ECE3411 – Spring 2019

Lecture#2

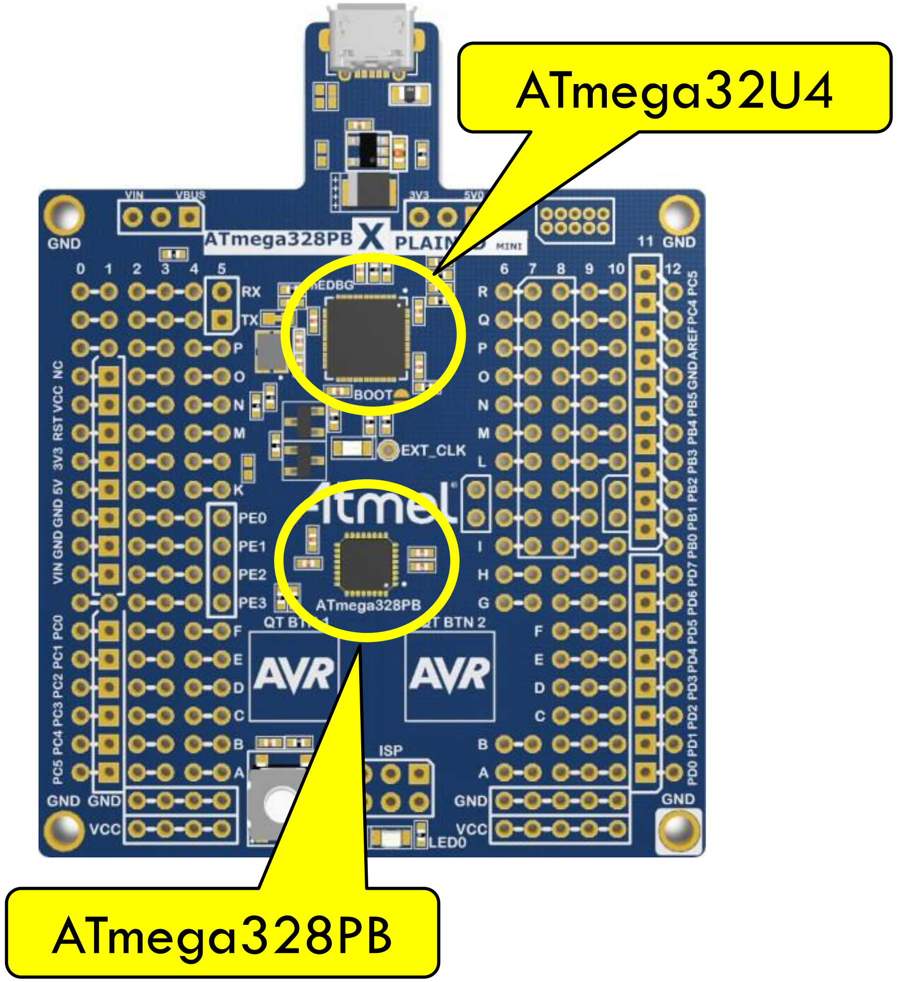
General Purpose Digital Output

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

Atmega328PB Xplained Mini Kit

- The ATmega328PB Xplained Mini evaluation board provides a development platform for the Atmel ATmega328PB Microcontroller.
- Target Microcontroller: ATmega328PB
- On-board Programming & Debugging capability using Atmel Studio
  - Programmer Microcontroller: ATmega32U4
- USB connectivity
- Headers & Connectors for accessing target microcontroller's I/O pins



ATmega328PB Features (1)

- High Performance, Low Power Atmel®AVR® 8-Bit Microcontroller
- Advanced RISC Architecture
  - 131 Powerful Instructions – Most Single Clock Cycle Execution
  - 32 x 8 General Purpose Working Registers
  - Fully Static Operation
  - Up to 20 MIPS Throughput at 20MHz
  - On-chip 2-cycle Multiplier
- High Endurance Non-volatile Memory Segments
  - 32KBytes of In-System Self-Programmable Flash program memory
  - 1K Byte EEPROM
  - 2K Bytes Internal SRAM
  - Write/Erase Cycles: 10,000 Flash/100,000 EEPROM
  - Data retention: 20 years at 85°C/100 years at 25°C
  - Optional Boot Code Section with Independent Lock Bits
    - In-System Programming by On-chip Boot Program
    - True Read-While-Write Operation
  - Programming Lock for Software Security

ATmega328PB Features (2)

