Module 2 - General Purpose I/O

Overview

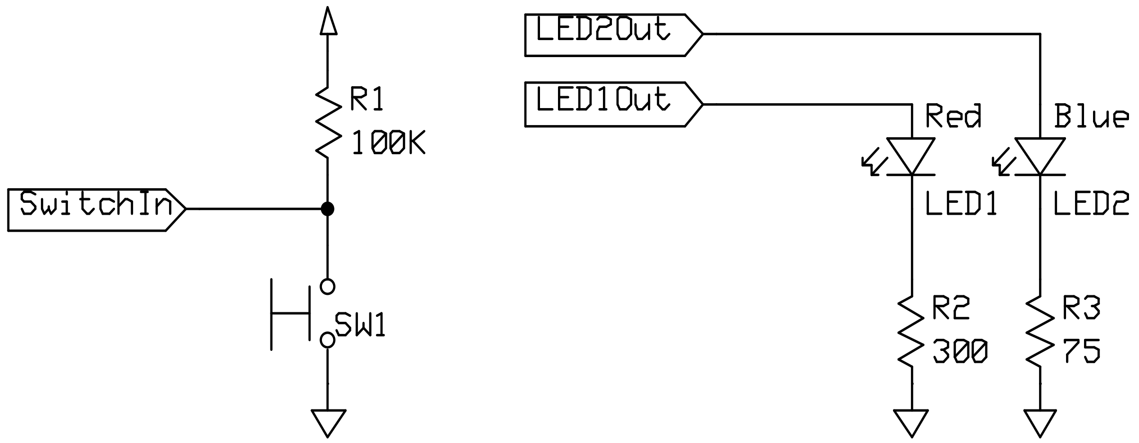
- How do we make a program light up LEDs in response to a switch?

- GPIO
  - Basic Concepts
  - Port Circuitry
  - Control Registers
  - Accessing Hardware Registers in C
  - Clocking and Muxing

- Circuit Interfacing
  - Inputs
  - Outputs

- Additional Port Configuration

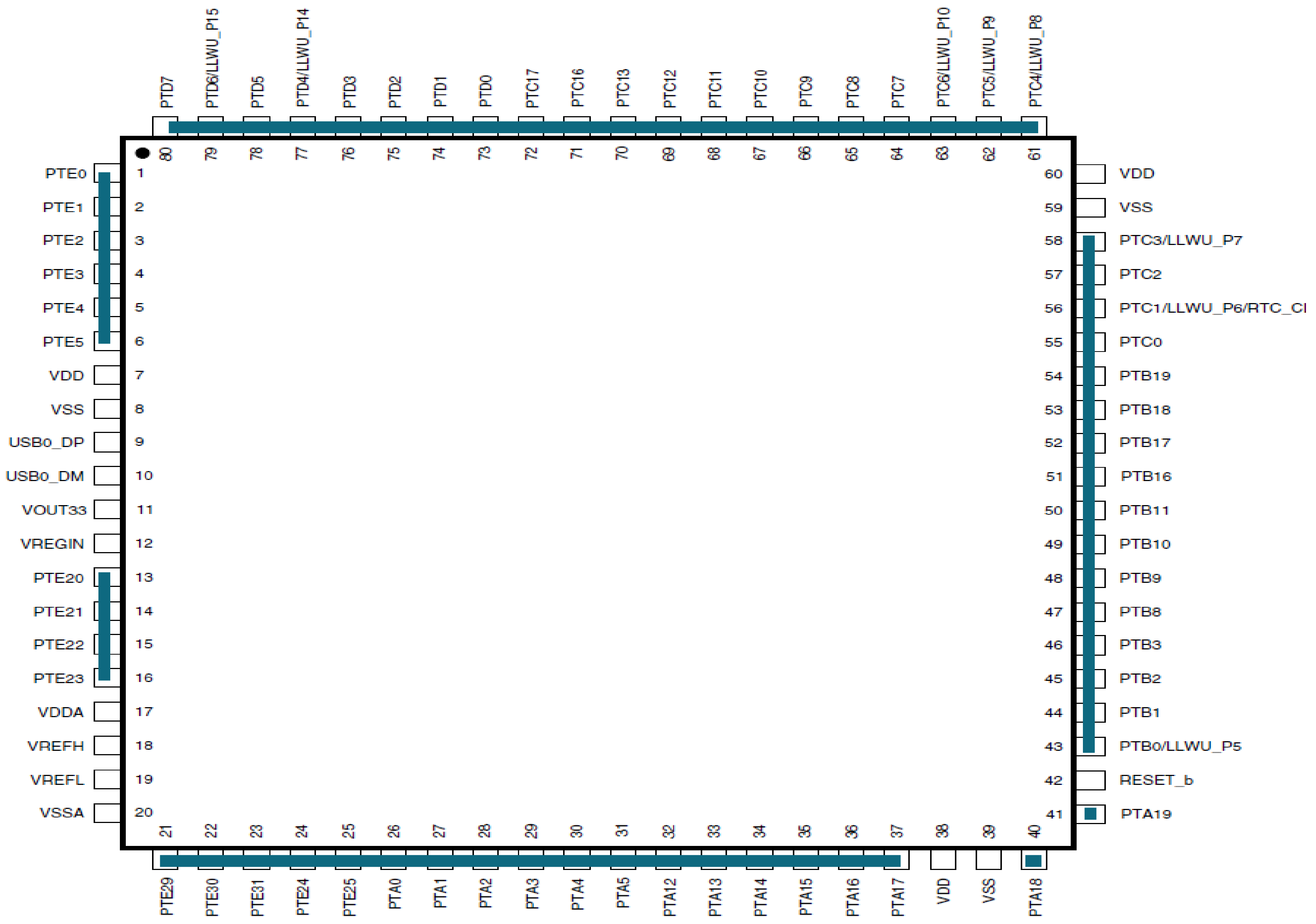
Basic Concepts



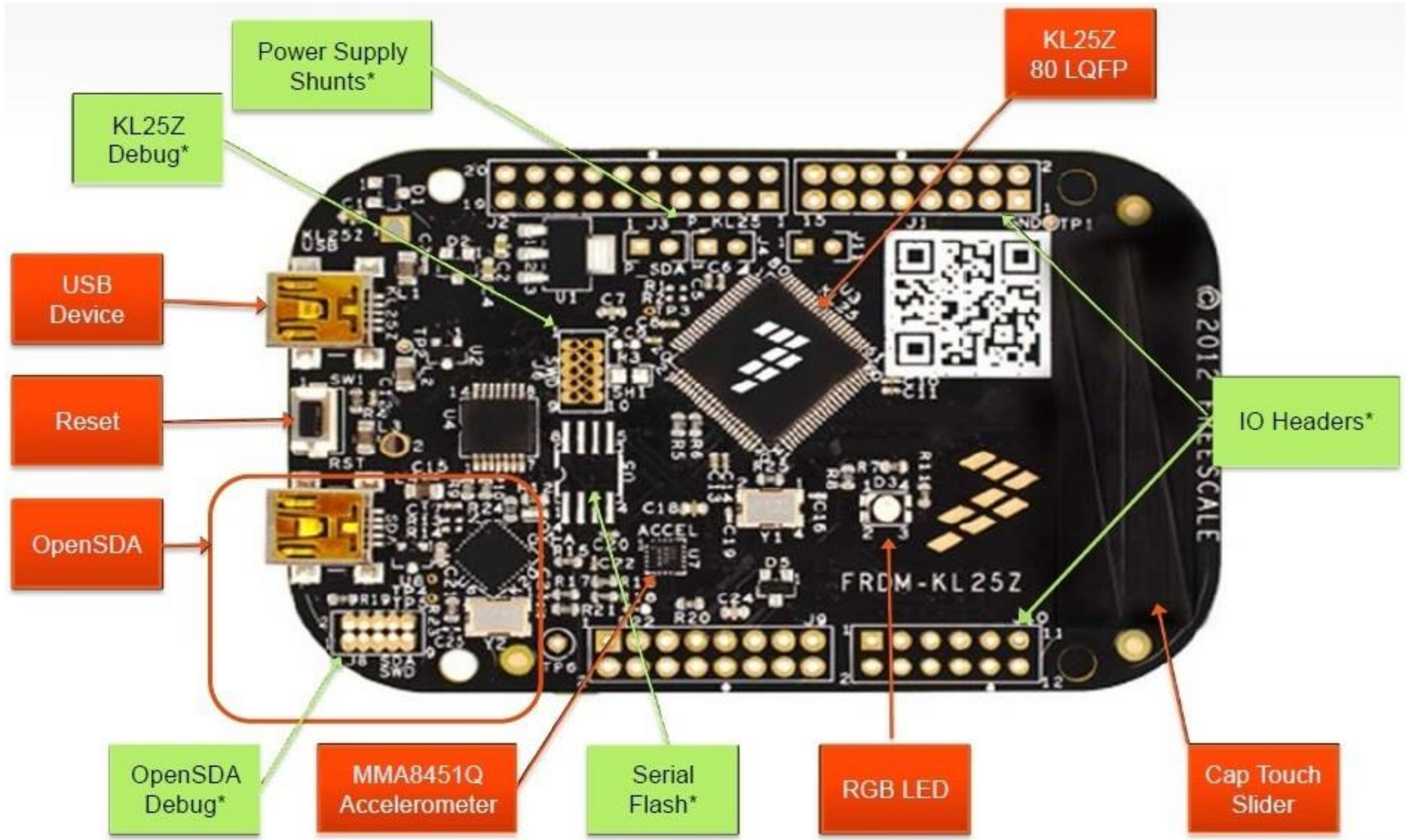
- Goal: light either LED1 or LED2 based on switch SW1 position
- GPIO = General-purpose input and output (digital)
  - Input: program can determine if input signal is a 1 or a 0
  - Output: program can set output to 1 or 0
- Can use this to interface with external devices
  - Input: switch
  - Output: LEDs

