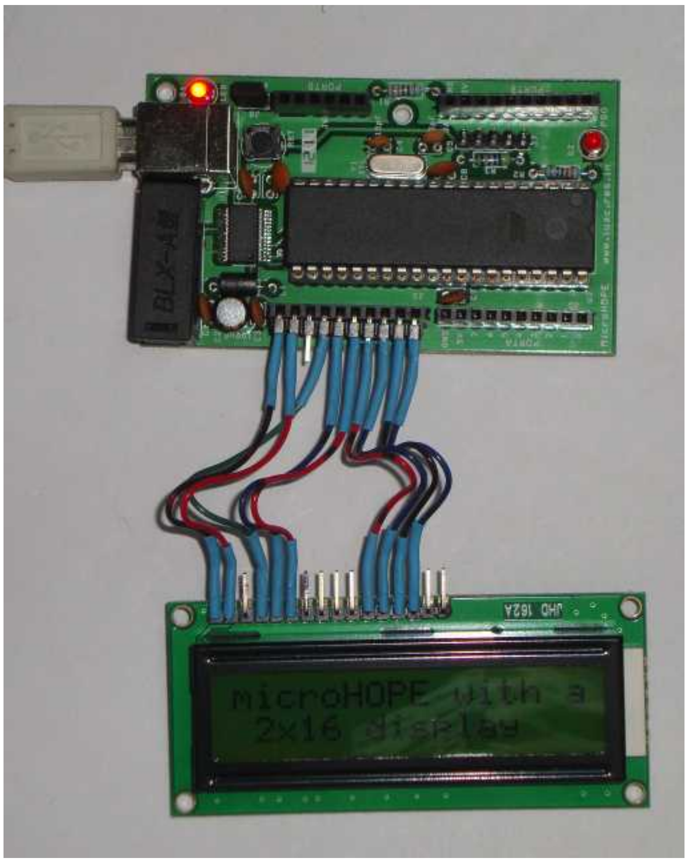
# microHOPE



User's Manual

# Micro-controllers for Hobby Projects and Education

Ajith Kumar B P
ajith@iuac.res.in

Inter-University Accelerator Centre
(A Research Centre of UGC)
New Delhi 110 067
http://expeyes.in

# Chapter 1

# Introduction

Most of computer systems in use today are embedded in other machinery, such as automobiles, telephones, appliances, and peripherals for computer systems. Most of them need minimal processing and memory requirements and can be implemented using micro-controllers (uC). A micro-controller is a small computer on a single integrated circuit consisting of a CPU combined with program and data memory, peripherals like analog to digital converters, timer/counters, serial communication ports and general purpose Input/Output ports. Intel 8051, Atmel AVR, PIC etc. are popular micro controllers available in the market. To design the microHOPE hardware, we have chosen ATmega32 micro-controller from Atmel, after considering the hardware resources available on it and the software support of GNU C compiler.

**Why microHOPE ?**

Many people who write programs that run on a PC find it difficult to get started on coding for a microcontroller, mainly due to two reasons: (1) Programming a uC require some knowledge about the target hardware. (2) Transferring the program to the target device generally require some special hardware and software. There are plenty of micro-controller development kits in the market, but most of them focus on the usage of their kit rather than the details of the the micro-controller. Programming the I/O pins of a development board using the library functions provided may be acceptable to get something done quickly, but our method is to directly deal with the the micro-controller, without hiding the uC details from the user.

A simple Graphical User Interface is provided to Edit, Compile and upload the program. We start by programming the Input/Output ports of Atmega32, which require some basic knowledge of binary number system, C language with its bit manipulation operators. After that we will proceed to the programming of the peripherals like ADC, Timer/Counter etc. Since they are more complex, we will start with a software library, in the form of C source files, that can be included in your program[1]. Once you learn how to write code peripherals, using their control/data registers, there is no need to use these functions.

Since microHOPE comes with a bootloader pre-installed inside the program memory of Atmega32, you can upload code using the USB interface with a single click, from the GUI provided. At the same time, executing the compile and upload programs from a text

[1]We are very much aware of the drawback of this method. When you include a file all the functions in that will get added to your executable, increasing its size. Once the code is working, copy the necessary functions to your source file, instead of including the whole file, to get rid of this difficulty.

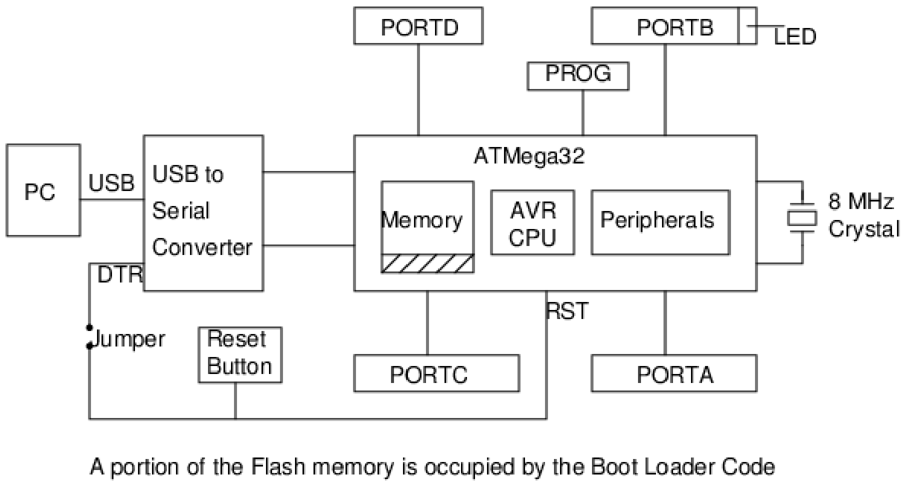


Figure 1.1: (a)MicroHOPE Block diagram.

terminal are also explained. For compiling the C program we use the **avr-gcc** compiler and **avrdude** for uploading it to the target.

## 1.1 The Hardware

A block diagram of microHOPE hardware is shown in figure 1.1. Programs can be uploaded from the PC through the USB port, using the pre-loaded boot loader code on the uC. To load a new program, the PC asserts the DTR signal of FT232, that resets ATmega32. On reset, the boot loader code will start to receive the new code from PC. After loading the new code, control is transferred to it.

Atmega32 has 32 Input/Output pins, organized as 4 ports, each 8 bit wide. The IC is available in DIP package, that can be socket mounted. The ATmega32 has 32 kB of Flash memory, 512 bytes EEPROM and 2 kB Static RAM. Three Timer/Counters, 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 peripheral devices on the chip.

