3.9.1 Polling

Polling, interrups and timers allow us to manage reactions to external event.

- Polling: actively check for an event to happen
- Interrupt: hardware support for an external event to tell us to react to something
- Timer: special type of interrupt – set a clock to do something after program-specified time

Polling
Actively check for an event:

- Relatively simple
- Gives tight control
- Often *very* inefficient
- Often inelegant solution

Polling example
A signal comes in on PINC0 – when it goes from high to low do some operation

```
wait: IN R17, PINC
      ANDI R17, 1
      BREQ wait
      ...
```

Since we read in all 8-bits from PINC and are only intersted in the least significant bit, we *mask* the top 7 bits by doing an *AND* with 1 (0b00000001). This keeps the LSB with the bit we read in and puts 0 elsewhere.

An alternative is:

```
wait: SBIC PINC, 0
      JMP wait
      ...
```

Another approach – read data whenever a polling line changes value

Example: Read into C whenever B0 changes

```
   CBI DDRB, 0
   CLR R0  ; prev B0 value
   OUT DDRC, R0
   LDI R16, 0x01

poll:
   IN R1, PINB
   AND R1, R16
   XOR R1, R0 ; change?
   BREQ poll
   XOR R0, R16 ; flip R0
   IN R2, PINC
   ...
   RJMP poll
```

Critique

- Simple case simple
- Inefficient because will be very very busy doing nothing – can't do anything else, power consumed
- Monitoring multiple events more difficult

3.9.2 External interrupts

Interrupt

Hardware support for asynchronous notification of event
Analogy: someone coming to see me

- *polling* – I get up every 15s to see if they're at the door;
- *interrupt* – I do other things, they press the door bell when they arrive.

Principles of interrupt common across different architectures, but details differ.

You will need to check the data sheet of your device – even within AVR family this changes.

Principles

- Interrupts can be enabled or not;
- Usually multiple interrupts – can be externally or internally set;
- Associate actions with each interrupt

- Program executes “normally” — but when interrrupt is signalled
  - action associated with interrupt executes
  - when complete control returns to “normal” flow.

There are many different interrupts supported by the AVR. We’ll only look at few.

Interrupt Vector Table

Key to managing the interrupts is the the IVT – table in memory

- Each interrupt has a location associated in the IVT – associated code *interrupt handler* is there.
- When interrupt occurs,
  - current instruction completes
  - current address pushed on stack
  - control jumps to location in IVT

    each interrupt only has 2 machine words associated with it, so usually, this is just a JMP somehere
  - RETI terminates

    when this instruction is executed, return address popped off stack, resetting the PC and control returns to normal flow.

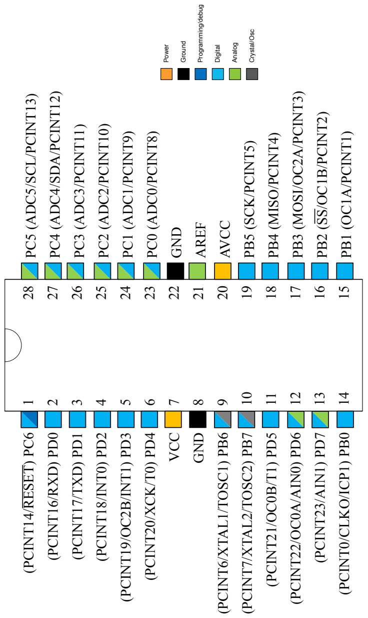
Structure of Interrupt Vector Table

Top part of *program* memory.

- Need to look at chip data sheet for specifics
- If you don’t use interrupts, can use it for “normal” code

1	0x0000	RESET
2	0x0002	INT0
3	0x0004	INT1
4	0x0006	PCINT0
5	0x0008	PCINT1
. . .	. . .	. . .

Recap of external packaging



Many pins have alternative uses – you as programmer decide which use it has.

External interrupts on AVR

Two general interrupts INT0, INT1

- INT0: PIND2
- INT1: PIND3

Pin change interrupts:

- PCINT0 : PCINT[7:0] toggles
- PCINT1 : PCINT[14:8] toggles
- PCINT2 : PCINT[23:16] toggles

The external pin controls whether an interrupt signals – we have control how

- NB: trade-off – if you use a PIN for an interrupt you can't use it for I/O
- This choice dynamic in execution of the program

Controlling interrupt

To use an interrupt:

- IVT: set instruction in the appropriate IVT location (typically a JMP)
- *Globally enable interrupt*: The I flag in the SREG controls whether interrupts are supported at all

  SEI/CLI

- Enable specific interrupt
- Determine mode of interrupt (low, high, edge)

Different interrupts have different capabilities.

