German-Russian Institute of Advanced Technologies

TU-Ilmenau (Germany) and KNTRU-KAI (Kazan, Russia)

Guidelines for laboratory work №1 of the subject

«Computer systems»

«AVR Studio Simulator Introduction and Exercises»

Kazan 2014

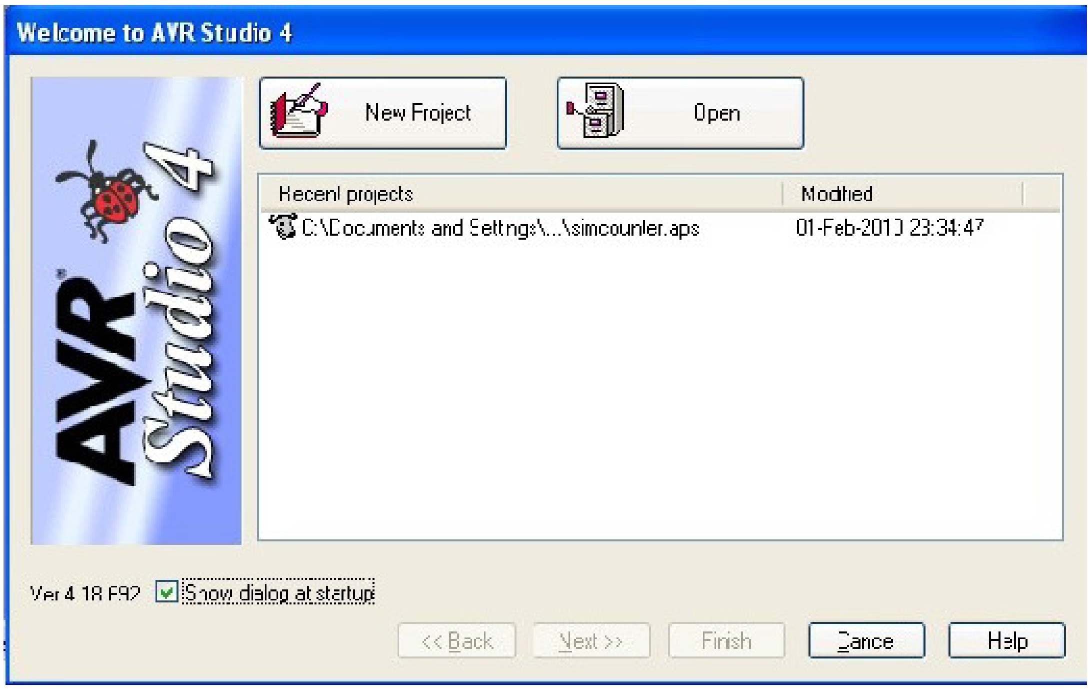
AVR Studio Simulator Introduction and Exercises

Introduction

The AVR Studio 4 is an Integrated Development Environment for debugging AVR software. The AVR Studio allows chip simulation and in-circuit emulation for the AVR family of microcontrollers. The user interface is specially designed to be easy to use and to give complete information overview. The AVR uses the same user interface for both simulation and emulation providing a fast learning curve.

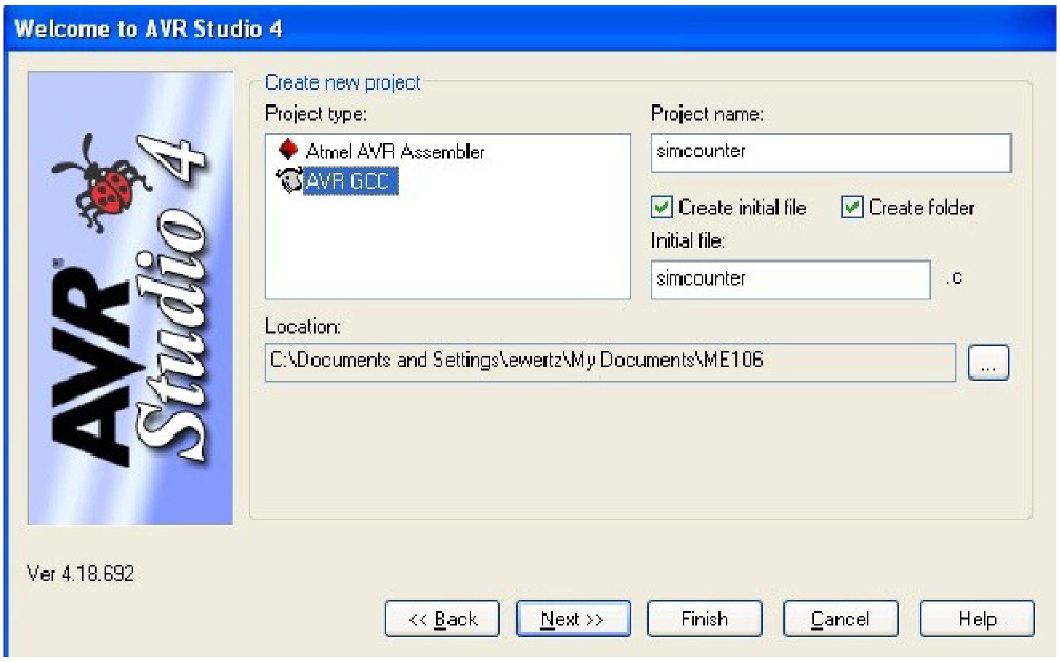
Figure 1. AVR Studio learning curve.

Creating a Project



Recent projects	Modified
C:\Documents and Settings\...\simcounter.aps	01-Feb-2010 23:34:47

Once AVR Studio is running you should be prompted for a project to work with, either by creating a new one, or by using an existing project. In this case, click **New Project**.