KL25Z GPIO Ports

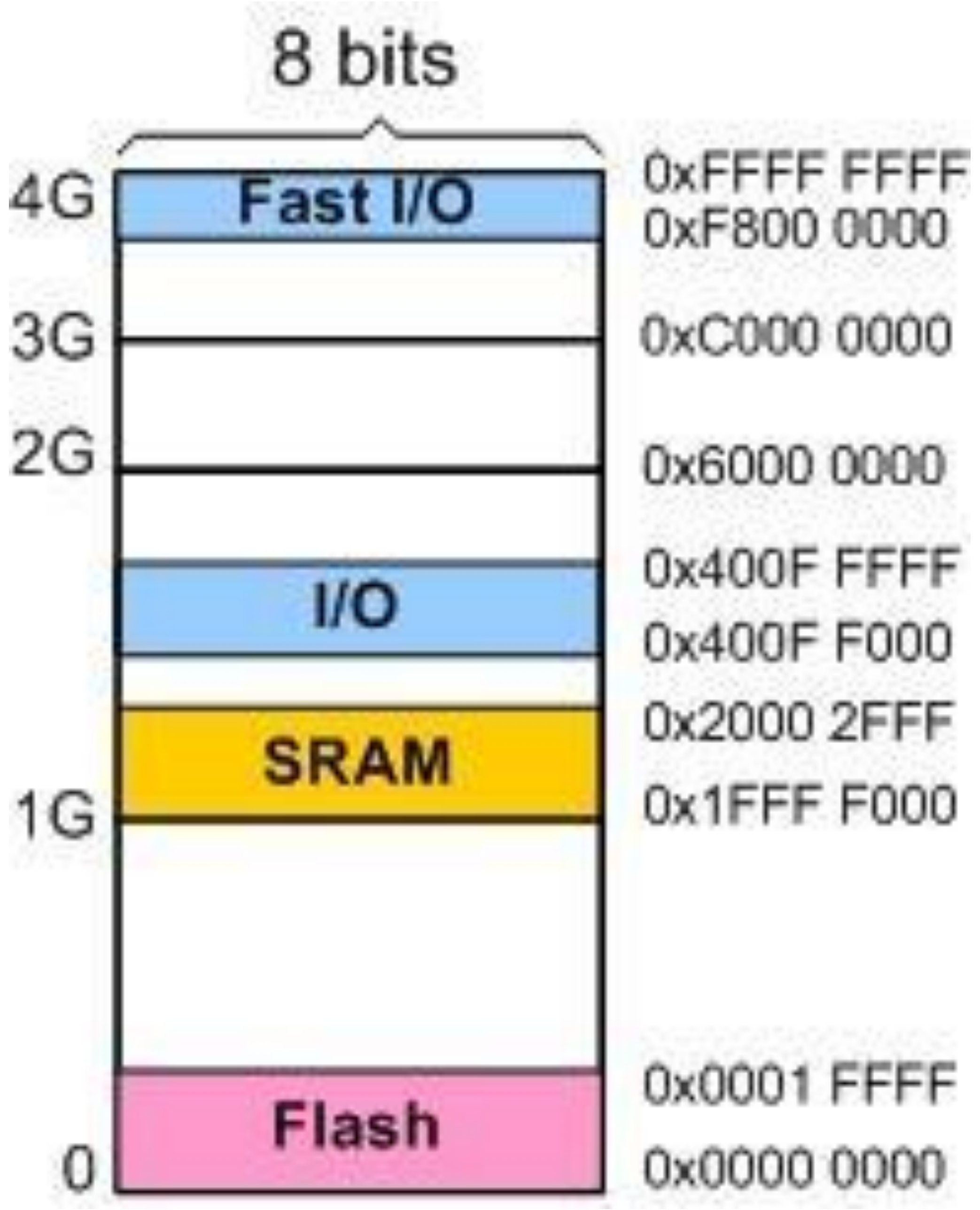
- Port A (PTA) through Port E (PTE)
- Not all port bits are available
- Quantity depends on package pin count

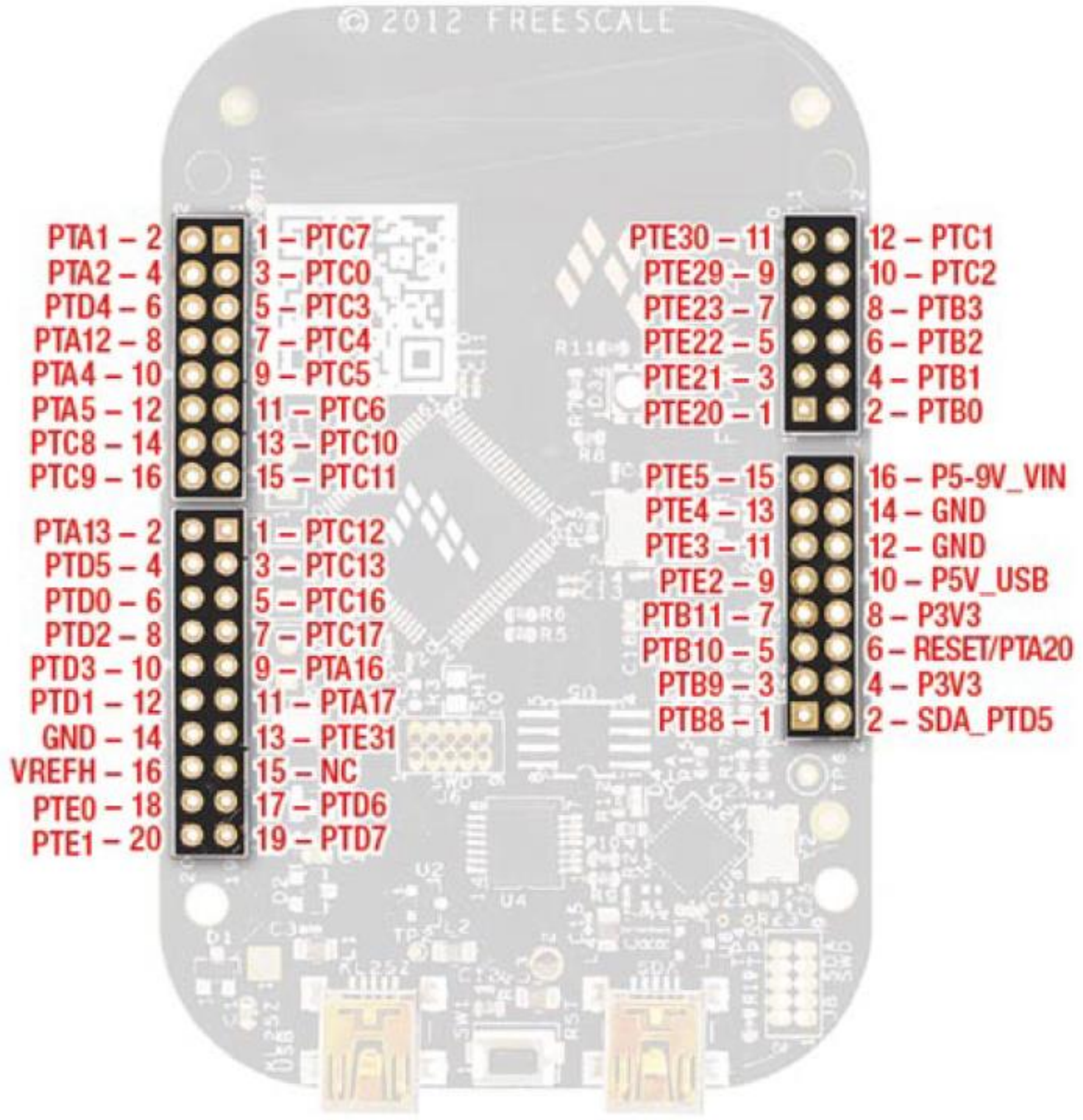


Freedom KL25Z

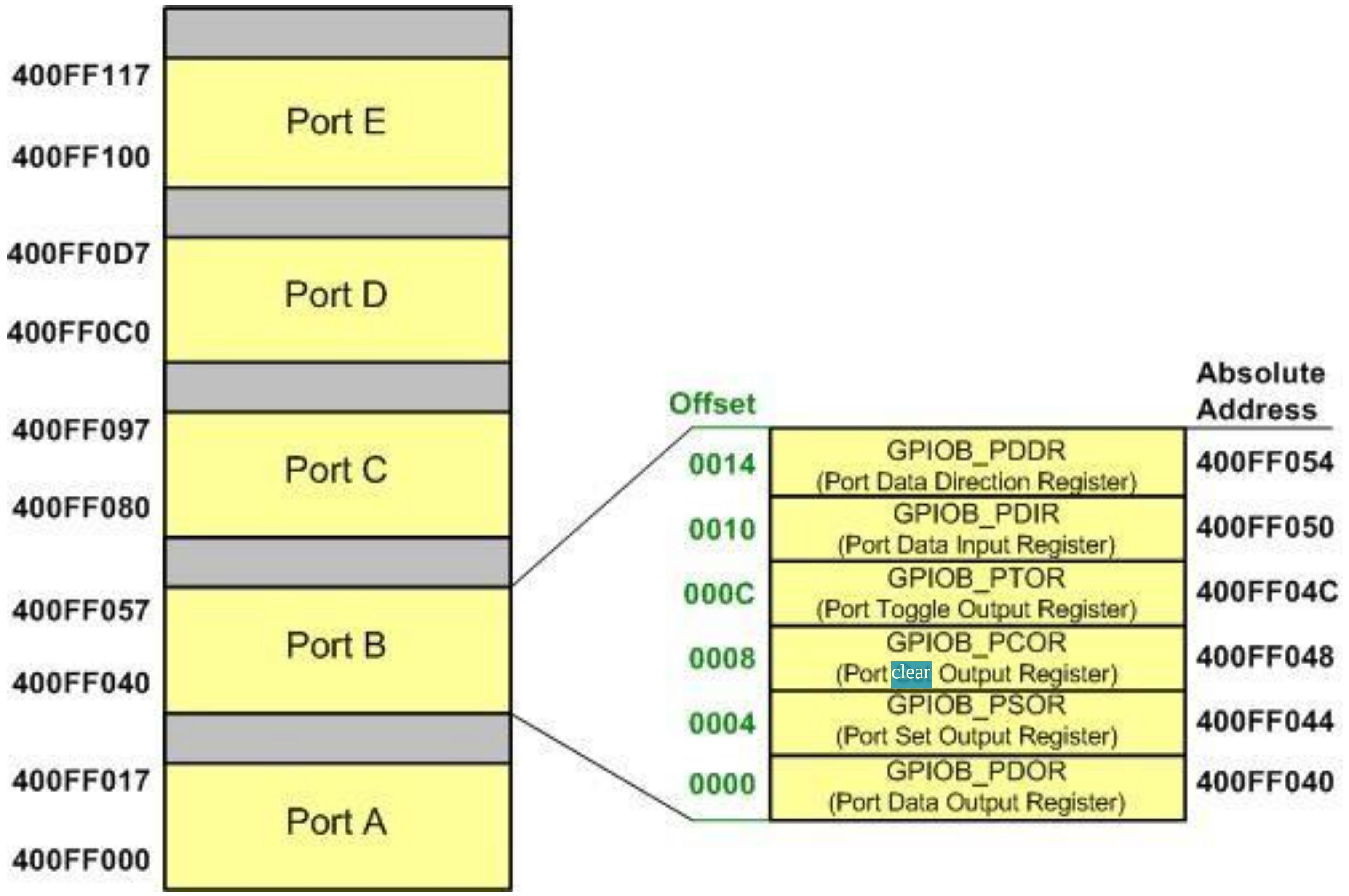


Memory Map

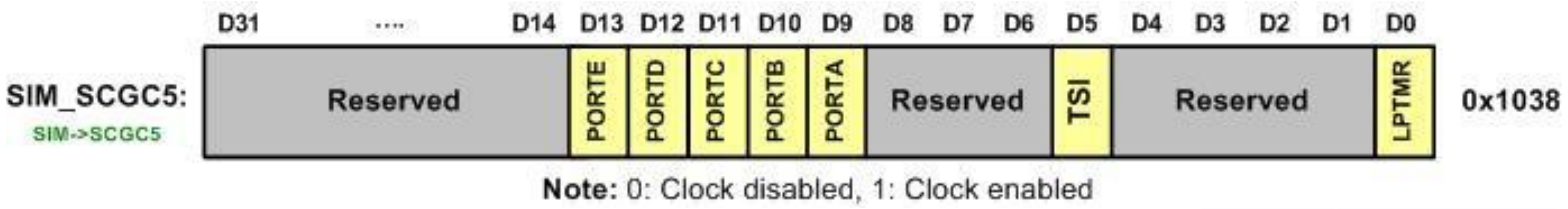

8 bits
4G
3G
2G
1G
0
Fast I/O
I/O
SRAM
Flash
0xFFFF FFFF
0xF800 0000
0xC000 0000
0x6000 0000
0x400F FFFF
0x400F F000
0x2000 2FFF
0x1FFF F000
0x0001 FFFF
0x0000 0000