Using INT0 and INT1

External Interrupt Mask Register

D7 (high bit) Low bit: D0

						INT1	INT0

Top enable or disable an interrupt, the relevant bit on the EIMR must be set. EIMR is an I/O register and so, the OUT command is executed. In the example, code below four alternatives of setting the register are shown (in any program you would use one of them only!!). The first disables interrupts, the first enables INT0 but disables INT1, the second enables INT1 but disables INT0, and the third enables both.

There are other possibilities – for example, using other command sequences you could specifically modify one bit and leave the other bits unchanged.

Enabling an interrupt

```
LDI R16, 0
OUT EIMSK, R16
```

INT0, INT1 not enabled

Enabling an interrupt

```
LDI R16, 0b00000001
OUT EIMSK, R16
```

INT0 enabled, INT1 not

Enabling an interrupt

```
LDI R16, 0b00000010
OUT EIMSK, R16
```

INT0 not enabled, INT1 is enabled

Enabling an interrupt

```
LDI R16, 0b00000011
OUT EIMSK, R16
```

Both INT0, INT1 is enabled

The problem with this code above is that it is not obvious what the values loaded into R16 mean. Better would be to use constants *int0* etc defined by the *.include* that you do. For the 328P, int0 and int1 are 0 and 1 respectively, but by using the constants you make your code more readable and more portable – you can just change the .include and your code will work on a different architecture.

Alternatives for specifying mask

```
LDI R16, (1<<int0)
LDI R16, (1<<int1)
LDI R16, (1<<int0)|(1<<int1)
```

Here there we are using the bit shift operator `<<` (as in C++ — if you are not sure read up about it). This is a pseudo-instruction and is computed at build time by the Atmel Studio. This code has the advantage of being clear about the purpose of what we are trying to do and in general more portable.

For example, on the 328P, the value of int0 is 0 and the value of int1 is 1. Then we have

- `1<<int0` is the same as `1<<0` which is just `1`

  Actuall it's `000000001`

- `1<<int1` is the same as `0b0000001<<1` which is `0b00000010`

- `(1<<int0)|(1<<int1)` is

  `000000001 | 00000010` which is `0b00000011`

Triggering an interrupt

What event on the pin makes an interrupt trigger?

- low-level trigger:

  the interrupt is signalled when the pin is low. This is the default for INT0 and INT1

- high-level trigger:

  interrupt is signalled when pin high. INT0 and 1 can be programmed to support this.

- edge triggered: when the input signal changes

The mode of when the trigger is set is determined by modifying the External Interrupt Control Register (EICRA) register. This is not an I/O register – you need to use STS to set the value.

EICRA

D7 (high bit) Low bit: D0

				ISC11	ISC10	ISC01	ISC00

The bit positions (e.g., ISC10) are available as constants. Simlarly, the registers like I.

INT0		Action
ISC01	ISC00	
0	0	Low INT0 triggers
0	1	Any INT0 change triggers
1	0	Falling INT0 triggers
1	1	Rising INT0 triggers

Same rules for INT1 with ISC11, ISC10

- Edge triggered interrupts: pulses should be at least one cycle in length;
- Level-triggered interrupts: puleses shoud be at least five cycles in length

Checklist for using interrupts

1. Know what you want to do!
2. Write interrupt handler.
3. Set up entry in the IVT
4. Globally enable interrupts
5. Enable specific interrupt
6. Choose mode of interrupt

Here's a simple example: we have an interrupt hander that periodically is told that a temperature reading is available. Based on the temperature is adjusts a fan value and and writes the desired fan value to a port.

First, the outline of the code

```
.include "m168pdef.inc"
.DSEG
fan: .BYTE 1
.CSEG
.ORG 0
        jmp main
.ORG 0x02  ; interrupt 0
        jmp check_temp

.org 0x200
main:
     ....
     ....

check_temp:
     ....
     ....
```

Now we can fill in the details. But there are some important points to pay attention to:

- The interrupt handler must take care that it protects the main code from any changes that it makes. Remember an interrupt can happen at point in execution and if the interrupt hander changes a register for example, the main code will get very confused. In particular, in most cases the interrupt handler will need to protect the status register! This is usually done by pushing any registers that they need and popping them afterwards.
- When an interrupt happens the global interrupt enable flag is cleared to prevent interrupts within interrupts and then when you do a RETI, it will automatically be reenabled. If you want to support interrupts within interrupts you can do so, but you need to take care and specifically enable the interrupt flag. Read about this if you want more.

```
main:
        clr R16
        sts fan, R16
        out ddrc, R16
        inc r16
        out ddrb, R16
        ldi R16, (1<<ISC01)|(1<<ISC00)
        sts eicra, r16
        ldi r16, (1<<int0)
        out eimsk,  r16
        sei
rpt:    sleep
    jmp rpt
```

