Piconomic Design Atmel AVR Course

Sections:

- HOME
- PRODUCTS
- SERVICES
- CLIENTS
- CONTACT US

VISIT OUR STORE

Short introduction to AVR instruction set

The AVR core has an advanced RISC architecture with most of the instructions being executed in a single clock cycle. The AVR uses a Harvard architecture with separated access to program and data. A load/store assembler instruction set is implemented with 32 general purpose registers (R0 to R31). The instructions are divided into the following categories:

Arithmetic and Logic Instructions
e.g. add Rd,Rr (Add without Carry : Rd = Rd + Rr)

Branch Instructions
e.g. rjmp k (Relative Jump : PC = PC + k + 1)

Data Transfer Instructions
e.g. mov Rd,Rr (Copy register : Rd = Rr)

Bit and Bit-test Instructions
e.g. sbi P,b (Set bit in I/O register : I/O(P,b) = 1)

The quickest way to learn the assembler instruction set is to refer to the Compiled Help file of Atmel AVR Studio:
Help > AVR Tools User Guide > AVR Assembler > Parts > ATmega128/1280/1281 and AT90CAN128 Instruction Set

The **Status Register(SREG)** contains flags that convey information about the most recently executed arithmetic instruction. Bit 7 (I) is different, as it is the flag that enables/disables interrupts globally:

	7	6	5	4	3	2	1	0
SREG:	I	T	H	S	V	N	Z	C

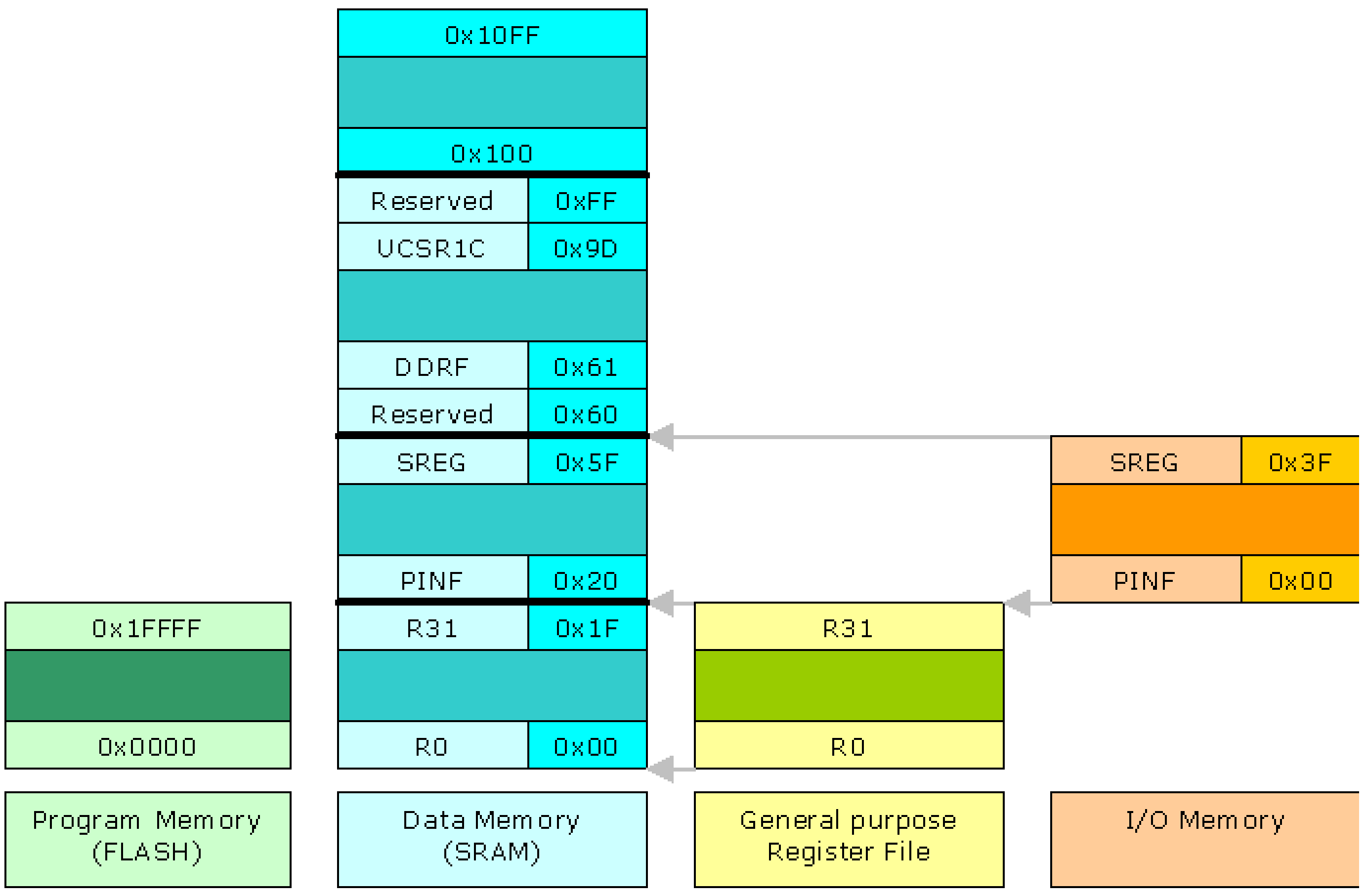
Bit 7 – I: **Global Interrupt Enable**
Bit 6 – T: Bit Copy Storage
Bit 5 – H: Half Carry Flag
Bit 4 – S: Sign Bit
Bit 3 – V: Two’s Complement Overflow Flag
Bit 2 – N: Negative Flag
Bit 1 – Z: Zero Flag
Bit 0 – C: Carry Flag

Introduction to AVR peripheral access

Tip: this section appears daunting, but will become essential knowledge on your way to master the AVR. Skim through it now, and return to read it in depth after working through "Tutorial 01 - Port IO".

