Installation & Verwendung

Version 1.0

WinAVR & AVR Studio 4

FACHHOCHSCHULE
TECHNIKUM WIEN

Allgemeines

Der eStick ist ein USB basiertes Entwicklungs-Board für Atmel AT90USB162 Mikrocontroller. Einfache, anschauliche Anwendungen und Beispiele basierend auf dieser Hardware Plattform verfolgen das Ziel jedem Interessierten an der Technik einen näheren Einblick in die Elektronik zu geben. Die benötigte Entwicklungssoftware ist frei über die Webseiten des Elektronik Studiengangs der Fachhochschule Technikum Wien verfügbar bzw. dort verlinkt. Der eStick selbst kann über die folgende Adresse gegen einen geringen Unkostenbeitrag erworben werden:

Fachhochschule Technikum Wien
Sekretariat Studiengang Elektronik, 5. Stock
Höchstädtplatz 5
A-1200 Wien
WEB: http://www.technikum-wien.at/bel
FAX: +43 1 333 40 77 268

Für die Entwicklung eigener Anwendungen für und mit dem eStick stehen verschiedene Varianten zur Verfügung. „Low-Level“ Programmierung in Assembler oder mit mit einer Programmiersprache einer höheren Abstraktion wie C. Assembly Programme resultieren zumeist in effizienteren Applikationen und vermeiden Probleme und Fehler die durch Verwendung eines doch eher komplexeren Compilers – insbesondere im Kontext von *Embedded Systems* – ungewollt passieren können. Andererseits erlaubt die Verwendung einer Hochsprache effizienteres Programmieren und ein leichteres *retargeting* einer Applikation auf einen anderen Mikrocontroller.

In beiden Fällen erfolgt die Programmeingabe in der Regel mittels Texteditor. Das Programm selbst wird dann mittels *Compiler*, *Assembler*, *Linker* und *Locator* in ein ausführbares Programm-Image übersetzt. Bei Embedded Systems (Mikrocontroller basierten) Anwendungen ist dies meist eine Textdatei im *Intel-HEX* Format. Das entwickelte Programm kann nun entweder mit Hilfe eines *Simulators* rein in Software oder mit Hilfe eines *Debuggers* auf der Hardware evaluiert werden, ob dies den prinzipiellen Anforderungen genügt. Letzten Endes wird das Programm-Image mit Hilfe eines Flashtools resident auf die Hardware geladen, von wo aus diese dann automatisch nach einem Reset exekutiert werden kann.

Das vorliegende „Getting Started“ zeigt die Programmentwicklung in der Programmiersprache C mit Hilfe des freien GNU C-Compilers WinAVR integriert mit der Entwicklungsumgebung AVRStudio4 der Firma Atmel, die u.a. neben einem Assembler auch einen Simulator bietet.

Installation

Vor der Installation von WinAVR muss das Programmpaket aus dem Internet geladen werden:

- WinAVR: http://winavr.sourceforge.net » Download (23 MB)

Nach dem Download starten Sie das Installationsprogramm und folgen Sie einfach den Anleitungen des Installationswizards. Im folgenden Beispiel wurde das Verzeichnis c:\estick\winavr gewählt.