Interrupt handler

```
check_temp:
        push r18 ;; protect reg
        push r20
        in r20, sreg
        push r20
        lds r20, fan
        in r18, pinc
        cpi r18, 25
        brlo checklower
        inc r20
        out portb, r20
        rjmp endi

checklower:
        cpi r18, 20
        brsh endi
        dec r20
        out portb, r20
endi:
        sts fan, r20
        pop r20
        out sreg, r20
        pop r20
        pop r18
        reti
```

PCINT

PCINT tells us whether a group of pins toggle

Interrupt	Which pins used	Other use
PCINT0	PCINT[7:0]	PB
PCINT1	PCINT[14:8]	PC
PCINT2	PCINT[23:16]	PD

Interrupt hander needs to work out which one toggled

- PCIE: Pin Change Interrupt Control register

  Used to enable specific PCINTs

- Pin Change Mask Register: PCMSK0, PCMSK1 PCMSK2

  PCMSKnb: says whether bit b of the pins for PCINTn should be used.

3.9.3 Timer

Timers are hardware components of the MCU, driven either by external clock or by on-chip oscillator that can be used to mark wall-clock time

- e.g. delay by a certain amount
- e.g. time-slice – give attention to tasks at different time units

Timers and their properties vary slightly from chip to chip

- basic ideas are the same
- we use timer 0 on atmel 168 at example

Controlling a timer

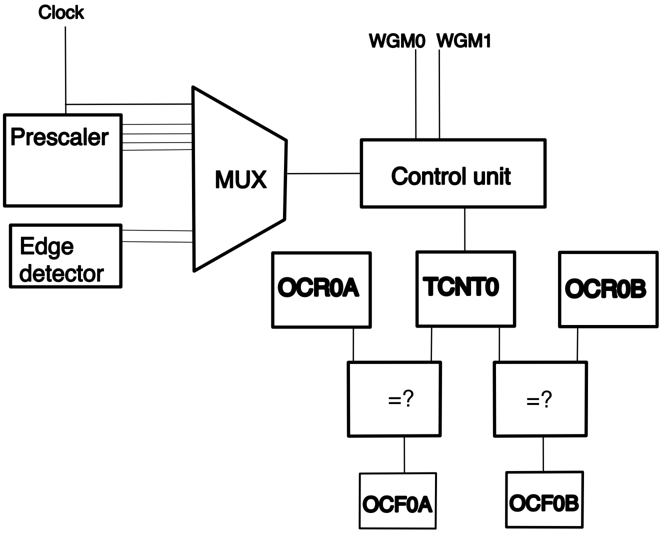
Can choose:

- where clock comes from
- possible clock frequency (range)
- what output looks like
- what happens when clock

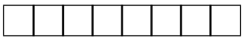
Basic idea simple

- Count a given number of clock ticks
- Do something
  - Can monitor via polling or interrupts

We look at TC0 – there are other timers on the chip, with different properties: for exampe TC1 is 16-bits

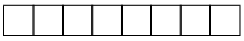


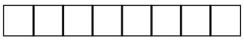
TCNT0



Current value of the clock – typically controlled by the clock and read (IN) but can be set.

TC0 Output Compare: OCR0A and OCR0B





What you're counting to.

Note that we can independently do check a timer with to two different values.

TC0 Control Registers

TCCR0A

D7 (high bit) Low bit: D0

COM 0A1	COM 0A0	COM 0B1	COM 0B0			WGM 01	WGM 00

Controls what happens what a match happens, and whether PWM used

TCCR0B

D7 (high bit) Low bit: D0

FOC0A	FOC0B			WGM02	CS02	CS01	CS00

- FOC: this can be used to force output compare output. We shan't look at this in any detail;
- WGM: Wave Generation Mode – what output on OCR0A looks like
- CS: select clock frequency.

Timer output
When timer goes off

- output to pin OC0A (PD6) and OC0B (PD5)
- NB: either use pin 9 as PD6 or OC0A
- output on OC0A depends on modes

Non-PWM mode

WGM=0x000	Normal	Count to 0xFF
WGM=0b010	CTC	Count to OCR0A

- the counter only counts up.
- When it hits its maximum value the TOV0 flag is set, and TCNT0 wraps
- Can use OCR0A to compare values – OC0A output, interrupt

Non-PWM mode

Compare output mode: Independently for A and B

COM 0A1	COM 0A0	Description
0	0	Normal port operation Pin 9 used as PD6
0	1	When match happens, toggle OC0A
1	0	When match, set OC0A to 0
1	1	When match, set OC0A to 1

Clearing the TOV bit
TOV bit is cleared

- automatically by the interrupt hander if interrupts enabled;
- only when explicitly when no interrupt

Similarly with other output bits

Clock controller: Pre-scalers
Scale clock used in fixed multiples : clock may be too fast

CS02	CS01	CS00	Action
0	0	0	No clock
0	0	1	Use clock
0	1	0	clock/8
0	1	1	clock/64
1	0	0	clock/256
1	0	1	clock/1024
1	1	1	use external clock

NB: Note that this applies to Timer 0: read the data sheet to see the rules for Timer 1 and 2.

