Embedded System Lab

Prof. Mehdi Pirahandeh
Electronic Engineering Department
Inha University

Email: *mehdi@inha.ac.kr*

Cortex
Intelligent Processors by ARM

인하대학교

Contents

Before we Start

1. Create a new project as last time

2. Download st_basic.h and st_basic.c files from learn.inha.ac.kr

3. Add downloaded files to your project

   - Select "**Add Existing Files to Group "Source Group 1"** " option

Interrupt

Interrupt

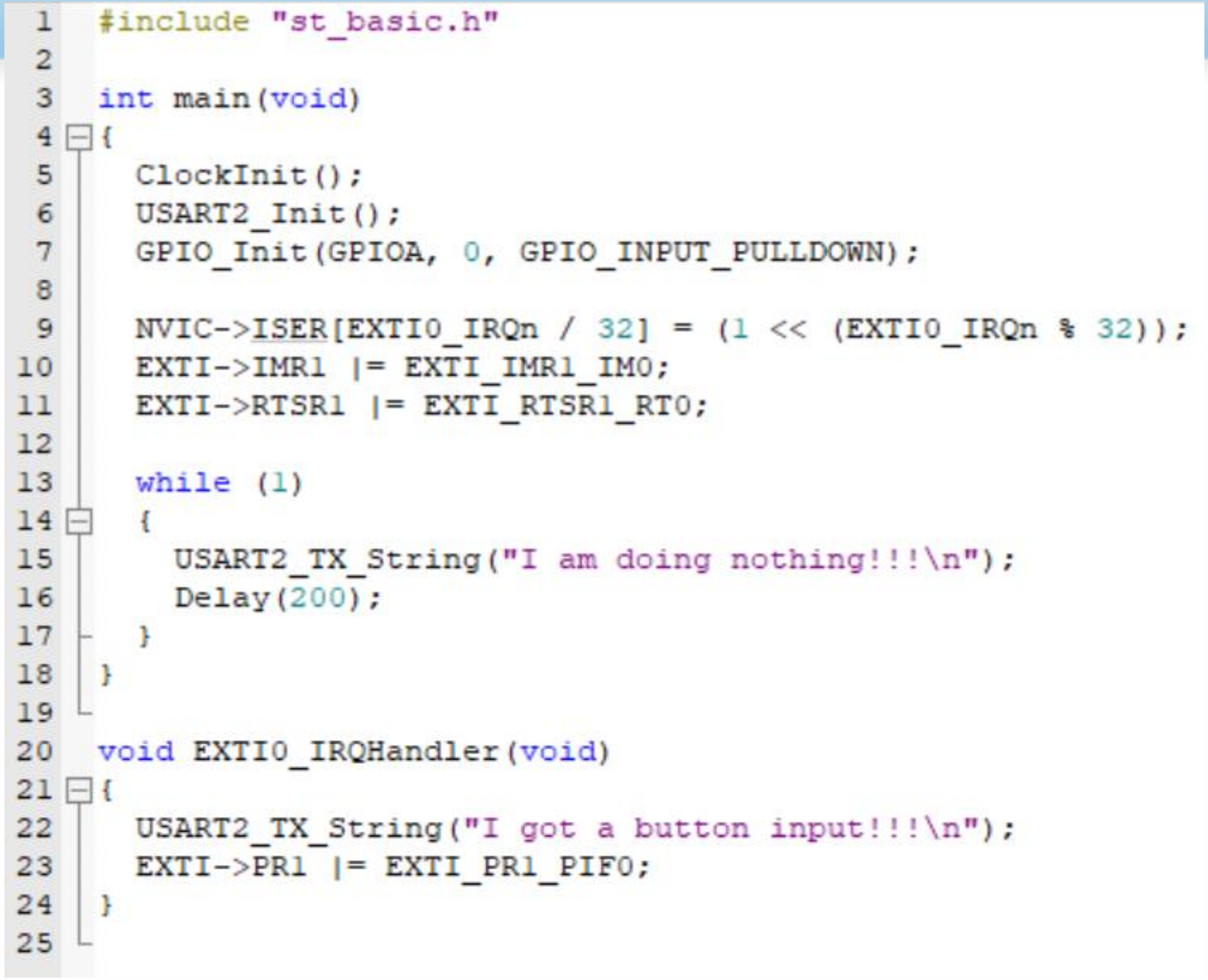
- Interrupt is a way the program can react to the external event.
- When the event occurs, the main program is halted, and the event is handled first.
- Arm Cortex-M4 handles the interrupts as a sort of exceptions.

The detailed description of the exceptions, interrupts and events are on the Ref. lecture note.
In this lab, we will focus on how to use the interrupts.

How to Use It?

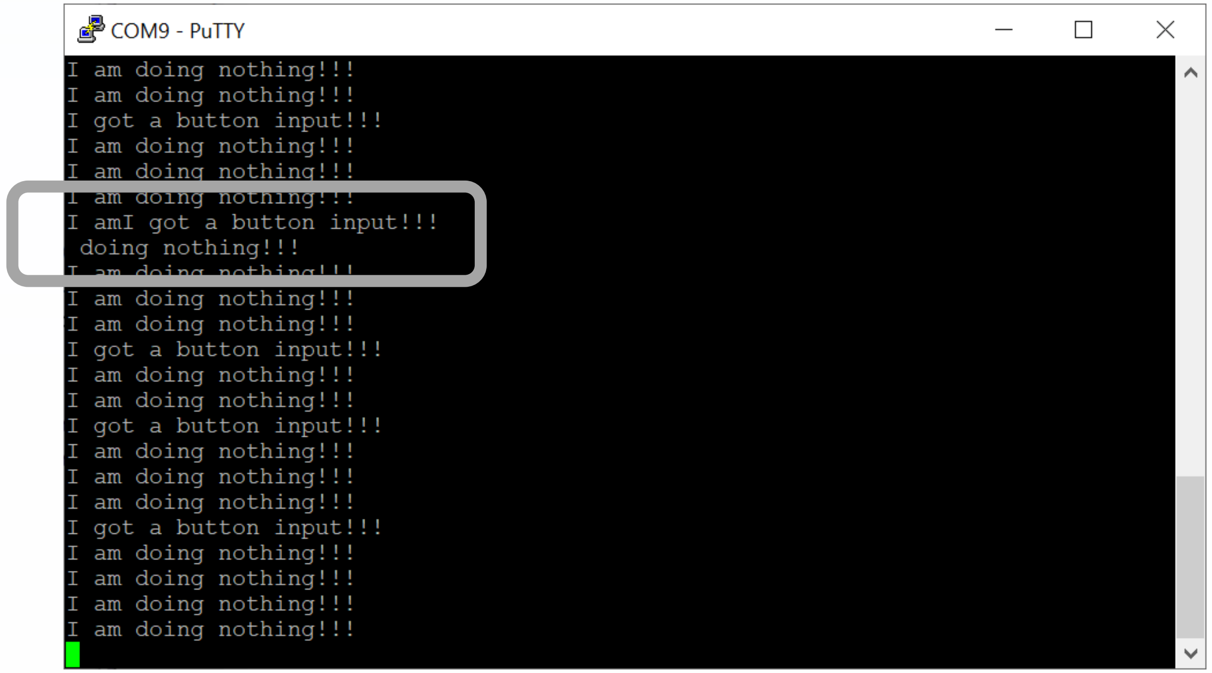
For this topic, there are 3 codes to compile.
Compile it one by one.
Otherwise the IDE will make an error about it.

1.



```c
#include "st_basic.h"

int main(void)
{
  ClockInit();
  USART2_Init();
  GPIO_Init(GPIOA, 0, GPIO_INPUT_PULLDOWN);

  NVIC->ISER[EXTI0_IRQn / 32] = (1 << (EXTI0_IRQn % 32));
  EXTI->IMR1 |= EXTI_IMR1_IM0;
  EXTI->RTSR1 |= EXTI_RTSR1_RT0;

  while (1)
  {
    USART2_TX_String("I am doing nothing!!!\n");
    Delay(200);
  }
}

void EXTI0_IRQHandler(void)
{
  USART2_TX_String("I got a button input!!!\n");
  EXTI->PR1 |= EXTI_PR1_PIF0;
}
```

How to Use It?



```
I am doing nothing!!!
I am doing nothing!!!
I got a button input!!!
I am doing nothing!!!
I am doing nothing!!!
I am doing nothing!!!
I amI got a button input!!!
 doing nothing!!!
I am doing nothing!!!
I am doing nothing!!!
I am doing nothing!!!
I got a button input!!!
I am doing nothing!!!
I am doing nothing!!!
I got a button input!!!
I am doing nothing!!!
I am doing nothing!!!
I am doing nothing!!!
I got a button input!!!
I am doing nothing!!!
I am doing nothing!!!
I am doing nothing!!!
I am doing nothing!!!
```

Press the center button of the on-board joystick.
The message will be printed when you push the button.
You may also see the IRQ message is ‘interrupting’ the normal message.

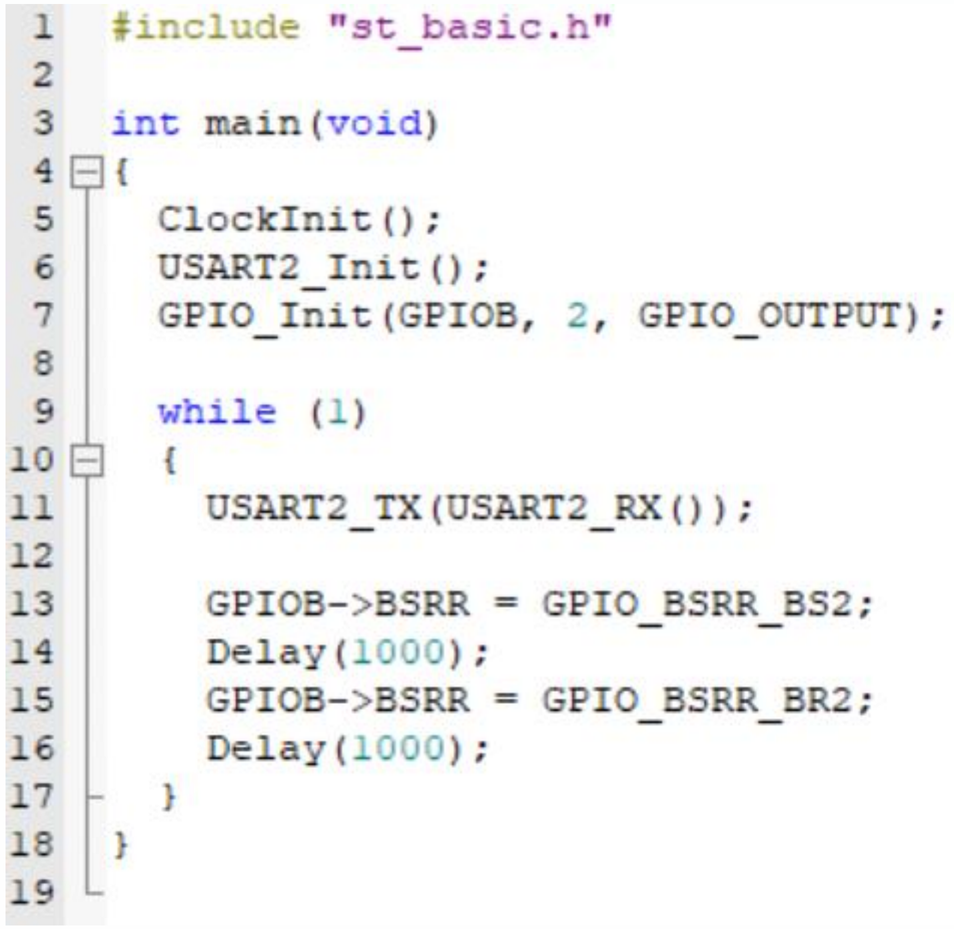
How to Use It?

