Shifting & Masking

A core technique

Problem

- `DDRB = 1;`
- Sets PB0 to be an output
- “Accidentally” sets PB1->PB7 to be inputs.
- What if PB3 is connected to an LED?

A more detailed look

- `DDRB = 1;`
- `DDRB = 0b00000001;`
- `DDRB = 0x01;`

How to change one bit?

- Concept:
  - Copy DDRB somewhere else – e.g. a variable
  - Change one bit
  - Write back to DDRB
- Copy-Modify-Write.

Copy

- Easy.
- `ddrbCopy = DDRB;`
- ddrbCopy is now a copy of DDBR.

- Why work with a copy?

Copy

- Not so easy?

-
  ```
  unsigned char ddrbCopy;
  ddrbCopy = DDRB;
  ```

- Need to know data sizes.

Why do we need to know data sizes?

- 8/16/32/64 bit systems
- Defines “size” of number CPU can deal with in one operation.
- 8 bit system
  - adds/subtracts/multiplies/divides numbers up to bits.
  - Not limited to 8^2 = 256!
  - Just limited to 8 bit chunks...
- Just has to deal with bigger numbers in 8-bit chunks doing carries etc to get the answer.
- 8 bit system needs FOUR add instructions to add TWO 32 bit numbers
- 32 bit systems only needs ONE. Four times faster?

Data Sizes

- What size is a char?
  - 1 byte (always)

- What size is an int?
  - Depends on your compiler
  - `unsigned int x;`
  - How big is x?

Data sizes

- In embedded programming we very often need to know exactly what size or variables are
- We know DDRB is 8 bits so to make a copy of it we would need an 8 bit variable
- Hence
  - `unsigned char ddrbCopy;`
  - A char is just a small integer
    - Often used to store an ASCII code that's why it is called a “char”... but it is really a number!

What if I need a 16 bit variable?

- `unsigned int y; //????`
- Might be 16 bits... need to check compiler documentation – might be 32 bits...
- A better way?

Data sizes

- What size is an int in C?
- Only one fixed type size original C – C89. char is one byte.
- C language standards
  - C89 original
  - C99
  - C11 (latest)
- C99 specified the following typedefs to be define in stdint.h
  - 8-bit: int8_t, uint8_t
  - 16-bit: int16_t, uint16_t
  - 32-bit: int32_t, uint32_t
  - 64-bit: int64_t, uint64_t

Data size ranges

uint8_t	0->255 (unsigned)
int8_t	-128->127 (signed)
uint16_t	0->65,535
int16_t	-32,768->32,767
uint32_t	0 ->4,294,967,295
int32_t	-2,147,483,648->2,147,483,647

Using new data defines

- `uint8_t x;`
  - x is 8 bits – replaced with `unsigned char x;` by compiler.
- `uint16_t y;`
  - y is 16 bits – replaced with appropriate type by compiler e.g. unsigned int y; or unsigned short int y;
  - Depends on your compiler but now guaranteed y is 16 bits.
-
- Same for uint32_t and signed versions int8_t, int16_t, etc
- This is a C99 feature.
  - May have to #include <stdint.h>

Back to Copy...

-
  ```
  uint8_t ddrbCopy;
  ddrbCopy = DDRB;
  ```

- ddrbCopy is 8 bits- same as DDRB
- It is unsigned – more in later lecture...

Modify?

- Concept:
  - Create a “mask”
  - Apply mask to copy
  - Mask will only affect the bit we're interested in.

What is a mask?

- A mask is a byte (in this case) with a “1” in the bit position(s) we are interested in.
- “0” everywhere else.

- Mask for bit position 0 is:
  - 0b00000001
  - 0x01
- Mask for bit position 3 is:
  - 0b00001000
  - 0x08
- Mask for bit position 5?

Applying the mask

- DDRB:
  - XXXXXXX?
  - ? represents bit we are interested in.
  - X represents don't care / don't touch.
- Mask:
  - 00000001

Applying the mask

- `XXXXXXX?  (ddrbCopy)`
- `00000001  (mask)`

- Add? Subtract? Multiply? Divide?

- AND, OR, XOR?