Setting interrupts with timers

TC0 Interrupt Mask Register: TIMSK0

D7 (high bit) Low bit: D0

					OCIEB	OCIEA	TOIE

To enable timer-interrupts, the I bit in the Status Register must be set. In addition to enable timer interrupts you must do at least one of the following:

- OCIEB: if this bit is set, the timer/counter interrupt is enabled on the B matcher;
- OCIEA: if this bit is set, the timer/counter interrupt is enabled on the A matcher is set
- TOIE: Interrupt when the TC0 overflows – when the TOV bit is set.

TC0 Interrupt Flag Regsiter: TIFR0

D7 (high bit) Low bit: D0

					OCFA	OCFB	TOV

Note :

- OC0A is a pin which is set when the count match happens
- OCF0A is a bit in the TIFR0
- An interrupt happens when: OCFA is set, when global interrupts are enabled; when the mask register is appropriately set

Pulse width modulation

General technique to use use the duration of a pulse to encode a message.

- **One** of the major applications: where output is fixed binary (e.g., 0V or 5V), can simulate variable output by switching on/off quickly with varying times for the output
- Examples: controlling machines, dimmers

  Has many advantages (easy digital control, good power consumption)

So: vary the on, off times

Controlling PWM

Need:

1. a clock or counter
2. a threshold

Output:

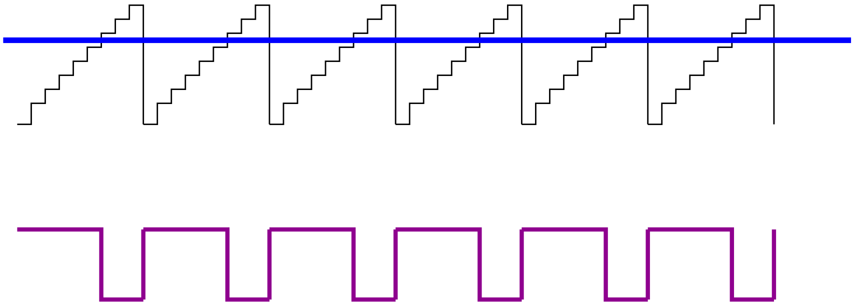
- high when clock/counter is above the threshold
- low otherwise

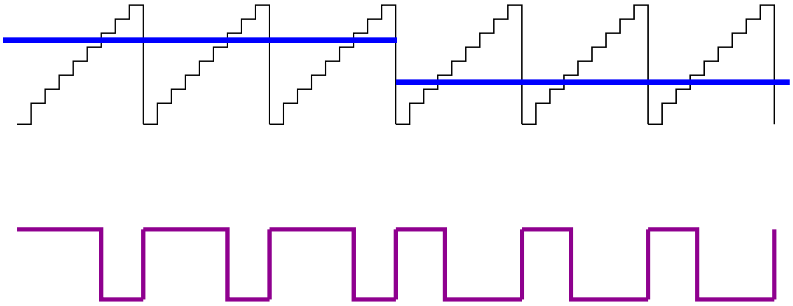
PWM on the AVR

Fast PWM, WGM $\in \{3, 7\}$

COM 0A1	COM 0A0	Description
0	0	Normal port op, OC0A disc
0	1	WGM==3: normal port usage; OC0A disabled WGM==7: toggle OC0A on compare match
1	0	Clear OC0A on match, set at bottom
1	1	Set OC0A on match, clear at bottom

There is a special case when OCR0A is set to 0xFF and COM0A1 is 1. Read the data sheet!