Next two codes have the requirements as follows:

- Blink the on-board red LED every 1 second.
- Echo the character back to PuTTY the discovery got from PuTTY.

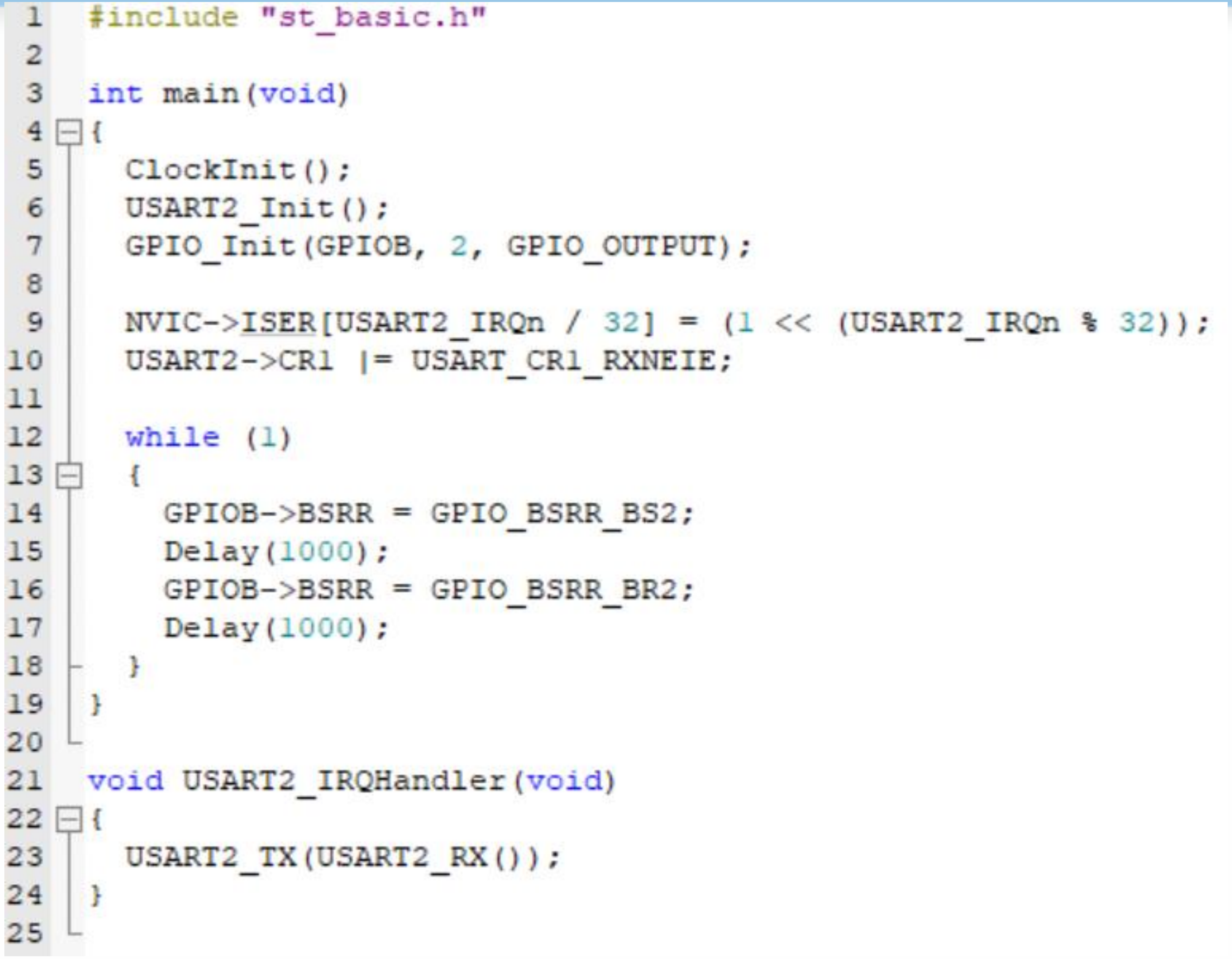
Type any character to PuTTY while you look at the red LED.
See if these two codes are working well or not.

2.



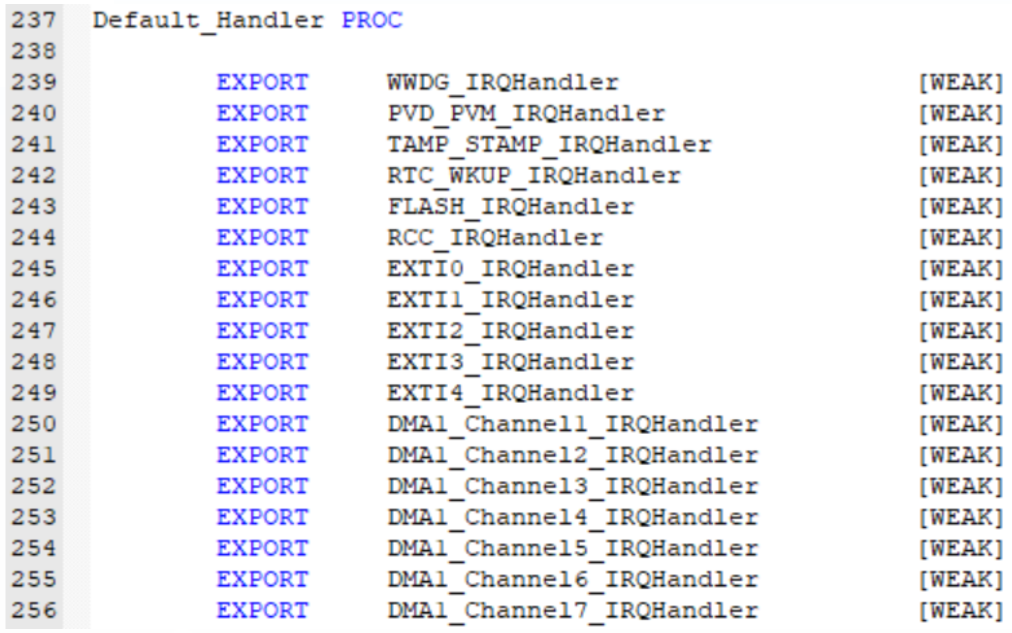
```
#include "st_basic.h"

int main(void)
{
  ClockInit();
  USART2_Init();
  GPIO_Init(GPIOB, 2, GPIO_OUTPUT);

  while (1)
  {
    USART2_TX(USART2_RX());

    GPIOB->BSRR = GPIO_BSRR_BS2;
    Delay(1000);
    GPIOB->BSRR = GPIO_BSRR_BR2;
    Delay(1000);
  }
}

```

3.



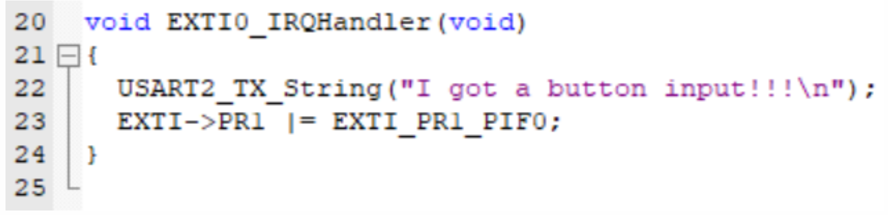
```c
#include "st_basic.h"

int main(void)
{
  ClockInit();
  USART2_Init();
  GPIO_Init(GPIOB, 2, GPIO_OUTPUT);

  NVIC->ISER[USART2_IRQn / 32] = (1 << (USART2_IRQn % 32));
  USART2->CR1 |= USART_CR1_RXNEIE;

  while (1)
  {
    GPIOB->BSRR = GPIO_BSRR_BS2;
    Delay(1000);
    GPIOB->BSRR = GPIO_BSRR_BR2;
    Delay(1000);
  }
}

void USART2_IRQHandler(void)
{
  USART2_TX(USART2_RX());
}

```

IRQ



```
Default_Handler PROC

        EXPORT     WWDG_IRQHandler                    [WEAK]
        EXPORT     PVD_PVM_IRQHandler                 [WEAK]
        EXPORT     TAMP_STAMP_IRQHandler              [WEAK]
        EXPORT     RTC_WKUP_IRQHandler                [WEAK]
        EXPORT     FLASH_IRQHandler                   [WEAK]
        EXPORT     RCC_IRQHandler                     [WEAK]
        EXPORT     EXTI0_IRQHandler                   [WEAK]
        EXPORT     EXTI1_IRQHandler                   [WEAK]
        EXPORT     EXTI2_IRQHandler                   [WEAK]
        EXPORT     EXTI3_IRQHandler                   [WEAK]
        EXPORT     EXTI4_IRQHandler                   [WEAK]
        EXPORT     DMA1_Channel1_IRQHandler           [WEAK]
        EXPORT     DMA1_Channel2_IRQHandler           [WEAK]
        EXPORT     DMA1_Channel3_IRQHandler           [WEAK]
        EXPORT     DMA1_Channel4_IRQHandler           [WEAK]
        EXPORT     DMA1_Channel5_IRQHandler           [WEAK]
        EXPORT     DMA1_Channel6_IRQHandler           [WEAK]
        EXPORT     DMA1_Channel7_IRQHandler           [WEAK]
```

In ‘startup_stm32l476xx.s’



```
void EXTI0_IRQHandler(void)
{
  USART2_TX_String("I got a button input!!!\n");
  EXTI->PR1 |= EXTI_PR1_PIF0;
}
```

