Tecnológico de Monterrey

SoC - Microcontrollers

Module 2

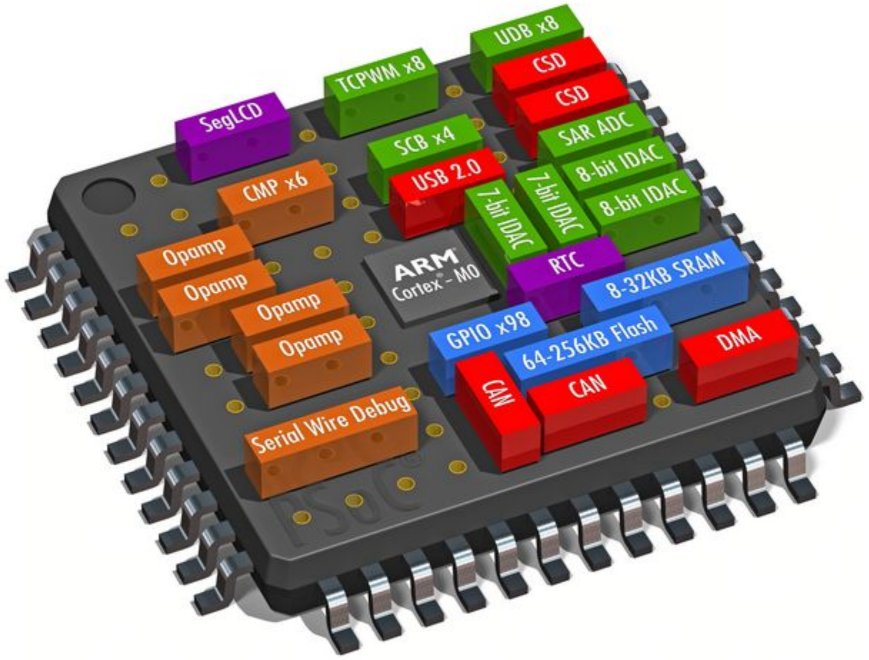
Microcontrollers

Topic 4.1.

I/O Peripherals

Duration
4 hours

Instructor
Dr. Gilberto Ochoa Ruiz



Introduction – Context and Motivation

Topics to be covered in this session

- Introduction KL25z GPIOs
- ARM I/O Programming
- Embedded “Hello World” (LED)
- Reading a Switch
- 8 segment Display
- Little Project with basic concepts

Introduction – Context and Motivation

What will we learn?

How the GPIO are configured and which options are available

How to interface simple signals using single channels

Identify which configurations for inputs are available

How to do signal multiplexing for controlling several 7 segment displays

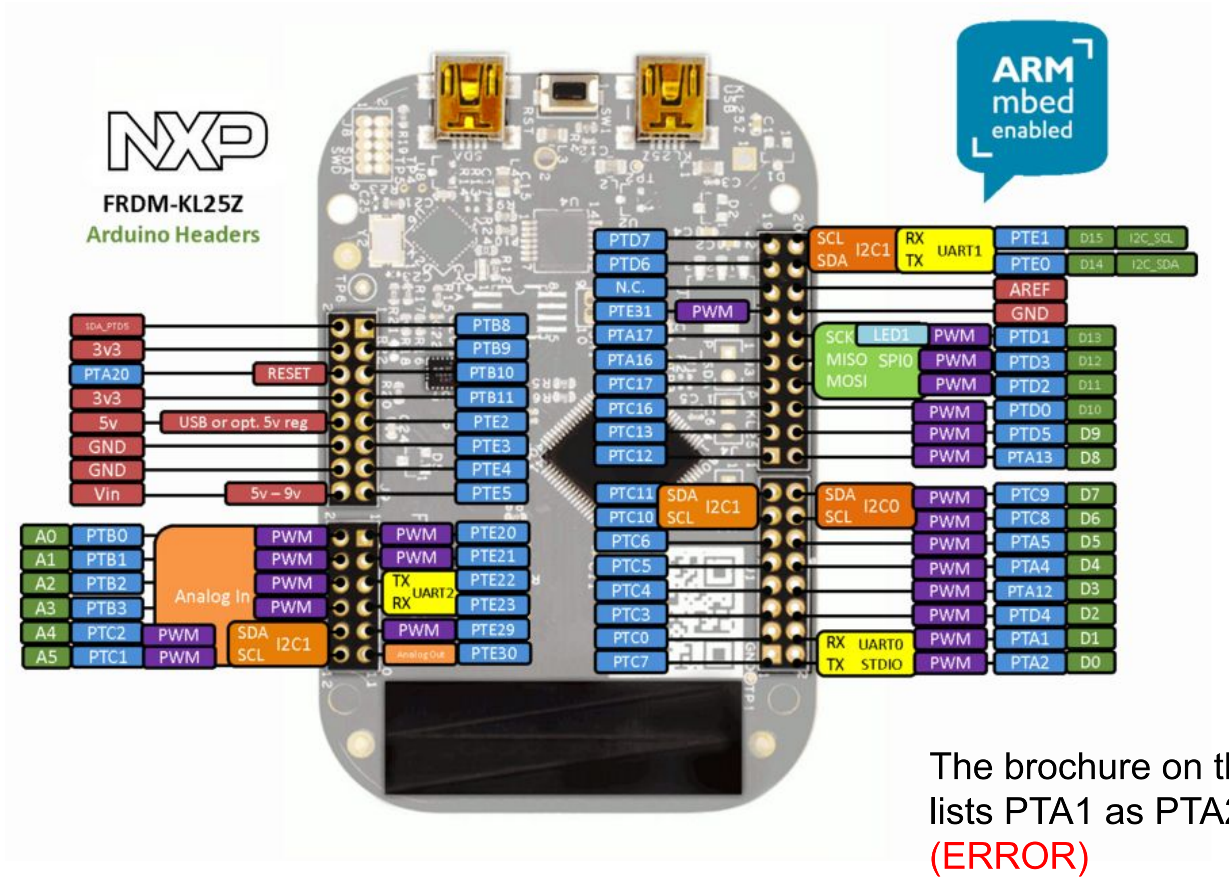
Next topic…

1.1. Fundamentals – Programmable Logic

Fundamentals – GPIO Programming

ARM I/O Programming

In microcontrollers, we use the general purpose input output (GPIO) pins to interface with LED, switch (SW), LCD, keypad, and so on.



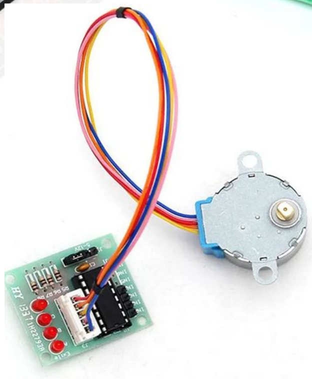
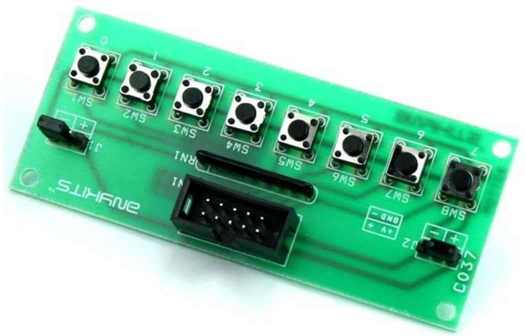
The brochure on the card lists PTA1 as PTA2 and vice versa (ERROR)

Fundamentals – GPIO Programming

ARM I/O Programming

This is a very important knowledge since the vast majority of embedded products have some kind of I/O.

More importantly, this chapter sets the stage for understanding of peripheral I/O addresses and how the are accessed and used in ARM processors.

Fundamentals – GPIO Programming

General Purpose I/O

The general purpose I/O ports in **KL25Z128VLK4** ARM are designated as port A to port E.

The following shows the address range assigned to each GPIO port:

- **GPIO Port A :** 0x400F F000 to 0x400F F017
- **GPIO Port B :** 0x400F F040 to 0x400F F057
- **GPIO Port C :** 0x400F F080 to 0x400F F097
- **GPIO Port D** : 0x400F F0C0 to 0x400F F0D7
- **GPIO Port E** : 0x400F F100 to 0x400F F117

Do you remember what is a memory map?

Why is important that I/Os are mapped to memory?

Fundamentals – GPIO Programming

GPIO Programming and Interfacing

While memory holds code and data for the CPU to process, the I/O ports are used by the CPU to access input and output devices. In the microcontroller, we have two types of I/O:

a. General Purpose I/O (GPIO):

The GPIO ports are used for interfacing devices such as LEDs, switches, LCD, keypad, and so on.

b. Special purpose I/O:

These I/O ports have designated function such as ADC (Analog-to-Digital), Timer, UART (universal asynchronous receiver transmitter), and so on.

Fundamentals – GPIO Programming

I/O Pins in Freescale FRDM board

In Freescale ARM chips, I/O ports are named with alphabets A, B, C, and so on.

Each port can have up to 32 pins and they are designated as PTA0-PTA31, PTB0-PTB31, and so on.

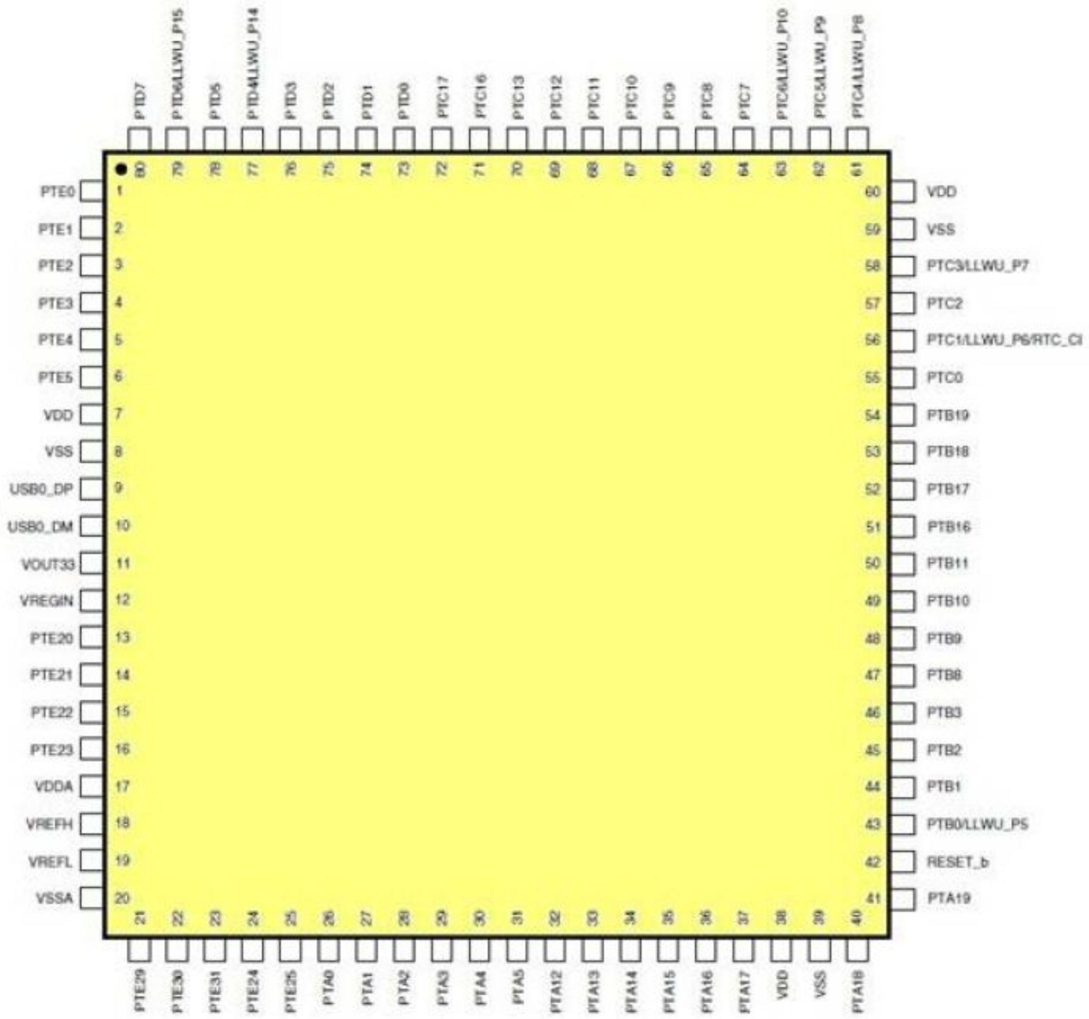
It must be noted that not all 32 pins of each port are implemented.

The ARM chip used in FRDM board is in Kinetis L series with part number KLxxZ128VLK4.

It has ports A, B, C, D, and E

Fundamentals – GPIO Programming

KL25Z pinout



Buses and Interfacing (2)

The **AHB bus** is much faster than APB.

The AHB allows one clock cycle access to the peripherals.

The **APB is slower** and its access time is minimum of 2 clock cycles.

The Base addresses for the GPIOs of AHB is as follow:

- **GPIO Port A (AHB):** 0xF80F F000
- **GPIO Port B (AHB):** 0xF80F F040
- **GPIO Port C (AHB):** 0xF80F F080
- **GPIO Port D (AHB):** 0xF80F F0C0
- **GPIO Port E (AHB):** 0xF80F F100

Fundamentals – GPIO Programming

Buses and Interfacing (1)

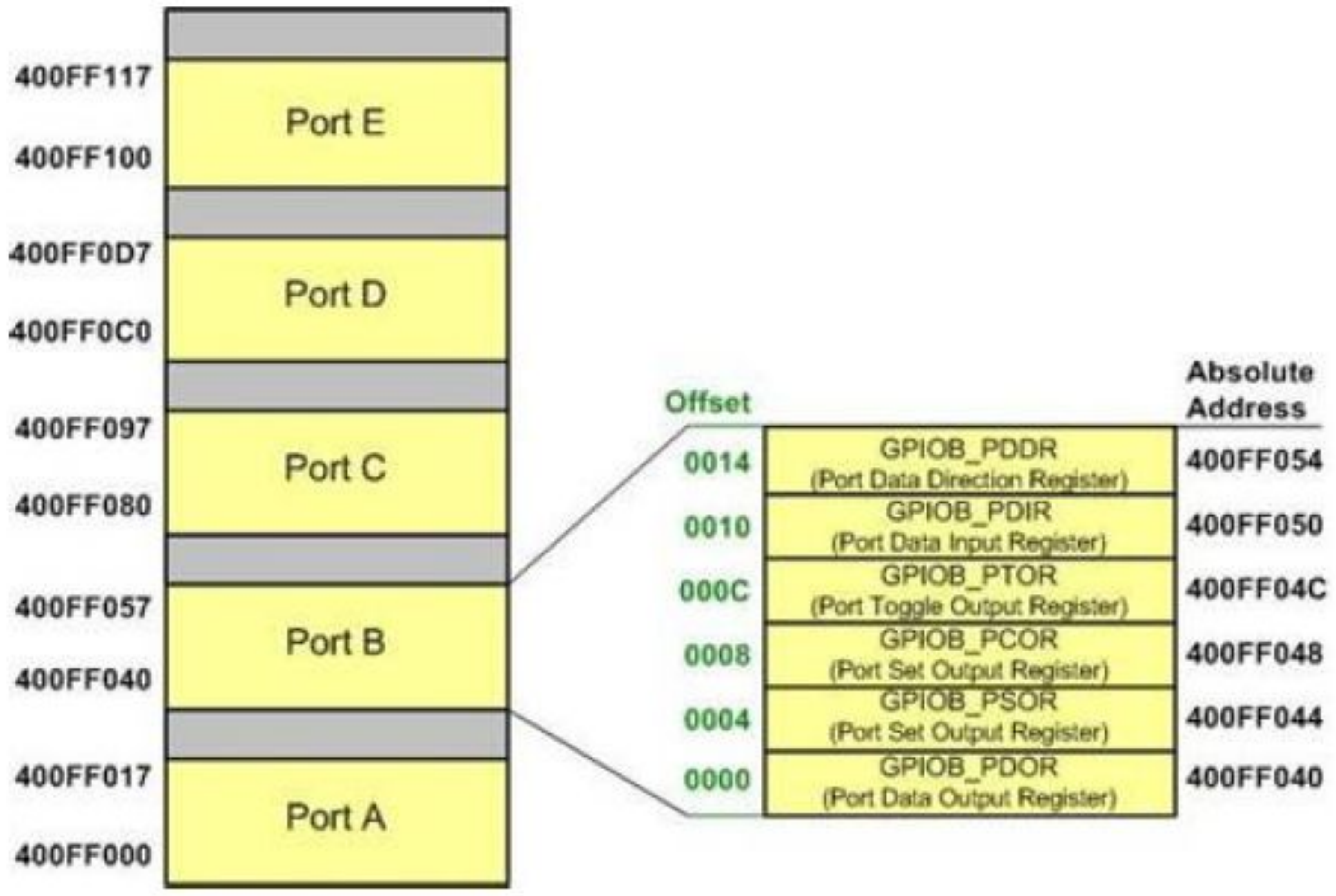
The ARM chips have two buses: **APB** (Advanced Peripheral Bus) and **AHB** (Advanced High-Performance Bus).