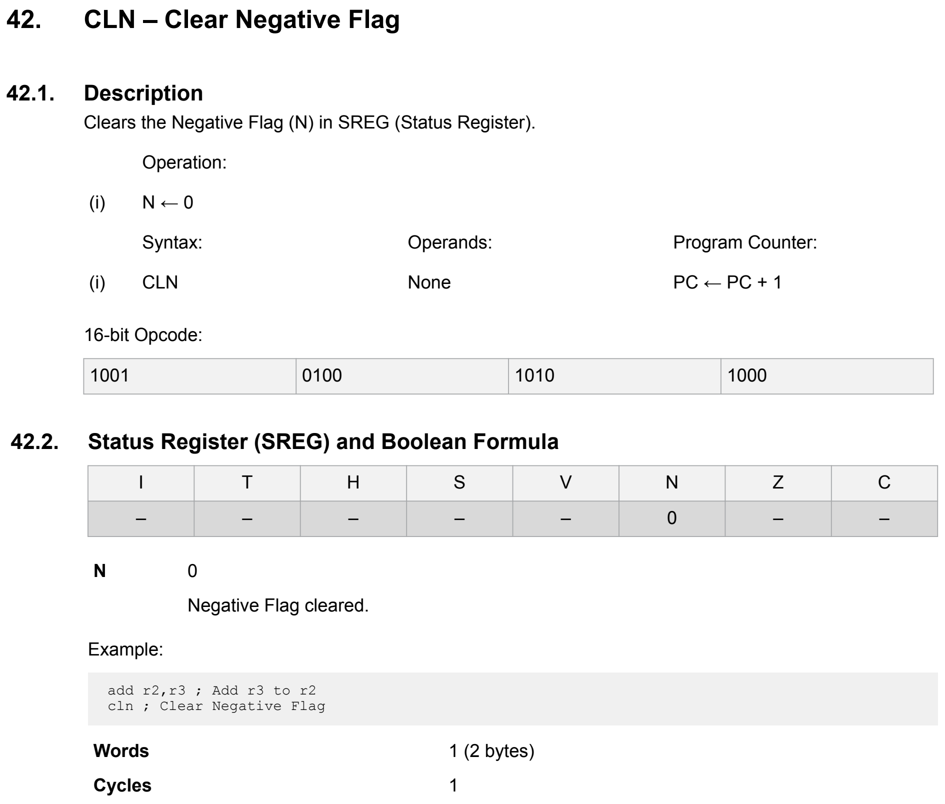




Phase-correct PWM is left for your self-study.

3.10 Interlude

Understanding machine code



42. CLN – Clear Negative Flag

42.1. Description

Clears the Negative Flag (N) in SREG (Status Register).

Operation:

(i) N ← 0

	Syntax:	Operands:	Program Counter:
(i)	CLN	None	PC ← PC + 1

16-bit Opcode:

1001	0100	1010	1000

42.2. Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	–	–	–	0	–	–

N 0

Negative Flag cleared.

Example:

```
add r2,r3 ; Add r3 to r2
cln ; Clear Negative Flag
```

Words	1 (2 bytes)
Cycles	1

48. COM – One’s Complement

48.1. Description

This instruction performs a One’s Complement of register Rd.

	Operation:		
(i)	Rd ← $FF - Rd		
	Syntax:	Operands:	Program Counter:
(i)	COM Rd	$0 \le d \le 31$	PC ← PC + 1

16-bit Opcode:

1001	010d	dddd	0000

48.2. Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	–	⇔	0	⇔	⇔	1

S N ⊕ V, for signed tests.

V 0

Cleared.

N R7

Set if MSB of the result is set; cleared otherwise.

Z $\overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$

Set if the result is $00; cleared otherwise.

C 1

Set.

R (Result) equals Rd after the operation.

73. LDI – Load Immediate

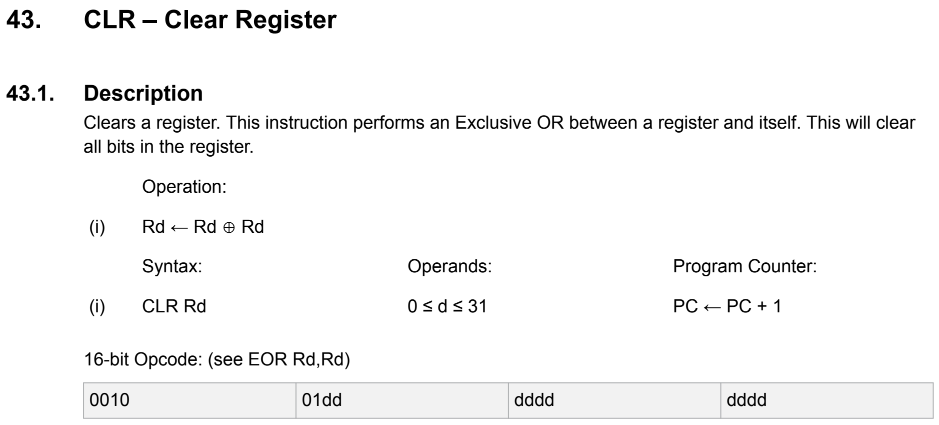
73.1. Description

Loads an 8-bit constant directly to register 16 to 31.

	Operation:		
(i)	Rd ← K		
	Syntax:	Operands:	Program Counter:
(i)	LDI Rd,K	$16 \le d \le 31$, $0 \le K \le 255$	PC ← PC + 1

16-bit Opcode:

1110	KKKK	dddd	KKKK



43. CLR – Clear Register

43.1. Description

Clears a register. This instruction performs an Exclusive OR between a register and itself. This will clear all bits in the register.

