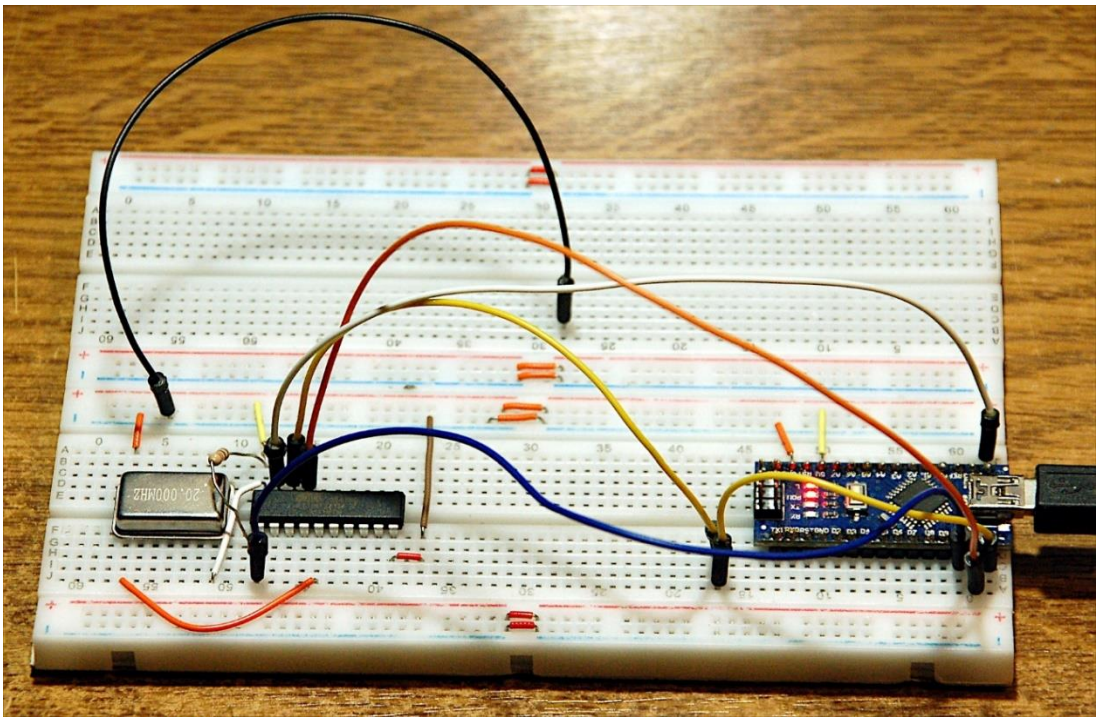

AT-Tiny2313 am Arduino-Programmer

Diesen Beitrag gibt es auch als PDF-Dokument.

In der vorangegangenen Episode hatten wir die DDS-Fähigkeiten (Direkte Digitale Synthese) des ESP32 untersucht. Auf der Jagd nach schnelleren Lösungen für unseren DDS-Generator müssen wir nach einem Micro-Controller suchen, der einen **acht Bit breiten Port** bietet und mit einem **externen Taktsignal** versorgt werden kann. In Frage kommen entweder die Riesen von Microchip, vormals ATMEL AT Mega 16 und AT Mega 32 oder, weil wir nicht mit Kanonen auf Spatzen schießen wollen, der Zwerg, ein AT Tiny 2313. Ich habe mich für den Kleinen entschieden, weil Port B die geforderten acht Ausgangs-Leitungen zur Verfügung stellt und weil er mit einem genauen, externen Quarzoszillator bis 20MHz versorgt werden kann, statt wie beim AT Mega16/32 nur mit 16 MHz. Welche Werkzeuge zum Verfassen und Programmieren eines AT Tiny 2313 erforderlich sind und wie man mit ihnen arbeitet, das erfahren Sie in der heutigen Episode aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Der ESP32 bekommt schnelle Verstärkung

Was steht also heute an?

- Beschaffung und Installation von ATMEL-Studio 7.0
- Beschaffung und Installation der Arduino IDE
- AZ-Arduino Nano V3 als Programmer von ATMEL-Chips einrichten
- Beschaffung und Einrichtung der Programmer-Software
- Assembler-Test-Programm für den AT Tiny 2313 erstellen und brennen und testen

Unser ESP32 kann sich heute erst einmal ausruhen, während wir uns um seinen schnellen DDS-Helfer kümmern. Was wird dafür an Hardware gebraucht?

Die Hardware

1	Nano V3.0 CH340 Chip oder AZ-Nano V3-Board mit USB-C Anschluss mit Atmega328 CH340 fertig verlötete, verbesserte Version
1	MB-102 Breadboard Steckbrett mit 830 Kontakten
1	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F
1	Logic Analyzer
1	Quarzoszillator 20MHz
1	AT Tiny 2313 (A)
1	Widerstand 10,0 kΩ

Im Lauf der Projektentwicklung brauchen wir zwei Breadboards, die an den Längsseiten über zwei Stromschienen zusammengesteckt werden, damit der Controller, der Quarzoszillator, die Widerstände und der Arduino Nano Platz haben.

Der erste Teil des Aufbaus zum Brennen des Flash-Inhalts des AT Tiny 2313 ist sehr einfach, wie die Abbildung 1 zeigt.