ORing sets a bit

```
XXXXXXX?
00000001
XXXXXXX1
```

- `ddrbCopy = ddrbCopy | 0x01;`

```
11110000
00000001
11110001
```

```
11110001
00000001
11110001
```

Set bit 0

- `ddrbCopy = DDRB;`
- `mask = 0x01;`
- `ddrbCopy = ddrbCopy | mask;`
- `DDRB = ddrbCopy;`

Set bit 0

- `ddrbCopy = DDRB;`
- `ddrbCopy = ddrbCopy | 0x01;`
- `DDRB = ddrbCopy;`

Set bit 0

- `ddrbCopy = DDRB;`
- `DDRB = ddrbCopy | 0x01;`

Set bit 0

- `DDRB = DDRB | 0x01;`

Shorthand

- Remember in C:

```
x = x + 2;
x += 2;

x = x – 3;
x –= 3;

x = x * 4;
x *= 4;
```

Applying shorthand

- `DDRB = DDRB | 0x01;`

- `DDRB |= 0x01;`

Making the mask

- Error-prone?
  - Bit 5
  - Hex?
  - 0x20;
  - Bit 3?
  - Bit 6?

Making the mask

- `Bit 0 – 0b00000001 – 0x01`
- `Bit 1 – 0b00000010 – 0x02`
- `Bit 2 – 0b00000100 – 0x04`
- `Bit 3 – 0b00001000 – 0x08`
- `Bit 4 – 0b00010000 – 0x10`
- `Bit 5 – 0b00100000 – 0x20`
- `Bit 6 – 0b01000000 – 0x40`
- `Bit 7 – 0b10000000 – 0x80`

A better way

- Bit 0 – 0b00000001 – (1<<0)
- Bit 1 – 0b00000010 – (1<<1)
- Bit 2 – 0b00000100 – (1<<2)
- Bit 3 – 0b00001000 – (1<<3)
- Bit 4 – 0b00010000 – (1<<4)
- Bit 5 – 0b00100000 – (1<<5)
- Bit 6 – 0b01000000 – (1<<6)
- Bit 7 – 0b10000000 – (1<<7)

Copy-Modify-Write

- `DDRB |= 0x01;`
- `DDBR |= (1<<0);`

- `DDRB |= 0x40;`
- `DDRB |= (1<<5);`

- Set bit 6 of PORTB register without affecting any other bits on PORTB?

Set bit y in any register?

- Set bit y of REGX?

- `REGX |= (1<<y);`

- This technique is called shifting & masking.

Clearing a bit?

- `XXXXXXX?   (ddrbCopy)`
- `00000001   (mask)`

- Add? Subtract? Multiply? Divide?

- AND, OR, XOR?

ANDing clears a bit

```
XXXXXXX?
00000001
0000000?
```

- Need to invert mask.

ANDing clears a bit

```
XXXXXXX?
11111110
XXXXXXX0
```

- `ddrbCopy = ddrbCopy & 0xFE;`

```
11110001
11111110
11110000
```

```
11110000
11111110
11110000
```

Clear bit 0

- `ddrbCopy = DDRB;`
- `mask = 0xFE;`
- `ddrbCopy = ddrbCopy & mask;`
- `DDRB = ddrbCopy;`

Clear bit 0

- `ddrbCopy = DDRB;`
- `ddrbCopy = ddrbCopy & 0xFE;`
- `DDRB = ddrbCopy;`

Clear bit 0

- `ddrbCopy = DDRB;`
- `DDRB = ddrbCopy & 0xFE;`

Clear bit 0

- `DDRB = DDRB & 0xFE;`

Remember shorthand?

- `DDRB &= 0xFE;`

Making the mask

- Error-prone?
  - Bit 5
  - Hex?
  - 0b11011111
  - 0xDF
  - Bit 3?
  - Bit 6?

Making the mask

- Bit 0 – 0b11111110 – 0xFE
- Bit 1 – 0b11111101 – 0xFD
- Bit 2 – 0b11111011 – 0xFB
- Bit 3 – 0b11110111 – 0xF7
- Bit 4 – 0b11101111 – 0xEF
- Bit 5 – 0b11011111 – 0xDF
- Bit 6 – 0b10111111 – 0xBF
- Bit 7 – 0b01111111 – 0x7F

A better way

- `Bit 0 – 0b11111110 – ~(1<<0)`
- `Bit 1 – 0b11111101 – ~(1<<1)`
- `Bit 2 – 0b11111011 – ~(1<<2)`
- `Bit 3 – 0b11110111 – ~(1<<3)`
- `Bit 4 – 0b11101111 – ~(1<<4)`
- `Bit 5 – 0b11011111 – ~(1<<5)`
- `Bit 6 – 0b10111111 – ~(1<<6)`
- `Bit 7 – 0b01111111 – ~(1<<7)`

Why ~(1<<y)

- Can't shift a zero.
- (0<<3)
  - 0 = 0b00000000
  - (0<<3) = 0b00000000
- Instead shift a “1” and invert all bits
  - 1 = 0b00000001
  - (1<<3) = 0b00001000
  - ~(1<<3) = 0b11110111

Clearing a bit

- `DDRB &= 0xFE;`
- `DDRB &= ~(1<<0);`

- `DDRB &= 0xD0;`
- `DDRB &= ~(1<<5);`

- Clear bit 6 of PORTB register without affecting any other bits on PORTB?

