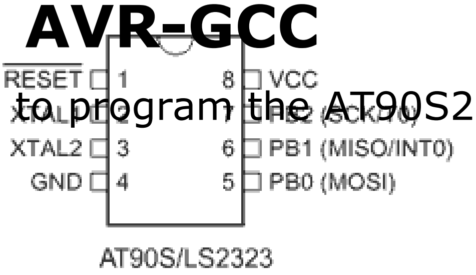
AVR-GCC

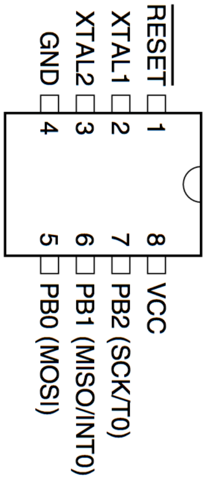
Using GCC to program the AT90S2323

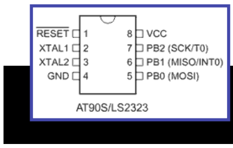


Raffi Krikorian
raffi@media.mit.edu

AT90S2323

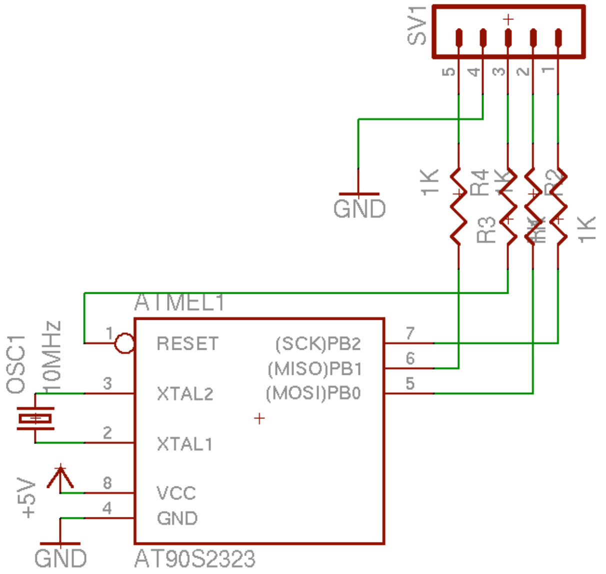
- 0-10MHz operation
- 2KB Flash
- 128B SRAM
- 128B EEPROM
- 8-bit Timer
- 3 I/O Pins
- 1 Interrupt on external pin
- ~10mA @ 4-6V





Raffi Krikorian
raffi@media.mit.edu

Wiring it up

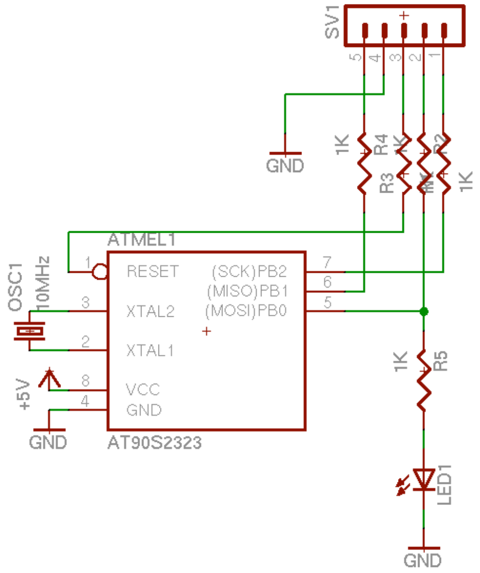


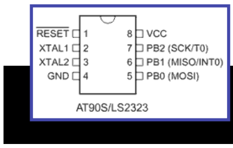
Installing devel software

```
root@phm2:/home/raffik# apt-get install gcc-avr avr-libc
```

- Install the compiler
  - gcc-avr
  - http://www.avrfreaks.net/AVRGCC/
- Install useful development libraries
  - avr-libc
  - http://www.nongnu.org/avr-libc/