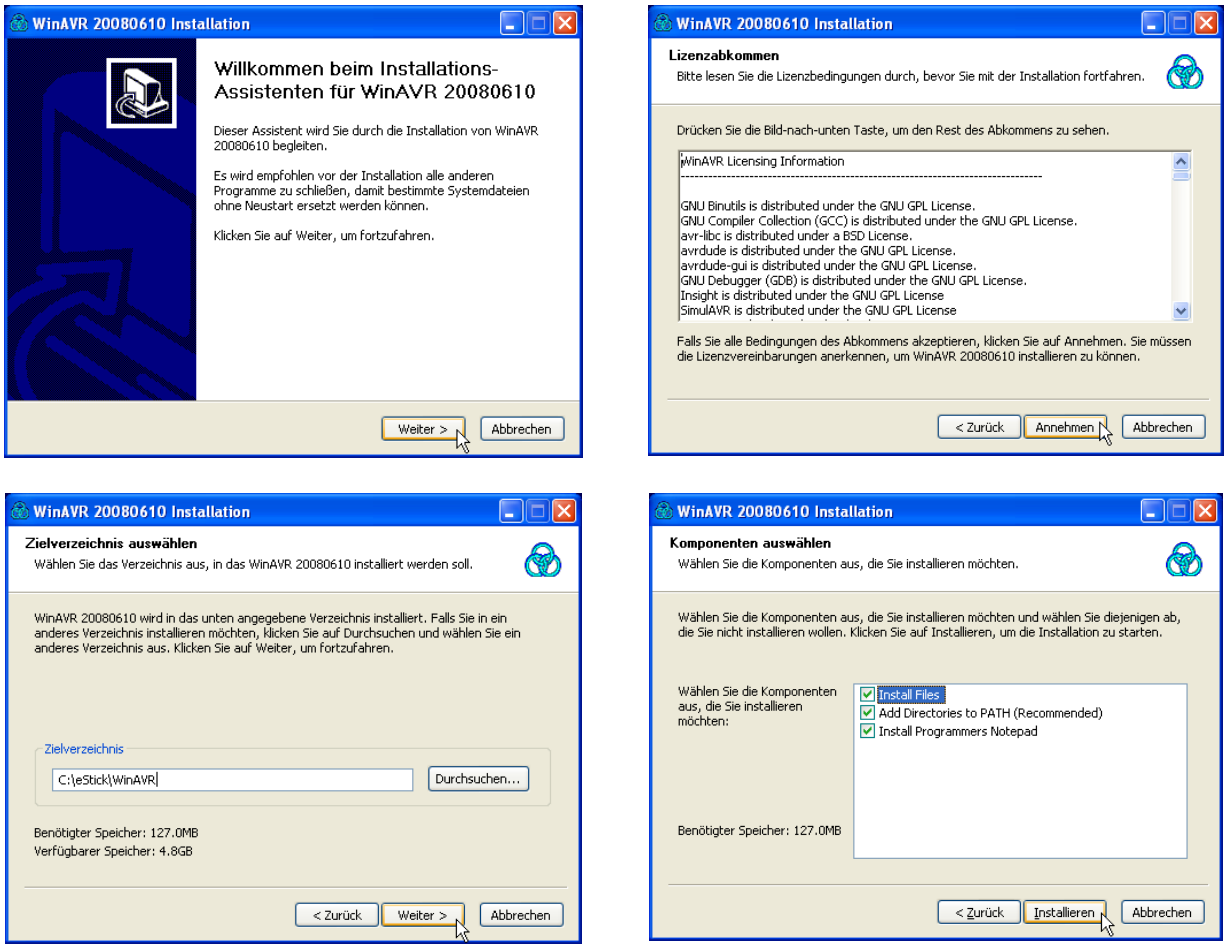

Abbildung 1: WinAVR Installation - Screenshots

Anmerkung: Mit WinAVR können bereits C-Programme in Intel-Hex Programm Images übersetzt werden. Einzig ein Simulator für den AT90USB162 Mikrocontroller, der auf dem eStick zum Einsatz kommt, fehlt hier.

Im nächsten Schritt muss das AVRStudio4 Programmpaket aus dem Internet von einer der beiden folgenden Adressen geladen werden:

- http://www.atmel.no/beta_ware (90MB, aktuellste Versionen)
- http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725 (90MB, Registrierung erforderlich)

Die folgenden Screenshots zeigen die Standard Installation bei der Sie Lizenzbedingungen von Amtel bestätigen, ein Installationsverzeichnis angeben (z.B: c:\eStick\AtmelAVR) und den USB Treiber aktualisieren müssen.

