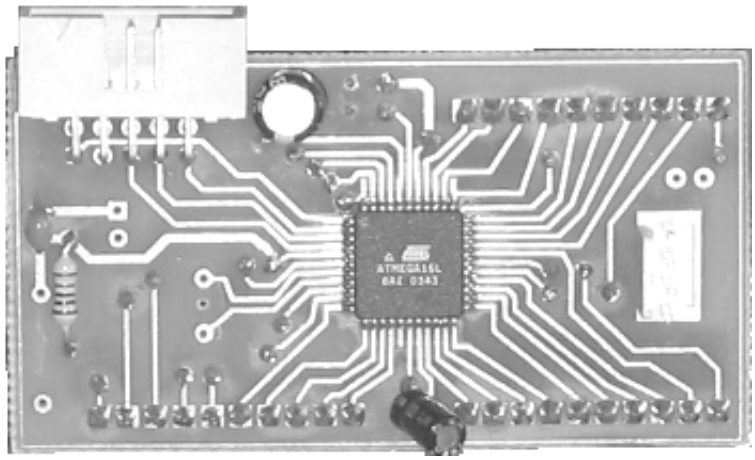
School of Computing,
Communication and Electronics

Robotic Intelligence Lab

THE UOP ATMEGA HANDBOOK

for the
UOP - ATmega 16 Developer Board



Joerg Wolf (jwolf@swrtec.de)

Version 1.0 , October 2004

Table of Content

1. Introduction

This booklet provides an introduction to programming an Atmel-Mega microcontroller. It is based on the ATMega16-Developer Board. The information provided is a good starting point for degree students looking for a micro-controller for their projects. The board can be used in combination with an "application board" or "main board" which students develop completely to implement their project. The main advantage of the developer board is that it runs without external circuitry. It can be connected to a bread-board, for initial development of the application. Bread-board developing safes a considerable amount of time, since it reduces the number of generations the application board is going through. Bread-board design would be impossible without the developer board, since it is a surface mount chip.

If you are already confident with basic ANSI C programming (statements such as if, else, do, while, for, main and others) and you have a basic understanding of microcontrollers, this document should be easy to understand.
This document can be downloaded from http://www.swrtec.de/swrtec/clinux/avrgcc/

There is related documentation around the university which targets the old AT90S8515 microcontroller [3,4] and the Robot Football Competition player (ATmega8). Please contact Alan Simpson for a copy.

1.1 What do I need ?

- avr-gcc version **3.x.x** (also called WinAVR)
- PonyProg 2.05a or 2.06c
- intropack.zip containing the PCB layouts and example source code
- University ATMega16-Developer-Board
- Atmel Parallel Port downloading cable
- University Serial Level converter board

1.2 How do I know I have got the right software ?

Open a command prompt and enter: **avr-gcc –version**

The output should look something like this:

```
C:\data\atmel\soo-mi>avr-gcc --version
avr-gcc (GCC) 3.3.1
Copyright (C) 2003 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
C:\data\atmel\soo-mi>
```

Make sure that the version number is >= 3.0.0. e.g. if the version is 2.95 then you have got the wrong software. The environment in the University Labs is currently mixed, check your computer, or better install it on your own laptop.

Start up PonyProg and check the version by the menu "?" , then "about". The version number should be 2.05a or higher.

1.3 Design tips

1.3.1 Interfacing a micro controller

The micro must have enough IO pins. Some IO pins, such as PWM outputs are usually fixed. Therefore it is best to do an account of :
- number of pins for digital Input / Output
- number of analog Input pins
- number of pins for PWM (e.g. for Servo or motor control)
- number of Hardware Interrupt lines (e.g. for shaft encoders)
The pinout list in the Appendix is

If the micro has not enough pins for the job the usual options are:
- Use Multiplex IO Lines with an 74138 for addressing and 74245s for data-bus/device separation
- Use Programmable Logic Devices (PLDs) such as the Altera MAX 7000 series
- Use off-the-shelf circuits, such as motor/servo controllers
- Use an I2C-Bus IO-expansion chip such as the PCF8575