Operation:

(i) Rd ← Rd ⊕ Rd

	Syntax:	Operands:	Program Counter:
(i)	CLR Rd	0 ≤ d ≤ 31	PC ← PC + 1

16-bit Opcode: (see EOR Rd,Rd)

0010	01dd	dddd	dddd

3.11 Multiple byte data

16 bit data

For simplicity and speed

- Most operations operate on 1 byte or even less

For example

- Most arithmetic
- Most conditional branches

But we've seen exceptions already for addresses: e.g., JMP, STS

- 16-bit support is provided but more complex

MUL

Multiplying two 8-bit numbers yields a 16-bit number

- Inputs in two specified registers
- Output into R0 (low), R1 (high)

Examples

- MUL R5, R6
- MUL R1, R6
- MUL R1, R0
- MUL R0, R1

Z and C are set

Register pairs

Registers can be paired up to support 16-bit operations

- R1:R0
- . . .
- R27:R26 – Register X (XH:XL)
- R29:R28 – Register Y (YH:YL)
- R31:R30 – Register Z (ZH:ZL)

Only limited operations allowed

```
CLR R31
CLR R30
MOVW R3:R4, Z
ADIW R25:R24, 5
```

Note that adding can only be done on the last four pairs.

- Some important operations coming now

Exercise 1

Read a 16-bit number from port C and D into register pair R25:R24. Delay 3 clock cycles. Read a 16-bit number from port C and D into register pair R27:R26.

- Add the two 16-bit numbers together and put into register Y
- Output result to port C and D
- Carry bit should be low bit of port B

Can you use ADIW?

Note that this is a technical exercise to show you how managing 16-bit arithmetic works. So I am forcing you to read into one set of registers, move between registers and then do addition or the other way round.

Exercise 2

Read two 8-bit numbers from port B and C into suitable registers. Do integer long division

- Quotient to port C
- Remainder to port D

Exercise 3

Read an 16-bit numbers from port C and D, wait 4 clock cycles. Read an 8-bit number from port C. Do integer long division

- Quotient to port B
- Remainder to port C

3.12 Indirect addressing

Primary purpose:

- Dynamic addressing
  - flexibility and power
  - helps support loops
- Allow greater range of memory to be used

Motivation

Read 20 numbers into location 512, 513, 514, . . .

```
IN R17, PINC
STS 512, R17
IN R17, PINC
STS 513, R17
IN R17, PINC
STS 514, R17
IN R17, PINC
STS 515, R17
IN R17, PINC
STS 516, R17
...
```

Not a solution!

ST

Move data from a register to memory

- Source: Register

  R0-R31
- Destination: A memory location

  Address of location found in register X, Y, Z

The source and destination registers should not be the same.

Example:

```
ST X, R5
```

- Register X contains a 16-bit number
- Use its *value* as address in memory
- Store the value of R5 in that address

```
      LDI XH, 2
      LDI XL, 0
      LDI R17, 20
loop: IN R18, PINC
      ST X, R18
      INC XL
      DEC R17
      BRNE loop
```

Note that the "INC XL" does not take care of the carry through so this is not an ideal solution

```
      LDI XH, 2
      LDI XL, 0
      LDI R17, 20
loop: IN R18, PINC
      ST X+, R18
      DEC R17
      BRNE loop
```

Also a `ST X-, R18` operation.

ST Y and Z

- Similar operations for Y and Z registers
- Slightly less efficient (depends on chip)
- More general

```
STD Z+6,R4
```

The displacement operation is not available for all chips

Exercise

Write code to set locations 600–700 to all high bits.

LD

The LD operation loads from memory into a register

- source location address is given by register X, Y, or Z
- destination is a register (R0-R31)

Again, there should be no overlap

Example
How many of the numbers stored in locations 512-573 are odd?

```
      LDI R17, 64
      LDI R18, 0; result
      LDI XH, 2
      LDI XL, 0
loop: LD  R19, X
      SBRC R19,0
      INC R18
else: DEC R17
      BRNE loop
```

This only works because I've carefully chosen the numbers so that XL starts at 0 and never overflows. We could of course either (a) Do an ADIW 1, or (b) check for overflow and then increment XH. But easier is:—

```
      LDI R17, 64
      LDI R18, 0
      LDI XH, 2
      LDI XL, 0
loop: LD  R19, X+
      SBRC R19, 0
      INC R18
else: DEC R17
      BRNE loop
```

LDD

Load indirect with displacement – Y and Z registers

```
LDD r5, Y+6
```

Load the byte at address [Y+6] into the register R5.