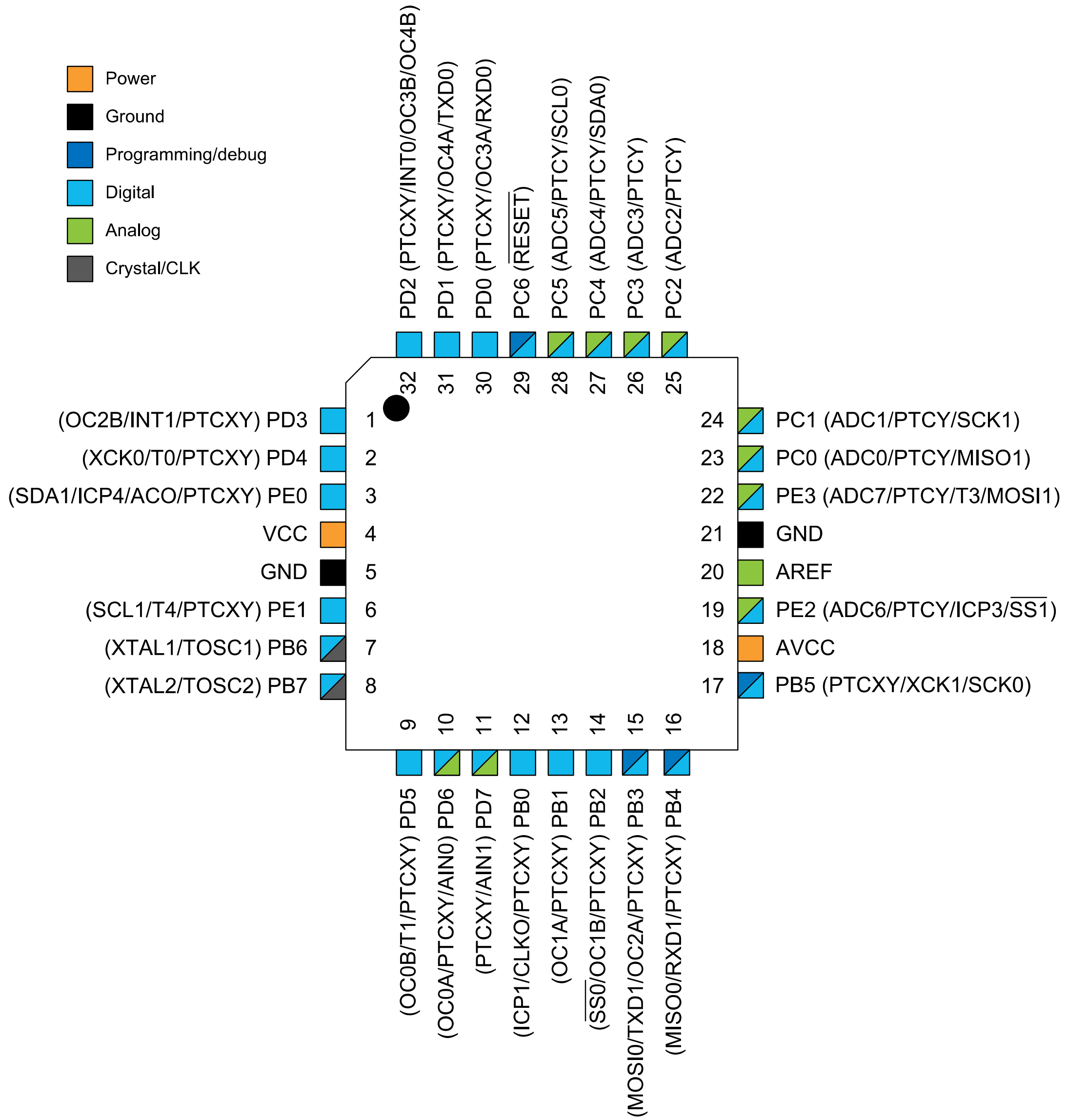
- Peripheral Features
  - Peripheral Touch Controller
  - Two 8-bit Timer/Counters with Separate Prescaler and Compare Mode
  - Three 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture Mode
  - Real Time Counter with Separate Oscillator
  - Ten PWM Channels
  - 8-channel 10-bit ADC with Temperature Measurement
  - Two Programmable Serial USARTs
  - Two Master/Slave SPI Serial Interfaces
  - Two Byte-oriented 2-wire Serial Interfaces (Phillips I2C compatible)
  - Programmable Watchdog Timer with Separate On-chip Oscillator
  - On-chip Analog Comparator
  - Interrupt and Wake-up on Pin Change