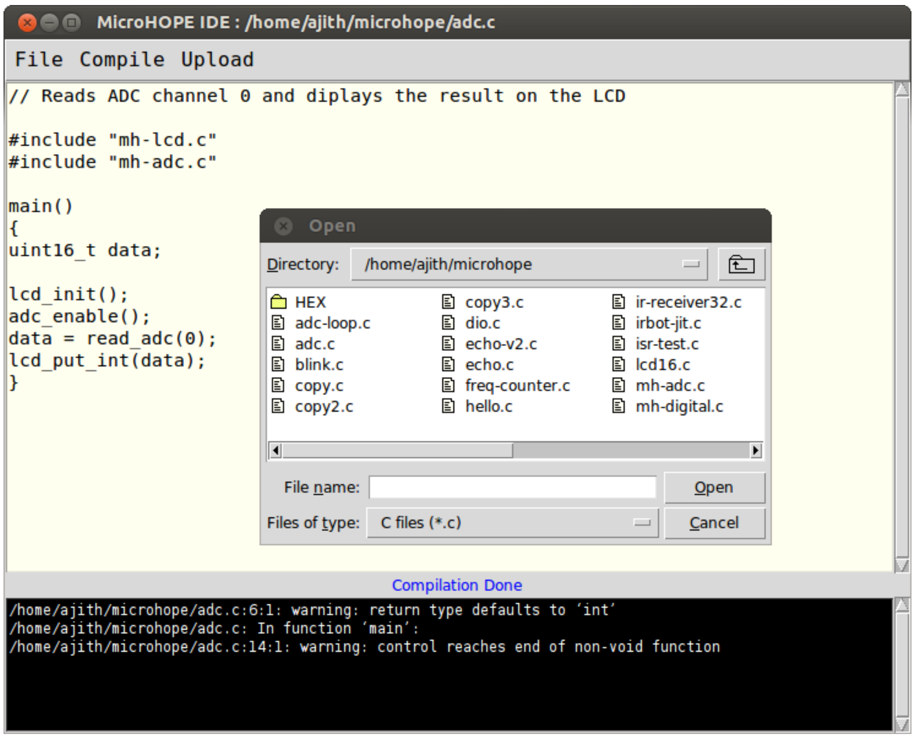
The processor on the microHOPE board runs at 8MHz, using the external crystal. All the I/O pins, except the UART Rx/Tx pins, are available to the user on the four I/O connectors. An LED is connected to Bit 0 of Port B, for quick testing of the board. A reset button is also provided. The 5V USB power is connected to both VCC and AVCC inputs. A jumper is provided to disable the reset option from the PC. This is required when the final product is used for communicating to a PC.

An alphanumeric LCD display is available as an accessory to microHOPE, to help the program development. It can be connected to the PORTC socket and C functions are provided to access the display.

## 1.2 The MicroHOPE Software

An Integrated Development Environment (IDE) is available for writing, compiling and uploading programs. It is available in .deb format, microhope-2.0.0.deb. To install it, under Debian/Ubuntu, open a Terminal and follow the steps given below:

```
$ sudo -s
```



```
// Reads ADC channel 0 and diplays the result on the LCD

#include "mh-lcd.c"
#include "mh-adc.c"

main()
{
uint16_t data;

lcd_init();
adc_enable();
data = read_adc(0);
lcd_put_int(data);
}
```

```
/home/ajith/microhope/adc.c:6:1: warning: return type defaults to ‘int’
/home/ajith/microhope/adc.c: In function ‘main’:
/home/ajith/microhope/adc.c:14:1: warning: control reaches end of non-void function
```

Figure 1.2: MicroHOPE User Interface

```
# apt-get install gdebi
# gdebi microhope-2.0.0.deb
```

After installing the package, drag the directory **/etc/skel/microhope** to your home directory. The program can be started by choosing Science->microHOPE from the Applications menu. A screen shot of the microhope IDE is shown in figure 1.2. By default it looks for files inside the microhope subdirectory inside your home directory. The IDE provides a menu to load/save files, compile and upload the program.

All the examples given in this document will appear inside the directory named 'microhope'. All files starting with **mh-** are the files containing library functions to access the various peripherals of Atmega32. To make the source code visible to the user, they are not compiled as a library file. Do not modify the files starting with **mh-**.

You can select any of the example programs, compile and upload them using the menu. Correct all the compile errors before doing Upload. On GNU/Linux systems, the device file created for microHOPE hardware should be '/dev/ttyUSB0'.

### 1.2.1 Compilation and Upload, without the IDE

Even though the IDE does the job, it is a good idea to learn about the programs used behind the seen, to compile and upload the code. **Those who like to use only the IDE may skip this section**. The software packages used are:

- avr-gcc : To compile the C program, also require the C library avr-libc
- avr-objcopy : To generate the HEX file
- avrdude : To upload the Hex file

These packages are available under GNU/Linux. For Debian/Ubuntu distributions they can be installed from the repository using the commands:

```
# apt-get install avr-libc
# apt-get install avrdude
# chmod u+s avrdude
```

Insatlling avr-libc, automatically installs gcc-avr and other required packages. The last command will enable non-root users to use avrdude. The installed programs can be invoked from the command line. Use a text editor to create your source program, for example **blink.c**, and compile it using:

```
$ avr-gcc  -Wall -O2 -mmcu=atmega32 -o blink blink.c
```

We have asked the compiler to print all the warnings, optimize to level 2, generate code for atmega32. The executable output stored in blink and input taken from blink.c. The executable file is converted into Intel Hex format using the following command:

```
$ avr-objcopy -j .text -j .data -O ihex blink blink.hex
```

The Hex file is now ready for upload. This can be done using the command:

```
$ avrdude -b 19200 -P /dev/ttyUSB0 -pm32 -c stk500v1 -U flash:w:blink.hex
```

We have specified a baudrate of 19200, the output device is /dev/ttyUSB0, m32 processor and the transfer protocol stk500v1.

#### 1.2.1.1 Some batch files

Since a lot of command line arguments are required to specify the compiler, linker and loader options, it is convenient to put them in small batch files or shell scripts. These files can be found inside the **microhope** directory, once the package is installed. The compilation of C code and generation of Intel Hex format file for uploading is done by **compile-mega32.sh**, listed below.

```
$ avr-gcc -Wall -O2 -mmcu=atmega32 -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
```

For example, to compile a program named 'hello.c', it should be invoked from the command line as;

```
$./compile-mega32.sh hello
```

The .c extension should not be specified. The script also generates the linker MAP file and a listing file, that may be used for examining the generated output.

