


Arbeitsblatt Nr. 04	Q2 Technikwissenschaft: Digitale Steuerungstechnik	 B S G G
Datum:	Thema: Debugging von Assemblerprogrammen	
Seite 1 von 4	Name:	

Debugging von Assemblerprogrammen

Zur Fehlersuche in Assemblerprogrammen steht bei der Arbeit mit dem Atmel Studio ein interner Debugger namens „Simulator“ zur Verfügung.

Lassen Sie und die wichtigsten Grundfunktionalitäten des Debuggers an einem Beispielprogramm anschauen.

Das Beispielprogramm soll eine LED blinken lassen. Die LED wird hierfür am Digital-Pin 3 des Arduinos angeschlossen. Das ist für den Prozessor ATmega 2560 der Anschluss PE5 (PortE Bit 5).

Die LED wird über einen Vorwiderstand von 330 Ω am Digital-Pin 3 angeschlossen, um den Strom durch die LED zu begrenzen.

Damit die LED blinkt, muss sie für eine bestimmte Zeit eingeschaltet und für eine bestimmte Zeit ausgeschaltet werden.

Dies lässt sich beispielsweise über eine Zeit- bzw. Warteschleife in Form eines Unterprogramms realisieren.

Schauen Sie sich mit mir dazu das Beispielprogramm an.

Nach der Initialisierung von PortE Bit 5 als Ausgang beginnt das Hauptprogramm.

Zuerst wird die LED ausgeschaltet, dann erfolgt mit dem Assembler-Befehl `rcall` ein Unterprogramm-Aufruf der Zeitschleife mit dem Namen `WAIT_10ms`.

`rcall` steht für „Relative Call“ und ermöglicht den Aufruf eines Unterprogramms, welches sich irgendwo im 8 KByte großen Programmspeicher befinden darf.

```

;=====
; Projekt Blink-LED
; Eine LED am PortE Bit5 wird ein- und ausgeschaltet.
; Der Vorgang wird durch eine Warteschleife von 10ms unterbrochen
;=====Start des Assembler-Programms=====
.org      0x0000
;=====Initialisierung=====
INIT:
    sbi    DDRE, DDE5          ; PortE Bit5 als Ausgang (2 MZ)

;====Haupt-Programm in Endlosschleife=====
MAIN:
    cbi    PORTE, PORTE5      ; Schalte LED aus (2 MZ)
    ldi    R30, 30             ; nur für Testzwecke
    ldi    R29, 29             ; nur für Testzwecke
    rcall  WAIT_10ms           ; Aufruf der Warteschleife (4 MZ)
    sbi    PORTE, PORTE5      ; Schalte LED ein (2 MZ)
    rcall  WAIT_10ms           ; Aufruf der Warteschleife (4 MZ)
    rjmp   MAIN                ; Zurück zum Anfang (2 MZ)
;=====

;====Warteschleife für 10ms=====
WAIT_10ms:
; Taktfrequenz des ATmega 2560 ist 16 MHz --> T = 62,5 ns
; 1 MZ dauert 62,5 ns
; 10 ms = 160.000 MZ
; Dauer der Warteschleife: 159.999 MZ
; Konstante Zyklen = (2+2+1+2+2+5) MZ = 14 MZ
; Summe der Zyklen = 160.013 MZ
; Zeitdauer = 160.013 MZ * 62,5 ns/MZ = 0,0100008125 s
;
; Benötigte Register sichern
    push  R30                ; 2 MZ
    push  R29                ; 2 MZ

; Beginn der Verzögerung
    ldi   R30, $A0            ; 1 MZ ($A0 = 160)


LOOP_2:
; 160 * LOOP2 =
; 160 * (5 MZ + 995 MZ) - 1 MZ =
; 159.999 MZ
    ldi   R29, $F9            ; 1 MZ ($F9 = 249)

LOOP_1:
; 249 * LOOP1 = 249 * 4 MZ - 1 MZ =
; 995 MZ
    nop                    ; 1 MZ
    dec   R29                ; 1 MZ
    brne  LOOP_1              ; 2 MZ bei true/1 MZ bei false

    nop                    ; 1 MZ
    dec   R30                ; 1 MZ
    brne  LOOP_2              ; 2 MZ bei true/1 MZ bei false

; Benötigte Register wieder herstellen
    pop   R29                ; 2 MZ
    pop   R30                ; 2 MZ
; Rücksprung
    ret                       ; 5 MZ
;=====

```

Arbeitsblatt Nr. 04	Q2 Technikwissenschaft: Digitale Steuerungstechnik		B S G G
Datum:	Thema: Debugging von Assemblerprogrammen		
Seite 2 von 4	Name:		

Hierzu wird die Rücksprungadresse (das ist die Adresse des nächsten Befehls hinter `rcall`) auf dem sogenannten Stapelspeicher („Stack“) abgelegt.

