AVR micro-controller programming with PHOENIX-MDK

July 9, 2009

Chapter 1

Getting Started

The majority of computer systems in use today are embedded in other machinery, such as automobiles, telephones, appliances, and peripherals for computer systems. Most of them have minimal processing and memory requirements and can be implemented using micro-controllers. A microcontroller is a small computer on a single integrated circuit consisting of a CPU combined with program and data memory, periferals like analog to digital converters, timers, serial I/O etc. Intel 8051, Atmel AVR, PIC etc. are popular microcontroller series available in the market. This document is about programming Atmel ATmega16 micro-controller in C language. The reason for choosing the AVR series is the availability of AVR GCC compiler which is under GNU General public license. ATmega16 is chosen due to its simple pinouts where the Input/Output ports are organized in a clean manner and the availability of it in DIP package, that can be mounted in a socket.

The ATmega16 has 16K bytes of Flash Program, 512 bytes EEPROM and 1K byte SRAM. Three Timer/Counters, Internal and External Interrupts, a serial programmable USART, a byte oriented Two-wire Serial Interface, an 8-channel, 10-bit ADC and an SPI serial port are some of the periferal devices on the chip. In this chapter, we will describe the minimal hardware and software to get started. All the example files and batch files given in this document can be downloaded from the Phoenix website.

1.1 The minimum hardware

A pinout of ATmega16 is shown in figure 1.1(a). AVR series micro-controllers support serial program loading. The minimum circuit to required to start experimenting with Atmega16 is shown in figure 1.2. A five pin header is used to connect the Ground, Reset and the three serial program loading pins to the

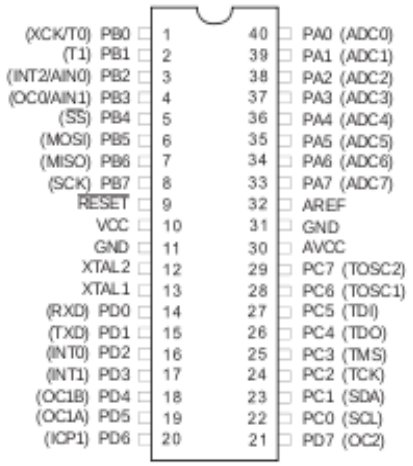

Figure 1.1: (a)Atmega16 pinout.

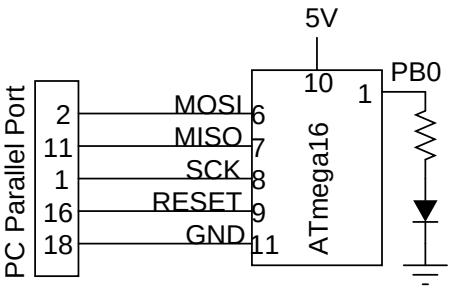

Figure 1.2: The minimum circuit required to get started.

parallel port of a PC[1].

An LED is connected from Bit 0 of Port B to run a blinking LED test program. You need a 5V power supply on pin VCC. The Analog supply input (AVCC, pin 30) is connected to the Digital supply input (VCC, pin 10) through a 10Ω resistor. Decoupling capacitors ($0.1\mu F$) are connected from both power supply pins to ground.

1.2 Getting the Software

We need the following software packages to develop programs from AVR series of micro-controllers:

- gcc-avr Gnu C Compiler for AVR
- binutils-avr Binary utilities (linker, assembler, etc.) for AVR
- avr-libc Basic C-library for AVR
- gdb-avr Gnu debugger for AVR
- uisp, a simple command line tool to upload compiled hex code into the target hardware

On a debian system they can be installed using the commands

```
# apt-get install gcc-avr
# apt-get install avr-libc
# apt-get install gdb-avr
# apt-get install uisp
```

Debian 5 (Lenny) has all of them on the distribution DVDs. You can also get it from the Debian repository.