© 2012 FREESCALE
PTA1 – 2
1 – PTC7
PTA2 – 4
3 – PTC0
PTD4 – 6
5 – PTC3
PTA12 – 8
7 – PTC4
PTA4 – 10
9 – PTC5
PTA5 – 12
11 – PTC6
PTC8 – 14
13 – PTC10
PTC9 – 16
15 – PTC11
PTA13 – 2
1 – PTC12
PTD5 – 4
3 – PTC13
PTD0 – 6
5 – PTC16
PTD2 – 8
7 – PTC17
PTD3 – 10
9 – PTA16
PTD1 – 12
11 – PTA17
GND – 14
13 – PTE31
VREFH – 16
15 – NC
PTE0 – 18
17 – PTD6
PTE1 – 20
19 – PTD7
PTE30 – 11
12 – PTC1
PTE29 – 9
10 – PTC2
PTE23 – 7
8 – PTB3
PTE22 – 5
6 – PTB2
PTE21 – 3
4 – PTB1
PTE20 – 1
2 – PTB0
PTE5 – 15
16 – P5-9V_VIN
PTE4 – 13
14 – GND
PTE3 – 11
12 – GND
PTE2 – 9
10 – P5V_USB
PTB11 – 7
8 – P3V3
PTB10 – 5
6 – RESET/PTA20
PTB9 – 3
4 – P3V3
PTB8 – 1
2 – SDA_PTD5

Ports


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

Clocking Logic



Bit	Port
13	PORTE
12	PORTD
11	PORTC
10	PORTB
9	PORTA

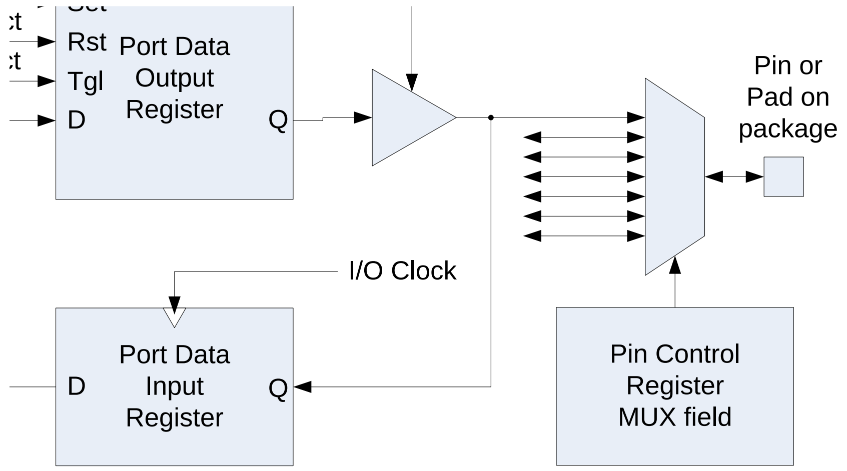
- Need to enable clock to GPIO module
- By default, GPIO modules are disabled to save power
- Writing to an unclocked module triggers a hardware fault!
- Control register SIM_SCGC5 gates clocks to GPIO ports
- Enable clock to Port A

```
SIM->SCGC5 |= (1UL <<  9);
```

- Header file MKL25Z4.h has definitions

```
SIM->SCGC5 |= SIM_SCGC5_PORTA_MASK;
```

Connecting a GPIO Signal to a Pin



- Multiplexer used to increase configurability - what pin should be connected with internally?
- Each configurable pin has a Pin Control Register

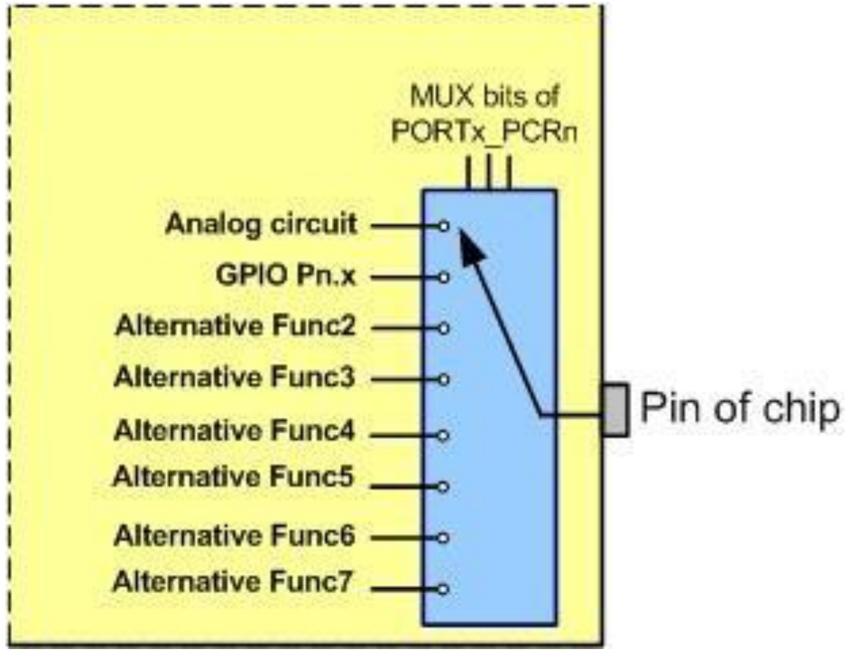
Pin Control Register

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0							ISF	0				IRQC			
W	▨	▨	▨	▨	▨	▨	▨	w1c	▨	▨	▨	▨				
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0					MUX			0	DSE	0	PFE	0	SRE	PE	PS
W	▨	▨	▨	▨	▨				▨		▨		▨			
Reset	0	0	0	0	0	x*	x*	x*	0	x*	0	x*	0	x*	x*	x*

80 LQFP	64 LQFP	48 QFN	32 QFN	Pin Name	Default	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7
64	52	40	28	PTC7	CMP0_IN1	CMP0_IN1	PTC7	SPI0_MISO			SPI0_MOSI		
65	53	—	—	PTC8	CMP0_IN2	CMP0_IN2	PTC8	I2C0_SCL	TPM0_CH4				

- MUX field of PCR defines connections



Structure Declarations

- Like other elements of C programming, the structure must be declared before it can be used. The declaration specifies the tagname of the structure and the names and types of the individual members. The following example has three members: one 8-bit integer and two word pointers

```
struct theport{
unsigned char mask;    // defines which bits are active
unsigned long volatile *addr;  // pointer to its address
unsigned long volatile *ddr;}; // pointer to its direction reg
```

- The above declaration does not create any variables or allocate any space. Therefore to use a structure we must define a global or local variable of this type. The tagname (theport) along with the keyword struct can be used to define variables of this new data type:

```
struct theport PortA,PortB,PortE;
```

- The above line defines the three variables and allocates 9 bytes for each of variable. Because the pointers will be 32-bit aligned the compiler will skip three bytes between mask and addr, so each object will occupy 12 bytes. If you knew you needed just three copies of structures of this type, you could have defined them as

```
struct theport{
unsigned char mask;    // defines which bits are active
unsigned long volatile *addr;
unsigned long volatile *ddr;}PortA,PortB,PortE;
```

Continued

- Definitions like the above are hard to extend, so to improve code reuse we can use typedef to actually create a new data type (called port in the example below) that behaves syntactically like char int short etc.

```
struct theport{
   unsigned char mask;    // defines which bits are active
   unsigned long volatile *addr;  // address
   unsigned long volatile *ddr;}; // direction reg
typedef struct theport port_t;
port_t PortA,PortB,PortE;
```

- Once we have used typedef to create port_t, we don't need access to the name theport anymore. Consequently, some programmers use to following short-cut:

```
typedef struct {
   unsigned char mask;    // defines which bits are active
   unsigned long volatile *addr;        // address
   unsigned long volatile *ddr;}port_t; // direction reg
port_t PortA,PortB,PortE;
```

CMSIS C Support for PCR

- MKL25Z4.h defines PORT_Type structure with a PCR field (array of 32 integers)

```
/** PORT - Register Layout Typedef */
typedef struct {
  __IO uint32_t PCR[32]; /** Pin Control Register n, array offset: 0x0, array step: 0x4 */
  __O  uint32_t GPCLR;        /** Global Pin Control Low Register, offset: 0x80 */
  __O  uint32_t GPCHR;        /** Global Pin Control High Register, offset: 0x84 */
       uint8_t RESERVED_0[24];
  __IO uint32_t ISFR; /** Interrupt Status Flag Register, offset: 0xA0 */
} PORT_Type;
```