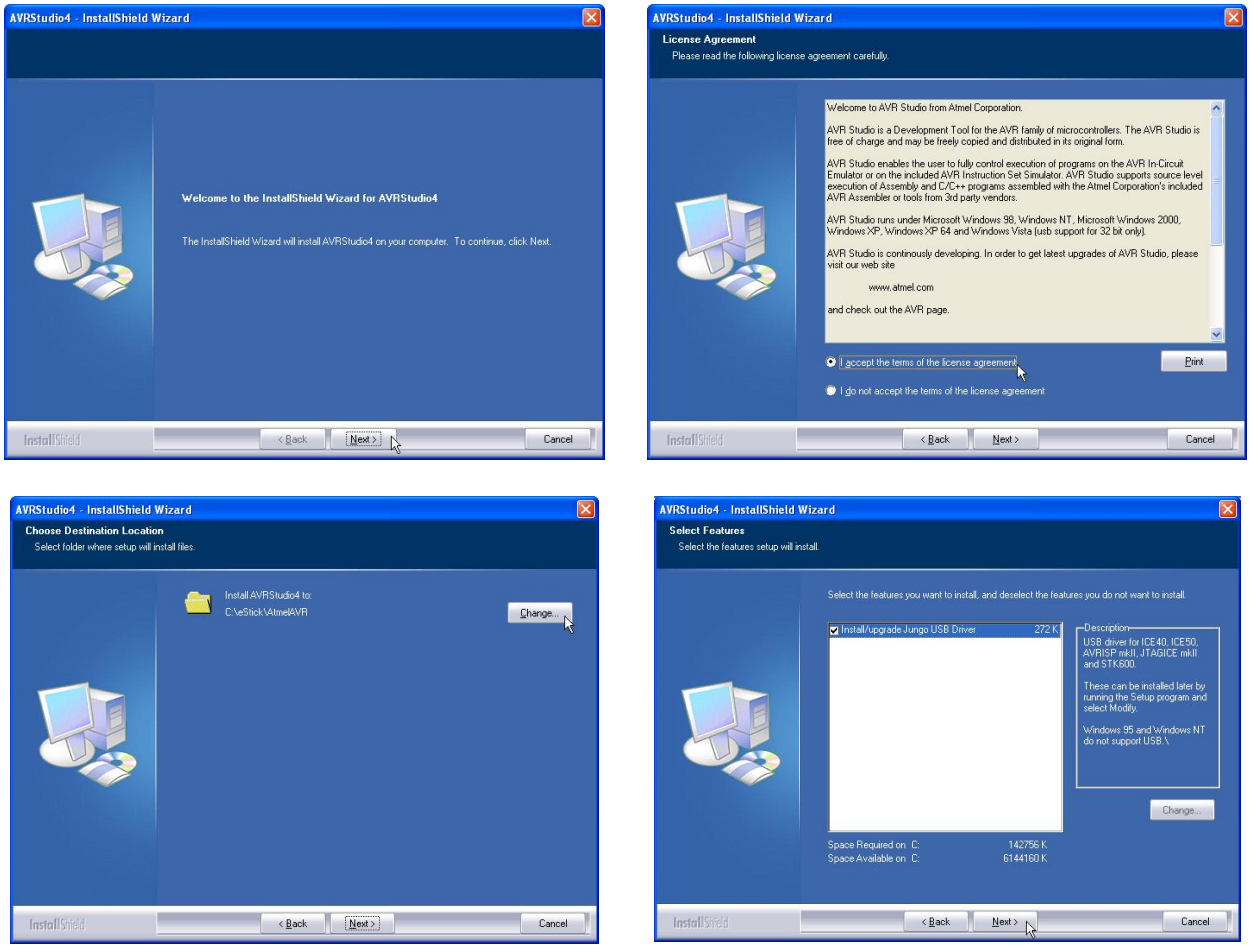

Abbildung 2: AVRStudio4 Installation – Screenshots

Erstellen eines ersten einfachen Projekts – blinky:

Nach erfolgter Installation starten Sie das AVRStudio4 über das Windows Startmenü, siehe Abbildung 3. Nach dem Programmstart wird standardmäßig ebenfalls ein „Projekt-Wizard“ gestartet; dieser kann auch jederzeit später über das Menü Project » Project Wizard gestartet werden.

Im ersten Dialog wählen Sie „New Project“; im zweiten Dialog „AVR GCC“, einen Projektnamen (z.B. blinky) und ein Arbeitsverzeichnis wo die Daten abgespeichert werden sollen. Im dritten Dialog wählen Sie schließlich die Debug Plattform und das Device – für den eStick ist hier AVR Simulator und AT90USB162 zu selektieren.