All of the AVR peripherals are manipulated by writing to and reading from the Peripheral Control Registers. Refer to "Register Summary" of the ATmega128 datasheet (p.365)

Here is a condensed visual representation of the ATmega128 memory map to highlight the Harvard architecture and access to the Peripheral Control Registers:



The memory map will make more sense after working through the tutorials, but it is displayed here to point out a specific mental stumbling block on the GCC / AVR-LIBC learning curve ("SFRs - Special Function Registers").

The data memory load/store instructions provide a different method to access the the general purpose registers (R0 to R31) and the I/O memory (0x00 to 0x3F). Thus the following assembler instructions are equivalent, but not optimal:

mov R16,R17 <--> lds R16,0x0017 (R17 can be accessed at address 0x0017 in data space)

and

in R19,0x00 <--> lds R19, 0x0020 ("PINF" is mapped to 0x00 in I/O space andaddress 0x0020 in data space).

To access the other Peripheral Control Registers that do not fit into I/O space (which have optimal bit manipulation instructions), data space load/store instructions must be used.

Luckily, the compiler takes care of these details in the background.

Here are two examples:

1. I/O space (DDRB - 0x17)

C: DDRB |= (1<<6); // Set I/O pin PB6 to output

Assembler: sbi 0x17,6;

2. Data space (DDRF - 0x61)

C: DDRF |= (1<<5); // Set I/O pin PF5 to output

Assembler:
```
lds R24,0x61;
ori R24,32;
sts ;0x61,R24;
```

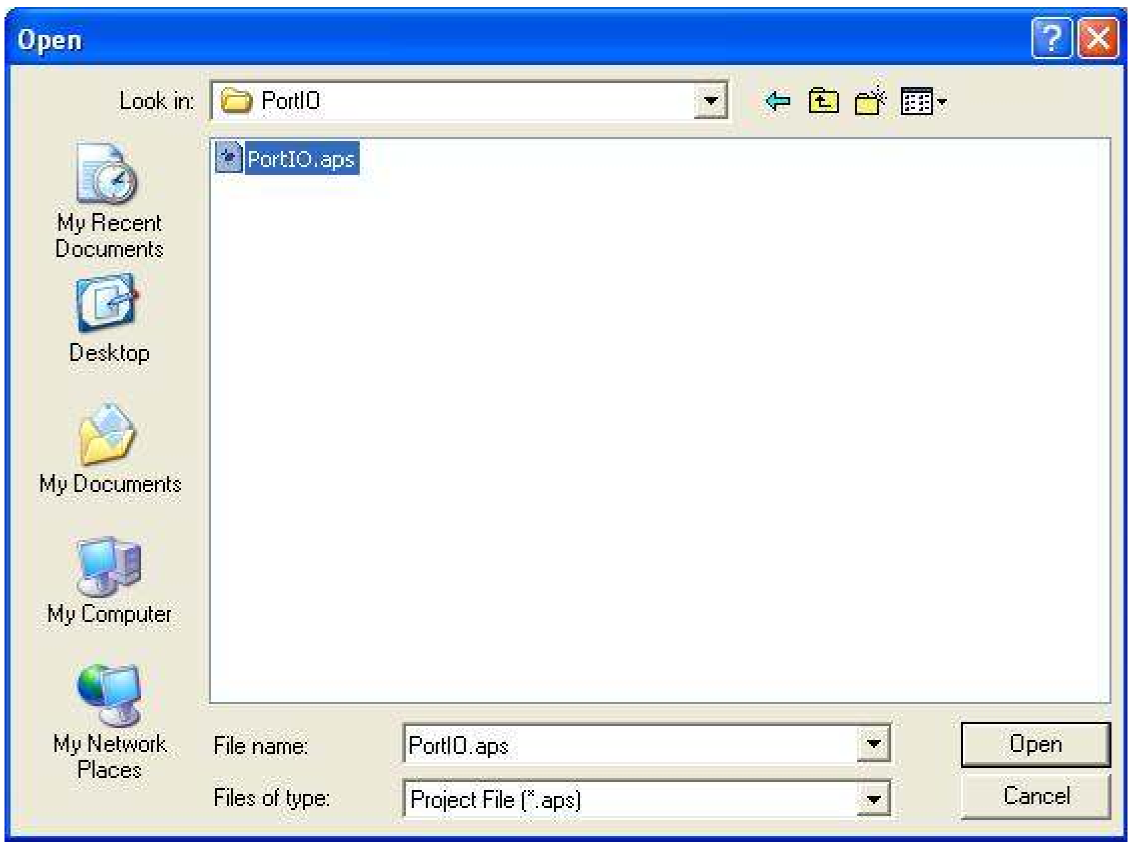
Access to Data memory mapped peripherals is not as efficient as I/O memory mapped peripherals.