In the source code

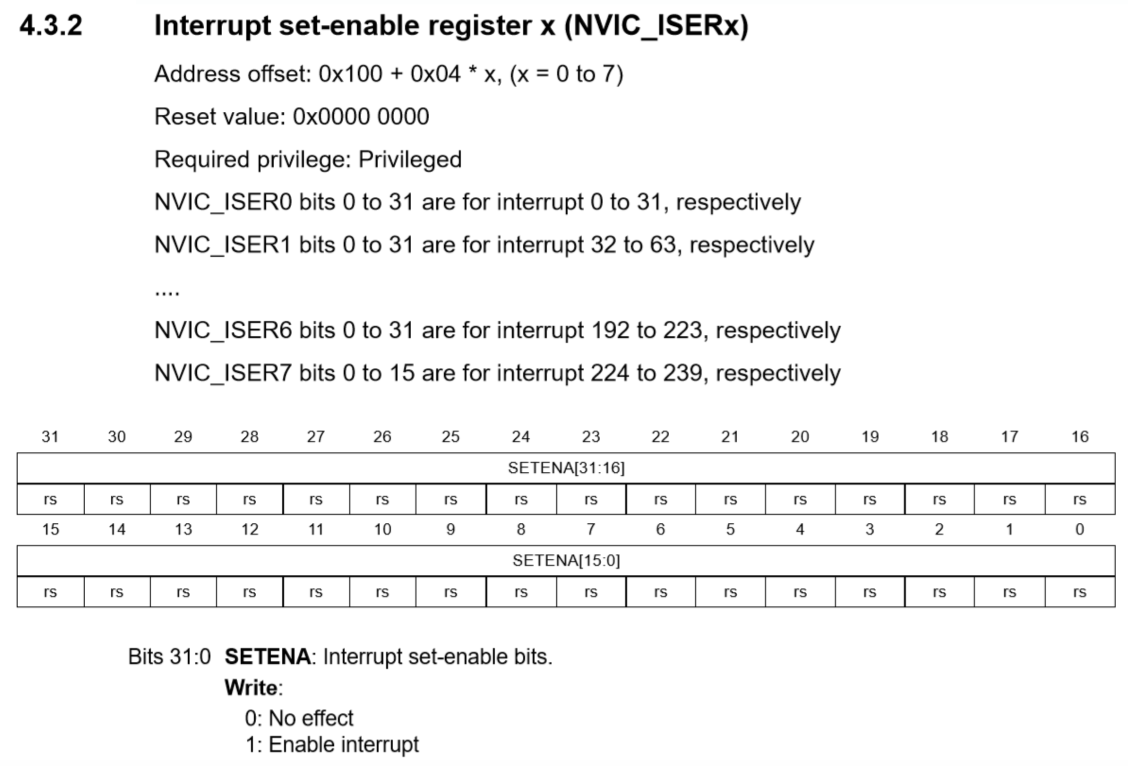
The special function that will be executed when the interrupt is generated, is called Interrupt Service Routine(ISR) in general.

But for STMicroelectronics products, they call it Interrupt Request handler(IRQ).

These are defined in ‘startup_stm32l476xx.s’, with `[weak]` attributes.
You can re-define it in your code, even in C style, thanks to `[weak]` attribute.

If you enable the interrupt but not write your own IRQ, the default handler is executed, which will just loop infinitely and stop your program.

NVIC



4.3.2 Interrupt set-enable register x (NVIC_ISERx)

Address offset: 0x100 + 0x04 * x, (x = 0 to 7)

Reset value: 0x0000 0000

Required privilege: Privileged

NVIC_ISER0 bits 0 to 31 are for interrupt 0 to 31, respectively

NVIC_ISER1 bits 0 to 31 are for interrupt 32 to 63, respectively

....

NVIC_ISER6 bits 0 to 31 are for interrupt 192 to 223, respectively

NVIC_ISER7 bits 0 to 15 are for interrupt 224 to 239, respectively

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SETENA[31:16]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SETENA[15:0]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs

Bits 31:0 **SETENA**: Interrupt set-enable bits.
Write:
0: No effect
1: Enable interrupt

This register deals the value 1 as ‘bit set’, and ‘0’ as nothing. No need to ‘|=’.

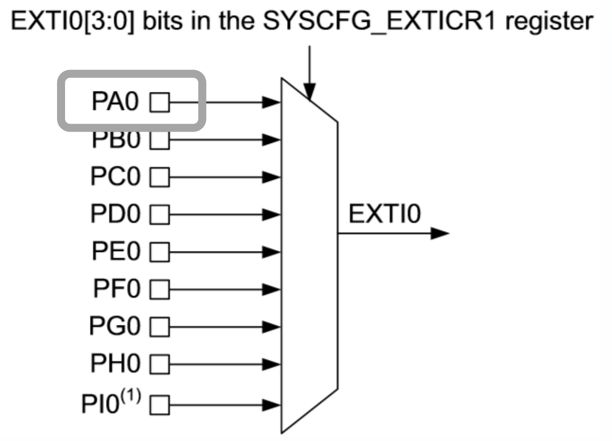
```
NVIC->ISER[EXTI0_IRQn / 32]
    = (1 << (EXTI0_IRQn % 32));
```

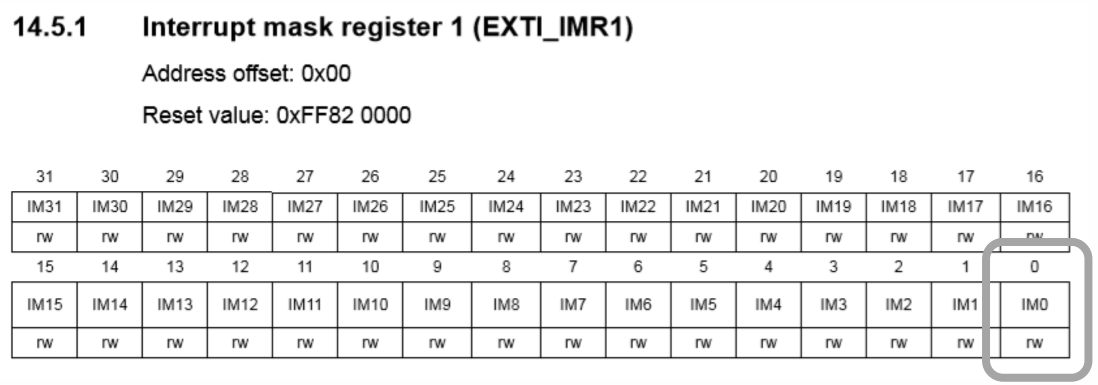
To use any kind of exceptions(including interrupts), the exception we want to use must be enabled by the Nested Vectored Interrupt Controller(NVIC).

There are too much IRQ numbers, so several registers are used for interrupts.

To decide which number and which bit to access and enable, is as the formula on the left side.

EXTI





14.5.1 Interrupt mask register 1 (EXTI_IMR1)

Address offset: 0x00

Reset value: 0xFF82 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IM31	IM30	IM29	IM28	IM27	IM26	IM25	IM24	IM23	IM22	IM21	IM20	IM19	IM18	IM17	IM16
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IM15	IM14	IM13	IM12	IM11	IM10	IM9	IM8	IM7	IM6	IM5	IM4	IM3	IM2	IM1	IM0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

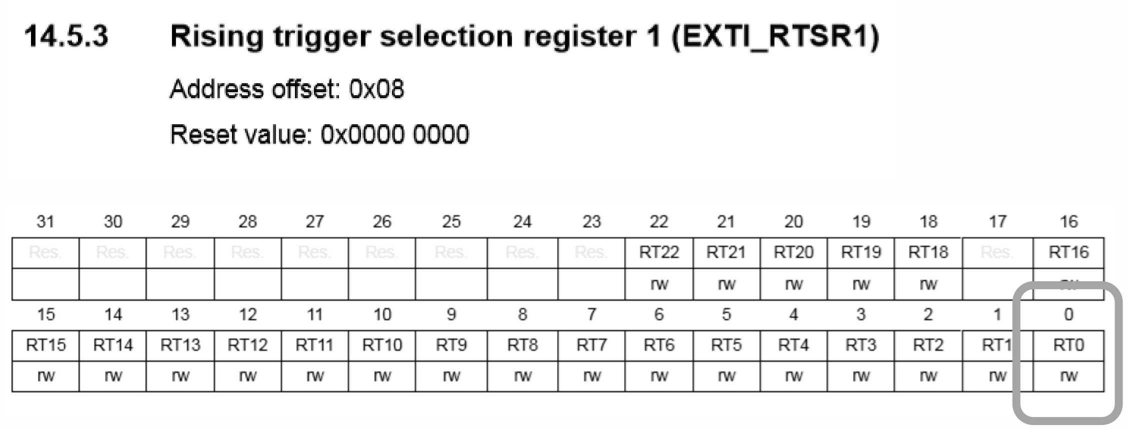
```
EXTI->IMR1 |= EXTI_IMR1_IM0;
```

Some of the interrupts are bound with only one interrupt line.
This is managed by EXTended Interrupts and events controller(EXTI).

For example, GPIO Pin x of every port are bounded together to EXTI Line 0.

We need to enable the Line 0 to use PA0 as an interrupt source.

EXTI



14.5.3 Rising trigger selection register 1 (EXTI_RTSR1)

Address offset: 0x08

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	RT22	RT21	RT20	RT19	RT18	Res.	RT16
									rw	rw	rw	rw	rw		rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RT15	RT14	RT13	RT12	RT11	RT10	RT9	RT8	RT7	RT6	RT5	RT4	RT3	RT2	RT1	RT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

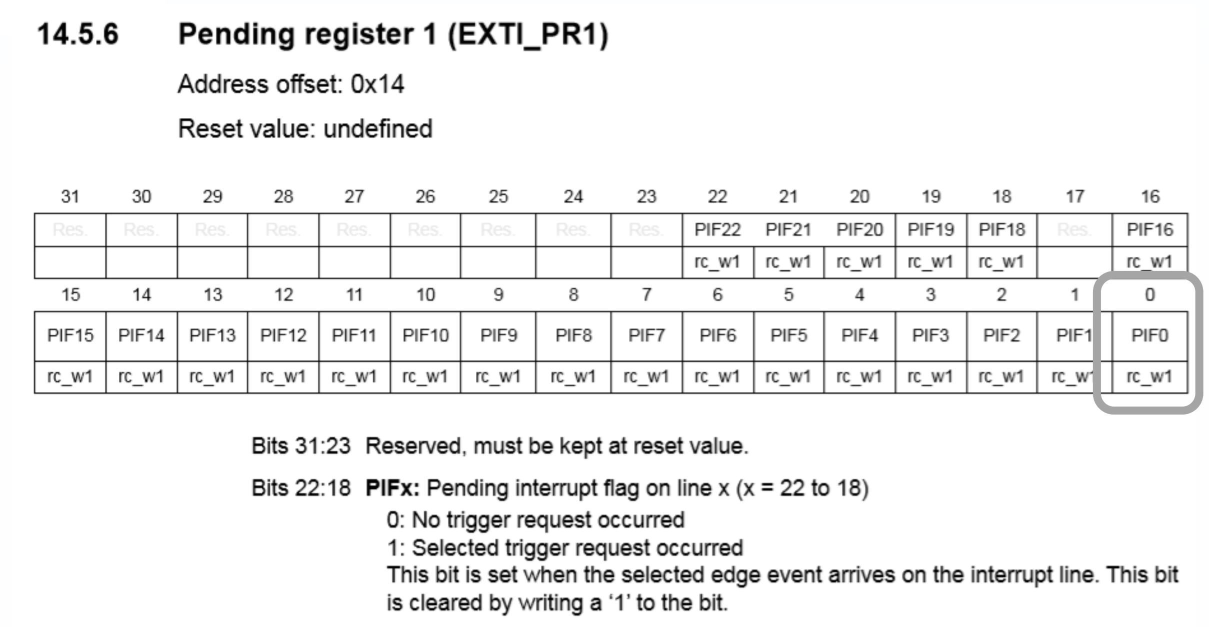
```
EXTI->RTSR1 |= EXTI_RTSR1_RT0;
```

PA0 is pulled down in the discovery board, so the signal generated when you push the button is rising edge.