Welcome to AVR Studio 4

Create new project

Project type: Atmel AVR Assembler, AVR GCC (selected)

Project name: simcounter

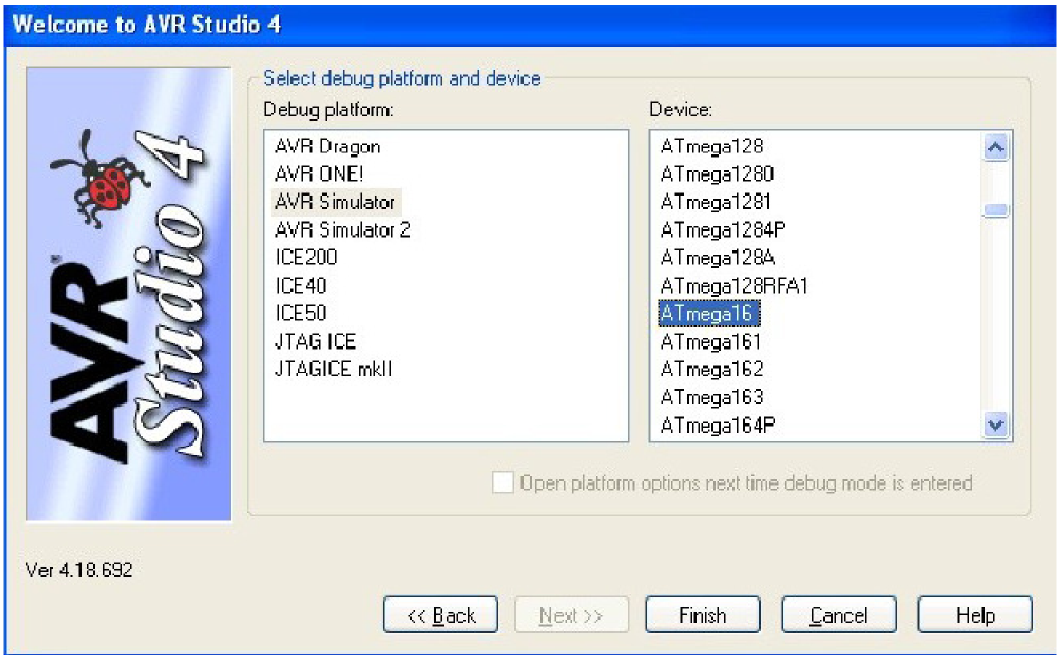
[✓] Create initial file [✓] Create folder

Initial file: simcounter .c

Location: C:\Documents and Settings\ewertz\My Documents\ME106

Ver 4.18.692

Select *Project type* **AVR GCC**, enter *Project name* **simcounter**, select both *Create initial file* and *Create folder* and select (and remember!) a directory in which to put the new **simcounter** project directory, then click **Next**.



Welcome to AVR Studio 4

Select debug platform and device

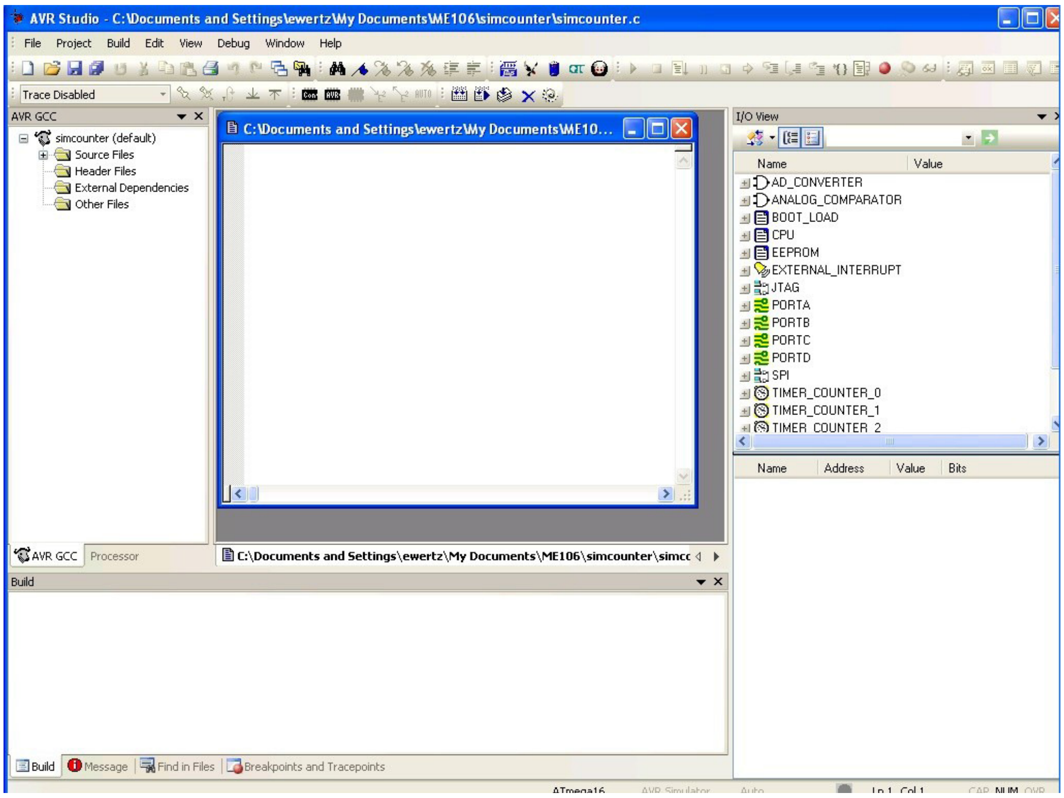
Debug platform: AVR Dragon, AVR ONE!, AVR Simulator (selected), AVR Simulator 2, ICE200, ICE40, ICE50, JTAG ICE, JTAGICE mkII

Device: ATmega128, ATmega1280, ATmega1281, ATmega1284P, ATmega128A, ATmega128RFA1, ATmega16 (selected), ATmega161, ATmega162, ATmega163, ATmega164P

[ ] Open platform options next time debug mode is entered

Ver 4.18.692

Select *Debug Platform* **AVR Simulator**, and *Device* **ATmega16**, then click **Finish**.