How to open an existing project in AVR Studio

AVR Studio offers a complete integrated development environment: editor, build system, simulator, debugger, programmer,...

All of the tutorials, bootloader and firmware framework are provided with a pre-configured AVR Studio project. External Makefiles are referenced, in stead of AVR Studio's build system, to support non-Windows users.

An existing project can be opened by navigating to the AVR Studio menu "Project>Open Project" and selecting the "*.aps" file, e.g. "..\Tutorials\01 Port IO\PortIO.aps"

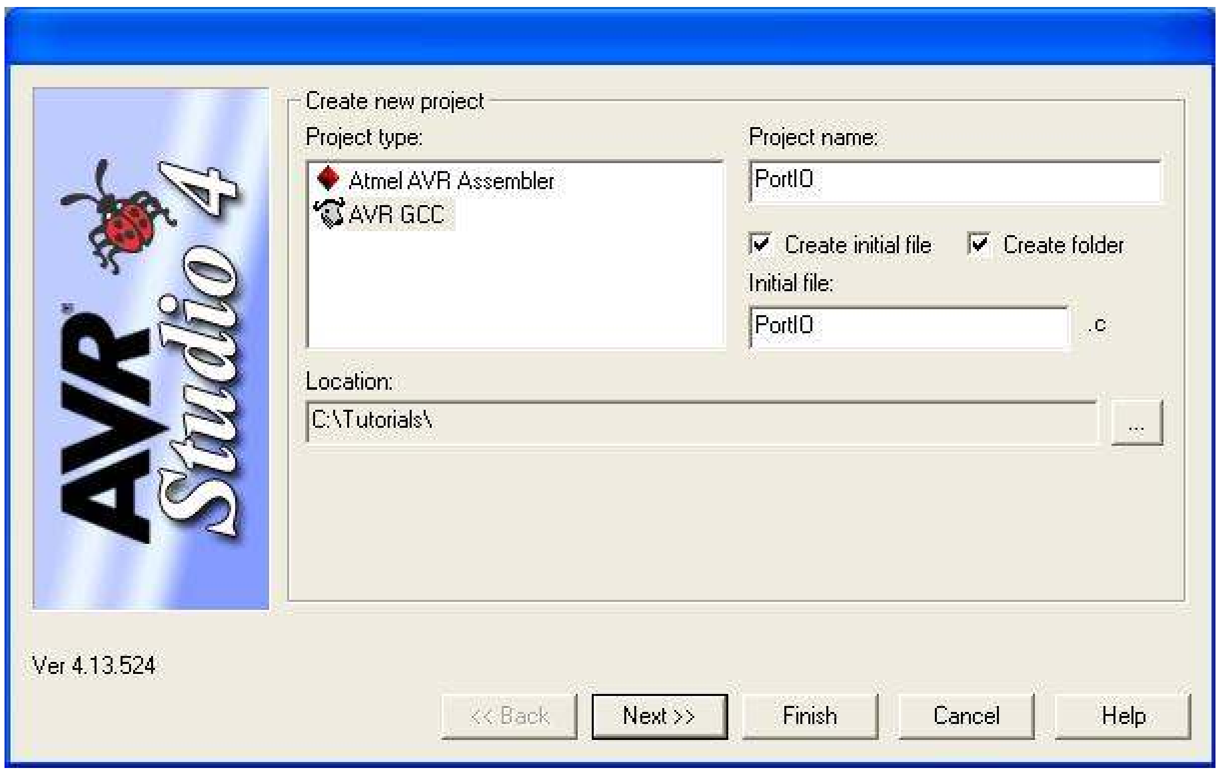


File name: PortIO.aps
Files of type: Project File (*.aps)