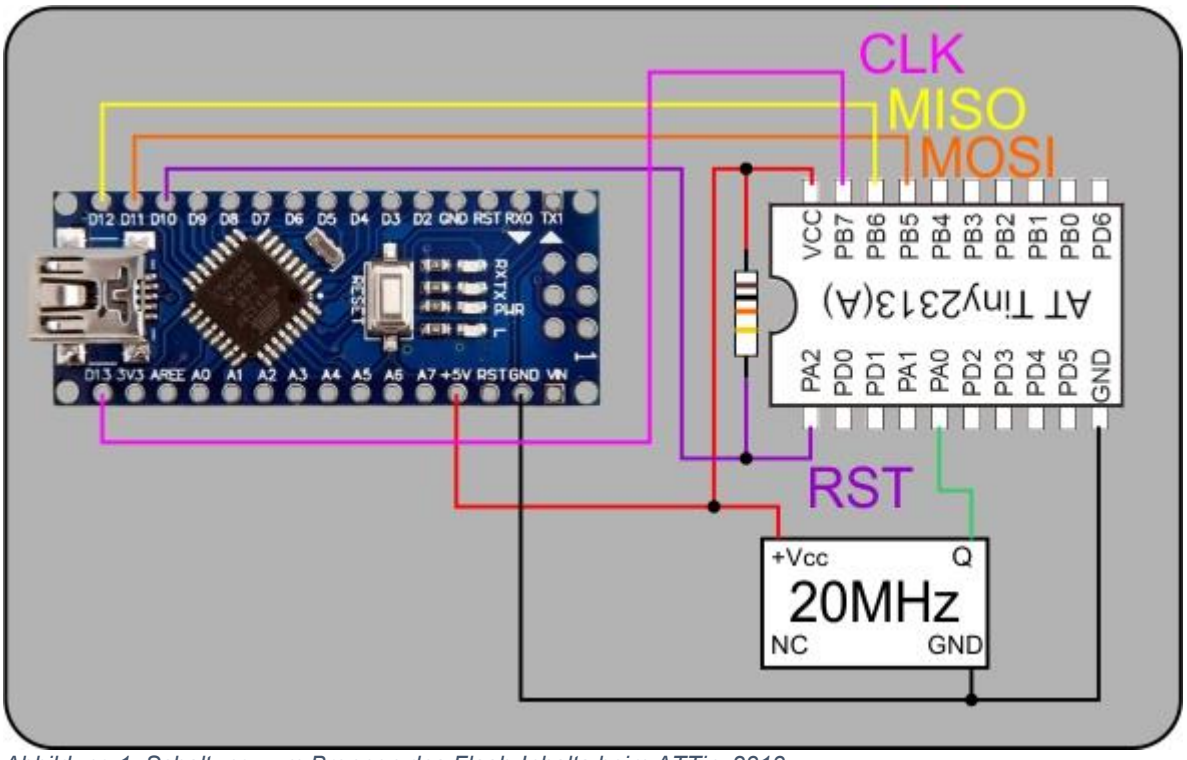

Abbildung 1: Schaltung zum Brennen des Flash-Inhalts beim ATTiny2313

Bevor wir ans Werk gehen, brauchen wir aber noch ein paar Softwarebausteine.

Die Software

Fürs Flashen und die Programmierung des AT Tiny 2313:

Atmel-Studio 7.0
AVRDude (in der Arduino IDE enthalten)
Arduino IDE

Signaldarstellung:

Saleae Logic 2

Die AT Tiny 2313-Programme zum Projekt:

tinytest.asm erstes Testprogramm

Wie kommt das Programm auf den AT Tiny 2313?

Was Thonny für die Entwicklung von MicroPython-Programmen ist, ist ATMEL-Studio 7.0 für das Schreiben von Assembler- (oder C++ -) Programmen für ATMEL-Bausteine. Mit Hilfe von Thonny schicken wir Daten und Firmware direkt über die USB-Leitung zum Controller.

Für ATMEL-Chips brauchen wir zum Brennen von Programmen einen sogenannten Programmer, der die Daten vom USB-Bus in ATMEL-konforme Signale umsetzt. Ideal ist ein AVR MK II (Preis um die 20 €), weil dieses Bauteil direkt aus dem ATMEL-Studio 7.0 heraus angesprochen werden kann und so den Programmierprozess deutlich vereinfacht.

Abbildung 2: AVR-MK II Programmer von ATMEL

Aber es geht auch billiger, wenn man nur mal zwischendurch einen ATMEL-Controller programmieren möchte und dafür etwas mehr Umstände in Kauf nimmt. Es geht nämlich auch mit einem Arduino-Clone. Das kann ein Mikrocontroller Board AZ-ATmega328, ein AZ-Mikrocontroller Board LGT8F328P oder ein AZ-Nano V3- sein. Weil meist irgendwo ein Arduino-Derivat verstaubt in einer Ecke rumliegt, habe ich mich für letztere Variante entschieden. In diesem Beitrag spielt ein AZ Nano V3 die Programmer-Rolle. Damit der Nano lernt, Programmer zu spielen, brauchen wir erst einmal die Arduino IDE.

Arduino-IDE besorgen und installieren

Laden Sie die Installationsdatei herunter und speichern Sie sie in einem beliebigen Verzeichnis.

Abbildung 3: Installationsdatei herunterladen

Im nächsten Fenster haben Sie die Wahl – mit oder ohne Spende herunterladen.

Abbildung 4:Sie haben die Wahl

Den Datenschutzbestimmungen müssen Sie zustimmen, eine e-Mailadresse brauchen Sie nicht anzugeben.

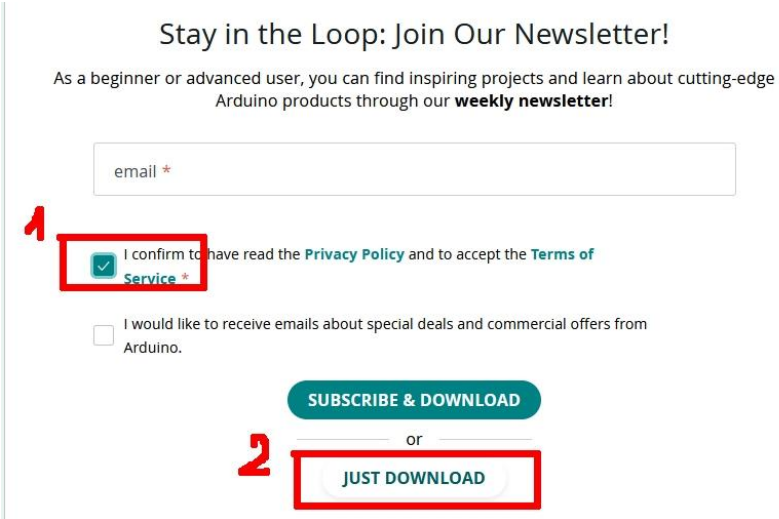

Abbildung 5: Den Datenschutzbestimmungen zustimmen

Navigieren Sie jetzt zum Zielverzeichnis und OK.

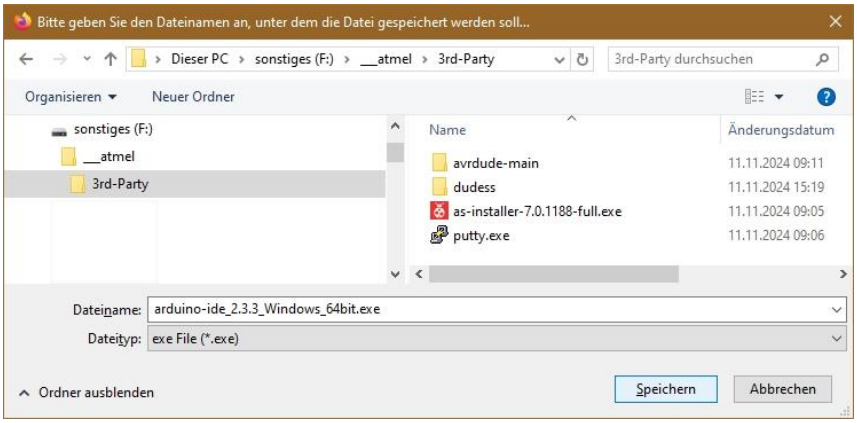

Abbildung 6: Datei abspeichern

Wechseln Sie nun im Explorer in das Zielverzeichnis und starten Sie die Installationsdatei.