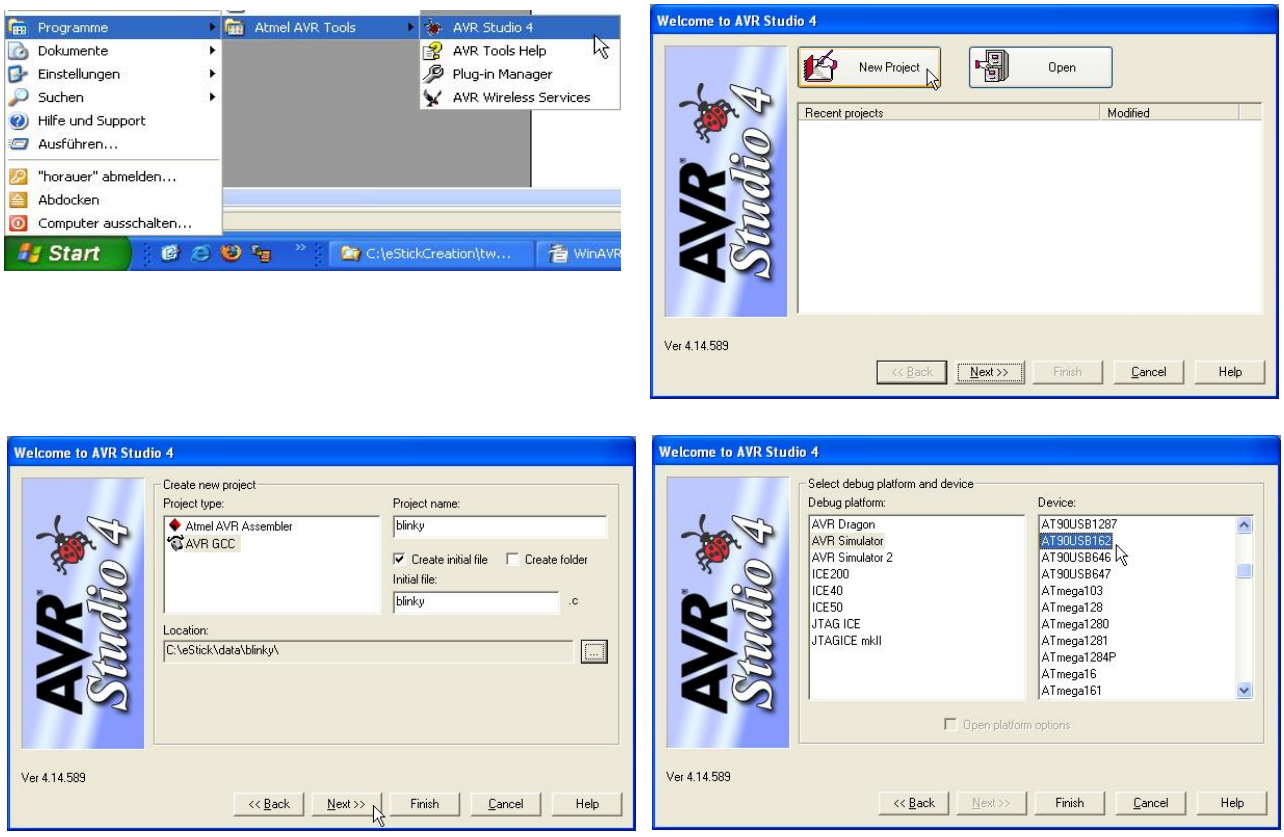

Abbildung 3: AVRStudio4 – Project Wizard

Wie bei den meisten Entwicklungsumgebungen üblich besteht auch das AVRStudio4 aus mehreren Bereichen. Im Bereich Projekt-Verwaltung wird das Projekt verwaltet indem Code Dateien zu einem Projekt (über die rechte Maustaste) in den entsprechenden Kategorien hinzugefügt und davon auch wieder entfernt werden können. Im Text-Editor Bereich wird editiert und im Bereich Output werden Status und Fehler Meldungen des Compilers beim Übersetzen des Quellcodes ausgegeben. Über die Bereiche Simulator und Simulator Details können die Werte einzelner Mikrokontroller Interna beobachtet und untersucht werden. Letzteres ist jedoch nur im „Debug-Modus“ möglich nachdem Sie ein Projekt erfolgreich übersetzt haben und über das Menü Debug » Start Debugging in diesen Modus gewechselt sind.

