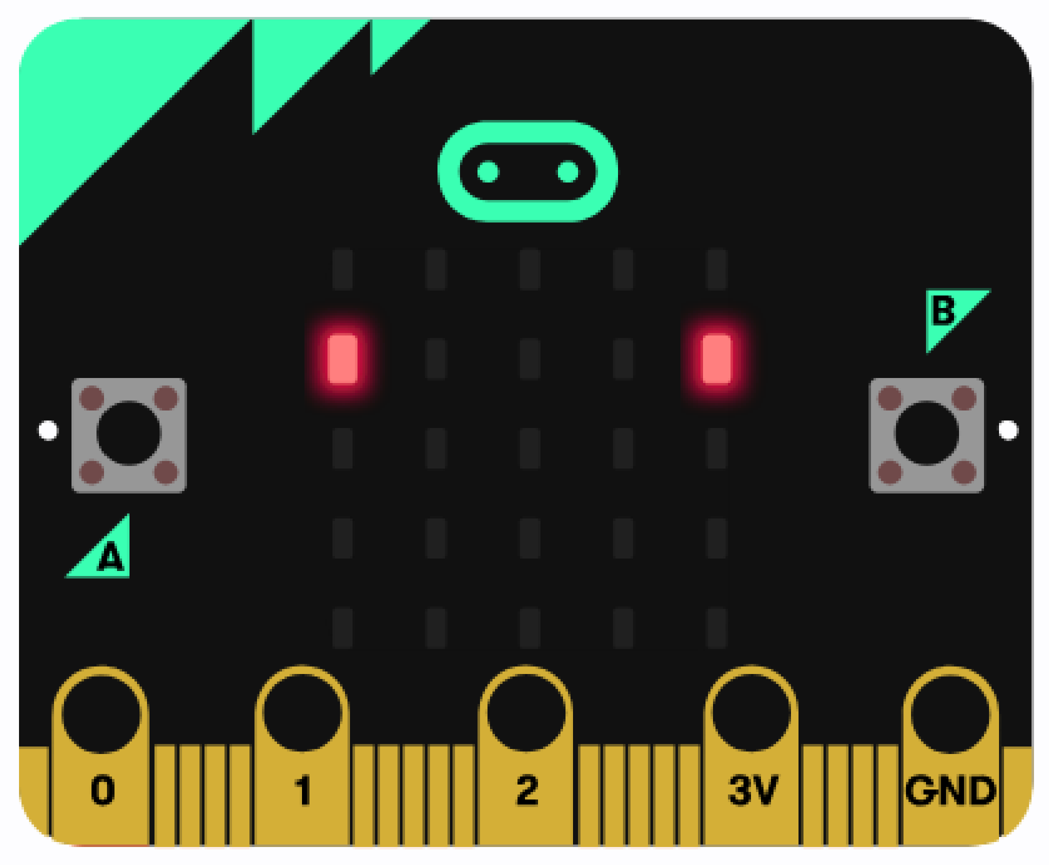
- Now if we do it fast enough , our human eyes will perceive it as if multiple LEDs across different rows and columns are ON simultaneously.

- Example of displaying "A" in the subsequent slides.