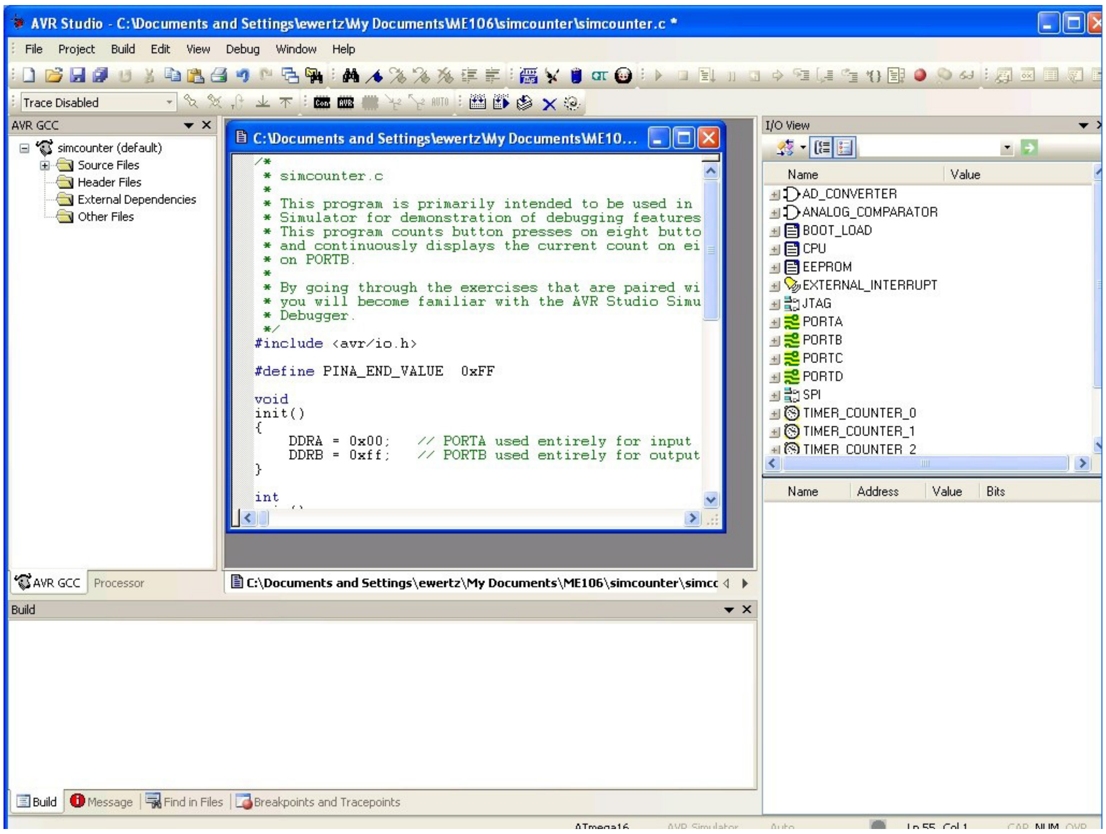
You should now see the default window layout in AVR Studio. Because you checked *Create initial file* when creating the project, AVR Studio has created a blank **simcounter.c** file and pre-opened this file for you.

If you click on the expand-group icon (the Plus icon) in front of *Source Files* in the lefthand window, you'll find that **simcounter.c** has been added to the **simcounter** project.

If you needed to include any additional files source files for your project, you would right-click on *Source Files* and choose the appropriate method to do this. You won't need to for this example, but you will for labs later in the semester.

At this point, cut-and-paste the contents of **simcounter.c** from the final section of this document into the blank window created for your project's **simcounter.c**. While you're at it, save the contents of this file by doing a File->Save (or typing **CTRL-S**, or pressing the floppy disk icon).

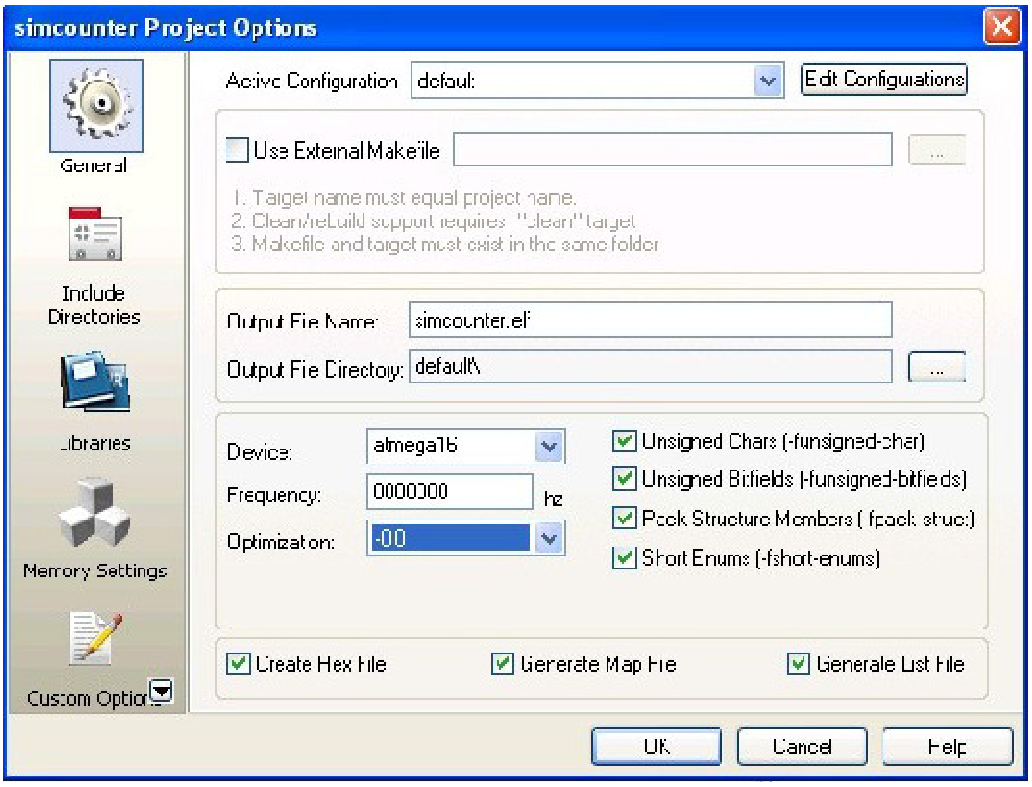
If you had a copy of simcounter.c already on your computer, you could also have used *Add Existing File* and pointed it to the file's location to add it to the project's source files.



```
/*
 * simcounter.c
 *
 * This program is primarily intended to be used in
 * Simulator for demonstration of debugging features
 * This program counts button presses on eight butto
 * and continuously displays the current count on ei
 * on PORTB.
 *
 * By going through the exercises that are paired wi
 * you will become familiar with the AVR Studio Simu
 * Debugger.
 */
#include <avr/io.h>

#define PINA_END_VALUE  0xFF

void
init()
{
    DDRA = 0x00;    // PORTA used entirely for input
    DDRB = 0xff;    // PORTB used entirely for output
}

int
```

Create a new file called **simcounter-PORTA.sti** by right-clicking on *Other Files* in the left-hand window and selecting *Create New File*. Cut-and-paste the contents of this file from the **simcounter-PORTA.sti** section at the bottom of this document into the window that was created for this file, and similarly save it to disk.