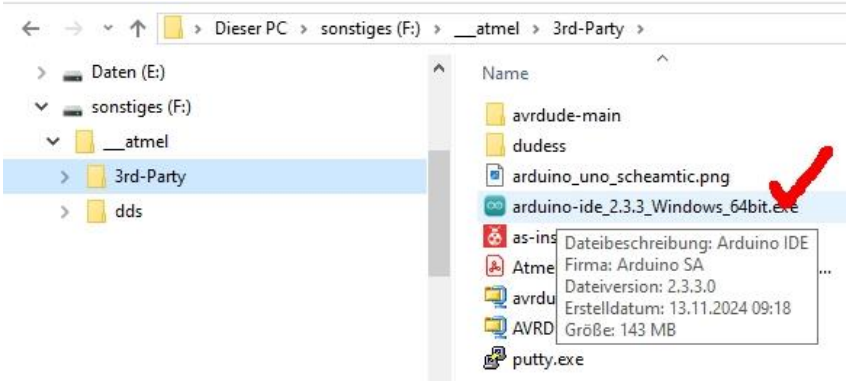

Abbildung 7: Installation ausführen

Die Lizenzvorgaben müssen akzeptiert werden, danach wählen Sie die Benutzergruppe aus.

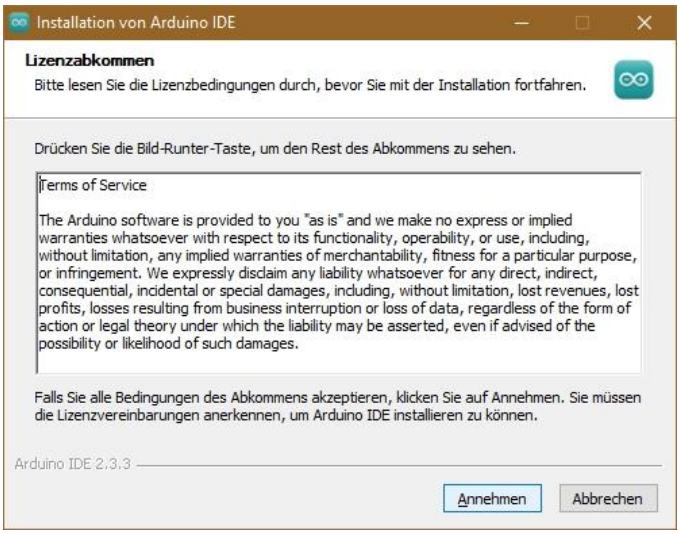

Abbildung 8: Lizenzabkommen annehmen

Um die Arduino IDE für alle Benutzer freizugeben, müssen Sie Admin-Rechte haben.

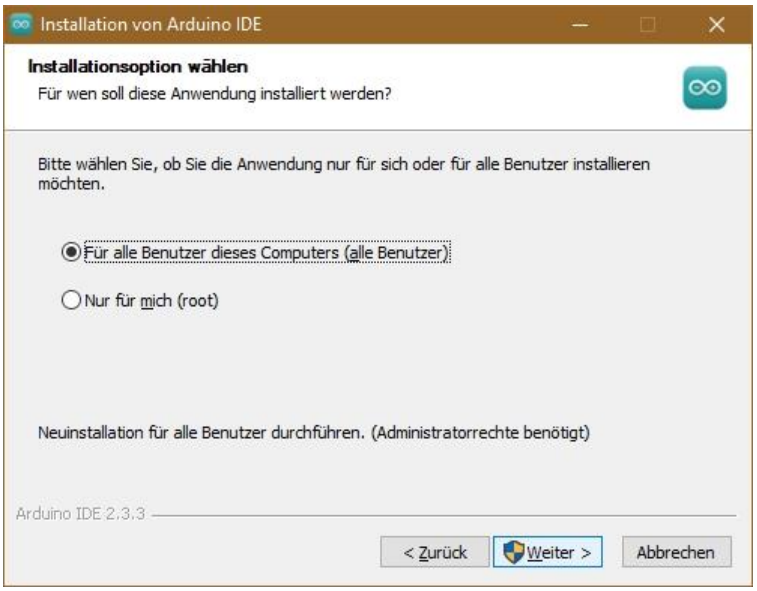

Abbildung 9: Benutzerauswahl

Bestätigen Sie am besten die Vorgabe des Zielverzeichnisses für die Installation.

Abbildung 10: Verzeichnisauswahl bestätigen

Nachdem die Installation beendet ist, lassen Sie die IDE gleich einmal starten.

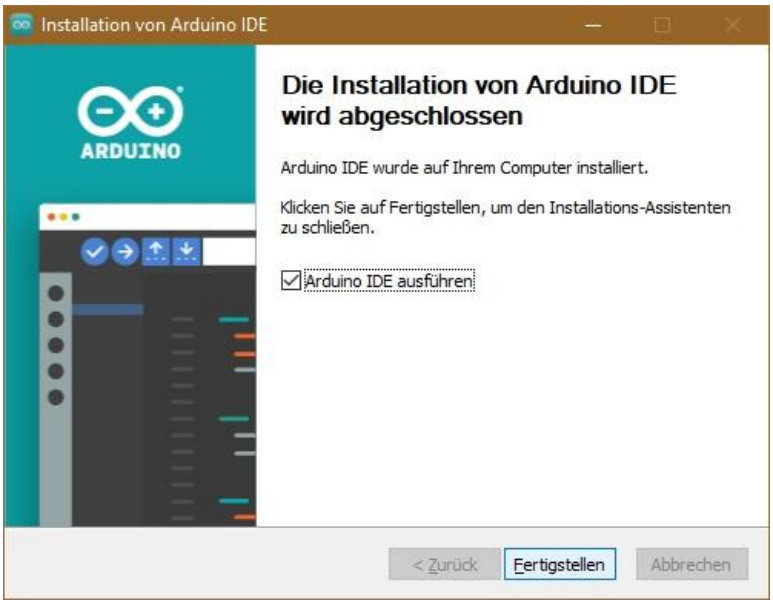

Abbildung 11: Installation abschließen

Nun laufen einige weitere Schritte automatisch ab. Die Treiber für den USB-Zugriff lassen wir installieren.

Abbildung 12: Treiber für USB-Zugriff installieren

Jetzt ist der richtige Zeitpunkt, das Arduino-Board mit dem PC zu verbinden, damit im übernächsten Schritt die virtuelle COM-Schnittstelle erkannt und zur Auswahl angeboten wird.

Die IDE bietet weitere Komponenten zur Einrichtung an, wir lassen das Update zu und lassen alles installieren, dann brauchen wir uns später nicht mehr darum zu kümmern.