Under GNU/Linux, microhope on the USB port will appear as file '/dev/ttyUSB0' and program uploading is done by **mh-upload.sh**, listed below

```
$ avrdude -b 19200 -P /dev/ttyUSB0 -pm32 -c stk500v1 -U flash:w:$1.hex
```

To upload hello.hex, use the command

```
$./mh-upload hello
```

### 1.2.2 Using under Raspberry Pi

The required packages are available on Raspberry Pi. Install and use as explained in section 1.2.1

### 1.2.3 Using under MS Windows

Download **winavr** package from sourceforge.net and install it.

For FT232 version, install the VCP drivers from

http://www.ftdichip.com/Drivers/VCP.htm

For MCP2200 version, install the ACM drivers from

http://www.expeyes.in/sites/default/files/debs/MCP2200-driver.zip

Download and install the MicroHOPE IDE from expeyes website

### Running from command prompt

Use a text editor like notepad to edit the source program and save it with a .c extension. The commands for compilation and uploading are:

```
C:\> avr32-gcc  -Wall -O2 -mmcu=atmega32 -o blink blink.c
C:\> avr32-objcopy -j .text -j .data -O ihex blink blink.hex
C:\> avrdude -b 19200 -P COMxx -pm32 -c stk500v1 -U flash:w:blink.hex
```

Where COMxx is the virtual com port number assigned by Windows. We have found it very difficult due to the arbitrary naming of COM ports and reliability issues in the hardware communication.

# Chapter 2

# Getting Started

After installing the Debian package, you must have copied the directory /etc/skel/microhope to you home directory. Start the microHOPE IDE from the menu Applications->Science. Choosing File->Open from the menubar will display all the C files inside the **microhope** directory. You can open any of the examples (except the filenames starting with mh-) , compile and upload it by clicking on the menubar. We will start by programming the Digital Input/Output ports of Atmega32, and them proceed to the peripheral devices. [1].

The avr-gcc compiler allows us to program the control/data registers and their individual bits using the same names given in the Atmega32 manual. The registers are accessed just like normal variables, the statement PORTB = 15 , writes the number 15 to Port B. The individual pins are referred using names like PA0, means BIT 0 of PORTA.

## 2.1 Testing the Hardware

Connect MicroHOPE hardware to a USB port and start the microHOPE IDE from the menu Applications->Science. The program will display the status of the hardware detected, it should appear as '/dev/ttyUSB0' in most cases. If you get an error message, reconnect the hardware and restart the program.

Using File->Open from the menubar, load **blink.c** from the **microhope** directory. Compile and Upload the program by clicking on the menu labels. It should give a message 'Upload Completed'. If not, check the USB connections first. If problem persists, try pressing and releasing the microHOPE resent button at the same time when you click on Upload.

Once the program is uploaded, the LED connected to PB0 should blink at 1 Hz rate. **If not, press the reset button on the board.**

## 2.2 Input/Output ports of Atmega32

The pinout diagram of Atmega32 is shown in figure 2.1. There are 32 pins organized as four ports named A, B, C and D, each 8 bit wide. Each pin can be configured as an input or output. The data direction and transfer are done by writing to the registers DDRX, PORTX and PINX (where X stands for A, B, C or D).

- 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

[1]For complete details of Atmega32 refer to http://www.atmel.in/Images/doc2503.pdf

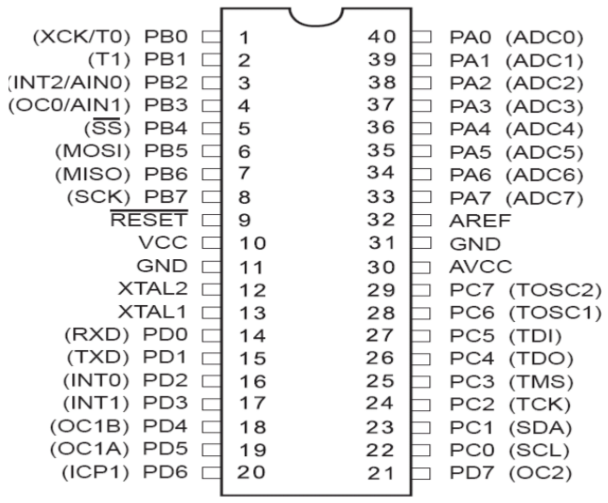


Figure 2.1: Atmega32 Pinout

bit in DDRX, and clear it to make the corresponding pin as input. For example, DDRA = 1 will configure BIT 0 of Port A (PA0) as output, and all other pins as input.

- PORTX : For pins that are configured as ouput, assigning a value to PORTX will set that data on them. For example PORTA = 1 will make PA0 high, that can be measured on the pin number 40 of the IC.
- PINX : For the pins configured as inputs, PINX will read the status of the external voltage level connected to the pins. For pins that are configured as outputs, PINX will return the data written to PORTX.

If the pins configured as inputs are left unconnected, there could be unwanted level changes due to electrical noise, this can be prevented by enabling the internal pull-up resistor. For pins that are configured as inputs, setting/clearing the bits in PORTX will enable/disable the corresponding internal pullup resistor.

The operations described above can be understood easily with some examples. For a quick test, MicroHOPE hardware has an LED connected to PB0, with a series resistor for current limiting.

## 2.2.1 Reading and Writing Ports

The program **copy.c** reads the voltage level at PA0 (Pin 0 of Port A) and sets the same on PB0, where we have the LED. We will enable the internal pullup resistor on PA0 so that and it will go LOW only when it is connected to ground using a piece of wire.

```
#include <avr/io.h>   // Include file for I/O operations
int main (void)
{
DDRA = 0;             // Port A as Input
PORTA = 1;            // Enable pullup on PA0
DDRB = 1;             // Configure PB0 as output
for(;;)
     PORTB = PINA;   // Read Port A and write it to Port B
}
```

To test this example, open **copy.c** from the File menu of microHOPE IDE, Click on Compile and then Upload from the menubar The LED on PB0 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 PB0 alone. Suppose we have something else connected to the other pins of Port B, they also will be affected by the action $PORTB = PINA$. To avoid such problems, we should manipulate individual bits. The include file **mh-digital.c** contains macros for setting and clearing bits by specifying their position.

### 2.2.2 Bit manipulation macros

These macros can be used on variables, defined in the program, and also on registers like DDRX, PORTX etc.

#### BITVAL(bit position)

The value of bit position could be 0 to 7 in the case of 8 bit integers and 0 to 15 for 16 bit integers. This macro returns (1 << bit position). For example BITVAL(3), will give 8, that is binary 1000, obtained by left shifting of 1 thrice.