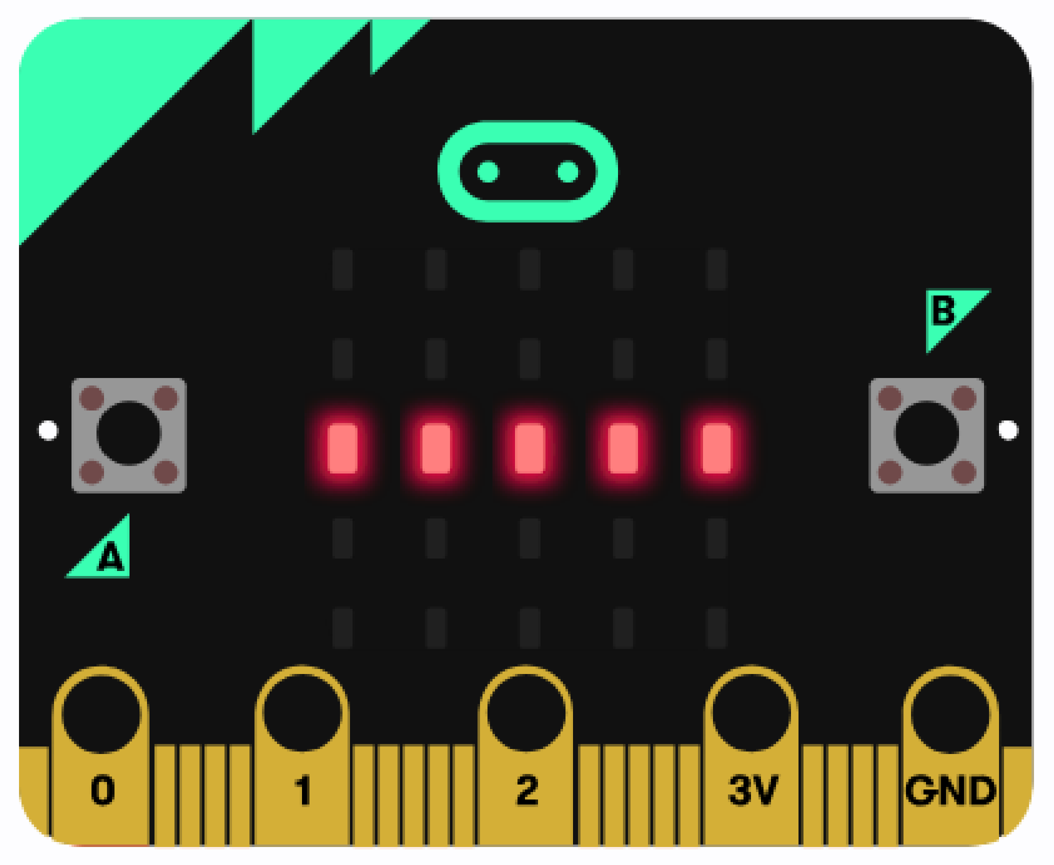
Controlling the LED MATRIX



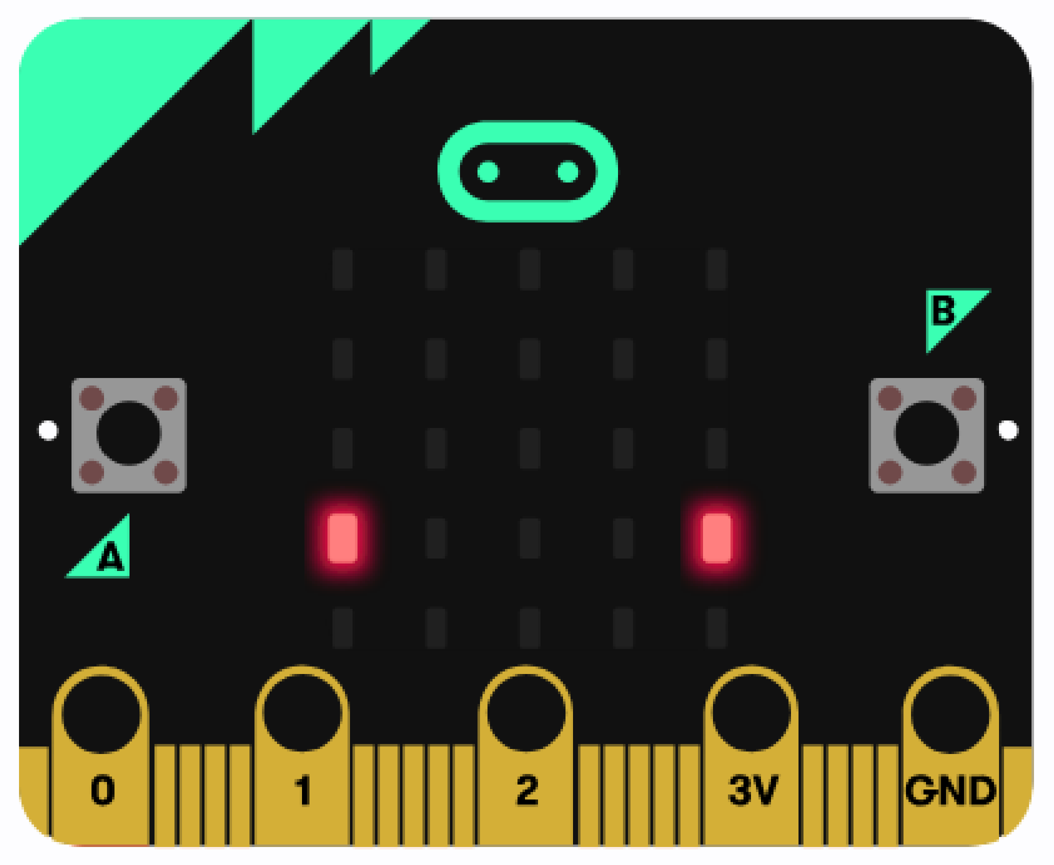