Pulsing a LED

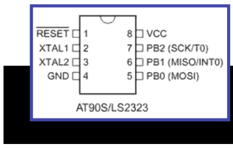




pulse.c

```
#include <avr/io.h>

int main( void )
{
  // define some variables
  int c, d = 0;
  int dir = 1;

  // set the “direction” of PORTB
  DDRB = _BV( PB0 );

  while (1)
    {
      // turn on the LED for a
      // constant amount
      PORTB |= _BV( PB0 );
      for( c=0;c<0x2ff;c++);
```

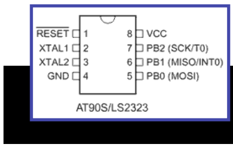
```
      // turn off the LED for a
      // computed amount of time
      PORTB &= ~_BV( PB0 );
      for( c=0;c<d;c++ );

      // compute the next round’s
      // down time
      d += dir;
      if( d == 0x4ff )
        dir = -1;
      else if( d == 0 )
        dir = 1;

    }

}
```



Dissecting the code (part 1)

- Io.h has constants for registers
- DDRB is “Data Direction Register”
  - a 1 in a bit position makes it an output
  - a 0 makes it an input
- PORTB is data register
  - a 1 in a bit position drives the pin high
  - a 0 grounds the pin

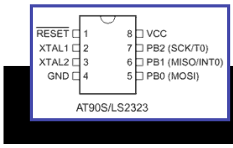
```
#include <avr/io.h>

DDRB = _BV( PB0 )

PORTB |= _BV( PB0 )
```



Dissecting the code (part 2)

- _BV is a macro to create a number with a bit “turned on”
  - _BV(0) = 0b00000001
  - _BV(3) = 0b00001000
- PB0 is a constant defined to be that pin number in PORTB

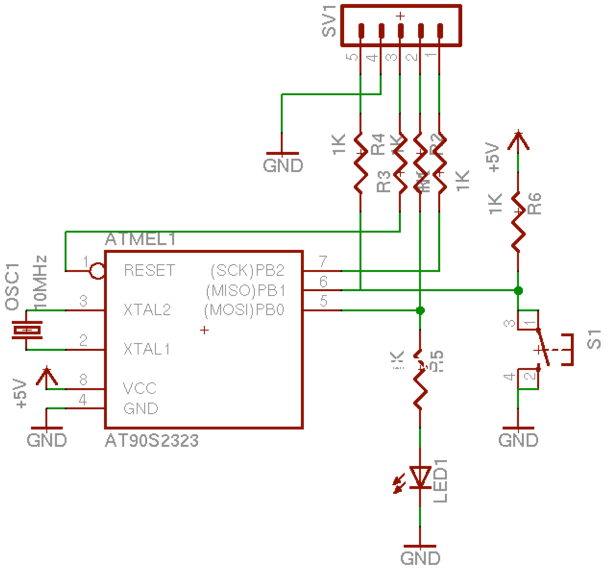
```
#include <avr/io.h>

DDRB = _BV( PB0 )

PORTB |= _BV( PB0 )
```



Makefile

```
CPU    = at90s2323

CC     = avr-gcc
CFLAGS = -mmcu=${CPU} -g -Os

all: pulses.hex

clean:
        rm -f *.elf *.hex *~

%.elf: %.c
        ${CC} ${CFLAGS} -o $@ $?

%.hex: %.elf
        avr-objcopy -j .text -j .data -O ihex $? $@

burn-pulses: pulses.hex
        uisp -dlpt=/dev/parport0 -dprog=dapa -dvoltage=5 -dt_sck=50 --erase \
        --upload if=pulse.hex
```

Reading an Input



input.c

```
#include <avr/io.h>

int main( void )
{
  DDRB = _BV( PB0 );
  PORTB = _BV( PB0 );

  while (1)
    PORTB = PINB >> 1;

}
```