Ein Unterprogramm wird immer mit dem Befehl `ret` (für „return“) beendet. Der Beginn des Unterprogramms wird durch eine Sprungmarkierung festgelegt. Eine Sprungmarkierung ist eine symbolischer Bezeichnung, gefolgt von einem Doppelpunkt, wie z.B. `WAIT_10ms`.

Bei einem Unterprogramm-Aufruf wird der Programmzähler-Register („Programcounter“, „PC“), und das ist das Register, welches die Adresse des nächsten auszuführenden Befehls enthält, mit der Startadresse des Unterprogramms geladen.

Um den Weg zurück zu finden, wird deshalb die Adresse des Befehls hinter dem `rcall` gesichert, in dem dieser Wert eben auf dem sogenannten Stack abgelegt wird. Der Befehl `ret` lädt dann am Ende des Unterprogramms die auf dem Stack gesicherte Adresse in das Programcounter-Register.

Nach Abarbeitung des Unterprogramms `WAIT_10ms` wird die Verarbeitung im Hauptprogramm fortgesetzt: Die LED wird eingeschaltet, es wird erneut das Unterprogramm `WAIT_10ms` aufgerufen und dann beginnt aufgrund des `rjmp`-Befehls hin zum Programmbeginn alles von vorn.

Somit blinkt die LED mit einer Periodendauer von zwei mal ca. 10 ms; also mit einer Frequenz von ca. 50 Hz. Dies lässt sich mit bloßem Auge allerdings kaum erkennen.

Unterprogramme

Schauen wir uns nun vor dem Debuggen erst einmal das Unterprogramm `WAIT_10ms` an.

Unterprogramme beginnen, wie bereits angegeben, mit einer symbolischen Sprungmarkierung, hinter der der erste Befehl des Unterprogramms zu finden ist. Unterprogramme enden immer mit der Assembleranweisung `ret`. Das kürzeste Unterprogramm ohne irgendwelchen Zweck wäre also eine symbolische Sprungmarke gefolgt von `ret`.

```
LABEL:
    ret
```

Okay...ein Aufruf dieses Unterprogramms „kostet“ neun Maschinenzyklen (MZ), vier für den `rcall`- und fünf für den `ret`-Befehl.

Nicht nur guter Stil, sondern oftmals auch notwendig bei der Erstellung von Unterprogrammen, ist die Sicherung von Registern, die innerhalb des Unterprogramms verwendet werden, und deren Wiederherstellung bevor der Rücksprung erfolgt.

Das Sichern von Registern erfolgt mit dem Befehl `push`. Dieser Befehl kopiert den Inhalt des betreffenden Registers auf den Stack und bewegt den Zeiger auf die nächste freie Speicherstelle vom Stack.

Der Befehl `pop` stellt den Zeiger für den Stapelspeicher auf den zuletzt abgelegten Wert und kopiert diesen in das angegeben Register.

In zuvor dargestellten Beispielprogramm werden im Unterprogramm die Register R29 und R30 verwendet. Deshalb sollte man diese beiden Register zu Beginn des Unterprogramms mit `push` „retten“ und am Ende des Unterprogramms mit `pop` „wieder herstellen“. Sollten (was hier im Beispiel **nicht** der Fall ist!) diese Register im aufrufenden Programmteil von Bedeutung sein, haben diese nun wieder ihren ursprünglichen Wert.

Aktivieren des Debuggers

Einem Projekt muss der zu verwendende Debugger zugeordnet werden. Dies erfolgt in den Projekteigenschaften (zu finden im Menü¹ „Projekt“ oder im Menü „Debuggen“).