On Fedora use the following commands to install them from the network

```
# yum install avr-gcc
# yum install avr-binutils
# yum install avr-libc
# yum install avr-gdb
# yum install uisp
```

You can also install them by downloading the RPM files. Visit the sites rpmfind.net or rpm.pbone.net and use the search options to locate the packages.

For Windows, you can download the entire package from http://sourceforge.net/projects/winavr/files/ as an executable file and install it.

[1] You can also do the program loading using the RS232 port or using a dedicated programmer, we have just chosen the parallel port option.

1.3 Compilation and Program Loading

Programming is done is C language and compiled using the AVRGCC compiler. An Intel HEX format file is generated and it is uploaded to the micro-controller using the program *uisp*. Since the compilation and uploading commands require a lot of command line options, we put them in two batch files.

The batch file *compile* for compiling the source is shown below. This batch file also geneartes a map file and listing file, which may not be required most of the time.

```
avr-gcc -Wall -O2 -mmcu=atmega16 -Wl,-Map, $1.map -o $1 $1.c
avr-objcopy -j .text -j .data -O ihex $1 $1.hex
avr-objdump -S $1 > $1.lst
```

The batch upload for loading the program to the micro-controller through the parallel port is shown below. It erases the existing code from the flash memory, loads the new program and sets the lock bits.

```
uisp -dprog=dapa -dpart=atmega16 -dlpt=0x378 –erase
uisp –verify -dprog=dapa -dpart=atmega16 -dlpt=0x378 –upload if=$1.hex
uisp -dprog=dapa -dpart=atmega16 -dlpt=0x378 –wr_lock=0xfe
```

1.4 Digital Input/Output

Atmega16 has 32 pins configured as four ports named A, B, C and D, each 8 bit wide. Their direction and data transfer can be controlled by using the registers DDRX, PORTX and PINX (where X stands for A, B, C or D). The AVRGCC compiler allows us to access the registers just like normal variables. For example, the statement DDRB = 15 , writes the number 15 to register DDRB.

- DDRX : Every pin of an I/O port can be configured as Input or Output using the Data Direction registers DDRX. To configure a pin as output, set the corresponding bit in DDRX. For example, DDRA = 3 will configure Bit 0 and Bit 1 of Port A as outputs.
- PORTX : For pins that are configured as ouputs, assigning a value to PORTX will set that data on them. For pins that are configured as inputs, setting the bits in PORTX will enable the corresponding internal pullup resistor.
- PINX : For the pins configured as inputs, PINX will read the status of voltage level at the pins. For pins that are configured as outputs, PINX will return the data written to PORTX.

The operations described above can be understood easily with some examples. In our hardware, we have an LED connected to Bit 0 of Port B. The program copy.c reads the voltage level at PA0 (Pin 0 of Port A) and sets the same on PB0, where we have connected the LED. We will pull up PA0 to 5V internally and it will go LOW only when you connect it to ground using a piece of wire.

Example copy.c

```
#include <avr/io.h>
  int main (void)
  {
  uint8_t  val;
  DDRA = 0;             // Port A as Input
  DDRB = 1;             // Pin 0 of Port B as output
  for(;;)
     PORTB = PINA;
  }
```

Compile and upload copy.c using the commands:

```
$./compile copy
$./upload copy
```

The LED should start glowing after uploading the program. LED will be off when you connect PA0 to ground. You may rewrite the program so that the LED may be controlled by some other bit configured as input.

The simple program given above has certain drawbacks. It changes PORTB as a whole instead of acting on Bit 0 alone. Suppose we had something else connected to the other pins of Port B, they also will be affected by $PORTB = PINA$. Here it does not make much difference since we are not using the other bits of Port B. If we want to preserve the status of other 7 bits of Port B , we may modify the for loop as

```
stat = PINA & 1;    # get only PA0
val = PORTB;        # current value of Port B
val &= ~1;          # clear Bit 0 only
val |= stat;        # Set it if PA0 is HIGH
PORTB = val;        # change Port B
```

The code fragment shown above uses the Bitwise AND, XOR and OR operators.