CMSIS C Support for PCR

- Header file defines pointers to PORT_Type registers

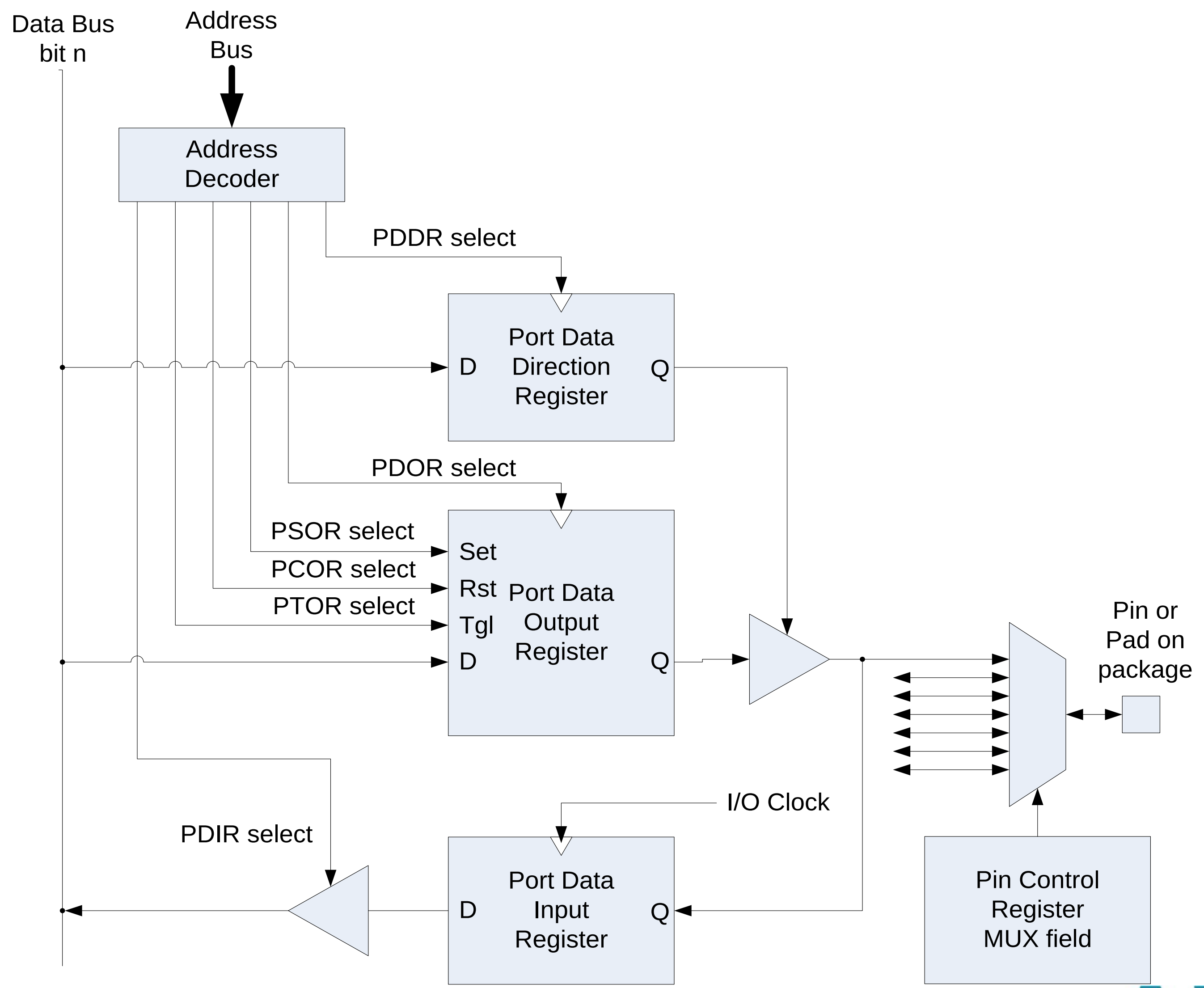
```
/* PORT - Peripheral instance base addresses */
/** Peripheral PORTA base address */
#define PORTA_BASE    (0x40049000u)
/** Peripheral PORTA base pointer */
#define PORTA         ((PORT_Type *)PORTA_BASE)
```

- Also defines macros and constants

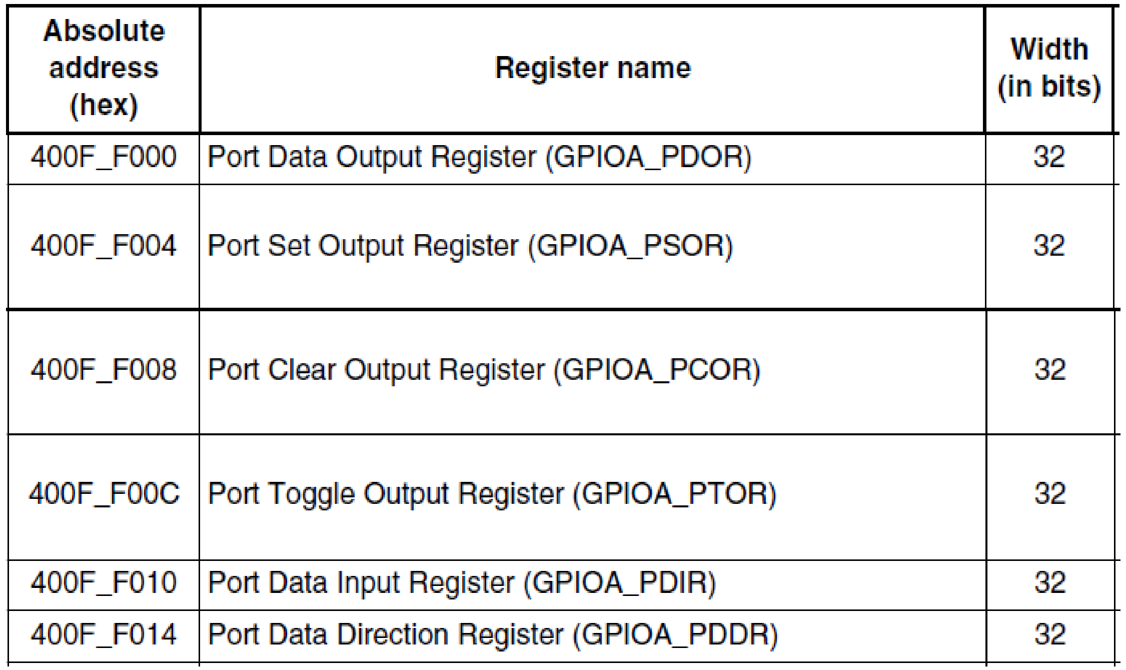
```
#define PORT_PCR_MUX_MASK     0x700u
#define PORT_PCR_MUX_SHIFT    8
#define PORT_PCR_MUX(x)(((uint32_t)(((uint32_t)(x))<<PORT_PCR_MUX_SHIFT))&PORT_PCR_MUX_MASK)
```

GPIO Port Bit Circuitry in MCU

- Control
  - Direction
  - MUX
- Data
  - Output (different ways to access it)
  - Input



Control Registers

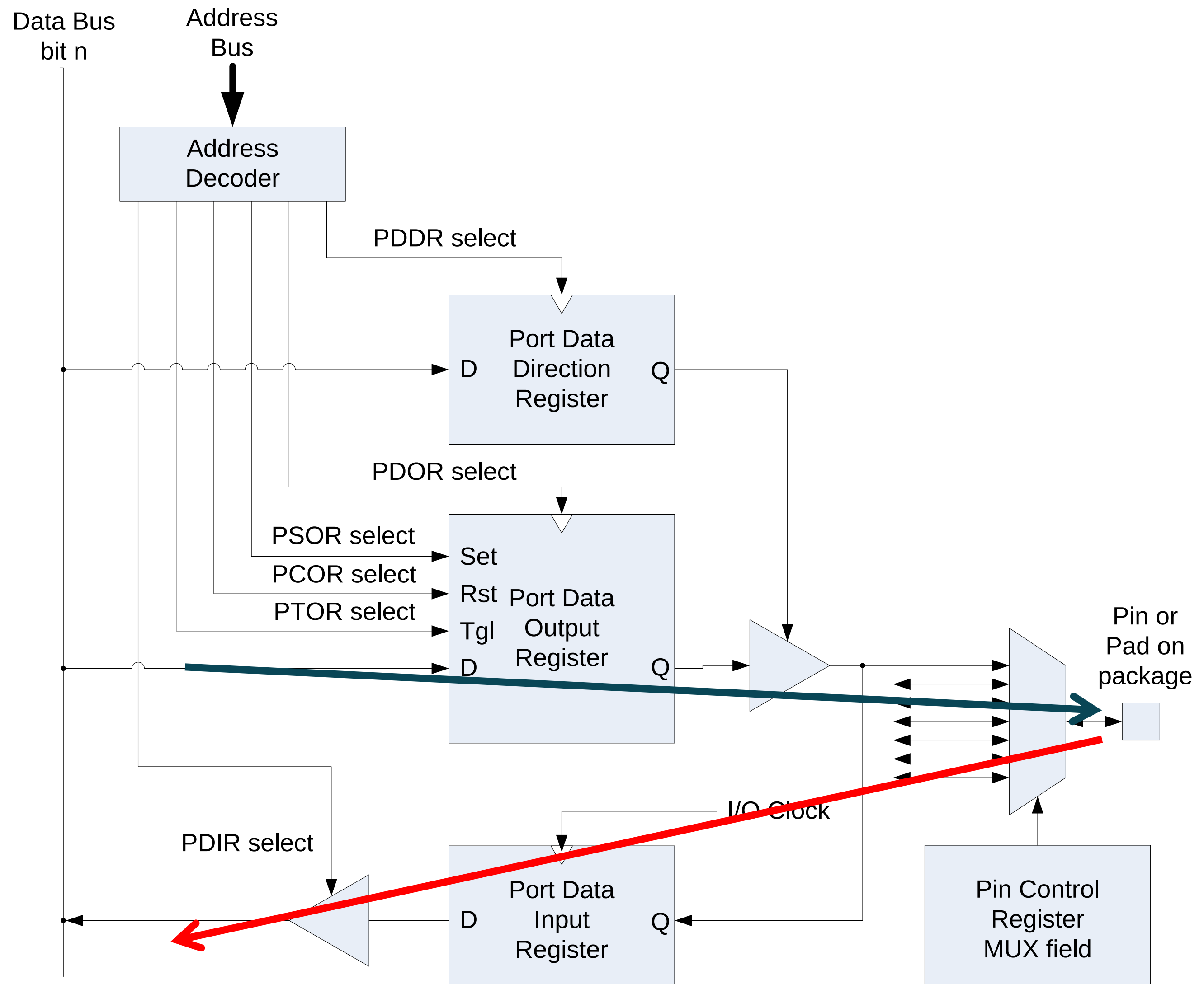


Absolute address (hex)	Register name	Width (in bits)
400F_F000	Port Data Output Register (GPIOA_PDOR)	32
400F_F004	Port Set Output Register (GPIOA_PSOR)	32
400F_F008	Port Clear Output Register (GPIOA_PCOR)	32
400F_F00C	Port Toggle Output Register (GPIOA_PTOR)	32
400F_F010	Port Data Input Register (GPIOA_PDIR)	32
400F_F014	Port Data Direction Register (GPIOA_PDDR)	32

- One set of control registers per port
- Each bit in a control register corresponds to a port bit