You can make the line sensitive to the rising edge with EXTI.

This way, you can detect the button push only just at the time you press it.

EXTI



14.5.6 Pending register 1 (EXTI_PR1)

Address offset: 0x14

Reset value: undefined

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	PIF22	PIF21	PIF20	PIF19	PIF18	Res.	PIF16
									rc_w1	rc_w1	rc_w1	rc_w1	rc_w1		rc_w1
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
PIF15	PIF14	PIF13	PIF12	PIF11	PIF10	PIF9	PIF8	PIF7	PIF6	PIF5	PIF4	PIF3	PIF2	PIF1	PIF0
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

Bits 31:23 Reserved, must be kept at reset value.

Bits 22:18 **PIFx:** Pending interrupt flag on line x (x = 22 to 18)
0: No trigger request occurred
1: Selected trigger request occurred
This bit is set when the selected edge event arrives on the interrupt line. This bit is cleared by writing a '1' to the bit.

After each interrupt, the IRQ must clear the interrupt pending bit.
Otherwise the IRQ will be executed again and again.

Most of these flags are cleared by writing 1, NOT 0.

```
EXTI->PR1 |= EXTI_PR1_PIF0;
```

USART Interrupt

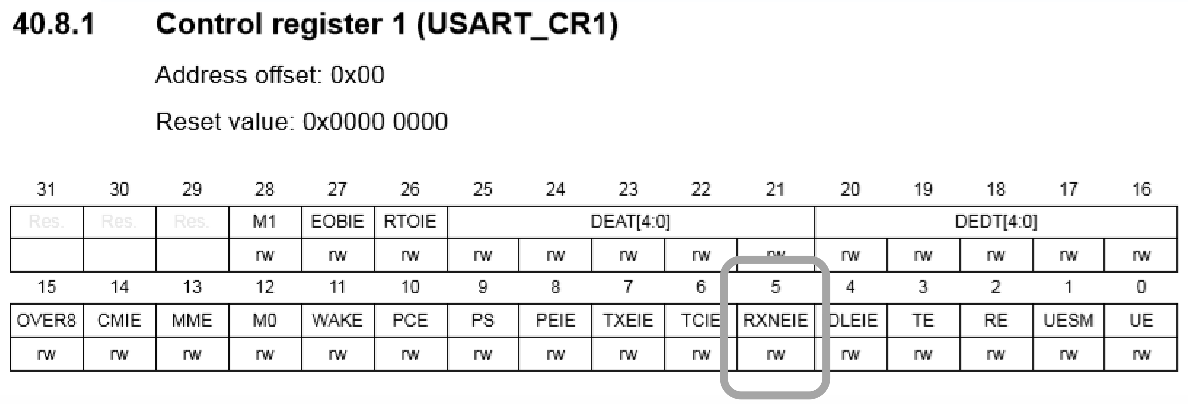
Not only GPIOs, but also almost all peripherals can generate their interrupts.
USART is one of the peripherals that have numerous interrupt and event sources.

Before using it, you should understand why using the interrupt is useful.

You might have noticed that the code #2 doesn't work well.
That is because `USART2_RX()` waits long until it gets any character from the outside.
The reason `USART2_RX()` must be used every `while` loop in main loop is, that your program cannot predict when the input will come in.
The other codes cannot be executed because the waiting function `USART2_RX()` is a part of your main code.

But as the code #3, you can just run your main code and let USART notify the CPU the reception of a message.
That way, the peripherals with the slow and unpredictable behavior, don't block the main program, continue their work and are serviced when they need.

USART Interrupt



40.8.1 Control register 1 (USART_CR1)

Address offset: 0x00

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	M1	EOBIE	RTOIE	DEAT[4:0]					DEDT[4:0]				
			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVER8	CMIE	MME	M0	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	UESM	UE
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

To use interrupts that are not bounded to EXTI, you should activate the interrupt, in the peripheral registers.
(But you ALSO have to enable the interrupt in NVIC)

We will use the receive complete interrupt for this topic, so set `RXNEIE` bit in `CR1`.

```
NVIC->ISER[USART2_IRQn / 32]
    = (1 << (USART2_IRQn % 32));
USART2->CR1 |= USART_CR1_RXNEIE;
```

USART Interrupt

The interrupt bit for USART receive complete event is RXNE bit in ISR register.
So like the case of EXTI, you should clear this bit after IRQ execution.

But you may remember that RXNE is auto-cleared after reading RDR.
So IRQ doesn’t need to clear it.

Like this case, you should check each way of clearing the interrupt bit of the peripherals.

Most of them are cleared by writing 1 to that bit.
Some of them are cleared by writing 1 to the bit of the other register(interrupt flag clear register).
Rarely they are cleared by writing 0.
Others are cleared automatically.

General Steps to Use Interrupts

1. Enable the exception you want to use in NVIC.
2. Check if your exception is bound to EXTI.
    1) If so, enable EXTI line.
    2) Configure the sensitive edge.
3. If not, enable the interrupt of the peripheral register.
4. Write down the exception handler(IRQ for interrupts).
5. Check the way the interrupt bit is cleared and add the code to clear the bit in the handler.

Message to You

For this entire course, we do not have enough time to explain both the peripherals themselves and the interrupts those peripherals invoke.

In addition, the interrupt methods are typically harder to use, since the code is not in the main program explicitly.

So we will focus on how to use the peripherals from now on.

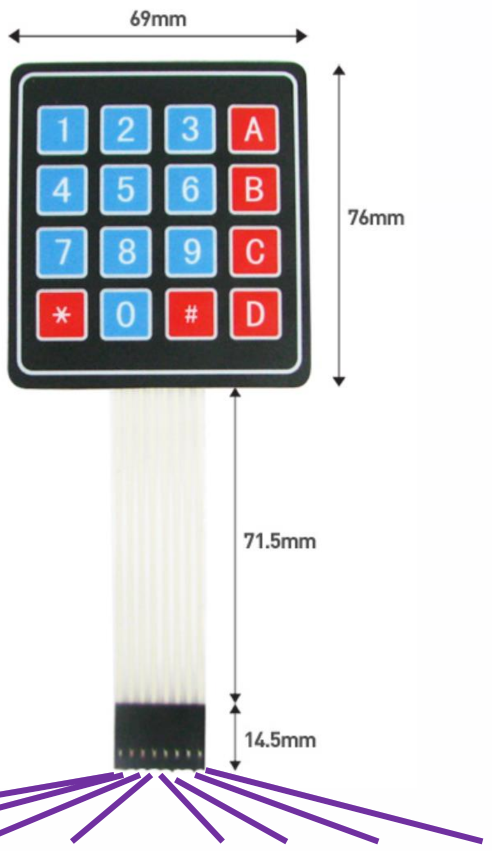
But you should remember the fact that, using interrupts is very powerful way to communicate with the world outside, especially for the embedded system.
Even though designing the algorithm with interrupts is hard, to enable it is very simple, as they all have the same way to activate, as shown in the previous page.

If you are interested in using embedded environment, please take a closer look at the interrupt part of each peripheral you learn.
It will give your project more sensitive reaction of the interaction.

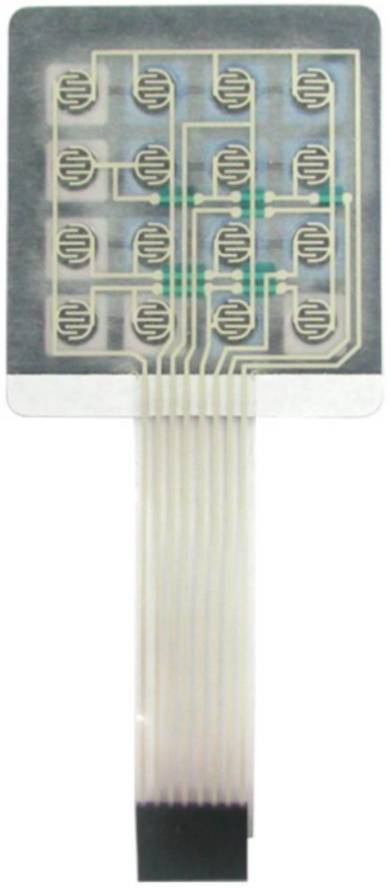
Keypad

Keypad

- 4 * 4 Keypad can be used to get 16 inputs(4 inputs, 4 outputs).
- Basically it is a switch matrix.



