

AVR128DB48 Digital Output

Sung-Yeul Park

Department of Electrical & Computer Engineering

University of Connecticut

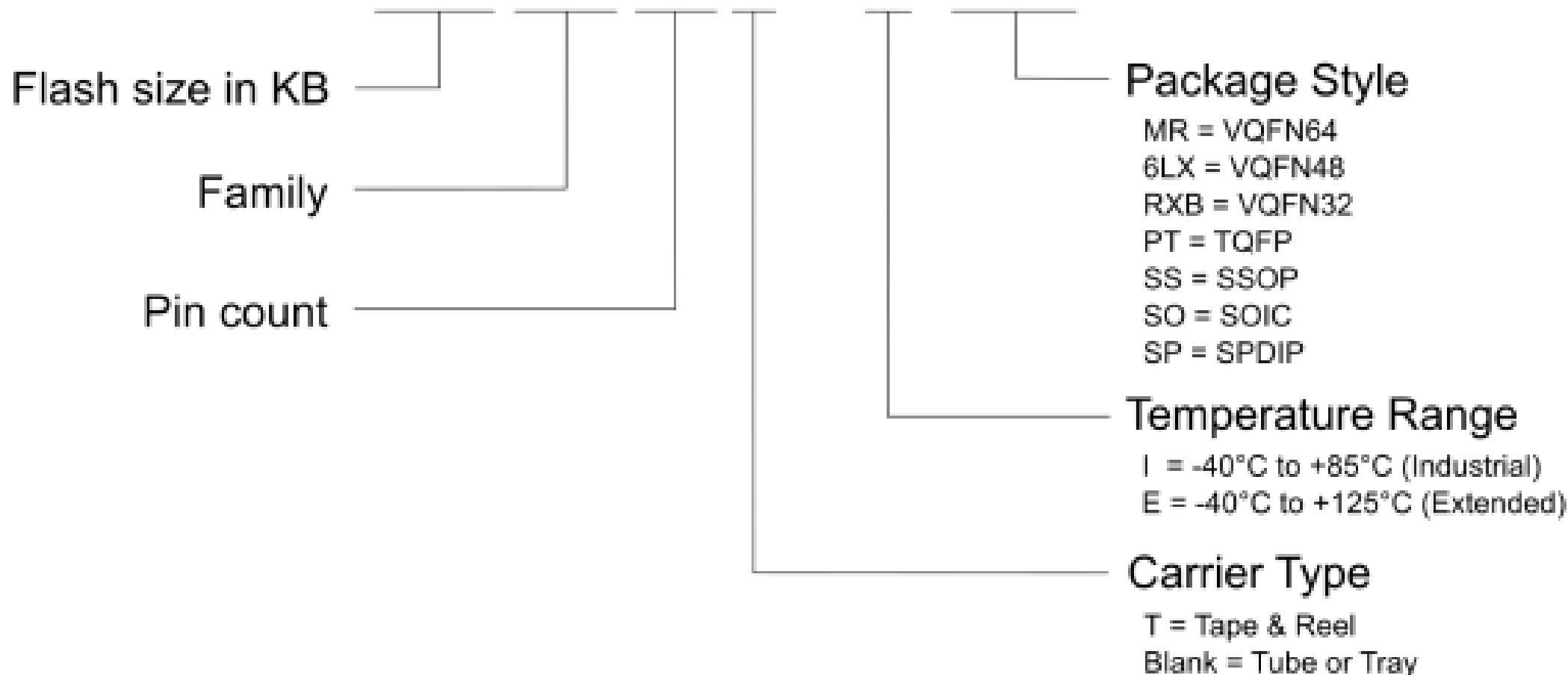
Email: sung_yeul.park@uconn.edu

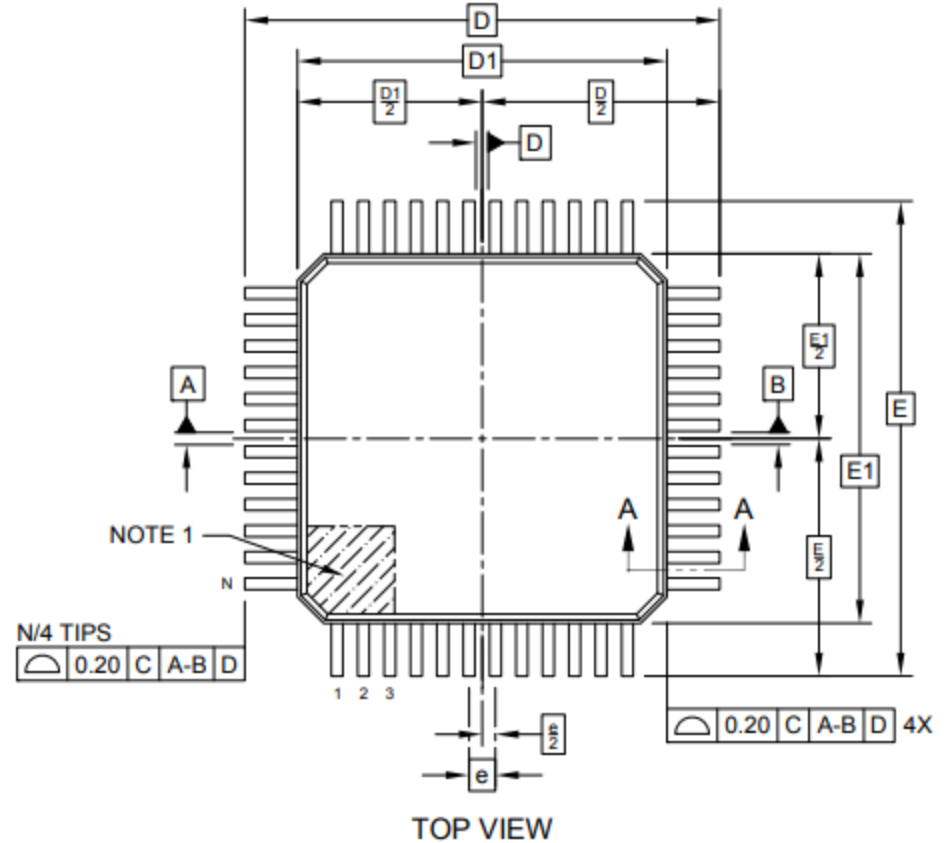
Slides adopted from Marten van Dijk, Syed Kamran Haider, ECE 3411 - Fall 2016

Slides adopted from John Chandy, ECE 3411 - Fall 2024

AVRDB Device Designations

AVR128DB64T - E/MR





48 Lead TQFP
(Thin Plastic Quad Flatpack)
- 7*7 mm body






AVR128DA vs AVR128DB Devices

Feature	AVR128DA	AVR128DB
Analog Features	Basic analog support	Enhanced analog support
Op-Amps	✗ Not available	✓ 3 integrated op-amps
Digital-to-Analog Converter (DAC)	✗ Not available	✓ 10-bit DAC
Analog Comparator	✓ 1 comparator	✓ 2 comparators
Event System	Basic routing	Enhanced routing with more flexibility
I/O Pin Functions	Standard	More alternate pin functions (e.g., for op-amps, DAC)
Package Options	Same (e.g., 28, 32, 48, 64-pin)	Same
Price	Slightly lower	Slightly higher due to analog features