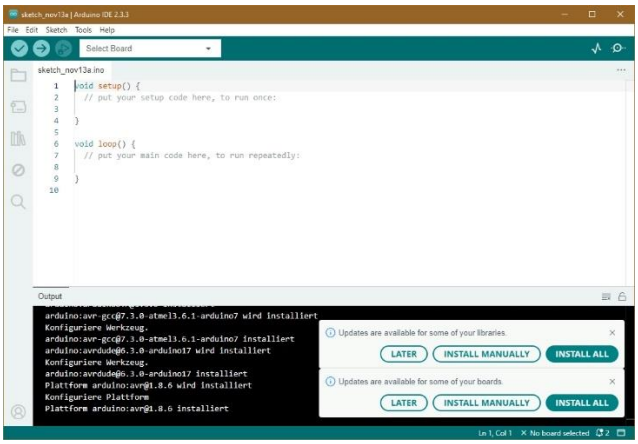

Abbildung 13: Startfenster nach der Installation

Die IDE zeigt jetzt die verfügbaren COM-Ports an, wir wählen den richtigen aus und picken außerdem unser Board aus der Liste.

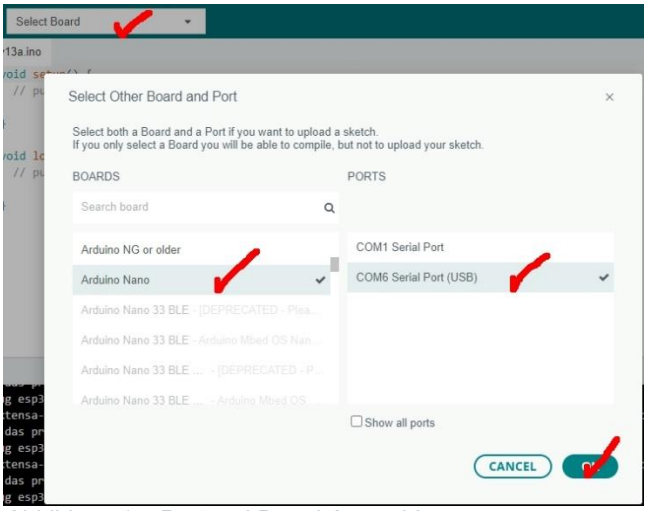

Abbildung 14: Port und Board-Auswahl

Port und Controller-Board können auch nachträglich über das **Tools**-Menü eingerichtet werden.

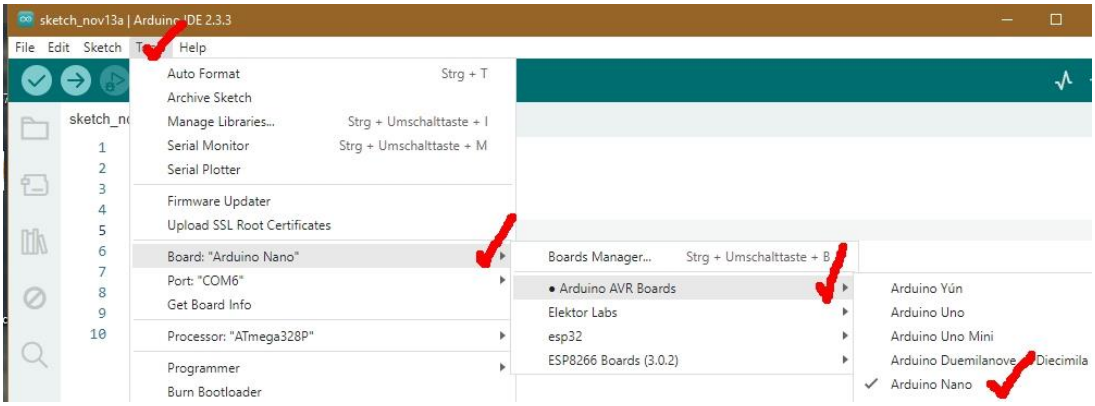

Abbildung 15: Boardauswahl über das Menü

Der Nano V3 wird ein ISP

Den AZ-Nano V3 in einen ISP (In System Programmer) zu verwandeln ist kein Kunststück. Der erforderliche Sketch wird in der IDE (Integrated Development Environment = Integrierte Entwicklungs-Umgebung) mitgeliefert. Wir holen ihn über das Menü **Files – Examples – 11 ArduinoISP – ArduinoISP** in den Editor.

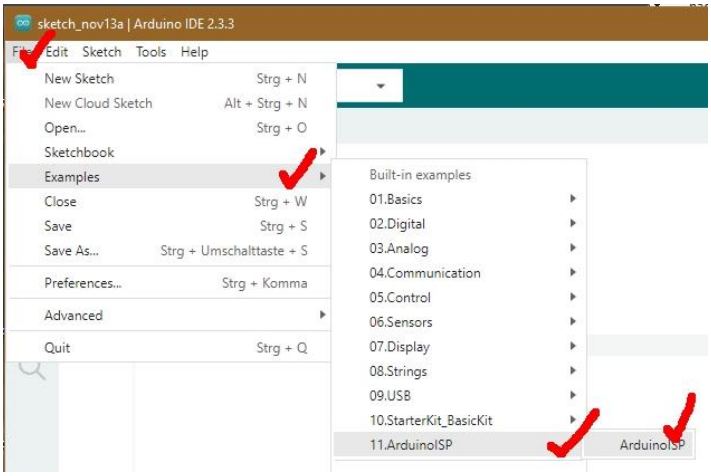

Abbildung 16: Sketch ArduinoISP laden

Abschließend lassen wir den Sketch compilieren und schicken das Resultat zum Nano V3.

Abbildung 17: Sketch compilieren und hochladen

Leider können wir die Arduino IDE nicht direkt benutzen, um unsere eigenen Assemblerprogramme in den AT Tiny 2313 zu brennen. Aber die IDE liefert einen Hinweis, für eine andere Lösung. Wenn man im Ausgabefenster nach oben scrollt, findet man die Zeile mit dem Befehl, durch den die übersetzte hex-Datei des Sketches an den AT Mega 328 auf dem Nano-Board geschickt wurde. Da diese Übertragung über die USB-Verbindung läuft, können wir das Tool AVRDude, das hier aufgerufen wird, auch für unsere Zwecke verwenden. Dazu später mehr.

Abbildung 18: Aufruf von AVRdude

Wir kommen auf diese Zeile zurück, wenn wir das erste Assembler -Testprogramm zum AT Tiny 2313 schicken werden. Doch zuerst müssen wir dieses Programm schreiben und assemblieren und dabei hilft uns ein weiteres kostenloses Tool.