Another simple program *blink.c* that makes pin PB0 HIGH and LOW in a closed loop so that the LED connected to it will blink, is listed below.

Example blink.c

```
#include <avr/io.h>
void delay (uint16_t k)
   {
   volatile uint16_t x = k;
   while (x) --x;
   }
int main (void)
{
   DDRB = 1;     // Data Direction Register
   for(;;)
```

```
    {
    PORTB = 1;
    delay(30000);
    PORTA = 0;
    delay(30000);
    }
}
```

If everything goes fine, you should see the LED blinking. The delay routine is required to make the switching slow, else we will not be able to see it. You can remove the delay and watch the high frequency pulses on PB1 using an oscilloscope.

1.5 Selecting a Clock Source

A micro-controller require a clock source for its operation. ATmega16 is set to use the internal RC oscillator at 1 MHz from the factory. That was good enough for our example programs but many applications may require an external crystal and higher frequency clocks. For ATmega16, there are several internal and external clock options that can be selected by programming the fuse bits (can be done using the program uisp). The fuse bytes used for selecting the clock source and frequency are shown in figure 1.3. The factory setting of Fuse Low Byte is 0xE1. When used with an external crystal, we set it to 0xEF.

The batch file *set_fuse* sets the fuses for external crystal oscillator. *DO NOT run this program without changing the Fuse Low Byte from 0xEF , if there is no crystal connected between pins 12 and 13.* The JTAG feature is disabled so that the pins PC2 to PC5 can be used for normal Digital Input/Output.

```
echo "Setting fuses for ATmega16: External Crystal, disable JTAG"
uisp -dprog=dapa -dpart=atmega16 -dlpt=0x378 --erase
uisp -dprog=dapa -dpart=atmega16 -dlpt=0x378 --wr_fuse_l=0xef
uisp -dprog=dapa -dpart=atmega16 -dlpt=0x378 --wr_fuse_h=0xd9
uisp -dprog=dapa -dpart=atmega16 -dlpt=0x378 --rd_fuses
```

For more details on fuse bit setting refer to the ATmega16 manual. It is also possible to set the lock bits so that the program cannot be erased until the lock bits are reset.

Fuse Low Byte	Bit No.	Description	Default Value
BODLEVEL	7	Brown-out Detector trigger level	1 (unprogrammed)
BODEN	6	Brown-out Detector enable	1 (unprogrammed, BOD disabled)
SUT1	5	Select start-up time	1 (unprogrammed)(1)
SUT0	4	Select start-up time	0 (programmed)(1)
CKSEL3	3	Select Clock source	0 (programmed)(2)
CKSEL2	2	Select Clock source	0 (programmed)(2)
CKSEL1	1	Select Clock source	0 (programmed)(2)
CKSEL0	0	Select Clock source	1 (unprogrammed)(2)

Fuse High Byte	Bit No.	Description	Default Value
OCDEN(4)	7	Enable OCD	1 (unprogrammed, OCD disabled)
JTAGEN(5)	6	Enable JTAG	0 (programmed, JTAG enabled)
SPIEN(1)	5	Enable SPI Serial Program and Data Downloading	0 (programmed, SPI prog. enabled)
CKOPT(2)	4	Oscillator options	1 (unprogrammed)
EESAVE	3	EEPROM memory is preserved through the Chip Erase	1 (unprogrammed, EEPROM not preserved)
BOOTSZ1	2	Select Boot Size (see Table 100 for details)	0 (programmed)(3)
BOOTSZ0	1	Select Boot Size (see Table 100 for details)	0 (programmed)(3)
BOOTRST	0	Select reset vector	1 (unprogrammed)

Device Clocking Option	CKSEL3..0
External Crystal/Ceramic Resonator	1111 - 1010
External Low-frequency Crystal	1001
External RC Oscillator	1000 - 0101
Calibrated Internal RC Oscillator	0100 - 0001
External Clock	0000

CKSEL3..0	Nominal Frequency (MHz)
0001(1)	1.0
0010	2.0
0011	4.0
0100	8.0

Figure 1.3: The LOW and HIGH fuse Bytes are shown in the first two tables. The third shows how to select the clock source using them. Table 4 shows how to select the frequency if the internal RC oscillator option is used.