Configuring and Building Project



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: simcounter.elf

Output File Directory: default\

Device: atmega16

Frequency: 0000000 hz

Optimization: -O0

☑ Unsigned Chars (-funsigned-char)

☑ Unsigned Bitfields (-funsigned-bitfields)

☑ Pack Structure Members (-fpack-struct)

☑ Short Enums (-fshort-enums)

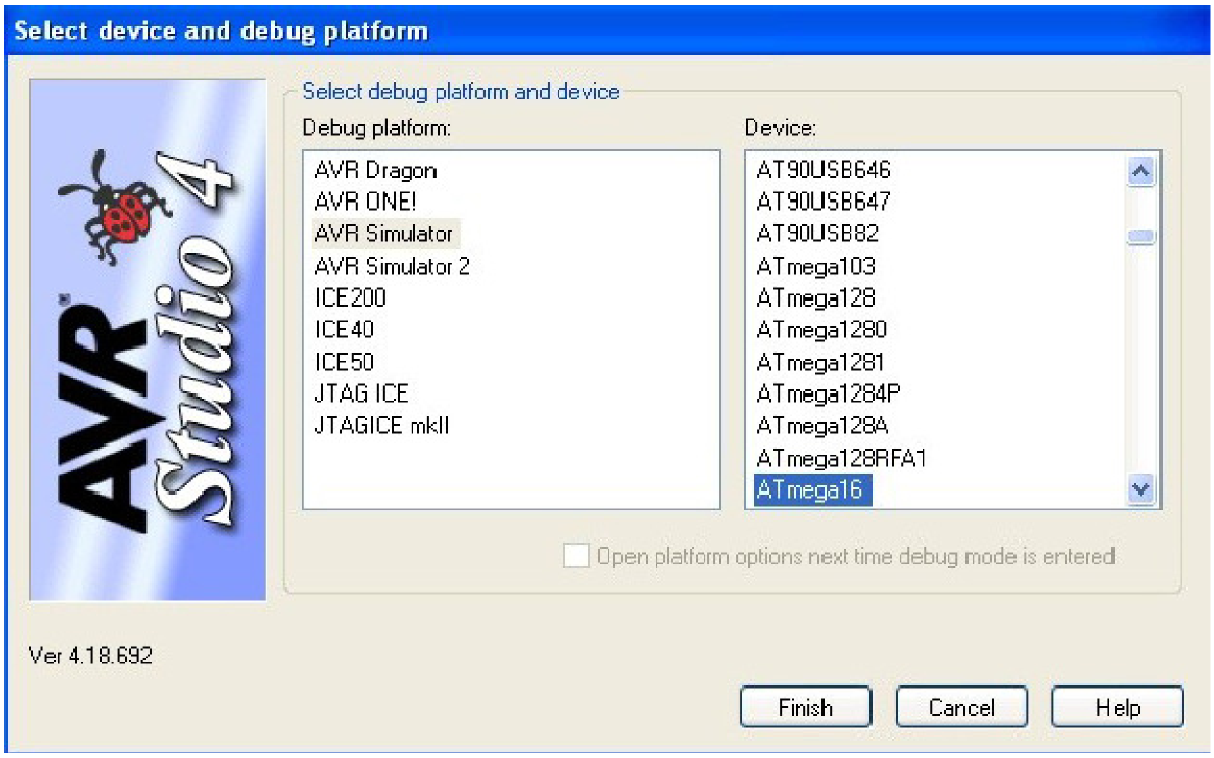
☑ Create Hex File ☑ Generate Map File ☑ Generate List File

In *Project->Configuration Options->General*, set *Frequency* to **8000000** and *Optimization* to **-O0** (note that this is capital-oh zero). It should be noted that this optimization setting is usually *not* the most popular for generating code to be downloaded to a real microcontroller, but it seems to be a safe choice for code to be simulated. In the future, once you're ready to start programming your microcontollers in the lab, you'll often be directed to change this setting **-O2** as you'll see in subsequent

weeks. Should you need to revert to simulating code again generating code for a real processor, don't forget to reset this option back to **-O0.** If you ever start seeing strange behavior when stepping execution through simulated programs, it's probable that you weren't using **-O0** for the code to be simulated. What usually happens is that while stepping through your program, the cursor doesn't accurately advance from line to line correctly.

After changing this optimization setting to **-O0**, do a *Build->Rebuild All*, and you should be all set.

Please note that doing a *Rebuild All* rather than just a *Build* is ***always*** a good idea after you have changed any project configuration options that have to do with compilation. In fact, it's always safe(st) to do a *Rebuild All* rather than just a *Build* – it's just that it may take somewhat longer.


Select device and debug platform
AVR Studio 4
Select debug platform and device
Finish
Cancel
Help

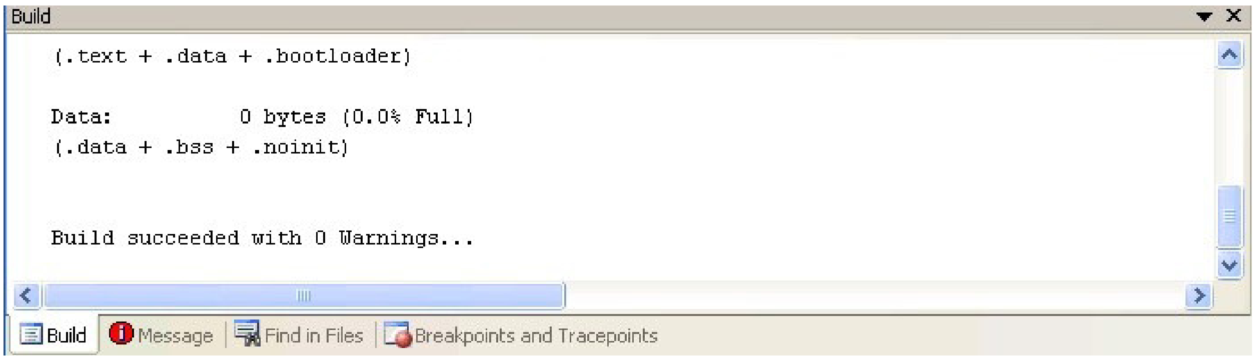
Debug platform:
AVR Dragon
AVR ONE!
AVR Simulator
AVR Simulator 2
ICE200
ICE40
ICE50
JTAG ICE
JTAGICE mkII

Device:
AT90USB646
AT90USB647
AT90USB82
ATmega103
ATmega128
ATmega1280
ATmega1281
ATmega1284P
ATmega128A
ATmega128RFA1
ATmega16

Open platform options next time debug mode is entered

Ver 4.18.692

In *Debug->Select device and debug platform*, set *Debug Platform* to **AVR Simulator** and *Device* to **Atmega16**, then click **Finish**.