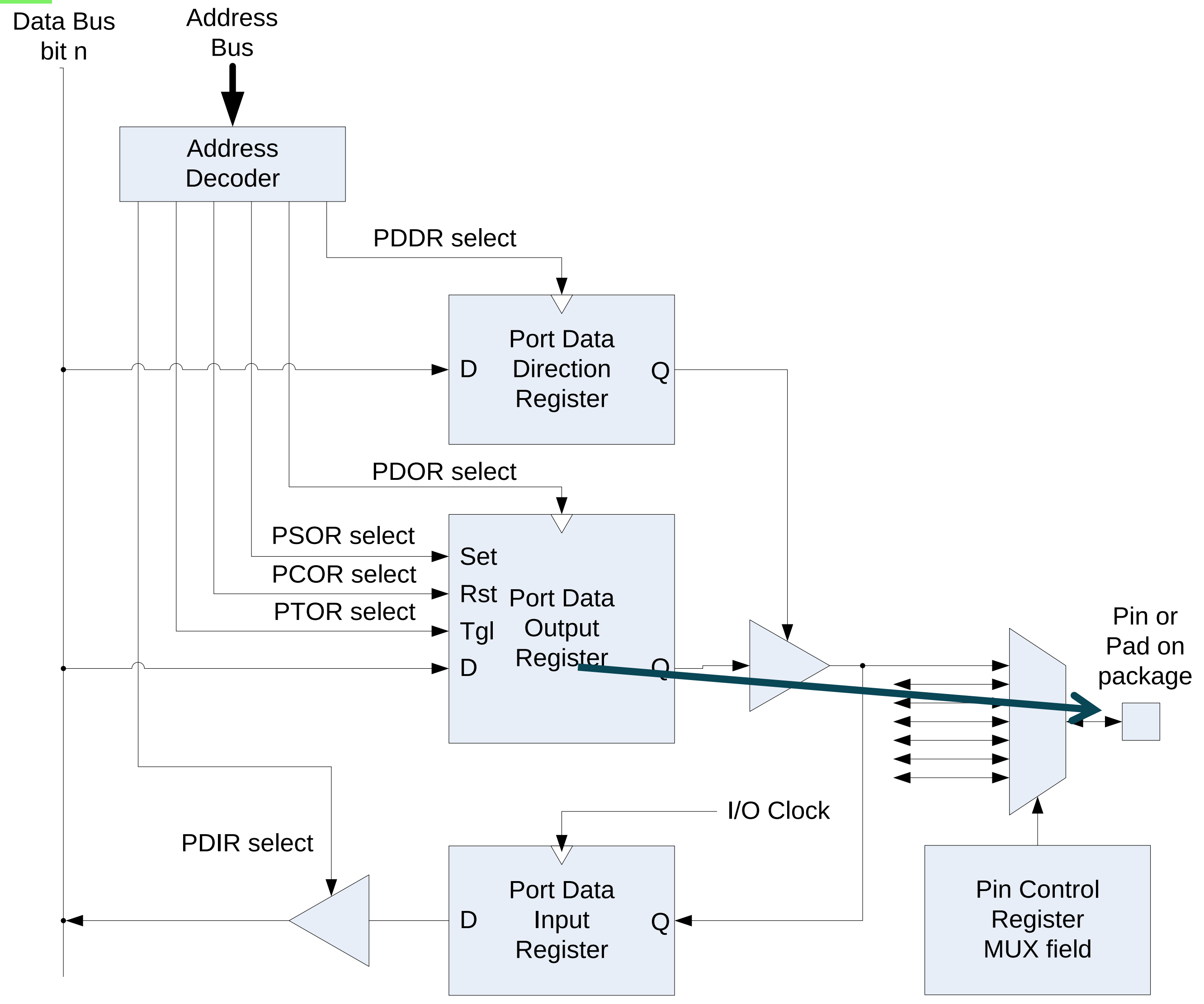
PDDR: Port Data Direction

- Each bit can be configured differently
- Input: 0
- Output: 1
- Reset clears port bit direction to 0



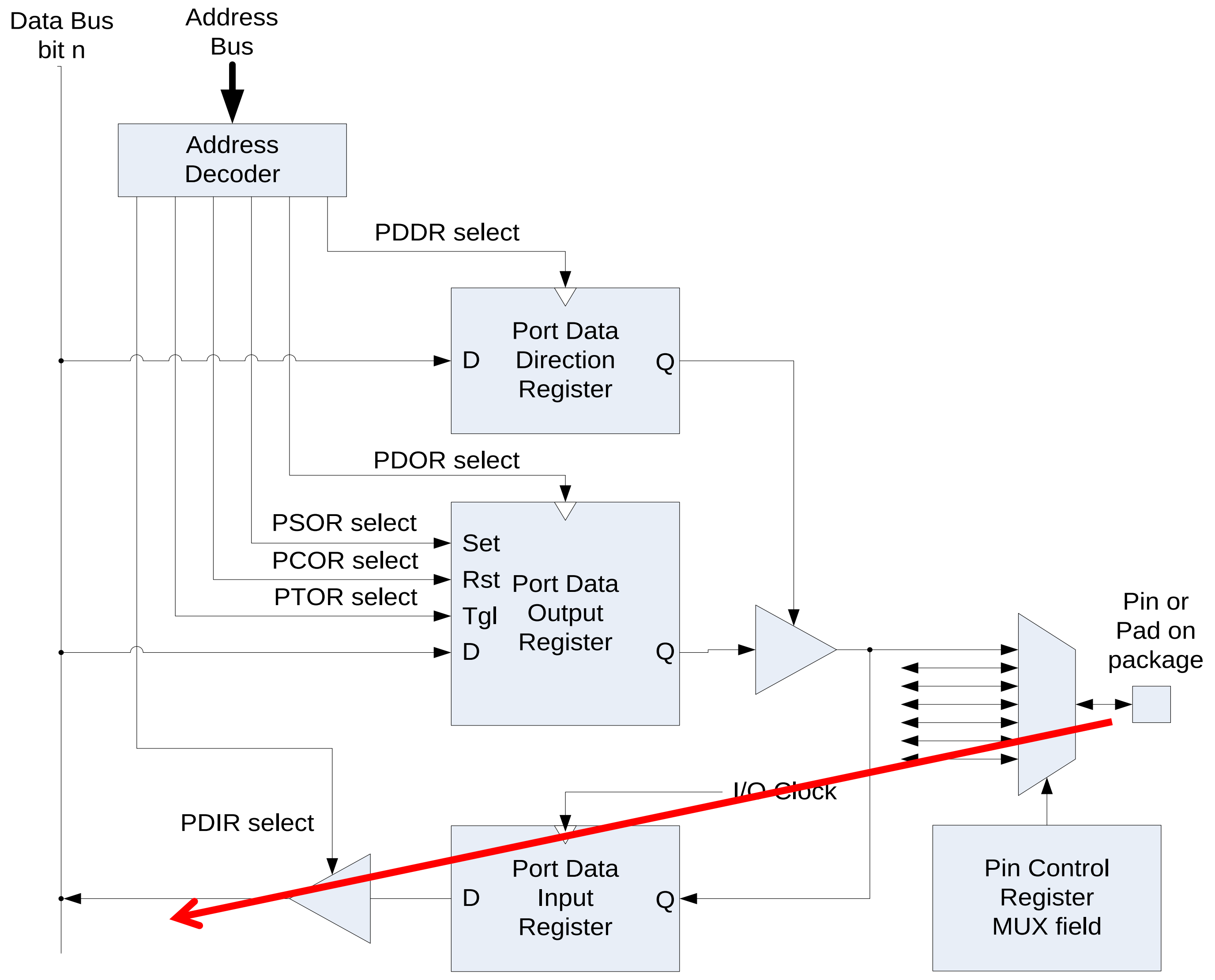
Writing Output Port Data

- Direct: write value to PDOR
- Toggle: write 1 to PTOR
- Clear (to 0): Write 1 to PCOR
- Set (to 1): write 1 to PSOR

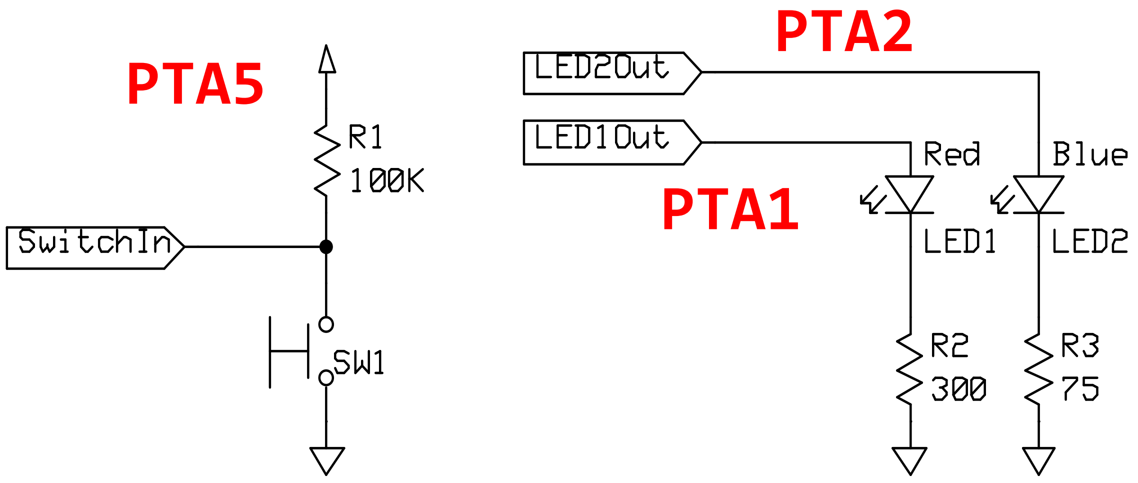


Reading Input Port Data

- Read from PDIR
- Corresponding bit holds value which was read



Pseudocode for Program



```
// Make PTA1 and PTA2 outputs
set bits 1 and 2 of GPIOA_PDDR
// Make PTA5 input
clear bit 5 of GPIOA_PDDR
// Initialize the output data values: LED 2
clear bit 1, set bit 2 of GPIOA_PDOR
// read switch, light LED accordingly
do forever {
        if bit 5 of GPIOA_PDIR is 1 {
                // switch is not pressed, then light LED 2
                set bit 2 of GPIOA_PDOR
                clear bit 1 of GPIO_PDOR
        } else {
                // switch is pressed, so light LED 1
                set bit 1 of GPIOA_PDOR
                clear bit 2 of GPIO_PDOR
        }
}
```

CMSIS - Accessing Hardware Registers in C

- Header file MKL25Z4.h defines C data structure types to represent hardware registers in MCU with CMSIS-Core hardware abstraction layer

```
#define __I volatile const
#define __O volatile
#define __IO volatile
/** GPIO - Register Layout Typedef */
typedef struct {
  __IO uint32_t PDOR;     /**< Port Data Output Register, offset: 0x0 */
  __O  uint32_t PSOR;     /**< Port Set Output Register, offset: 0x4 */
  __O  uint32_t PCOR;     /**< Port Clear Output Register, offset: 0x8 */
  __O  uint32_t PTOR;     /**< Port Toggle Output Register, offset: 0xC */
  __I  uint32_t PDIR;     /**< Port Data Input Register, offset: 0x10 */
  __IO uint32_t PDDR;     /**< Port Data Direction Register, offset: 0x14 */
} GPIO_Type;
```

Accessing Hardware Registers in C (2)

- Header file MKL25Z4.h declares pointers to the registers

```
/* GPIO - Peripheral instance base addresses */
/** Peripheral PTA base address */
#define PTA_BASE      (0x400FF000u)
/** Peripheral PTA base pointer */
#define PTA           ((GPIO_Type *)PTA_BASE)

PTA->PDOR = …
```

Coding Style and Bit Access

- Easy to make mistakes dealing with literal binary and hexadecimal values
  - "To set bits 13 and 19, use 0000 0000 0000 1000 0010 0000 0000 0000 or 0x00082000"

- Make the literal value from shifted bit positions

```
n = (1UL << 19) | (1UL << 13);
```

- Define names for bit positions

```
#define GREEN_LED_POS (19)
#define YELLOW_LED_POS (13)
n = (1UL << GREEN_LED_POS) | (1UL << YELLOW_LED_POS);
```

- Create macro to do shifting to create mask

```
#define MASK(x) (1UL << (x))
n = MASK(GREEN_LED_POS) | MASK(YELLOW_LED_POS);
```

Using Masks

- Set in n all the bits which are one in mask, leaving others unchanged

```
n |= MASK(foo);
```

- Clear in n all the bits which are zero in mask, leaving others unchanged

```
n &= ~MASK(foo);
```

- Testing a bit value in register

```
if ( n & MASK(foo) == 0) ...
if ( n & MASK(foo) == 1) ...
```