Controlling the LED MATRIX



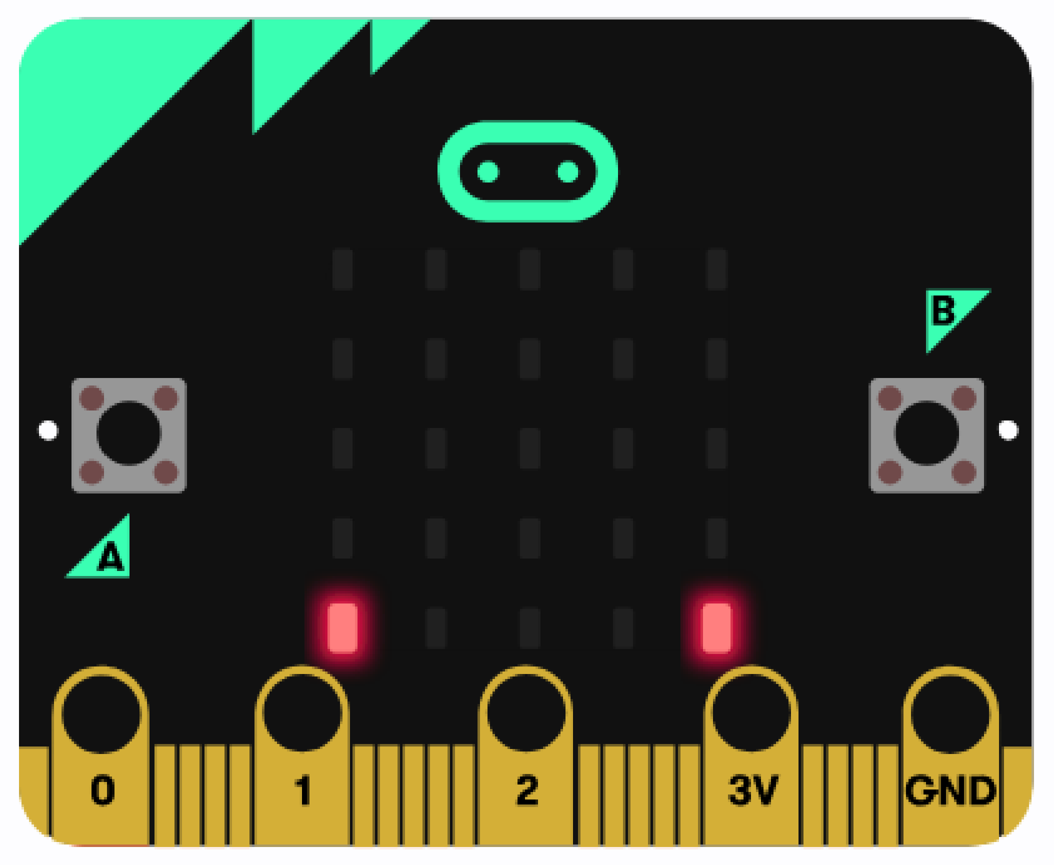
Controlling the LED MATRIX



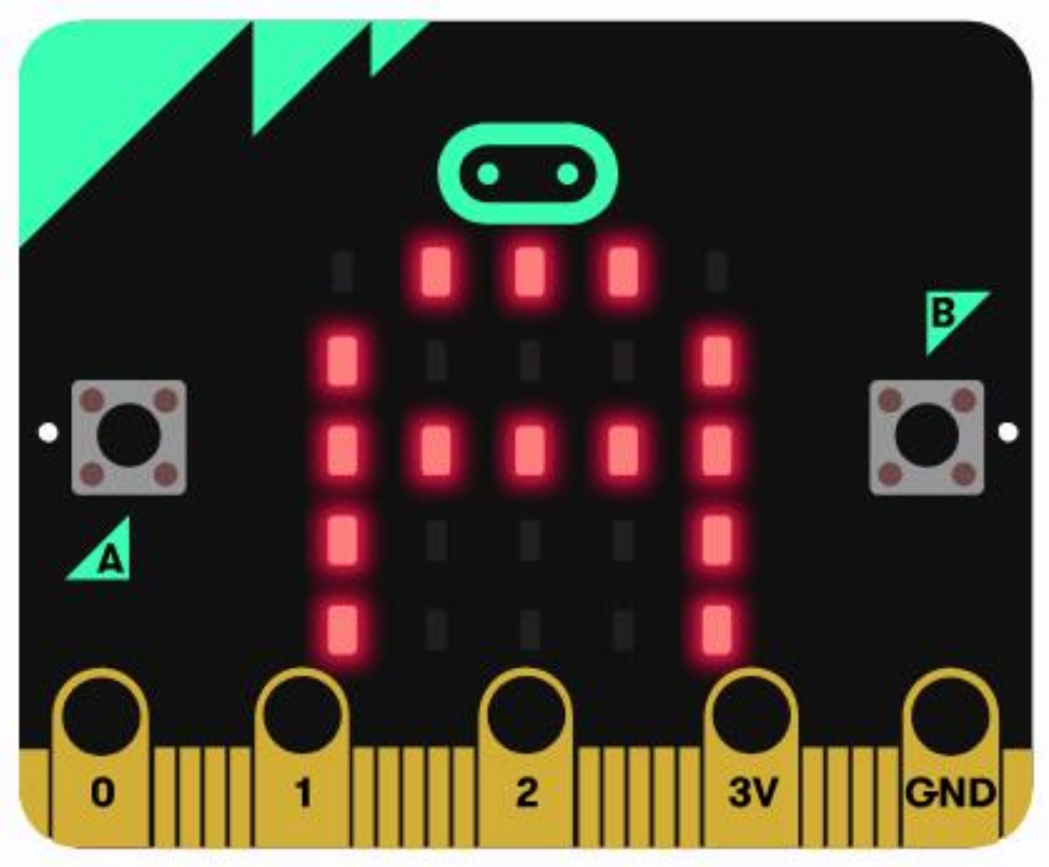
Controlling the LED MATRIX



Controlling the LED MATRIX



Controlling the LED MATRIX



Library to do the same.