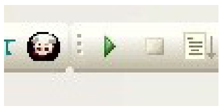
```
(.text + .data + .bootloader)

Data:          0 bytes (0.0% Full)
(.data + .bss + .noinit)


Build succeeded with 0 Warnings...
```

Compile the *simcounter* project with *Build->Build* (or **F7**). When the build is complete, confirm that the last line in the Build window at the bottom of the screen is **Build succeeded with 0 Warnings...** as shown above. If this isn't what you see, somehow you boffed your cut-and-paste, so go back and try it again.

For some reason, there are certain configuration settings in AVR Studio that cannot be set unless you've already started simulating a program, and the following is one of them. To do so, initiate the simulation with *Debug->Start Debugging*. Now in *Debug->AVR Simulator Options* you should set *Frequency* to **8.00MHz**. Go figure..

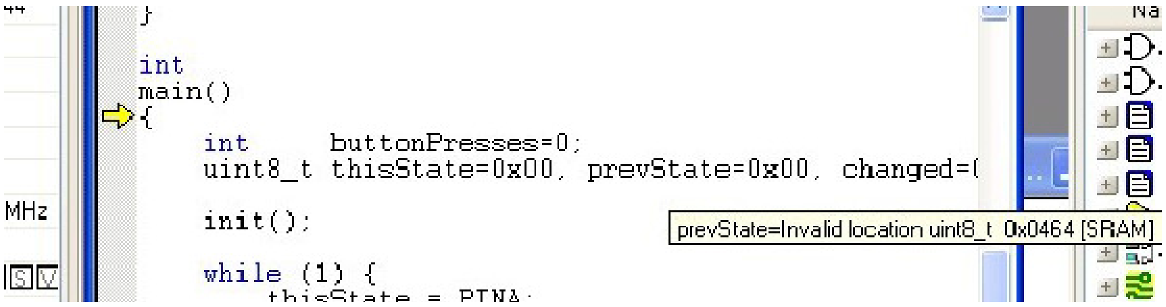
Starting and Stepping through Execution

If you hadn't just done so immediately above, normally you'd start simulating the execution of the program with *Debug->Start Debugging*, or by pressing the green right-facing triangle in the toolbar.



The first thing that the simulator then does is to start-up and then stop before the very

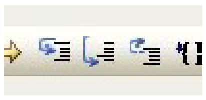
first simulated instruction is executed. It stops here to give you the chance to do all the setup you need before starting the run.



```
}

int
main()
{
    int     buttonPresses=0;
    uint8_t thisState=0x00, prevState=0x00, changed=(

    init();

    while (1) {
        thisState = PINA;
```

prevState=Invalid location uint8_t 0x0464 [SRAM]

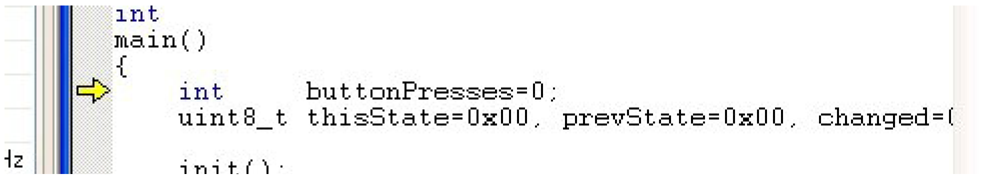
The important thing to notice at this point is that there should be a little yellow right-facing arrow pointing to the opening curly-brace of the function **main()**. This arrow indicates the ***next*** statement to be executed.



At any point when you have control of the running program (that is, when it isn't free-running at fullspeed uninterrupted), you can step through the program slowly, line-by-line, at your own pace. There are three ways to step – *step into*, *step over*, and *step out*. *Step into* and *step over* behave similarly except when a function call is the next statement to be executed. *Step into* steps into (imagine that!) the function and pauses execution at the beginning of the called function. *Step over* executes the entire function as a single statement and advances to the next statement after the function call. *Step over* is a huge time-saver once you know that you don't care about stepping through that function line-by-line.

However if you want/need to see what's going on in the function, by all means step into it. *Step out* is what's used once you're in a function and you no longer care about what's going on inside the rest of it and just want to get back to the code that called the function. *Step Over* completes executing the function and gets you back to where you were when you called it.

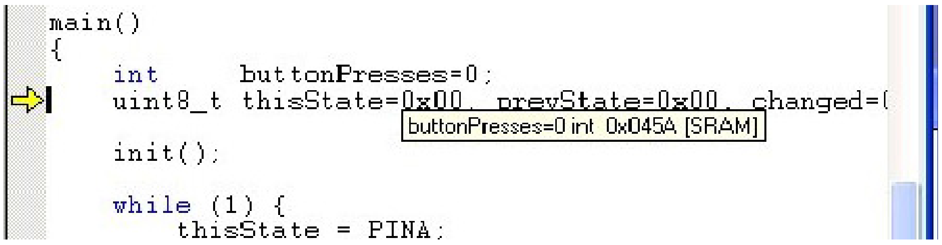
Just to get the hang of single-stepping, we're going to start with *Step Into*. You can do this (and most of the simulator functions) one of three ways. In decreasing order of pain, they are: in the menu bar do *Debug->Step Into*, in the toolbar click on the *Step Into* icon (the one immediately to the right of the yellow arrow in the toolbar, or by pressing the **F11** key. Learn to know and love the **F11** (and **F10**, which is coming-up later) key, as the function keys are by far the easiest ones to use 500 times in an hour.



```
int
main()
{
    int      buttonPresses=0;
    uint8_t  thisState=0x00, prevState=0x00, changed=(

    init();
```

Press the **F11** key once. It should now point to the statement where **buttonPresses** is intialiized.