How to create a new project in AVR Studio using the AVR GCC plugin

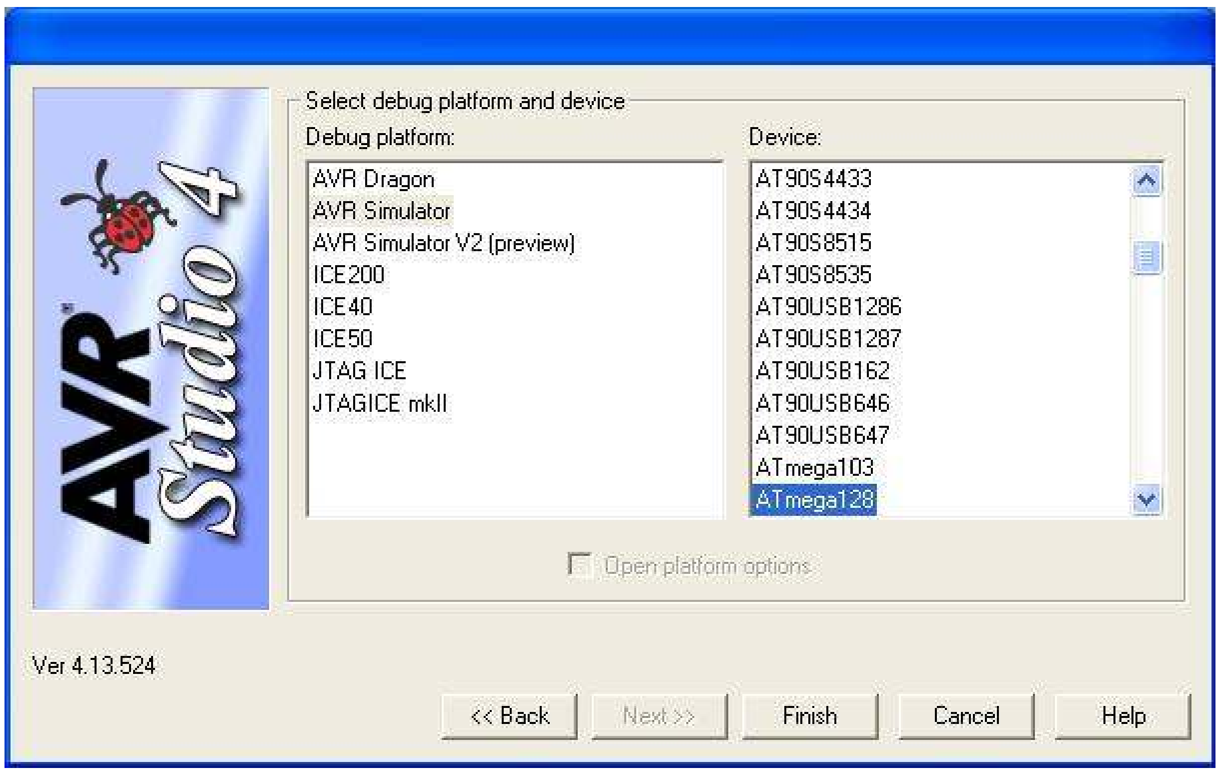
Here are the steps to create a new AVR GCC project in AVR Studio:

1. Navigate to the AVR Studio menu "Project>New Project" and select "AVR GCC" as project type.
2. Type a project name, e.g. "PortIO" and create an inital C file, e.g. "PortIO.c"
3. Select a location, e.g. "C:\Tutorials" and create a folder.



Project name: PortIO
Create initial file [✓] Create folder [✓]
Initial file: PortIO .c
Location: C:\Tutorials\

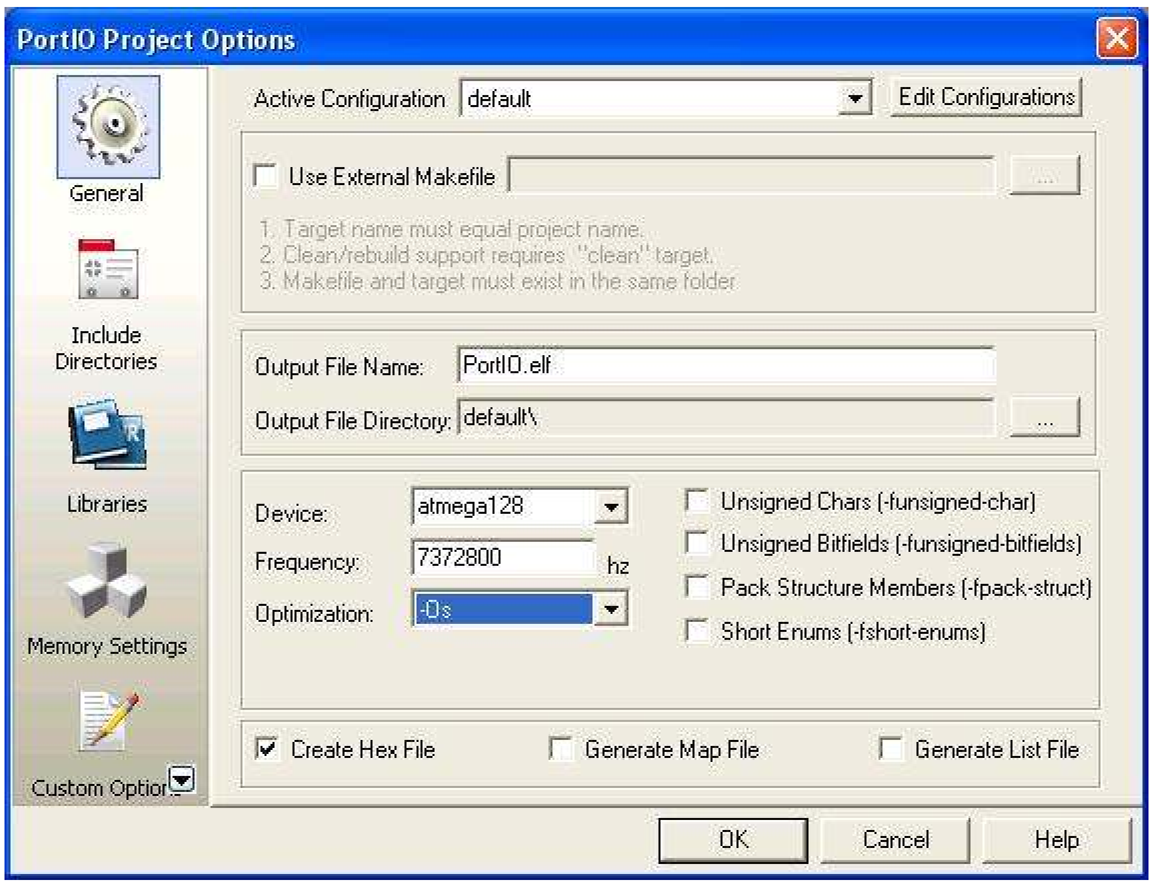
4. Select "Next>" and choose "AVR Simulator" and "ATmega128" as the device.