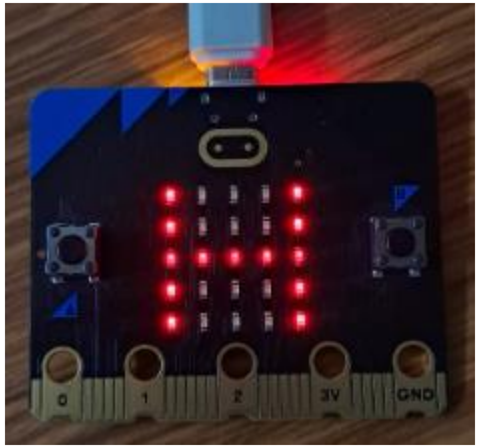
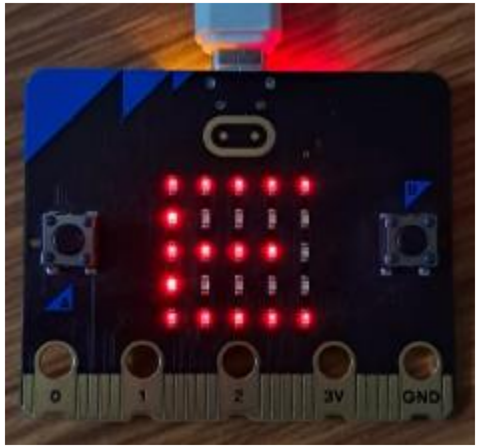
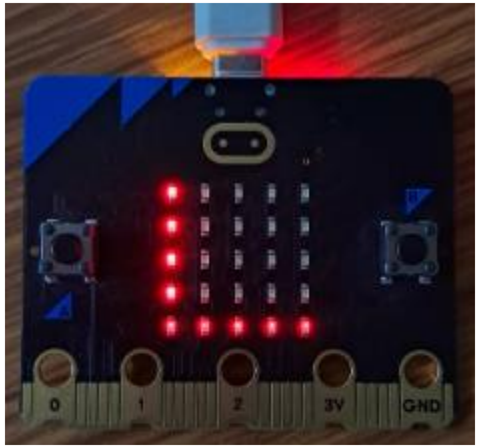
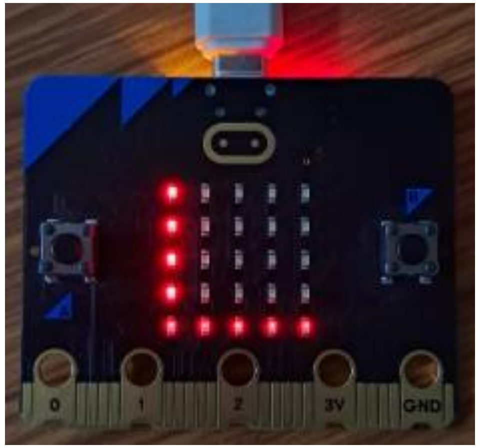
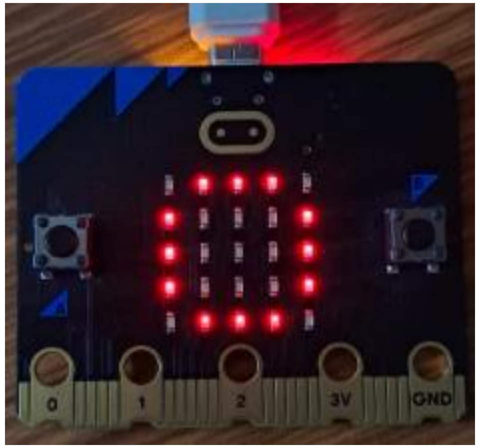
- There's a library called **Led.S**, made by **ANU-COMP2300 students**, that lets you control specific LEDs in a **5x5 matrix**.
- You can find it in **workfolder/lib**.
- Using that, I made another library called **text.S** to make things easier.
- Just load the **ASCII value** (capital letters or numbers) into **R0** and call the function—it'll handle the multiplexing for one cycle.
- If you want the character to stay on the screen, just keep calling it in a loop.

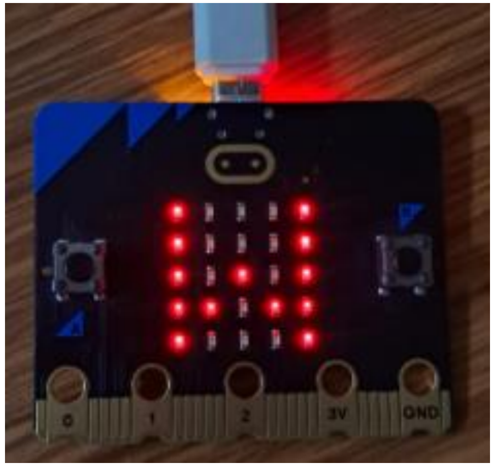
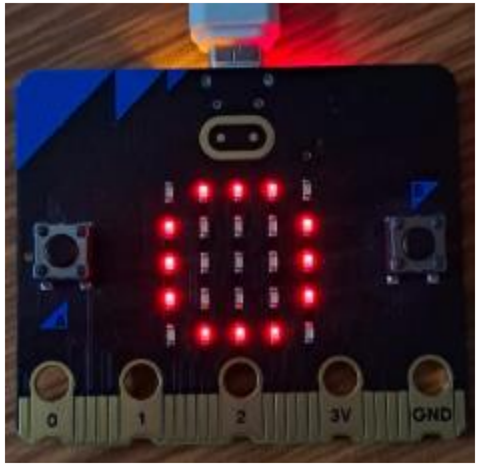
src > main.S