Atmel Studio verfügt über einen Software-Debugger mit dem Namen „Simulator“. Im Fenster der Projekt-Eigenschaften wählt man „Tool“ aus und in dem Kombinationslistenfeld dann den Eintrag „Simulator“ (siehe Bild rechts).

Dieser Debugger arbeitet ausschließlich im Arbeitsspeicher des PCs² und kann daher auch ohne das angeschlossene Arduino-Board verwendet werden.

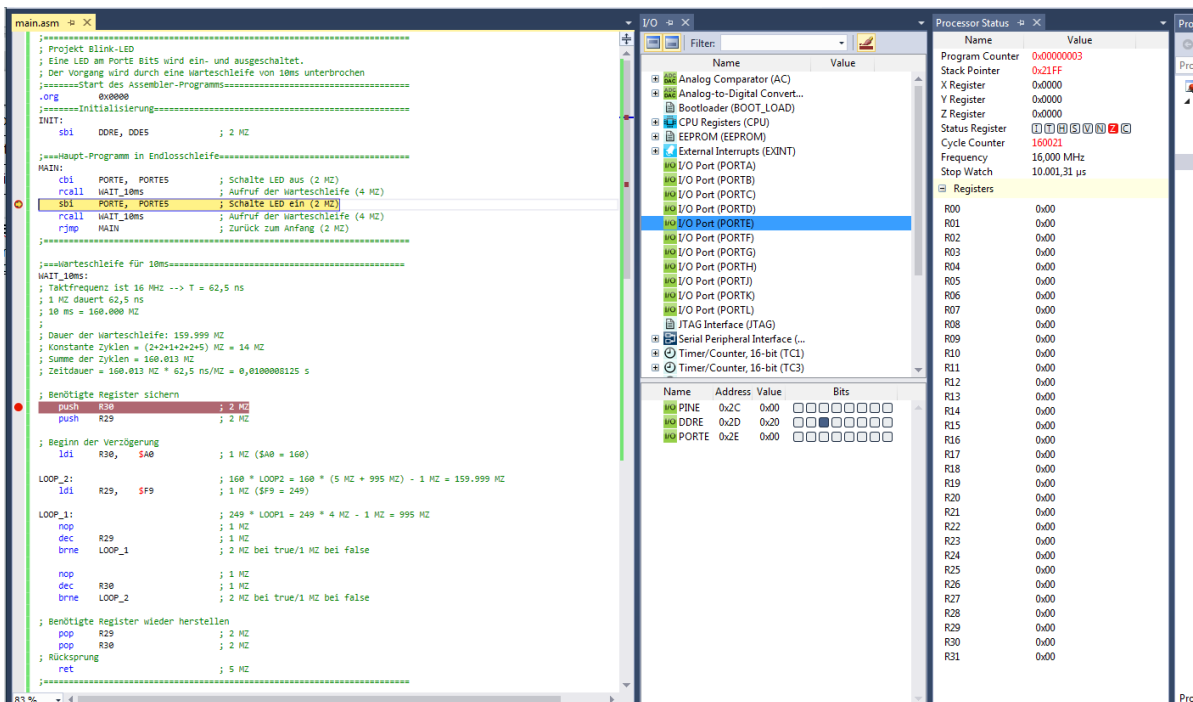
Der Debug-Vorgang wird mit im Menü „Debuggen“ mit dem Menüpunkt „Start Debugging and Break“ oder mit der Tastenkombination **Alt-F5** ausgelöst.

Sinnvollerweise lässt man sich im Menü „Ansicht-Symbolleiste“ die Symbolleiste „Debuggen“ anzeigen.




Durch einen Klick auf das oben dargestellte Symbol ganz links wird eben dieser Menüpunkt „Start Debugging and Break“ ausgeführt. Das Programm startet und bleibt auf dem ersten auszuführenden Befehl (Im Beispiel: `sbi DDRE, DDE5`) stehen.

Es öffnen sich -je nach vorheriger Benutzung- weitere Fenster, die Auskunft über den Zustand der CPU und andere Hardwareelemente geben. Sinnvoll sind hier zumindest die beiden Fenster „Processor Status“ und „I/O“. Diese und andere Fenster lassen sich während des Debuggens anzeigen, indem man im Menü „Debuggen-Fenster“ die gewünschten Fenster auswählt.