ATMEL-Studio 7.0 – Entwicklungsumgebung für ATMEL-Chips

ATMEL-Studio 7.0 ist ein Verbund von Editor, Assembler, Debugger, Simulator und Disassembler. Schauen wir uns an, wie man mit dem Programm ein Assemblerprogramm schreibt, testet, übersetzt und zum Controller hochlädt.

Laden Sie erst einmal die Installationsdatei herunter und speichern Sie diese in einem Verzeichnis Ihrer Wahl. Navigieren Sie zum Downloadverzeichnis und starten Sie die Installation.

Den Lizenzvereinbarungen müssen wir zustimmen, damit die Installation beginnt.

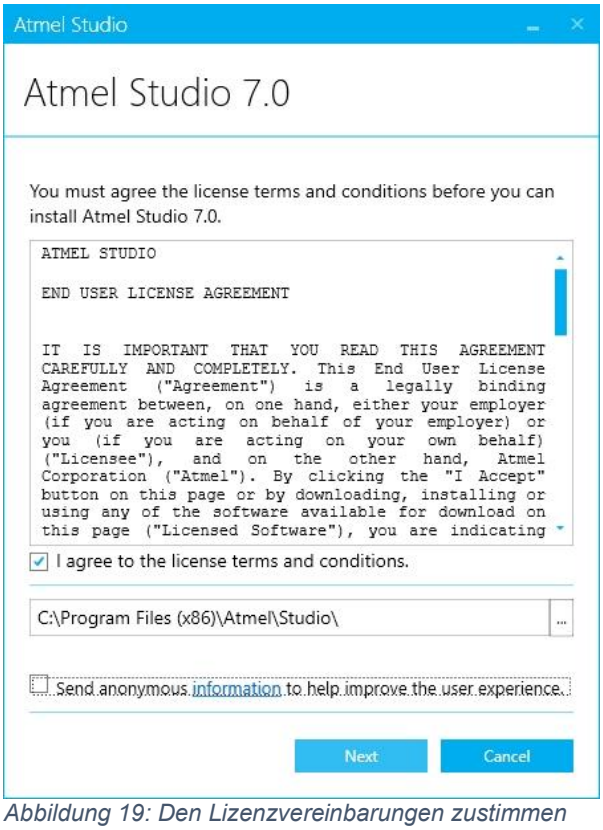

Abbildung 19: Den Lizenzvereinbarungen zustimmen

An Controllern brauchen wir lediglich die 8-Bit-Familie.

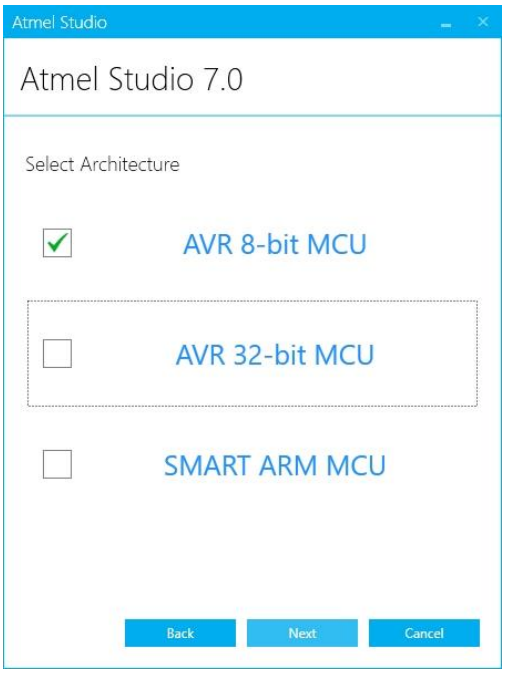

Abbildung 20: Controllerfamilie auswählen

Auch die Beispiele können wir gegebenenfalls brauchen.

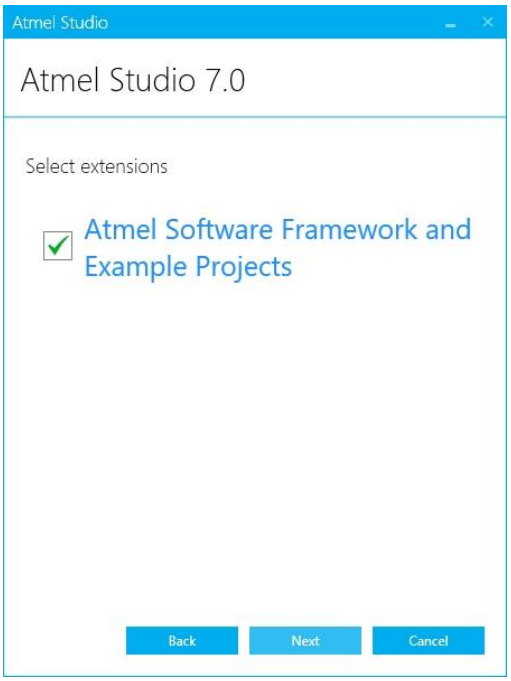

Abbildung 21: Erweiterungen mit installieren

Den Systemcheck quittieren wir mit **Next**, falls alle Punkte einen grünen Haken haben.

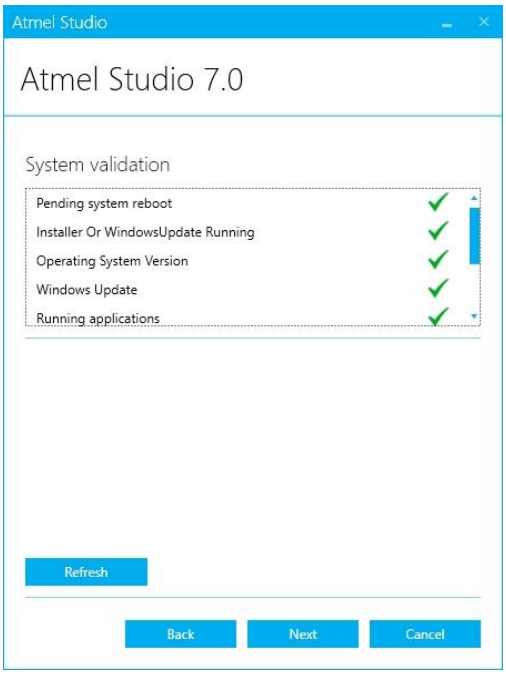

Abbildung 22: Systemcheck Ergebnis

Nach der gelungenen Installation starten wir auch gleich zur ersten Assemblersitzung durch.

