An Atmel AVR Notebook

Michael A. Covington
Artificial Intelligence Center
The University of Georgia
Athens, GA 30602-7415
www.ai.uga.edu/mc

Last revised August 2006

This paper consists of my notes made while learning to use Atmel AVR microcontrollers with an STK500 starter kit (development system). I present them here in the hope that they will be useful to others.

The processor

For concreteness, I have chosen to work with the 8-pin ATtiny13 processor. This presently costs $1.20 per chip at Futurlec (www.futurlec.com). Cheaper microcontrollers exist (Atmel ATtiny11, Microchip PIC12F508), but they are appreciably less powerful. The ATtiny11 is distinctly harder to work with because it does not support ISP (in-circuit serial programming).

There are 32 registers, R0 to R31. Only the upper half of these (R16 to R31) can use **ldi** (load immediate) and some related instructions. Thus, it is customary to use registers starting at R16.

There are also 64 bytes of RAM (data memory) and 1K bytes (512 words) of program memory (flash ROM).

The ATtiny13 also has a 10-big analog-to-digital converter with 4 inputs. There is also an analog comparator input (with a built-in 1.1-volt reference) and a timer with provision for PWM output.

The control bits are called “fuses” and are set in a separate menu in the programmer. They are not actually fusible links in present-day chips; they are reprogrammable.

By default, the ATtiny13 uses its internal RC oscillator at 9.6 MHz prescaled by a factor of 8, giving a clock speed 1.2 MHz. *Oscillator calibration is done automatically* at chip initialization. You can read the OSCCAL value and change it during program execution if you wish.

The other options comprise a low-power 128-kHz internal oscillator, external RC oscillator, or external clock input (but *not* a crystal; on that point the PIC12F508 has the advantage).

A total of 6 port pins are available (not 5 as on the PIC12F508).

To use PB3 as a port pin, you must be using the internal oscillator.

To use PB5 as a port pin, you must set the "Reset Disabled" fuse. PB5 cannot source or sink substantial current (its practical limit is about 1 mA).

The ATtiny13 runs on 2.6 to 5.5 V; the ATtiny13V, 1.8 to 5.5 V.

Disadvantages and "gotchas"

Atmel .hex files do not contain the "fuses" (configuration bits). They have to be set separately on the menu of whatever programmer you are using.

The STK500 development kit has to have jumper wires installed in order to program microcontrollers. They are different depending on which chip is being programmed.

Within AVR Studio, the STK500 programming tool does *not* notice if you have switched to a different project (a different .hex file) than you were previously using. You must set the .hex file manually every time you change projects. "Use current simulator flash memory" is a good option to use, provided you always simulate your programs before programming the AVR.

The AVR Dragon development kit does not have a socket for the device that you will be programming. Atmel gives instructions for adding sockets and jumper wires to suit the CPU you are working with.

STK500 operation

I am using the STK500 development kit with AVR Studio version 4.12 SP3. Right out of the box, *the STK500 firmware needs upgrading* and AVR Studio automatically takes you through this process. Note that you have to unplug the AT90S8515 that is supplied in one of the programming sockets of the STK500.

To my surprise the STK500 did not come with a power supply. You have to find your own 10- to 15-volt supply (500 mA) and wire it to the cable supplied. Mine is rated at 9 V but supplies more than that voltage and works fine.

You will need an IC puller or any tool that you can use as a miniature crowbar, or possibly a pair of needlenose pliers that will open wide, in order to remove microcontrollers from the sockets on the STK500. If you try to pry them loose with a screwdriver, you will bend pins on the numerous cable headers.

Also, the STK500 has to be wired together with jumper cables (which are supplied) to configure it for any particular CPU.