1.3.2 Power Rail Design:

Here are some general tips for designing a PCB.
Noise is the micro's biggest enemy ! Therefore:

a) Capacitors
Use plenty of capacitors across the PCB when designing the main-board. Connect these capacitors as close as possible to the chip's VCC and GND. At least one 10nF or 100nF capacitor per chip and a couple of 100uF on the power rail of your board. If you are using any fast switching high current devices, such as Motor control circuits, check with the datasheet, how to place capacitors.
But here is my rule of thumb for motor control circuits:
Per chip, have one capacitor with a capacity of 1000uF per Ampere. So if the motor uses 10 Ampere you should use at least a 10000uF capacitor.

b) Make the power tracks on your PCB as thick as possible.

c) Design a ground plane.

2. Get going

2.1 Hardware

Three circuits need to be soldered:
- the developer board
- the download cable
- the serial cable

2.1.1 Tips for the construction of the UOP ATmega-16 Developer Board:

- solder the Power Connector (J6,J7) on the component side (TOP)
- solder the SIL-Pins of J2,J4,J5 on the solder-side (BOTTOM)
- connect Pin J2-9 to VCC , even if you don't use the AD-Converter (see [1] page 341)
- connect Pin J4-10 to GND, even if you don't use the AD-Converter
- a ATmega16L needs 2.7 – 5.5 Volt to run and can do up to 8 MHz
- a ATmega16 needs 4.5 – 5.5 Volt to run and can do up to 16 MHz
- check if the crystal (X1) needs external capacitors

2.2 Software Installation

2.2.1 Windows : WinAVR 3.X

Download WinAVR from www.avrfreaks.net

Just double click the setup….
Next..next..next….
WinAVR suggests to register the folders C:\WinAVR\bin; C:\WinAVR\utils\bin in the $PATH variable.

A program group called WinAVR will be created. For editing source code, use “Programmers Notepad” which installs automatically with WinAVR.

Open up Programmers Notepad and click on Tools/Options in the menu.
Change “Line Endings” to “Unix (LF)”.

2.2.2 Linux: gcc 3.x

This is a guide to install avr-gcc 3.3 or 3.4 under linux
There is no warranty that this will work for every system.
But I successfully compiled avr-gcc under linux with the following packages:

avr-libc-1.0.3.tar.bz2
binutils-2.14.tar.bz2
gcc-core-3.4-20040310.tar.bz2
uisp-20040311.tar.bz2

The packages should be compiled in the following order:
1. binutils for the system
2. binutils for AVR-target
3. gcc-core
4. libc
5. uisp

The first step is to install the new binutils for the linux gcc:

```
tar xvfj binutils-2.14.tar.bz2
cd binutils-2.14
./configure
make
make install
```

The next step is to install binutils again, but this time for AVR.
Everything will be installed to the target folder /usr/local/atmel

```
make distclean
./configure --prefix=/usr/local/atmel --target=avr --enable-languages=c --
host=avr --disable-nls
make
make install
cd ..
```

Now the AVR-GCC can be compiled. The new avr-gcc is included in the original package from gnu.org. No patches are required.

```
tar xvfj gcc-core-3.4-20040310.tar.bz2
cd gcc-3.4-20040310
./configure --prefix=/usr/local/atmel --target=avr --enable-languages=c --
host=avr --disable-nls
make
make install
```

Now the AVR files should be renamed in order to avoid confusion if the system gcc and the avr-gcc

are both in the path.

```
cd /usr/local/atmel/bin
mv gcc avr-gcc
mv objcopy avr-objcopy
mv ar avr-ar
mv as avr-as
mv cpp avr-cpp
mv ranlib avr-ranlib
mv ld avr-ld
mv objdump avr-objdump
mv nm avr-nm
```

...