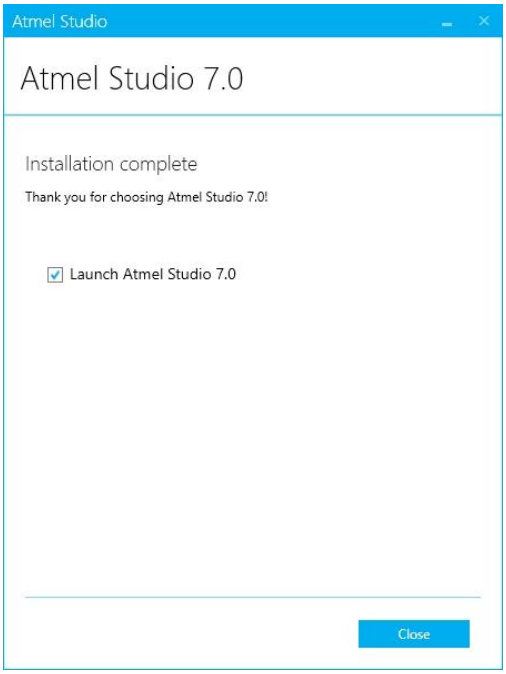

Abbildung 23: Installation beendet

Auf der Startseite stoßen wir ein neues Projekt an – **New Project**.

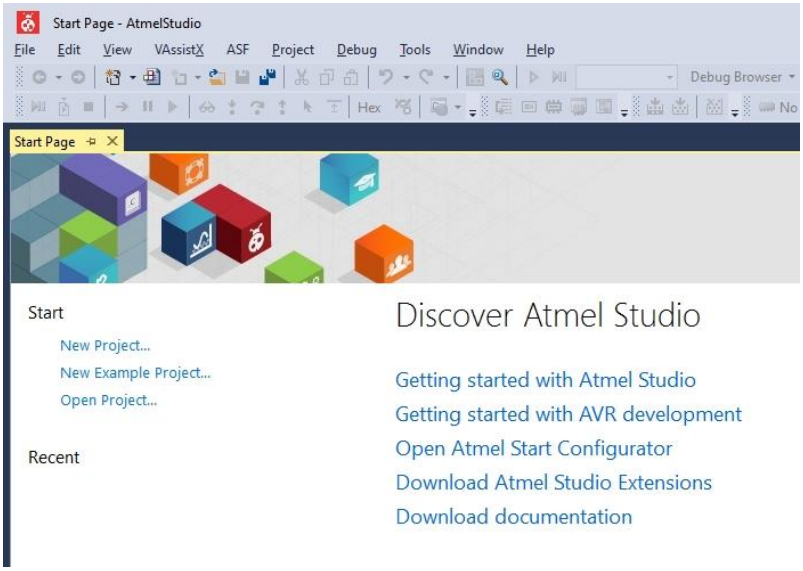

Abbildung 24: Oberfläche nach dem ersten Start

Für das Assembler-Projekt vergeben wir einen Namen - **tinytest**. Zur Speicherung der Projektdateien wird ein Verzeichnis gleichen Namens angelegt.

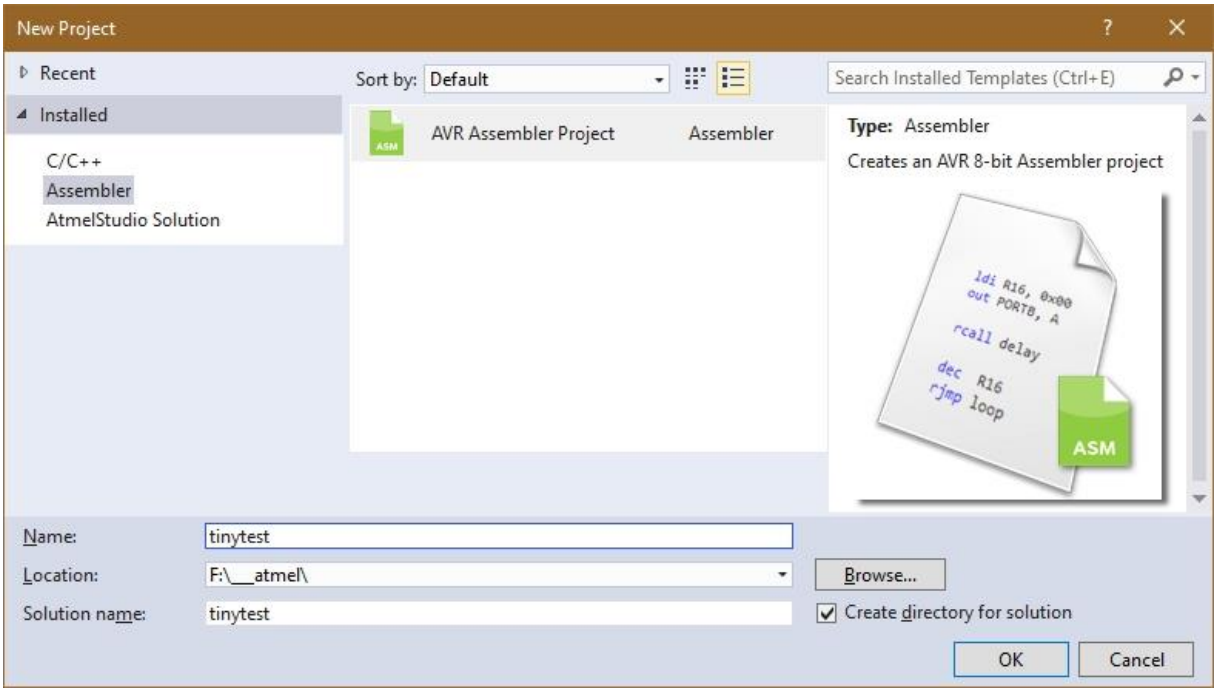

Abbildung 25: Neues Projekt anlegen

Jetzt müssen wir nur noch den Controllertyp auswählen - AT Tiny 2313(A) – dann öffnet sich das Editorfenster. Dort hin Kopieren wir den Programmtext aus der Datei **tinytest.asm**, nachdem wir den vorgelegten Text gelöscht haben. In unserem Projekt wird daraus somit die Datei **main.asm**.

Das erste Assemblerprogramm

Was ist Assembler?

Assembler ist eine maschinennahe Sprache. Durch die Verwendung von sogenannten Mnemoniks wird dem Programmierer erspart, dass er sich alle Hex-Codes der Maschinenbefehle und deren Zusammenwirken mit Daten merken muss. Die meisten Befehle setzen sich aus einem Befehlsteil und dem Datenteil zusammen. Für beide Bereiche sind entsprechende Bitpositionen reserviert. ATMEL-Studio 7.0 übernimmt den Job, den Befehlscode mit den nachfolgenden Parametern zu einem 16-Bit-Befehlswort zusammenzufügen und im Codeteil des Programms abzulegen. Alle Codes zusammen bilden das Maschinenprogramm, das dann im Controller ohne weitere Bearbeitung ausgeführt werden kann. Zum Vergleich, in MicroPython liegt das Programm als Script vor, das dann bei der Ausführung erst durch den Interpreter in Maschinencode übersetzt werden muss. Das macht MicroPython-Programme langsam.