The I/O ports addresses assigned to the PTA-PTE for APB are as follow:

- ***GPIO Port A (APB):*** *0x400F F000*
- ***GPIO Port B (APB):*** *0x400F F040*
- ***GPIO Port C (APB):*** *0x400F F080*
- ***GPIO Port D (APB):*** *0x400F F0C0*
- ***GPIO Port E (APB):*** *0x400F F100*

Fundamentals – GPIO Programming

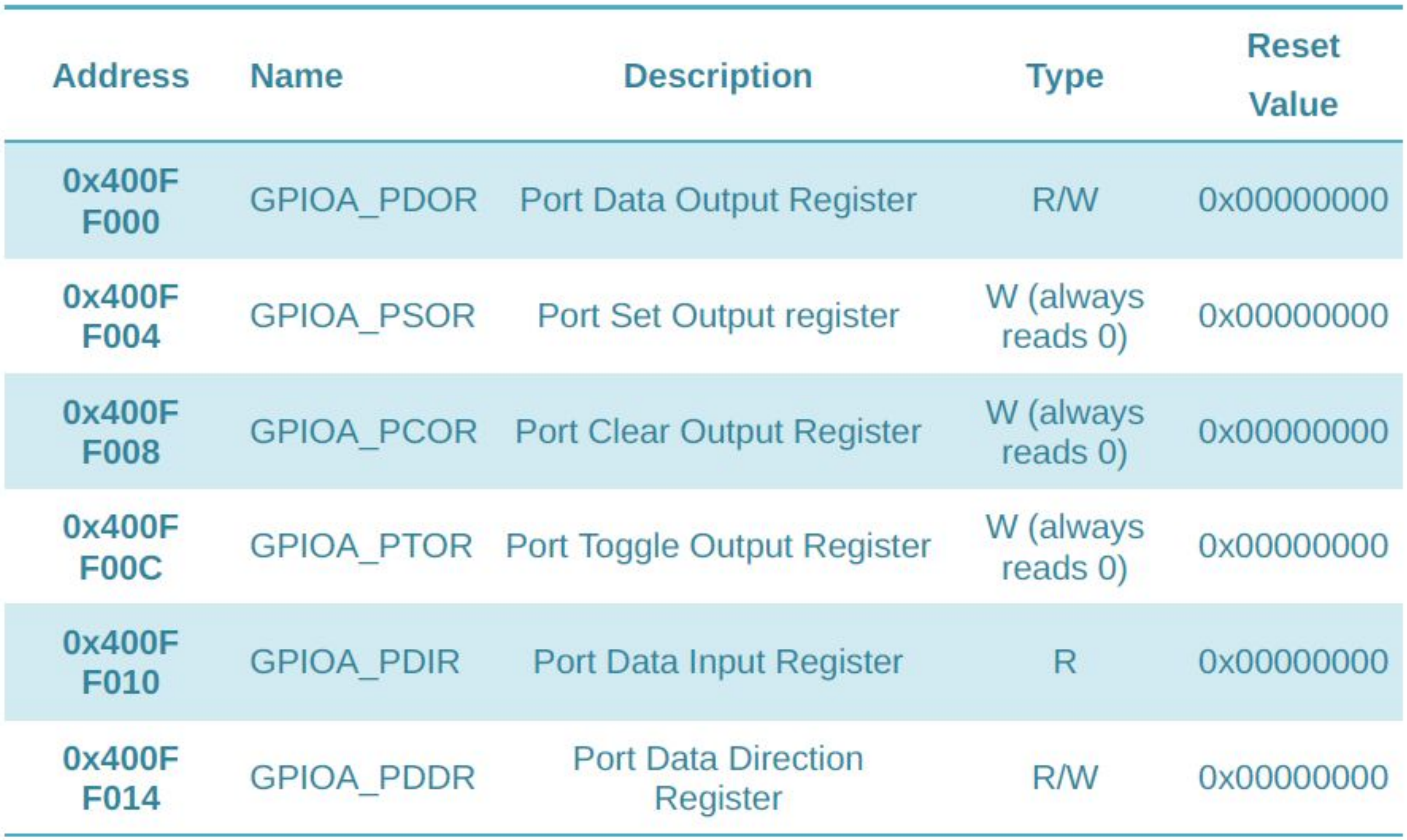
GPIO Memory Map



400FF117
400FF100
Port E
400FF0D7
400FF0C0
Port D
400FF097
400FF080
Port C
400FF057
400FF040
Port B
400FF017
400FF000
Port A
Offset
Absolute Address
0014
GPIOB_PDDR (Port Data Direction Register)
400FF054
0010
GPIOB_PDIR (Port Data Input Register)
400FF050
000C
GPIOB_PTOR (Port Toggle Output Register)
400FF04C
0008
GPIOB_PCOR (Port Set Output Register)
400FF048
0004
GPIOB_PSOR (Port Set Output Register)
400FF044
0000
GPIOB_PDOR (Port Data Output Register)
400FF040

Fundamentals – GPIO Programming

Direction and Data Registers



Address	Name	Description	Type	Reset Value
0x400F F000	GPIOA_PDOR	Port Data Output Register	R/W	0x00000000
0x400F F004	GPIOA_PSOR	Port Set Output register	W (always reads 0)	0x00000000
0x400F F008	GPIOA_PCOR	Port Clear Output Register	W (always reads 0)	0x00000000
0x400F F00C	GPIOA_PTOR	Port Toggle Output Register	W (always reads 0)	0x00000000
0x400F F010	GPIOA_PDIR	Port Data Input Register	R	0x00000000
0x400F F014	GPIOA_PDDR	Port Data Direction Register	R/W	0x00000000

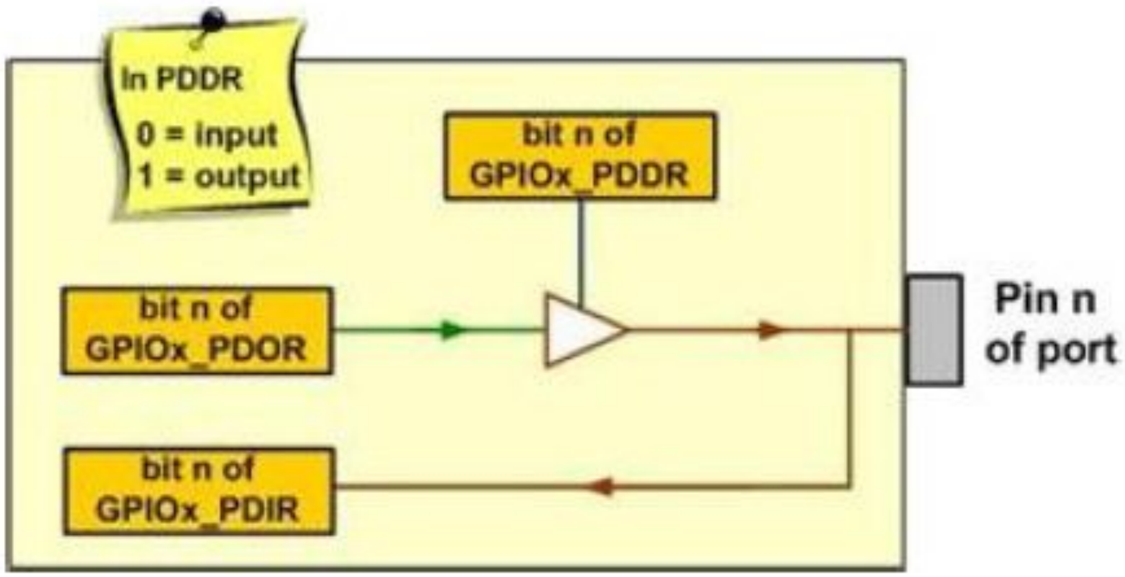
Fundamentals – GPIO Programming

Direction and Data Registers

Generally every microcontroller has minimum of two registers associated with each of I/O port.

They are ***Data Register*** and ***Direction Register***.

The Direction register is used to make the pin either input or output.



Fundamentals – GPIO Programming

Alternate pin functions and the simple GPIO

Each pin of the Freescale ARM chip can be used for one of several functions including GPIO.

We choose the function by programming a **special function register (SFR)**.

Using a single pin for multiple functions is called ***pin multiplexing*** and is widely used by microcontrollers.

In the absence of pin multiplexing, a microcontroller will need several hundred pins to support all of its on-chip features.

For example, a given pin can be used as simple digital I/O (GPIO), analog input, or I2C pin

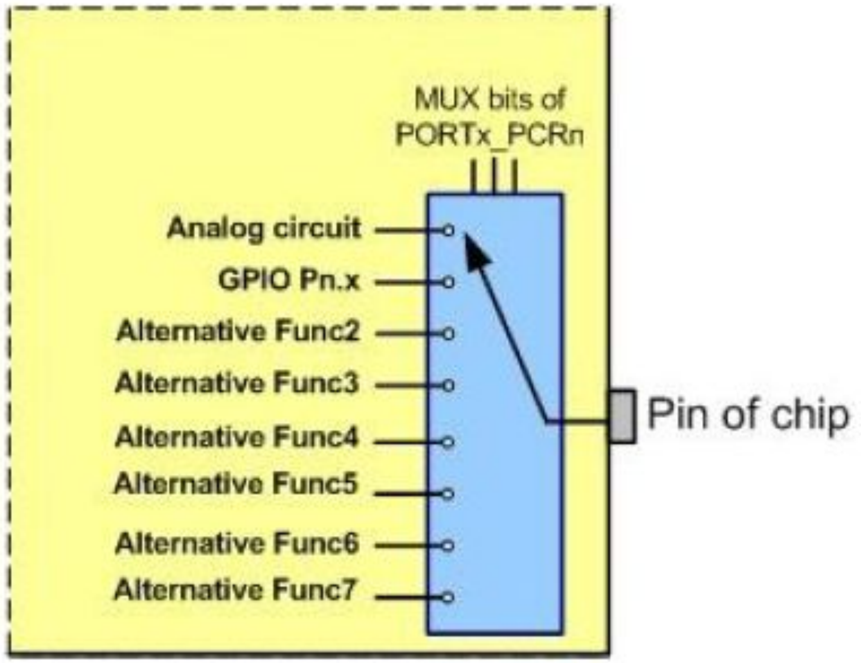
Fundamentals – GPIO Programming

Alternate pin functions and the simple GPIO

The **PORTx_PCRn** (Portx Pin Control) special function register allows us to program a pin to be used for a given alternate function.

It must be noted that **each pin** of ports A-E has its own **PORTx_PCRn register**.

The most important bits of PORTx_PCRn are D10-D8 (Mux control).



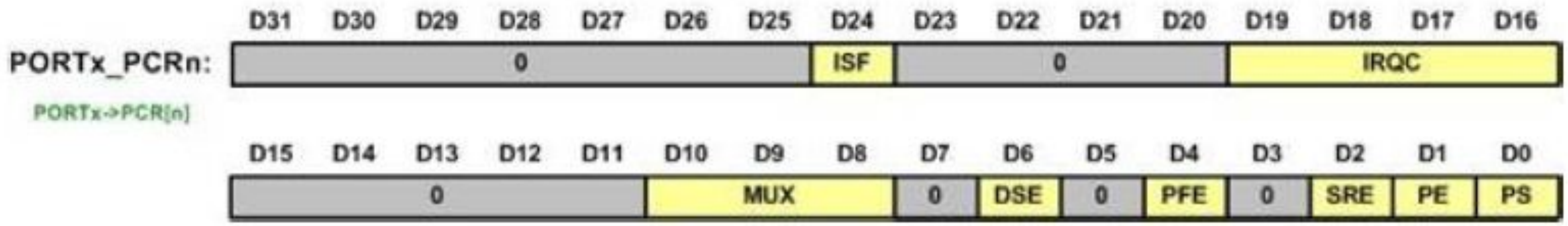
To use a pin as simple digital I/O, we must choose MUX=001

Fundamentals – GPIO Programming

GPIO Control Register

With the **PORTx_PCRn register**, not only we select the alternate I/O function of a given pin

We can also control its **Drive Strength** and its internal **Pull-up** (or Pull-down) resistor



PORTx_PCRn:

PORTx->PCR[n]

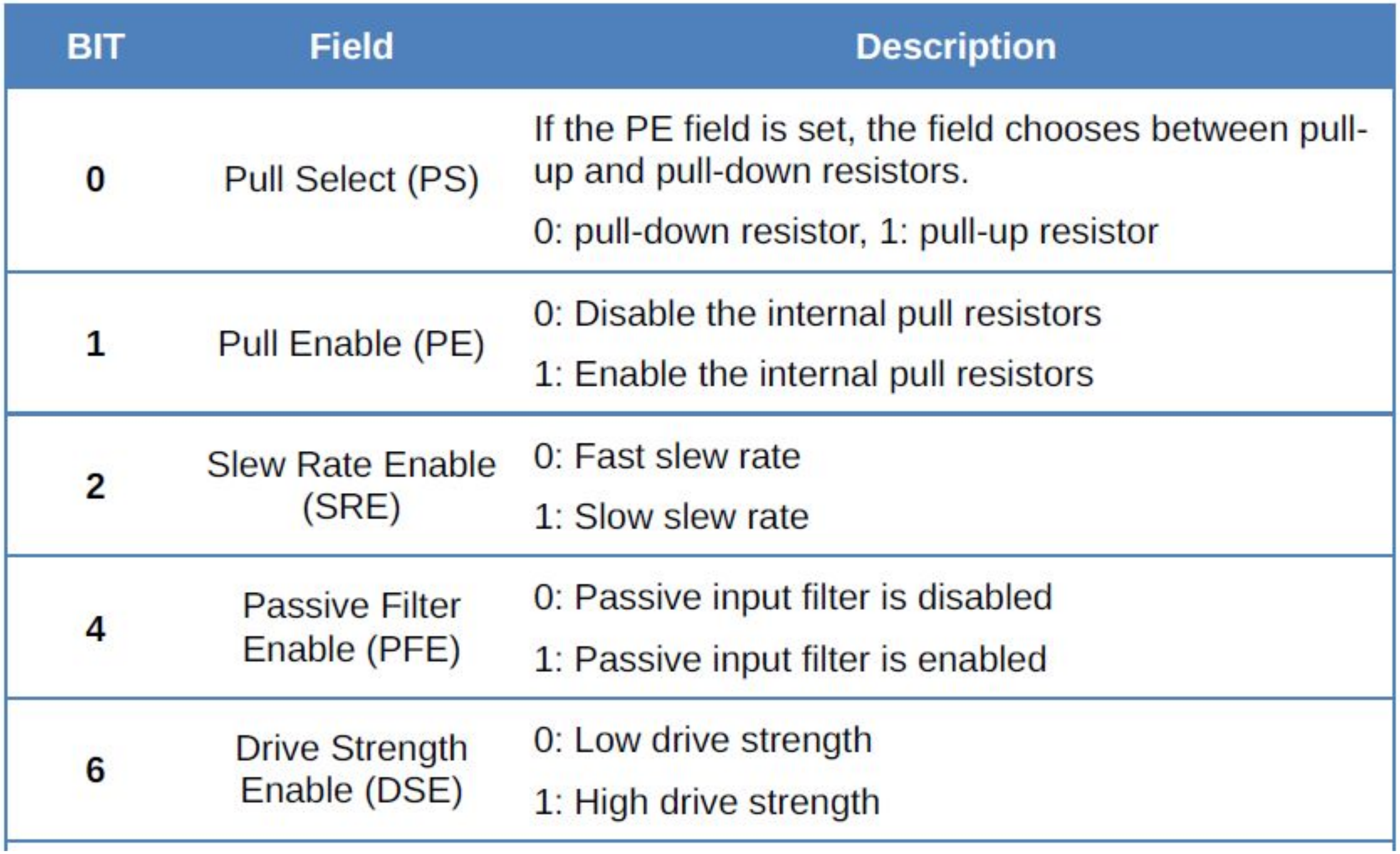
D31	D30	D29	D28	D27	D26	D25	D24	D23	D22	D21	D20	D19	D18	D17	D16
0							ISF	0				IRQC			

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0					MUX			0	DSE	0	PFE	0	SRE	PE	PS

We can also control the drive capability (fan-out and fan-in) of a digital I/O pin with D6 (Drive Strength Enable) bit.

Fundamentals – GPIO Programming

GPIO Control Register (bit descriptions)



BIT	Field	Description
0	Pull Select (PS)	If the PE field is set, the field chooses between pull-up and pull-down resistors. 0: pull-down resistor, 1: pull-up resistor
1	Pull Enable (PE)	0: Disable the internal pull resistors 1: Enable the internal pull resistors
2	Slew Rate Enable (SRE)	0: Fast slew rate 1: Slow slew rate
4	Passive Filter Enable (PFE)	0: Passive input filter is disabled 1: Passive input filter is enabled
6	Drive Strength Enable (DSE)	0: Low drive strength 1: High drive strength

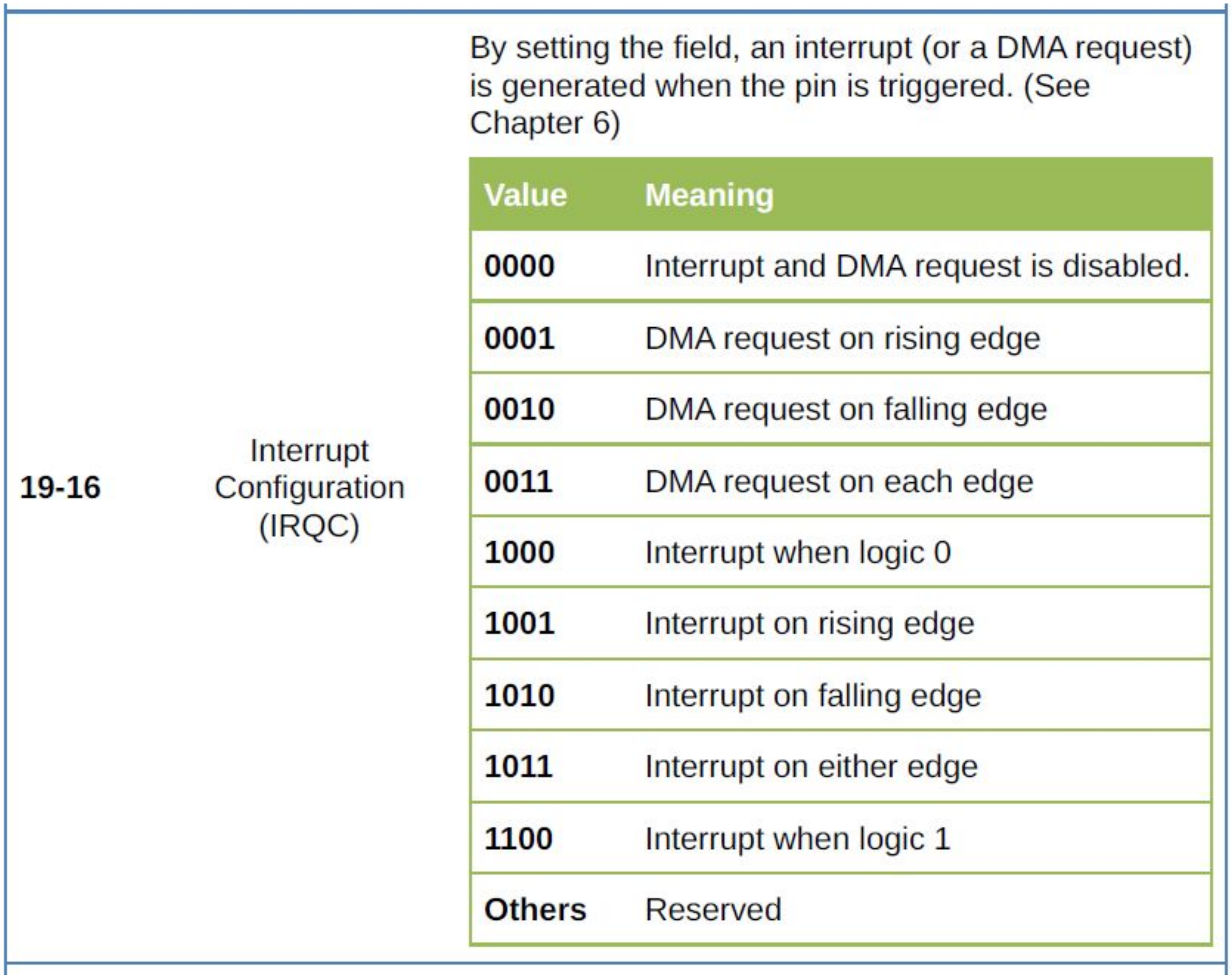
Fundamentals – GPIO Programming

GPIO Control Register (bit descriptions)

		value	Meaning
10-8	Pin Mux Control (MUX)	**000**	Pin disabled (analog)
		001	Alternative 1 (GPIO)
		010	Alternative 2 (Chip-specific)
		011	Alternative 3 (Chip-specific)
		100	Alternative 4 (Chip-specific)
		101	Alternative 5 (Chip-specific)
		110	Alternative 6 (Chip-specific)
		111	Alternative 7 (Chip-specific)

Fundamentals – GPIO Programming

GPIO Control Register (bit descriptions)



19-16	Interrupt Configuration (IRQC)	By setting the field, an interrupt (or a DMA request) is generated when the pin is triggered. (See Chapter 6)

Value	Meaning
0000	Interrupt and DMA request is disabled.
0001	DMA request on rising edge
0010	DMA request on falling edge
0011	DMA request on each edge
1000	Interrupt when logic 0
1001	Interrupt on rising edge
1010	Interrupt on falling edge
1011	Interrupt on either edge
1100	Interrupt when logic 1
Others	Reserved

Fundamentals – GPIO Programming

Example

In a given FRDM board the PTB18 and PTB19 are connected to LEDs to be used as simple I/O.

a) Configure the **GPIOB Direction** register for pins PTB18 and PTB19 to be digital output.

b) Give the address of the register in part a.

c) Configure the **PORTx_PCRn** registers for PTB18 and PTB19 pins. Assume slow slew rate, high drive, and no pull-up.

d) Give the address of each register in part C.