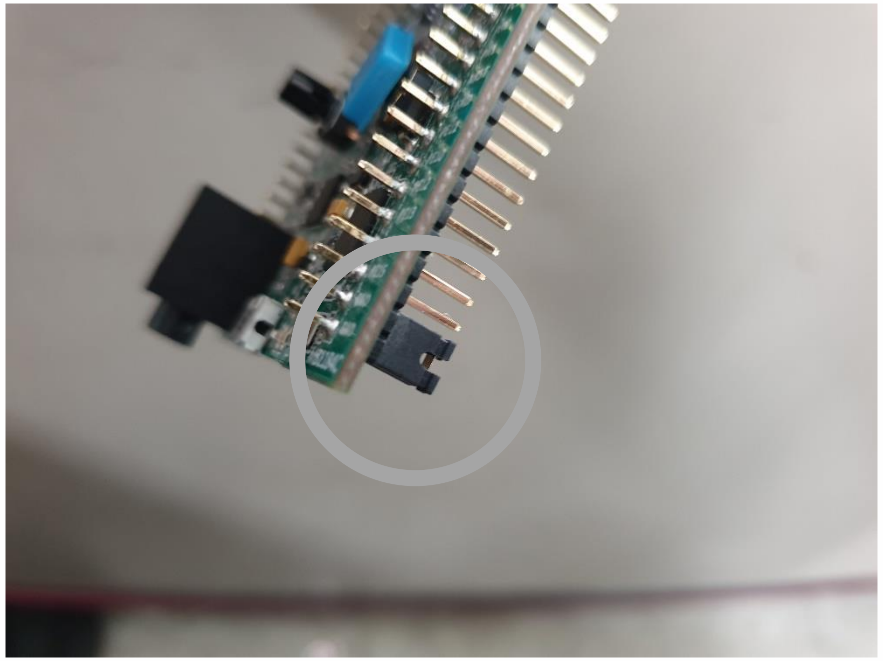
PE12 PE13 PE14 PE15 / PA0 PA1 PA2 PA3

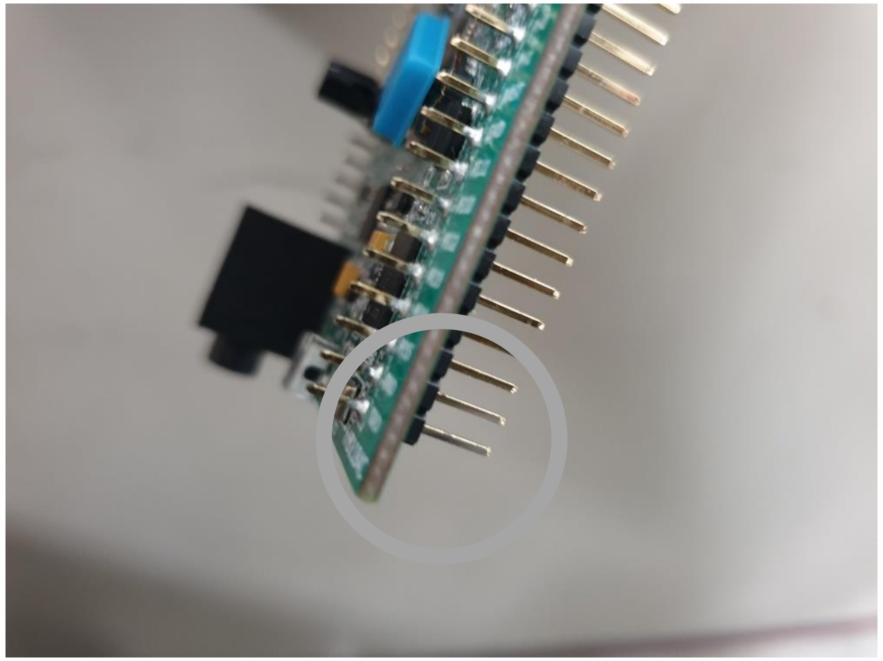


How to Use It?

You can link those pins directly using the combinations of M-M / F-F wires.
But try this way below. It will keep your workspace clean and neat.
If you are using two wires anyway, it is recommended to use the pins below the board, since they are much longer than the pins above.

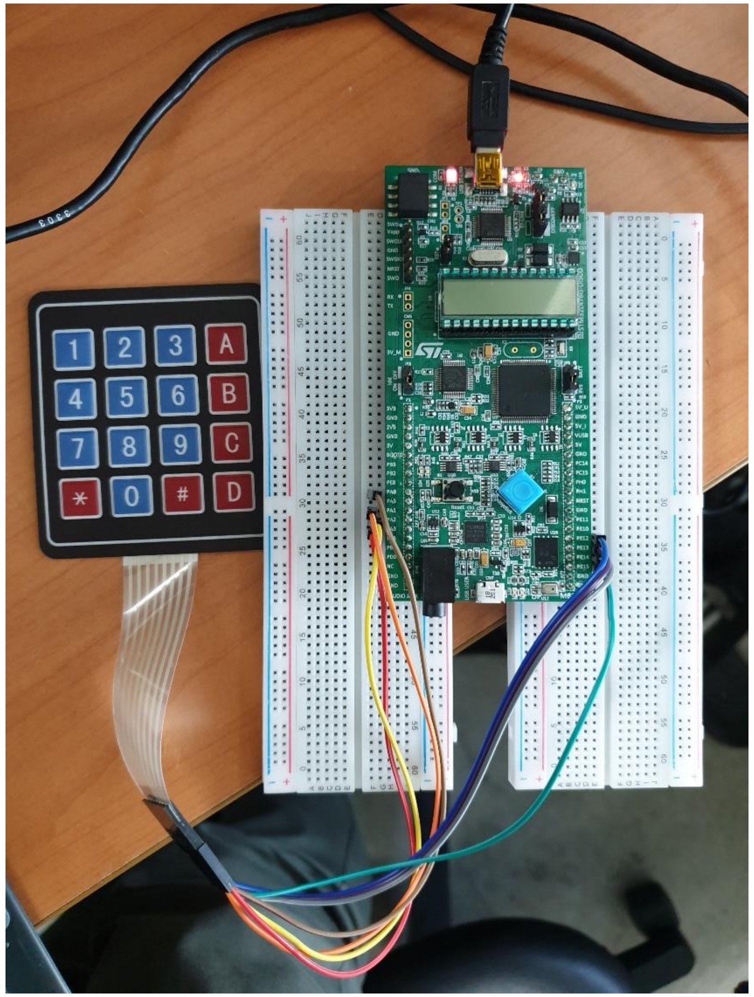
First, move off the jumpers below the board, that links two GND pins.





How to Use It?

Then prepare two breadboards and put the board on it.



Now you can just use M/M jumper cables.

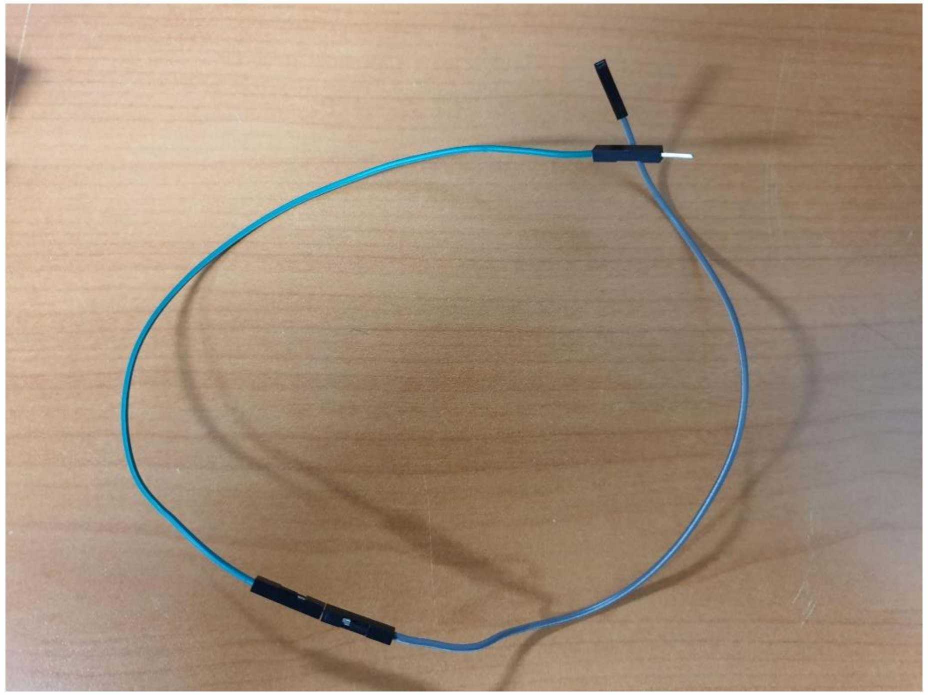
When you look at the keypad from above, the pin map is as the following sequence:

PE12 PE13 PE14 PE15 PA0 PA1 PA2 PA3

How to Use It?

If the board is too loose on the breadboard, then you should use the board with only the jumper cables.
You will need M/F jumper cables, but there is no cable prepared.
But you can make M/F cables, as the following picture.

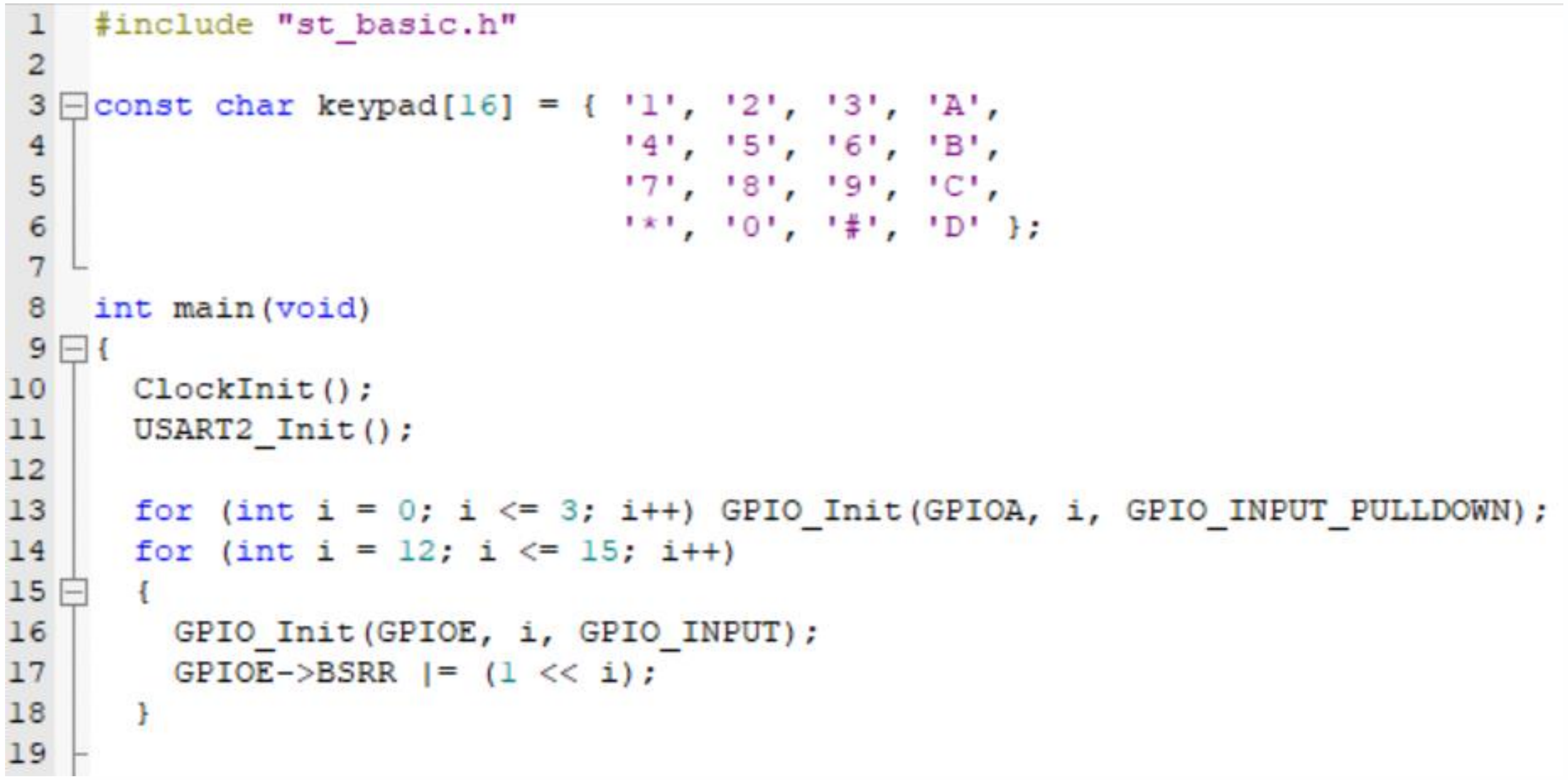
This is done by connecting one M/M cable with one F/F cable.