You can change this choice at a later stage by navigating to "Debug>Select Platform and Device..."

5. Select Finish. Your new C file will now be created and open for editing.

6. The build options, which changes an AVR Studio generated Makefile, is selected by navigating to "Project > Configuration Options". Set the frequency to 7372800 (7.3728 MHz) and optimization to -Os (optimized for size) and select OK.


PortIO Project Options

Active Configuration: default

☐ Use External Makefile

1. Target name must equal project name.
2. Clean/rebuild support requires "clean" target.
3. Makefile and target must exist in the same folder

Output File Name: PortIO.elf

Output File Directory: default\

Device: atmega128

Frequency: 7372800 hz

Optimization: -Os

☐ Unsigned Chars (-funsigned-char)

☐ Unsigned Bitfields (-funsigned-bitfields)

☐ Pack Structure Members (-fpack-struct)

☐ Short Enums (-fshort-enums)

☑ Create Hex File ☐ Generate Map File ☐ Generate List File

7. The source code can be compiled by selecting "Build > Build"

How to simulate a project in AVR Studio

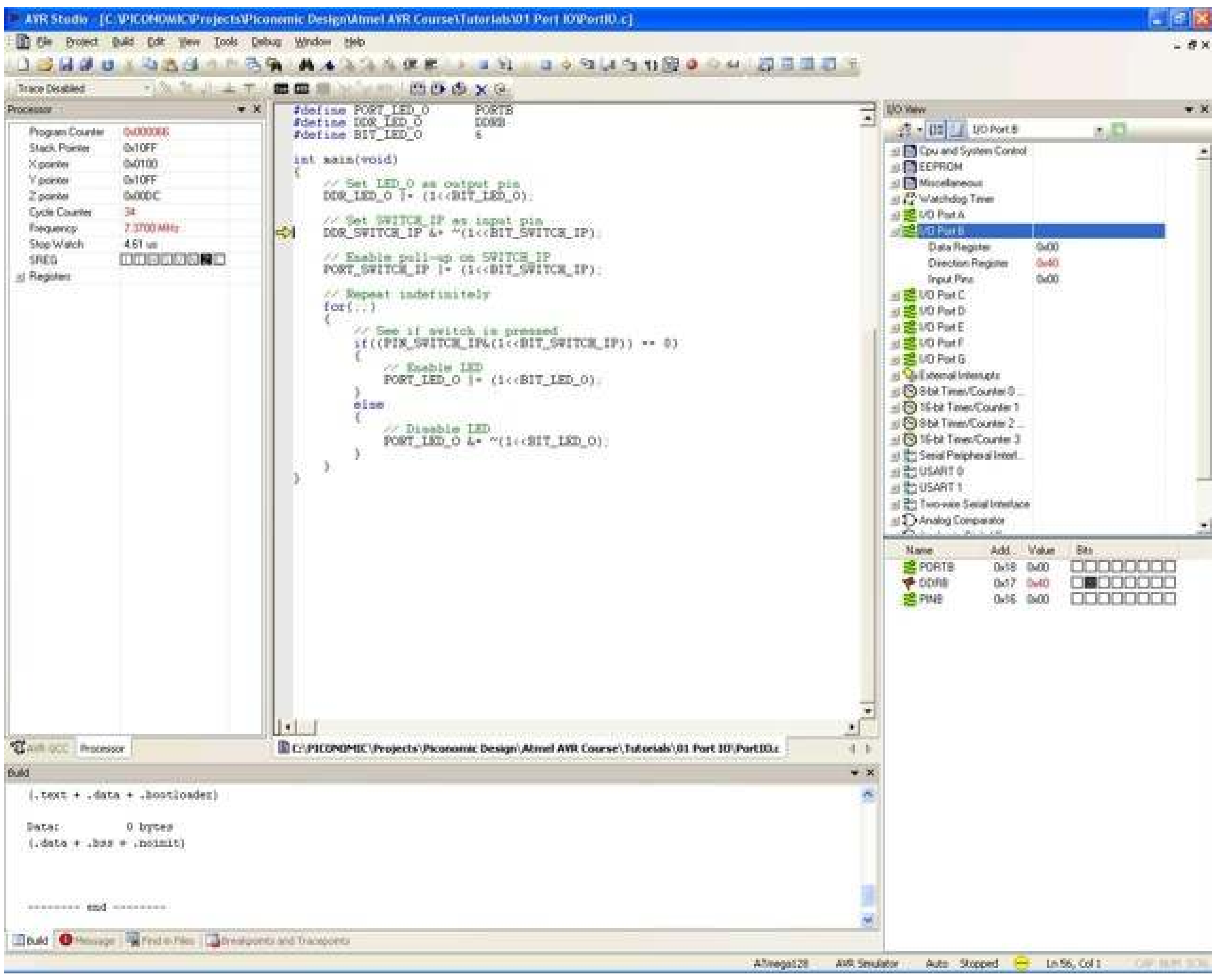
AVR Studio is an invaluable development tool that should be used vigorously to simulate the code and verify it's correctness, before downloading it to the target.