AVR128DB48 Package

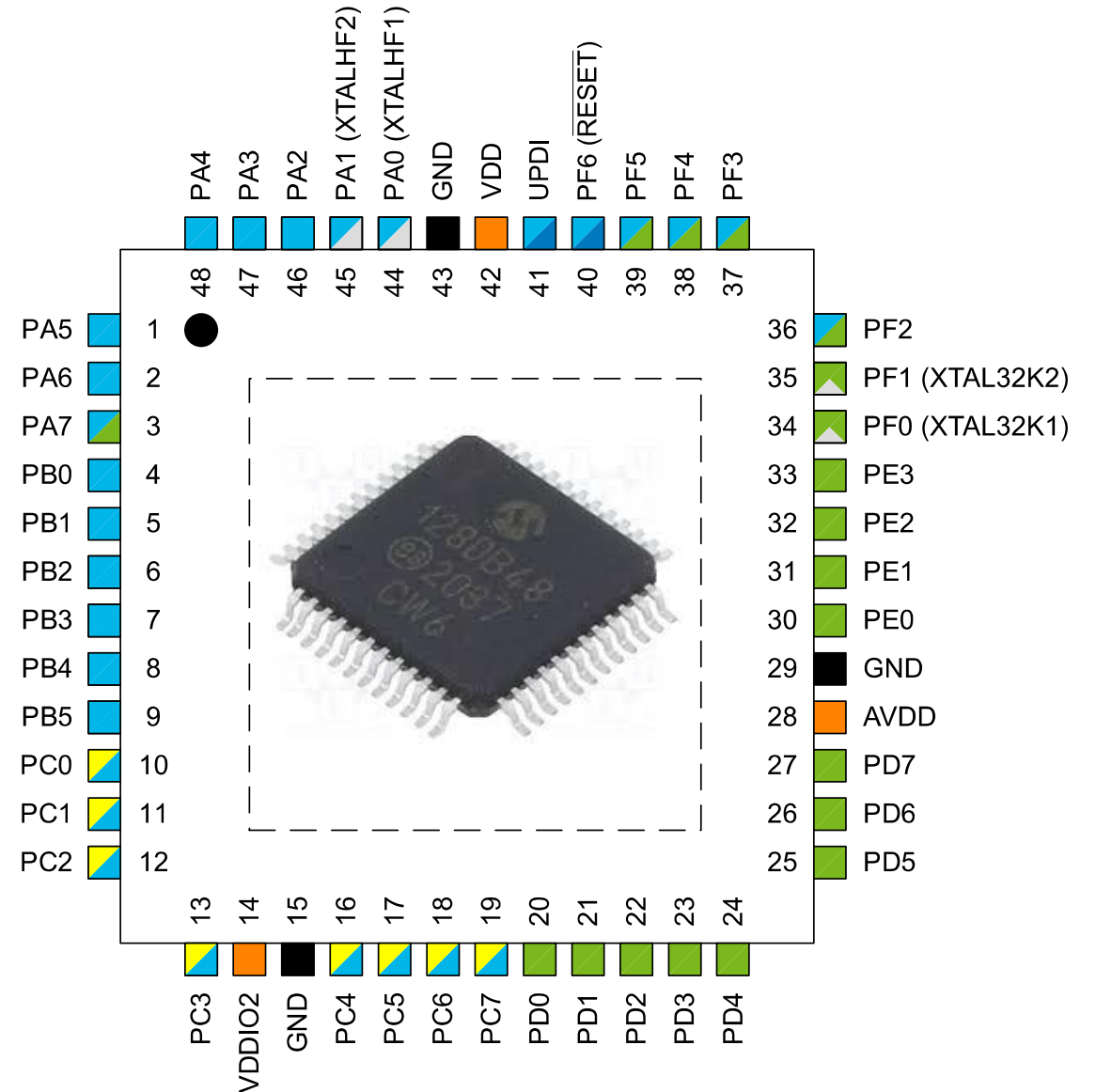
Thin Profile Plastic Quad Flat Package (TQFP)

Power

-  Power Supply
-  Ground
-  Pin on VDD Power Domain
-  Pin on AVDD Power Domain
-  Pin on VDDIO2 Power Domain

Functionality

-  Programming/Debug
-  Clock/Crystal
-  Digital Function Only
-  Analog Function

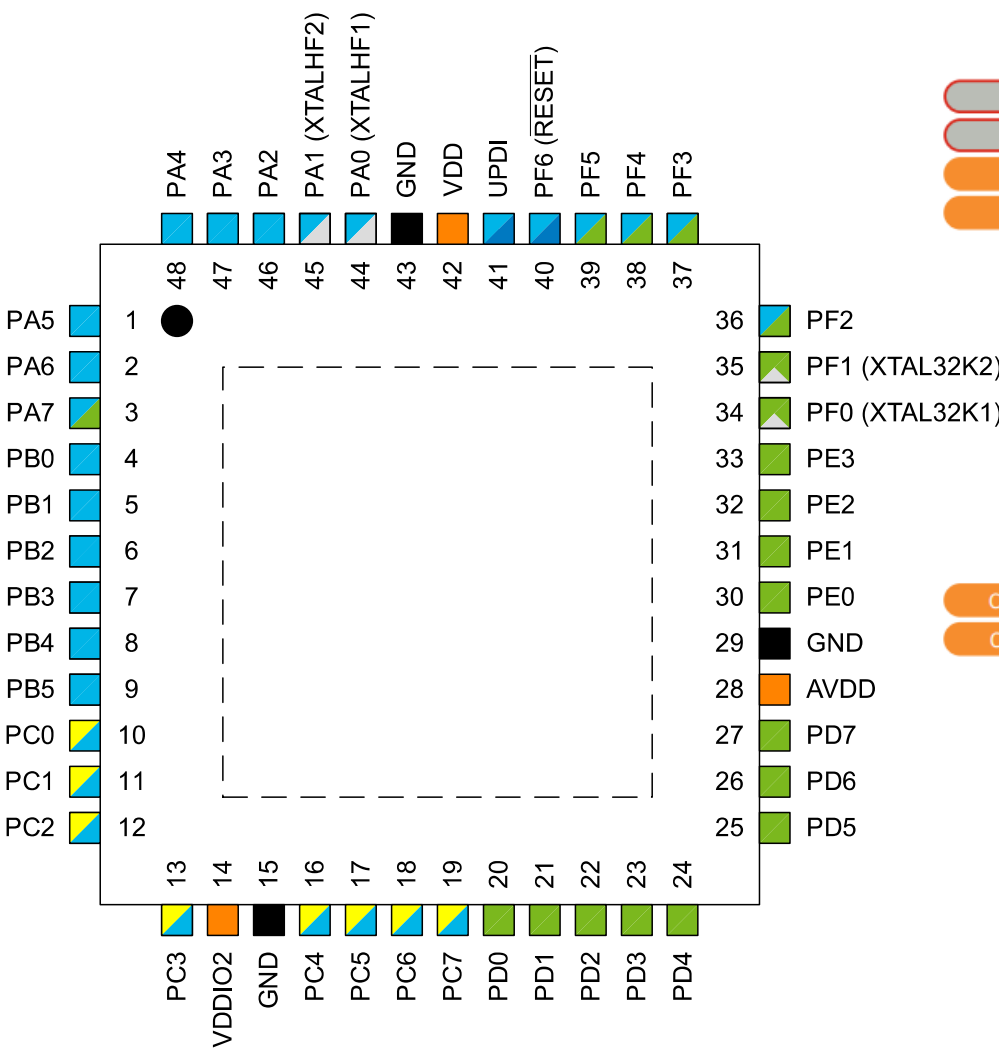


I/O Multiplexing

VQFN48/ TQFP48	Pin name(1,2)	Special	ACn	ZCDn	USARTn	SPIn	TWIn(4)	TCAn	TCBn	TCD0	EVSYS	CCL-LUTn
44	PA0	XTALHF1 EXTCLK			0, TxD			0, WO0				0, IN0
45	PA1	XTALHF2			0, RxD			0, WO1				0, IN1
46	PA2	TWI Fm+			0, XCK		0, SDA(HC)	0, WO2	0, WO		EVOUTA	0, IN2
47	PA3	TWI Fm+			0, XDIR		0, SCL(HC)	0, WO3	1, WO			0, OUT
48	PA4				0, TxD ⁽³⁾	0, MOSI		0, WO4		WOA		
1	PA5				0, RxD ⁽³⁾	0, MISO		0, WO5		WOB		
2	PA6				0, XCK ⁽³⁾	0, SCK				WOC		0, OUT ⁽³⁾
3	PA7	CLKOUT	0, OUT 1, OUT 2, OUT	0, OUT 1, OUT 2, OUT	0, XDIR ⁽³⁾	0, \overline{SS}				WOD	EVOUTA ⁽³⁾	