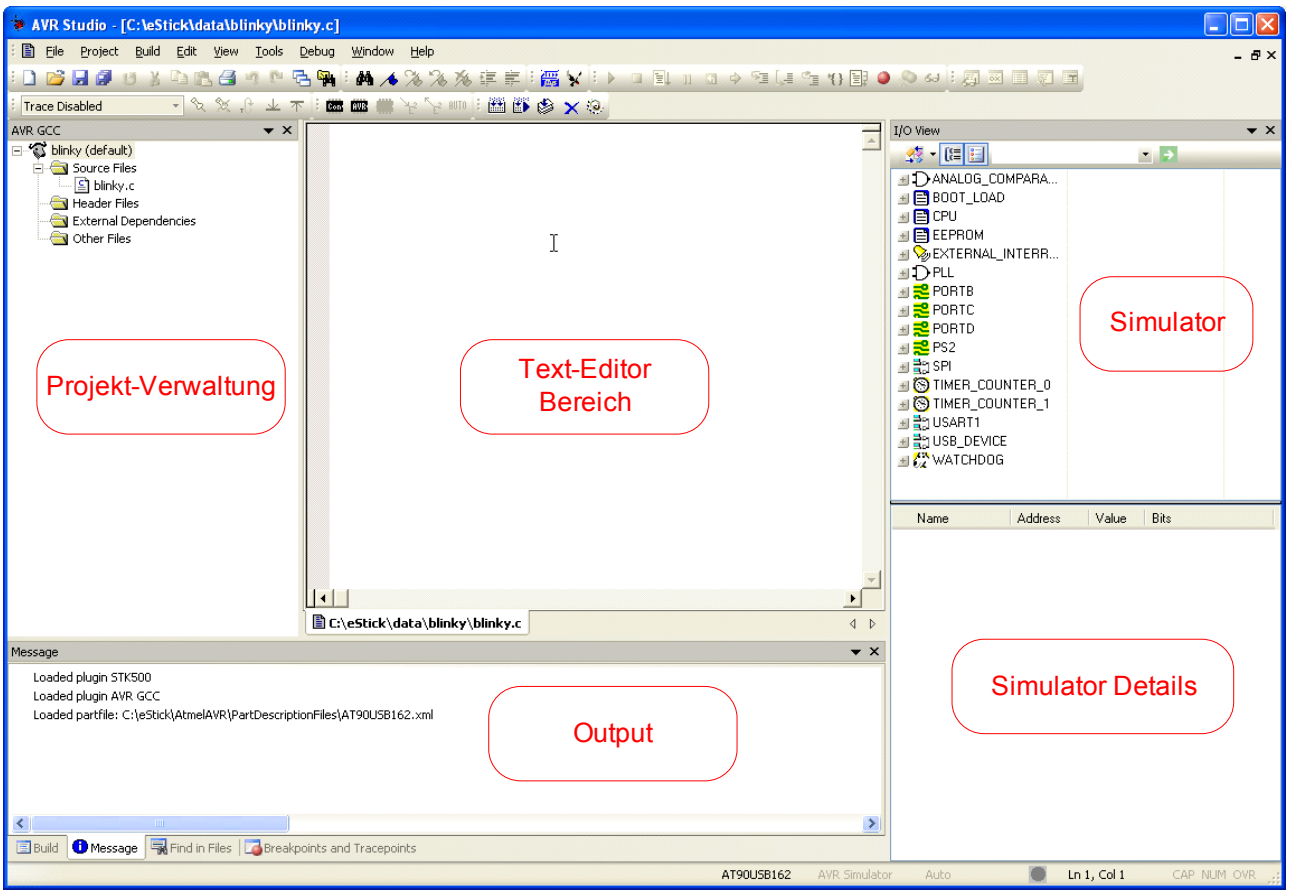

Abbildung 4: AVRStudio4 – Arbeitsbereiche

Geben Sie im nächsten Schritt, das Programm aus Listing 1 im Text-Editor Bereich ein und speichern Sie dieses unter dem Dateinamen blinky.c ab. Erläuterungen zum Source Code Listing 1 finden Sie im Anschluss an diesen Abschnitt.

```
01: #include <avr/io.h>
02: #include <util/delay.h>
03:
04: /* C- Macro Definitions */
05: #define SET_LED(x) PORTB=PORTB&(~(1<<x))
06: #define CLEAR_LED(x) PORTB=PORTB|(1<<x)
07:
08: int main(void)
09: {
10:   DDRB = 0xFF;      /* set the Port PORTB to output       */
11:   PORTB = 0xFF;     /* write all 1s to PORTB              */
12:   DDRC = 0x04;      /* set the Port Pin PORTC.2 to output */
13:   PORTC = 0x04;     /* set the Port Pin PORTC.2 (LE)      */
14:   while(1)          /* endless loop                       */
15:   {
16:     SET_LED(0);     /* turn LED 0 on                      */
17:     _delay_ms(50);  /* wait 50 ms                         */
18:     CLEAR_LED(0);   /* turn LED 0 off                     */
19:     _delay_ms(50);  /* wait 50 ms                         */
20:   }
21: }
```

Listing 1: blinky.c

Über den Dialog Project » Configuration Options können grundlegende Einstellungen zum Projekt vorgenommen werden, wie z.B. die Taktfrequenz `8000000` oder die Optimierungslevel des Compilers. (`-O0` ... keine Optimierung, `-O3` oder `-Os` ... hohe Optimierung).

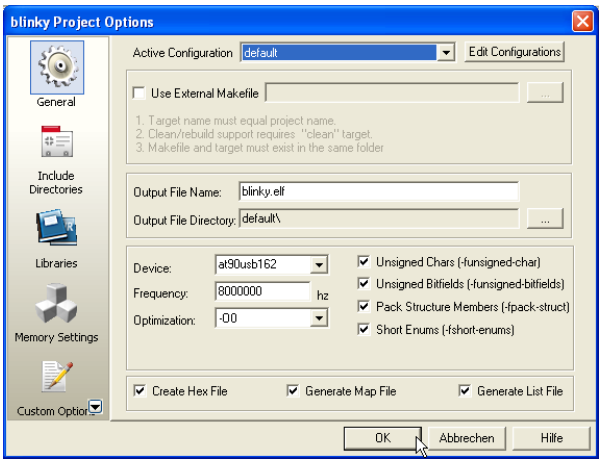

Abbildung 5: Einstellung der Optionen für das Projekt

Im Anschluss kann das Projekt übersetzt werden; wählen Sie entweder den Build Toolbar Button oder über das Menü Build » Build bzw. Rebuild All. Treten Fehler auf so werden diese Output-Bereich aufgelistet, andernfalls erhalten Sie hier eine Erfolgsmeldung. Im Arbeitsverzeichnis wurde ein Unterverzeichnis „default“ angelegt, wo der Compiler nun eine Datei blinky.hex angelegt hat. Diese kann mit einem Flashtool, z.B. dem eStickFashTool auf den eStick resident programmiert werden. Alternativ kann die Programmausführung im Simulator evaluiert werden (siehe den folgenden Abschnitt Simulation).

Zum flashen starten Sie das Flashtool, verbinden Sie den eStick bei gedrückt gehaltener Taste mit einem freien USB Port, laden/öffnen Sie die Datei blinky.hex, führen Sie die Programmierung durch (ERASE, PROGRAM, VERIFY) und starten Sie die Applikation. Danach sollte eine Leuchtdiode am eStick langsam blinken.

Erläuterungen zum Source Code in Listing 1:

In den Zeilen `01` und `02` werden vordefinierte Bibliotheken über die Preprozessoranweisung `#include` eingebunden. Diese befinden sich im Unterverzeichnis „avr“ wo WinAVR installiert wurde. Die Datei `io.h` bindet Registerdefinitionen ein, sodass diese mit demselben Namen, wie im Datenblatt des Mikrocontroller angegeben, verwendet werden können. Erst durch das Einbinden dieser Datei kann im Folgenden auf Register wie `DDRB` oder `PORTB` zugegriffen bzw. geschrieben werden. Die Datei `delay.h` definiert diverse Wartefunktionen wie bspw. `_delay_ms()`, die die Programmausführung um eine angegebene Anzahl an *ms* verzögert.

In den Zeilen `08` bis `21` ist die `main()` Funktion realisiert. Jedes C Programm benötigt eine

`main()` Funktion – hier beginnt und endet zumeist die Anwendung. Die Zeilen `10-13` umfassen den Initialisierungsteil, der einmal am beginn der Exekution der Anwendung durchlaufen wird. In den Zeilen `14-20` wird eine Endlosschleife realisiert; jedes Mikrocontroller Programm weist im Hauptteil typischerweise eine Endlosschleife auf, da *Embedded Systems* Applikationen in der Regel nicht enden.