Fundamentals – GPIO Programming

Solution:

a) For any pins of **PORTB** to be used as an output, we need to set bits in **GPIOB_PDDR** (PORTB direction) to **high**. Now, we have 0x000C 0000 since D18 and D19 bit are set high)

b) The **PORTB_PDDIR** register is located at address 0x4000 F054 location.

c) For **PTB18**, we **have PORTB_PCR18=0x0000 0144** to configure it for I/O alternate, High Drive, slow slew rate, no pull resistor. The same is for PB19 which is **PORTB_PCR19=0x0000 0144**.

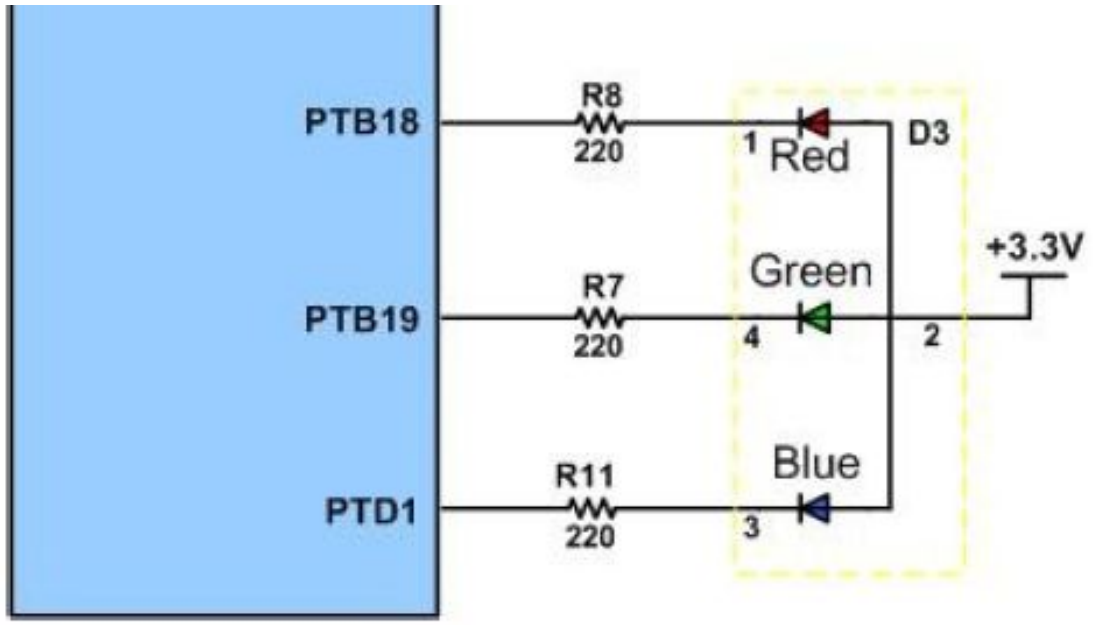
d) **PTB18** pin we have register PORTB_PCR18 and is located at address 0x4004 A048.

Fundamentals – GPIO Programming

LED connection in Freescale FRDM board

In the Freescale Freedom board we have a **tri-color RGB LED** connected to PTB18 (red), PTB19 (green), and PTD1 (blue).

The tri-color RGB (**red**, **blue**, **green)** LED is popular in many trainer kit for embedded systems.



Fundamentals – GPIO Programming

Toggling LEDs in Freescale FRDM board in C

To toggle the green LEDs of the FRDM board, the following steps must be followed:

1) enable the clock to PORTB, since access is denied to the port registers until the clock is enabled,
2) configure PortB_PCR19 (Pin Control Register) to select GPIO function for PTB19
3) set the Direction register bit 19 of PTB as output,
4) write HIGH to PTB19 in data register,
5) call a delay function,
6) write LOW to PTB19 in data register,
7) call a delay function,
8) Repeat steps 4 to 7.

Fundamentals – GPIO Programming

Program (1): Toggling a LED

```
#define GPIOB_PDOR (*((volatile unsignedint*)0x400FF040))

int main (void) {

void delayMs(int n);

SIM_SCGC5 |= 0x400; /* enable clock to Port B */
PORTB_PCR19 = 0x100; /* make PTB19 pin as GPIO */
GPIOB_PDDR |= 0x80000; /* make PTB19 as output pin */

while (1) {
GPIOB_PDOR &= ~0x80000; /* turn on green LED */
delayMs(500);
GPIOB_PDOR |= 0x80000; /* turn off green LED */
delayMs(500);
}}
```

Fundamentals – GPIO Programming

Program (1): Toggling a LED use inline if you wan to use it)

```
/* Delay n milliseconds
* The CPU core clock is set to MCGFLLCLK at 41.94 MHz
in SystemInit().*/

void delayMs(int n) {
int i;
int j;
for(i = 0 ; i < n; i++)
for (j = 0; j < 7000; j++) {}
}
```

Notice how we define the physical address of the special function registers belonging to the I/O ports. This is tedious and error prone

Often the manufacturer of the device will provide these definitions in a C header file.

Fundamentals – GPIO Programming

Program (1): Toggling a LED (use inline if you wan to use it)

```
/* Delay n milliseconds
* The CPU core clock is set to MCGFLLCLK at 41.94 MHz
in SystemInit().*/

void delayMs(int n) {
int i;
int j;
for(i = 0 ; i < n; i++)
for (j = 0; j < 7000; j++) {}
}
```

Notice how we define the physical address of the special function registers belonging to the I/O ports. This is tedious and error prone

Often the manufacturer of the device will provide these definitions in a C header file.

Fundamentals – Timers and Counters

Program (5): Delay using TMP delay

```
void delayMs(int n) {
int i;
SysTick->LOAD = 41940 - 1;
SysTick->CTRL = 0x5; /* Enable the timer and
choose sysclk as the clock
source */
for(i = 0; i < n; i++) {
while((SysTick->CTRL & 0x10000) == 0)
/* wait until the COUNT flag is set */
{ }
}
SysTick->CTRL = 0;
/* Stop the timer (Enable = 0) */
}
```

Fundamentals – GPIO Programming

Program (2): Toggling a LED

```
/*  Toggling  LED  in  C  using  Freescale  header  file
register definitions.
* This program toggles green LED for 0.5 second ON and
0.5 second OFF.
* The green LED is connected to PTB19.
* The LEDs are low active (a ‘0’ turns ON the LED).
*/

#include “C:\Freescale\CW MCU
v10.5\MCU\ProcessorExpert\lib\Kinetis\iofiles\MKL25Z4.H”
#include

“C:\Freescale\KDS_1.0\eclipse\ProcessorExpert\lib\Kineti
s\iofiles\MKL25Z4.H”
*/
```

Fundamentals – GPIO Programming

Program (2): Toggling a LED

Keil MDK-ARM uses a different syntax to define the registers to be in compliant with CMSIS (Cortex Microcontrollers Software Interface Standard).

In this syntax, each port is defined as a pointer to a struct with the registers as the members of the struct.

For example:

the Direction Register of Port B is referred to as

PTB->PDDR and

the Data Register of Port B is referred to as

PTB->PDOR and so on.

Fundamentals – GPIO Programming

Program (2): Toggling a LED

```
#include <MKL25Z4.H>
int main (void) {
void delayMs(int n);

SIM->SCGC5 |= 0x400; /* enable clock to Port B */
PORTB->PCR[19] = 0x100; /* make PTB19 pin as GPIO */
PTB->PDDR |= 0x80000; /* make PTB19 as output pin */

while (1) {
PTB->PDOR &= ~0x80000; /* turn on green LED */
delayMs(500);
PTB->PDOR |= 0x80000; /* turn off green LED */
delayMs(500);
}}
```

Fundamentals – GPIO Programming

GPIO Auxiliary Registers

The **Kinetis Family GPIO ports** have three additional registers that make turning a pin (or more) on and off easier.

They are **PSOR** (Port Set Output Register), **PCOR** (Port Clear Output Register), **PTOR** (Port Toggle Output Register).

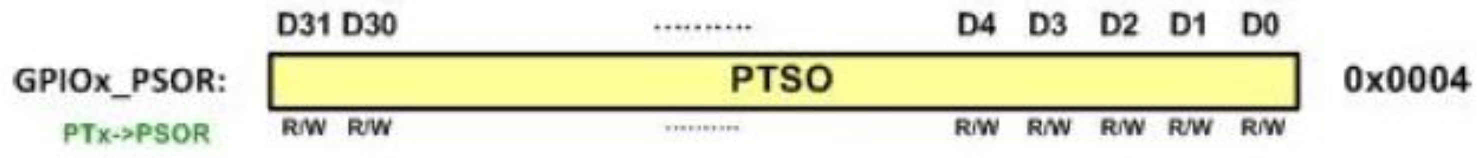
Writing to these registers only affects the pin(s) that the corresponding bit(s) in the value written.

This makes it **easier to turn on or off a single pin** or a few pins without affecting the other pins.

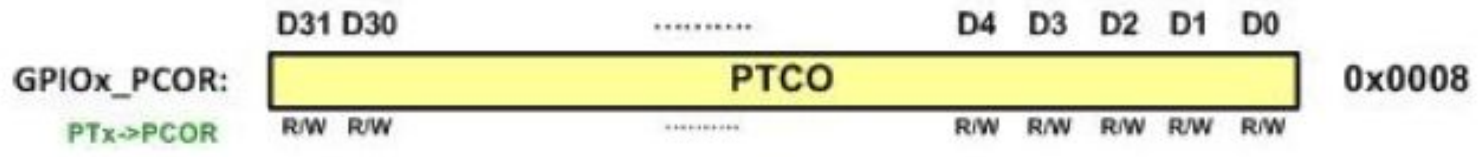
For example, writing a value 4 (0100 in binary) to PCOR will turn off bit 2 of that port without modifying any other pins.

Fundamentals – GPIO Programming

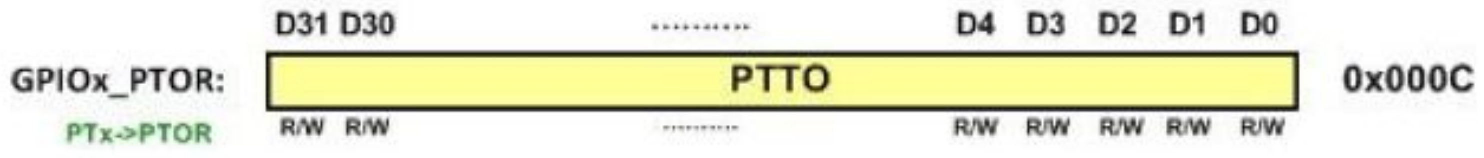
GPIO Auxiliary Registers


D31 D30 .... D4 D3 D2 D1 D0
GPIOx_PSOR:
PTx->PSOR
PTSO
0x0004
R/W R/W .... R/W R/W R/W R/W R/W

PSOR (Port Set Output Register)


D31 D30 .... D4 D3 D2 D1 D0
GPIOx_PCOR:
PTx->PCOR
PTCO
0x0008
R/W R/W .... R/W R/W R/W R/W R/W

PCOR (Port Clear Output Register)


D31 D30 .... D4 D3 D2 D1 D0
GPIOx_PTOR:
PTx->PTOR
PTTO
0x000C
R/W R/W .... R/W R/W R/W R/W R/W

PTOR (Port Toggle Output Register)

Fundamentals – GPIO Programming

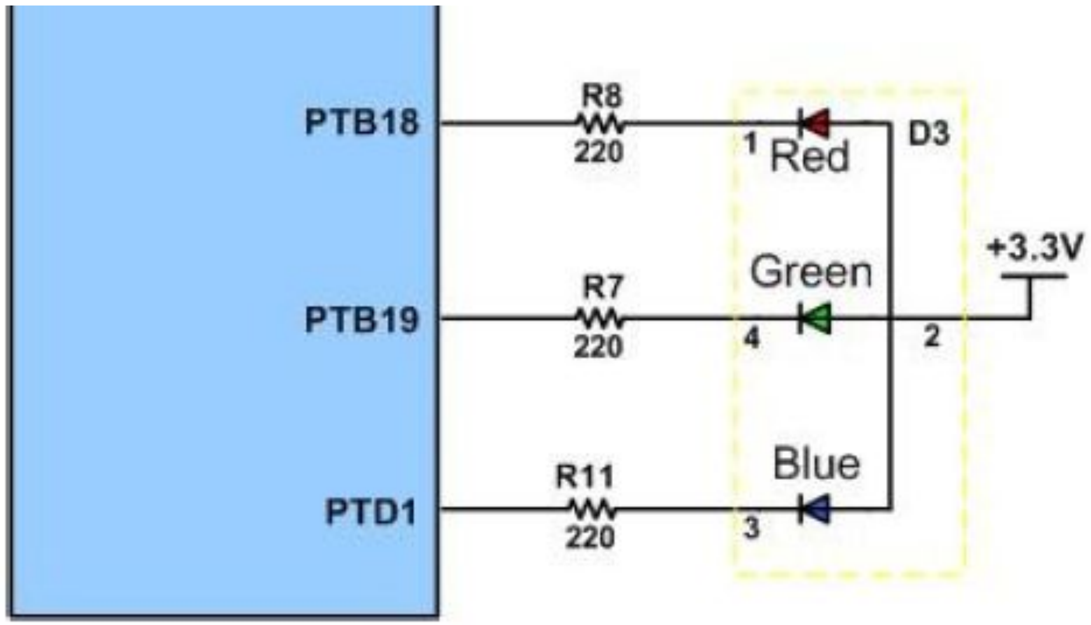
Program (2): Toggling a LED using auxiliary registers

```
#include <MKL25Z4.H>

int main (void) {
void delayMs(int n);
SIM->SCGC5 |= 0x400; /* enable clock to Port B */
PORTB->PCR[19] = 0x100; /* make PTB19 pin as GPIO)*/
PTB->PDDR |= 0x80000; /* make PTB19 as output pin */
while (1) {
PTB->PCOR = 0x80000; /* turn on green LED */
delayMs(500);
PTB->PSOR = 0x80000; /* turn off green LED */
delayMs(500);
PTB->PTOR = 0x80000; /* Toggle green LED */
delayMs(500);
PTB->PTOR = 0x80000; /* Toggle green LED */
delayMs(500);}}
```

Fundamentals – GPIO Programming

Short exercise 1:

Write a C program for toggling all the 3 LEDs in the KL25Z board.



Fundamentals – GPIO Programming

```
#include <MKL25Z4.H>
int main (void) {

void delayMs(int n);

SIM->SCGC5 |= 0x400; /* enable clock to Port B */

SIM->SCGC5 |= 0x1000; /* enable clock to Port D */

PORTB->PCR[18] = 0x100; /* make PTB18 pin as GPIO */

PORTB->PCR[19] = 0x100; /* make PTB19 pin as GPIO */

PTB->PDDR |= 0xC0000; /* make PTB18, 19 as output pin */

PORTD->PCR[1] = 0x100; /* make PTD1 pin as GPIO */

PTD->PDDR |= 0x02; /* make PTD1 as output pin */
```

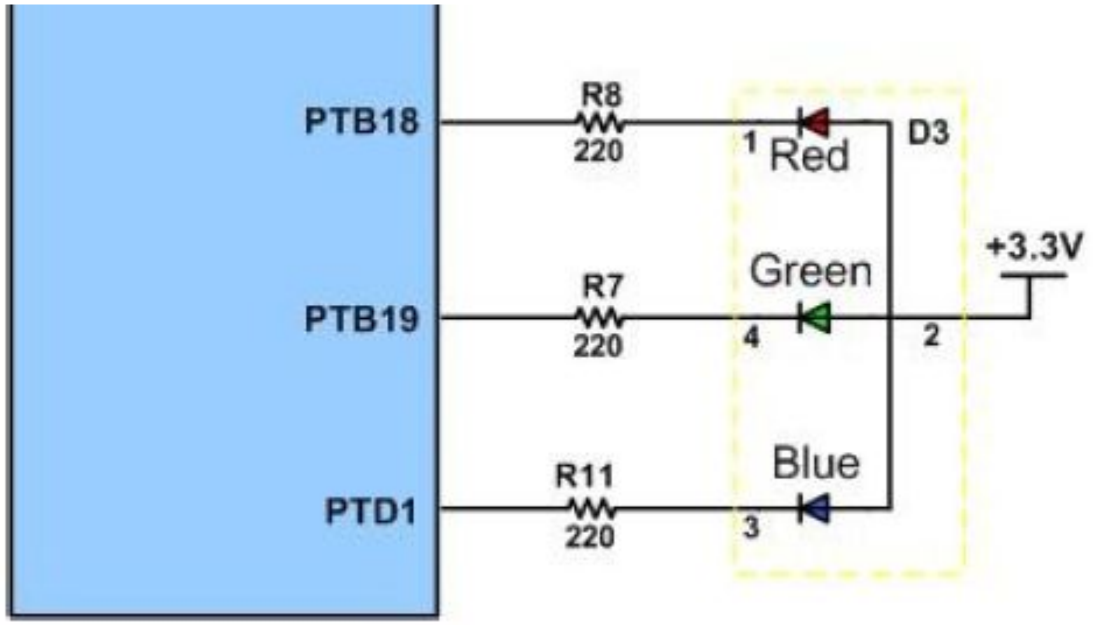
Fundamentals – GPIO Programming

```
while (1) {

/* turn on red and green LED */
PTB->PDOR &= ~0xC0000;

/* turn on blue LED */
PTD->PDOR &= ~0x02;

delayMs(500);

/* turn off red and green LED */
PTB->PDOR |= 0xC0000;

/* turn off blue LED */
PTD->PDOR |= 0x02;
delayMs(500);
}}
```

Fundamentals – GPIO Programming

Short exercise 2:

Write a program to generate all the combinations of to drive the tri-color led, an incrementing counter should be used.