The ATtiny13 came out after the printed/PDF STK500 manual was written. The AVR Tools User Guide, part of the AVR Studio help system, is up to date. From it, we learn that the ATtiny13 configuration is:

- Device in socket 3400D1 (blue);
- All jumpers in default positions (they are documented on the bottom of the board);
- Connect ISP6 to SPROG1 with a 6-pin cable;
- Connect PORTE.RST to PORTB.PB5 with a 1-pin cable;
- Connect PORTE.XT1 to PORTB.PB3 with a 1-pin cable;
- Set ISP frequency (in AVR Studio STK500 menu) to 1/5 the target clock frequency, or lower. (By default, target clock frequency is 9.6 / 8 = 1.2 MHz.)

No 1-pin cables are supplied with the STK500. I made some with crimp terminals. Alternatively, you can use the supplied 2-pin cables, leaving half of each cable unused.

Minimal assembly-language program

Here is a program that just turns on an LED connected to PORTB.PB0 and turns off an LED on PORTB.BP1 (so you can see that the pins are not all in the same state, as they might be if the program had had no effect). It can be run on the STK500 by running a 2-wire jumper from PB0 and PB1 to LED0 and LED1, without disconnecting the other jumpers needed for programming.

```
; LEDon.asm - M. Covington 2006
; For ATtiny13.
; Turns on the LED attached to PB0 and
; turns off the LED attached to PB1.

.include "tn13def.inc"

.def  temp  = R16

start:      ser   temp
            out   PORTB, temp ; port B all high
            out   DDRB,temp   ; port B all outputs
            cbi   PORTB,0     ; lower PB0 to turn LED on
            rjmp  start
```

The STK500 LEDs turn on when connected to a *low* (binary 0) output.

Assembly-language syntax

The assembler used by AVR Studio 4 is AVRASM2 and is documented in the online help. You will also need to read the documentation for the original AVR Assembler, since the AVRASM2 documentation only covers changes.

By default, numbers are decimal. Hex and binary numbers are written like **`0xAB`** and **`0b10101011`** respectively. Numbers that begin with 0 are octal, such as **`0253`**.

LOW(expression) and **HIGH(expression)** return the low and high bytes of an expression such as the predefined **RAMEND**.

All labels have to end with a colon. That's how this assembler distinguishes labels from instructions. It doesn't matter whether the label or instruction starts in column 1.

Pseudo instructions begin with periods. Some important ones are:

.def	symbol = register	Assign a name to a register
.equ	symbol = expression	Assign a symbol to an expression
.set	symbol = expression	Like **.equ** but can be reassigned
.include	"filename"	Include a file in project directory or in set of files provided with the assembler, such as tn13def.inc
.org	address	Assemble at address

	.cseg		Assemble into program memory (default)
	.dseg		Assemble into data memory
	.eseg		Assemble into EEPROM
label:	**.db**	expression, expression...	Define constant bytes
label:	**.dw**	expression, expression...	Define constant words
label:	**.byte**	number	Reserve bytes

Comments are set off with **;** or **//** or **/* ... */.**

There is a C-like preprocessor whose directives begin with #.

Oddly, there is no **.end** pseudo instruction, but you can say **.exit**, which means "stop reading from this file."

Delay loops

It would be more fun if we could make the LED blink. For that, we'll need a delay loop. Here is a generic delay loop in AVR assembly language:

```
.def  temp = some register in range R16 to R31

      ldi   temp,k            ; k is the loop count
L1:   waste time for n cycles
      dec   temp
      brne  L1
```

This loop takes $k(n + 3)$ clock cycles. Here k ranges from 1 to 255, or if preset to 0, is effectively 256. The reason there is no constant term in the formula is that although we add a cycle by performing the **ldi** at the beginning just once, we save a cycle the last time through, when **brne** takes one cycle instead of two because it doesn't branch.

Note that the "waste time" code can be another loop. Then you have a structure like this:

```
.def  temp1 = some register in range R16 to R31
.def  temp2 = another register in range R16 to R31

      ldi   temp1,j           ; j is outer loop count
L1:   ldi   temp2,k           ; k is inner loop count
L2:   waste time for n cycles
```

```
      dec   temp2
      brne  L2
      dec   temp1
      brne  L1
```

This takes $j(k(n + 3) + 3)$ clock cycles, where, again, j and k range from 1 to 255, or if preset to 0 will be effectively 256.