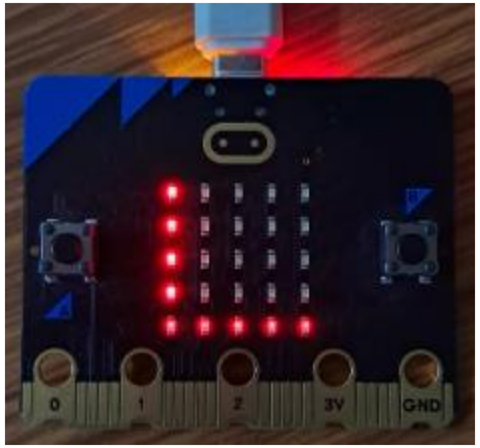
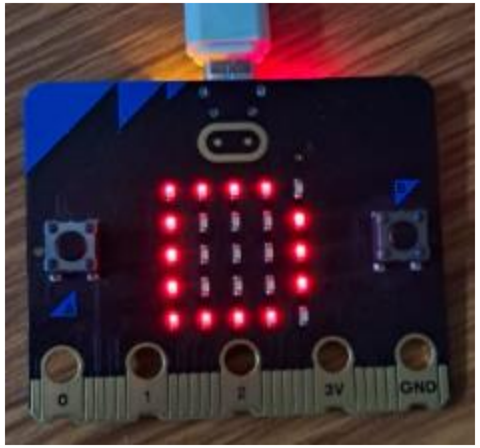
```
.syntax unified
.section .text
.global main

main:

bl init_text
loop:
ldr r0 , =65
bl display_char
b loop

```

- **bl init_text** Initializes the Row 1-5 and Col 1-5 Pins
- **bl display_char** takes the value in R0 and displays the character.

Library to do the same.