Before compiling avr-libc the following environment variables must be exported:

```
export CC=/usr/local/atmel/bin/avr-gcc
export AR=/usr/local/atmel/bin/avr-ar
export AS=/usr/local/atmel/bin/avr-as
export RANLIB=/usr/local/atmel/bin/avr-ranlib
export PREFIX=/usr/local/atmel

cd "download-folder"
tar xvfj avr-libc-1.0.3.tar.bz2
cd avr-libc-1.0.3
./doconf
./domake
./domake install
cd ..
```

The last step is to install the ISP-Programmer software.
I opened a new console, so there are none of the libc export variables around.

```
tar xvfj uisp-20040311.tar.bz2
cd uisp-20040311
./configure --prefix=/usr/local/atmel/
make
make install
```

Known problems:

Fedora Core 2:
uisp linker error : "undefined reference to `rpl_malloc' "
go to the src/config.h file and delete the line "#define malloc rpl_malloc" then continue with "make" and "make install" as usual

Congratulations, you have successfully compiled avr-gcc.
I must admit it is a bit more work than WinAVR.

2.2.3 PonyProg configuration

Download and install PonyProg.
(Available at www.lancos.com)

After installing PonyProg:
1. Open PonyProg and go to the menu point Setup / Interface Setup
2. Click on Parallel
3. Select “AVR ISP I/O
4. Click on LPT1
5. Click OK
6. Open menu point Setup / Calibration
It should say , calibration OK

Every time you load up PonyProg, make sure that the right device (type of micro.) is set.
Set the device in the menu by: e.g. Device / AVR micro / ATmega16

2.3 Setting up the fuses

Start up PonyProg and go to the menu point Command / Security and Configuration bits and then press read.

When the micro comes from the factory the settings in PonyProg look like this :

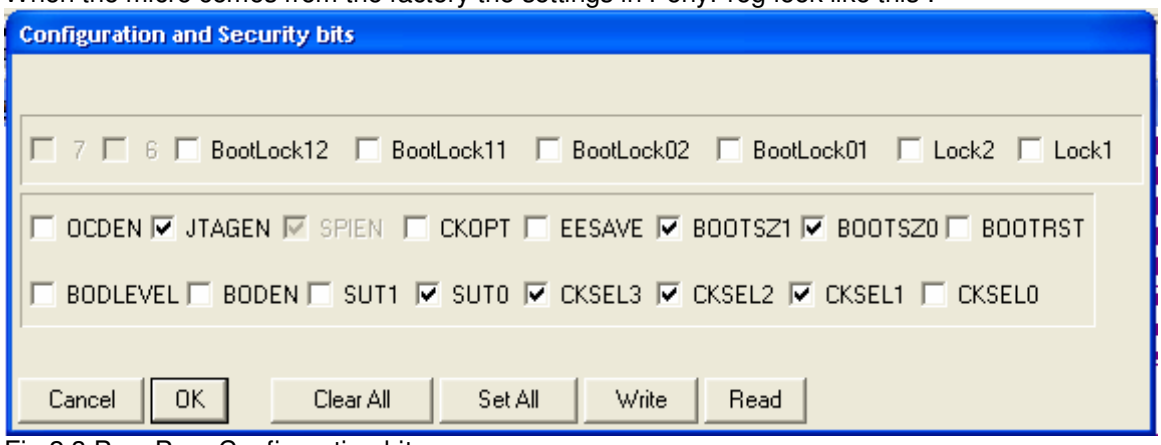


Fig 2.3 PonyProg Configuration bits

This setup means that the micro is running with its 1 MHz internal clock and it doesn't care even if you have soldered in a external resonator already. Be aware that the JTAGEN flag is set by factory default, which means that PORTC doesn't work for normal IO !
There are many possible settings depending on the required configuration. Setting up the fuse bits is a dangerous business, since you could lose control over the micro if they are set wrong.

Power up instructions:
1. Connect the download cable
2. Check that the AVCC Pin is connected to VCC
3. Power up the micro

2.3.1 1 MHz Internal Clock Setup

So the minimum to get going is to disable the JTAG interface by doing the following steps:
1. Power up (see above)
2. Press Read
3. remove the tick in front of the JTAGEN
4. press Write

2.3.2 16MHz External Clock Setup

1. Power up (see above)
2. Solder in a 16 MHz crystal with the capacitors
3. Press Read
4. remove all ticks ! (Except SPIEN of course)
5. tick CKOPT
6. press Write

3. Writing, Compiling and Downloading

3.1 Creating a project

1. Create a folder for your project on the hard disk
2. Open Programmers notepad, click on File / New / “C / C++”
3. Write a program.

e.g.