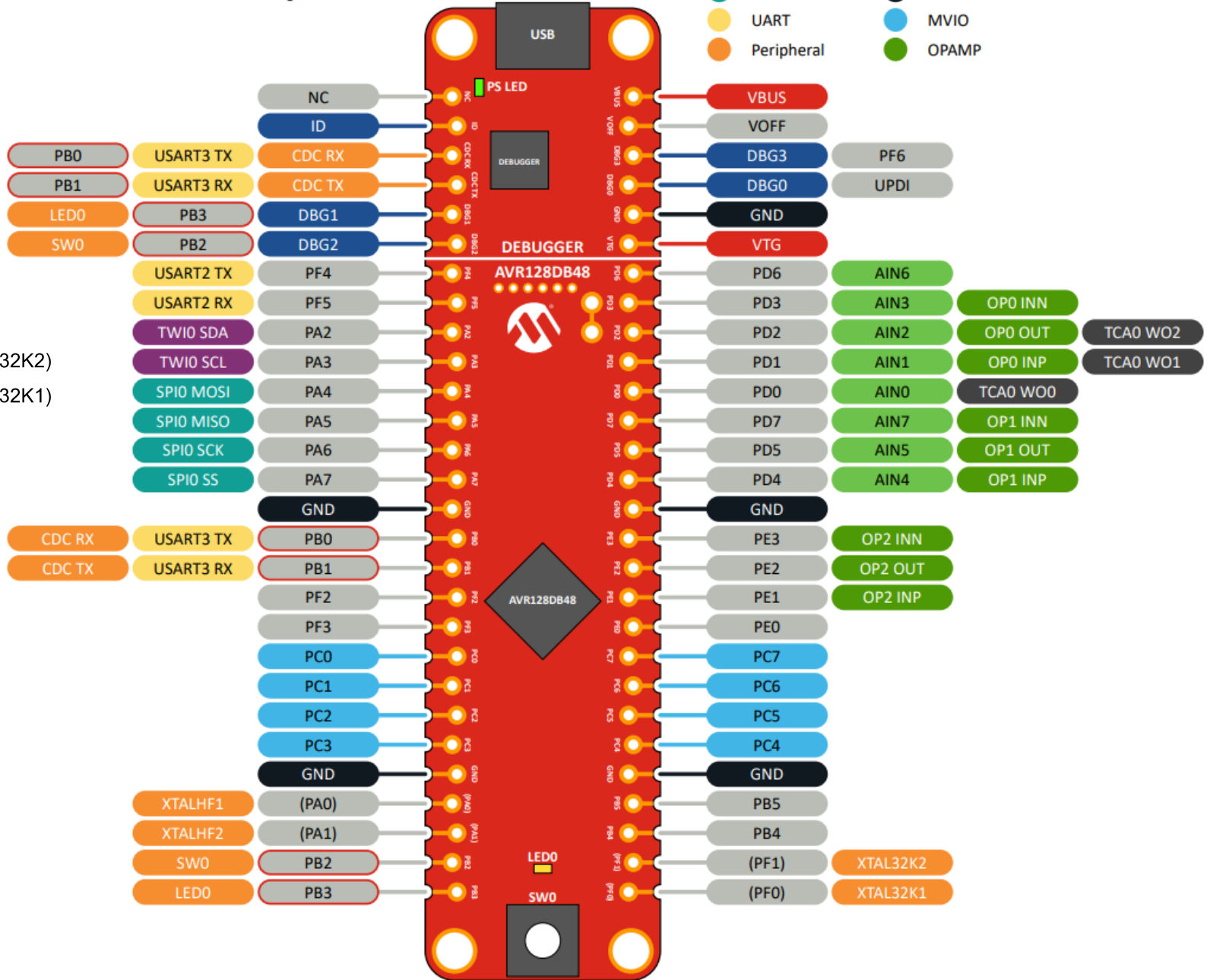
Curiosity Nano Board

Pinout



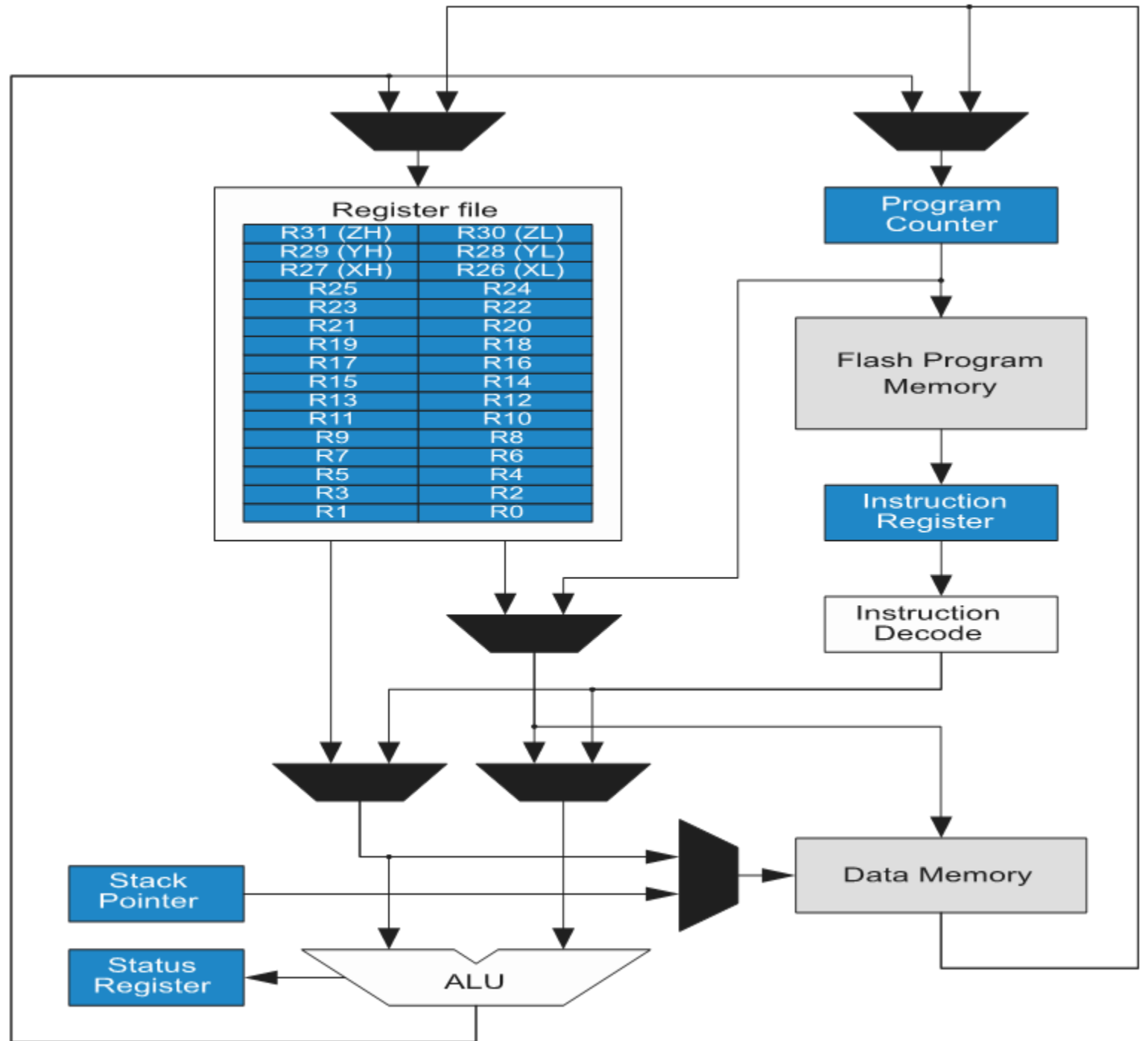
AVR128DB48

Curiosity Nano



- Analog
- Debug
- I2C
- SPI
- UART
- Peripheral
- Port
- PWM
- Power
- Ground
- MVIO
- OPAMP