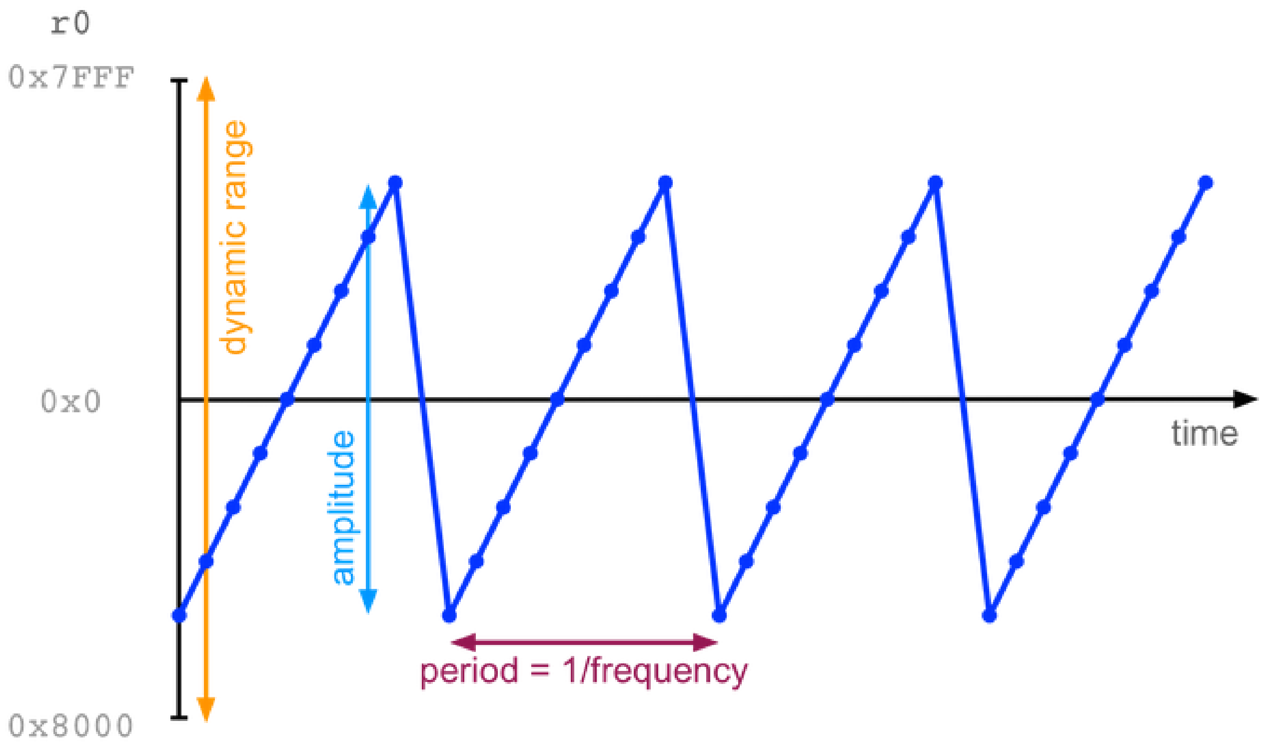






Controlling Speakers.

There is a cool library by Benjamin Gray ANU which lets you playout a certain sequence of audio sample.



In the image, the little points each indicate a sample, that is *a number* stored in r0.

The x-axis is time, and the y-axis is the number in r0.

- **audio_init**: this function sets up sound on the microbit which basically means setting the speaker (GPIO P0.00) to output, enabling PWM (pulse width modulation) on this pin, and setting up some buffers and timers to ensure smooth audio.

- **audio_play_sample**: this function takes r0 as an argument. It takes the lowest 8 bits of r0 and treats them as the next audio sample to play.

Controlling Speakers.

Using this library , I make another library through which you can play specific notes.

```
/* Play C4 (261.63 Hz) */
.type play_c4, %function
.global play_c4
play_c4:
    push {lr}
    ldr r0, =548288
    bl play_note
    pop {pc}

/* Play D4 (293.66 Hz) */
.type play_d4, %function
.global play_d4
play_d4:
    push {lr}
    ldr r0, =615424
    bl play_note
    pop {pc}
```

```
    bl play_e4
    bl play_e4
    bl play_e4
    bl play_e4
    bl play_e4
    bl play_stop
    bl play_c4
    bl play_c4
    bl play_stop
    bl play_c4
    bl play_c4
    bl play_d4
    bl play_d4
    bl play_d4
    bl play_d4
    bl play_d4
    bl play_c4
    bl play_c4
    bl play_c4
```

Then we can literally play any song / tune that we want.

There are more functions for all major notes.

Other Cool Stuff

This was just a glimpse of what can be done.

I did more complex things like:

- Using I2C protocol to talk to different sensors and retrieving data from them.
- Like Magnetometer and Accelerometer

- Using in-chip temperature sensor to get the current temperature (Although it might be slightly higher than real surrounding temperatures because of the heat produced by the MCU itself)

- Handling Interrupts using GPIOTE module.

Currently I am trying to use the onboard Radio/Bluetooth module , but it's very complex and requires a deep understanding of complex Bluetooth protocol.
If I can do it successfully , I will surely give an update

Thank you so much for this amazing opportunity! I had a great time experimenting and learning new things, especially diving deep into how microcontrollers actually work. I’ve worked with microcontrollers before, like **Arduino** and **ESP8266**, but always programmed them in **C/Python**. Writing in **assembly** gave me a whole new appreciation for the beauty of the architecture and how everything truly functions at a lower level.

References

- ARMv7-cheat-sheet.pdf (All important ISA instructions)
- ARMv7-M-architecture-reference-manual.pdf (Detailed ARM architecture - optional)
- lsm303agr_magetometerAndAccelerometer.pdf (On board Magnetometer and Accelerometer Datasheet)
- MicroBit_schematic.PDF (Micro:Bit official schematic)
- nRF52833.pdf (Main MCU Datasheet)
- https://tech.microbit.org/ (Official Website for technical details regarding Micro:Bit)

Thank You!