ATmega328PB Features (3)

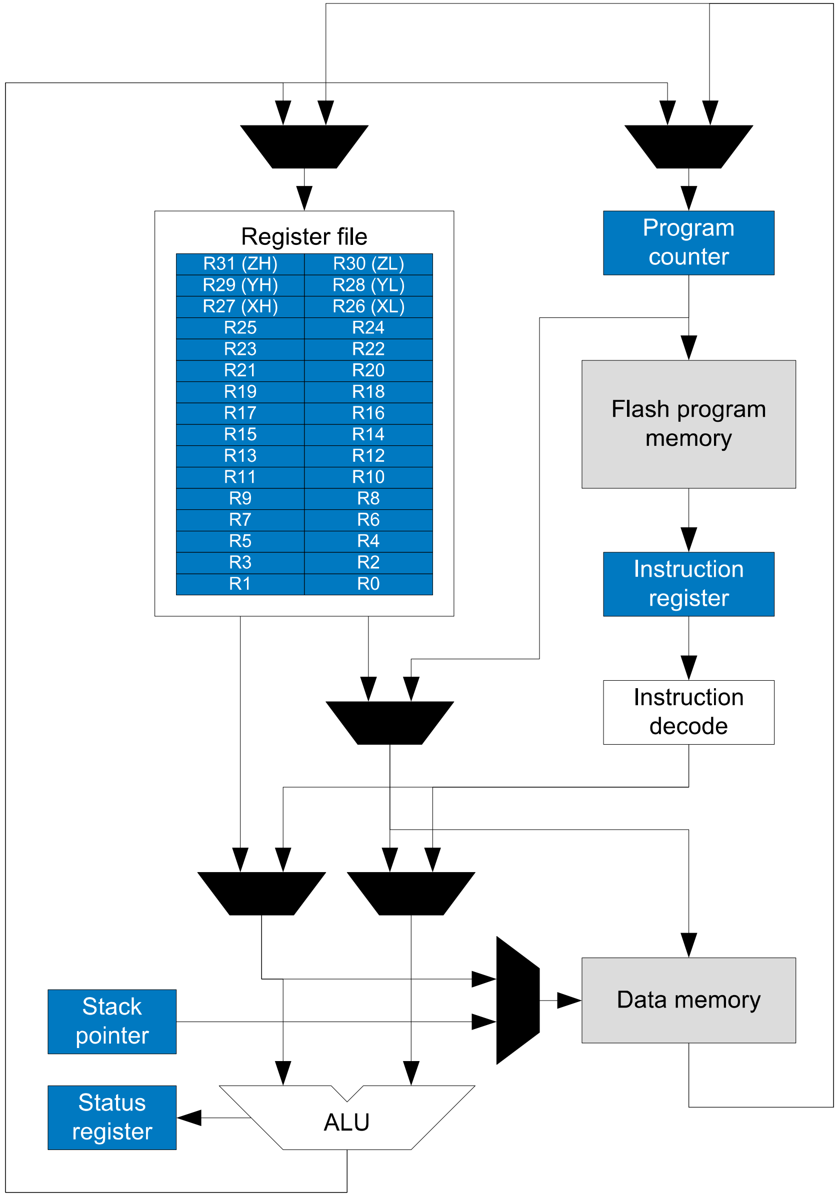
- Special Microcontroller Features
  - Power-on Reset and Programmable Brown-out Detection
  - Internal Calibrated Oscillator
  - External and Internal Interrupt Sources
  - Six Sleep Modes: Idle, ADC Noise Reduction, Power-save, Power-down, Standby, and Extended Standby
  - Unique Device ID
- I/O and Packages
  - 27 Programmable I/O Lines
  - 32-pin TQFP and 32-pad QFN/MLF
- Operating Voltage: 1.8 - 5.5V
- Temperature Range: -40°C to 105°C
- Speed Grade: 0 - 20MHz @ 1.8 - 5.5V
- Power Consumption at 1MHz, 1.8V, 25°C
  - Active Mode: 0.24 mA
  - Power-down Mode: 0.2µA
  - Power-save Mode: 1.3 µA (Including 32kHz RTC)