Resulting C Code for Clock Control and Mux

```
// Enable Clock to Port A
SIM->SCGC5 |= SIM_SCGC5_PORTA_MASK;

// Make 3 pins GPIO
PORTA->PCR[LED1_POS] &= ~PORT_PCR_MUX_MASK;
PORTA->PCR[LED1_POS] |= PORT_PCR_MUX(1);
PORTA->PCR[LED2_POS] &= ~PORT_PCR_MUX_MASK;
PORTA->PCR[LED2_POS] |= PORT_PCR_MUX(1);
PORTA->PCR[SW1_POS] &= ~PORT_PCR_MUX_MASK;
PORTA->PCR[SW1_POS] |= PORT_PCR_MUX(1);
```

C Code

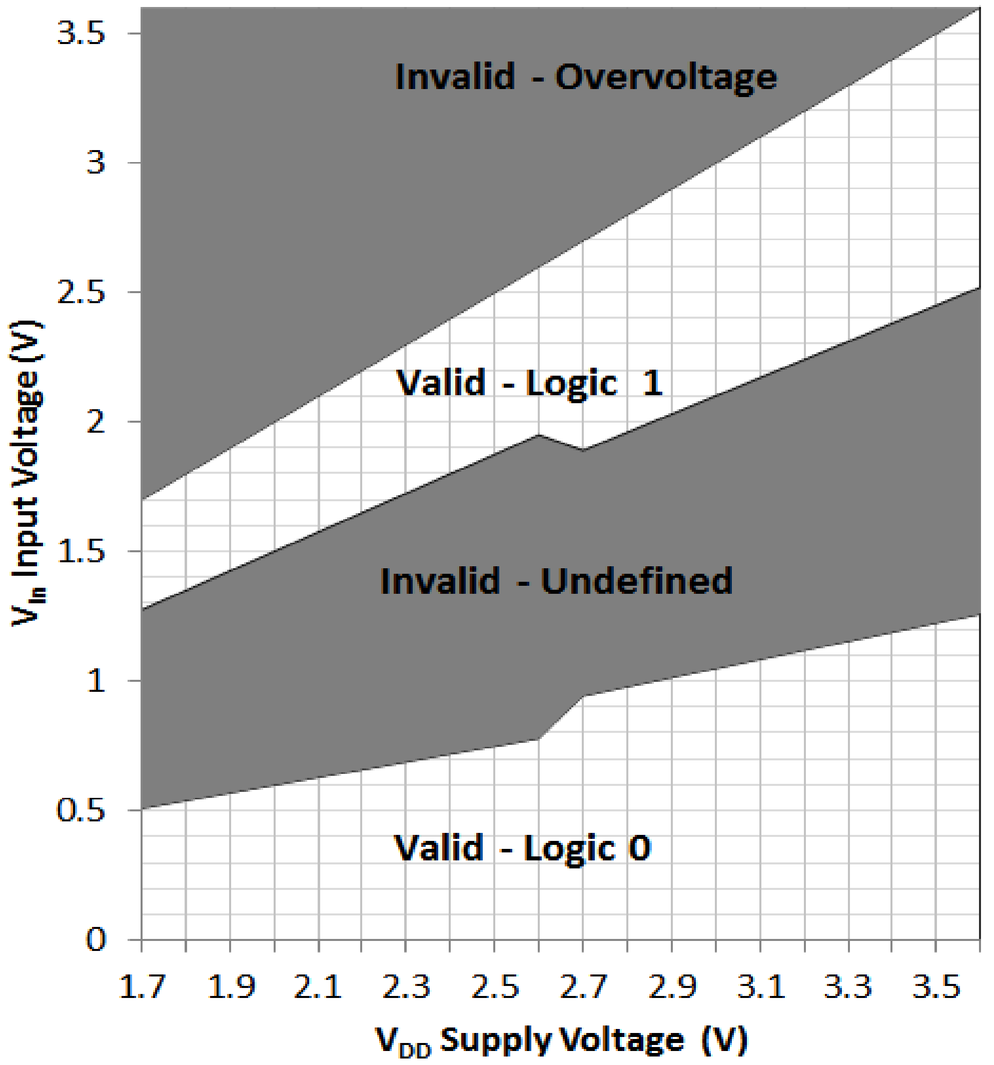
```
#define LED1_POS (1)
#define LED2_POS (2)
#define SW1_POS (5)
#define MASK(x) (1UL << (x))

PTA->PDDR |= MASK(LED1_POS) | MASK (LED2_POS); // set LED bits to outputs
PTA->PDDR &= ~MASK(SW1_POS); // clear Switch bit to input

PTA->PDOR = MASK(LED1_POS);  // turn on LED1, turn off LED2

while (1) {
     if (PTA->PDIR & MASK(SW1_POS)) {
       // switch is not pressed, then light LED 2
       PTA->PDOR = MASK(LED2_POS);
     } else {
       // switch is pressed, so light LED 1
       PTA->PDOR = MASK(LED1_POS);
     }
}
```

Inputs and Outputs, Ones and Zeros, Voltages and Currents

INTERFACING

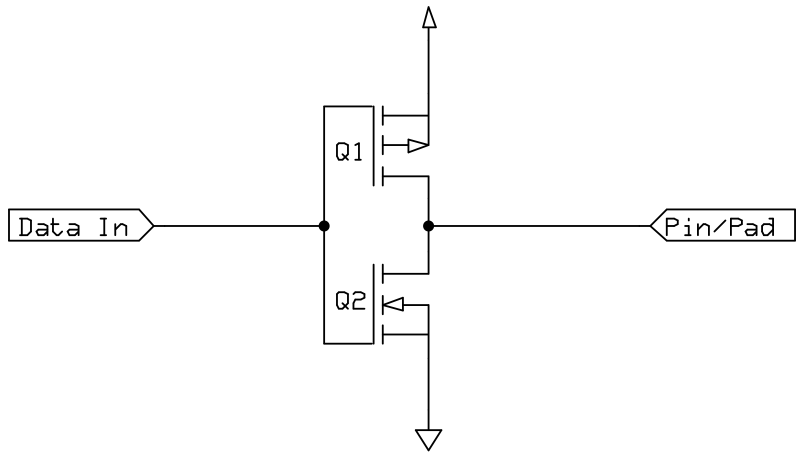
Inputs: What's a One? A Zero?

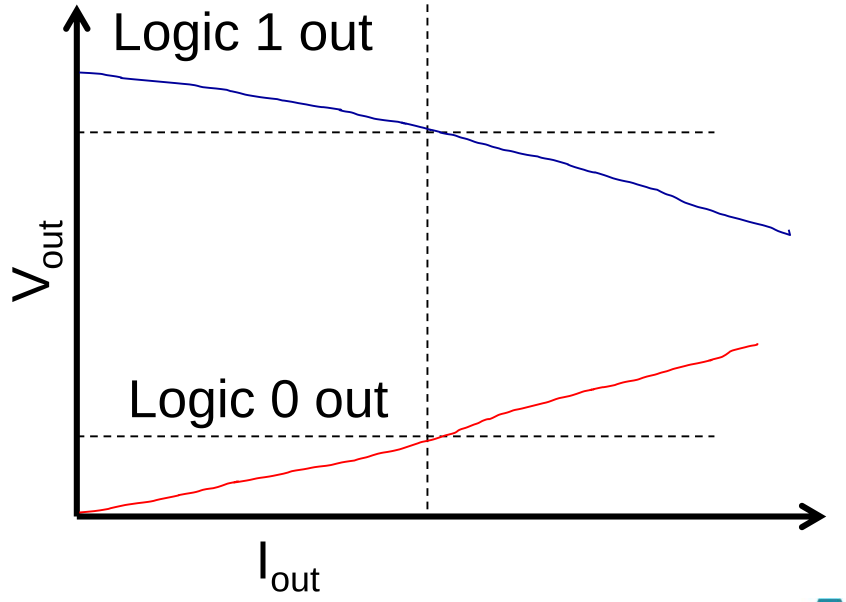
- Input signal's value is determined by voltage
- Input threshold voltages depend on supply voltage V_{DD}
- Exceeding V_{DD} or GND may damage chip



Outputs: What's a One? A Zero?

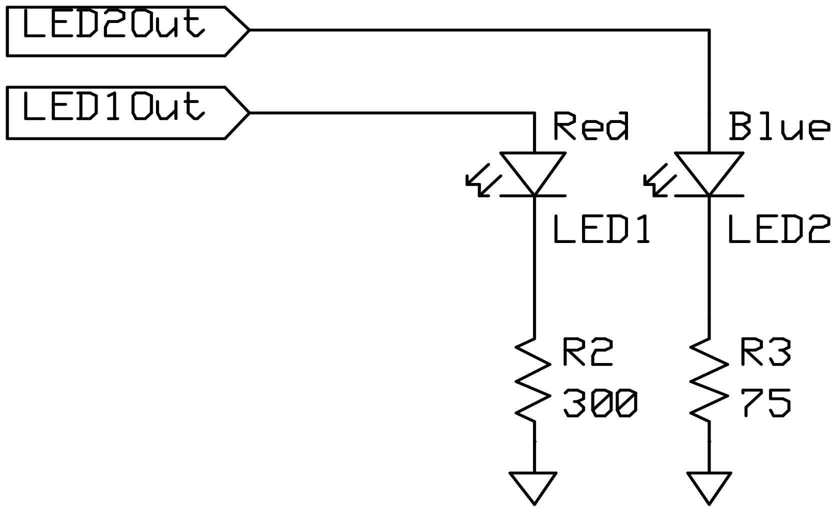
- Nominal output voltages
  - 1: V_{DD}-0.5 V to V_{DD}
  - 0: 0 to 0.5 V
- Note: Output voltage depends on current drawn by load on pin
  - Need to consider source-to-drain resistance in the transistor
  - Above values only specified when current < 5 mA (18 mA for high-drive pads) and V_{DD} > 2.7 V
- As with many other low-power ARM chips, the KL25Z can only provide (source) or sink a very limited amount of current: up to 5 mA per pin and no more than 100mA across all pins at a time
- If you source or sink more current than 5mA continuously or 20mA instantaneously, you will damage the board.