#### SETBIT(variable, bit position)

This macro SETS the specified bit in the given variable, without affecting the other bits. For example SETBIT(DDRB, 7), will make the last bit of DDRB high.

#### CLRBIT(variable, bit position)

This macro clears the specified bit of the given variable. For example CLRBIT(val, 0), clears the least significant bit of 'val', that is an integer type variable.

#### GETBIT(variable, bit position)

This macro returns the value the specified bit if the specified bit of the variable is 1, else it returns zero. For example: if x = 3, GETBIT(x, 1) will return 2 and GETBIT(x,3) will return zero.

Let us rewrite the previous program as **copy2.c**, using these macros as:

```
#include <avr/io.h>
  int main (void)
  {
  uint8_t  val;
  DDRA = 0;             // Port A as Input
  PORTA = 1;            // Enable pullup on PORTA, bit 0
  DDRB = 1;             // Pin 0 of Port B as output
  for(;;)

     {
        val = GETBIT(PORTA, 0);

        if (val != 0)
           SETBIT(PORTB, 0);
        else
            CLRBIT(PORTB, 0);
        }

    }
```

The same can be done, without using the bit manipulation macros, as shown in **copy3.c**

```
#include <avr/io.h>    // Include file for I/O operations
int main (void)
{
uint8_t  val;          // 8 bit unsigned word
DDRA = 0;              // Port A as Input
PORTA = 1;             // Enable pullup on PA0
DDRB = 1;              // Configure PB0 as output
for(;;)
   if(PINA & 1)           // If PA0 is set
       PORTB |= 1;        // Set PB0, by ORing with 00000001b
   else                   // otherwise clear PB0
       PORTB &= ~1;       // by ANDing with 11111110b (~00000001b)
}
```

The code fragment shown above uses the Bitwise AND, OR and NOT operators.

### 2.2.3 Blinking LED

Making pin PB0 HIGH and LOW in a closed loop result in the blinking of the LED conencted to it. We need to slow down the rate of blinking so that it can be perceived by our eyes. This can be done by making the processor wait for a while between writing to PORTB. There are some delay functions provided for this. The file **mh-utils.c** contains the following functions:

#### delay_100us(int n)

This function will make the CPU idle for n x100 microseconds. For example to insert a 200 microsecond delay, call delay_100us(2)

#### delay_ms(int n)

This function will make the CPU idle for n milliseconds. For example to insert a 500 millisecond delay, call delay_ms(500)

The program blink.c lis listed below:

```
#include "mh-utils.c"
int main (void)
{
   DDRB = 1;     // configure PB0 as output
```

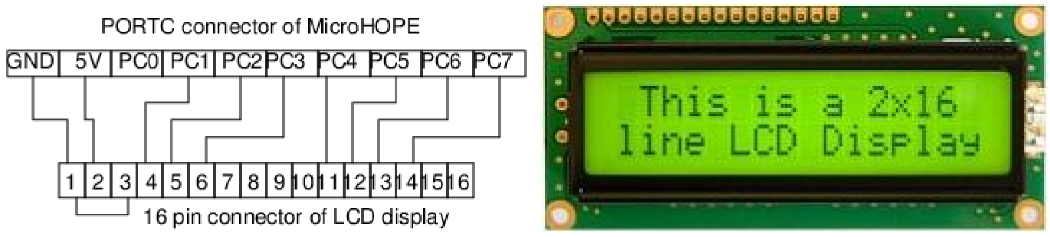

PORTC connector of MicroHOPE
GND 5V PC0 PC1 PC2 PC3 PC4 PC5 PC6 PC7
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
16 pin connector of LCD display
This is a 2x16
line LCD Display

Figure 2.2: LCD display. (a) connections to microHOPE (b) Photograph of 2x16 character display

```
    for(;;)
    {
    PORTB = 1;
    delay_ms(500);
    PORTB = 0;
    delay_ms(500);
    }
}
```

If everything goes fine, you should see the LED blinking. You can remove the delay statements and watch the high frequency pulses on PB0 using an oscilloscope.

## 2.3 The LCD Display

For many applications, it is necessary to have a local display. The HD44780 controller, or compatible IC, based displays are widely available. They come with a 16 pin connector and the transfer protocol is well documented [2]. The connections between microHOPE and the LCD display are shown in figure 2.2(a). Pins 4,5 and 7 of the LCD display are control lines, connected to PC1, PC2 and PC4. The ASCII codes are transferred in the 4bit mode, using pins 11 to 14 connected to PC4, PC5, PC6 and PC7. The file 'mh-lcd.c' contains functions to access the display. Most of the examples given below make use of the LCD display.

The example program **hello.c** listed below demonstrates the usage of the LCD display.

```
#include "mh-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

[2]For details refer to http://en.wikipedia.org/wiki/Hitachi_HD44780_LCD_controller

- 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
- lcd_put_long(uint32_t i) : Displays a 32 bit unsigned integer

## 2.4 Analog to Digital Converter

Most of the I/O PORT pins of Atmega32 has alternate functions. PA0 to PA7 can be used as ADC inputs by enabling the built-in ADC. All the pins configured as inputs in the DDRA will become ADC inputs, but the ones configured as outputs will remain as digital output pins. The ADC converts the analog input voltage in to a 10-bit digital value. The minimum value represents GND and the maximum value represents the ADC reference voltage. The reference inputs could be AVCC, an internal 2.56V or a voltage connected to the AREF pin. The selection is done in software. The ADC operation is controlled via the registers ADMUX and ADCSRA. The data is read from ADCH and ADCL.

The include file 'mh-adc.c' provides the following functions:

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.

### 2.4.1 Reading an Analog Voltage

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

```
#include "mh-lcd.c"
#include "mh-adc.c"
main()
{
uint16_t data;
lcd_init();
adc_enable();
data = read_adc(0);
lcd_put_int(data);
}
```

### 2.4.2 Programmig ADC registers

The operation of the ADC is controlled mainly by the registers ADCSRA and ADMUX. Setting ADEN will enable the ADC and setting ADSC will start a conversion. The bit ADIF is set after a conversion and this bit can be cleared by writing a '1' to it. The

ADSP bits decide the speed of operation of the ADC, by pre-scaling the clock input. The channel number is selected by the MUX0 to MUX4 bits in the ADMUX rregister. The reference input is selected by the REFS0 and REFS1 bits.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| ADEN | ADSC | ADATE | ADIF | ADIE | ADPS2 | ADPS1 | ADPS0 |
| R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| REFS1 | REFS0 | ADLAR | MUX4 | MUX3 | MUX2 | MUX1 | MUX0 |
| R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

The program adc-v2.c, demonstrates the usage of these registers.