A triple loop would take $j(k(m(n + 3) + 3) + 3)$ cycles.

Note that **rjmp PC+1** (jump to next instruction) takes 2 cycles whereas **nop** takes only one cycle. Each of these is a single-word instruction.

Delay loop example

Here is a concrete example, a 0.1-second delay. At 1.2 MHz, that is 120,000 cycles. Exploring the formula $j(k(n + 3) + 3)$ with a spreadsheet, we find that:

$n = 0$ $k = 199$ $j = 200$ $n(k + 3)(j + 3) = 120{,}000$

We can implement this:

```
      ldi   temp1,200         ; outer loop count
L1:   ldi   temp2,199         ; inner loop count
L2:   dec   temp2
      brne  L2
      dec   temp1
      brne  L1
      rjmp  PC+1              ; 2-cycle no-op
      nop                     ; 1-cycle no-op
```

Blinking LED program

Here is a program similar to the previous one except that the LEDs swap states every 0.1 second (at 1.2 MHz; note that this will not be precise if the internal oscillator is used and not calibrated).

```
; LEDblink.asm - M. Covington 2006
; For ATtiny13.
; Blinks the LEDs attached to PB0 and PB1.

.include "tn13def.inc"

.def  temp  = R16
.def  temp1 = R17
.def  temp2 = R18
.def  mask  = R19

start:      ldi   temp,0b00000010
```

```
            out   PORTB,temp  ; initialize port B
            ser   mask
            out   DDRB,mask   ; port B all outputs

blink:
            ; Delay 0.1 sec (1.2 MHz)
            ldi   temp1,200         ; outer loop count
L1:         ldi   temp2,199         ; inner loop count
L2:         dec   temp2
            brne  L2
            dec   temp1
            brne  L1


            ; Toggle PB0 and PB1
            ldi   mask,0b00000011
            eor   temp,mask
            out   PORTB,temp

            rjmp  blink

; End of program
```

Analog-to-digital conversion

Here is a demonstration of ADC that can be run on the STK500 with a potentiometer connected across Vtg and GND with its wiper connected to PB4. (PB2 was not used because it is a programming input and is not open circuit on the STK500; PB1 and PB0 are fed to diodes.) In this case the upper 2 bits of the 10-bit conversion result are displayed on the diodes.

```
; adcdemo.asm - M. Covington 2006
; For ATtiny13.
; Analog input to PB4 is digitized and its highest 2 bits
; are shown on LEDs (active low) wired to PB1 and PB0.

.include "tn13def.inc"

.def  temp  = R16

start:      ldi         temp,0b00000011
            out         PORTB,temp  ; initialize port B
            out         DDRB,temp   ; PB0 and PB1 outputs


            ; Set up ADMUX.
            ; Reference voltage = Vdd (default)
            ; Input is PB4 (ADC2)
            ldi         temp,0b00000010
            out         ADMUX,temp

            ; Set up ADCSRB
            ; Let the ADC trigger itself (free-running mode)
```

```
            ; This is already all zeroes

            ; Set up ADCSRA
            ; Run continuously (free-running mode)
            ldi         temp,0b11100000
            out         ADCSRA,temp

            ; Grab top 2 bits, invert, transfer to LEDs
loop:       in          temp,ADCH
            com         temp
            andi  temp,0b00000011
            out         PORTB,temp

            rjmp  loop

; End of program
```

ImageCraft C

The free demo version of ImageCraft C (www.imagecraft.com) is limited to 4 KB of program output after the trial period expires. That is larger than the ATtiny13., so it is not a problem. Although my lab has bought a licensed copy of ImageCraft C, we have not yet installed it because, once installed, it is not easily transferrable from one CPU to another, so we're going to have to give the installation careful thought.