Ähnlich wie bei einem C-Programm, gibt es aber keine Möglichkeit, Befehle interaktiv zu erkunden, wie das in MicroPython in REPL der Fall ist. Aber immerhin bietet ATMEL-Studio 7.0 mit seinem Debugger die Chance, das Programm schrittweise zu durchlaufen und dabei den Inhalt der Register und Speicherbereiche sowie IO-Operationen einzusehen und auf korrekte Ausführung der Operationen zu überprüfen.

Eine Liste von Befehls-Mnemoniks findet man im Datenblatt des AT Tiny 2313 auf den Seiten 257 und 258. Keine Angst, wir werden in unseren Programmen nur einen kleinen Teil davon verwenden.

Der Timingtest

Hinweis: AVR-Assemblercode ist nicht case sensitive, es wird also nicht zwischen Groß- und Kleinschreibung unterschieden.

Wie bei unserem ESP32 wollen wir mal nachschauen, wie schnell der Kleine ist, wenn es um die Abfrage und Ausgabe von DDS-Daten geht. Wir orientieren uns dabei an dem Programm dds_generator_irq.py. Wieder lassen wir zwei Pins ein- und ausschalten und schauen mit dem Logic Analyzer nach, wie das Timing läuft.

Den Text des Programms, das wir jetzt besprechen, kopieren wir aus der Datei tinytest.asm in das Editorfenster von ATMEL-Studio 7.0, falls das noch nicht schon geschehen ist. Dadurch wird es zur Datei **main.asm** in unserem Projekt. Den Namen **main.asm** legt ATMEL-Studio 7.0 automatisch vor.

Konstanten werden mit **.equ** festgelegt. Wir weisen die Taktfrequenz unseres Quarzoszillators von 20MHz dem Bezeichner **takt** zu.

```
    .equ takt       = 20000000      ; 20 MHz ist der Systemtakt
```

Beim AT Tiny 2313 gibt es spezielle Speicherzellen, auf denen Operationen ausgeführt werden können, die Register. Beim AT Tiny 2313 gibt es davon 32 Stück. Sie erfüllen also die Aufgabe einer CPU. Auch Register können Alias-Namen

erhalten. Hier werden die Register r16 bis r18 umbenannt. Zur Namensgebung von Registern wird **.def** verwendet.

```
    .def temp1  = r16        ; Umbenennen einiger Register
    .def temp2  = r17
    .def temp3  = r18
```

Register können nur 8-Bit-Werte aufnehmen. Zum Verarbeiten von 32-Bit-Werten brauchen wir also jeweils vier Stück davon.

```
    .def ddsp0     = r20              ; DDS-Phase
    .def ddsp1     = r21              ;
    .def ddsp2     = r22              ;
    .def ddsp3     = r30                   ;

    .def ddsd0     = r24              ; DDS-Phasen-Increment
    .def ddsd1     = r25              ;
    .def ddsd2     = r26              ;
    .def ddsd3     = r27
```

Anders als beim ESP32 haben wir beim AT Tiny 2313 nicht einzelne GPIOs, sondern Ports mit sechs (Port D) und acht Bits (Port B) Breite zur Verfügung. Das erlaubt uns überhaupt erst die simultane Ausgabe von acht Bits, was für den selbstkonstruierten DAC essentiell ist. Der AT Tiny 2313 besitzt keinen eingebauten DAC.

Für die Messung des Timings setzen wir die Pins **PortD.4** und **PortD.6** ein. Der Port ist ebenfalls ein Register, das aber zu keinen arithmetischen oder logischen Operationen fähig ist. Zu jedem Port gehört ein Daten-Richtungs-Register, das festlegt, welches Pin als Eingang (DDR-Bit = 0) oder Ausgang (DDR-Bit=1) ist. IO-Register haben einen eigenen Adressbereich, der von 0x00 bis 0x3F reicht. Neben den Ein- Ausgabe-Registern befinden sich in dem Adressbereich weitere Register, die die zum Beispiel die Funktion der Timer, Schnittstellen etc. steuern.

```
    .equ loopPin = 4
    .equ timerPin = 6
    .equ mPort = portD ; portD = 0x12
    .equ mDDR = ddrD   ; ddrD  = 0x11
```

Die Samplefrequenz für unsere DDS-Anwendung ergibt sich aus den Messungen, die wir später durchführen werden. **isrValue** ist ein Vergleichswert für den Hardwaretimer durch den die IRQ-Folgefrequenz festgelegt wird. Erreicht der Timer diesen Wert, dann wird er zurückgesetzt und beginnt erneut Systemtakte zu zählen. Ein Systemtakt entspricht bei 20MHz einer Zeitspanne von 50ns. Bei 40 Systemtakten kommen wir somit auf 2µs IRQ-Folgezeit.

```
; Maximale Samplefrequenz: 20000000 Hz / 24 = 833333 Hz
; gewählte Samplefrequenz: 500 kHz
; TOP-Wert für Timer1:
;
    .equ isrValue = 20000000 / 500000 ; = 40
```

Die Berechnung des DDSd-Werts erfolgt in der gleichen Weise wie im vorangegangenen Beitrag. Wegen der Beschränkung von ATMEL-Studio 7.0 auf 32-Bitzahlen muss man bei der Berechnung einen Klimmzug machen. Wir dividieren zuerst 2^{31} durch die Samplefrequenz in kHz und multiplizieren dann noch mit 2. Im späteren Produktionsbetrieb erledigt diese Berechnung der ESP32 und sendet den Kleinen nur noch den fertigen DDSd-Wert.

```
; DDS-Versatz für 1kHz Ausgangssignal
; ddsd / 2^32 = 1000 / 500000 oder
; ddsd = 1kHz/500kHz*(2^32)=8589934
     .equ ddsd   = EXP2(31) / 500 * 2;
```