Im Initialisierungsteil in Zeile `10` wird der `PORTB` des Mikrocontrollers, an dem die Leuchtdioden über einen D-Latch Baustein angeschlossen sind, zunächst auf Ausgang geschaltet (`DDRx` steht für Data Direction Register). Danach wird in Zeile `11` auf allen acht `PORTB` Leitungen eine 1 ausgegeben (0xFF ist die hexadezimale Zahlendarstellung für das binäre Bitmuster 1111 1111). Die Leuchtdioden sind low aktiv, und leuchten wenn diese mit einer 0 angesteuert werden; d.h. initial sind somit alle Leuchtdioden dunkel.

In Zeile 12 wird das dritte Bit von `PORTC` (d.h. `PORTC(2)`) auf Ausgang konfiguriert. Danach wird in Zeile 13 der Pin Latch Enable des Latch Bausteins auf 1 gesetzt indem an PORTC ein Bitmuster ausgegeben wird, dass an der dritten Position eine 1 aufweist (0x04 hexadezimal entspricht binär 0000 0100). Wann immer das Latch Enable eines Latch Bausteins 1 ist, ist dieses transparent, d.h. die Eingänge werden an die Ausgänge durchgeschaltet. Ist der Latch Enable Pin jedoch 0 so bleibt der letzte Zustand der Ausgänge erhalten.

In Zeile `16` wird der Pin `POTRB(0)` über ein Makro, welches in Zeile `05` deklariert wurde, auf 0 gesetzt. Dadurch wird die dort – über den nun transparent konfigurierten Latch Baustein – angeschlossene Leuchtdiode eingeschaltet. Danach wird die weitere Programmausführung über die Wartefunktion `_delay_ms(50)` verzögert. Nach dem verstreichen dieser Zeit wird `PORTB(0)` in Zeile `18` über das in Zeile `06` deklarierte `CLEAR_PORT(x)` Makro wieder auf 1 gesetzt; die Leuchtdiode erlischt. Im Anschluss wird in Zeile `19` abermals gewartet bevor die Endlosschleife erneut bei Zeile `16` erneut von vorne beginnt.

Simulation:

Simulation ist – neben *Debugging* auf der Hardware – eine Möglichkeit um dem Programmierer einen besseren Einblick in die Funktionsweise seines Programms zu geben und somit die Programmentwicklung zu erleichtern. Dazu simuliert AVRStudio4 den gesamten Mikrocontroller in Software und bietet dem Entwickler die Möglichkeit sein Programm Schritt für Schritt ablaufen zu lassen und gleichzeitig Interna des Mikrocontrollers und Programms zu beobachten.

Um den Simulator zu starten muss dieser und der „Target-Mikrocontroller“ zuvor ausgewählt worden sein. Diese wurde bereits zuvor (siehe Abbildung 3) mit Hilfe des *Project Wizards* vorgenommen; nachträglich kann dies auch über das Menü Debug » Select Platform and Device ... vorgenommen werden.

Um die Ausführung des Programms im Simulator zu starten wählen Sie entweder das „Start Debugging“ Icon in der Toolbar oder über das Menü Debug » Start Debugging.