```
#include <avr/io.h>
#include "mh-lcd.c"
// convert channel 0, set pre-scaler to 7
main()
{
uint16_t data;
lcd_init();
ADCSRA = (1 << ADEN) |  7;      // Enable ADC, set clock pre-scaler
ADMUX =  (1 << REFS0);   // AVCC reference, channel 0
ADCSRA |=  (1 <<ADSC);          // Start ADC
while ( !(ADCSRA & (1<<ADIF)) ) ;  // wait for ADC conversion
data = (ADCH << 8) | ADCL;    // make 10 bit data from ADCL and ADCH
lcd_put_int(data);
}
```

### 2.4.3 Reading in a Loop

The example program **adc-loop.c** , reads an ADC input in a loop and display the result on the LCD. If the input is left unconnected, the displayed value could be anywhere between 0 an 1023. Connecting PA0 to 5V will display 1023, the maximum output.

```
#include "mh-lcd.c"
#include "mh-adc.c"
#include "mh-utils.c"
main()
{
uint16_t data;
lcd_init();
adc_enable();
for (;;)
    {
     data = read_adc(0);
      lcd_clear();
     lcd_put_int(data);
      delay_ms(100);
    }
}
```

Modify the code for reading other ADC channels.

### 2.4.4 Temperature Controller

The program adc-loop.c can be easily modified to make a temperature monitor/controller using the LM35 temperature sensor. Connect LM35 output to PA0. At 100$^0$C , the output of LM35 will be 1 volt. With the internal 2.56 volts as reference, the ADC output will be around 400 (1.0 / 2.56 * 1023).

Drive the relay contact controlling the heater from PB0, via a transistor. Insert the following line in the beginning

```
DDRB = 1
```

and within the loop:

```
if (data > 400)       // switch off heater

    PORTB = 0;

else if (data < 395)   // switch on heater

    PORTB = 1;
```

The heater will be switched OFF when the ADC output is greater than 400. It will be switched ON only when the output goes below 395. The window of 6 is given to avoid the relay chattering.

## 2.5 Timer/Counters

ATmega16 has three counter/timer units. Two of them are of 8 bit size and one is 16 bit. The counter input could be from derived from the internal clock or from an external source. The output of the counter is compared with setpoint registers and different types of actions are taken on compare match. The mode of operation of Counter/Timer is programmed by setting the bits in the control registers. These circuits are useful for time interval measurements and generating different kinds of waveforms.

### 2.5.1 8 bit Timer/Counter0

A block diagram of Timer/Counter0 is shown in figure2.3. The counter TCNT0 gets its input and control signals from the control logic circuit. The counter output is compared with a Output Compare Register OCR0 and a compare match can trigger different types of actions, like generating a waveform on OC0 (pin 4 of Atmega32, same as PB3). The mode of operation is decided by the register TCCR0, shown below:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| FOC0 | WGM00 | COM01 | COM00 | WGM01 | CS02 | CS01 | CS00 |
| W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

Let us start using Timer/Counter0 with the help of the following functions.

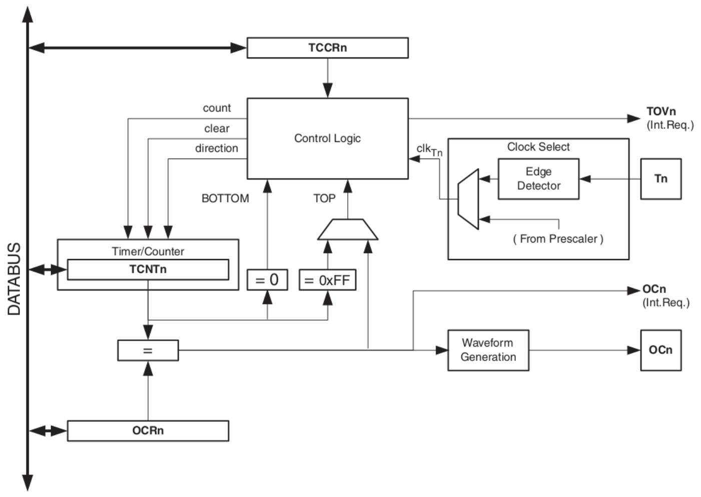


Figure 2.3: 8 bit Timer/Counter0 Schematic

### sqwave_tc0(csb, ocrval)

This function generates a square wave on OC0, whose frequency is decided by the clock select bits (csb) and ocrval. Example **sqwave-tc0.c** listed below demonstrates the usage of this function.

```
// example : sqwave-tc0.c
#include "mh-timer.c"
csb = 2;        // Clock select bits
ocrval = 99;    // Output Compare register vaule
int main()
{
sqwave_tc0(csb, ocrval);
}
```

The 8MHz system clock is divided by 8 (csb =2, refer to table below) to get a 1MHz input to the counter. The OCR0 register is set to 99. The mode bits are set such that the when the counter value reaches the OCR0, the output is toggled and counter is cleared. This will result in the waveform generator output toggles after every 100 clock cycles, giving a 5kHz sqaurewave on pin OC0 (PB3). You may view this on an oscilloscope. If you do not have one, connect a loudspeaker with a 100Ω series resistor from PB3 to ground. We have used expEYES for viewing and characterizing the waveforms generated by microHOPE.

Changing ocrval to 199 will give output 2.5kHz on the output. The output frequency is given by the relation

$$f = \frac{f_{clock}}{2.N.(1 + OCR0)}$$

where $f_{clock}$ is the system clock and $N$ is the clock division factor, as shown below.

| CS02 | CS01 | CS00 | Description |
|---|---|---|---|
| 0 | 0 | 0 | No clock source (Timer/Counter stopped). |
| 0 | 0 | 1 | $clk_{I/O}$/(No prescaling) |
| 0 | 1 | 0 | $clk_{I/O}$/8 (From prescaler) |
| 0 | 1 | 1 | $clk_{I/O}$/64 (From prescaler) |
| 1 | 0 | 0 | $clk_{I/O}$/256 (From prescaler) |
| 1 | 0 | 1 | $clk_{I/O}$/1024 (From prescaler) |
| 1 | 1 | 0 | External clock source on T0 pin. Clock on falling edge. |
| 1 | 1 | 1 | External clock source on T0 pin. Clock on rising edge. |

### pwm_tc0(csb, ocrval)

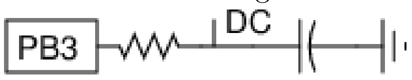
This function generates a Pulse Width Modulated waveform on OC0, whose frequency is decided by the clock select bits (csb) and the duty cycle by the ocrval. The output OC0 is cleared when the counter reaches the OCR0 value, the counter proceeds upto 255 and then sets OC0. The program **pwm-tc0.c** generates a 3.9 kHz PWM with 25% dutycycle.