Indirect Jumping/Calling

Can store *address* in the Z-register – jump to that location

```
ICALL   ;; implicit argument

IJMP
```

For example, consider this C/C++ switch statement.

```
switch (a) {
   case 0: blah0();
           break;
   case 1: blah1();
           break;
   case 2: blah2();
           break()
}
```

Let's look at one possible implementation in assemby. We'd need write the *called* routines – a label, the code, and then the return. You might need to push/pop registers, save state and so on.
Setup 1:

```
blah0: ....
       ....
       ret

blah1: ....
       ....
       ret

blah2: ...
       ret
```

The second part of the set-up is to set up a *jumptable*. The label (the name is arbitrary – you could use any sensible name) marks the beginning of the jump table — then follows a series of jumps – one for each possible case in the switch statment. Note that

- We deliberately want to use RJMP because it is one word long.

- The first RJMP is at location *jumptable*+0
- The second RJMP is at location *jumptable*+1
- The third RJMP is at location *jumptable*+2
- and so on

Set-up 2

```
jumptable:
   rjmp blah0
   rjmp blah1
   rjmp blah2
   ....
```

The idea is that we'll set up the Z register so that we can use IJMP to jump to the correct position in the jump table – and there we'll do an RJMP to the actual code.

In our example, we'll assume that R17 is the switch value. First we set up the Z register so that it contains the address of the *start* of the jump table. Then we add to Z the value of R17 (note we have to take care of the possible carry into the high bit – have a look at the SBCI command, we could do this a little more cleanly). Then we can do the IJMP.

switch(R17):

```
clr r0
ldi zh, high(jumptable)
ldi zl, low(jumptable)
add zl, r17
add zh, r0
ijmp
```

Note how we don't specify an operand for IJMP – it always uses the Z-register.

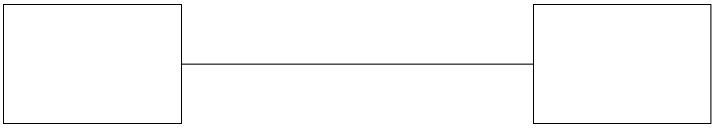
There is a lot of work in this set up, and if we only had three options then an if/then/else type of code would be much easier. But if you have 50 options, then that would be very tedious and inefficient.

- What is the relationship between this and how the IVT works?

3.13 USART

3.13.1 General principles

Universal Synchronous/Asynchronous Receiver Transmitter
Problem: send characters (multi-bit) data on a *serial* line to another device



Some issues

- serialisation/deserialisation
- character size (5-9 bits)
- detecting errors
- data transfer rate
- duplex, half-duplex, simplex
- synchronous – same clock (rise, fall)
  asynchronous – same clock rate, but rise and fall may be different

External pins

Standard

- XCK (PD4) – clock for external device in synchronous mode
- RXD (PD0) – receive
- TXD (PD1) – transmit

There is another mode which we won't explore now that implements the SPI (Serial Peripheral Interface) protocol.

External pins

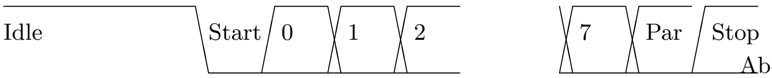
SPI

- SCK (PB5)
- MISO (PB4)
- MOSI (PB3)
- $\overline{\text{SS}}$ (PB2)

Baud and Sampling rate

USART typically runs *much* slower than the clock speed

- Particularly in *asynchronous* mode, communicating devices need to agree on what the bits are – detect when signal changes
- The USART *samples* input values several times for each bit input
- Pre-scaler controls speed



Above example: frame of 8 data bits and 1 stop bit. Note that idle is indicated by a high bit.

There is a clock and data recovery unit that attempts to synchronise the receiver in asynchronous mode.

- In normal mode, every clock cycle (that is the clock of the USART, not the clock of the AVR), the input value is sampled 16 times.
- When idle, the communicating line is kept high.
- When sampling detects that the line has gone low there is a possible start of communication – it then checks to see that this isn't noise by resampling – are samples 8, 9, 10 also low? Provided at least 2 of them, the USART assumes there is a signal.
- Once communication starts, the sender and receiver are synchronised though there may be a small skew and the synchronisation may wander over the communication. The receiver takes the middle three samples of the 16 samples of each cycle and does majority voting.