Output Example: Driving LEDs

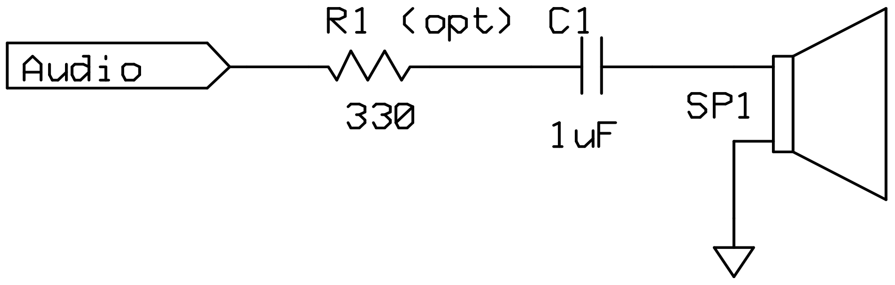
- Need to limit current to a value which is safe for both LED and MCU port driver
- Use current-limiting resistor
  - $R = (V_{DD} – V_{LED})/I_{LED}$
- Set I_{LED} = 4 mA
- V_{LED} depends on type of LED (mainly color)
  - Red: ~1.8V
  - Blue: ~2.7 V
- Solve for R given VDD = ~3.0 V
  - Red: 300 Ω
  - Blue: 75 Ω
- Demonstration code in Basic Light Switching Example

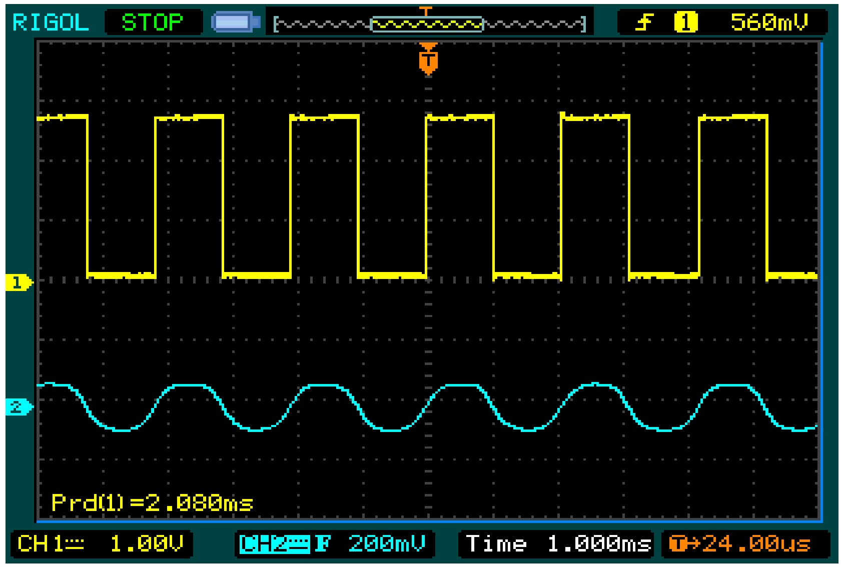


Output Example: Driving a Speaker

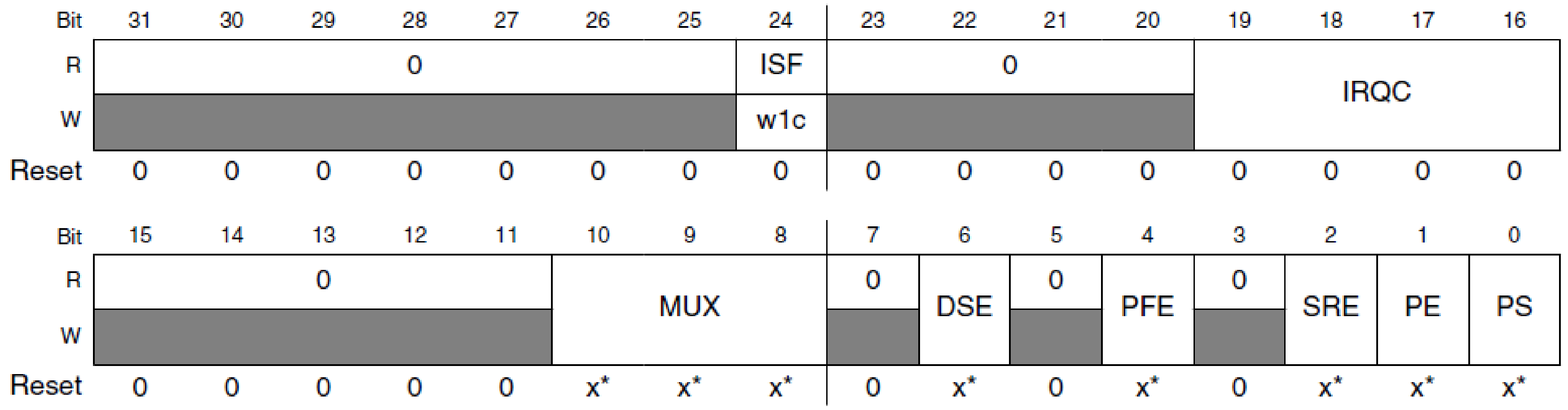
- Create a square wave with a GPIO output
- Use capacitor to block DC value
- Use resistor to reduce volume if needed
- Write to port toggle output register (PTOR) to simplify code

```
void Beep(void) {
    unsigned int period=20000;
    while (1) {
        PTC->PTOR = MASK(SPKR_POS);
        Delay(period/2);
    }
}
```





Additional Configuration in PCR

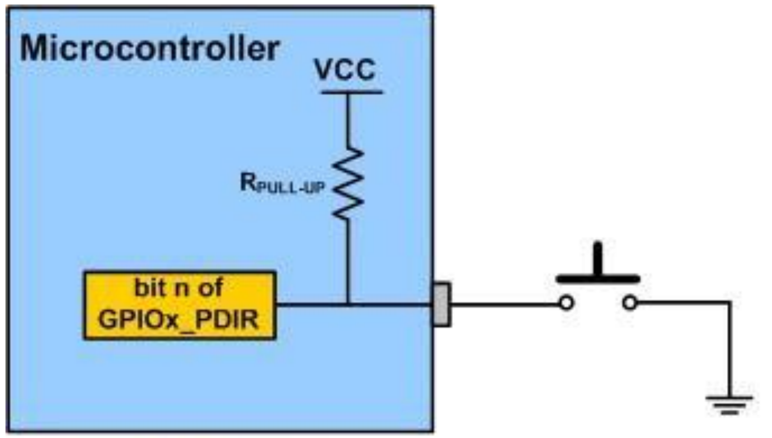


Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0							ISF	0				IRQC			
W	▨	▨	▨	▨	▨	▨	▨	w1c	▨	▨	▨	▨				
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

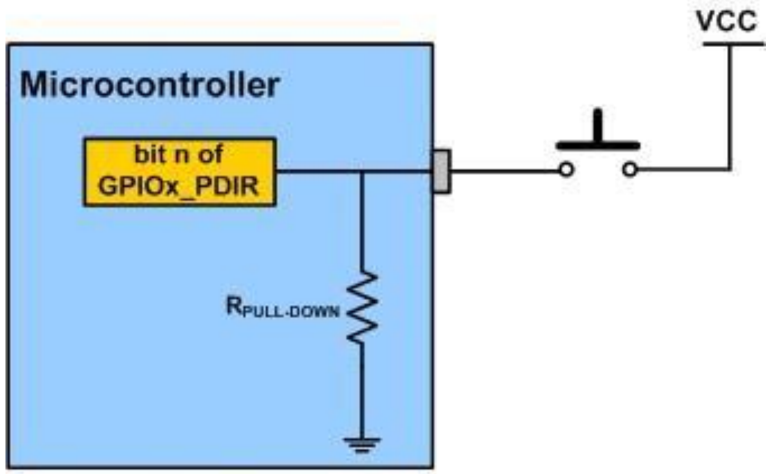
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0					MUX			0	DSE	0	PFE	0	SRE	PE	PS
W	▨	▨	▨	▨	▨				▨		▨		▨			
Reset	0	0	0	0	0	x*	x*	x*	0	x*	0	x*	0	x*	x*	x*

- Pull-up and pull-down resistors
  - Used to ensure input signal voltage is pulled to correct value when high-impedance
  - PE: Pull Enable. 1 enables the pull resistor
  - PS: Pull Select. 1 pulls up, 0 pulls down.
- High current drive strength
  - DSE: Set to 1 to drive more current (e.g. 18 mA vs. 5 mA @ > 2.7 V, or 6 mA vs. 1.5 mA @ <2.7 V)
  - Available on some pins - MCU dependent