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

void pause();

void pause (){
        int i;
        for(i=0;i<20000;i++){i++;i--;   }// pause a few milliseconds
        for(i=0;i<20000;i++){i++;i--;   }// and a few more
        for(i=0;i<20000;i++){i++;i--;   }// and more
}

int main(void){
        DDRA = 0x40;                    // Set up Pin 6 on port A for Output


        for(;;){
                pause();
                PORTA = 0x00;           // LED On
                pause();
                PORTA = 0x40;           // LED off
        }
        return 0;
}
```

4. Safe source code in project folder (call it “blink.c”)
5. Create a file called “Makefile” which contains the code below. Be aware that a “Makefile is sensitive to the tab key. The tab key must be used after a label. It is best to copy a Makefile from an existing project; see intropack.zip.

```
PRG             = blink
OBJ             = blink.o
MCU_TARGET      = atmega16
OPTIMIZE        =
DEFS            =
LIBS            =

# You should not have to change anything below here.

CC              = avr-gcc
override CFLAGS         = -g -Wall $(OPTIMIZE) -mmcu=$(MCU_TARGET) $(DEFS)
override LDFLAGS        = -Wl,-Map,$(PRG).map
OBJCOPY         = avr-objcopy
OBJDUMP         = avr-objdump

all: $(PRG).elf lst text eeprom

$(PRG).elf: $(OBJ)
        $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^ $(LIBS)

clean:
        rm -rf *.o $(PRG).elf *.bak
        rm -rf *.lst *.map $(EXTRA_CLEAN_FILES)
        rm -rf *.hex

lst:  $(PRG).lst

%.lst: %.elf
        $(OBJDUMP) -h -S $< > $@

# Rules for building the .text rom images

text: hex

hex:  $(PRG).hex

%.hex: %.elf
        $(OBJCOPY) -j .text -j .data -O ihex $< $@

# Rules for building the .eeprom rom images

eeprom: ehex

ehex:  $(PRG)_eeprom.hex
```

```
%_eeprom.hex: %.elf
	$(OBJCOPY) -j .eeprom --change-section-lma .eeprom=0 -O ihex $< $@
```

6. Click on “Tools” / “Make all” in the Programmers Notepad menu.
The “Make all” command initiates the compilation.

3.2 Compiling the program

see step 6. in section 3.1 : Click on “Tools” / “Make all” in Programmers Notepad or
1. open a command-shell (DOS-Environment)
2. change the directory to your project directory containing Makefile and source
3. type in make

A compilation process may look something like this:

```
avr-gcc -g -Wall  -mmcu=atmega16    -c -o blink.o blink.c
avr-gcc -g -Wall  -mmcu=atmega16  -Wl,-Map,blink.map -o blink.elf blink.o
avr-objdump -h -S blink.elf > blink.lst
avr-objcopy -j .text -j .data -O ihex blink.elf blink.hex
avr-objcopy -j .eeprom --change-section-lma .eeprom=0 -O ihex blink.elf
link_eeprom.hex
```

The first line is the compilation of the source code *blink.o* into an object file. Object files are linked to and *.elf* file in the second line. A *.elf* file is something like an “.exe” file in Microsoft. Avr-objdump then disassembles the file again and creates an *.lst* file. The *.lst* file is a readable assembler listing. It can be used to calculate instruction cycles or for debugging purposes. avr-objcopy now translates the binary .elf file into an Intel-Hex file *(.hex*). This file-format is understood by PonyProg. A hex-file for the flash and the eeprom is created. The linker (second line) also creates a *.map* file. It contains information about how objects are mapped into memory.

3.3 Downloading a program with PonyProg

1. Open PonyProg
2. Check the device setting
3. Select File / “Open Device File” from the menu
4. Change “Files of type” from “*.e2p” to “*.hex
5. Load the file (e.g. blink.hex) from the project folder
6. Connect the download cable
7. Power up the micro-controller
8. Press the “Write device” button in the toolbar

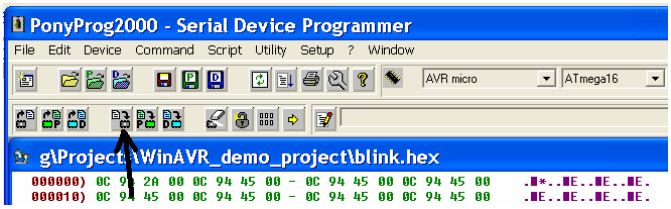


Fig. 3.3 Write Device Button

The application should reply with “Write successful” after a few seconds.
The program starts working as soon as this message appears.

Note: Every time the source code has changed the process is: recompile, press the reload button  in PonyProg, and finally the “Write device button”

4 Introduction to Atmel-Mega programming

4.1 Setting up a Port

The Mega16 has 4 ports labelled A to D. A port is a group of 8 physical pins on the micro. Each of these pins can be used as Input or Output pins. In order to tell the micro which pin is an input and which is an output the **DDRx** register must be programmed. (where x indicates port A,B,C or D).
A logical one indicates that the corresponding bit on the port is an output.

Example:
```
DDRA = 0x08;
```

In this example 0x08 is a hexadecimal number that represents 0000 1000b. Therefore bit 3 will be an output, the rest of the port will be used as input.

4.2 Using a Port for Output

Using a port like this
```
PORTA = 0xF0;
```
will actually make pins high and low. It is a good idea to represent the bits as a hexadecimal value. This is indicated through the 0x in front of the hexadecimal number F0. In this case the lower 4 significant bits are low (0) and the 4 higher ones are set to high (F).
The Atmel Mega16 has 4 ports called PORTA PORTB PORTC PORTD. Some bits are already used for special functions such as serial port, programming port others are general purpose.

Often it is necessary to change only one bit at a time. You might have two simultaneously running programs which control different pins on the same port. Here is an example how to switch on bit 3 from PORTA.

```
PORTA = PORTA | 0x08;
```

This line of code takes the existing content of port A (the current status of highs and lows on the pins) and combines it with a logical OR and then stores the result in PORTA. The line can also be written as

```
PORTA |= 0x08;          // pin 3 high
```

In Binary the bits look like:

PORTA before	0 1 0 0 0 0 0 1	← arbitrary value
0x08	0 0 0 0 1 0 0 0	
PORTA new (**OR**)	0 1 0 0 1 0 0 1	

And here an example how to switch bit 3 off again:

```
PORTA &= ~0x08;          // pin low
```

In Binary the bits look like:

(0x08	0 0 0 0 1 0 0 0)	
PORTA before	0 0 1 0 1 0 0 1	
~0x08	1 1 1 1 0 1 1 1	← complement of 0x08
PORTA **AND** ~0x08	0 0 1 0 0 0 0 1	

Note: for switching off, the logical AND and the complement of the mask is used. Whereas switching on uses the logical OR with the mask.

4.3 Using a Port for Input

self-explaining Example:

```
if( PINA & 0x01){
                // The pin is high
}else{
                // The pin is low
}
```

Note: The corresponding register for reading from PORTA is called PINA.
The statement `if( PORTA & 0x01 ){..}` **doesn't work !**

4.4 Interrupt programming

Global variables that are used in more than one interrupt or in the interrupt plus in the main program must be declared `volatile`. If you are unsure, just declare all global (shared) variables `volatile`.
If interrupts are used, the main program must contain the statement `sei();` which will enable interrupts in general. If an interrupt occurs a specially named function will be called.
e.g.