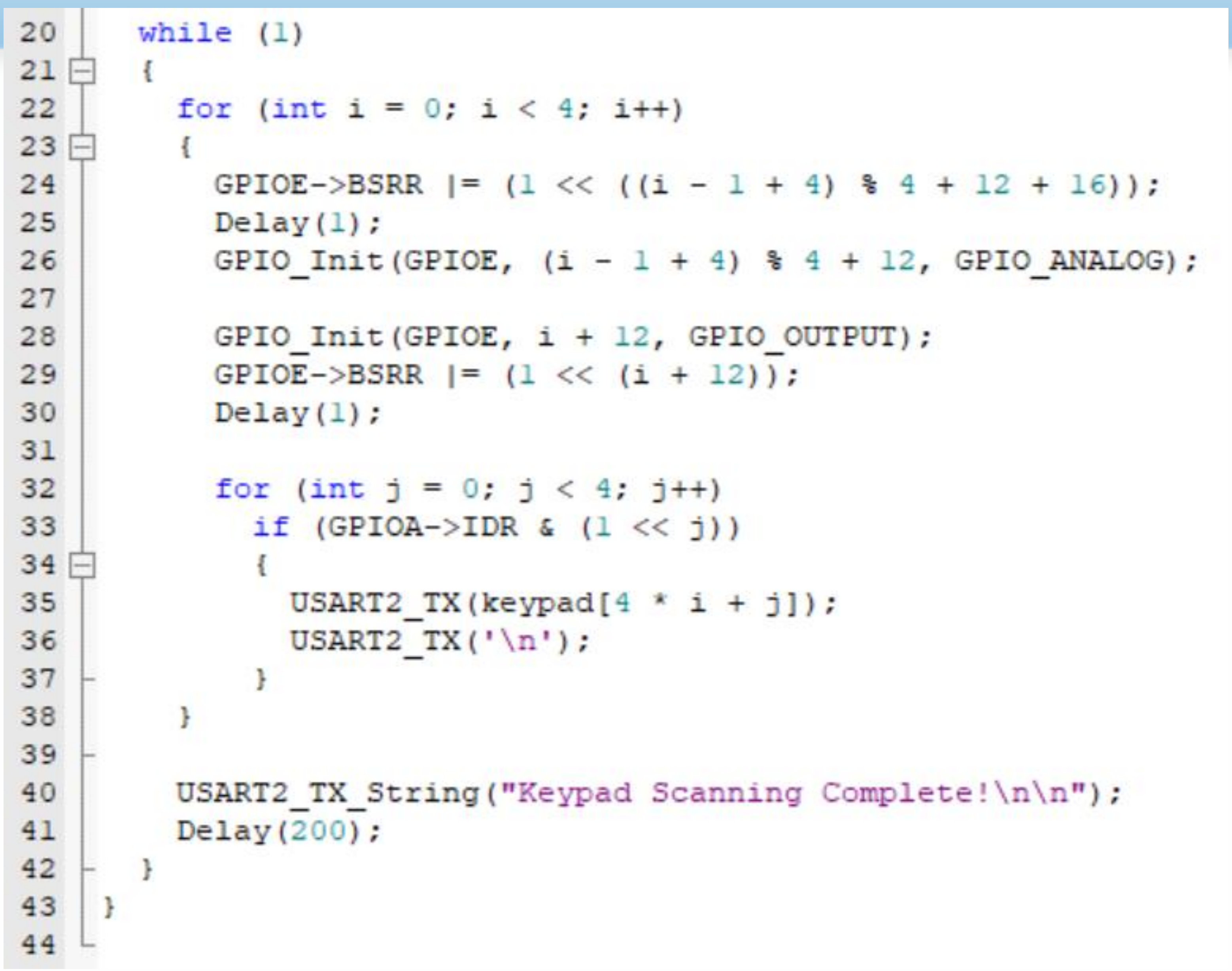
How to Use It?

There are two codes for this topic.
Compile it one by one.
Otherwise the IDE will make an error about it.

1.

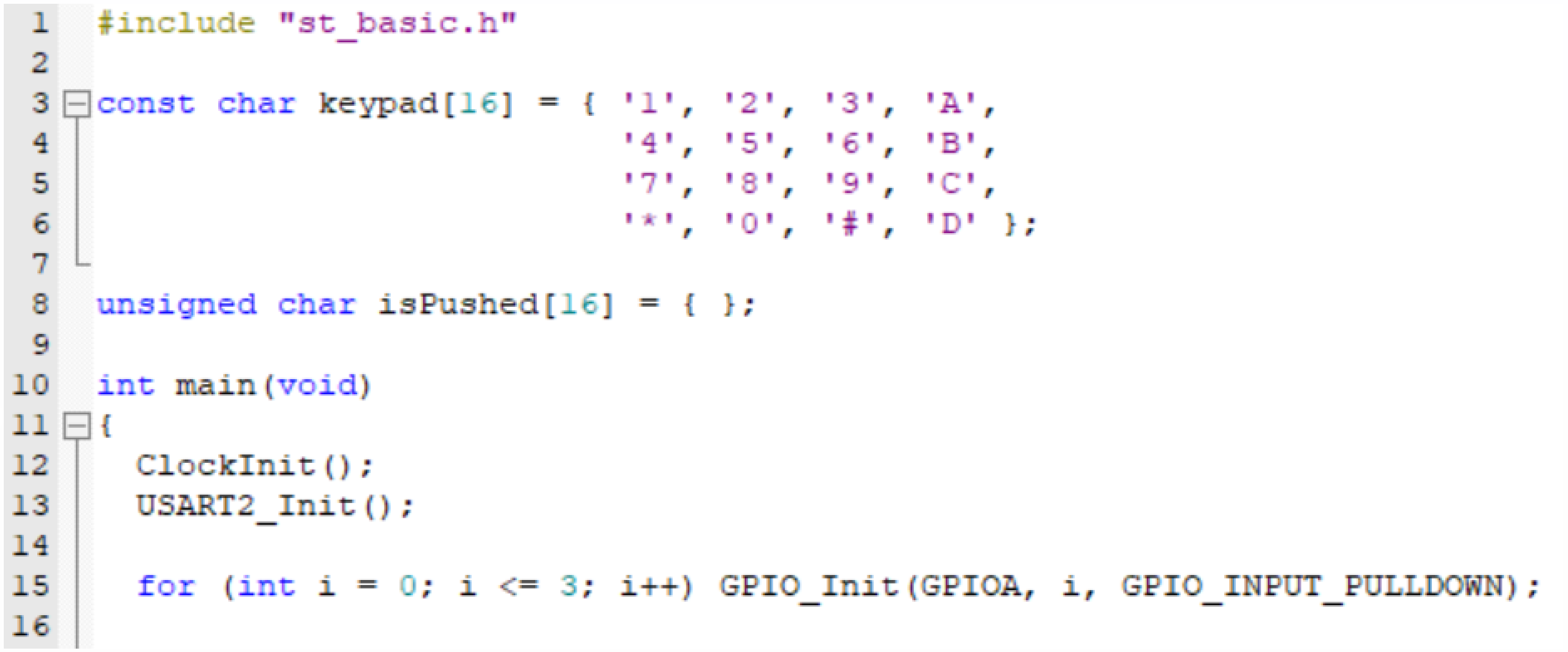


```c
#include "st_basic.h"

const char keypad[16] = { '1', '2', '3', 'A',
                          '4', '5', '6', 'B',
                          '7', '8', '9', 'C',
                          '*', '0', '#', 'D' };

int main(void)
{
  ClockInit();
  USART2_Init();

  for (int i = 0; i <= 3; i++) GPIO_Init(GPIOA, i, GPIO_INPUT_PULLDOWN);
  for (int i = 12; i <= 15; i++)
  {
    GPIO_Init(GPIOE, i, GPIO_INPUT);
    GPIOE->BSRR |= (1 << i);
  }

```

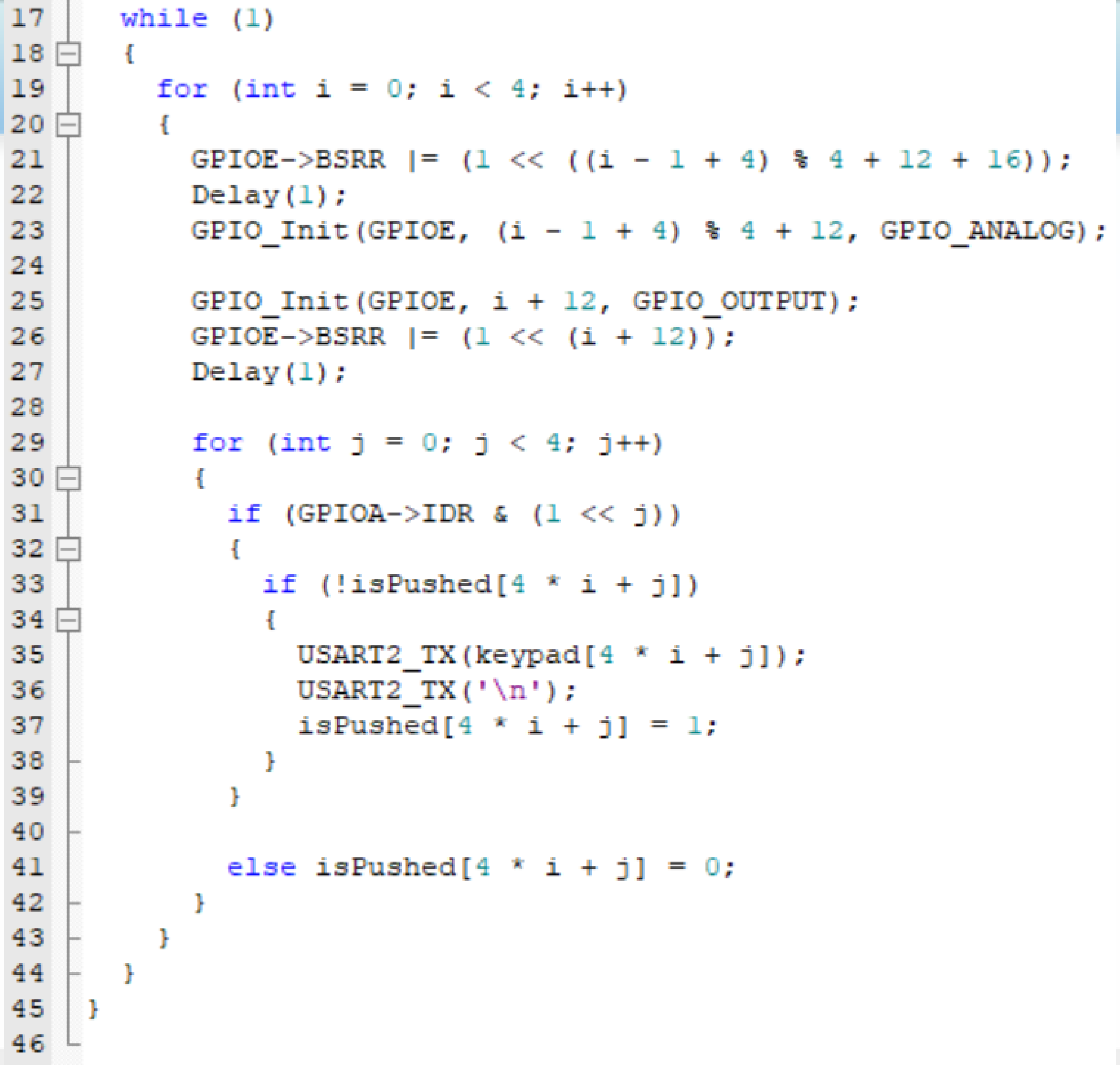


```
  while (1)
  {
    for (int i = 0; i < 4; i++)
    {
      GPIOE->BSRR |= (1 << ((i - 1 + 4) % 4 + 12 + 16));
      Delay(1);
      GPIO_Init(GPIOE, (i - 1 + 4) % 4 + 12, GPIO_ANALOG);

      GPIO_Init(GPIOE, i + 12, GPIO_OUTPUT);
      GPIOE->BSRR |= (1 << (i + 12));
      Delay(1);

      for (int j = 0; j < 4; j++)
        if (GPIOA->IDR & (1 << j))
        {
          USART2_TX(keypad[4 * i + j]);
          USART2_TX('\n');
        }
    }

    USART2_TX_String("Keypad Scanning Complete!\n\n");
    Delay(200);
  }
}
```