Chapter 2

The Phoenix-MDK

The Phoenix Micro-controller Development KIT makes it easy to start developing code on ATmega16. The hardware makes all the I/O pins available on sockets. The built-in LCD display with software support makes development faster. Functions are written to access most of the periferal devices so that the beginner need not get in to the register details of them.

2.1 The Hardware

The basic hardware board has a socket mounted ATmega16, a 5V DC regulator circuit that accepts 9V unregulated DC. All the 32 I/O pins are labelled and available on sockets. A 6 pin connector is used for program uploading through the PC parallel port. A 16 character LCD display can be plugged in to a socket that is wired to PORT C. The hardware schematic of the basic board is shown in figure 2.1. The versions with RS232 / USB interfaces to the PC will be described later.

2.2 The software

On the software side Phoenix-MDK provides you several functions that can be used to access various on-chip periferals of ATmega16 and the 16 character LCD display. To make the source code visible to the user, they are not compiled as a library. The functions are provided as source files that can be included into your own code. The source files providing the library functions are kept along with the example programs. To distinguish them, the names of the library files start with a pmdk_ , for example the LCD related functions are inside *pmdk_lcd.c* .

Reading the working source code along with the ATmega16 manual is a good way to understand how the periferals are programmed. The example program *hello.c* listed below shows how to use it.