This section assumes that the code has been built successfully.

First enable cycle accurate timing information with "Debug > AVR Simulator Options", set the clock frequency to "7.37 MHz" and "OK". This setting is saved with the project and needs only to be done once.

Select "Debug > Start Debugging". You can now single-step, set breakpoints,...

Select and expand the I/O View in the right-hand pane to view the status of the processor and the peripherals.


AVR Studio - [C:\PICONOMIC\Projects\Piconomic Design\Atmel AVR Course\Tutorials\01 Port IO\PortIO.c]

Program Counter	0x000066
Stack Pointer	0x10FF
X pointer	0x0100
Y pointer	0x10FF
Z pointer	0x00DC
Cycle Counter	34
Frequency	7.3700 MHz
Stop Watch	4.61 us

```
#define PORT_LED_O       PORTB
#define DDR_LED_O        DDRB
#define BIT_LED_O        6

int main(void)
{
    // Set LED_O as output pin
    DDR_LED_O |= (1<<BIT_LED_O);

    // Set SWITCH_IP as input pin
    DDR_SWITCH_IP &= ~(1<<BIT_SWITCH_IP);

    // Enable pull-up on SWITCH_IP
    PORT_SWITCH_IP |= (1<<BIT_SWITCH_IP);

    // Repeat indefinitely
    for(;;)
    {
        // See if switch is pressed
        if((PIN_SWITCH_IP&(1<<BIT_SWITCH_IP)) == 0)
        {
            // Enable LED
            PORT_LED_O |= (1<<BIT_LED_O);
        }
        else
        {
            // Disable LED
            PORT_LED_O &= ~(1<<BIT_LED_O);
        }
    }
}
```

Data Register	0x00
Direction Register	0x40
Input Pins	0x00

Name	Add.	Value
PORTB	0x18	0x00
DDRB	0x17	0x40
PINB	0x16	0x00

```
(.text + .data + .bootloader)

Data:          0 bytes
(.data + .bss + .noinit)


-------- end --------
```

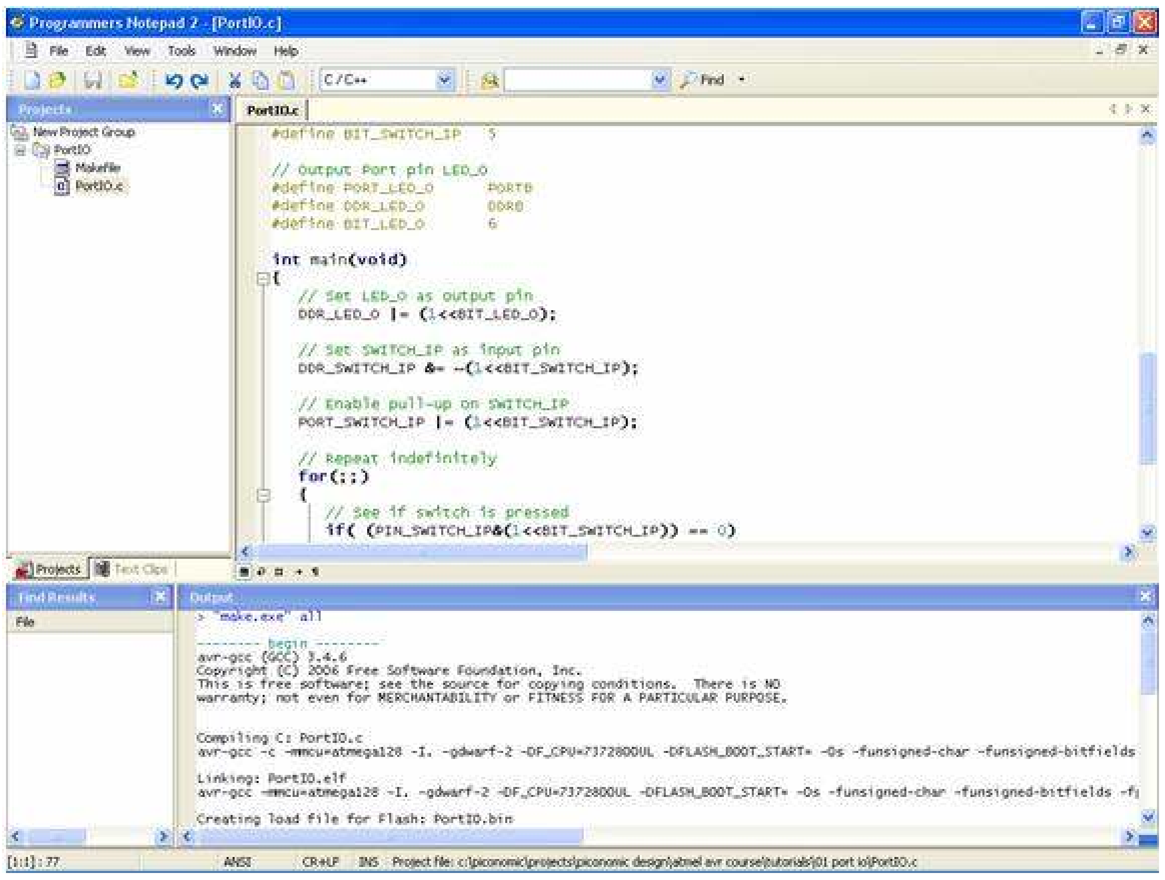
How to edit and build a project using Programmers Notepad 2

Programmer's Notepad may be used if you wish to edit and build the source files without using AVR Studio. It is bundled with the WinAVR distribution.