Fundamentals – GPIO Programming

Program (4): Combining different colors of the RBG LED

```
#include <MKL25Z4.H>

int main (void) {
void delayMs(int n);
int counter = 0;

SIM->SCGC5 |= 0x400; /* enable clock to Port B */
SIM->SCGC5 |= 0x1000; /* enable clock to Port D */
PORTB->PCR[18] = 0x100; /* make PTB18 pin as GPIO */
PTB->PDDR |= 0x40000; /* make PTB18 as output pin */
PORTB->PCR[19] = 0x100; /* make PTB19 pin as GPIO */
PTB->PDDR |= 0x80000; /* make PTB19 as output pin */
PORTD->PCR[1] = 0x100; /* make PTD1 pin as GPIO */
PTD->PDDR |= 0x02; /* make PTD1 as output pin */
```

Fundamentals – GPIO Programming

```
while (1) {

/* use bit 0 of counter to control red LED */
if (counter & 1)

/* turn on red LED */
PTB->PCOR = 0x40000;
else
PTB->PSOR = 0x40000; /* turn off red LED */

/* use bit 1 of counter to control green LED */
if (counter & 2)
/* turn on green LED */
PTB->PCOR = 0x80000;
else
PTB->PSOR = 0x80000; /* turn off green LED */
```

Fundamentals – GPIO Programming

```
/* use bit 2 of counter to control blue LED */
if (counter & 4)

 /* turn on blue LED */
PTD->PCOR = 0x02;
else
PTD->PSOR = 0x02; /* turn off blue LED */

counter++;

delayMs(500);

}
```

Fundamentals – GPIO Programming

Reading a switch in Freescale FRDM board

The FRDM KL25Z board does not come with any user programmable pushbutton switches.

We can connect an external SW to the board and experiment with the concept of inputting data via a port.

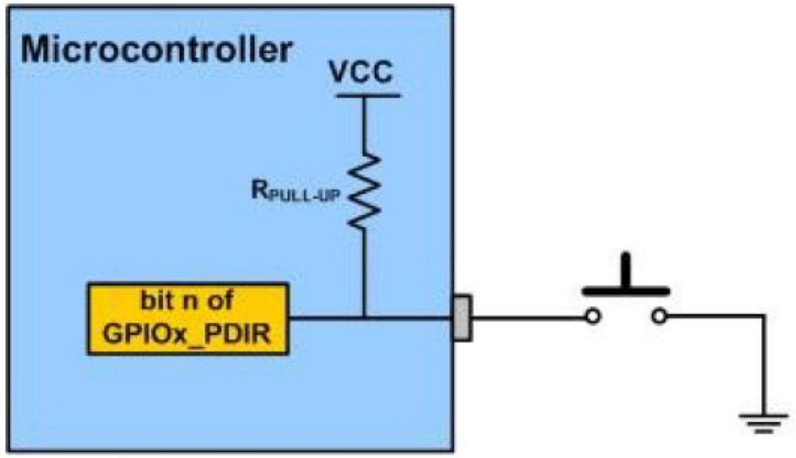
Depending on how we connect an external SW to a pin, we need to enable the internal pull-up or pull-down resistor for a pin.

Using the PORTx_PCRn register, not only we select the alternate I/O function of a given pin, we can also control its Drive Strength and its internal Pullup (or Pull-down) resistor.

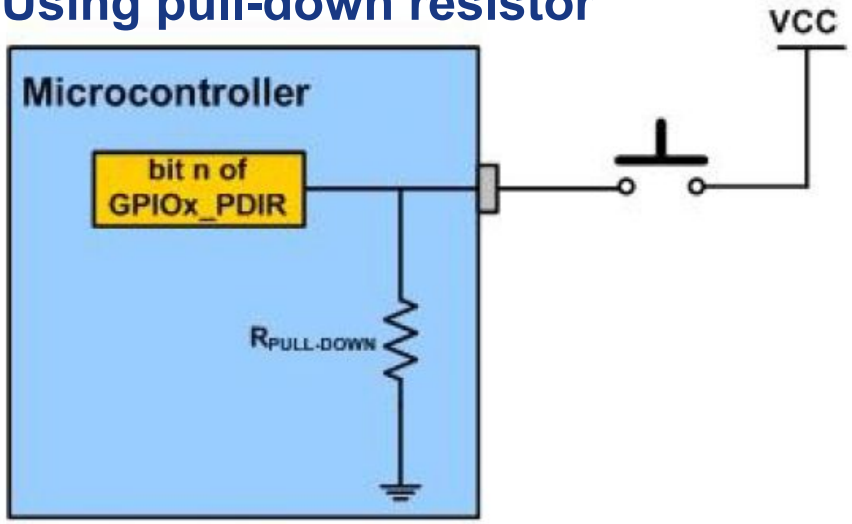
Fundamentals – GPIO Programming

Reading a switch in Freescale FRDM board

Using pull-up resistor



Using pull-down resistor

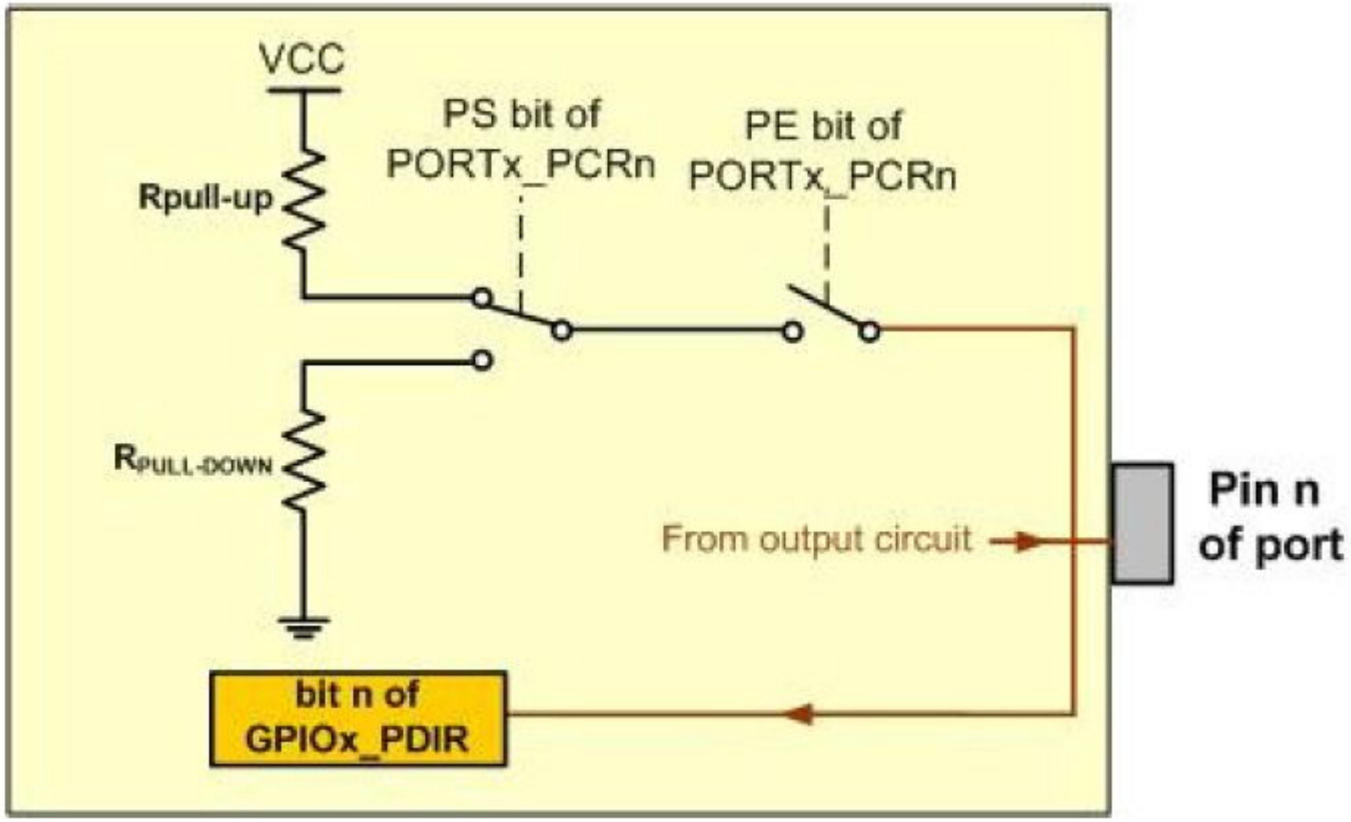


The D1 (PE, pull enable) bit of the PORTx_PCRn is used to enable the internal Pull resistor option.

If PE=1, then we use the D0 bit (PS, pull select) to enable the pull-up (or pull-down) option

Fundamentals – GPIO Programming

Reading a switch in Freescale FRDM board



Fundamentals – GPIO Programming

Example

Find the contents of the PORTA_PCR1 register for PTA1 to use PTA1 pin as input connected to SW. Use the pull-up resistor.

0	0	0	0000	00000	001	0	0	0	0	0	1	1	1
Reserved	ISF	Reserved	IRQC	Res.	MUX	0	DSE	0	PFE	0	SRE	PE	PS

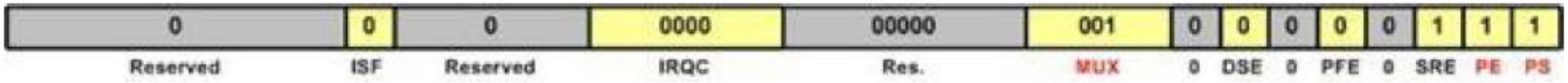
Solution:

To read a switch on PTA1 and display it on the LED on PTB19, the following steps must be taken.
1) enable the clock to PORTB,
2) configure PTB19 as GPIO in PORTB_PCR19 register,
3) make PTB19 output in PDDR register

Fundamentals – GPIO Programming

Then, the following steps are used for setting up the input pin and the variables

4) enable the clock to PORTA
5) configure PTA1 as GPIO and enable the pull-up resistor in PORTA PCR1 register,
6) make PTA1 input in PDDR register,
7) read switch from PORTA,
8) if PTA1 is high, set PTB1
9) else clear PTB1
10) Repeat steps 7 to 9.



0	0	0	0000	00000	001	0	0	0	0	0	1	1	1
Reserved	ISF	Reserved	IRQC	Res.	MUX	0	DSE	0	PFE	0	SRE	PE	PS

Fundamentals – GPIO Programming

Program (5): Reading the value from a digital input

```
int main (void) {

SIM->SCGC5 |= 0x400; /* enable clock to Port B */

PORTB->PCR[19] = 0x100; /* make PTB19 pin as GPIO */

PTB->PDDR |= 0x80000; /* make PTB19 as output pin */

SIM->SCGC5 |= 0x200; /* enable clock to Port A */
PORTA->PCR[1] = 0x103; /* make PTA1GPIO & en pull-up */

PTA->PDDR &= ~0x02; /* make PTA1 as input pin */
```

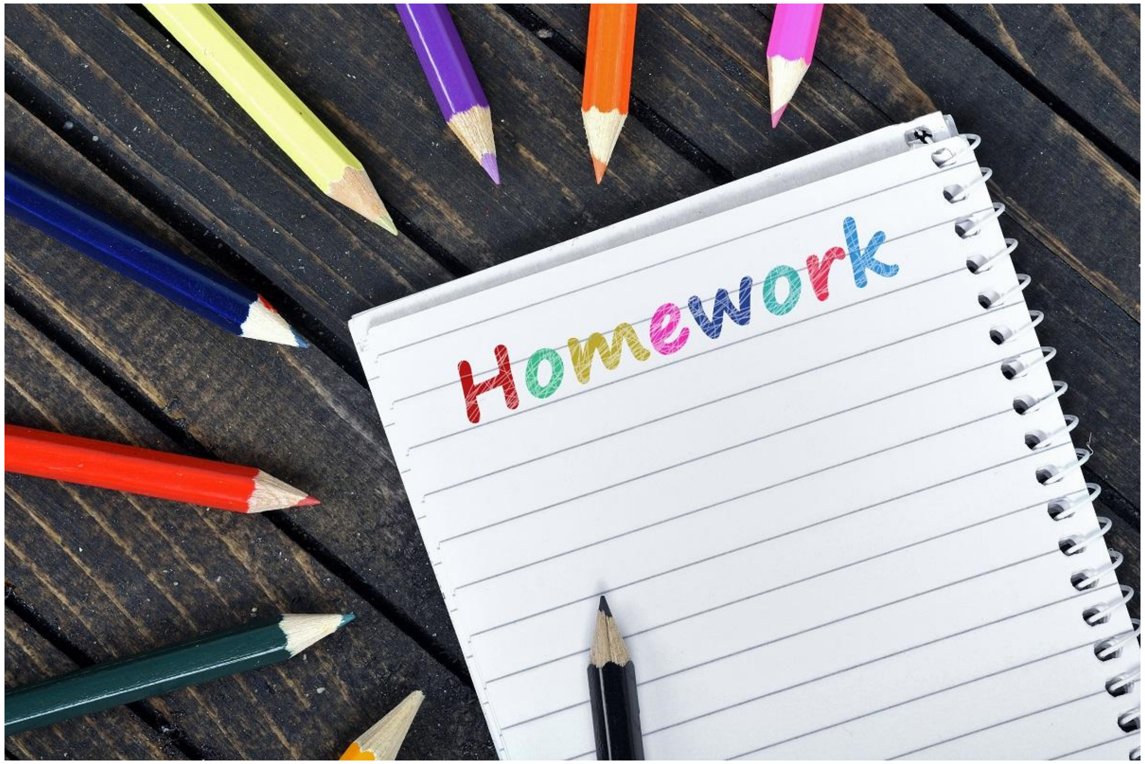
Fundamentals – GPIO Programming

Program (5): Reading the value from a digital input

```
while (1) {

/* check to see if switch is pressed */
if (PTA->PDIR & 2)
/* if not, turn off green LED */
PTB->PSOR = 0x80000;
else
PTB->PCOR = 0x80000; /* turn on green LED */
}
}
```

Fundamentals – GPIO Programming

Contact Bounce and Debounce



Fundamentals – Parallel GPIO Programming

What will we learn?

Interfacing to an LCD

Sending Commands and Data to an LCD

LCD 4-bit option

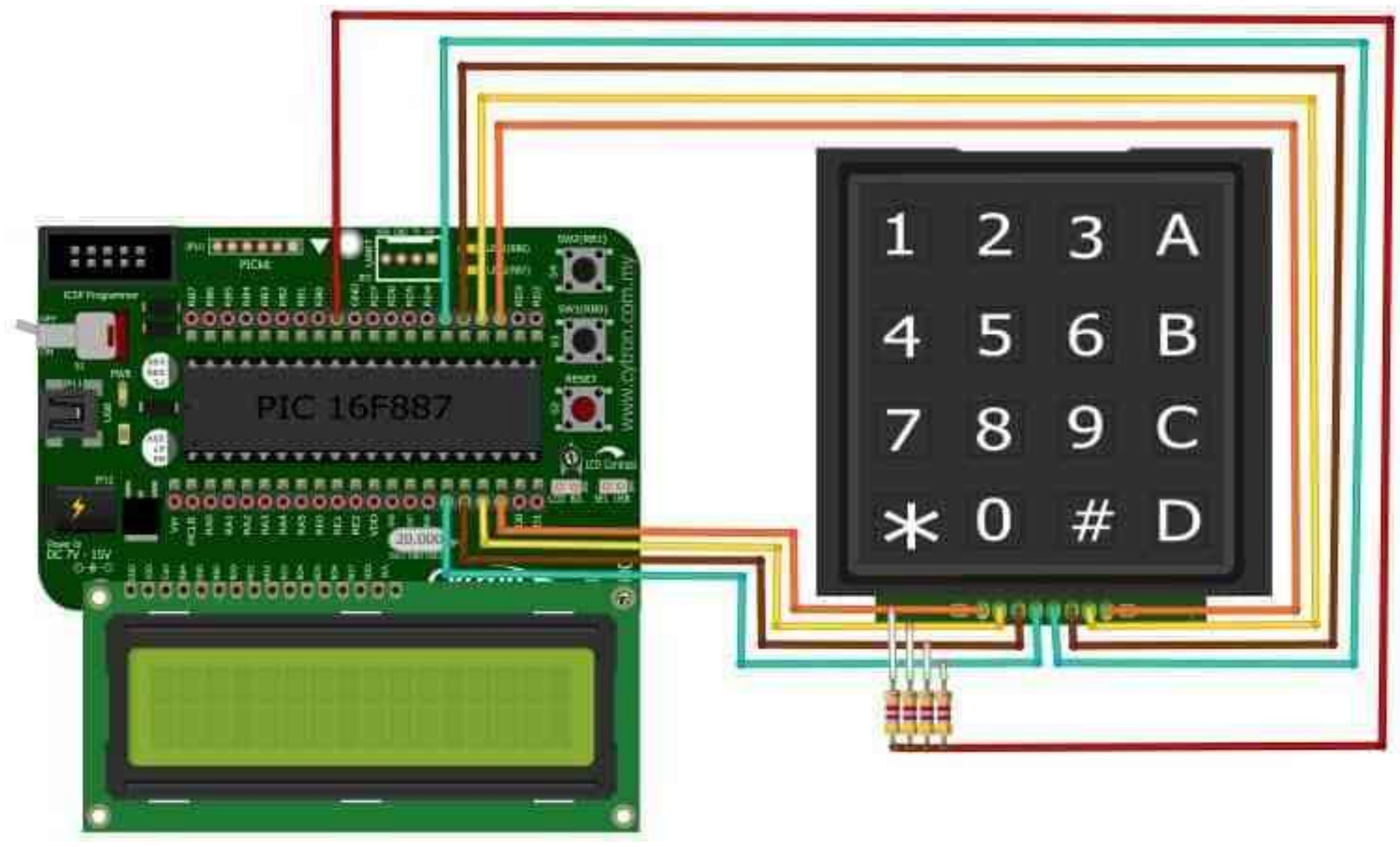
Interfacing the Keyboard to the Microcontroller

Key press detection and ID

Project: LCD + Keyboard!