ATmega328PB Package

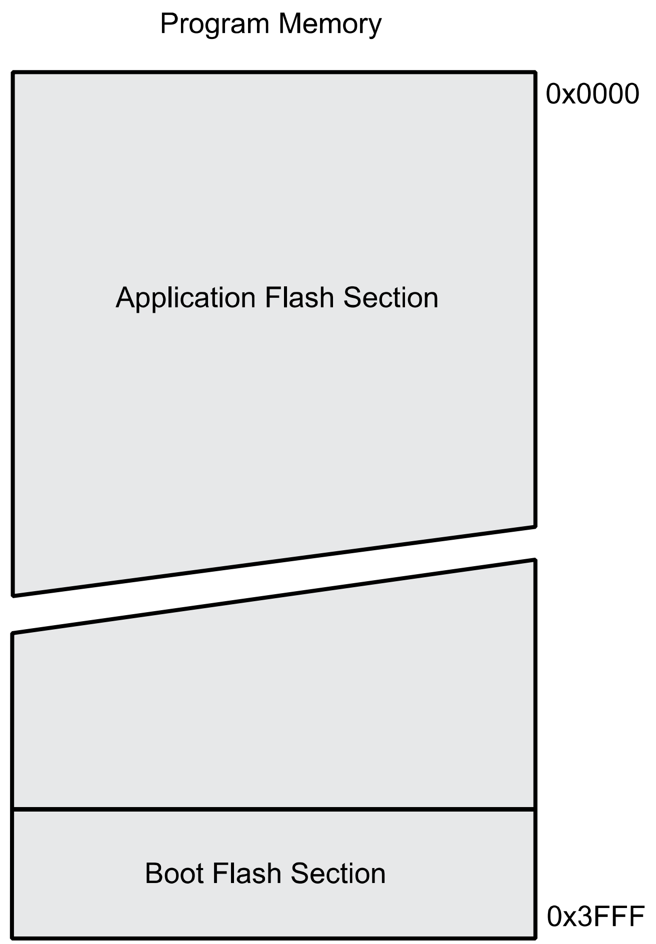
Thin Profile Plastic Quad Flat Package (TQFP)

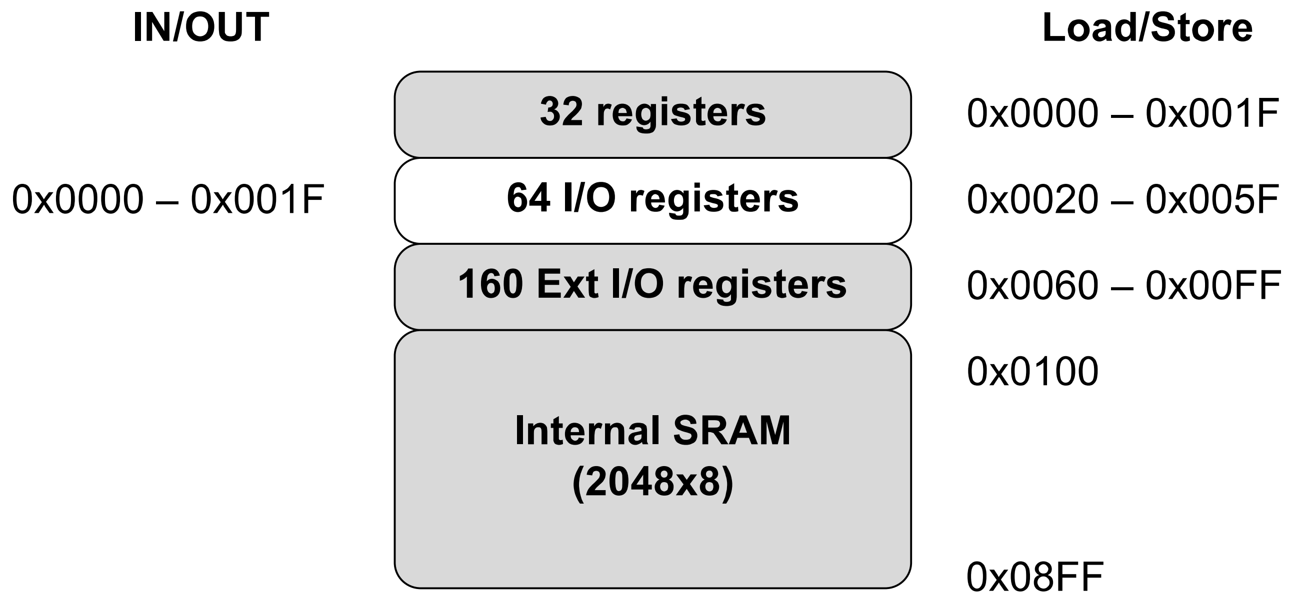


ATmega328PB Architecture



ATmega328PB Architecture





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

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0

- **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. PORTB, PORTC, PORTD etc.