Example hello.c

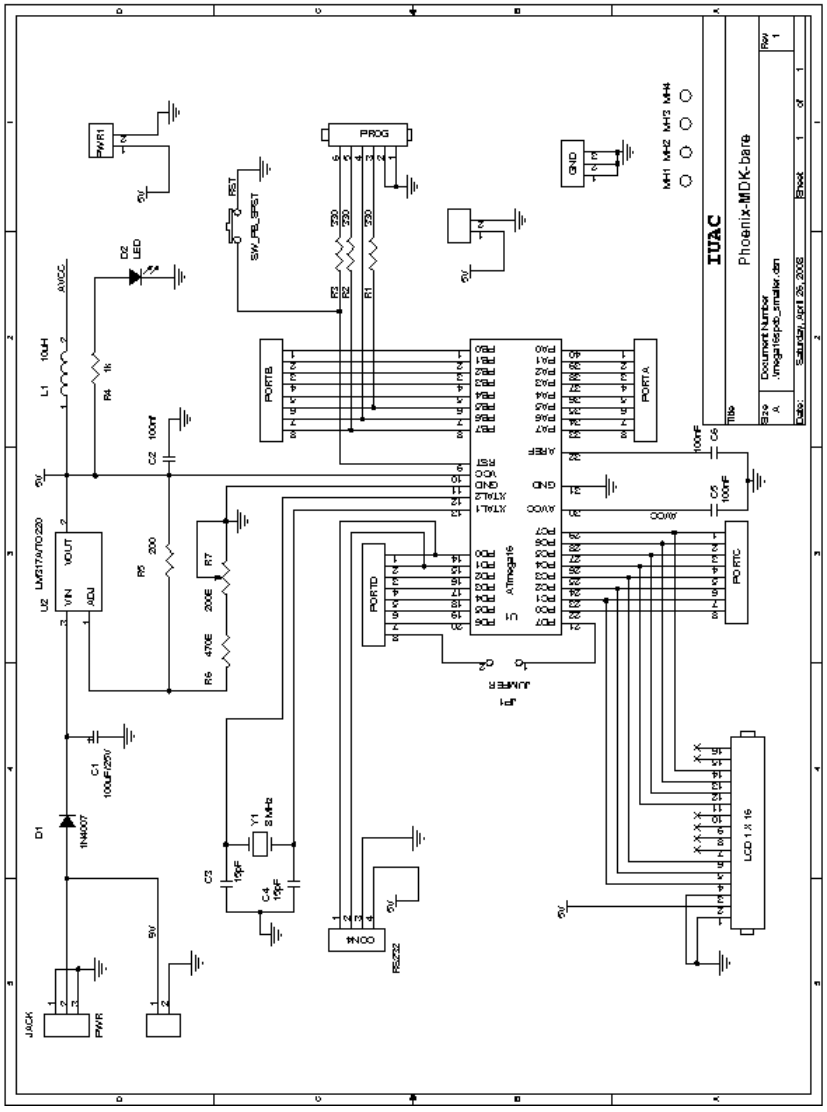

Figure 2.1: Circuit of the PMDK board.

```
#include "pmdk_lcd.c"
int main()
{
lcd_init();
lcd_put_string("Hello World");
}
```

The file pmdk_lcd.c provides the following functions :

- lcd_init() : Initializes the LCD display, must be called once in the beginning
- lcd_clear() : Clears the display
- lcd_put_char(char ch) : Outputs a single character to the LCD display
- lcd_put_string(char* s) : Displays a string to the LCD
- lcd_put_byte(uint8_t i) : Diplays an 8 bit unsigned integer
- lcd_put_int(uint16_t i) : Displays a 16 bit unsigned integer

2.3 ATmega16 Periferals

2.3.1 Analog to Digital Converter

ATmega16 has eight channels of ADC inputs with 10 bit resolution. The ADC converts an analog input voltage to a 10-bit digital value through successive approximation. The minimum value represents GND and the maximum value represents the voltage on the AREF pin minus 1 LSB. Optionally, AVCC or an internal 2.56V reference voltage may be connected to the AREF pin by writing to the REFSn bits in the ADMUX Register. The internal voltage reference may thus be decoupled by an external capacitor at the AREF pin to improve noise immunity. The ADC operation is controlled by programming the registers ADMUX and ADCSRA. The data is read from ADCH and ADCL.

The PMDK library provides the following functions to use the ADC:

1. adc_enable() : Enables the ADC
2. adc_disable() : Disables the ADC
3. adc_set_ref(ref) : Select the reference, where ref is REF_EXT is an external voltage is applied to the AVREF pin, REF_INT to use the internal 2.56 V reference and REF_AVCC to connect the AVCC supply internally to AVREF.
4. read_adc(ch) : Converts the voltage on channel ch and returns it in a 16 bit number.

The example program *adc.c* , reads an ADC input and display the result on the LCD.

Example adc.c

```
#include "pmdk_lcd.c"
#include "pmdk_adc.c"
main()
{
uint16_t data;
lcd_init();
adc_enable();
data = read_adc(0);
lcd_put_int(data);
}
```

2.3.2 Timer/Counters

ATmega16 has two numbers of 8 bit counter/timers and one 16 bit timer counter. They can be operated in different modes. For details, refer to the manual. In this section we have the following functions that utilizes the Counter/Timers. They are all written for a CPU clock requency of 8 MHz and the accuracy depends on the accuracy of the crystal used.

1. set_frequency(freq) : Generates a square wave on PD7 (OC2)
2. set_voltage(dac) : Generates a 31.25 KHz square wave on PD7(OC2) whose dutycycle is decided by the 8 bit unsigned integer dac. Filtering this Pulse Width Modulated wave will result in a DC ranging from to 5V.
3. measure_frequency() : Returns the frequency of a square wave connected to PB0 (T0) in Hertz. This function counts the number of pulses for one second.