Note again that this is the *next* statement to be executed. Press the **F11** key again and you should see the arrow pointing to **uint8_t thisState, prevState=0x00, changed;**

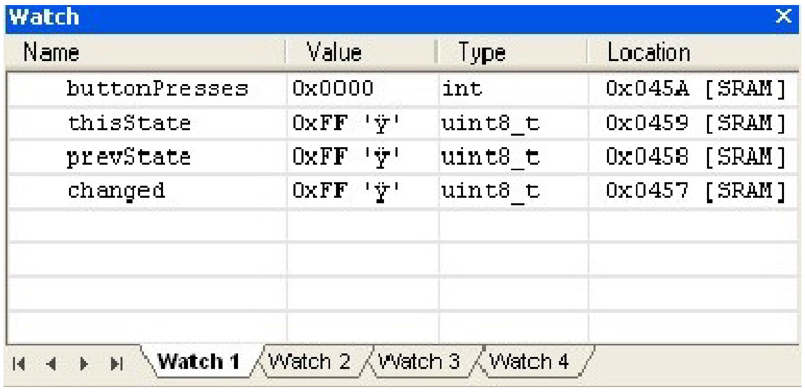


```
main()
{
    int      buttonPresses=0;
    uint8_t thisState=0x00, prevState=0x00, changed=(
                      buttonPresses=0 int 0x045A [SRAM]
    init();

    while (1) {
        thisState = PINA;
```

At this point, wouldn't it be cool to know that the value of **buttonPresses** is actually zero? Well, you can, and there are two ways to do this. The first way is to use your cursor to point to the variable **buttonPresses** anywhere in the source window and it'll pop-up a floating balloon that says that the current value is **0** (along with some other details about the variable). The second way is the power-user way, and you're also going to learn that now.

You can maintain a window on the screen containing any variables that you want to watch along with their current values. This window is called the *Watch window*, and you can open it by doing a *View- >Watch*. Do this and right-click the first cell under the label *Name* and click *Add Item*. If you enter **buttonPresses,** it will give you an entry showing the current name and value for this variable.

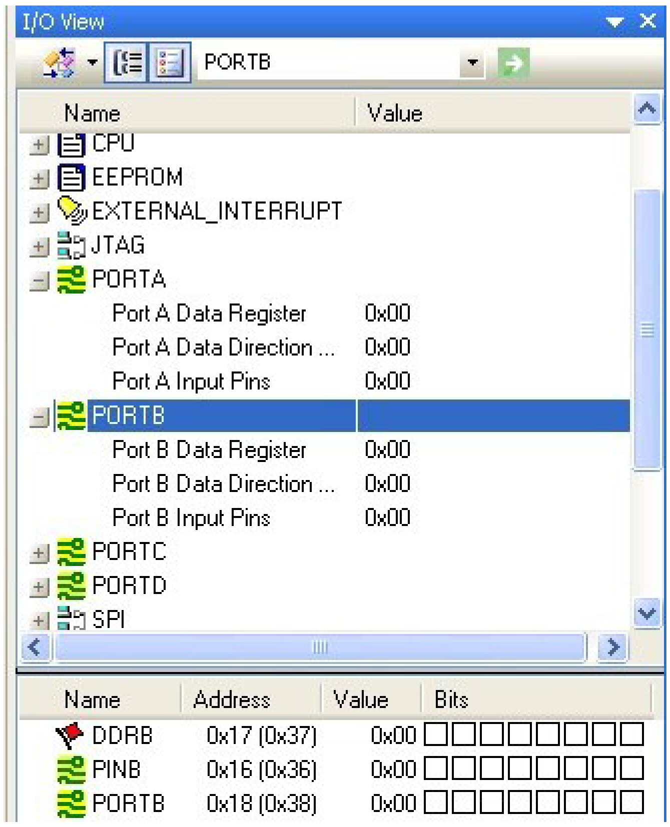
Not only that, this value will be updated as you execute the program under your control (it isn't updated as the program free-runs though). An alternate way to add a new variable to be watched is to doubleclick in an empty *Name* field cell and you can enter a variable name that way. If that weren't enough, there's third way to add a variable to the watch window – just right-click on the variable name anywhere the editing window and select *Add Watch*.



Name	Value	Type	Location
buttonPresses	0x0000	int	0x045A [SRAM]
thisState	0xFF 'ÿ'	uint8_t	0x0459 [SRAM]
prevState	0xFF 'ÿ'	uint8_t	0x0458 [SRAM]
changed	0xFF 'ÿ'	uint8_t	0x0457 [SRAM]

While you are here, use any of these three methods to add watch entries for the variables **thisState**, **prevState** and **changed**. You should notice at this point in program execution that we have not yet initialized these three variables. If you look where the yellow arrow is pointing, we're still before their initialization statements. They are currently what's known as “initialized to garbage”. This is programmer-speak for a variable that has been declared but not yet set (initialized) to a known value – it contains whatever garbage was there when that particular memory location was last used. It is a very, very, very bad thing to ever use a value that has not been initialized. Multibillion dollar satellites (and perhaps your job with it) can be lost forever when this happens, so it's usually best to try to avoid it.

So, observe that these three variables currently contain some random, likely non-zero, value. Then go ahead an press **F11** again to execute their initialization statements. Zero now for all of them!



Name	Value
CPU	
EEPROM	
EXTERNAL_INTERRUPT	
JTAG	
PORTA	
Port A Data Register	0x00
Port A Data Direction ...	0x00
Port A Input Pins	0x00
PORTB	
Port B Data Register	0x00
Port B Data Direction ...	0x00
Port B Input Pins	0x00
PORTC	
PORTD	
SPI	

Name	Address	Value	Bits
DDRB	0x17 (0x37)	0x00	☐☐☐☐☐☐☐☐
PINB	0x16 (0x36)	0x00	☐☐☐☐☐☐☐☐
PORTB	0x18 (0x38)	0x00	☐☐☐☐☐☐☐☐

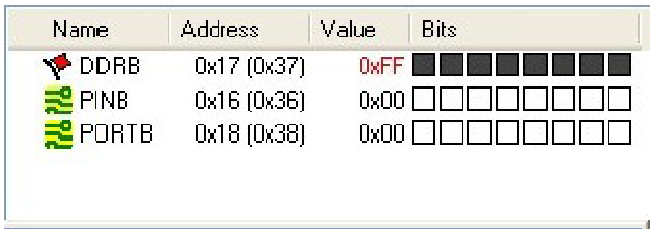
Before you go any farther you're going to learn how to view the I/O registers for the microcontroller.