2.

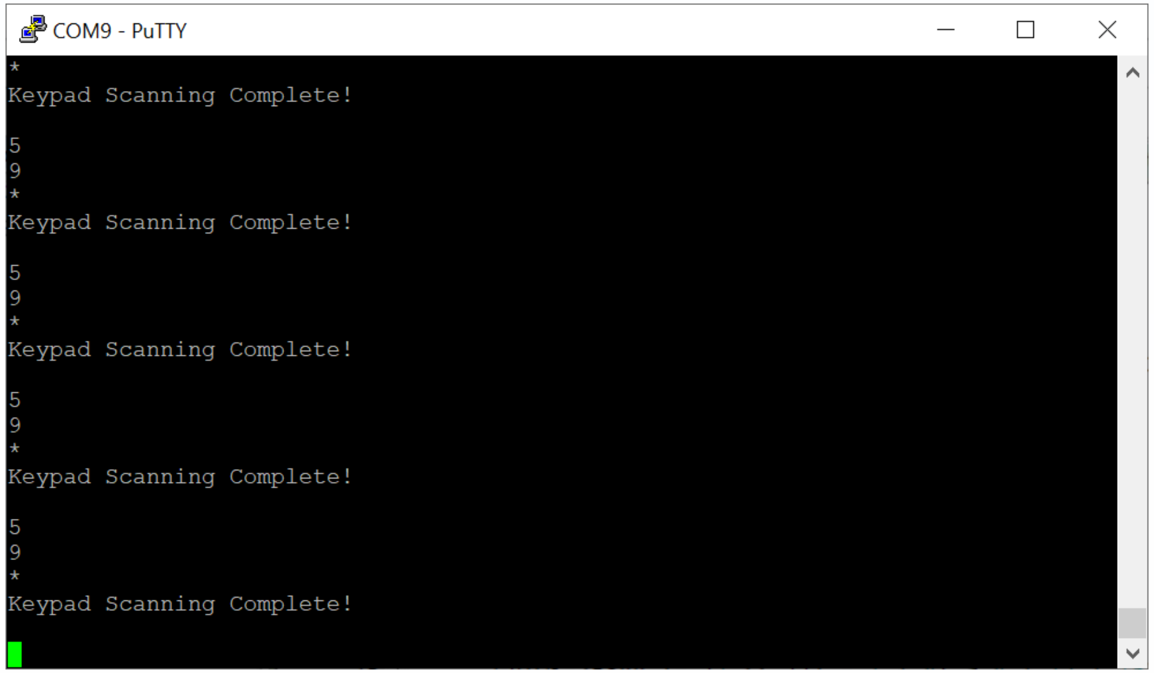


```c
#include "st_basic.h"

const char keypad[16] = { '1', '2', '3', 'A',
                          '4', '5', '6', 'B',
                          '7', '8', '9', 'C',
                          '*', '0', '#', 'D' };

unsigned char isPushed[16] = { };

int main(void)
{
  ClockInit();
  USART2_Init();

  for (int i = 0; i <= 3; i++) GPIO_Init(GPIOA, i, GPIO_INPUT_PULLDOWN);

```



```c
  while (1)
  {
    for (int i = 0; i < 4; i++)
    {
      GPIOE->BSRR |= (1 << ((i - 1 + 4) % 4 + 12 + 16));
      Delay(1);
      GPIO_Init(GPIOE, (i - 1 + 4) % 4 + 12, GPIO_ANALOG);

      GPIO_Init(GPIOE, i + 12, GPIO_OUTPUT);
      GPIOE->BSRR |= (1 << (i + 12));
      Delay(1);

      for (int j = 0; j < 4; j++)
      {
        if (GPIOA->IDR & (1 << j))
        {
          if (!isPushed[4 * i + j])
          {
            USART2_TX(keypad[4 * i + j]);
            USART2_TX('\n');
            isPushed[4 * i + j] = 1;
          }
        }

        else isPushed[4 * i + j] = 0;
      }
    }
  }
}
```

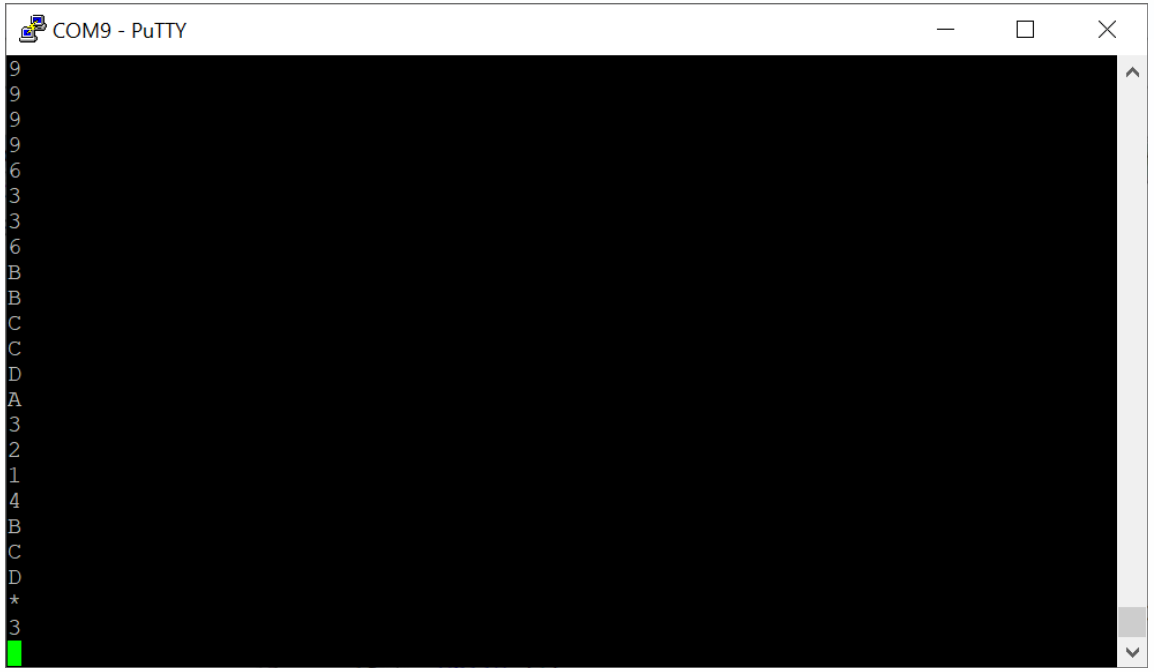
How to Use It?


COM9 - PuTTY

```
*
Keypad Scanning Complete!

5
9
*
Keypad Scanning Complete!

5
9
*
Keypad Scanning Complete!

5
9
*
Keypad Scanning Complete!

5
9
*
Keypad Scanning Complete!
```

The first code will scan the keypad every 200ms.
Try pushing several buttons at a time.

How to Use It?

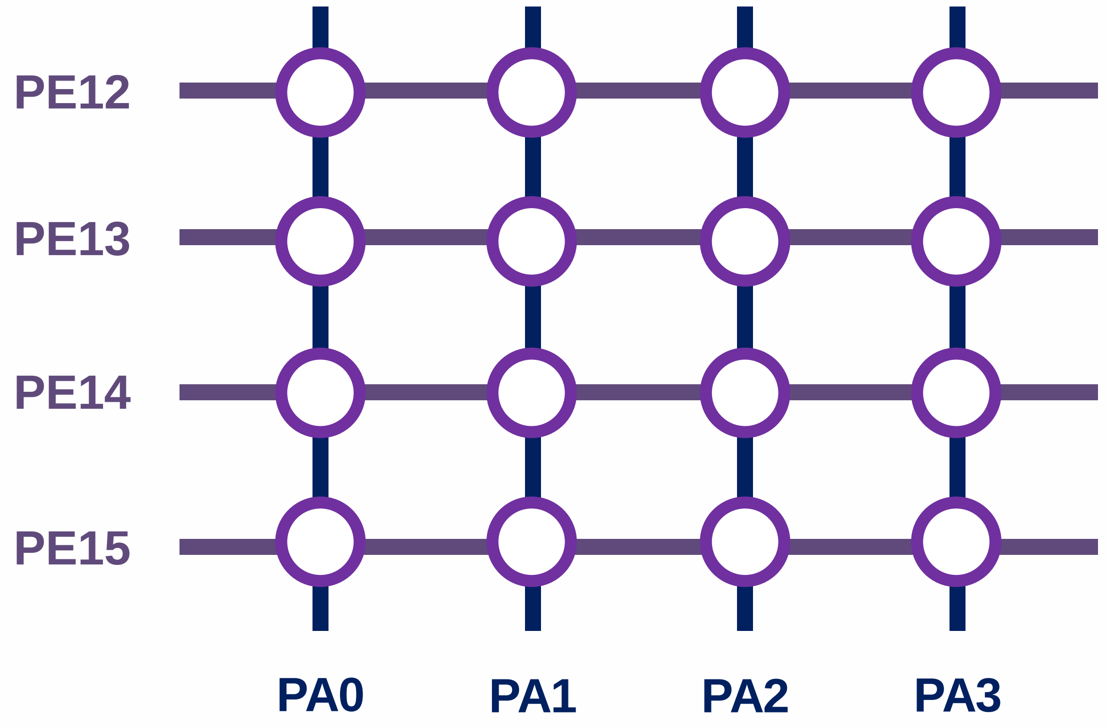


```
9
9
9
9
6
3
3
6
B
B
C
C
D
A
3
2
1
4
B
C
D
*
3
```

The second code will only actively display the pressed button at the time the button is pushed.