Connect PD7 to PB0 and upload the program freq.c to read the frequency on the LCD display.

```
#include "pmdk_timer.c"
#include "pmdk_adc.c"
#include "pmdk_lcd.c"
int main()
{
uint16_t fr;
lcd_init();
set_frequency(500);     // Generate square wave on PD7 (OC2)
fr = measure_frequency(); // Measure on pin PB0
lcd_put_int(fr);
}
```

2.3.3 Simplified Digital I/O

On the Phoenix-MDK board, all the 32 I/O pins are available on sockets and are grouped into Digital I/O pins and Analog Input pins. The Digital I/O pins are numbered from 0 to 20. The three pins (SCK, MISO and MOSI) used for program loading are excluded. Simple functions are written to cotrol/monitor these pins, and are listed below.

1. make_pin_input(pin) : value of pin from 0 to 20. Configure it as input.
2. make_pin_output(pin) : Configure as output.
3. enable_pullup(pin) : Enable the internal pullup if the pin is input.
4. disable_pullup(pin) : Disable the internal pullup on the pin.
5. set_high(pin) : Sets a HIGH on the pin, if it is an output pin.
6. set_low(pin) : Sets a LOW on the pin.
7. read_pin(pin) : Returns the status on the input pin, 0 or 1 depending on the voltage level.

The example program *dio.c* given below sets the pin Digital I/O pin number 13 to HIGH.

Example dio.c

```
#include "pmdk_digital.c"
main()
{
make_pin_output(13);
set_high(13);
}
```

2.4 RS232 Serial Communication to PC

ATmega16 supports a Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART), that can be used for communicating to a PC through the RS232 port of the PC. The HIGH and LOW logic levels used in RS232 communications are around -9V and +9V. We need to use a level shifter circuit to connect the 0 to 5V USART signals to the RS232 port of the PC. The PMDK-Serial board incorporates this circuits and has a 9 pin connecter compatible with the PC serial port. The circuit schematic is shown in figure 2.2.

The following functions are available for communicating to the PC. On the PC side, we use a simple Python program to communicate to the micro-controller. You need to install Python interpreter and the python-serial module on the PC for this to work.