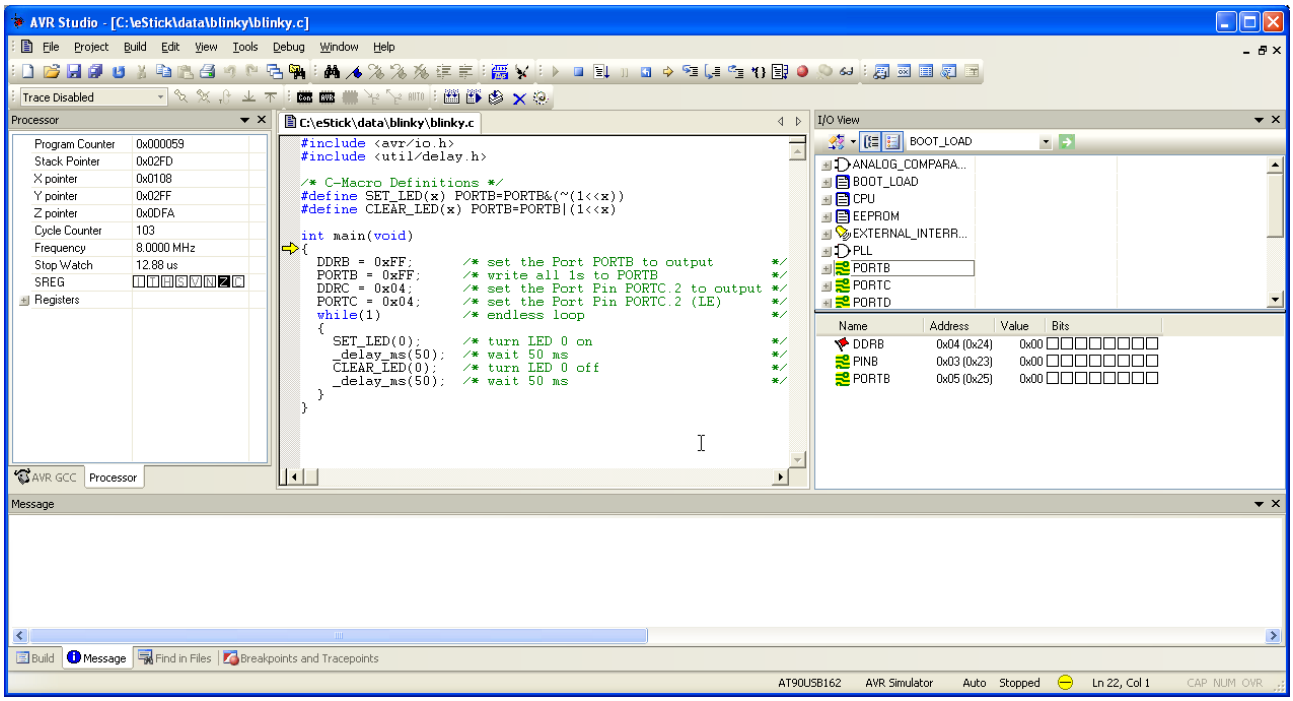

Abbildung 6: AVRStudio4 – Simulator Ansicht

Das Programm (blinky.elf) wird im Simulator geladen; ein kleiner Pfeil erscheint rechts neben der `main()` Funktion, wo die Programmausführung beginnt. Im Simulator-Bereich können einzelne Peripherieelemente – wie z.B. PORTB an dem über den Latch Baustein die Leuchtdioden angeschlossen sind – ausgewählt werden. Die entsprechenden, relevanten Register – in diesem Fall DDRB, PINB und PORTB – und deren Werte werden darunter angezeigt. Über das View Menü kann man sich zusätzliche Dinge wie den Assembler Code, Speicherbereiche, CPU Register oder die Werte von Variablen im so genannten *Watch Window* ansehen.

Bevor die Simulation über Debug » Run, Step Into, etc. ausgeführt wird, sollte die Simulations-frequenz über den Dialog Debug » AVR Simulator Options auf 8MHz gestellt werden (siehe Abbildung 7).

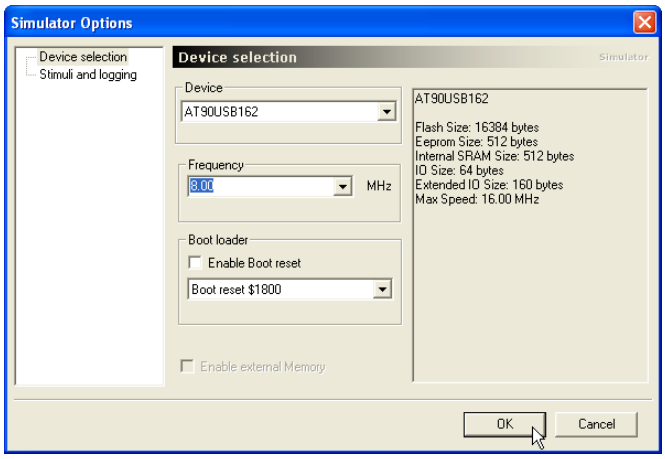

Abbildung 7: AVRStudio4 – Einstellen der Simulationsfrequenz

Über die Debug Toolbar oder das Menü Debug kann das Programm in Einzelschritten abgearbeitet werden, können so genannte *Breakpoints* gesetzt werden (dies sind Stellen im Code wo die Ausführung des Programms anhalten soll) und das Programm bis zu diesen exekutiert werden. Wann immer die Programmausführung hält, können diverse Peripherieelemente und Variablen inspiziert werden.