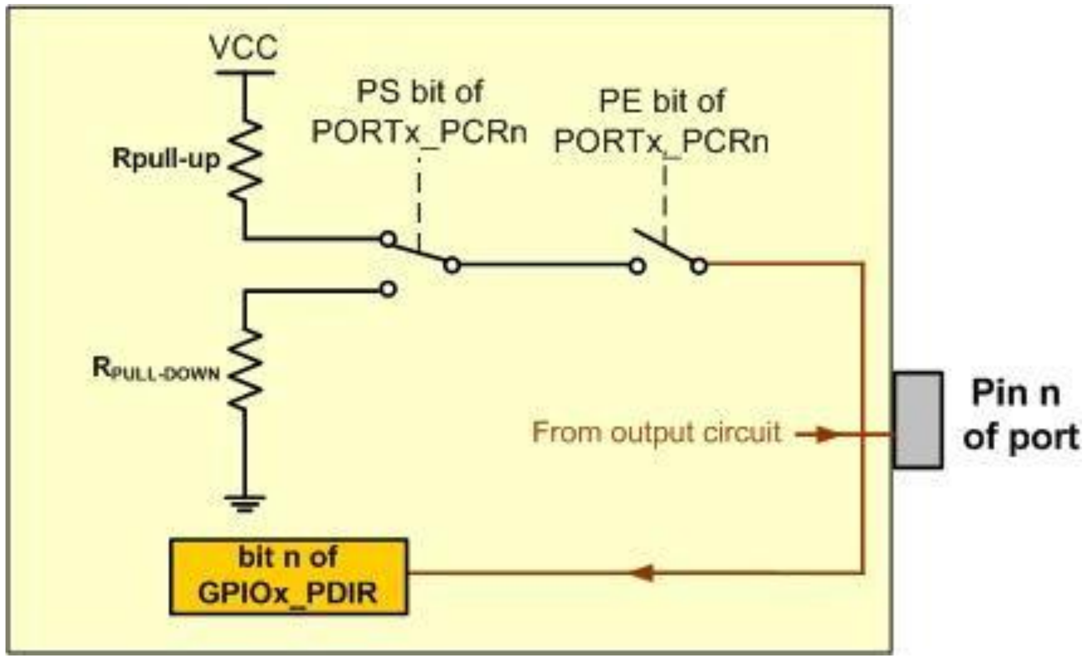
Connecting External Switches



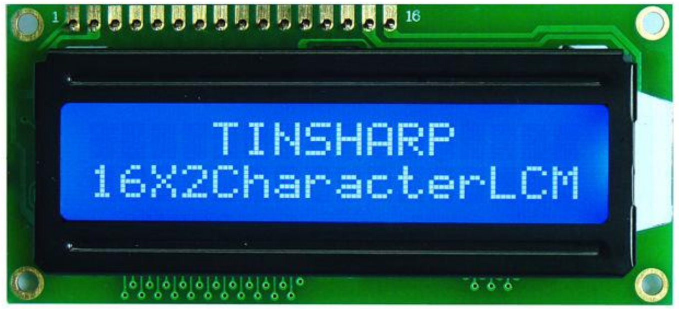
(a) Using Pull-up Resistor



(b) Using Pull-down Resistor



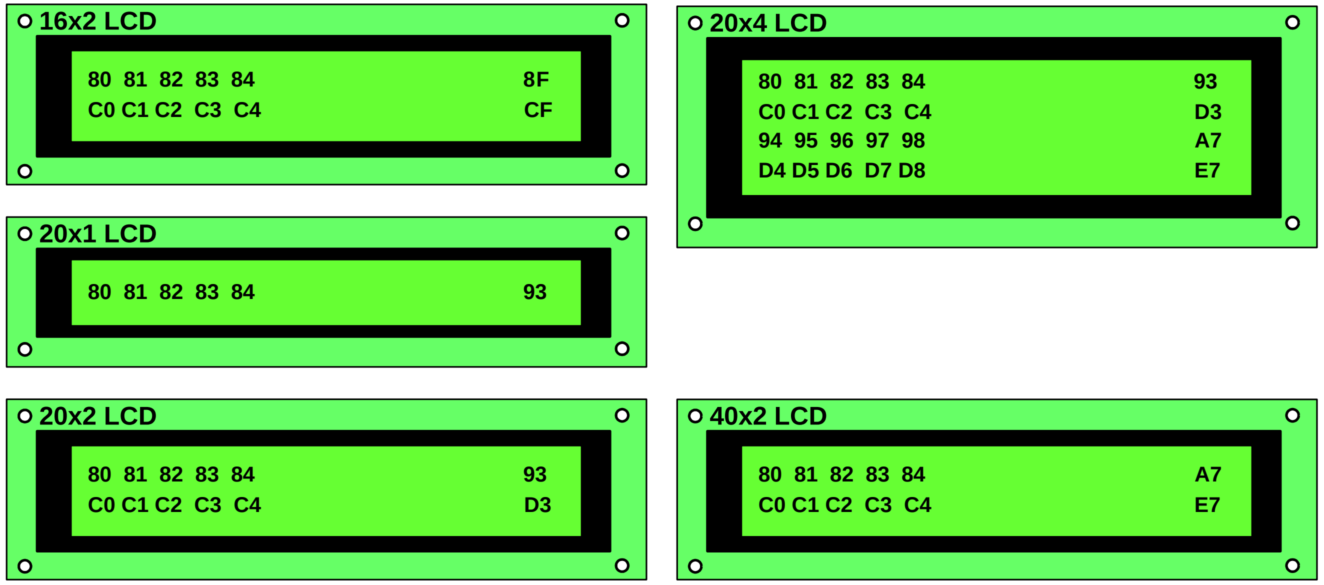
LCD Interfacing



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
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

Commands

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)

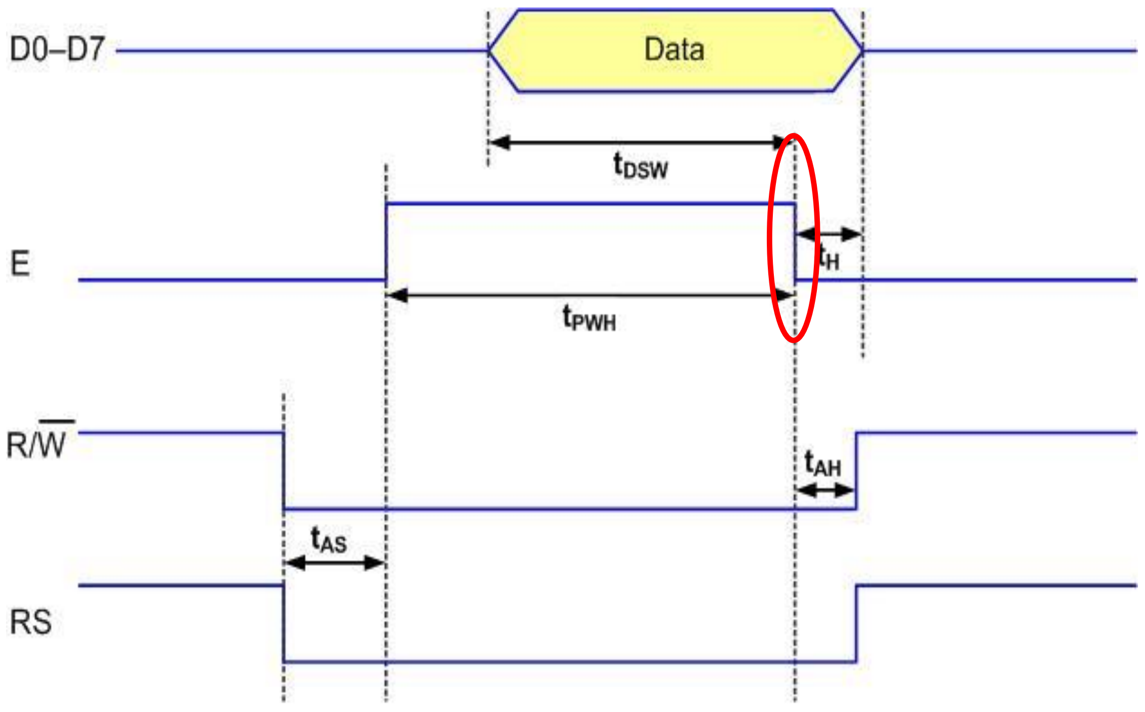


Operation

- **VCC, VSS, and VEE:** While VCC and VSS provide +5V power supply and ground, respectively, 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 (or query the busy status bit of the controller). 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.
- **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, a pulse (Low-to-High-to-Low) must be applied to this pin in order for the LCD to latch in the data present at the data pins. This pulse must be a minimum of 230 ns wide, according to Hitachi datasheet.
- **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.

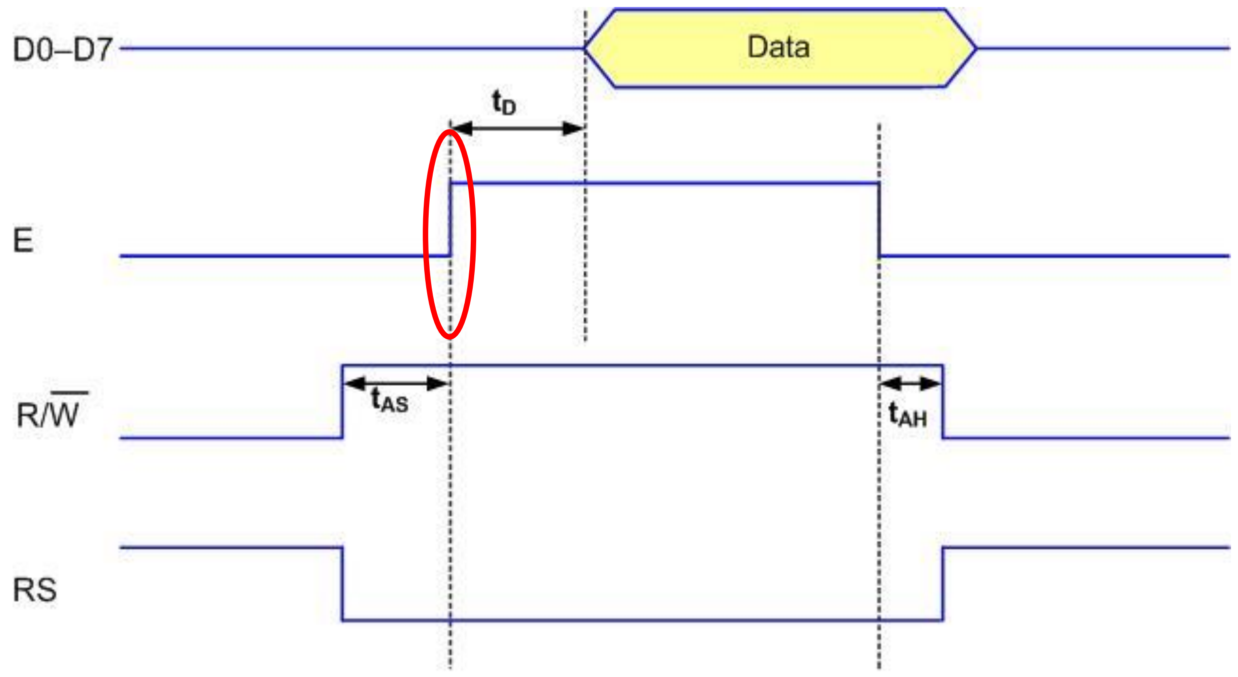
Timing

- Write 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)

- Read Timing



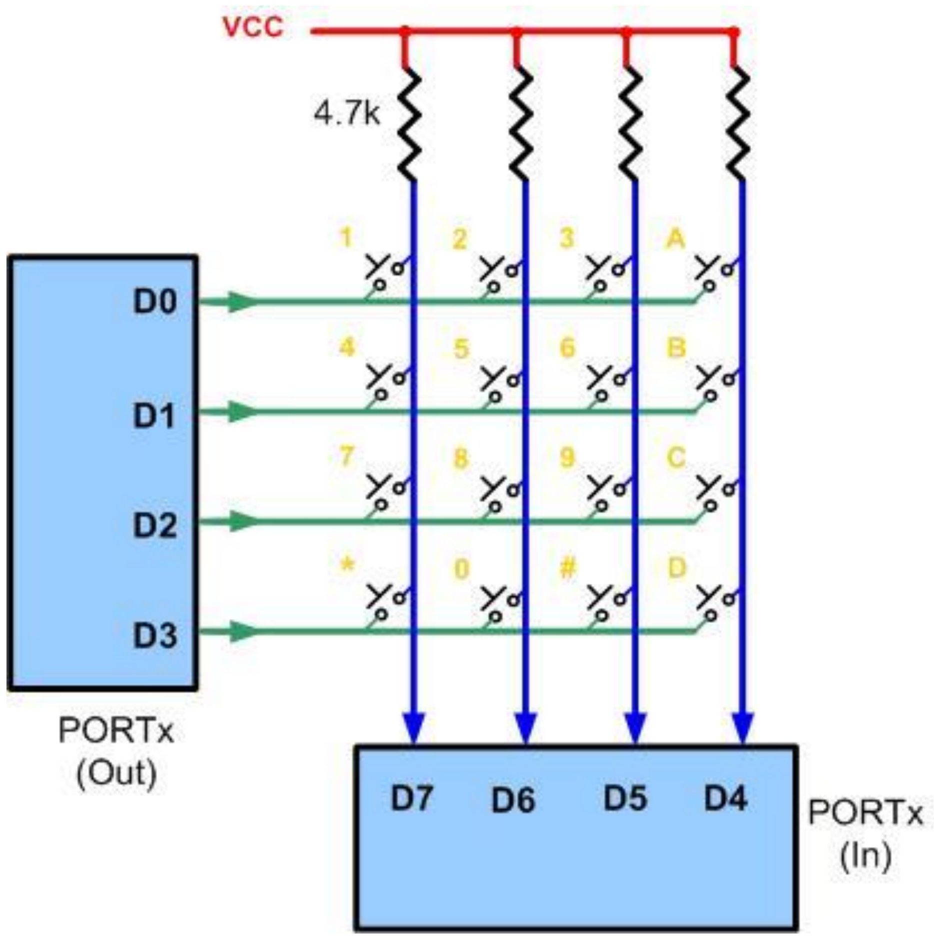
t_D = Data output delay time
t_{AS} = Setup time prior to E (going high) for both RS and R/W = 40 ns (minimum)
t_{AH} = Hold time after E has come down for both RS and R/W = 10 ns (minimum)

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

LCD Operation Contd

- We can monitor the busy flag and issue data when it is ready.
- To check the busy flag, we must read the command register (R/W = 1, RS = 0).
- 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.

Keypad



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