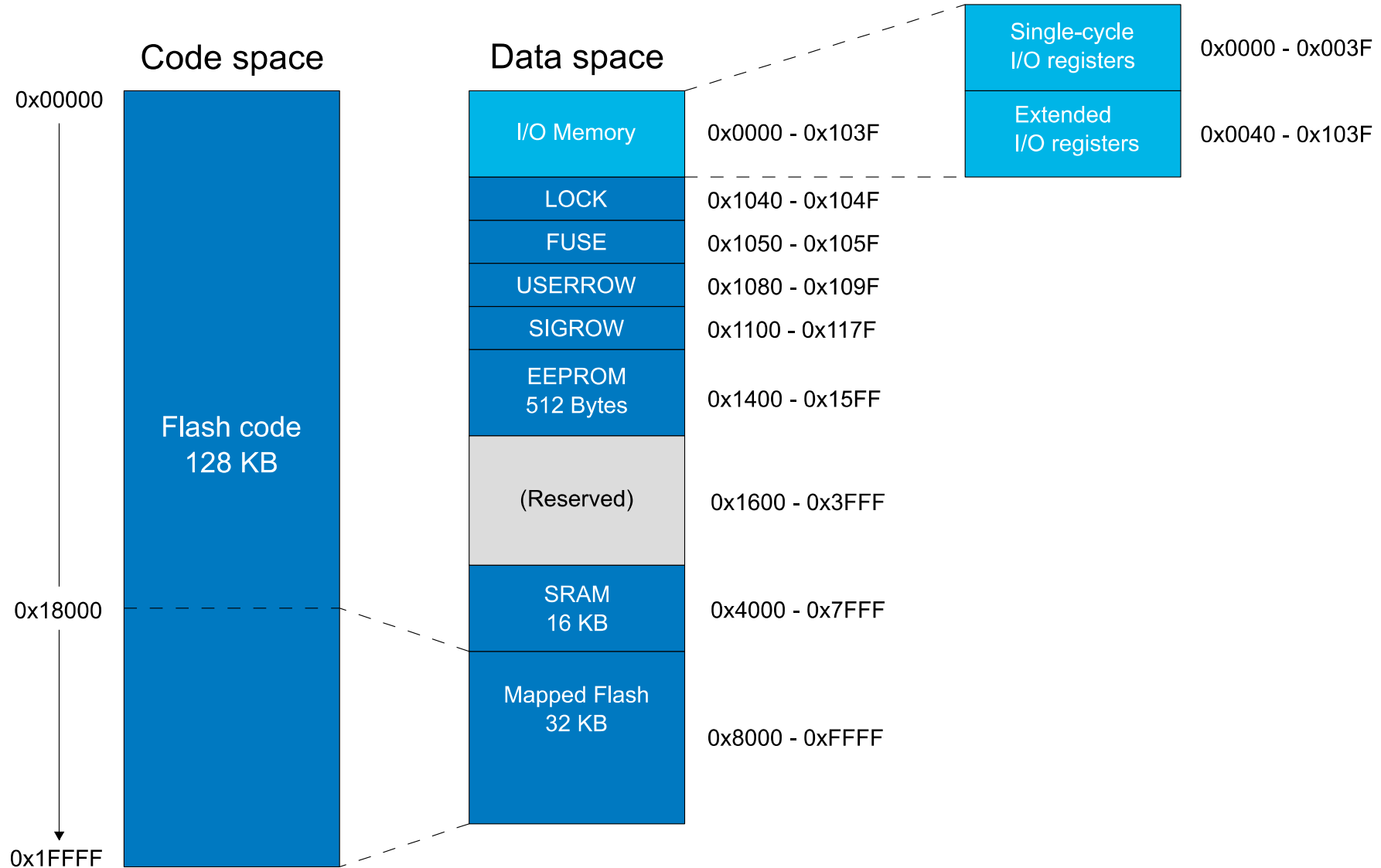
AVR Architecture



AVR Microcontroller Architecture

Component	Description
Register File (R0–R31)	Contains 32 general-purpose registers used for fast data access. Some are paired for 16-bit operations (e.g., XH/XL, YH/YL, ZH/ZL).
Program Counter (PC)	Holds the address of the next instruction to be fetched from Flash memory.
Flash Program Memory	Non-volatile memory where your compiled program code is stored.
Instruction Register	Temporarily holds the instruction fetched from Flash before execution.
Instruction Decode	Decodes the instruction and generates control signals for execution.
Data Memory	RAM used for storing variables and temporary data during program execution.
ALU (Arithmetic Logic Unit)	Performs arithmetic (add, subtract) and logical (AND, OR, XOR) operations.
Stack Pointer (SP)	Points to the current top of the stack in RAM, used for function calls and interrupts.
Status Register (SREG)	Contains flags like Zero (Z), Carry (C), Negative (N), etc., which reflect the result of ALU operations.

AVR128DB48 Memory Map



Register & Port

■ Register

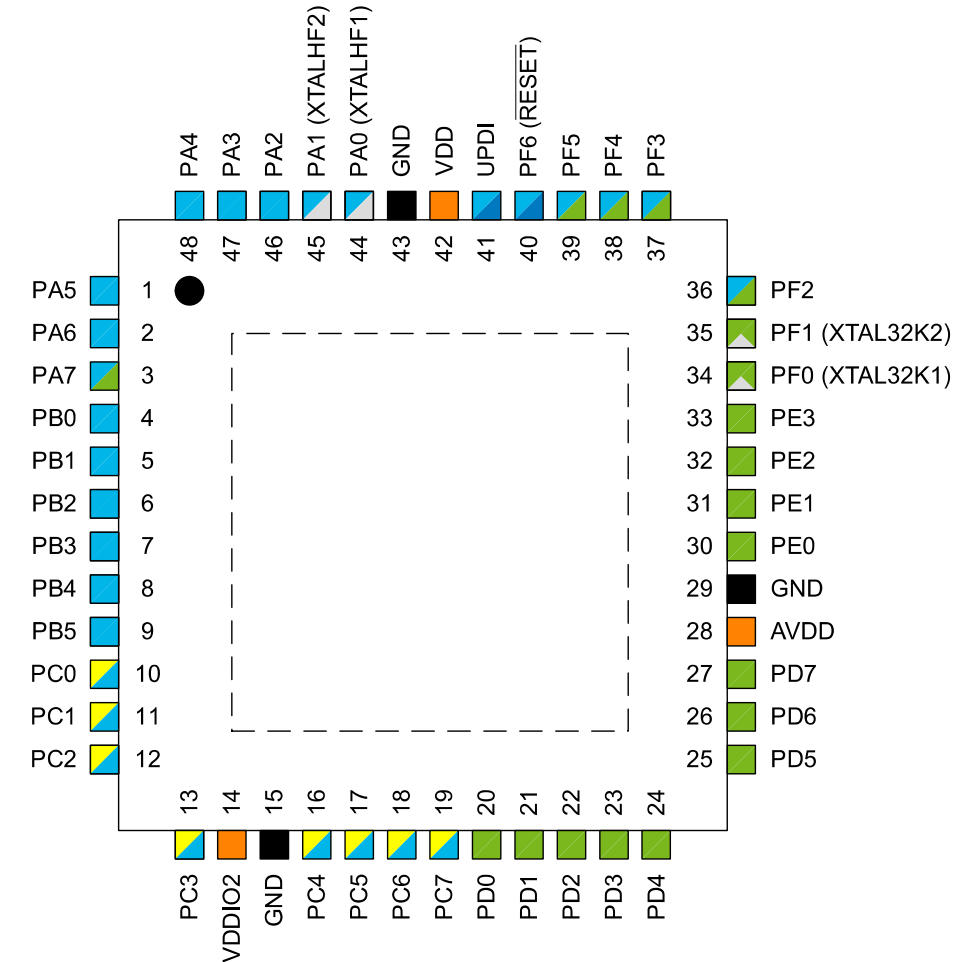
- A collection of flip-flops
- Simultaneously loaded (written) in parallel or read
- Interface between users and subsystems
- Viewed as a software configurable switch