Here is the blinking LED program in C. ImageCraft C does not provide any precision delay subroutines; the one here is based on one of their examples and has not been timed precisely.

```
/* ImageCraft C example of controlling 2 LEDs */
/* on the STK500 */

#include <iot13v.h>

void delay (unsigned char n)
{
  unsigned char a, b, c;
  for (a=0; a<n; a++)
    for (b=0; b<n; b++)
        for (c=0; c<n; c++);
}

void main ()
{
  DDRB  = 0b00000011;   /* PB1 and PB0 are outputs */
  PORTB = 0b00000011;   /* PB1 off, PB0 on */
  while (1)
  {
    PORTB ^= 0b00000011;
    delay(50);
  }
```

```
}
```

When you "Compile to Output," ImageCraft C interfaces directly with the STK500. However, it provides no easy way to set the fuse bits; for that, it is still more convenient to use AVR Studio.

Be sure to set the right processor (ATtiny13) in Project Options as well as including the appropriate include file.

Gnu C Compiler (GCC)

This time-honored freeware compiler (which was used to build Linux and tremendous amounts of UNIX software) has been ported to the AVR and is known as AVR-GCC. It is distributed as part of a package called WINAVR (http://winavr.sourceforge.net).

Beginning with version 4.12, AVR Studio comes partly preconfigured for WINAVR. You still have to download WINAVR and install it separately. When you do, AVR Studio finds it and uses it.

Much of the appeal of AVR-GCC is that it provides a large library of predefined functions.

Here is the AVR-GCC version of the blinking LED program. Precision time delays in microseconds are also provided, but they use an arithmetic library that is too big to fit in the ATtiny13.

```
/* Blinking LEDs in AVR-GCC */

#include <avr/io.h>

#define F_CPU 1200000
#include <util/delay.h>

int main()
{
      DDRB  = 0b00000011;
      PORTB = 0b00000010;
      while (1)
      {
            PORTB ^= 0b00000011;
            _delay_loop_2(50000);
      }
}
```

Both C compilers produced output of about the same size; with a program this small, much of the code had to do with stack setup and initialization.

Gnu C is much more useful on the larger AVR microcontrollers, but as the example shows, it can be used for quick development of simple programs on even the smallest ones.

The C string problem

C was designed for computers with a Von Neumann architecture, i.e., a single type of memory addressing for program and data memory, such as the VAX, Pentium, or in the world of microcontrollers, 68HC11 or MSP430.

Like most microcontrollers, the AVR series has a Harvard architecture, in which the instructions that retrieve data from program memory (ROM) and from data memory (RAM) are different.

String constants normally go into data memory, which, on the ATtiny13, is very small. Even one string can fill it up. Consult each C compiler's manual for advice on how to work around this.

BASCOM (BASIC)

BASCOM-AVR, available from www.mcselec.com, is a BASIC compiler for the AVR. This is arguably an easier way to program small CPUs than either C or assembly language. There is a free demonstration version of the compiler as well as a commercial version (79 euros). The demonstration version will generate up to 4 KB of code, which is larger than the ATtiny13.

```
' LEDinBASIC.bas - M. Covington 2006
' Blinking LEDs on PB0 and PB1

   $regfile = "attiny13.dat"  ' pull in definitions for the ATtiny13
   $crystal = 1200000         ' inform compiler of clock speed

   Config Portb = &B00000011  ' data direction register
   Portb = &B00000011

L1:
   Portb = Portb Xor &B11
   Waitms 200
   Goto L1

   End
```

BASCOM will program microcontrollers directly once you tell it where to find STK500.EXE (a file that comes with the STK500).

BASCOM has very good built-in routines for such common tasks as LCD output, UART i/o, and I2C i/o. It is integrated with the STK500 and other programmers.

Be sure to declare **$hwstack = 16** if you want to declare any variables; otherwise all of RAM is used for stack. The ATtiny13 does not require the **$tiny** declaration.

-end-