Fundamentals – Parallel GPIO Programming

LCD and Keyboard Interfacing



Fundamentals – Parallel GPIO Programming

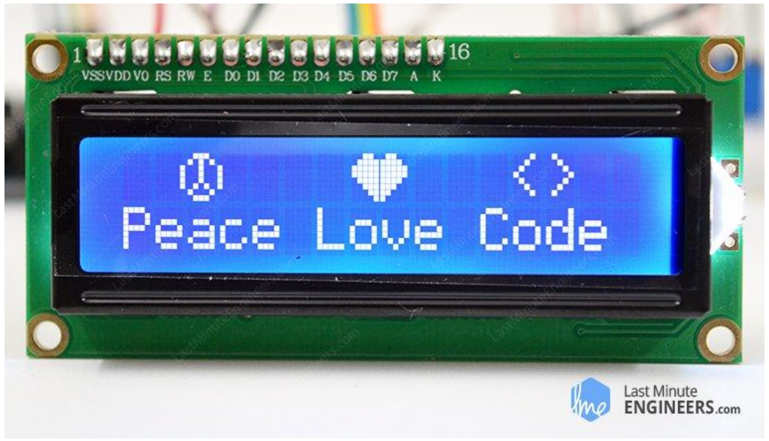
LCD and Keyboard Interfacing (examples)

Fundamentals – Parallel GPIO Programming

Interfacing to an LCD

This section describes the operation modes of the LCDs, then describes how to program and interface an LCD to the Freescale FRDM board.

LCD operation


1
VSS VDD V0 RS RW E D0 D1 D2 D3 D4 D5 D6 D7 A K
16
Peace Love Code
Last Minute ENGINEERS.com

In recent years the LCD is replacing LEDs (seven-segment LEDs or other multi-segment LEDs). This is due to the following reasons:

Fundamentals – Parallel GPIO Programming

Interfacing to an LCD

Why are LCDs so popular?

1. The declining prices of LCDs.

2. The ability to display numbers, characters, and graphics. This is in contrast to LEDs, which are limited to numbers and a few characters.

3. Incorporation of the refreshing controller into the LCD itself, thereby relieving the CPU of the task of refreshing the LCD.

4. Ease of programming for both characters and graphics.

5. The extremely low power consumption of LCD (when backlight is not used).

LCD module pin descriptions

For many years, the use of Hitachi HD44780 LCD controller dominated the character LCD modules.

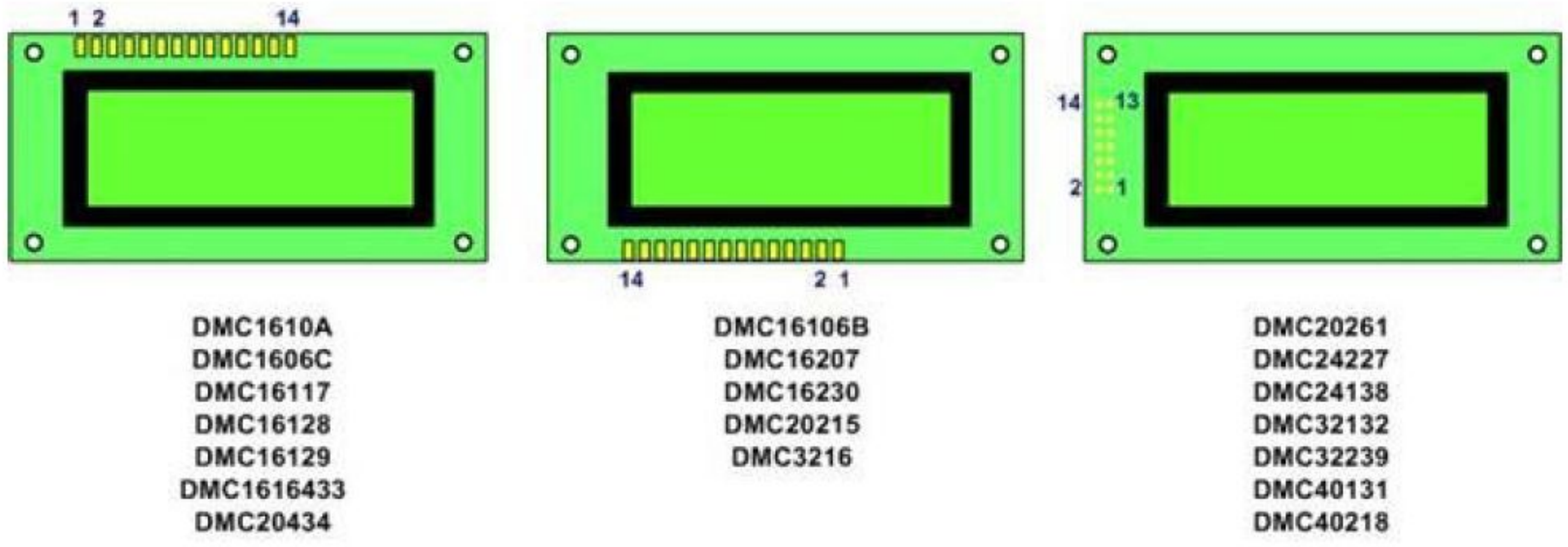
Even today, most of the character LCD modules still use HD44780 or a variation of it.

The HD44780 controller has a 14 pin interface for the microprocessor.

We will discuss this 14 pin interface in this section.

Fundamentals – Parallel GPIO Programming

LCD module pin descriptions



Fundamentals – Parallel GPIO Programming

Online Resources

https://learningmicro.wordpress.com/interfacing-lcd-with-kl25z-freedom-board/

https://www.electroduino.com/16x2-lcd-display-module-how-its-works/

https://exploreembedded.com/wiki/LCD_16_x_2_Basics

LCD module pin descriptions

Pin	Symbol	I/O	Description
1	VSS	—	Ground
2	VCC	—	+5V power supply
3	VEE	—	Power supply to control contrast
4	RS	I	RS = 0 to select command register, RS = 1 to select data register
5	R/W	I	R/W = 0 for write, R/W = 1 for read
6	E	I	Enable
7	DB0	I/O	The 8-bit data bus

LCD module pin descriptions

VEE: VEE is used for controlling the LCD contrast.

RS, register select: There are two registers inside the LCD and the RS pin is used for their selection as follows.

If RS = 0, the instruction **command code register is selected**, allowing the user to send a command such as clear display, cursor at home, and so on.
If RS = 1, the **data register is selected**, allowing the user to send data to be displayed on the LCD (or to retrieve data from the LCD controller).

R/W, read/write: R/W input allows the user to write information into the LCD controller or read information from it. R/W = 1 when reading and R/W = 0 when writing.

Fundamentals – Parallel GPIO Programming

LCD module pin descriptions

Pin	Symbol	I/O	Description
8	DB1	I/O	The 8-bit data bus
9	DB2	I/O	The 8-bit data bus
10	DB3	I/O	The 8-bit data bus
11	DB4	I/O	The 4/8-bit data bus
12	DB5	I/O	The 4/8-bit data bus
13	DB6	I/O	The 4/8-bit data bus
14	DB7	I/O	The 4/8-bit data bus

Fundamentals – Parallel GPIO Programming

LCD module pin descriptions

E, enable: The enable pin is used by the LCD to latch information presented to its data pins. When data is supplied to data pins

D0–D7: The 8-bit data pins are used to send information to the LCD or read the contents of the LCD's internal registers.

The LCD controller is capable of operating with 4-bit data and only D4-D7 are used. We will discuss this in more details later.

To display letters and numbers, we send ASCII codes for the letters A–Z, a–z, numbers 0–9, and the punctuation marks to these pins while making RS = 1

Fundamentals – Parallel GPIO Programming

LCD module pin descriptions ASCII

Fundamentals – Parallel GPIO Programming

Sending commands to LCDs

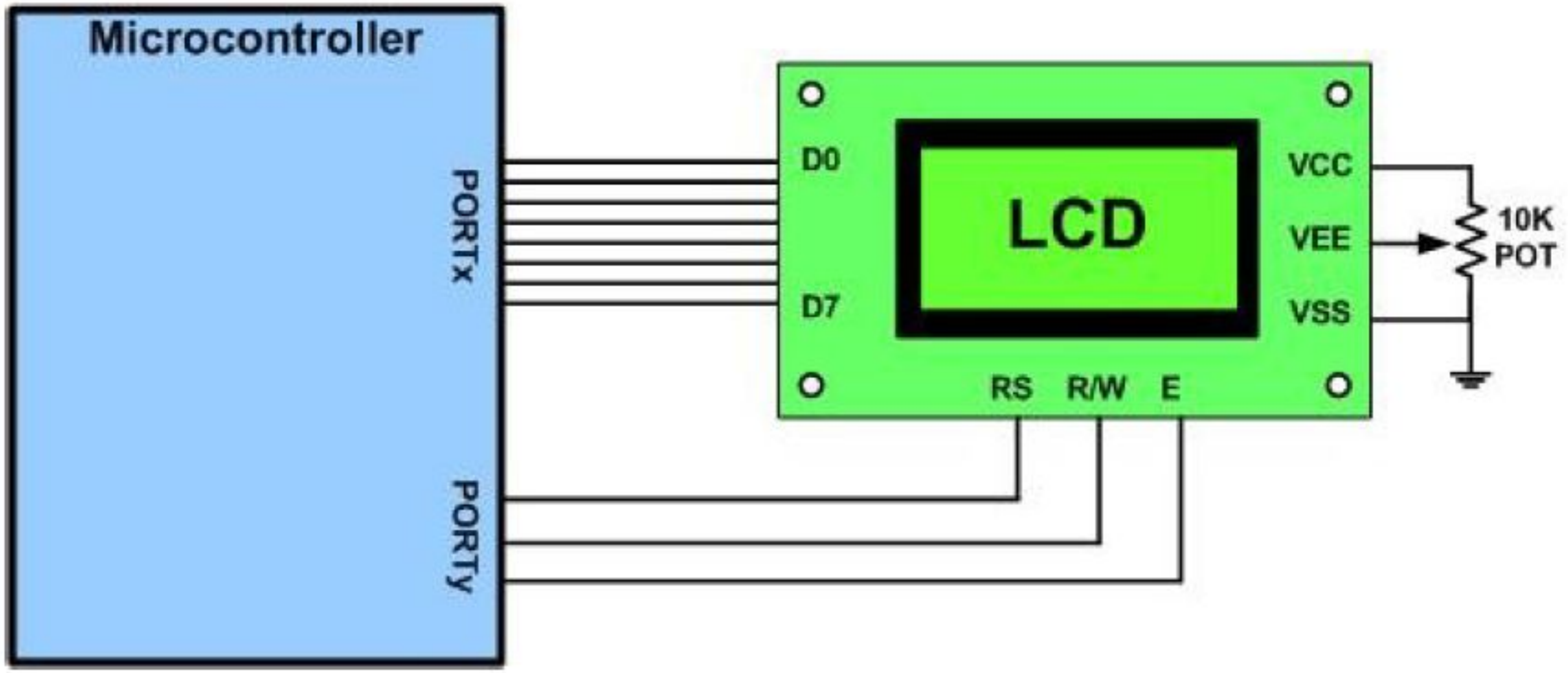
There are also instruction command codes that can be sent to the LCD in order to clear the display, force the cursor to the home position, or blink the cursor

Code (Hex)	Command to LCD Instruction Register
1	Clear display screen
2	Return cursor home
6	Increment cursor (shift cursor to right)
F	Display on, cursor blinking
80	Force cursor to beginning of 1st line
C0	Force cursor to beginning of 2nd line
38	2 lines and 5x7 character (8-bit data, D0 to D7)
28	2 lines and 5x7 character (4-bit data, D4 to D7)

Fundamentals – Parallel GPIO Programming

Sending commands to LCDs

To send any of the commands to the LCD, make pins RS = 0, R/W = 0, and send a pulse (L-to-H-to-L) on the E pin to enable the internal latch of the LCD.



Fundamentals – Parallel GPIO Programming

Sending commands to LCDs

Notice the following for the connection in previous slide

1. The **LCD's data pins** are connected to **PORTD** of the microcontroller.

2. The **LCD's RS** pin is connected to **Pin 2 of PORTA** of the microcontroller.

3. The **LCD's R/W** pin is connected to **Pin 4 of PORTA** of the microcontroller.

4. The **LCD's E pin** is connected to **Pin 5 of PORTA** of the microcontroller. 5. Both Ports D and A are configured as output ports.

Sending data to the LCD

In order to send data to the LCD to be displayed, we must set pins RS = 1, R/W = 0, and also send a pulse (L-to-H-to-L) to the E pin to enable the internal latch of the LCD.

Because of the extremely low power feature of the LCD controller, it runs much slower than the microcontroller

After one command or data is written to the LCD controller, one must wait until the LCD controller is ready before issuing the next command/data

An easy way (not as efficient though) is to delay the microcontroller for the previous command.

Program (7): Sending data to the LCD

```
/*  Initialize and display “Hello” on the LCD using
8-bit data mode.

* Data pins use Port D, control pins use Port A.
* This program does not poll the status of the LCD.
* It uses delay to wait out the time LCD controller is
busy.

* Timing is more relax than the HD44780 datasheet to
accommodate the variations among the LCD modules.

* You may want to adjust the amount of delay for your
LCD controller. */
```

Fundamentals – Parallel GPIO Programming

Program (7): Definitions and initialization

```
#include <MKL25Z4.H>

#define RS 0x04 /* PTA2 mask */
#define RW 0x10 /* PTA4 mask */
#define EN 0x20 /* PTA5 mask */

void delayMs(int n);

void LCD_command(unsigned char command);

void LCD_data(unsigned char data);

void LCD_init(void);
```

Fundamentals – Parallel GPIO Programming

Program (7): Main program “Hello World”

```
int main(void){
LCD_init();
for(;;)
{
LCD_command(1); /* clear display */
delayMs(500);
LCD_command(0x80); /* set cursor at first line */
LCD_data(‘H’);     /* write the word */
LCD_data(‘e’);
LCD_data(‘l’);
LCD_data(‘l’);
LCD_data(‘o’);
delayMs(500);
} }
```

Fundamentals – Parallel GPIO Programming

Program (7): Init/Configuring the LCD

```
void LCD_init(void)
{
SIM->SCGC5 |= 0x1000; /* enable clock to Port D */
PORTD->PCR[0] = 0x100; /* make PTD0 pin as GPIO */
PORTD->PCR[1] = 0x100; /* make PTD1 pin as GPIO */
……

PORTD->PCR[6] = 0x100; /* make PTD6 pin as GPIO */
PORTD->PCR[7] = 0x100; /* make PTD7 pin as GPIO */
PTD->PDDR = 0xFF; /* make PTD7-0 as output pins */
SIM->SCGC5 |= 0x0200; /* enable clock to Port A */
PORTA->PCR[2] = 0x100; /* make PTA2 pin as GPIO */
PORTA->PCR[4] = 0x100; /* make PTA4 pin as GPIO */
PORTA->PCR[5] = 0x100; /* make PTA5 pin as GPIO */
PTA->PDDR |= 0x34; /* make PTA5, 4, 2 as out pins*/
```

Fundamentals – Parallel GPIO Programming

Program(7): Init/Configuring the LCD

```
delayMs(30); /* initialization sequence */

LCD_command(0x30);
delayMs(10);
LCD_command(0x30);
delayMs(1);
LCD_command(0x30);
/* set 8-bit data, 2-line, 5x7 font */
LCD_command(0x38);
/* move cursor right */
LCD_command(0x06);
/* clear screen, move cursor to home */
LCD_command(0x01);
/* turn on display, cursor blinking */
LCD_command(0x0F);}
```

Program(7): Sending data to the LCD

```
void LCD_command(unsigned char command)
{
PTA->PCOR = RS | RW; /* RS = 0, R/W = 0 */
PTD->PDOR = command;
PTA->PSOR = EN; /* pulse E */

delayMs(0);
PTA->PCOR = EN;

if (command < 4)
delayMs(4); /* command 1 and 2 needs up to 1.64ms */
else
delayMs(1); /* all others 40 us */
}
```

Fundamentals – Parallel GPIO Programming

Program(7): Sending data to the LCD

```
void LCD_data(unsigned char data)
{
PTA->PSOR = RS; /* RS = 1, R/W = 0 */

PTA->PCOR = RW;

PTD->PDOR = data;

PTA->PSOR = EN; /* pulse E */

delayMs(0);
PTA->PCOR = EN;
delayMs(1);
}
```

Fundamentals – Parallel GPIO Programming

Checking the LCD busy flag

The above programs used a time delay before issuing the next data or command.

This allows the LCD a sufficient amount of time to get ready to accept the next data.

However, the LCD has a busy flag. We can monitor the busy flag and issue data when it is ready. This will speed up the process.

To check the busy flag, we must read the command register (R/W = 1, RS = 0).

Checking the LCD busy flag

The busy flag is the D7 bit of that register.

Therefore, if R/W = 1, RS = 0.

When D7 = 1 (busy flag = 1), the LCD is busy taking care of internal operations and will not accept any new information.

When D7 = 0, the LCD is ready to receive new information.

Doing so requires switching the direction of the port connected to the data bus to input mode when polling the status register then switch the port direction back to output mode to send the next command.