All PORTA-related registers can be viewed by expanding the **PORTA** group (click the plus-sign) in the right-hand *I/O View* window, as well as for all of the other functional groupings of I/O registers. Here you will find the current simulated values for **PORTA**, **DDRA** and **PINA** both there as well as in their own more-detailed-view window below. Go ahead and expand the **PORTB** set also. Now if you can't see the full contents of each of these entries' columns, especially in the lower of the two windows, stretch your windows out so that you can see everything. In particular, you want to be able to see all eight squares in the *Bits* column for each of the three **PORTA/PORTB**-related registers. These types of

registers are very important as they are the chip's interface to the outside world.

At this point, program execution should be right before the function call to **init()**. Let's demonstrate S*tep Into* first. The first time that you hit **F11** you'll find yourself entering **init()**. As you hit **F11** to execute the following two lines one at a time,

DDRA = 0x00; // PORTA used entirely for input DDRB = 0xff; // PORTB used entirely for output

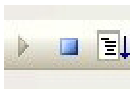


Name	Address	Value	Bits
DDRB	0x17 (0x37)	0xFF	■■■■■■■■
PINB	0x16 (0x36)	0x00	□□□□□□□□
PORTB	0x18 (0x38)	0x00	□□□□□□□□

You should notice that the values for DDRA and DDRB (also known as Port A/B Data direction register) in the *I/O View* window are updated. Especially note that the hollow squares in the lower window are blackened for bits that contain high (1) values. This representation is very useful for quickly visualizing which bits are set within an 8-bit value. In the case of **DDRB**, all bits are high/black – assuming that you have the PORTB group selected in the upper window. You will often be using these windows frequently when following your code's execution.

At this point you should be done executing the last statement in **init()** and ready to return. Using **F11** will return you to the next statement after the one that called this function.

Since we've missed our only opportunity to demonstrate the use of *Step Out* (because there's only one function call in our demonstration program, and we're never calling it again), let's re-run the program from the beginning. You'll do this often when you get to a point where you've missed something that you mindlessly blew-over and need to start over again. Unfortunately, few simulators are able to backup through a simulation, so you'll have to start at the beginning again.



To halt execution and start over again, press the *Stop* icon in the menu bar, the blue square (or *Debug->Stop Debugging*). To restart, as before, press the green triangle (essentially, start) to continue and halt at the first executable statement. Press **F11** until you get just inside **init()**. Rather than stepping through the rest of the function that you no longer care about, you can use *Step Out* to leave this function. As usual, you can use the *Step Out* icon to do this, *Debug->Step Out*, or the appropriate function key.

To demonstrate the final stepping control, do a *Debug->Stop Debugging*, then *Debug->Start Debugging* to start the program again, and **F11** until you are just about to call **init()** again. Now use *Step Over* to execute **init()** without bothering to step into it. You'll do this often once you are comfortable with what a function does and no longer care about crawling through it again. Stepping into functions that you don't care about is often a huge waste of time. Even better, notice that **F10** is the keyboard shortcut for *Step Over*.

F10 is even more useful than **F11** in many circumstances. To be honest, I never know remember which one is which. I'll either use the menu or the toolbar to pick which one that I want the first time, look at the FunctionKey label that is displayed, then just use the FunctionKey from that point on. Just trust me and use the keys – repetitive-stress injuries aren't pretty and every mouse click adds up, trust me.

By the way, one thing that's easy to do is to keep on pressing **F11** and then end up in a function you didn't intend to enter. In that case, just do a *Step Out* and you're back to where you wanted to be.

In summary, those are the three flavors of single-stepping and how each of them are useful, or more efficient, depending on how deliberate you need to be when stepping through code.

Viewing/changing Variables' and I/O Registers' State

At this point you should have just entered the **while** loop. This loop is coded to run forever until **PINA** is read and has the value **PINA_END_VALUE** (all buttons pressed), at which time it breaks out of the loop. Otherwise, every time through the loop the current contents of **PINA** are compared to the previous value to see if any bits have changed, indicating that a button has changed state. Press **F10** a couple dozen times and see if you can figure out what the program is doing so far.

It should be eventually become obvious that nothing's going to change because, well, nothing's changing. Clearly there aren't any buttons connected our simulated ATmega16 on **PINA/PORTA**, and until that happens, you'll be stuck only executing the boring part of the loop forever. However, a key feature of the simulator is that you can manually change values on-the-fly to get the program to behave like physical input changes have occurred on the chip's digital inputs.

First, let's change the variable **thisState** which contains a copy of the current contents of **PINA**.

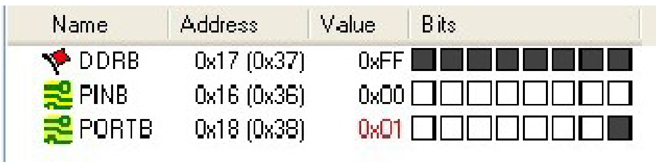
Open the *Watch* window that was introduced before if it's no longer around. If the variable **thisState** is no longer being watched for some reason, add it again as was done above. Single-step with **F10** until you get to the following statement:

if (thisState == PINA_END_VALUE)

So far it's been the case that **thisState** has always been zero, causing you to do little in the rest of the loop and to return to the top again. However, let's force a change. Use the *Watch* window to change the value of **thisState** from **0x00** to **0x01**. You do this by double-clicking on the **0x00** value and entering **0x01** in its place (and pressing **ENTER/RETURN**). Now, with a non-zero value for **thisState** at this point, pressing **F10** should eventually get us into the **if** block that we've never been in before. The first press of **F10** should get you past the **break** statement, as usual.