An 8-bit wide Register



■ Port

- A Port in AVR Microcontrollers represents a bank of pins.
- A port provides an interface between the central processing unit and the internal and external hardware and software components.
- E.g., PORTA, PORTB, PORTC, PORTD, PORTE, PORTF



Hardware Registers of a Port

Each Port on the AVRDXs has three primary hardware registers associated with it:

- **DIR** : *Direction Register for Port*

- Controls whether each pin is configured for input or output.
- By default, all pins are configured as inputs.
- E.g. to enable a pin as output, a '1' is written to its slot in the DIR.

- **OUT** : *Port Output Register*

- When the DIR bits are set to '1' (output) for a given pin, the OUT register controls whether that pin is set to logic high or low.
- E.g. writing a '1' to a bit position in the OUT register will produce VCC voltage at that pin.

- **IN** : *Port Input Register*

- The IN register is used to read the digital voltage values for each pin that's configured as input.
- E.g. a value '0' of a bit in the IN register indicates a low voltage at that pin & vice versa.

AVRDX PORT Registers

■ PORT Registers

- Aside from the DIR, OUT, and IN, there are several other registers associated with each port
- The registers are grouped together in the I/O memory space

Table 10-1. Peripheral Address Map

Base Address	Name	Description	28-pin	32-pin	48-pin	64-pin
0x0400	PORTA	Port A Configuration	X	X	X	X
0x0420	PORTB	Port B Configuration			X	X
0x0440	PORTC	Port C Configuration	X	X	X	X
0x0460	PORTD	Port D Configuration	X	X	X	X
0x0480	PORTE	Port E Configuration			X	X
0x04A0	PORTF	Port F Configuration	X	X	X	X
0x04C0	PORTG	Port G Configuration				X

Digital I/O Registers

Offset	Name	Bit Pos.	7	6	5	4	3	2	1	0
0x00	DIR	7:0	DIR[7:0]							
0x01	DIRSET	7:0	DIRSET[7:0]							
0x02	DIRCLR	7:0	DIRCLR[7:0]							
0x03	DIRTGL	7:0	DIRTGL[7:0]							
0x04	OUT	7:0	OUT[7:0]							
0x05	OUTSET	7:0	OUTSET[7:0]							
0x06	OUTCLR	7:0	OUTCLR[7:0]							
0x07	OUTTGL	7:0	OUTTGL[7:0]							
0x08	IN	7:0	IN[7:0]							
0x09	INTFLAGS	7:0	INT[7:0]							
0x0A	PORTCTRL	7:0								SRL
0x0B	PINCONFIG	7:0	INVEN	INLVL			PULLUPEN		ISC[2:0]	
0x0C	PINCTRLUPD	7:0	PINCTRLUPD[7:0]							
0x0D	PINCTRLSET	7:0	PINCTRLSET[7:0]							
0x0E	PINCTRLCLR	7:0	PINCTRLCLR[7:0]							
0x0F	Reserved									
0x10	PIN0CTRL	7:0	INVEN	INLVL			PULLUPEN		ISC[2:0]	
0x11	PIN1CTRL	7:0	INVEN	INLVL			PULLUPEN		ISC[2:0]	
0x12	PIN2CTRL	7:0	INVEN	INLVL			PULLUPEN		ISC[2:0]	
0x13	PIN3CTRL	7:0	INVEN	INLVL			PULLUPEN		ISC[2:0]	
0x14	PIN4CTRL	7:0	INVEN	INLVL			PULLUPEN		ISC[2:0]	
0x15	PIN5CTRL	7:0	INVEN	INLVL			PULLUPEN		ISC[2:0]	
0x16	PIN6CTRL	7:0	INVEN	INLVL			PULLUPEN		ISC[2:0]	
0x17	PIN7CTRL	7:0	INVEN	INLVL			PULLUPEN		ISC[2:0]	

Predefined Registers

- How do you write to the PORTB OUT register?
- PORTB is at address 0x420 and the OUT is at offset 0x04

```
uint8_t* portboutaddr = (uint8_t*) 0x424;  
*portboutaddr = 0b00100100;
```

- Luckily, the AVR library has some predefined macros for each register

```
#define _SFR_MEM8(mem_addr) (*(volatile uint8_t *) (mem_addr))  
#define PORTB_OUT _SFR_MEM8(0x0424)  
  
PORTB_OUT = 0b00100100;
```

AVR128DB48 Header file snippet

```
#define PORTB_DIR           _SFR_MEM8(0x0420)
#define PORTB_DIRSET        _SFR_MEM8(0x0421)
#define PORTB_DIRCLR        _SFR_MEM8(0x0422)
#define PORTB_DIRTGL        _SFR_MEM8(0x0423)
#define PORTB_OUT           _SFR_MEM8(0x0424)
#define PORTB_OUTSET        _SFR_MEM8(0x0425)
#define PORTB_OUTCLR        _SFR_MEM8(0x0426)
#define PORTB_OUTTGL        _SFR_MEM8(0x0427)
#define PORTB_IN            _SFR_MEM8(0x0428)
#define PORTB_INTFLAGS      _SFR_MEM8(0x0429)
#define PORTB_PORTCTRL      _SFR_MEM8(0x042A)
#define PORTB_PINCONFIG     _SFR_MEM8(0x042B)
#define PORTB_PINCTRLUPD    _SFR_MEM8(0x042C)
#define PORTB_PINCTRLSET    _SFR_MEM8(0x042D)
#define PORTB_PINCTRLCLR    _SFR_MEM8(0x042E)
#define PORTB_PIN0CTRL      _SFR_MEM8(0x0430)
#define PORTB_PIN1CTRL      _SFR_MEM8(0x0431)
#define PORTB_PIN2CTRL      _SFR_MEM8(0x0432)
#define PORTB_PIN3CTRL      _SFR_MEM8(0x0433)
#define PORTB_PIN4CTRL      _SFR_MEM8(0x0434)
#define PORTB_PIN5CTRL      _SFR_MEM8(0x0435)
#define PORTB_PIN6CTRL      _SFR_MEM8(0x0436)
#define PORTB_PIN7CTRL      _SFR_MEM8(0x0437)
```


Predefined Registers

- These registers can be thought of as regular **variables**
 - You can read their values in your code
 - You can write values to these registers (except the **IN** register)
- AVR library also has predefined keywords for each bit position of each port register
 - E.g. for 5th bit position of register, the predefined keyword is **PIN5_bp**

```
#define PIN5_bp 5
```
 - Most times, it is more useful to have the bit position reflected as the value in the register – sometimes called the mask.

```
#define PIN5_bm 0x20
```
 - Setting pin 5 in the PORTB OUT register can be done one of two ways:

```
PORTB_OUT = (1<<PIN5_bp) ;  
PORTB_OUT = PIN5_bm;
```

Hardware Registers of a Port

What about those other PORT registers?