Program (8): Checking the LCD busy flag

```
/* Initialize and display “hello” on the LCD using
8-bit data mode.

* Data pins use Port D, control pins use Port A.

* Polling of the busy bit of the LCD status bit is
used for timing. */

#include <MKL25Z4.H>

#define RS 0x04 /* PTA2 mask */
#define RW 0x10 /* PTA4 mask */
#define EN 0x20 /* PTA5 mask */
```

Fundamentals – Parallel GPIO Programming

Program (8): Main program “Hello World”

```
int main(void)
{
LCD_init();
for(;;) {
LCD_command(1);     /* clear display */
delayMs(500);
LCD_command(0xC0); /* set cursor at 2nd line */
LCD_data(‘h’);      /* write the word on LCD */
LCD_data(‘e’);
LCD_data(‘l’);
LCD_data(‘l’);
LCD_data(‘o’);
delayMs(500);
}}
```

Program(7): Init/Configuring the LCD

```
void LCD_init(void){

SIM->SCGC5 |= 0x1000; /* enable clock to Port D */
PORTD->PCR[0] = 0x100;/* make PTD0 pin as GPIO */
PORTD->PCR[1] = 0x100;/* make PTD1 pin as GPIO */

…… /* more signals are defined */

PORTD->PCR[7] = 0x100;/* make PTD7 pin as GPIO */
PTD->PDDR = 0xFF;  /* make PTD7-0 as output pins */
SIM->SCGC5 |= 0x0200; /* enable clock to Port A */
PORTA->PCR[2] = 0x100;/* make PTA2 pin as GPIO */
PORTA->PCR[4] = 0x100;/* make PTA4 pin as GPIO */
PORTA->PCR[5] = 0x100;/* make PTA5 pin as GPIO */
PTA->PDDR |= 0x34; /*make PTA5, 4, 2 as output pins*/
```

Program(7): Init/Configuring the LCD

```
delayMs(20); /* initialization sequence */
/* LCD does not respond to status poll */
LCD_command_noWait(0x30);
delayMs(5);
LCD_command_noWait(0x30);
delayMs(1);
LCD_command_noWait(0x30);
/* set 8-bit data, 2-line, 5x7 font */
LCD_command(0x38);
/* move cursor right */
LCD_command(0x06);
/* clear screen, move cursor to home */
LCD_command(0x01);
/* turn on display, cursor blinking */
LCD_command(0x0F);}
```

Program (8): Checking the LCD busy flag

```
/* This function waits until LCD controller is ready to
accept a new command/data before returns. */
void LCD_ready(void)
{
char status;
PTD->PDDR = 0; /* PortD input */
PTA->PCOR = RS; /* RS = 0 for status */
PTA->PSOR = RW; /* R/W = 1, LCD output */
do { /* stay in the loop until it is not busy */
PTA->PSOR = EN; /* raise E */
delayMs(0);
status = PTD->PDIR; /* read status register */
PTA->PCOR = EN;
delayMs(0); /* clear E */
} while (status & 0x80); /* check busy bit */
```

Fundamentals – Parallel GPIO Programming

Program(7): Sending Commands to the LCD

```
PTA->PCOR = RW; /* R/W = 0, LCD input */
PTD->PDDR = 0xFF; /* PortD output */
}

void LCD_command(unsigned char command)
{
LCD_ready(); /* wait until LCD is ready */
PTA->PCOR = RS | RW; /* RS = 0, R/W = 0 */
PTD->PDOR = command;
PTA->PSOR = EN; /* pulse E */
delayMs(0);
PTA->PCOR = EN;
}
```

Program (8): Sending Data to the LCD

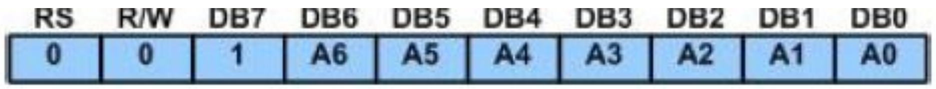
```
void LCD_command_noWait(unsigned char command){
PTA->PCOR = RS | RW; /* RS = 0, R/W = 0 */
PTD->PDOR = command;
PTA->PSOR = EN; /* pulse E */
delayMs(0);
PTA->PCOR = EN; }

void LCD_data(unsigned char data)
{ LCD_ready(); /* wait until LCD is ready */
PTA->PSOR = RS; /* RS = 1, R/W = 0 */
PTA->PCOR = RW;
PTD->PDOR = data;
PTA->PSOR = EN; /* pulse E */
delayMs(0);
PTA->PCOR = EN;}
```

Fundamentals – Parallel GPIO Programming

LCD Cursor Position

In the LCD, one can move the cursor to any location in the display by issuing an address command. The next character sent will appear at the cursor position.

For the two-line LCD, the address command for the first location of line 1 is 0x80, and for line 2 it is 0xC0.

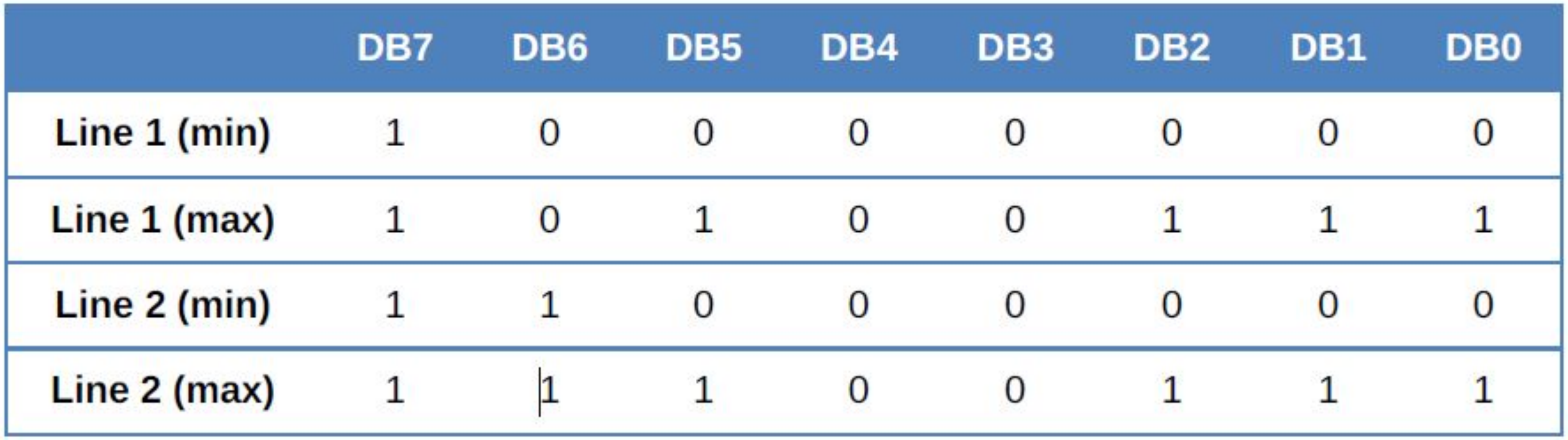


RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	A6	A5	A4	A3	A2	A1	A0

where $A_6A_5A_4A_3A_2A_1A_0$= 0000000 to 0100111 for line 1 and $A_6A_5A_4A_3A_2A_1A_0$ = 1000000 to 1100111 for line 2. See Table 3-3.

Fundamentals – Parallel GPIO Programming

LCD Cursor Position



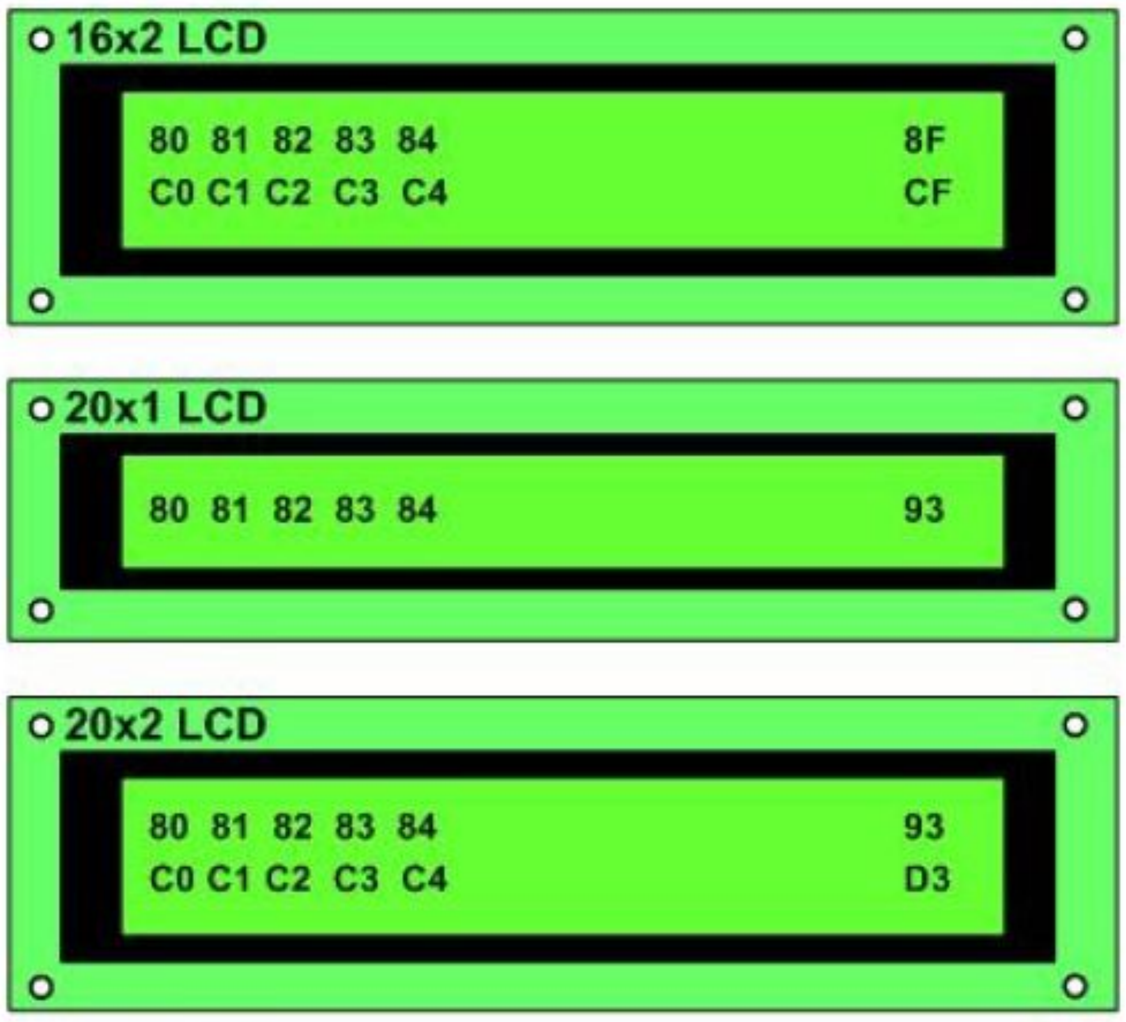
	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Line 1 (min)	1	0	0	0	0	0	0	0
Line 1 (max)	1	0	1	0	0	1	1	1
Line 2 (min)	1	1	0	0	0	0	0	0
Line 2 (max)	1	1	1	0	0	1	1	1

The upper address range can go as high as 0100111 for the 40-character wide LCD

For the 20-character-wide LCD the address of the visible positions goes up to 010011 (19 decimal = 10011 binary).

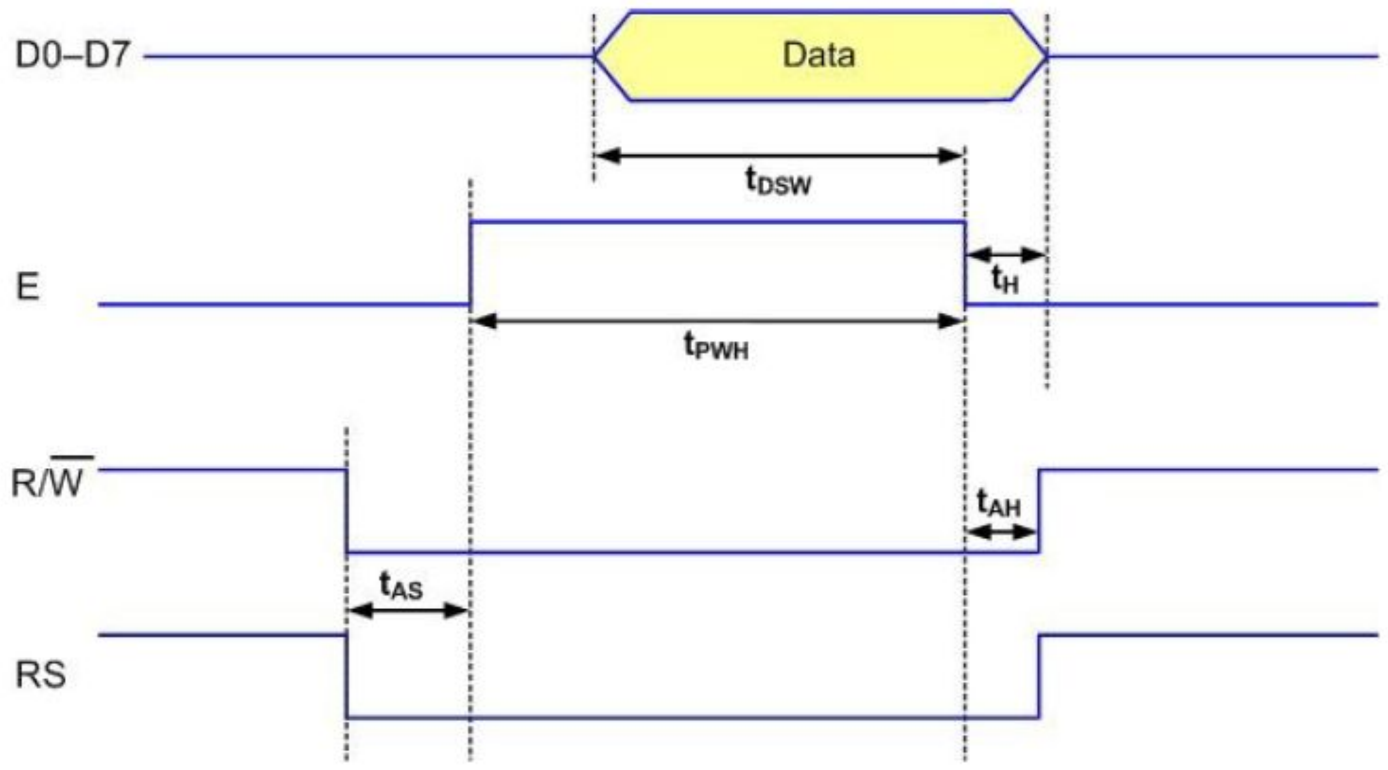
Fundamentals – Parallel GPIO Programming

LCD Cursor Position



Fundamentals – Parallel GPIO Programming

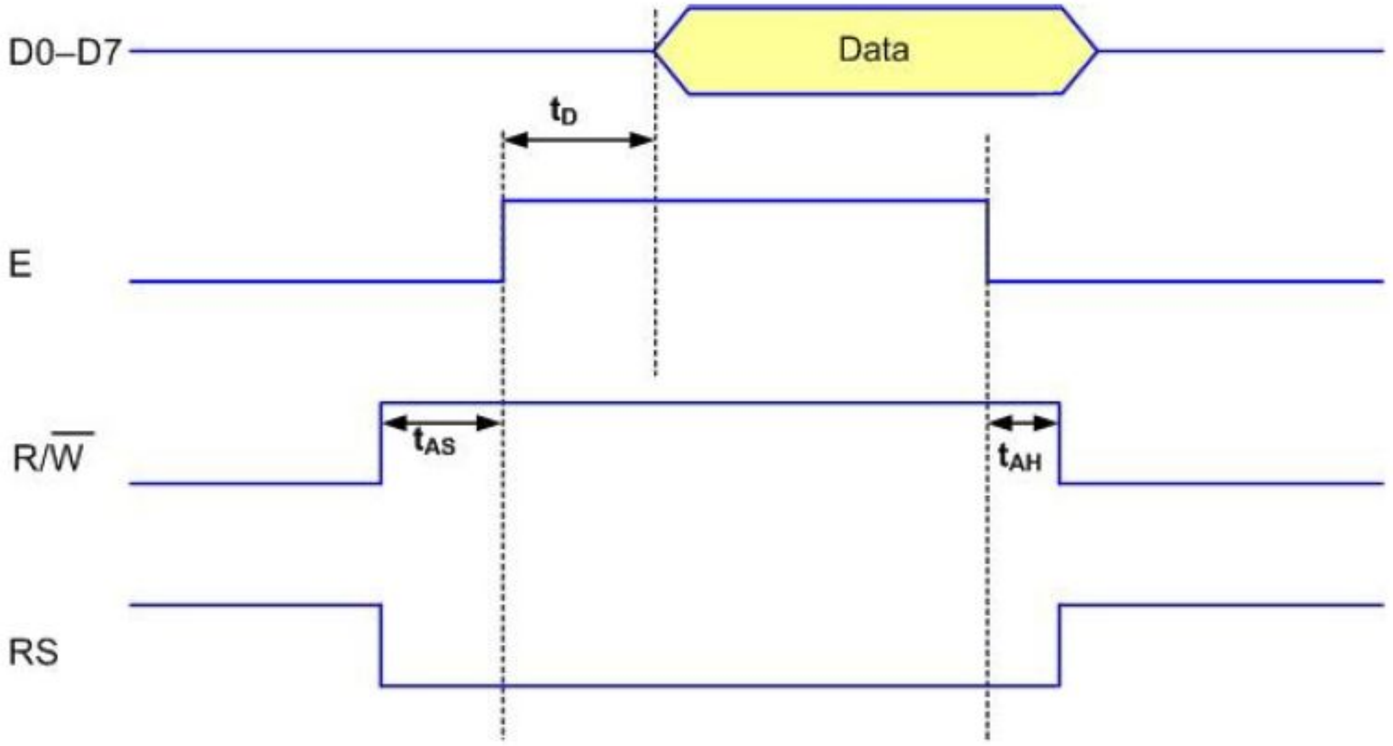
LCD timing and datasheet (writing timing)



t_{PWH} = Enable pulse width = 230 ns (minimum)
t_{DSW} = Data setup time = 80 ns (minimum)
t_H = Data hold time = 10 ns (minimum)
t_{AS} = Setup time prior to E (going high) for both RS and R/W = 40 ns (minimum)
t_{AH} = Hold time after E has come down for both RS and R/W = 10 ns (minimum)

Fundamentals – Parallel GPIO Programming

LCD timing and datasheet (read timing)



t_D = Data output delay time
t_{AS} = Setup time prior to E (going high) for both RS and R/W = 40 ns (minimum)
t_{AH} = Hold time after E has come down for both RS and R/W = 10 ns (minimum)

Note: Read requires an L-to-H pulse for the E pin.