Hardware Registers of a Port

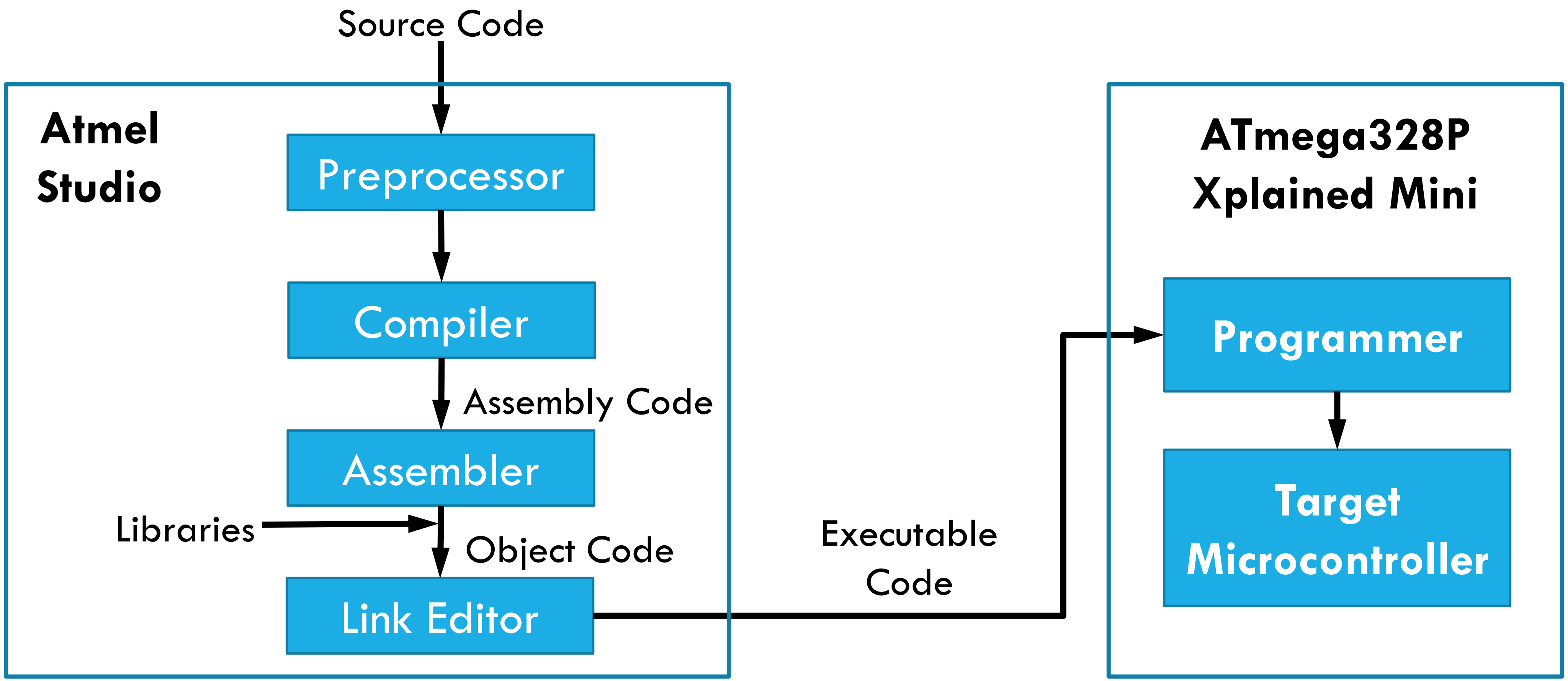
Each Port on the Mega AVRs has three hardware registers associated to it:

- **DDRx** : *Data-Direction Register for Port x*
  - 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 DDRx.
- **PORTx** : *Port x Data Register*
  - When the DDRx bits are set to ‘1’ (output) for a given pin, the PORT register controls whether that pin is set to logic high or low.
  - E.g. writing a ‘1’ to a bit position in PORT register will produce VCC voltage at that pin & vice versa.
- **PINx** : *Port x Input Address*
  - The PIN register addresses are used to read the digital voltage values for each pin that’s configured as input.
  - E.g. a value ‘0’ of a bit of PIN register indicates a low voltage at that pin & vice versa.