Clear bit y in any register?

- Clear bit y of REGX?
- `REGX &= ~(1<<y);`
- This technique is called shifting & masking.

XORing toggles a bit

$$\begin{array}{r} 1\,1\,1\,1\,0\,0\,0\,1 \\ \underline{0\,0\,0\,0\,0\,0\,0\,1} \\ 1\,1\,1\,1\,0\,0\,0\,0 \end{array}$$

$$\begin{array}{r} 1\,1\,1\,1\,0\,0\,0\,0 \\ \underline{0\,0\,0\,0\,0\,0\,0\,1} \\ 1\,1\,1\,1\,0\,0\,0\,1 \end{array}$$

- PORTB ^= (1<<5); //toggle PB5

Shifting and masking summary

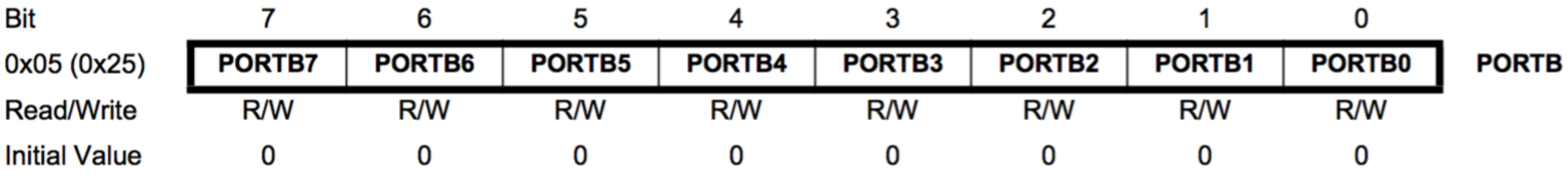
- ORing sets a bit
  - `PORTB |= (1<<5); //make bit 5 a 1`

- ANDing clears a bit
  - `PORTB &= ~(1<<5);//clears bit 5 to a zero`

- XORing toggles a bit
  - `PORTB ^= (1<<5); //toggles bit 5`

- All of the above only affect the selected bit
  - Copy-modify-write

Using datasheet bit names

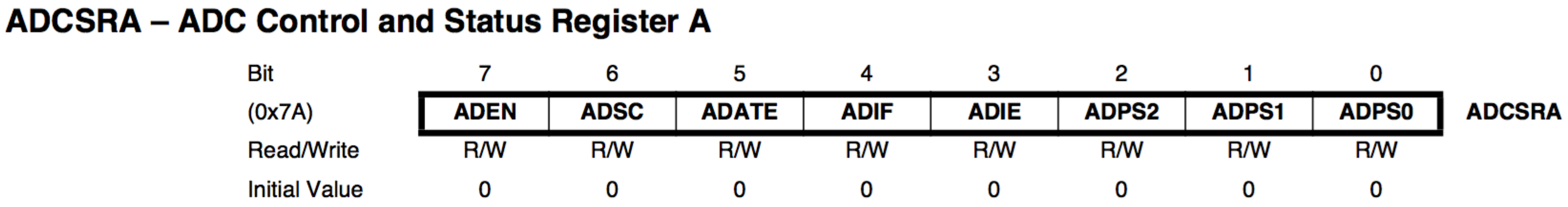


Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- All bits in registers have names – see datasheet
- So can do:
  - `PORTB |= (1<<5);`
    - Or else
  - `PORTB |= (1<<PORTB5);`

- Not a big deal when using I/O ports and names match bit positions – e.g. PORTB4 is bit 4... but...

Using datasheet bit names



ADCSRA – ADC Control and Status Register A

Bit	7	6	5	4	3	2	1	0	
(0x7A)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Is very useful when bit position has nothing to do with functionality of bit e.g.
  - To get the ADC to do a conversion you have to write a 1 into the ADSC (AD Start Conversion) bit in the ADCSRA register.

Using datasheet bit names

- A pain to have to remember that is bit 6
  - `ADCSRA |= (1<<6);`
  - Will you remember what bit 6 is next week, month? Code needs a comment to help you whereas:

- `ADCSRA |= (1<<ADSC);`
  - Better chance of remembering what this does next week/month...
  - Self documenting code

- Also less chance of making a mistake -
  - `ADCSRA |= (1<<5); //Start the ADC conversion.....`
  - `ADCSRA |= (1<<ADSC);`

Check if a PIN is high or low?

-
  ```
  if(PINB & (1<<3))
  {
      //PB3 is high
  }
  else
  {
      //PB3 is low
  }
  ```