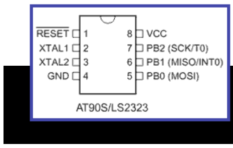
Dissecting the code

- PINB is “Input pins”
  - if DDRB is setup properly, PINB reflects the values of the input pins
- While loop sets PORTB to the value of PINB shifted right one
  - Input is on PORTB pin 1, output is PORTB pin 0

```
DDRB = _BV( PB0 );

while (1)
    PORTB = PINB >> 1;
```

iinput.c (interrupt driven)

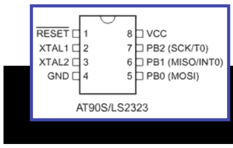
```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>

SIGNAL(SIG_INTERRUPT0)
{
  PORTB ^= _BV( PB0 );
  MCUCR ^= _BV( ISC00 );
}

int main( void )
{
  DDRB = _BV( PB0 );
  PORTB = 0;

  MCUCR = _BV( ISC01 );
  GIMSK = _BV( INT0 );
  SREG = 0x80;

  while (1);
}
```



Dissecting the code (part 1)

- Define the interrupt handler
  - SIG_INTERRUPT0 is the interrupt vector assigned to PORTB
- MCUCR is the “Control Register”
  - setting various bits puts the micro into sleep mode, controls interrupts, etc.
  - the ISC00 bit controls whether interrupts catches the rising or falling edge

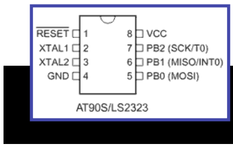
```
SIGNAL(SIG_INTERRUPT0)
{
  PORTB ^= _BV( PB0 );
  MCUCR ^= _BV( ISC00 );
}
```



Dissecting the code (part 2)

- GIMSK is "General Interrupt Mask Register"
  - on the AT90S2323, the only interesting bit is INT0 which enables the interrupts on PORTB
- SREG is "Status Register"
  - it holds the Z, N, etc. bits for arithmetic
  - the 8th bit enables interrupts

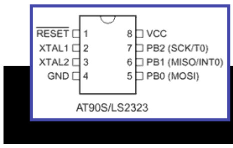
```
MCUCR = _BV( ISC01 );
GIMSK = _BV( INT0 );
SREG = 0x80;
```



Viewing the compiler's code

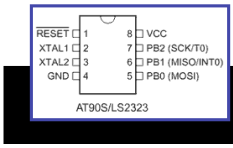
```
raffik@phm2:~/$ avr-gcc -mmcu=at90s2323 -g -Os  -o input.elf
input.c
raffik@phm2:~/$ avr-objdump -DS input.elf > input.dmp
```

- Compiling with -g turns on the debugging information in the compiled executable
- avr-objdump takes a compiled ELF and outputs the assembler code
- You'll not only get the assembler for the code you wrote, but you'll see all the initialization routines and how GCC operates



input.dmp

```
0000003a <main>:
#include <avr/io.h>

int main( void )
{
  3a:   cf ed           ldi     r28, 0xDF       ; 223
  3c:   d0 e0           ldi     r29, 0x00       ; 0
  3e:   de bf           out     0x3e, r29       ; 62
  40:   cd bf           out     0x3d, r28       ; 61
  int c, d = 0;
  int dir = 0;

  DDRB = _BV( PB0 );
  42:   81 e0           ldi     r24, 0x01       ; 1
  44:   87 bb           out     0x17, r24       ; 23
  PORTB = _BV( PB0 );
  46:   88 bb           out     0x18, r24       ; 24

  while (1)
    {
        PORTB = PINB >> 1;
  48:   86 b3           in      r24, 0x16       ; 22
  4a:   86 95           lsr     r24
  4c:   fc cf           rjmp    .-8             ; 0x46
```



In closing...

- Writing in C will make your life easier
  - if you do want assembler code, use the asm block and insert it into the C
- Poke around in /usr/avr/include and /usr/avr/include/avr
  - all the constants you need have probably been defined as they appear in the datasheets
- Higher level AVRs can
  - make use of the more complex parts of libc like printf
  - usually have JTAG, so can use gdb for debugging