1. Open the project in Programmer's Notepad: "File > Open Project(s)..." and select a project, e.g. "...\Tutorials\01 Port IO\PortIO.pnproj", and "Open"

2. Delete all output files (optional) : "Tools > [WinAVR] Make Clean"

3. Build the project: "Tools > [WinAVR] Make All"



```
#define BIT_SWITCH_IP    5

// Output Port pin LED_O
#define PORT_LED_O       PORTB
#define DDR_LED_O        DDRB
#define BIT_LED_O        6

int main(void)
{
    // Set LED_O as output pin
    DDR_LED_O |= (1<<BIT_LED_O);

    // Set SWITCH_IP as input pin
    DDR_SWITCH_IP &= ~(1<<BIT_SWITCH_IP);

    // Enable pull-up on SWITCH_IP
    PORT_SWITCH_IP |= (1<<BIT_SWITCH_IP);

    // Repeat indefinitely
    for(;;)
    {
        // See if switch is pressed
        if( (PIN_SWITCH_IP&(1<<BIT_SWITCH_IP)) == 0)
```

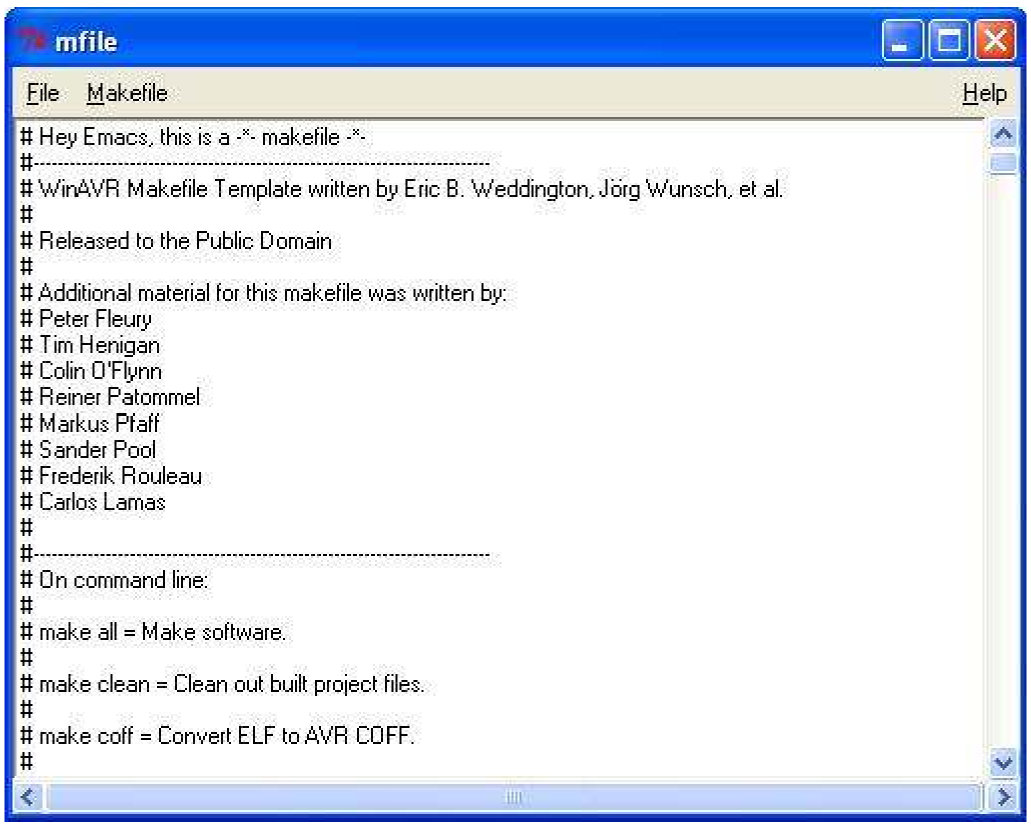
```
> "make.exe" all

-------- begin --------
avr-gcc (GCC) 3.4.6
Copyright (C) 2006 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.

Compiling C: PortIO.c
avr-gcc -c -mmcu=atmega128 -I. -gdwarf-2 -DF_CPU=7372800UL -DFLASH_BOOT_START= -Os -funsigned-char -funsigned-bitfields

Linking: PortIO.elf
avr-gcc -mmcu=atmega128 -I. -gdwarf-2 -DF_CPU=7372800UL -DFLASH_BOOT_START= -Os -funsigned-char -funsigned-bitfields -f

Creating load file for Flash: PortIO.bin
```

```
[1:1] : 77    ANSI    CR+LF    INS    Project file: c:\piconomic\projects\piconomic design\atmel avr course\tutorials\01 port io\PortIO.c
```

How to create a new Makefile if not using AVR Studio

A Makefile is used to automate the process of compiling and linking the source code of a project.