1 Die Menünamen lauten in der englischen Sprachversion anders

2 Es gibt auch Hardware-Debugger, die direkt mit dem verbundenen Mikrocontroller kommunizieren und dessen Werte verwenden.

Arbeitsblatt Nr. 04	Q2 Technikwissenschaft: Digitale Steuerungstechnik		B S G G
Datum:	Thema: Debugging von Assemblerprogrammen		
Seite 4 von 4	Name:		

In den Fenstern lassen sich die angezeigten Werte auch bearbeiten. Sinnvollerweise setzt man -vor allem für Zeitmessungen- die Taktfrequenz der CPU auf den korrekten Wert. Für den ATmega 2560 sind dies 16 MHz.

Es lassen sich -wie auch aus Hochsprachen gewohnt- Haltepunkte setzen, die die Programmausführung anhalten, um sich beispielsweise den Wert von Registern oder von I/O-Ports anzusehen.

Das Programm kann im Einzelschritt (**F11**) Befehl für Befehl abgearbeitet werden, oder es können Unterprogrammaufrufe als Prozedurschritt (**F10**) ausgeführt werden. Auch eine Programmfortführung mit „Weiter“ (**F5**) bis zum nächsten Haltepunkt ist möglich.

Übungen

1. Informieren Sie sich zu Beginn über die Funktion der neuen Assembler Befehle **rcall**, **rjmp**, **ret**, **push**, **pop**, **ldi**, **nop**, **brne** und **dec**.
Nutzen Sie hierzu das im Pool befindliche PDF „Atmel-0856-AVR-Instruction-Set-Manual“.
2. Erstellen Sie ein Assembler-Projekt namens „Blink-LED“, fügen Sie das obige Programm in die Datei **main.asm** ein, assemblieren Sie das Programm und übertragen Sie es auf das Arduino-Board.
Die am Digital-Pin 3 mittels Vorwiderstand von 330 Ω angeschlossene LED sollte leuchten. Sie blinkt so schnell, dass es als Leuchten wahrgenommen wird.
3. Setzen Sie einen Haltepunkt auf die erste Anweisung im Unterprogramm und einen zweiten Haltepunkt auf die Anweisung im Hauptprogramm, die die LED einschaltet.
Starten Sie das Debugging und notieren Sie sich den Wert des Cycle Counters beim ersten Halt. Starten Sie das Debugging erneut und notieren Sie sich den Wert des Cycle Counters erneut. Überprüfen Sie, ob die Warteschleife die richtige Anzahl von Maschinenzyklen dauert.
4. Setzen Sie einen weiteren Haltepunkt auf die erste **push**- und auf die erste **pop**-Anweisung im Unterprogramm. Starten Sie das Debuggen und sehen Sie sich nach Erreichen des Haltepunkts die Werte der Register R29 und R30 an und wie sich der Stack Pointer verändert. Führen Sie nun die ersten Befehle im Einzelschritt aus und beobachten Sie die Änderungen im Fenster „Processor Status“. Machen Sie das gleiche bei Erreichen des Haltepunktes bei dem **pop**-Befehl.
5. Nutzen Sie das Fenster „I/O“ und beobachten Sie die Änderungen vom PortE.
Ermitteln Sie, welches Register durch welchen Befehl wie beeinflusst wird.
6. Die Zeitschleife arbeitet prinzipiell so, dass ein Register mit einem Startwert geladen wird, anschließend wird das Register dekrementiert und danach der neue Wert mit 0 verglichen. Solange der Registerwert größer 0 ist, wird dies wiederholt.
Sobald der Registerwert 0 erreicht hat, wird die Programmausführung hinter **brne** fortgesetzt. Da ein Register nur aus 8 Bit besteht, ist der maximale Wert 255 (\$FF), so dass diese Schleife nur 255 mal durchlaufen wird.
Um höhere Werte zu ermöglichen, arbeitet man mit mehr als einem Register und mit geschachtelten Strukturen. Im Beispiel sind dies zwei Register.

Verwenden Sie ein drittes Register, um aus der momentanen Warteschleife mit 10 ms eine Warteschleife mit 250 ms zu machen. Orientieren Sie sich an dem Beispielprogramm!
Dann sollte die LED sichtbar blinken!