- **DIRSET, DIRCLR, DIRTGL**

```
PORTB_DIRSET = PIN5_bm; // sets bit 5, leaves others untouched
```

```
PORTB_DIRCLR = PIN5_bm; // clears bit 5, leaves others untouched
```

```
PORTB_DIRTGL = PIN5_bm; // toggles bit 5, leaves others untouched
```

- **OUTSET, OUTCLR, OUTTGL**

```
PORTB_OUTSET = PIN5_bm; // sets bit 5, leaves others untouched
```

```
PORTB_OUTCLR = PIN5_bm; // clears bit 5, leaves others untouched
```

```
PORTB_OUTTGL = PIN5_bm; // toggles bit 5, leaves others untouched
```

- We'll talk about the other registers later

Hardware Registers of a Port

- PORT registers are “extended I/O” registers. These registers take two cycles to access
- The AVR Dx has a smaller set of “single-cycle I/O” registers, and some of the frequently used PORT registers are mirrored in that region as “virtual ports”

Single-cycle I/O registers	0x0000 - 0x003F
Extended I/O registers	0x0040 - 0x103F

18.6 Register Summary - VPORTx

Offset	Name	Bit Pos.	7	6	5	4	3	2	1	0
0x00	DIR	7:0	DIR[7:0]							
0x01	OUT	7:0	OUT[7:0]							
0x02	IN	7:0	IN[7:0]							
0x03	INTFLAGS	7:0	INT[7:0]							

0x0400	PORTA
0x0420	PORTB
0x0440	PORTC
0x0460	PORTD
0x0480	PORTE
0x04A0	PORTF
0x04C0	PORTG

```
VPORTB_DIR = 0b01101100;
```

is equivalent to

```
PORTB_DIR = 0b01101100;
```

but faster.

Base Address	Name	Description
0x0000	VPORTA	Virtual Port A
0x0004	VPORTB	Virtual Port B
0x0008	VPORTC	Virtual Port C
0x000C	VPORTD	Virtual Port D
0x0010	VPORTE	Virtual Port E
0x0014	VPORTF	Virtual Port F
0x0018	VPORTG	Virtual Port G

Predefined Registers

- Since the PORT registers are grouped together, you could map each register group into a struct.

```
typedef volatile uint8_t register8_t;
typedef struct PORT_struct
{
    register8_t DIR;           /* Data Direction */
    register8_t DIRSET;       /* Data Direction Set */
    register8_t DIRCLR;       /* Data Direction Clear */
    register8_t DIRTGL;       /* Data Direction Toggle */
    register8_t OUT;          /* Output Value */
    register8_t OUTSET;       /* Output Value Set */
    register8_t OUTCLR;       /* Output Value Clear */
    register8_t OUTTGL;       /* Output Value Toggle */
    register8_t IN;           /* Input Value */
    register8_t INTFLAGS;     /* Interrupt Flags */
    register8_t PORTCTRL;     /* Port Control */
    register8_t PINCONFIG;    /* Pin Control Config */
    register8_t PINCTRLUPD;   /* Pin Control Update */
    register8_t PINCTRLSET;   /* Pin Control Set */
    register8_t PINCTRLCLR;   /* Pin Control Clear */
    register8_t reserved_1[1];
    register8_t PIN0CTRL;     /* Pin 0 Control */
    register8_t PIN1CTRL;     /* Pin 1 Control */
    register8_t PIN2CTRL;     /* Pin 2 Control */
    register8_t PIN3CTRL;     /* Pin 3 Control */
    register8_t PIN4CTRL;     /* Pin 4 Control */
    register8_t PIN5CTRL;     /* Pin 5 Control */
    register8_t PIN6CTRL;     /* Pin 6 Control */
    register8_t PIN7CTRL;     /* Pin 7 Control */
    register8_t reserved_2[8];
} PORT_t;
```

Offset	Name
0x00	DIR
0x01	DIRSET
0x02	DIRCLR
0x03	DIRTGL
0x04	OUT
0x05	OUTSET
0x06	OUTCLR
0x07	OUTTGL
0x08	IN
0x09	INTFLAGS
0x0A	PORTCTRL
0x0B	PINCONFIG
0x0C	PINCTRLUPD
0x0D	PINCTRLSET
0x0E	PINCTRLCLR
0x0F	Reserved
0x10	PIN0CTRL
0x11	PIN1CTRL
0x12	PIN2CTRL
0x13	PIN3CTRL
0x14	PIN4CTRL
0x15	PIN5CTRL
0x16	PIN6CTRL
0x17	PIN7CTRL
0x18	PIN8CTRL
0x19	PIN9CTRL
0x1A	PIN10CTRL
0x1B	PIN11CTRL
0x1C	PIN12CTRL
0x1D	PIN13CTRL
0x1E	PIN14CTRL
0x1F	PIN15CTRL

Predefined Registers

- PORT definitions using struct

```
#define PORTB      (* (PORT_t *) 0x0420) /* I/O Ports */
```

```
PORTB.DIRSET = PIN5_bm;
```

is equivalent to

```
PORTB_DIRSET = PIN5_bm;
```

- Allows you to pass pointers to PORT structures, and make code independent of the port
- Can do the same thing with VPORTs

```
VPORTB.OUT = 0b11011011;
```

The Structure of AVR C Code

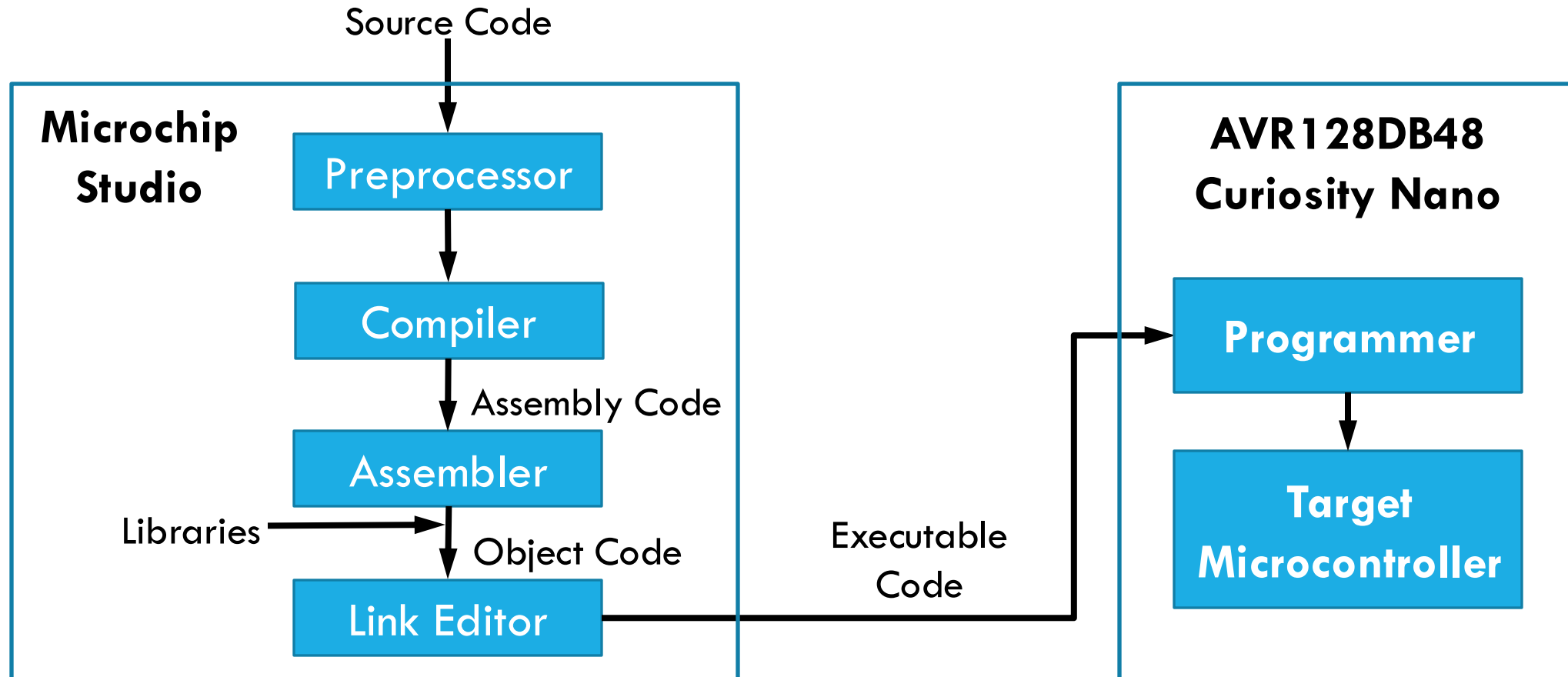
```
[preamble & includes]

[possibly some function definitions]

int main(void) {
    [chip initializations]
    while(1) {
        [do this stuff forever]
    }
    return(0);
}
```