3.13.2 Parity

The chip automatically does parity computation/checking – you need to respond

- Parity : is the number of 1 bits odd or even
- $P_e = d_0 \oplus d_1 \oplus d_2 \oplus \ldots \oplus d_{n-1}$
- $P_o = d_0 \oplus d_1 \oplus d_2 \oplus \ldots \oplus d_{n-1} \oplus 1$

3.13.3 Key registers for the USART

USART I/O data register
8 bit register where the data for sending/receiving

UCSR0A: USART Control and Status Register 0 A

- RXC0 : USART Receive complete
- TXC0 : USART Transmit complete

- UDRE0: USART Data Register Empty
- FE0: Frame error
- DOR: Data OverRun
- UPE0: USART Parity Error
- U2X0: Double speed (in Master SPI Mode only)
- MPCM0: Multiprocessor Communication Mode (Master SPI)

- FE0: A frame error occurs when we have already received the expected number of data bits and now expect a stop bit (i.e, high signal) but there is a data bit (signal is low). This picks up an error condition. Of course, we cannot distinguish between an unexpected 1 data bit arriving and the stop bit.
- DOR0: Data overrun: The receive buffer is full and a new start bit is detected. We need to make sure we have read the receive buffer before the next data arrives

UCSR0B: USART Control and Status Register 0 B

- RXCIE0 : RX Complete Interrupt Enable
- TXCIE0: TX Complete Interrupt Enable
- UDRIE0: USART Data Register Empty Interrupt Enable
- RXEN0: Receiver Enable 0 – enable (disable) RXD (PD0)
- TXEN0: Transmit Enable 0 – enable (DISABLE) TXD (PD0)
- UCSZ02: Character Size 0 (Master SPIM)
- RXB80: 9th bit if needed
- TXB80: 9th bit if needed

UCSR0C: USART Control and Status Register 0 C

7:6 UMSEL0n (mode: asynch?/MSPIM?)

5:4 UPM0n: USART Parity mode

3 USBS0: USART Stop Bit Select 0 (num stop bits)

2 UCSZ01

1 UCSZ00

0 UCPOL0: Clock polarity – for synhcronous mode
Set to 0 in asynchronous mode

The UCSZ bits set the size of the data frame

UCSZ0[2:0]	Size
000	5
001	6
010	7
011	8
111	9

Note that in Master SPIM mode, bits 1 and 2 have other meanings. Please look at the manual.

The UCPOL0 determines when the transmitter data changes and the receiver samples. When set to 0 (1) the transmitted data changes on a rising (falling) XCK0 edge and the receiver is sampled on a falling (rising) XCK0 clock edge

Baud rate registers – 12 bits (two 8 bit registers)

- UBRR0L[7:0] (lower 2 nibbles)
- UBRR0H[3:0] (high nibble)

Note that

- Baud rate changes when UBRR0L written to
- In asynchronous normal mode:

$$textrmBaudrate = \frac{f_{\mathrm{OSC}}}{16(UBRR0 + 1)}$$

What are the advantages and disadvantages of faster/slower baud rate?

3.13.4 Simple Example Code

This is lifted from section 24.6-8 of the ATMEL 328P data sheet. Code may be slightly different for other AVR machines

Code use : initialisation

Assume r17:r16 has the baud rate parameter

```
out  ubrr0h, r17
out  ubrr0l, r16
; enable
ldi  r17, (1<<RXEN0)|(1<<TXEN0)
out  ucsr0b, r16
; 8-bit, 2 stop bits
ldi  r16, (1<<USBS0)|(3<<UCSZ00)
out ucscr0c, r16
```

Obviously you could set up differently – e.g., only for receiving, 5 bits, 1 stop bit. You can also set up interrupt handling – so that an interrupt executes when the receive or transmit completes. This would be quite common.

Data transmission

Assume data in R16

```
transmit:
   ; wait for previous transmit to complete
   in  r17, ucsr0a
   sbrs r16, udre
   rjmp transmit
   ; send
   out udr0, r16
```

Note that we can only send one byte at a time. The USART on our behalf serialises and sends each byte one bit at a time. While we don't have to worry about the detail of how the USART does this we have to wait for the USART to complete the sending of the previous byte before we can send the next.

This example uses active polling. In practice, we would often use interrupts – an interrupt would be invoked each time a send is finished and the next send is done.

The UDR0 register has the data to be transmitted – if the frame size is < 8 then the upper bits are ignored. If the frame bit size is 9, transmission is a little more complex – you need to set the TXB8 bit of the UCSR0B register – see the example in the manual.

Receiving

```
; has data been received
receive:
  in r16, ucsrc0a
  sbrs r16, rxc
  rjmp receive
  in r16, udr0
```

This is active polling for receiving – it would be more common to use interrupt handing for receiving. Cleaner and much more efficient.

If your frame size is < 8 then the upper bits of the UDR0 are set to 0. If your frame size is 9, you need to do a little more PT – see the manual.

3.13.5 Other communication possibilities

- Master SPI Mode
- Two wire interface: TWI

Provides:

- Faster communication
- Addressing multiple devices