```
SIGNAL(SIG_INTERRUPT0){…}      // hardware interrupt (triggered by physical input pin INT0)
SIGNAL(SIG_INTERRUPT1){…}
SIGNAL(SIG_OVERFLOW1){…}       // software interrupt (timer routine)
```

4.4.1 Using Timers

PWM Example using 2 Channels:

Declarations:

```
#define RMOTOR_PWM            OCR1AL
#define LMOTOR_PWM            OCR1BL
```

In the main program:

```
int main(){
        DDRD   |= 0x30;               // OC1A and OC1B as Output : RPWM and LPWM respectively
        TCCR1A  = (1 << WGM10) ;      // Fast PWM 8Bit Mode
        TCCR1A |= (1 << COM1A1)| (1 << COM1A0)|(1 << COM1B1)|(1 << COM1B0); //inverted mode PWM
        TCCR1B  = (1 << WGM12) | (1 << CS11);               // Fast PWM 8Bit Mode
                                                            // prescaling div 8 gives 7.8 KHz
        LMOTOR_PWM = 128;      // 50%
        RMOTOR_PWM = 64;       // 25% inverted -> 75%

        for(;;){               // Loop forever , to stop micro

        }
        return 0;
}
```

This code will create a rectangular waveform on pin OC1A and OC1B. OC1A will have a duty cycle of 75% on (note inverted PWM). OC1B will have a duty cycle of 50%.

4.4.2 Hardware Interrupts

Typical quadrature shaft encoder example for two motors.

Global variables:

```
volatile int LEncoderCount = 0,REncoderCount = 0;
```

Functions:

```
SIGNAL(SIG_INTERRUPT1) //      Left Motor encoder
{
        if (inp(PINC) & '\x02')
                LEncoderCount ++;
        else
                LEncoderCount --;
}
```

```
SIGNAL(SIG_INTERRUPT0) //      Right Motor encoder
{
        if (inp(PINC) & '\x01')
                REncoderCount ++;
        else
                REncoderCount --;
}
```

at the beginning of the main program:

```
        DDRC = 0x0C;    // Quadrature Read Pins on PC0 and PC1
        DDRD&= ~0x0C;   // Quadrature Int. Pins on PD2 and PD3
        GICR = 0xC0;    // Enable external int0, int 1
        MCUCR = 0x0F;   // int0,int 1 - triggered on rising edge
        LEncoderCount=0;
        REncoderCount=0;
        sei();
```

4.5 Setting up the Makefile

It is best to use an existing Makefile and just modify the target cpu (MCU_TARGET) and the PRG,OBJ variables to match the source code filename-prefix. So if your source code file is called xyz.c then the variables should be:
PRG = xyz
OBJ = xyz.o
The example Makefile must be in the same folder as the source code.

To edit the Makefile you can use:

- Windows Notepad
- Programmers Notepad
- MS Visual C++
- Windows WordPad

But you must **NOT** use:

- MS-DOS Edit

Editors like MS-DOS Edit do not use real tabs (ASCII 9), they substitute tabs with spaces (ASCII 32). This will confuse “make”.

Some more tips:

- Add `-lm` to the linker flags in the make file if you use <math.h>. The linker flags are usually called `LFLAGS` of `LDFLAGS`.
- Leave out any optimisation flags such as –O2 if your program contains a primitive pause loop without a timer.
- See section 3.1 (Creating a project) for some more information about Makefiles.

References

[1] “The Atmel Mega 16 Datasheet”,doc2466.pdf , Atmel Corporation,San Jose CA ,USA ,2003
[2] Eric Weddington, Marek Michalkiewicz, Reiner Patommel, Theodore A. Roth, Joerg Wunsch , “The avr-libc Manual”, http://savannah.nongnu.org/projects/avr-libc/ ,2004