- The preamble is where you include information from other files, define global variables, and define functions.
- **main()** is where the AVR starts executing the code when the power first goes on.
- Any configurations, e.g. configuring I/O pins etc., are done in **main()** before the **while(1)** loop.
- **while(1)** loop represents the core functionality of the program. It keeps on executing whatever is in the loop body forever (or as long as the AVR is powered).

AVR Software Development Process



Test code

```
// ----- Preamble ----- //
#define F_CPU 4000000UL      /* Tells the Clock Freq to the Compiler. */
#include <avr/io.h>          /* Defines pins, ports etc. */
#include <util/delay.h>      /* Functions to waste time */

int main(void) {

    /* Data Direction Register D: writing a one to the bit enables output. */
    VPORTD.DIR = 0b11111111;
    VPORTD.OUT = 0b00000100;

    // ----- Event loop ----- //
    while (1) {
        VPORTD.OUT = 0b11111111; /* Turn on the LED bits/pins in PORTD */
        _delay_ms(1000);          /* wait for 1 second */
        VPORTD.OUT = 0b00000000; /* Turn off all D pins/LEDs */
        _delay_ms(1000);          /* wait for 1 second */
    } /* End event loop */
    return (0); /* This line is never reached */
}
```


The Delay Library

- AVR supports a delay library to introduce delay between the execution of two code statements.
 - `<util/delay.h>` header file needs to be included in the code
- The delay library provides two functions
 - `_delay_us(x)` for introducing a delay of x microseconds
 - `_delay_ms(x)` for introducing a delay of x milliseconds
- `<util/delay.h>` library needs to know the Microcontroller's clock frequency for accurate time measurements
 - Clock frequency is defined by defining `F_CPU` in the code
- The AVR128DB48 runs at 4MHz by default but can support up to 24MHz frequency either through internal oscillator or external crystal

```
#define F_CPU 4000000UL
```

Changing the clock frequency

- Configuring AVR128DB48 to use 16MHz internal oscillator

```
#define F_CPU 16000000UL
void init_clock() {
    CPU_CCP = CCP_IOREG_gc;
    CLKCTRL.OSCHFCTRLA = CLKCTRL_FRQSEL_16M_gc;
}
```

- Configuring AVR128DB48 to use 16MHz external crystal – more accurate

```
#define F_CPU 16000000UL
void init_clock() {
    CPU_CCP = CCP_IOREG_gc;
    CLKCTRL.XOSCHFCTRLA = CLKCTRL_FRQRANGE_16M_gc |
                          CLKCTRL_ENABLE_bm;

    CPU_CCP = CCP_IOREG_gc;
    CLKCTRL.MCLKCTRLA = CLKCTRL_CLKSEL_EXTCLK_gc;
}
```

CCP: Configuration Change Protection

7.6.1 Configuration Change Protection

Name: CCP
Offset: 0x04
Reset: 0x00
Property: -

Bit	7	6	5	4	3	2	1	0
	CCP[7:0]							
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Bits 7:0 – CCP[7:0] Configuration Change Protection

Writing the correct signature to this bit field allows changing protected I/O registers or executing protected instructions within the next four CPU instructions executed.

All interrupts are ignored during these cycles. After these cycles are completed, the interrupts will automatically be handled again by the CPU, and any pending interrupts will be executed according to their level and priority.

When the protected I/O register signature is written, CCP[0] will read as '1' as long as the CCP feature is enabled.

When the protected self-programming signature is written, CCP[1] will read as '1' as long as the CCP feature is enabled.

CCP[7:2] will always read as '0'.

Value	Name	Description
0x9D	SPM	Allow Self-Programming
0xD8	IOREG	Unlock protected I/O registers

MCLKCTRLA: Main Clock Control A

12.5.1 Main Clock Control A

Name: MCLKCTRLA
Offset: 0x00
Reset: 0x00
Property: Configuration Change Protection

Bit	7	6	5	4	3	2	1	0
	CLKOUT					CLKSEL[3:0]		
Access	R/W				R/W	R/W	R/W	R/W
Reset	0				0	0	0	0

Bit 7 – CLKOUT Main Clock Out

This bit controls whether the main clock is available on the Main Clock Out (CLKOUT) pin or not, when the main clock is running.

This bit is cleared when a '0' is written to it or when a Clock Failure Detection (CFD) condition with the main clock as source occurs.

This bit is set when a '1' is written to it.

Value	Description
0	The main clock is not available on the CLKOUT pin
1	The main clock is available on the CLKOUT pin

Bits 3:0 – CLKSEL[3:0] Clock Select

This bit field controls the source for the Main Clock (CLK_MAIN).

Value	Name	Description
0x0	OSCHF	Internal high-frequency oscillator
0x1	OSC32K	32.768 kHz internal oscillator
0x2	XOSC32K	32.768 kHz external crystal oscillator
0x3	EXTCLK	External clock or external crystal, depending on the SELHF bit in XOSCHFCTRLA
Other	Reserved	Reserved

XOSCHFCTRLA: External High-Frequency Oscillator Control A(1)

12.5.12 External High-Frequency Oscillator Control A

Name: XOSCHFCTRLA
Offset: 0x20
Reset: 0x00
Property: Configuration Change Protection

Bit	7	6	5	4	3	2	1	0
	RUNSTDBY		CSUTHF[1:0]		FRQRANGE[1:0]		SELHF	ENABLE
Access	R/W		R/W	R/W	R/W	R/W	R/W	R/W
Reset	0		0	0	0	0	0	0

Bit 7 – RUNSTDBY Run Standby

This bit controls whether the External High-Frequency Oscillator (XOSCHF) is always running or not, when the ENABLE bit is '1'.

Value	Description
0	The XOSCHF oscillator will only run when requested by a peripheral or by the main clock ⁽¹⁾
1	The XOSCHF oscillator will always run in Active, Idle and Standby sleep modes ⁽²⁾

Notes:

1. The requesting peripheral, or the main clock, must take the oscillator start-up time into account.
2. The oscillator signal is only available if requested, and will be available after two XOSCHF cycles, if the initial crystal start-up time has already ended.

XOSCHFCTRLA: External High-Frequency Oscillator Control A(2)

Bits 5:4 – CSUTHF[1:0] Crystal Start-up Time

This bit field controls the start-up time for the External High-Frequency Oscillator (XOSCHF), when the Source Select (SELHF) bit is '0'.