Conclusion – Summary & Takeways

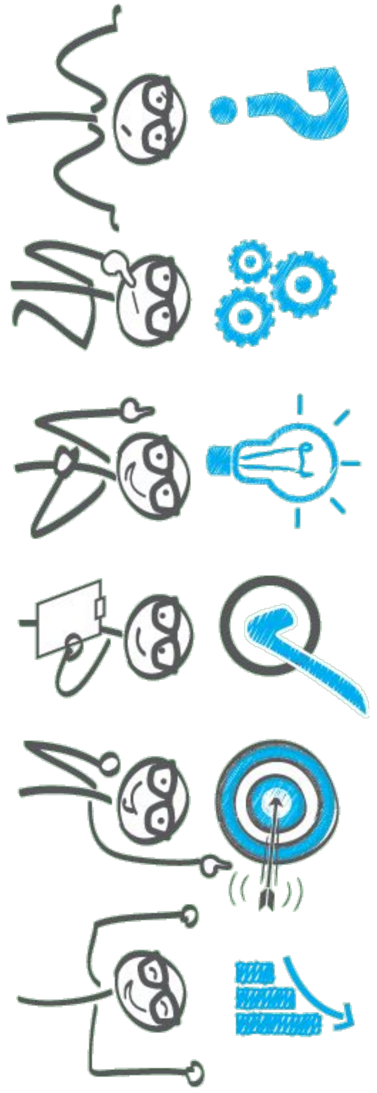
Questions for further reflection

LCDs are virtually everywhere, they represent and effective way of interacting with the user or displaying information about the system

What kind of applications can you think of? When might be better to have a touch screen? In next topic we will cover a middle-ground solution: keyboards…

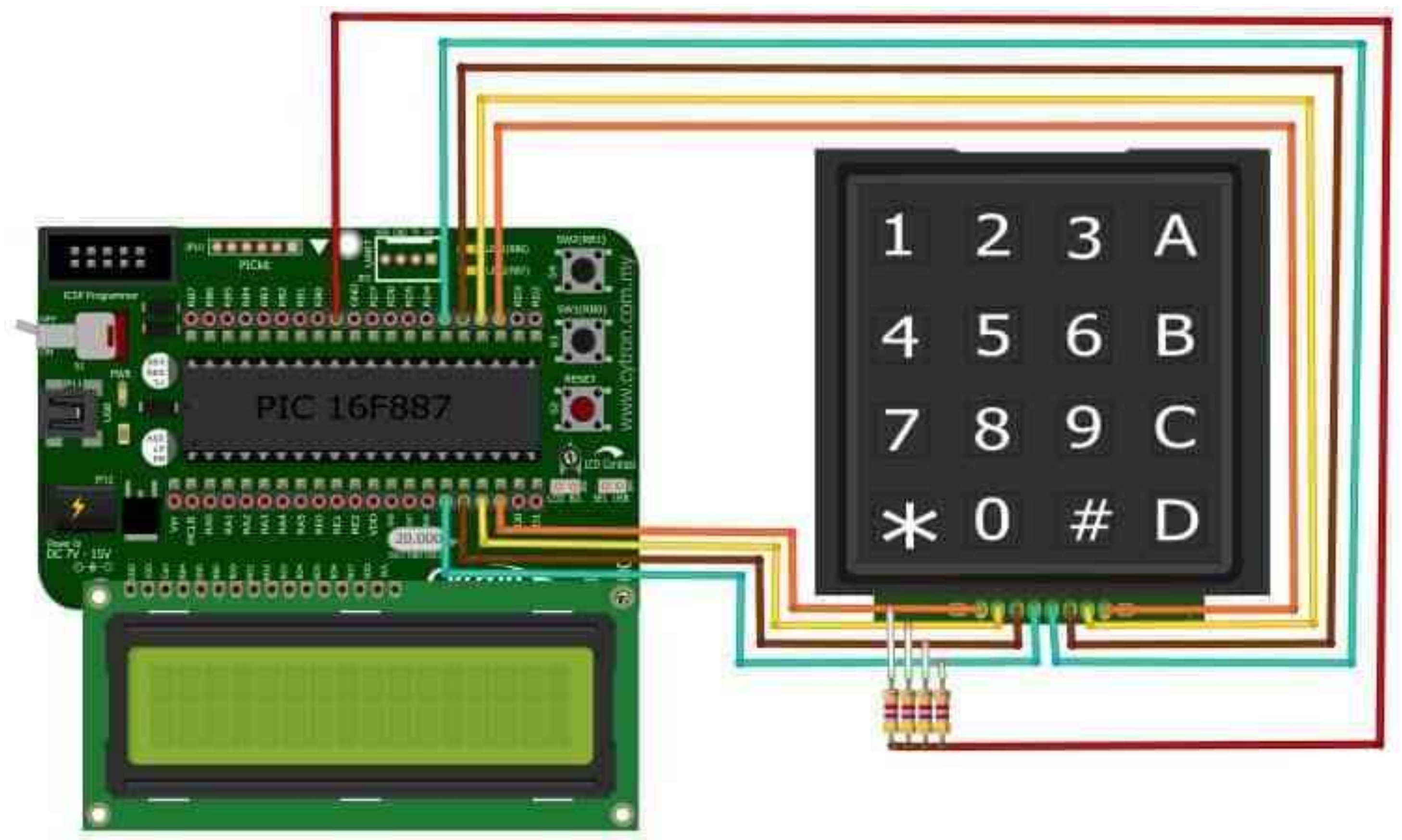
We have seen that there are 8 and 4 bit options for interfaces the LCD, but other options exist

For instance, I2C variants exists, we will cover this topic in a subsequent session



Fundamentals – Parallel GPIO Programming

Interfacing the Keyboard to the CPU



Fundamentals – Parallel GPIO Programming

Interfacing the Keyboard to the CPU

To reduce the microcontroller I/O pin usage, keyboards are organized in a matrix of rows and columns.

The CPU accesses both rows and columns through ports; therefore, with two 8-bit ports, an 8 × 8 matrix of 64 keys can be connected to a microprocessor.

When a key is pressed, a row and a column make a contact; otherwise, there is no connection between rows and columns

In this section, we look at the mechanism by which the microprocessor scans and identifies the key.

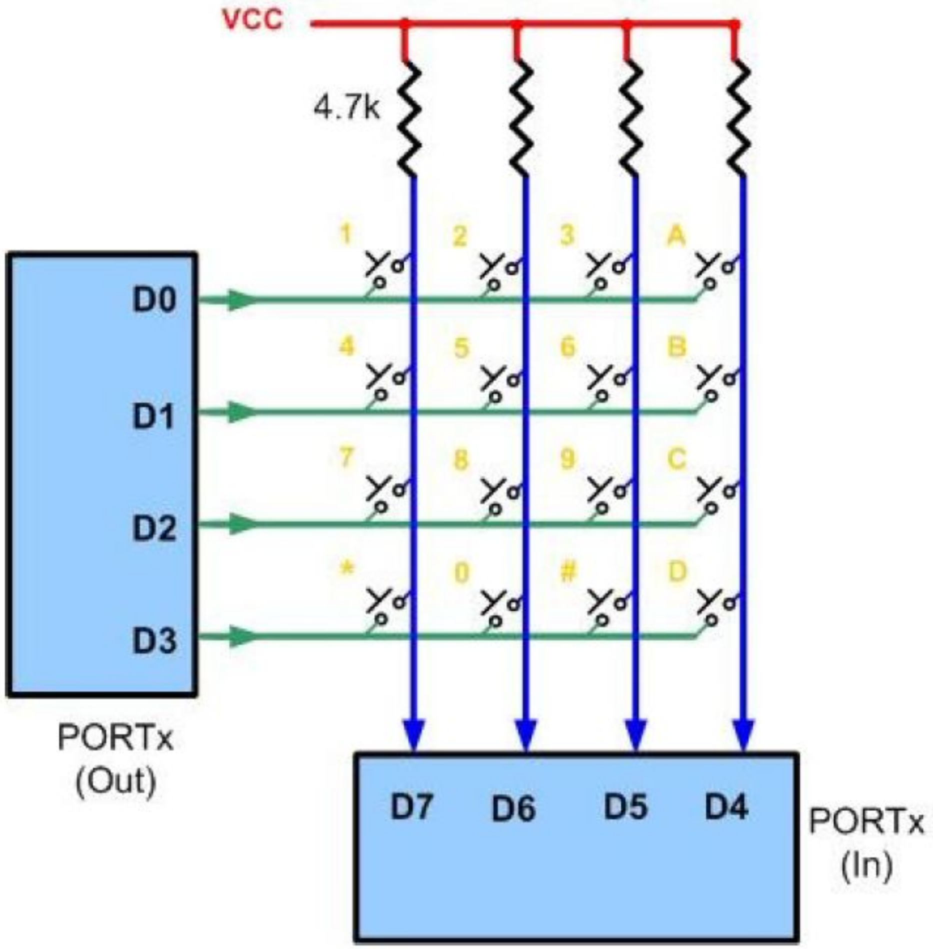
Fundamentals – Parallel GPIO Programming

Scanning and identifying the key

The figure shows a 4 × 4 matrix connected to two ports.

The rows are connected to an output port and the columns are connected to an input port.

All the input pins have pull-up resistor connected.



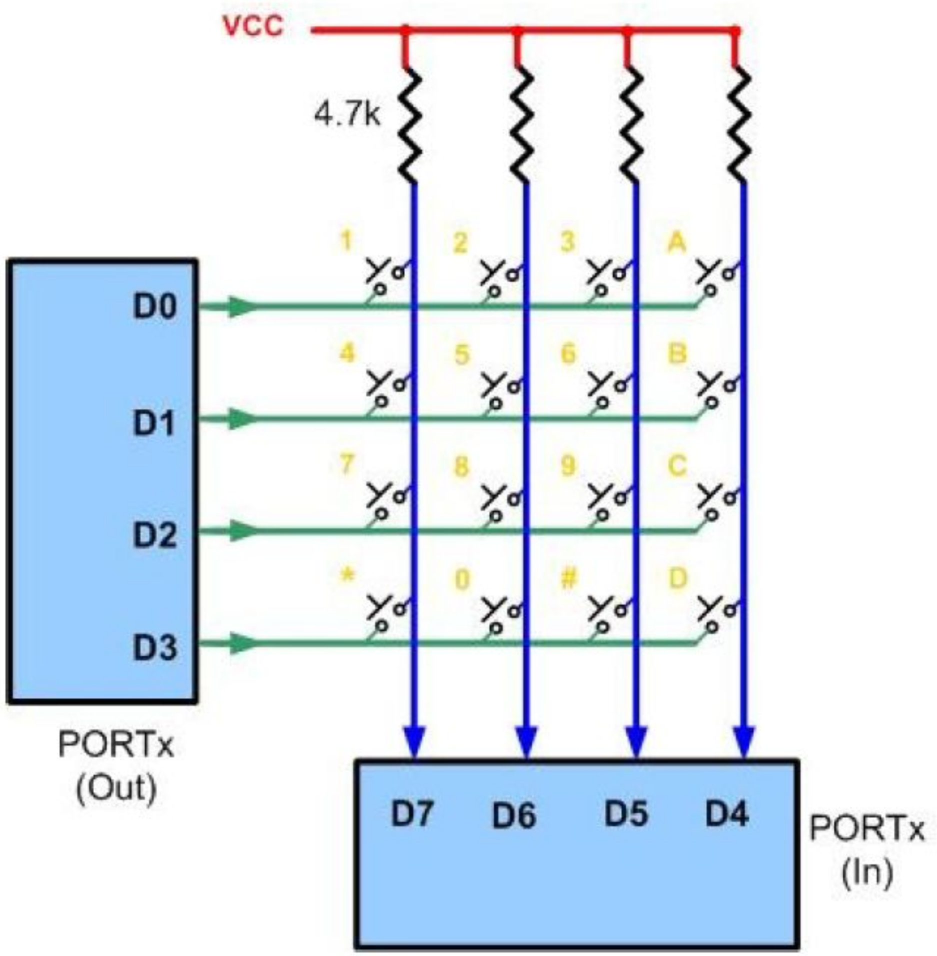
Fundamentals – Parallel GPIO Programming

Scanning and identifying the key

If no key has been pressed, reading the input port will yield 1s for all columns.

If all the rows are driven low and a key is pressed, the column will read back a 0

It is the function of the microprocessor to scan the keyboard continuously to detect and identify the key pressed.



Fundamentals – Parallel GPIO Programming

Key press detection

To detect the key pressed, the microprocessor drives all rows low then it reads the columns.

If the data read from the columns is D7–D4 = 1111, no key has been pressed and the process continues until a key press is detected

However, if one of the column bits has a zero, this means that a key was pressed.

For example,

If D7–D4= 1101, this means that a key in the D5 column has been pressed

Fundamentals – Parallel GPIO Programming

Program (10): Key press detection

```
/* Matrix keypad detect
* This program checks a 4x4 matrix keypad to see 
whether
* a key is pressed or not. When a key is pressed, 
it turns
* on the blue LED.
*
* PortC 7-4 are connected to the columns and PortC 
3-0 are connected
* to the rows.
*/
#include <MKL25Z4.h>
void delayUs(int n);
void keypad_init(void);
char keypad_kbhit(void);
```

Fundamentals – Parallel GPIO Programming

Program (10): Port Initialization

```
int main(void)
{
keypad_init();

SIM->SCGC5 |= 0x1000; /* enable clock to Port D */
PORTD->PCR[1] = 0x100; /* make PTD1 pin as GPIO */
PTD->PDDR |= 0x02; /* make PTD1 as output pin */
while(1)
{ if ( keypad_kbhit() != 0) /* if a key is pressed? */

PTD->PCOR |= 0x02; /* turn on blue LED */
else
PTD->PSOR |= 0x02; /* turn off blue LED */
}}
```

Fundamentals – Parallel GPIO Programming

Program (10): GPIO Keyboard Initialization

```
/* Initializes PortC that is connected to the keypad.
/*  Pins as GPIO input pin with pullup enabled.*/
void keypad_init(void)
{
SIM->SCGC5 |= 0x0800;  /* enable clock to Port C */
PORTC->PCR[0] = 0x103; /* PTD0, GPIO, enable pullup*/
PORTC->PCR[1] = 0x103; /* PTD1, GPIO, enable pullup*/
PORTC->PCR[2] = 0x103; /* PTD2, GPIO, enable pullup*/
PORTC->PCR[3] = 0x103; /* PTD3, GPIO, enable pullup*/
PORTC->PCR[4] = 0x103; /* PTD4, GPIO, enable pullup*/
PORTC->PCR[5] = 0x103; /* PTD5, GPIO, enable pullup*/
PORTC->PCR[6] = 0x103; /* PTD6, GPIO, enable pullup*/
PORTC->PCR[7] = 0x103; /* PTD7, GPIO, enable pullup*/
PTD->PDDR = 0x0F; /* make PTD7-0 as input pins */
}
```

Fundamentals – Parallel GPIO Programming

Program (10): Key Press Detection

```c
/* If a key is pressed, it returns 1. */
* Otherwise, it returns a 0 (not ASCII ‘0’). */
char keypad_kbhit(void)
{
int col;
PTC->PDDR |= 0x0F; /* enable all rows */
PTC->PCOR = 0x0F;
delayUs(2); /* wait for signal return */
col = PTC->PDIR & 0xF0; /* read all columns */
PTC->PDDR = 0; /* disable all rows */
if (col == 0xF0)
return 0; /* no key pressed */
else
return 1; /* a key is pressed */
}
```

Key Identification

After a key press is detected, the microprocessor will go through the process of identifying the key.

Starting from the top row, the microprocessor drives one row low at a time; then it reads the columns.

If the data read is all 1s, no key in that row is pressed and the process is moved to the next row. It drives the next row
low, reads the columns, and checks for any zero.

This process continues until a row is identified with a zero in one of the columns.

Key Identification

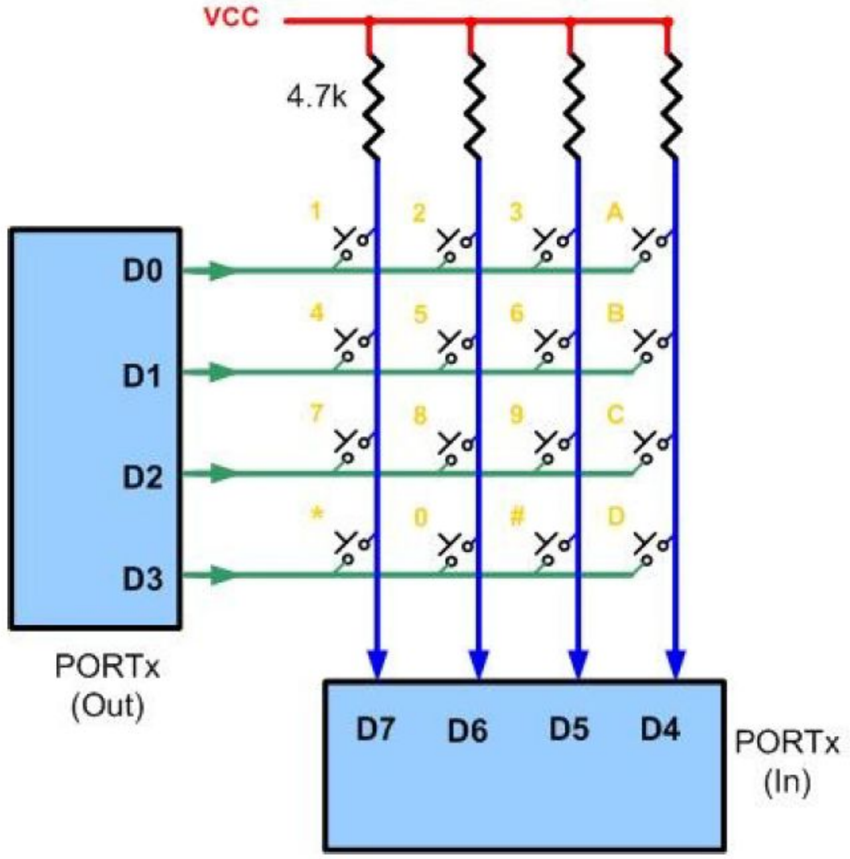
The next task is to find out which column the pressed key belongs to. This should be easy since each column is connected to a separate input pin

Example

identify the row and column of the pressed key for each of the following.