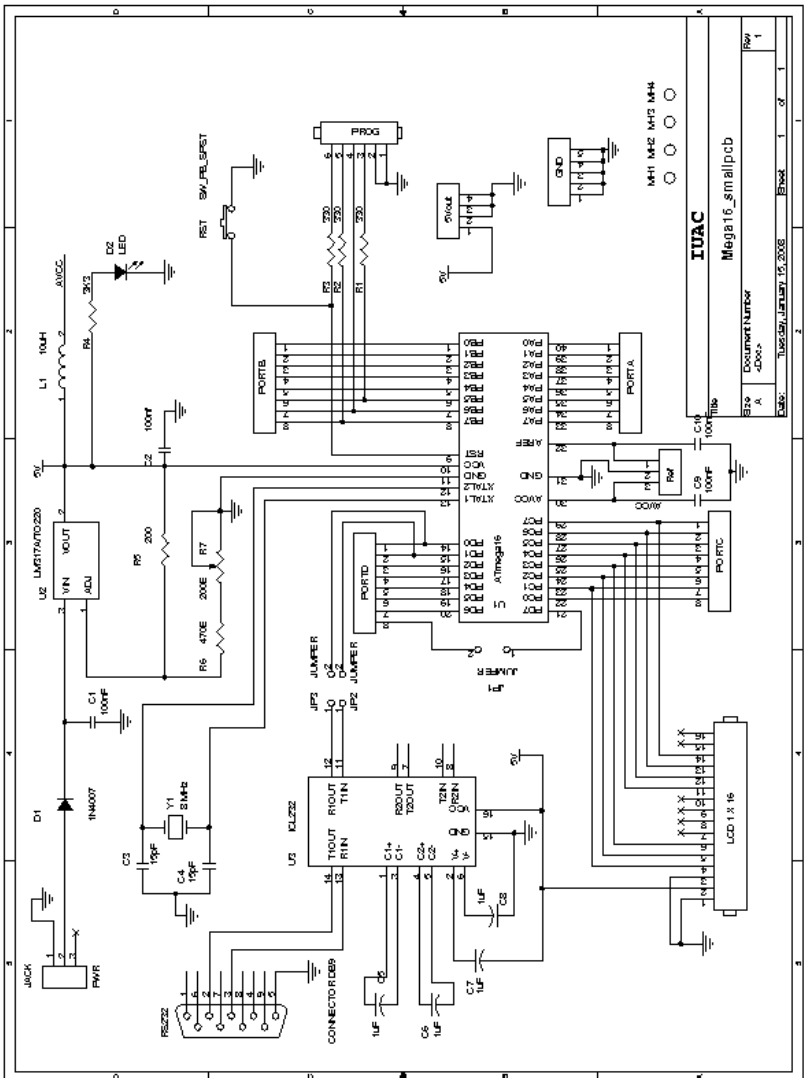

Figure 2.2: PMDK-Serial. Schematic of the Phoenix-MDK with RS232 level shifter and 9 pin connector.

1. uart_init(baud) : 38400 is the maximum baudrate supported. You can use any submultiple of that. We use 1 Stop Bit and the parity is Even.
2. uart_recv_byte() : Waits on the UART receiver for a character and returns it
3. uart_send_byte(c) : Sends one character over the UART transmitter.

The program *senddata.c* reads the ADC and converts the binary data in to a string using the sprintf function. This string is displayed on the LCD display and send over the RS232 port also.

Example hello.c

```
#include "pmdk_adc.c"
#include "pmdk_uart.c"
#include "pmdk_lcd.c"
main()
{
uint16_t data;
char   ss[10], *p;
lcd_init();
adc_enable();
uart_init(38400);
data = read_adc(0);
sprintf(ss,"%5d",data);
lcd_put_string(ss);
p = ss;
while (*p++)
     uart_send_byte(*p);
}
```

The data send by the micro-controller is received by the Python program *receive.py*. The baud rate, stop bits and parity should be set the same at both ends.

Example receive.py

```
import serial
ser = serial.Serial('/dev/ttyS0', 38400, stopbits=1)
count = 0
val = ''
while(1):
        while ser.inWaiting() == 0:
                pass
        x=ser.read()
        if ord(x) == 0:     #Print when end of string
                print val
                val = ''
        else:
                val = val + x
```

The example programs given below implements a two way transmission of data between the PC and ATmega16. Every character send from the PC is diplayed on the terminal and also send back to the PC. The programs *echo.c* and *rs232echo.py* are listed below.

Example echo.c

```
#include "pmdk_lcd.c"
#include "pmdk_uart.c"
int main(void)
{
uint8_t data;
lcd_init();
uart_init(38400);
for(;;)
  {
    data = uart_recv_byte();
    lcd_put_char(data);
    uart_send_byte(data);
  }
}
```

Example rs232echo.py

```
import serial
fd = serial.Serial('/dev/ttyS0', 38400, stopbits=1, \
    timeout = 1.0, parity=serial.PARITY_EVEN)
while 1:
  c = raw_input('Enter a character : ')
  fd.write(c)
  print 'Receiced ', fd.read()
```

2.5 USB Communication to PC

Most of the PCs available today have USB ports and no RS232. The Atmega16 does not have any built-in USB interface. We need to use a USB to Serial converter in-between to connect ATmega16 to the USB port of a PC. PMDK-USB does it by using an ATmega8 micro-controller running the firmware written by Igor Cesko. The circuit schematic is shown below.