Another **F10** will cause **changed** to be set to the difference between the current value and the value from the previous time through the loop. **Changed** becomes 0x01 because the lower bit has changed since the last time, which then gets you to the inner **for** loop. By doing enough **F10**s and following along carefully, you should see that a button press is detected, and **buttonPresses** is incremented to **1**.

If you continue **F10**-ing enough you'll find yourself back in the outer loop. Assuming that you don't manually alter the value of **thisState** as we did the last time, **PINA** will be read again as **0**, transferred into **thisState**, and it'll look like the low bit has changed back to **0**, signifying that the button has been released. You'll head into the inner **for** loop, nothing will appear as a new key *press*, and eventually you'll be back to where you started, everything zero, nothing changing, looping in the outer part of the **while** loop forever.

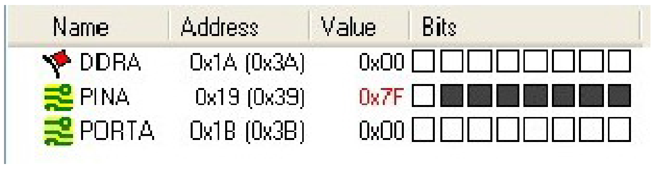


Name	Address	Value	Bits
DDRB	0x17 (0x37)	0xFF	■■■■■■■■
PINB	0x16 (0x36)	0x00	□□□□□□□□
PORTB	0x18 (0x38)	0x01	□□□□□□□■

Interesting things happened along the way though. **buttonPresses** is now set to one because we've “seen” a button press. And if you look in the *I/O View* window, you should notice that the value of **PORTB** has changed to reflect the running count of presses detected. If you click on **PORTB** in the upper window, you'll see the detailed view of the **PORTB**-related registers in the lower window. Since **PORTB** would be connected to LEDs in our “real” hardware, one of them would now be lit-up, just like the blackened-square in the lower-right window, in the 0x01 bit position.

Another important feature in addition to being able to change variables' values on-the-fly is that you can also change the I/O registers' values on-the-fly. Just as you manually changed **thisState** to modify the captured value of **PINA**, you can modify **PINA** directly in a similar fashion. Go ahead and click on **PORTA** in the upper I/O View window. **PINA**'s bits in the lower window should all be zero –

clear squares in the *Bits* display. Now click on one of the clear squares for **PINA**, and you'll see it switch to black. You've turned this bit on manually, and you'll notice that the displayed numerical value changes with it, highlighted in red. This red high-lighting is very helpful feature also, and it's used to point out what value was most recently changed on the display.



Name	Address	Value	Bits
DDRA	0x1A (0x3A)	0x00	☐☐☐☐☐☐☐☐
PINA	0x19 (0x39)	0x7F	☐■■■■■■■
PORTA	0x1B (0x3B)	0x00	☐☐☐☐☐☐☐☐

Go ahead and press on all these squares to light them all up ***except one***. This would denote that all buttons except one are currently being pressed/held at the same time. Now if you go back and step though the **while** and **for** loops, you'll see these additional (fake) button presses counted, eventually resulting in a **buttonPresses** value of **8**, and the binary representation for eight displayed on **PORTB**.

And just as you can simulate button presses, you can simulate releases by clearing the bits in the display for **PINA** also. If you set all eight **PINA** bits at the same time, that's the magic sequence that causes the program to break-out of the outermost **while** loop and essentially end the program. All buttons pressed results in **PINA** being **0xFF**, to which we've associated the constant label **PINA_END_VALUE.**

So in summary, you've seen in this section that you're able to modify C-language variables on-the-fly in the *Watch* window, and able to modify **PORT** input pins on-the-fly, and can view their output state in the *I/O View* window.

TASKS

1. Modify the program so that it acts as a “dead man's switch”, that is, that an alarm sounds whenever a button is not being pressed. Assume that an alarm is connected to the low pin of PORTC and that it sounds when you drive that line's output high.

2. Modify the program to detect when a button is pressed when one are more buttons are already down, and sound the alarm when this occurs.

3. Use the existing stimulus file to drive the same sequence of LEDs flashing as output on **PORTB**. When any button is pressed on **PORTA**, the corresponding LED should be lit for the duration of the press, and turned-off when released.

CONTROL QUESTIONS

1. How to create a new project?
2. How to build a project?
3. How to configure a project?
4. How to debug a project?

SUPPLEMENTAL FILES

simcounter.c:

```
/*
* simcounter.c
*
* This program is primarily intended to be used in the AVR Studio
* Simulator for demonstration of debugging features.
* This program counts button presses on eight buttons on PORTA,
* and continuously displays the current count on eight LEDs
* on PORTB.
*
* By going through the exercises that are paired with this program,
* you will become familiar with the AVR Studio Simulator and
* Debugger.
*/
#include <avr/io.h>
Eric B. Wertz 2010/02/04 (v0.02)
SJSU ME106
#define PINA_END_VALUE 0xFF
void
init()
{
DDRA = 0x00; // PORTA used entirely for input
DDRB = 0xff; // PORTB used entirely for output
}
int
main()
{
int buttonPresses=0;
uint8_t thisState=0x00, prevState=0x00, changed=0x00;
init();
while (1) {
thisState = PINA;
if (thisState == PINA_END_VALUE)
break;
changed = thisState ^ prevState;
if (changed != 0) {
for (int i=0; i<8; i++) {
uint8_t theBit = (1 << i);
if ((changed & theBit) && (thisState & theBit)) {
buttonPresses++;
PORTB = buttonPresses;
}
}
}
prevState = thisState;
}
PORTB = buttonPresses; // just a place to put a final breakpoint
while (1)
;
/*NOTREACHED*/
}
```

simcounter-PORTA.sti:

```
00000000:00
00010000:01
Eric B. Wertz 2010/02/04 (v0.02)
SJSU ME106
00011000:00
00012000:01
00013000:00
00014000:01
00015000:03
00016000:01
00017000:03
00018000:01
00019000:00
00020000:0F
00021000:00
00022000:03
00023000:02
00024000:01
00025000:00
00100000:FF
99999999:00
```

REPORT FORM

German-Russian Institute of Advanced Technologies

TU-Ilmenau (Germany) and KNTRU-KAI (Kazan, Russia)

Laboratory work 1

«AVR Studio Simulator Introduction and Exercises»

Student:	____________________
Teacher:	____________________

Kazan 2014