(a) D3–D0 = 1110 for the row, D7–D4= 1011 for the column

(b) D3–D0 = 1101 for the row D7–D4= 0111 for the column



Fundamentals – Parallel GPIO Programming

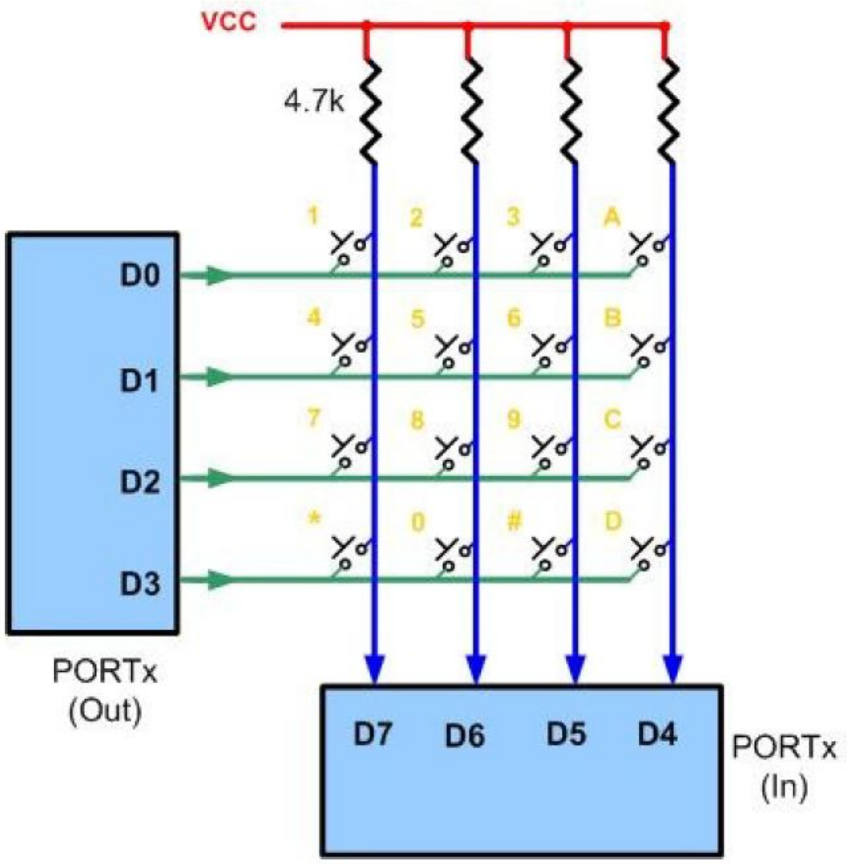
Key Identification

The next task is to find out which column the pressed key belongs to. This should be easy since each column is connected to a separate input pin

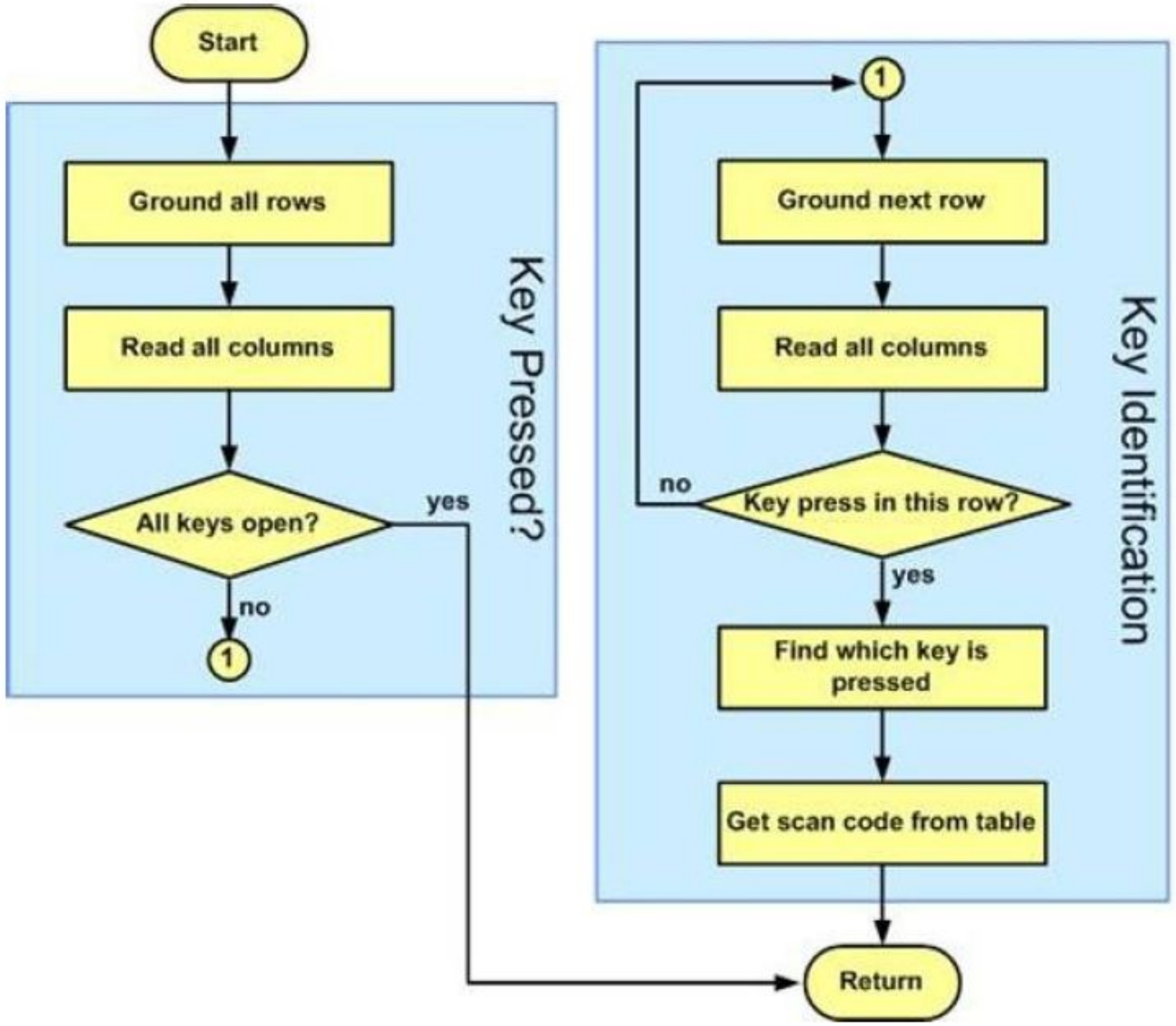
Solution

(a) The row belongs to D0 and the column belongs to D6; therefore, the key number 2 was pressed.

(b) The row belongs to D1 and the column belongs to D7; therefore, the key number 4 was pressed.



Key Press Detection and Identification (1)



Key Press Detection and Identification (2)

The next program provides an implementation of the detection and identification algorithm in C language

First for the initialization of the ports, Port C pins 3-0 are used for rows. The Port C pins 7-4 are used for columns.

They are all configured as input digital pin to prevent accidental short circuit of two output pins.

If output pins are driven high and low and two keys of the same column are pressed at the same time by accident, they will short the output low to output high of the adjacent pins and cause damages to these pins.

Key Press Detection and Identification (3)

To prevent this, all pins are configured as input pin and only one pin is configured as output pin at a time.

Since only one pin is actively driving the row, shorting two rows will not damage the circuit.

The input pins are configured with pull-up enabled so that when the connected keys are not pressed, they stay high and read as 1.

The key scanning function is a non-blocking function, meaning the function returns regardless of whether there is a key pressed or not.

Key Press Detection and Identification (4)

The function first drives all rows low and check to see if any key pressed.

If no key is pressed, a zero is returned. Otherwise the code will proceed to check one row at a time by driving only one row low at a time and read the columns.

If one of the columns is active, it will find out which column it is.

With the combination of the active row and active column, the code will find out the key that is pressed and return a unique numeric code.

Program (11): Functions declarations

```
/* This program scans a 4x4 matrix keypad and
returns a unique number or each key pressed.
The number is displayed on the tri-color
* LEDs using previous code
* PortC 7-4 are connected to the columns and PortC
3-0 are connected to the rows. */

#include <MKL25Z4.h>

void delayMs(int n);
void delayUs(int n);
void keypad_init(void);
char keypad_getkey(void);
void LED_init(void);
void LED_set(int value);
```

Fundamentals – Parallel GPIO Programming

Program (11): Main program

```
int main(void)
{
unsigned char key;

keypad_init();
LED_init();

while(1)
{

key = keypad_getkey();
LED_set(key); /* set LEDs according to the key code */

}
}
```

Fundamentals – Parallel GPIO Programming

Program (11): GPIO Keyboards initializations

```
/* Initializes PortC that is connected to the keypad.
/*  Pins as GPIO input pin with pullup enabled.*/

void keypad_init(void)
{
SIM->SCGC5 |= 0x0800;  /* enable clock to Port C */
PORTC->PCR[0] = 0x103; /* PTD0, GPIO, enable pullup*/
PORTC->PCR[1] = 0x103; /* PTD1, GPIO, enable pullup*/
PORTC->PCR[2] = 0x103; /* PTD2, GPIO, enable pullup*/
PORTC->PCR[3] = 0x103; /* PTD3, GPIO, enable pullup*/
PORTC->PCR[4] = 0x103; /* PTD4, GPIO, enable pullup*/
PORTC->PCR[5] = 0x103; /* PTD5, GPIO, enable pullup*/
PORTC->PCR[6] = 0x103; /* PTD6, GPIO, enable pullup*/
PORTC->PCR[7] = 0x103; /* PTD7, GPIO, enable pullup*/
PTC->PDDR = 0x0F; /* make PTD7-0 as input pins */
}
```

Program (11): Get Key Function Description

```
/* keypad_getkey()
* If a key is pressed, it returns a key code.
Otherwise, a zero is returned.
• The upper nibble of Port C is used as input. Pull-ups
  are enabled when the keys are not pressed
* The lower nibble of Port C is used as output that
drives the keypad rows.
* First all rows are driven low and the input pins are
read. If no key is pressed, it will read as all ones.
Otherwise, some key is pressed.
* If any key is pressed, the program drives one row low
at a time and leave the rest of the rows inactive
(float) then read the input pins.
* Knowing which row is active and which column is
active, the program can decide which key is pressed. */
```

Fundamentals – Parallel GPIO Programming

Program (11): Identify Key – scanning rows i

```
char keypad_getkey(void) {

int row, col;
const char row_select[] = {0x01, 0x02, 0x04, 0x08};
/* one row is active */
/* check to see any key pressed */

PTC->PDDR |= 0x0F; /* enable all rows */
PTC->PCOR = 0x0F;
delayUs(2); /* wait for signal return */
col = PTC-> PDIR & 0xF0; /* read all columns */
PTC->PDDR = 0; /* disable all rows */
if (col == 0xF0)
return 0; /* no key pressed */
```

Fundamentals – Parallel GPIO Programming

Program (11): Identify Key – scanning rows ii

```
/* If a key is pressed, we need find out which key.*/
for (row = 0; row < 4; row++)
{ PTC->PDDR = 0; /* disable all rows */

PTC->PDDR |= row_select[row]; /* enable one row */
PTC->PCOR = row_select[row]; /* drive active row low*/

delayUs(2); /* wait for signal to settle */
col = PTC->PDIR & 0xF0; /* read all columns */

if (col != 0xF0) break;
/* if one of the input is low, some key is pressed. */
}
```

Fundamentals – Parallel GPIO Programming

Program (11): Identify Key – scanning columns

```
PTC->PDDR = 0; /* disable all rows */

if (row == 4)
return 0; /* if we get here, no key is pressed */

/* gets here when one of the rows has key pressed*/
/*check which column it is*/

if (col == 0xE0) return row*4+ 1; /* key in column 0 */
if (col == 0xD0) return row*4+ 2; /* key in column 1 */
if (col == 0xB0) return row*4+ 3; /* key in column 2 */
if (col == 0x70) return row*4+ 4; /* key in column 3 */
return 0; /* just to be safe */
}
```

Fundamentals – Parallel GPIO Programming

Program (11): Init leds for displaying info

```
/* initialize all three LEDs on the FRDM board */
void LED_init(void)
{
SIM->SCGC5 |= 0x400; /* enable clock to Port B */
SIM->SCGC5 |= 0x1000; /* enable clock to Port D */
PORTB->PCR[18] = 0x100; /* make PTB18 pin as GPIO */
PTB->PDDR |= 0x40000; /* make PTB18 as output pin */
PTB->PSOR |= 0x40000; /* turn off red LED */
PORTB->PCR[19] = 0x100; /* make PTB19 pin as GPIO */
PTB->PDDR |= 0x80000; /* make PTB19 as output pin */
PTB->PSOR |= 0x80000; /* turn off green LED */
PORTD->PCR[1] = 0x100; /* make PTD1 pin as GPIO */
PTD->PDDR |= 0x02; /* make PTD1 as output pin */
PTD->PSOR |= 0x02; /* turn off blue LED */
}
```

Fundamentals – Parallel GPIO Programming

Program (11): Displaying returned value

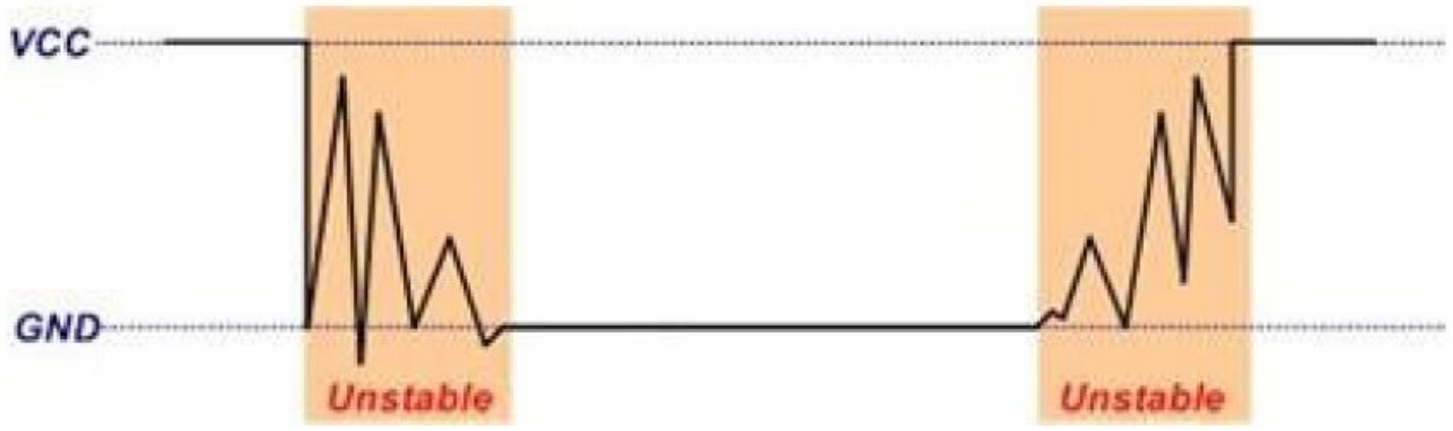
```
/* turn on or off the LEDs wrt to bit 2-0 of the value */
void LED_set(int value)
{
/* use bit 0 of value to control red LED */
if (value & 1)
PTB->PCOR = 0x40000; else /* turn on red LED */
PTB->PSOR = 0x40000; /* turn off red LED */
/* use bit 1 of value to control green LED */
if (value & 2)
PTB->PCOR = 0x80000; else /* turn on green LED */
PTB->PSOR = 0x80000; /* turn off green LED */
/* use bit 2 of value to control blue LED */
if (value & 4)
PTD->PCOR = 0x02; else /* turn on blue LED */
PTD->PSOR = 0x02; /* turn off blue LED */
}
```

Fundamentals – Parallel GPIO Programming

Contact Bounce and Debounce

When a mechanical switch is closed or opened, the contacts do not make a clean transition instantaneously, rather the contacts open and close several times before they settle.

This event is called contact bounce



Fundamentals – Parallel GPIO Programming

Contact Bounce and Debounce

So it is possible when the program first detects a switch in the keypad is pressed but when interrogating which key is pressed, it would find no key pressed.

This is the reason we have a return 0 after checking all the rows. Another problem manifested by contact bounce is that one key press may be recognized as multiple key presses by the program.

Contact bounce also occurs when the switch is released. Because the switch contacts open and close several times before they settle, the program may detect a key press when the key is released.

Contact Bounce and Debounce

For many applications, it is important that each key press is only recognized as one action.

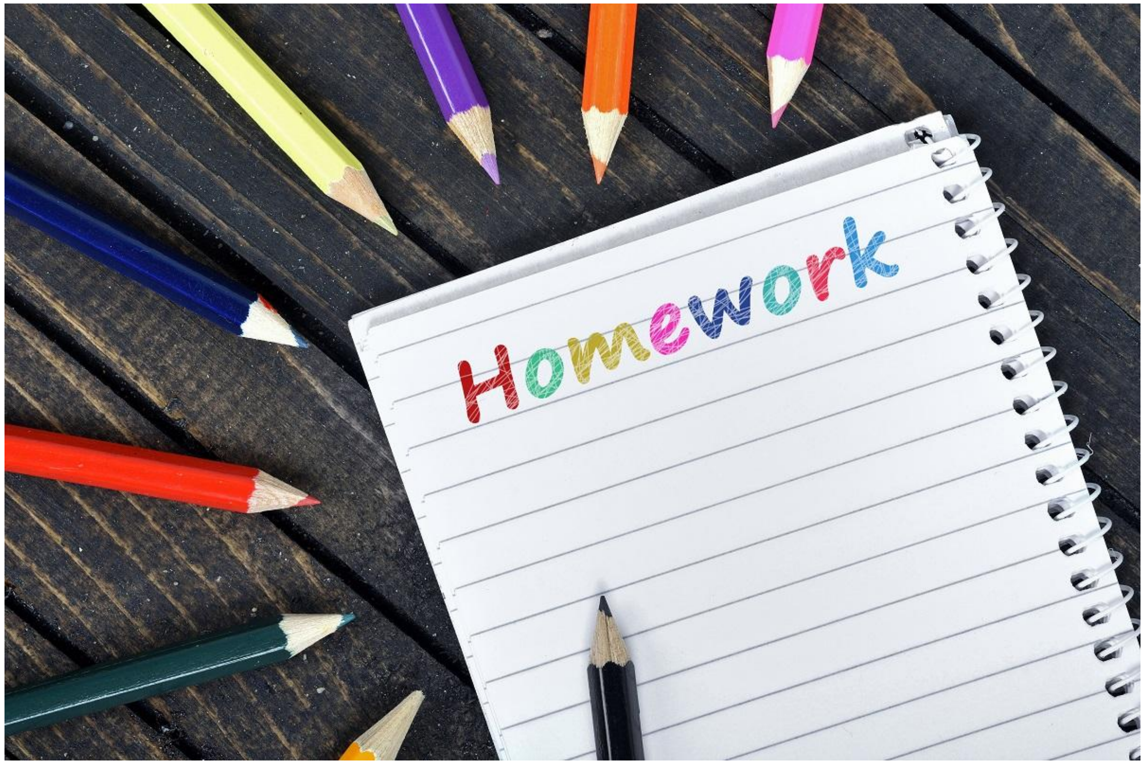
When you press a numeral key of a calculator, you expect to get only one digit. A contact bounce results in multiple digits entered with a single key press.

A simple software solution is that when a transition of the contact state change is detected such as a key pressed or a key released, the software does a delay for about 10 – 20 ms to wait out the contact bounce.

After the delay, the contacts should be settled and stable.

Fundamentals – Parallel GPIO Programming

Contact Bounce and Debounce



Conclusion – Summary & Takeways

Questions for further reflection

Keyboards or touch screes are everywhere, what advantages can have a keyboard over a screen?

In this session we saw only numerical keyboards, but in fact any kind of things or controls could be added, reflect about some possible applications…

Apart from introducing numerical or alphanumerical information intro the application, what other uses do you think keyboards can have in an application

Do you think polling a keyboard is an effective implementation in an embedded system?

Next topic: interrupts!!!