```
// example : pwm-tc0.c
#include "mh-timer.c"
uint8_t  csb = 2;   // Clock select bits uint8_t
ocrval = 63;       // Output Compare register vaule
int main()
{
  pwm_tc0(csb, ocrval);
}
```

PWM waveforms are often used for generating analog DC voltages, in 0 to 5 volts range, by filtering it using an RC circuit. It is better to set a higher frequency so that the filter RC value could be small. The frequency can be made 31.25kHz by setting csb=1. The DC level is decided by the value of OCR0, ranging from 0 to 255. Once you learn howto manipulate the control registers, the same thing can be done without calling the library function, as shown below.

```
// example : pwm-tc0-v2.c
#include <avr/io.h>
uint8_t csb = 1;      // Clock select bits uint8_t
ocrval = 254/4;       // Output Compare register vaule
int main()
{
// Set TCCR0 in the Fast PWM mode
TCCR0 =(1 << WGM01) | (1 << WGM00) | (1 << COM01) | csb;
OCR0 = ocrval;
TCNT0 = 0;
DDRB |= (1 << PB3); // Set PB3(OC0) as output
}
```

Connect a 1k resistor and 100uF capacitor in series from PB3 to ground,as shown below, and measure the voltage across the capacitor using a voltmeter.



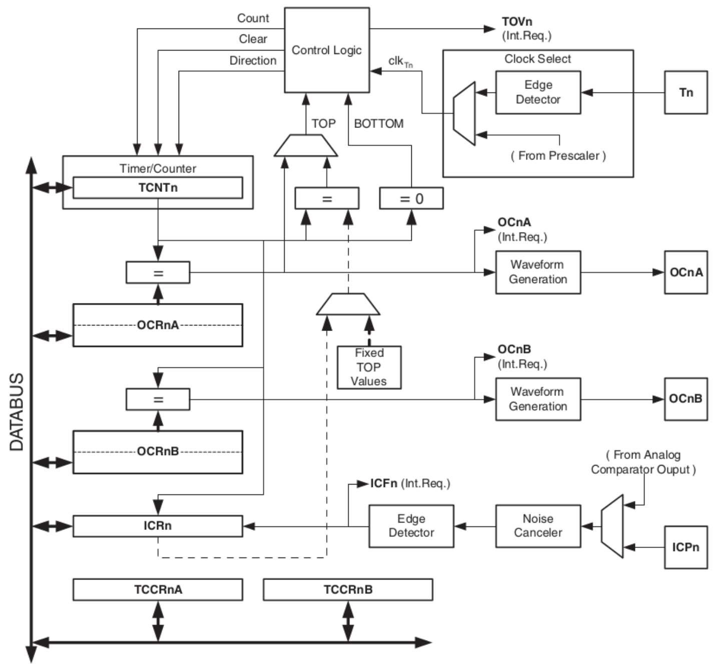


Figure 2.4: 16 bit Timer/Counter1 schematic

## 2.5.2 16 bit Timer/Counter1

The Timer/Counter1 has more features like two Output Compare Registers, Input Capture unit etc., as shown in figure2.4. The frequency and duty cycle of the waveforms can be controlled better due to the 16 bit size of the counters. Some C functions to use the T/C1 are given below.

**sqwave_tc1(csb, OCRA)**

```
// example : sqwave-tc1.c
#include "mh-timer.c"
uint8_t  csb = 2;       // 2 is divide by 8 option, 1MHz clock in
uint16_t  ocra = 50000;  // Output Compare register A
int main()

    {
    sqwave_tc1(csb, ocra);
    }
```

**pwm10_tc1(csb, OCRA)**

This function generates a PWM waveform with 10bit resolution. The value of ocra should be from 0 to 1023 to set the duty cycle.

```
// example : pwm-tc1.c
#include "mh-timer.c"
uint8_t  csb = 1;        // 1 => 8MHz clock in
uint16_t  ocra = 1024/3;  // Duty cycle arounf 33%
int main()
```

```
{
pwm10_tc1(csb, ocra);
}
```

### 2.5.3 8 bit Timer/Counter2

This one is similar to Timer/Counter0.

#### sqwave_tc2(uint32_t freq)

This function generates a square wave on OC2. The clock selction bits and the OCR2 value are calculated. It is not possible to set all frequency values using this method. The actual frequency set is returned and displayed on the LCD.

```
//Example sqwave-tc2.c
#include "mh-timer.c"
#include "mh-lcd.c"
int main()
{
uint32_t f;
lcd_init();
f = set_sqr_tc2(1500);
lcd_put_long(f);
}
```

#### PWM by programming the registers

The example given below demonstrates the usage of T/C2 as a PWM waveform generator, by setting the control register bits. The duty cycle is set to 25% by setting the OCR2 to one fourth of the maximum.

```
// example : pwm-tc2.c
#include <avr/io.h>
uint8_t csb = 2;      // Clock select bits uint8_t
ocrval = 255/4;       // Output Compare register vaule
int main()
{
// Set TCCR0 in the Fast PWM mode
TCCR2 =(1 << WGM21) | (1 << WGM20) | (1 << COM21) | csb;
OCR2 = ocrval;
TCNT0 = 0;
DDRD |= (1 << PD7); // Set PD7(OC2) as output
}
```

### 2.5.4 More applications of Timer/Counter

Timer/Counter can be used for timing applications, like measuring the time elapsed between two events or counting the number of pulse inputs during a specified time interval.

### measure_frequency()

This function counts the number of pulses received on the external input of Timer/Counter1 (PB1) during 500 milliseconds to calculates the frequency of the input pulse.

```
// Example freq-counter.c
#include "mh-utils.c"
#include "mh-timer.c"
#include "mh-lcd.c"
int main()
{
uint32_t f;
set_sqr_tc2(1500);    // Set a square wave on TC2 output (PD7)
lcd_init();
while(1)
   {
   f = measure_freq();
   lcd_clear();
   lcd_put_long(f);
   delay_ms(200);
   }
return 0;
}
```

Connect PD7 to PB1 and upload the program **freq-counter.c** to read the frequency on the LCD display. You can also connect PB1 to an external pulse source to measure its frequency. The maximum frequency that can be measured is limited by the size of the counter, that is 63535, means we it can handle upto around 126 kHz.

### Time Interval Measurement

The T/C units can be programmed to keep track of time interval between two events. The program **r2ftime.c** measures the rising edge to falling edge time on PB1.