Examples of Predefined Registers

- AVR library has some predefined register names for each port.
  - E.g. for Port B, the registers are **DDRB**, **PORTB**, and **PINB**
- 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 PINx register)
- AVR library also has predefined keywords for each bit position of each port register
  - E.g. for 7^{th} bit position of PINB register, the predefined keyword is **PINB7**
  - Similarly **PORTB5** represents 5^{th} bit position of PORTB register
- Notice that the keywords for bit positions are constants
  - They simply define the bit number, not the bit value. E.g. you can't do PORTB5 = 5
  - These keywords are read-only, you cannot write any value to them.

AVR Software Development Process



ATmega328P Header file snippet

```
#define  PINB    _SFR_IO8(0x03)
#define  PINB0   0
#define  PINB1   1
#define  PINB2   2
#define  PINB3   3
#define  PINB4   4
#define  PINB5   5
#define  PINB6   6
#define  PINB7   7
```

```
#define  DDRB    _SFR_IO8(0x04)
#define  DDB0    0
#define  DDB1    1
#define  DDB2    2
#define  DDB3    3
#define  DDB4    4
#define  DDB5    5
#define  DDB6    6
#define  DDB7    7
```

```
#define  PORTB   _SFR_IO8(0x05)
#define  PORTB0  0
#define  PORTB1  1
#define  PORTB2  2
#define  PORTB3  3
#define  PORTB4  4
#define  PORTB5  5
#define  PORTB6  6
#define  PORTB7  7
```

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).

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
- Xplained Mini kit runs the ATmega328PB on 16MHz frequency
  - #define F_CPU 16000000UL is included in the code to define the frequency for the delay library
- Only use delay functionality in order to define access functionality for e.g. LCD screen which requires precise timing sequences:
  - Never use delay functionality in your main program
  - We want to do other useful computation while waiting

Test Program to Blink LED

```c
// ------- Preamble --------- //
#define F_CPU 16000000UL   /* 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) {
    // -------- Inits --------- //
    /* Data Direction Register B: writing a one to the bit enables output. */
       DDRB |= (1 << DDB5);

    // ------ Event loop ------ //
    while (1) {
        PORTB = 0b00100000;  /* Turn on the LED bit/pin in PORTB */
        _delay_ms(1000);          /* wait for 1 second */
        PORTB = 0b00000000;  /* Turn off all B pins, including LED */
        _delay_ms(1000);          /* wait for 1 second */
    } /* End event loop */
    return (0); /* This line is never reached */
}
```

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`

Test Code

```c
/* ------- Preamble --------- */
#define F_CPU 16000000UL   // 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) {
    /* -------- Inits --------- */
    /* Data Direction Register D: Setting Port D as output. */
    DDRD = 0b11111111;

    /* ------ Event loop ------ */
    while (1) {
        PORTD = 0b01010101;  // Turn on alternate LEDs in PORTD
        _delay_ms(1000);     // wait for 1 second
        PORTD = 0b10101010;  // Toggle the LEDs
        _delay_ms(1000);     // wait for 1 second
    }
    /* End event loop */

    return (0); /* This line is never reached */
}
```

Task 1: Blinking a single LED

- Blink a single LED at two different rates based on a push switch.
  - When the switch is not pressed, LED should blink at 2Hz frequency.
  - As long as the switch is pressed, LED should blink at 8Hz frequency.
- The blinking duty cycle should be 50%
  - E.g. for 2Hz frequency, the LED should be on for $1/4^{th}$ of a second, then off for next $1/4^{th}$ of a second and so on.
- You may use the on-board LED and push button for this task.

Task 2: Blinking 8 LEDs one after another

Extend the Task1 with another switch which activates the blinking to loop through all 8 LEDs one after another.

- When the system starts, LED 0 is active and blinks at 2Hz.
- As long as switch 1 is pressed, the currently active LED blinks at 8Hz. Otherwise it blinks at 2Hz.
- As long as switch 2 is pressed, the currently active LED keeps shifting towards left at the frequency depending upon the position of switch 1, and starts from 0 again.
  - E.g. if LED 0 is active currently, pressing switch 2 shifts the blinking to LED 1, 2, 3, … , 7 and then again LED 0 and so on.
- When switch 2 is released, the last active LED should keep blinking without anymore shifting.