The communication software changes only on the PC side, instead of the Python serial module, we need to use the Python USB module and send the characters to the port. The program *usbecho.py* is listed below. The included class file is available on the website along with the other example programs.

```
import avr309, sys
con = avr309.avrusb()
if con.fd == None:
  print 'AVR309 USB to Serial Adapter not found. Exiting'
```

Figure 2.3: PMDK-USB with LCD Display

```
  sys.exit()
con.setbaud(38)          # 38 for 38400 and 12 for 115200
while 1:
    c = raw_input('Enter a character : ')
    con.write(ord(c))
    print chr(con.read_one())
```

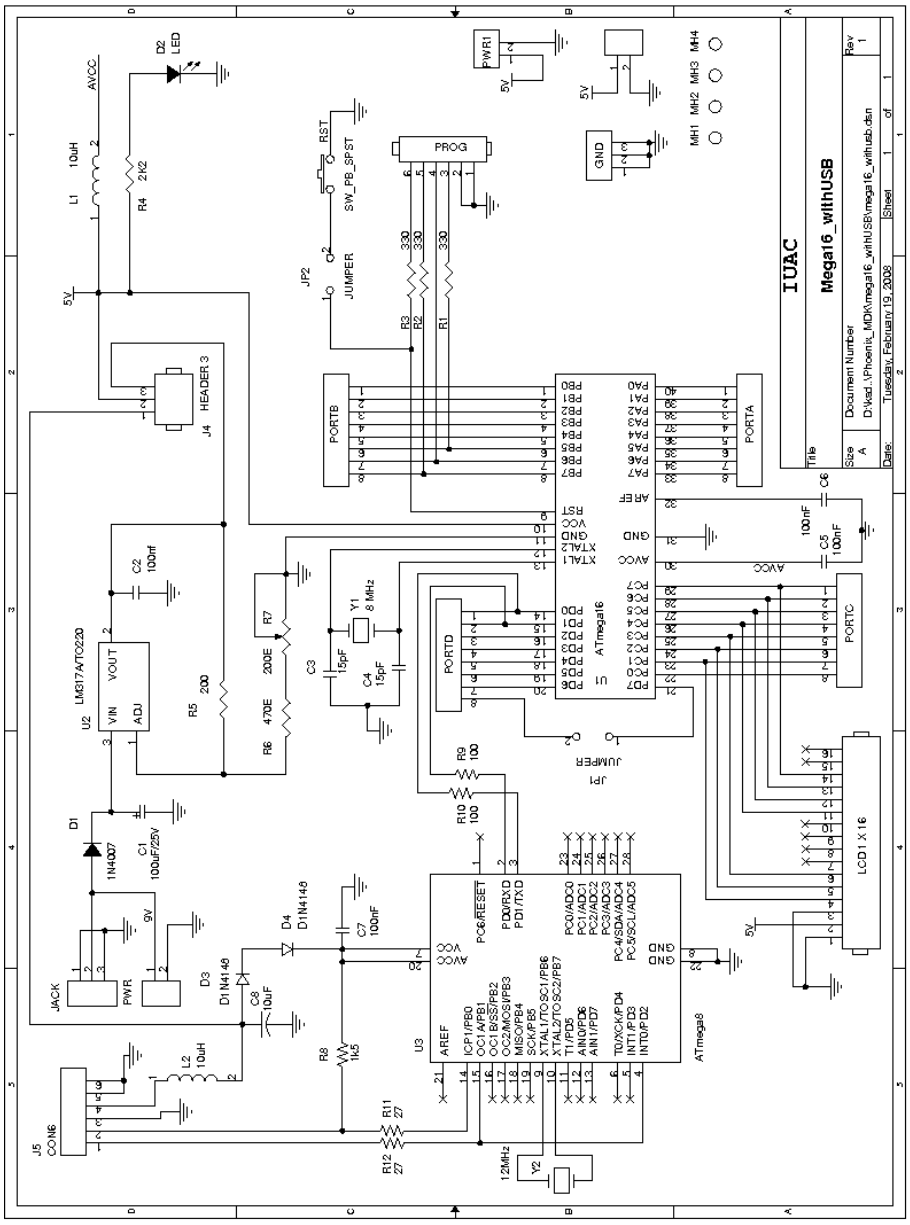

Figure 2.4: PMDK-USB schematic. The communication to the PC USB port is done through the USBtoSerial converter implemented using ATmega8.