```
// Example r2ftime.c
#include "mh-utils.c"
#include "mh-timer.c"
#include "mh-lcd.c"
int main()
{
lcd_init();
set_sqr_tc2(500);    // Test signal on PD7
while(1)
   {
   lcd_clear();
   lcd_put_long(r2ftime(PB1));
   delay_ms(100);
   }
}
```

The function r2ftime() uses two other functions, called start_timer() and read_timer(), that are explained below.

- void start_timer() : Start the counter with a 1 MHz clock input. An interrupt service routine is activated when the count reached 50000, that increments another interger.
- uint32_t read_timer() : Stops the counter and returns the microseconds elapsed after calling start_timer(). There will be an error of 2 to 3 microseconds, that is due to the overhead of the function calls.

#### 2.5.4.1 Distance Measurement

This technique is used for measuring distance using an ultrasound echo module HY-SRF05[3], using **ultra-sound-echo.c**. The trigger is connected to PB0 and the echo is connected to PB1. The distance is measured by

```
// Example ultra-sound-echo.c
#include "mh-utils.c"
#include "mh-timer.c"
#include "mh-lcd.c"
int vsby2 = 17;  // velocity of sound in air = 34 mS/cm
int main()
{
uint32_t x;
DDRB |=  (1 << PB0);  // set PB0 as output
DDRB &= ~(1 << PB1);  // and PB1 as inpt
lcd_init();
while(1)
   {
   PORTB |=  (1 << PB0);   // set PB0 HIGH
   delay_100us(1);
   PORTB &=  ~(1 << PB0);  // set PB0 LOW
   delay_100us(5);         // Wait for a while to avoid false triggering
   start_timer();
   while( (PINB & 2) != 0 ) ;   // Wait for LOW on PB1
   x = read_timer() + 400;
   lcd_clear();
   lcd_put_long(x*vsby2/1000);  // distance in cm
   delay_ms(500);
   }
}
```

## 2.6 Talking to the PC, via USB

On the microHOPE board, the Rx/Tx pins of ATmega32 are connected to the USB to Serial Converter IC. User programs also can use this path to communicate to the PC via the USB port.

[3]http://www.robot-electronics.co.uk/htm/srf05tech.htm

The following functions are available for handling the UART

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.

On the PC side, we use a simple Python program to communicate to the micro-controller. The USB to Serial interface will appear as a virtual COM port on the PC, on GNU/Linux systems it can be accessed as /dev/ttyUSB0. You need to install Python interpreter and the python-serial module on the PC for this to work. These Python programs should be terminated before using microHOPE again to upload programs.

## 2.6.1 Send/receive Characters

The program **echo.c** waits for data from the PC, vis the USB to serial converter, increment it by one and sends it back. The received data is also displayed on the local LCD display.

```
#include "mh-lcd.c"
#include "mh-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);
     }
}
```

After uploading this program, open a terminal window, change to the directory named microhope and run the python program **echo.py** listed below, using the commands:[4]

```
$ cd microhope
$ python echo.py
```

```
import serial
fd = serial.Serial('/dev/ttyUSB0', 38400, stopbits=1, \
    timeout = 1.0)
while 1:
    c = raw_input('Enter the character to send: ')
    fd.write(c)
    print 'Receiced ', fd.read()
```

[4]If you are using microHOPE with MCP2200 IC, edit echo.py to replace ttyUSB0 with ttyACM0. The same thing applies to programs like cro.py, pymicro.py etc.

We can rewrite echo.c without using the library functions. The program **echo-v2.c** listed below id functionally identical to echo.c

```
#include "mh-lcd.c"
int main(void)
{
  uint8_t data;
  lcd_init();
  // set UART to 38400 baudrate, 8 databit, 1 stopbit, No parity
  UCSRB = (1 << RXEN) | (1 << TXEN);
  UBRRH = 0;
  UBRRL = 12;            // At 8MHz (12 =>38400)
  UCSRC = (1<<URSEL) | (1<<UCSZ1) | (1<< UCSZ0);
  for(;;)
     {
     while ( !(UCSRA & (1<<RXC)) );  //wait on the receiver
     data = UDR;                     // read a byte
     lcd_put_char(data);
     while ( !(UCSRA & (1<<UDRE)) ); // wait on Data Reg Empty flag
     UDR = data;
  }
}
```

### 2.6.2 Sending ADC data

The program **remote-adc.c**, listed below, on receiving a channel number, in 0 to 7 range, reads the corresponding channel and send the data to the PC using the UART, via the USB to Serial converter. Use the Python program **remote-adc.py** on the PC side.

```
#include "mh-lcd.c"
#include "mh-uart.c"
#include "mh-adc.c"
int main(void)
{
uint8_t chan, low, hi;
uint16_t adcval;
lcd_init();
uart_init(38400);
adc_enable();
for(;;)
    {
    data = uart_recv_byte();
    if (chan <= 7)
      {
      adcval = read_adc(chan);
  lcd_clear();
      lcd_put_int(low);
      low = adcval & 255;
      hi = adcval >> 8;
```

```
            uart_send_byte(low);       // send LOW byte
            uart_send_byte(hi);        // send HI byte
            }
        }
    }
```

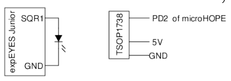
### 2.6.3 A simple Oscilloscope

The program **cro.c** can digitize the input at PA0 in a block mode and send the data to the PC. The number of samples and the time interval between consecutive digitizations are send from the PC. The program **cro.py** can communicate to cro.c and graphically display the received data. While running cro.py , the PCRST jumper should be open.

### 2.6.4 Infrared Receiver

The program ir-recv.c can receive data using the TSOP1738 IR receiver. The output of the chip is connected to bit 2 of PORTD. The received byte is displayed on the LCD display. The receiver is for the non-standard data senb by expEYES junior using an infra-red LED.

To test ir-recv.c, make the connections as shown below:



Inside the command window of expEYES junior GUI, type **irsend1(nn)** , where nn could be any number from 0 to 255, and press Enter. The received number will be displayed on the LCD display of microHOPE.

### 2.6.5 Controlling the uC from Python

This section demonstrates a simple method to read/write the Input/Output ports and other registers of the micro-controller, from Python. A program called **pymicro.c** is running on the micro-controller. It listens over the serial port for two commands, READB or WRITEB. The first one should be followed by the address of the register to be read. The WRITE command is followed by the register address and the data to be written.

On the PC side, **pymicro.py** handles the communication to the micro-controller. It defines a class named atm32, that contains the communication routines. The example program listed below demonstrates a blinking LED code in Python