A TCL/Tk script called "Mfile" is bundled with WinAVR that automates the process of creating a new Makefile. The other option is to copy and modify an existing Makefile.



```
# Hey Emacs, this is a -*- makefile -*-
#----------------------------------------------------------------------------
# WinAVR Makefile Template written by Eric B. Weddington, Jörg Wunsch, et al.
#
# Released to the Public Domain
#
# Additional material for this makefile was written by:
# Peter Fleury
# Tim Henigan
# Colin O'Flynn
# Reiner Patommel
# Markus Pfaff
# Sander Pool
# Frederik Rouleau
# Carlos Lamas
#
#----------------------------------------------------------------------------
# On command line:
#
# make all = Make software.
#
# make clean = Clean out built project files.
#
# make coff = Convert ELF to AVR COFF.
#
```

How to use the XMODEM-CRC bootloader

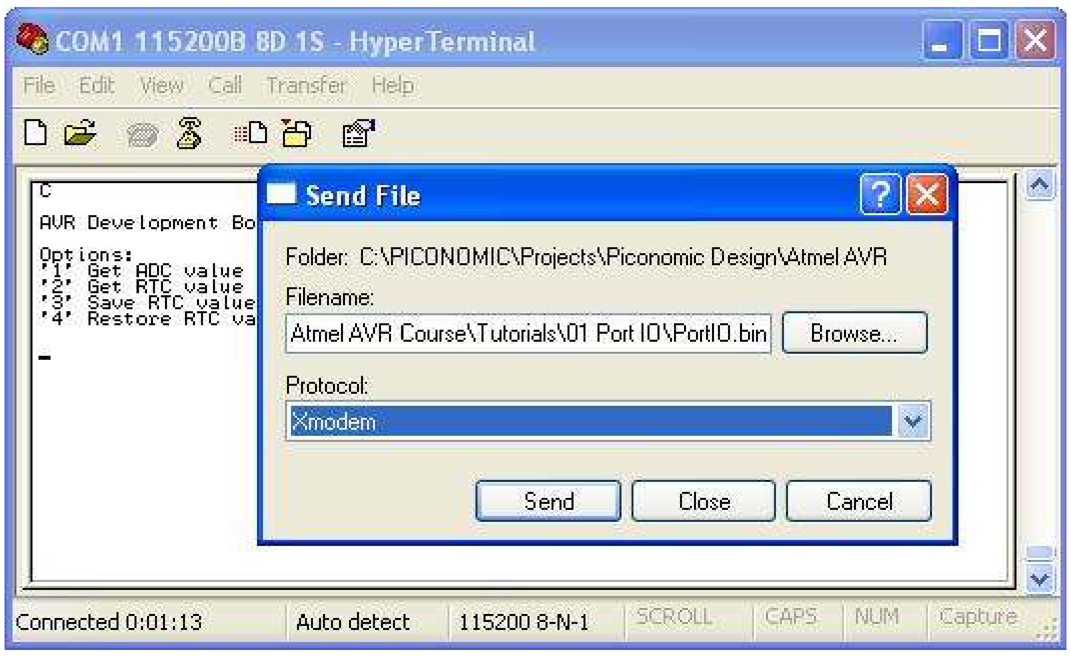
1. Compile and link the firmware application and generate a **binary** programming file (not Intel HEX!)

Use "..\Tutorials\01 PortIO\PortIO.bin" distributed with the set of tutorials as a first test.

2. Create a new HyperTerminal serial port session, configured to 115200 BAUD, 8 Data Bits, No Parity, 1 Stop Bit, No Flow Control.

At this stage you should verify that the communication between HyperTerminal and the board is OK, by powering and/or resetting the board and verifying that HyperTerminal displays at least one received "C" character.

3. Select "Transfer > Send File ... ". Select "Protocol > Xmodem". Select your application file. The screen should look similar to image:


COM1 115200B 8D 1S - HyperTerminal
File Edit View Call Transfer Help
Send File
Browse...
Send
Close
Cancel
SCROLL
CAPS
NUM
Capture

```
C
AVR Development Bo
Options:
'1' Get ADC value
'2' Get RTC value
'3' Save RTC value
'4' Restore RTC va
```

Folder: C:\PICONOMIC\Projects\Piconomic Design\Atmel AVR

Filename: Atmel AVR Course\Tutorials\01 Port IO\PortIO.bin

Protocol: Xmodem

Connected 0:01:13 Auto detect 115200 8-N-1

3. Select "Send".

HyperTerminal will now wait for a 'C' character from the board to start the transfer.

5. Power and/or reset the board to start the transfer.

If the transfer is successful, the HyperTerminal dialog window will disappear. The bootloader automatically jumps to the start of the application at address 0x0000.

Final Note:
The AVR fuse bits have been set so that execution starts from the boot vector address. This means that the bootloader will always be executed first. The bootloader sends a 'C' character to start a transfer and waits for 1 second for a valid XMODEM-CRC data packet. If the transfer is not successful, it will jump to address 0x0000 and execute the application.

A DOS batch file "AVRISP_ProgBoot.bat" has been included to automate the programming of the board with the bootloader using an AVRISP. It also sets the fuse bits to the correct values.