[3] Alan Simpson, “PRACTICAL ROBOTICS, A First Course in MicroController Programming and Usage”, University of Plymouth, Nov. 2002
[4] Alan Simpson, “Programming the Miabot”, University of Plymouth, Sept. 2004

Appendix

Layout

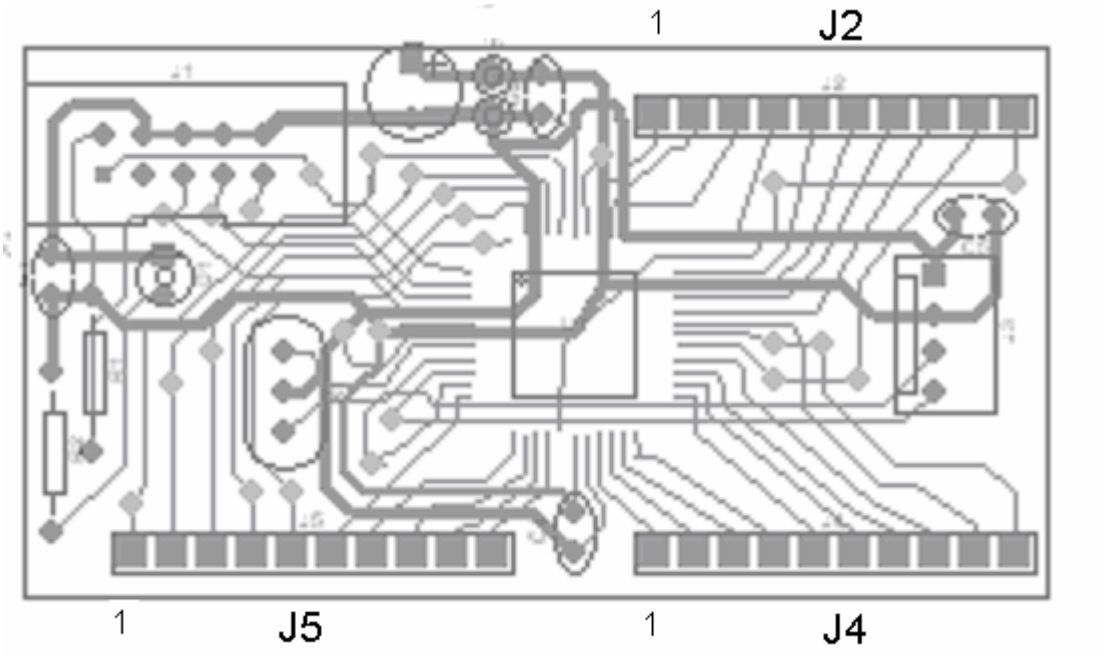


Fig. A1 Layout, Top view , both layers visible

Pinout Table for MEGA16 Developer Board V1.1

Pin on Micro	Name	Connector Pin	on Mainboard used for:
37	PA0 (ADC0)	J2 - 1	
36	PA1 (ADC1)	J2 - 2	
35	PA2 (ADC2)	J2 - 3	
34	PA3 (ADC3)	J2 - 4	
33	PA4 (ADC4)	J2 - 5	
32	PA5 (ADC5)	J2 - 6	
31	PA6 (ADC6)	J2 - 7	
30	PA7 (ADC7)	J2 - 8	
40	PB0 (XCK/T0)	J5 - 1	
41	PB1 (T1)	J5 - 2	
42	PB2 (AIN0/INT2)	J5 - 3	
43	PB3 (AIN1/OC0)	J5 - 4	
44	PB4 (SS)	J5 - 5	
1	PB5 (MOSI)	-	
2	PB6 (MISO)	-	
3	PB7 (SCK)	-	
19	PC0 (SCL)	J4 - 1	
20	PC1 (SDA)	J4 - 2	
21	PC2 (TCK)	J4 - 3	
22	PC3 (TMS)	J4 - 4	
23	PC4 (TDO)	J4 - 5	
24	PC5 (TDI)	J4 - 6	
25	PC6 (TOSC1)	J4 - 7	
26	PC7 (TOSC2)	J4 - 8	
9	PD0 (RXD)	-	
10	PD1 (TXD)	-	
11	PD2 (INT0)	J5 - 6	
12	PD3 (INT1)	J5 - 7	
13	PD4 (OC1B)	J5 - 8	
14	PD5 (OC1A)	J5 - 9	
15	PD6 (ICP1)	J5 - 10	
16	PD7 (OC2)	J2 - 10	
4	RESET	-	
5	VCC	-	
17	VCC	-	
38	VCC	-	
8	XTAL1	-	
7	XTAL2	-	
28	AGND *	J4 - 10	
29	AREF	J4 - 9	
27	AVCC **	J2 - 9	
6	GND	-	
18	GND	-	
39	GND	-	

*** connect to ground, even if the AD Converter is not used**
**** connect to VCC, even if the AD Converter is not used !!**
(if this pins are not connected, the micro tends to sporadic behaviour, such as downloading not possible or program doesn't run. See datasheet [1] Page 272 and datasheet [1] Page 341, update 10)

Circuit Diagram

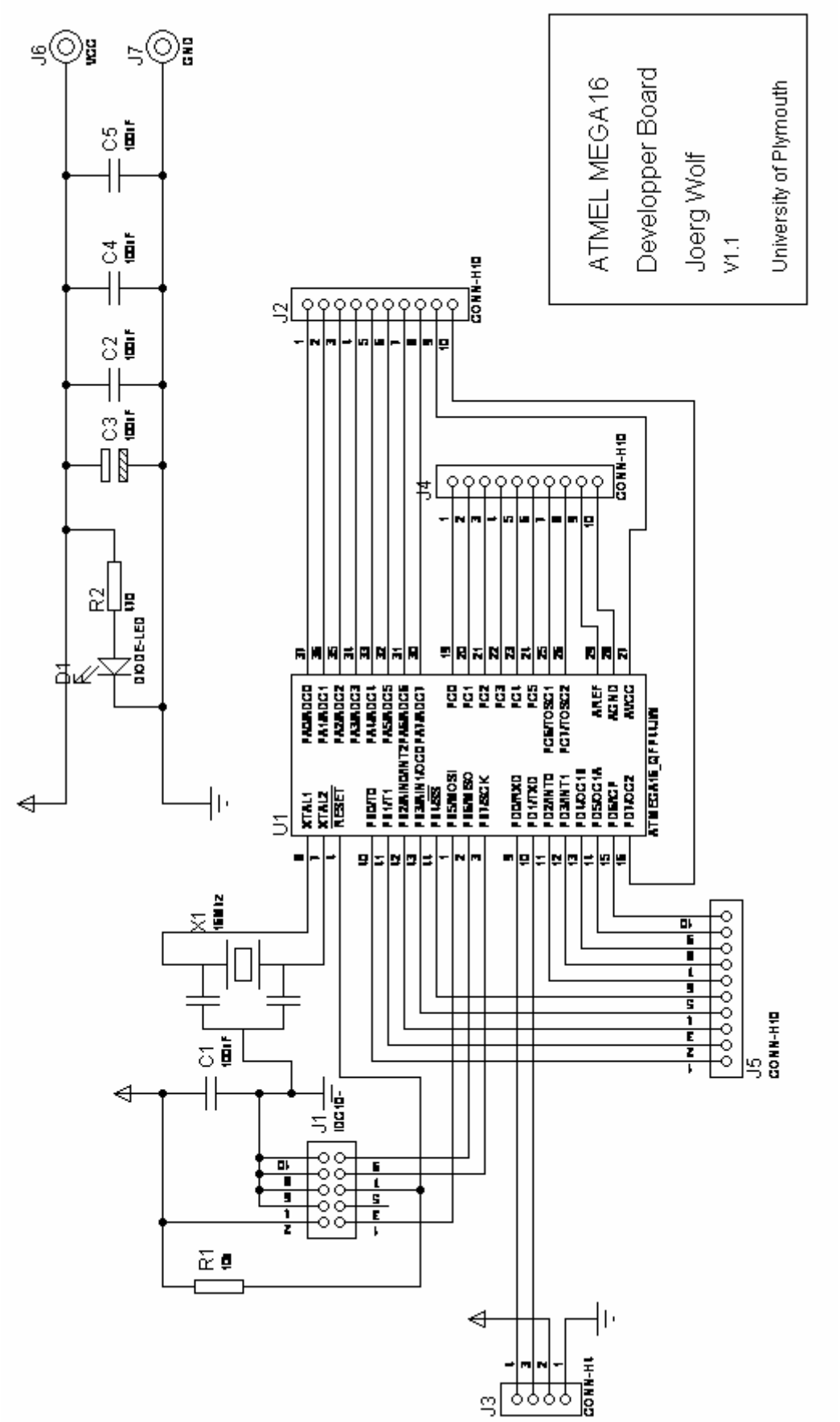


Fig A2 Circuit Diagram of the UoP module