```
import time
from pymicro import *
u=atm32()
while 1:

    u.outb(PORTB, 1)
    time.sleep(0.5)
    u.outb(PORTB, 0)
    time.sleep(0.5)
```

To run this program, compile and upload pymicro.c, remove the PCRST jumper and then run blink.py. It is very easy to implement many programs, for example a stepper motor controller, in Python using this method.

# Chapter 3

# Serial Loading of Program memory

We are able to load programs to microHOPE through Rx/Tx pins of the UART only because of the pre-loaded boot loader program. How to program a micro-controller using the In-System Programming (ISP) feature is explained below. This is implemented using the Serial Peripheral Interface (SPI) interface of the micro-controller. The SPI consists of three wires, Serial ClocK (SCK), Master-In–Slave-Out (MISO) and Master-Out–Slave-In (MOSI). All types of memory on the micro-controller can be accessed using the SCK, MISO and MOSI pins, while holding the RESET pin LOW. These pins, along with ground, are available on the 5 pin header J7 on the microHOPE board. For details, refer to the circuit schematic shown in figure 3.3.

The SPI pins can be accessed by connecting to the Parallel port of the PC, using a cable as shown is figure 3.1. We can also use In-System Programmers that can be connected to the USB port of the PC. We are using an ISP called the USBASP, that is open hardware.

The microHOPE IDE can upload programs using the USBASP programmer

### 3.0.6 Software

The program **avrdude** can be used for programming the micro-controller by using Parallal port or the USBASP programmer. The commands to use, as root user, are:

```
# avrdude -c dapa -patmega32 -U flash:w:blink.hex
# avrdude -c usbasp -patmega32 -U flash:w:blink.hex
```

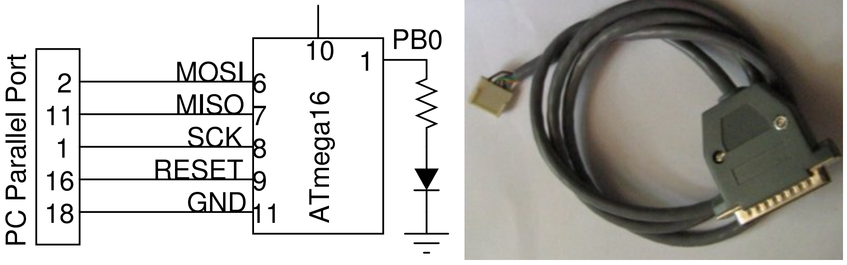


Figure 3.1: PC Parallel port cable for Serial loading of program memory.

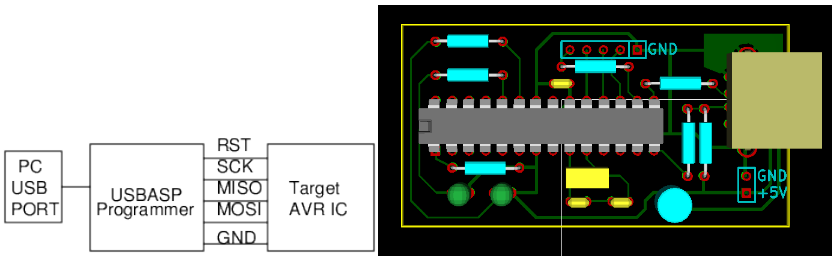


Figure 3.2: USBASP programmer.(a) block diagram (b) layout

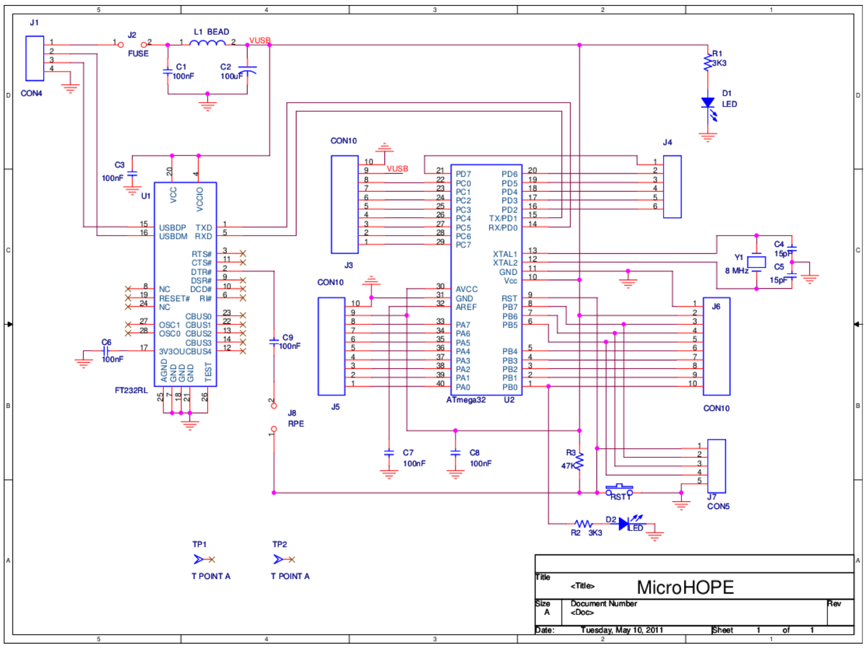


Figure 3.3: Circuit schematic of microHOPE

The -c option is used for specifying the programmer to be used. The commands should be given from a terminal, after changing to the directory 'microhope', where all the data files are kept.

## Setting up the Boot Loader

We can use one of these methods for uploading the bootloader program of microHOPE. The commands for uploading the hex file and setting the fuses, using the parallel port cable, are:

```
avrdude -c dapa -patmega32 -U flash:w:ATmegaBOOT_168_atmega32.hex
avrdude -c dapa -patmega32 -U lfuse:w:0xef:m -U hfuse:w:0xda:m
```

If you are using USBASP, use:

```
avrdude -c usbasp -patmega32 -U flash:w:ATmegaBOOT_168_atmega32.hex
avrdude -c usbasp -patmega32 -U lfuse:w:0xef:m -U hfuse:w:0xda:m
```

For more details refer to the microhope section of the website expeyes.in

The circuit schematic of microHOPE is shown in figure 3.3