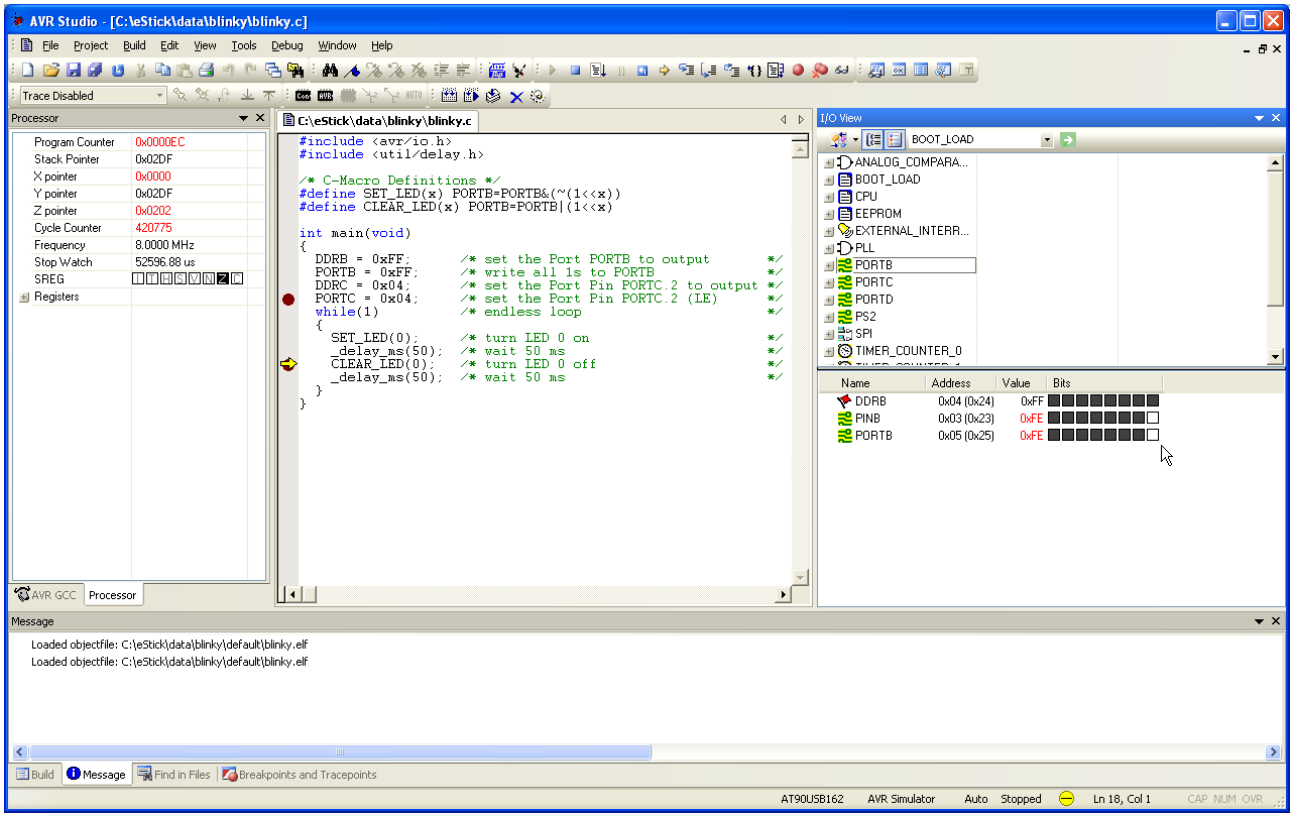

Abbildung 7: AVRStudio4 – Simulation mit Breakpoints

In Abbildung 7 sind bspw. zwei gesetzte Breakpoints sichtbar: der erste in der Zeile mit der Anweisung „`POTRC = 0x04;`“ und der zweite in der Zeile mit der Anweisung „`CLEAR_LED(0);`“. Breakpoints können gesetzt/gelöscht werden, indem der Cursor in die entsprechende Zeile positioniert wird und über Debug » Toggle Breakpoint (Taste F9) ein solcher eingefügt/gelöscht wird. Der gelbe Pfeil markiert die Code Stelle die im nächsten Schritt ausgeführt werden soll. In diesem Fall soll die Leuchtdiode auf dunkel (sprich PORTB auf 1 gesetzt) werden. In der Simulations Detailansicht rechts ist ersichtlich, dass die höherwertigen Bits von `PORTB` und PINB noch dunkel eingefärbt (entspricht einer 1) sind und `PORTB(0)` hell markiert (entspricht einer 0) ist. Im linken Bereich sind Details zum Prozessor dargestellt. So weist bspw. „Stop Watch“ einen Wert von 52596.88us auf. Dies entspricht der Anzahl an Mikrosekunden, die seit dem Programmstart der Simulation verstrichen sind. Dies entspricht ungefähr dem Wert von 50ms für die eine, erste Verzögerung durch die `_delay_ms(50)` Funktion plus dem Zeit für die Exekution der Befehle im Initialisierungsteil und dem Ausführen des `SET_LED(0)` Makros.

Weitere Hinweise und Links:

Diverse weitere Unterlagen und Dokumente können über die Webseiten der Firma Atmel bezogen werden.

- http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725

Anhang – Zahlensysteme:

Für das Programmieren müssen oft Bits an bestimmten Positionen auf 0 oder 1 gesetzt werden. Dazu ist es meist einfacher die Zahlen Hexadezimal anzuschreiben. Mit der binären Schreibweise ist es zwar schneller offenkundig welche Bitposition beeinflusst werden soll, die langen Kombinationen an 0en und 1en sind jedoch sehr rasch fehleranfällig. Daher fast man üblicherweise vier Bits (ein Nibble) zusammen und rechnet mit unten stehender Tabelle die Binärzahl ins Hexadezimale Pendant um. Dass eine Zahl im hexadezimalen Format angeschrieben wird, wird zumeist durch ein vorangestelltes 0x gekennzeichnet.
Beispiel: Das Bitmuster 0b 1011 entspricht der hexadezimalen Zahl 0xB.

Dezimal	Hexadezimal	Binär
0	0x00	0b 0000 0000
1	0x01	0b 0000 0001
2	0x02	0b 0000 0010
3	0x03	0b 0000 0011
4	0x04	0b 0000 0100
5	0x05	0b 0000 0101
6	0x06	0b 0000 0110
7	0x07	0b 0000 0111
8	0x08	0b 0000 1000
9	0x09	0b 0000 1001
10	0x0A	0b 0000 1010
11	0x0B	0b 0000 1011
12	0x0C	0b 0000 1100
13	0x0D	0b 0000 1101
14	0x0E	0b 0000 1110
15	0x0F	0b 0000 1111
100	0x64	0b 0110 0100
255	0xFF	0b 1111 1111

Tabelle 1: Dezimal-, Hexadezimal- und Binärdarstellung