Value	Name	Description
0x0	256	256 XOSCHF cycles
0x1	1K	1K XOSCHF cycles
0x2	4K	4K XOSCHF cycles
0x3	-	Reserved

Note: This bit field is read-only when the ENABLE bit or the External Crystal/Clock Status (XOSCHFS) bit in the Main Clock Status (MCLKSTATUS) register is '1'.

Bits 3:2 – FRQRANGE[1:0] Frequency Range

This bit field controls the maximum frequency supported for the external crystal. The larger the range selected, the higher the current consumption by the oscillator.

Value	Name	Description
0x0	8M	Max. 8 MHz XTAL frequency
0x1	16M	Max. 16 MHz XTAL frequency
0x2	24M	Max. 24 MHz XTAL frequency
0x3	32M	Max. 32 MHz XTAL frequency

Note: If a crystal with a frequency larger than the maximum supported CLK_CPU frequency is used and used as the main clock, it is necessary to divide it down by writing the appropriate configuration to the PDIV bit field in the Main Clock Control B register.

XOSCHFCTRLA: External High-Frequency Oscillator Control A(2)

Bit 1 – SELHF Source Select

This bit controls the source of the External High-Frequency Oscillator (XOSCHF).

Value	Name	Description
0	CRYSTAL	External Crystal on the XTALHF1 and XTALHF2 pins
1	EXTCLOCK	External Clock on the XTALHF1 pin

Note: This bit field is read-only when the ENABLE bit or the External Crystal/Clock Status (XOSCHFS) bit in the Main Clock Status (MCLKSTATUS) register is '1'.

Bit 0 – ENABLE Enable

This bit controls whether the External High-Frequency Oscillator (XOSCHF) is enabled or not.

Value	Description
0	The XOSCHF oscillator is disabled
1	The XOSCHF oscillator is enabled, and overrides normal port operation for the respective oscillator pins

AVR Register Macro Suffix Sheet

Suffix	Meaning	Purpose	Example
_bm	Bit Mask	Used to set, clear, or check a specific bit in a register	CLKCTRL_ENABLE_bm → 0x01
_bp	Bit Position	Indicates the bit position (used for shifting)	CLKCTRL_ENABLE_bp → 0
_gc	Group Constant	Represents a predefined value for a register field (often enums)	CCP_IOREG_gc → 0xD8
_gm	Group Mask	Bit mask for a multi-bit field (used with _gc)	TCA_SINGLE_CLKSEL_gm → 0x07
_val	Value Constant	Used in some libraries to represent a raw value	USART_BAUD_9600_val
_enum	Enumeration Type	Used in some toolchains to define enum types for register fields	PORT_PULLUP_enum

Internal oscillator vs External crystal

◆ Internal Oscillator

- **Built-in** to the microcontroller.
- Common types: RC oscillator, calibrated oscillator.
- **Typical frequency**: 1 MHz, 8 MHz, 20 MHz (varies by MCU).
- **Pros**:
 - No external components needed.
 - Faster startup time.
 - Lower cost and simpler design.
- **Cons**:
 - **Lower accuracy** ($\pm 1\%$ to $\pm 10\%$ depending on calibration).
 - **More sensitive** to temperature and voltage changes.
 - Not ideal for precise timing (e.g., UART baud rates, real-time clocks).

◆ External Crystal Oscillator

- Requires an external **quartz crystal** or **ceramic resonator**.
- **Typical frequency**: 4 MHz, 8 MHz, 16 MHz, etc.
- **Pros**:
 - **High accuracy** (± 50 ppm or better).
 - **Stable** across temperature and voltage.
 - Ideal for **precise timing**, communication protocols, and clocks.
- **Cons**:
 - Requires external components (crystal + capacitors).
 - Slightly longer startup time.
 - Higher cost and board complexity.

Bit Masking Operations

- Bit masking operations allow us to modify a single bit in a register
- Let's say you want to modify bit i in a register called `BYTE` = `0b01100000`
- To Set i^{th} bit
 - `BYTE |= (1 << i);` or `BYTE |= 0b00010000;`
 - E.g. if $i = 4$ then
`BYTE |= (1 << i)` → `BYTE = 0b01100000 | 0b00010000 = 0b01110000`
- To Clear i^{th} bit
 - `BYTE &= ~(1 << i);` or `BYTE &= 0b10111111;`
 - E.g. if $i = 6$ then
`BYTE &= ~(1 << i)` → `BYTE = 0b01110000 & ~(0b01000000)`
→ `BYTE = 0b01110000 & 0b10111111 = 0b00110000`
- To Toggle i^{th} bit
 - `BYTE ^= (1 << i);` or `BYTE ^= 0b00000010;`
 - E.g. if $i = 1$ then
`BYTE ^= (1 << i)` → `BYTE = 0b00110000 ^ 0b00000010 = 0b00110010`

Bit Masking Macro Operations

```
1 #define SET_BIT(byte, i)    ((byte) |= (1 << (i)))
2 #define CLEAR_BIT(byte, i) ((byte) &= ~(1 << (i)))
3 #define TOGGLE_BIT(byte, i) ((byte) ^= (1 << (i)))
4 #define CHECK_BIT(byte, i) (((byte) >> (i)) & 1)

1 uint8_t BYTE = 0b01100000;
2
3 SET_BIT(BYTE, 4);    // BYTE becomes 0b01110000
4 CLEAR_BIT(BYTE, 6);  // BYTE becomes 0b00110000
5 TOGGLE_BIT(BYTE, 1); // BYTE becomes 0b00110010

1 if (CHECK_BIT(BYTE, 4)) {
2     // Bit 4 is set
3 }

1 // Set PA4 as output
2 SET_BIT(PORTA.DIR, 4);
3
4 // Set PA4 high
5 SET_BIT(PORTA.OUT, 4);
6
7 // Clear PA4 (set low)
8 CLEAR_BIT(PORTA.OUT, 4);
9
10 // Toggle PA4
11 TOGGLE_BIT(PORTA.OUT, 4);
12
13 // Check if PA4 is high
14 if (CHECK_BIT(PORTA.IN, 4)) {
15     // PA4 is high
```

Task 1: Blinking an LED

- Blink a single LED at two different rates
 - The LED should blink at 2Hz frequency and 4Hz and switch between the two frequencies every 2 seconds.
- The blinking duty cycle should be 50%
 - I.e. for 2Hz frequency, the LED should be on for $1/4^{\text{th}}$ of a second, then off for next $1/4^{\text{th}}$ of a second.
 - The duty cycle is defined as:
 - $\text{Duty Cycle (\%)} = (\text{High Period Time}) / (\text{Total Time Period}) \times 100 (\%)$
 - Time High: Duration when the signal is ON (logic high).
 - Total Period: One complete cycle of the waveform (ON + OFF time).
 - Example
 - If a PWM signal is high for 2 ms and low for 8 ms:
 - Total period = 2 ms + 8 ms = 10 ms
 - Duty cycle = $2 / (2 + 8) \times 100 = 20\%$
 - This means the signal is ON for 20% of the time and OFF for 80%.

Task 2: Blinking 8 LEDs

Extend Task1 to do the blinking on all 8 LEDs one after another.

- When the system starts, LED 0 is active and blinks at 2Hz.
- After two seconds, it will blink at 4Hz.
- After two more seconds, the blinking LED will go back to 2Hz and also shift to LED 1
- This will continue
 - Every two seconds, the frequency will flip between 2 and 4 Hz.
 - Every four seconds, the LED will shift to the left. When the active LED reaches the left-most position (LED 7) it should start from 0 again.
- You can do this with nested for loops that do the appropriate number of blinks at each frequency
- Or, without nested loops, you need to
 - Keep track of the current frequency
 - Keep track of the LED bit position
 - Keep track of the second or millisecond count so you know when to switch frequency of position. Update this count after you blink the LEDs