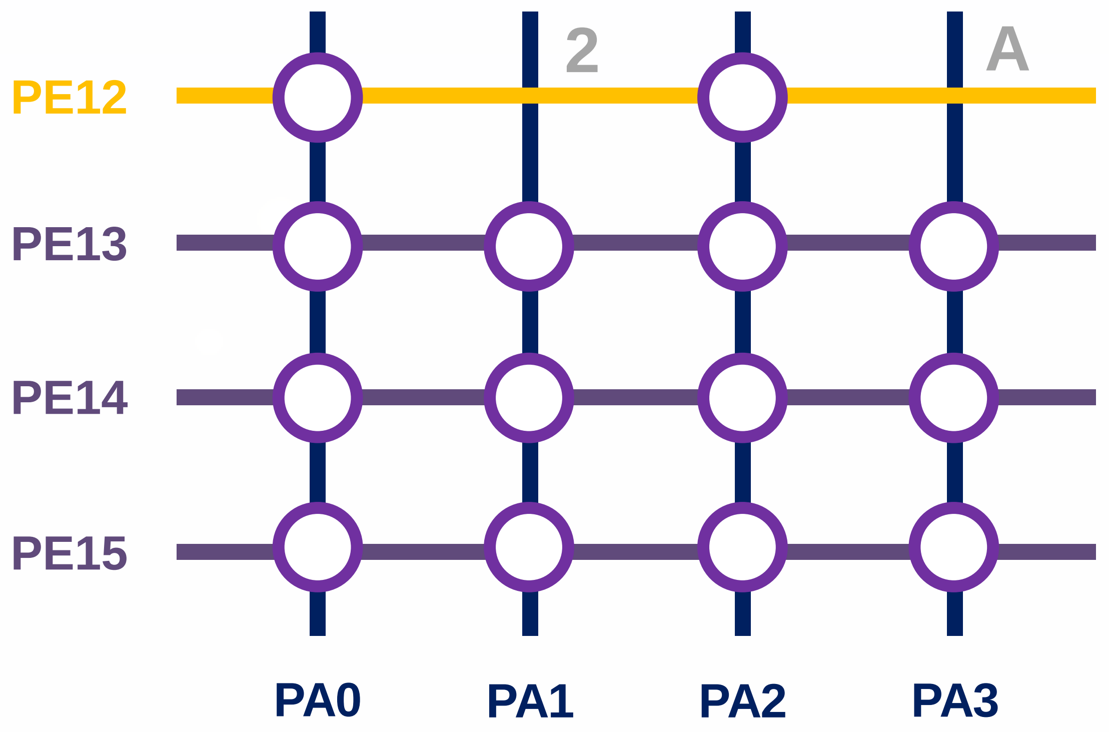
Switch Multiplexing

We will use 16-key(4 x 4) keypads.
If you make 16 switches with some pull-up/down resistors, it may cost 17(including common) wires.
But there are only 8 wires needed, thanks to the multiplexing.



Switch Multiplexing

Port A has the inputs with pull-down resistors and debouncing capacitors(on the discovery board).
Port E has the outputs that will be logical 1, one by one.
The cross sections are normally open(not contacted) and are closed when the user presses the button.



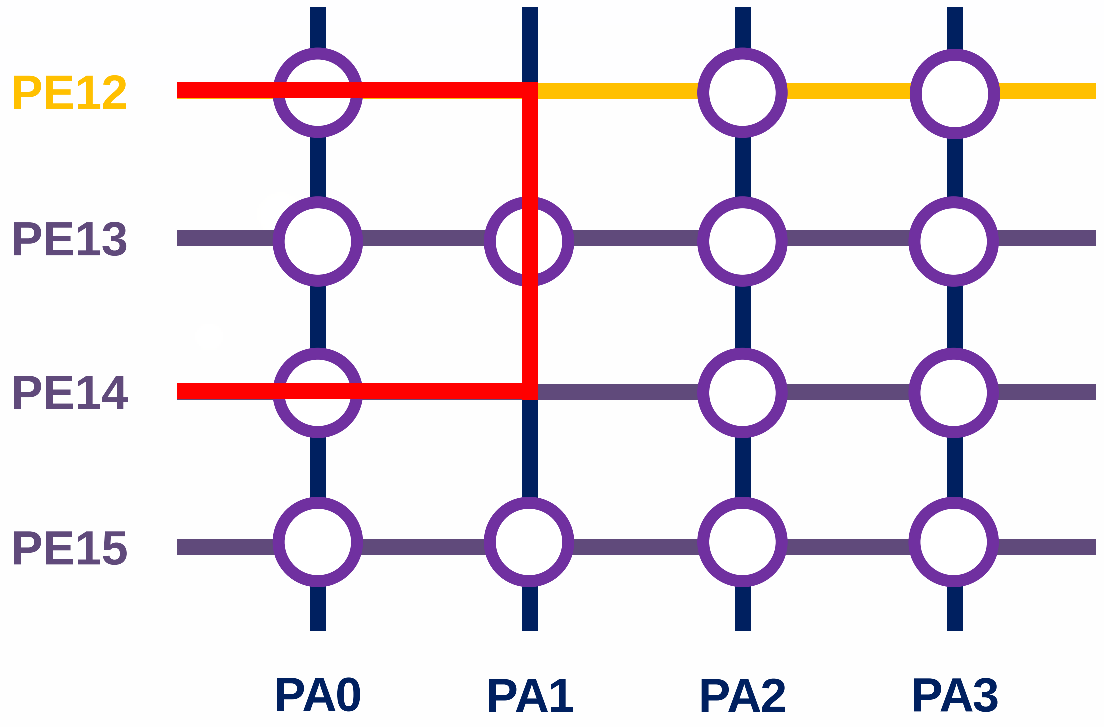
When PE12 is logical 1, the keypad can read the first row of the matrix.

If ‘2’ and ‘A’ are pushed, PA1 and PA3 will go logical 1, while PA0 and PA2 remain 0.

So in vice versa, when there is PEx output, if you read PAy and the value was 1, that means the button in row (x – 11), column (y + 1) is pushed.

Switch Multiplexing

There is a problem when you are multiplexing the switches.
If more than 2 buttons in a column are pushed simultaneously, there is a short circuit between two different output pins.

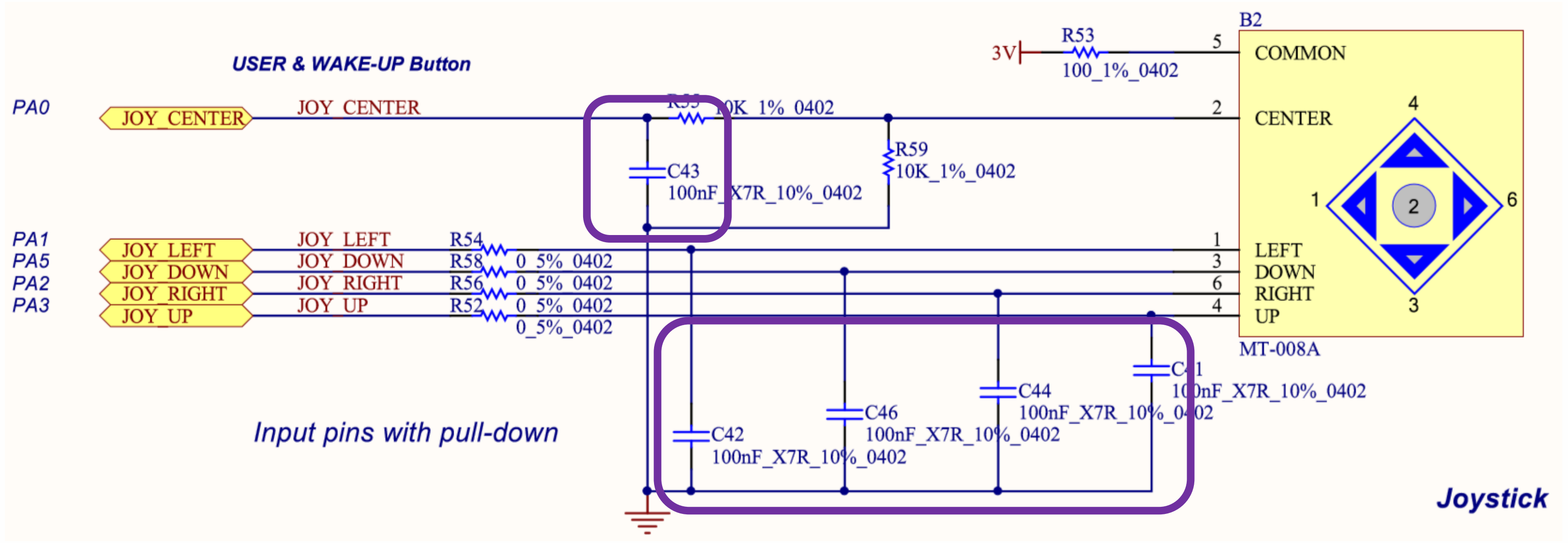


It not only disturbs the proper reading but can also damage the GPIOs permanently.

To prevent that, unused output pins must be input pins.
(That is after the output becomes logical 0, for discharging the debouncing capacitors. See the next page.)

Why there is Delay(1)?

If you carefully see the code, you may find `Delay(1)` before reading any pins.
That is due to the output load capacitance, especially in this case, the debouncing capacitors.
Those capacitors prevent chattering during the buttons are mechanically be stabilized.
But those also slows down the change of the voltage.



Exercises

Ghost Key

When using the keypad with the code #1, you might have noticed that at a certain combination of pressed keys, the key which is not pressed is read as it is pushed.

What is that combination of the keys?
Why the input pins read that key as it is pushed?

Password Check

You’ve learned how to read the keypad.

Make a lock that is open when you enter the correct number through the keypad.

Requirements:

- The password is in your code.
- Normally red LED is on.
- When the device gets the correct 4 sequence of numbers, the green LED is on.
- After 3 seconds, the red LED is on again.

THANK YOU

인하대학교