Die Kurvendaten hatten wir beim ESP32 in Form von Listen abgelegt. In Assembler gibt es keine Listen. Stattdessen legen wir die DDSs-Daten als Bytes (.db = define Byte) im Programmspeicher (.cseg = Code Segment) ab der Adresse 0x0200 ab. mit den 128 RAM-Bytes des AT Tiny 2313 hätten wir da eh keine Chance. Weil der Programmspeicher in Words (zwei 8-Bit-Speicherstellen = ein Word) organisiert ist, landen die Sinus-Werte also an Byteposition 0x0400.

```
; ************************************************************
; * Kurvendaten
; ************************************************************
.cseg
.org 0x0200
Sinus:
.db 128, 131, 134, 137, 140, 144, 147, 150
.db 153, 156, 159, 162, 165, 168, 171, 174
.db 177, 180, 182, 185, 188, 191, 194, 196
.db 199, 201, 204, 206, 209, 211, 214, 216
.db 218, 220, 222, 224, 226, 228, 230, 232
.db 234, 236, 237, 239, 240, 242, 243, 244
.db 246, 247, 248, 249, 250, 251, 251, 252
.db 253, 253, 254, 254, 254, 255, 255, 255
.db 255, 255, 255, 255, 254, 254, 253, 253
.db 252, 252, 251, 250, 249, 248, 247, 246
.db 245, 244, 242, 241, 240, 238, 236, 235
.db 233, 231, 229, 227, 225, 223, 221, 219
.db 217, 215, 212, 210, 208, 205, 203, 200
.db 197, 195, 192, 189, 187, 184, 181, 178
.db 175, 172, 169, 167, 164, 160, 157, 154
.db 151, 148, 145, 142, 139, 136, 133, 130
.db 126, 123, 120, 117, 114, 111, 108, 105
.db 102, 99, 96, 92, 89, 87, 84, 81
.db 78, 75, 72, 69, 67, 64, 61, 59
.db 56, 53, 51, 48, 46, 44, 41, 39
.db 37, 35, 33, 31, 29, 27, 25, 23
.db 21, 20, 18, 16, 15, 14, 12, 11
.db 10, 9, 8, 7, 6, 5, 4, 4
.db 3, 3, 2, 2, 1, 1, 1, 1
.db 1, 1, 1, 2, 2, 2, 3, 3
.db 4, 5, 5, 6, 7, 8, 9, 10
.db 12, 13, 14, 16, 17, 19, 20, 22
```

```
.db 24, 26, 28, 30, 32, 34, 36, 38
.db 40, 42, 45, 47, 50, 52, 55, 57
.db 60, 62, 65, 68, 71, 74, 76, 79
.db 82, 85, 88, 91, 94, 97, 100, 103
.db 106, 109, 112, 116, 119, 122, 125, 128
```

Im Programmspeicherbereich sind die Words an den Adressen 0x0000 bis 0x0012 für die Interruptvektoren reserviert. An die hier hinterlegten Programmadressen springt die Programmausführung, wenn eine der 13 Interruptquellen aktiv wird. Wir setzen den Programmzeiger auf die Adresse 0x0000: .org 0x0000. Dort steht ein Sprung (**rjmp**) zum Start-Label des Programms. An der Adresse 0x0005 steht ein Sprung zur ISR (Interrupt Service Routine), die sich um den Überlauf des Timers 1 kümmert.

```
;*************************************************************
; * IRQ-Vektoren
;*************************************************************
     .org 0x0000                    ; Programmstart
     rjmp Start                     ; Resetvektor

     .org 0x0005                    ; Timer1 Overflow Pointer
     rjmp t1ovl
```

Hinter allen IRQ-Vektoren kann das Programm beginnen.

```
     .org 0x0013                    ; hinter allen IRQ-Vektoren

;*************************************************************
; * Hauptprogramm Vorbereitungen
;*************************************************************
```

Beim Programmstart muss zunächst der Stackpointer initialisiert werden. Der Stack ist ein Teil des RAM-Speichers und wird zum Beispiel benötigt, um beim Aufruf einer Unterprogramm-Routine die Rücksprungadresse zu speichern. Der Stack liegt am Ende des RAM-Bereichs und wird von oben nach unten gefüllt. Wir laden das LSB der Adresse des RAM-Endes direkt (immediate) in das Register r16. Und schaufeln den Wert in das LSB des Stackzeigers (Stack Pointer Low), Das ist das Register 0x3D im IO-Bereich.

```
Start:
     ldi temp1, low(ramend)   ; Stackpointer setzen
     out spl, temp1
```

Wir setzen den Phasenzeiger **ddsp** auf Anfang und belegen das Inkrement **DDSd** vor, indem wir die Register direkt mit den Werten laden. Den zuvor berechneten DDSd-Wert zerlegen wir in die vier Bytes. Das geht hier übersichtlicher als in MicroPython.

```
    ldi ddsp0,0             ; Phasen-Register auf 0
    ldi ddsp1,0
    ldi ddsp2,0
    ldi ddsp3,0

    ldi ddsd0,low(ddsd)         ; Phaseninkrement definieren
    ldi ddsd1,high(ddsd)
    ldi ddsd2,byte3(ddsd)
    ldi ddsd3,byte4(ddsd)
```

Im DDR-Register setzen wir die Bits 4 und 6. Das macht die Leitungen des Ports zu Ausgängen (sbi = set Bit in IO), deren Zustand wir auf 0 setzen (cbi = clear Bit in IO) .

```
    sbi mDDR, loopPin       ; Leitung loopPin auf Ausgang
    cbi mPORT, loopPin          ; Pin low
    sbi mDDR, timerPin      ; Leitung loopPin auf Ausgang
    cbi mPORT, timerPin         ; Pin low
```

Bei Port B setzen wir alle Leitungen auf Ausgang.

```
    ldi r16, 0b11111111         ; PortB auf Ausgang
    out ddrb, r16
```

Der Timer soll nach 40 System-Takten überlaufen, die sind erreicht, wenn er bis 39 gezählt hat. Der Timer arbeitet mit 16-Bit-Werten, deshalb müssen wir das MSB und LSB getrennt ermitteln und den beiden Bytes des ICR-Registers zuweisen. Das Handbuch des AT Tiny 2313 schreibt vor, dass das MSB vor dem LSB ausgegeben werden muss.

```
    ldi r17, high(isrValue-1)   ; Vergleichswert für Timer 1
    ldi r16, low(isrValue-1)    ; Modus 14
    out icr1h, r17
    out icr1l, r16
```

Bit 7 im Register **timsk** (0x39) schaltet den Timer 1-Overflow-IRQ frei.

```
    ldi r16, 0b10000000         ; Timer 1 overflow enable
    out timsk,r16
```

Wir verwenden den Modus 14 des Timers 1. Die genaueren Hintergründe zu erläutern würde hier zu weit führen. Für Interessierte empfehle ich die Seiten 97 ff und die Tabelle 12-5 auf Seite 113 des Handbuchs.

```
    ldi r16, 0b00011001         ; Modus 14 setzen
    out tccr1b, r16
    ldi r16, 0b00000010
    out tccr1a, r16
```

Das 16-Bit-breite Z-Register setzt sich aus den Registern r30 (**zl**) und r31 (**zh**) zusammen und dient in seiner Spezialaufgabe als Zeiger in den Programmspeicher

dazu, dort abgelegte Daten mit dem Befehl **lpm** (load program memory) auszulesen. Das Highbyte der Wort-Adresse 0x0200 ist 0x02. Damit füttern wir **zh**. **zl** wird später durch die ISR gesetzt. Danach lassen wir allgemein Interrupts zu. Der Befehl sei setzt das I-Flag im Flagregister **sreg** (=0x3F).

```
     ldi zh, 0x02

     sei

;
************************************************************
; * Hauptprogramm
************************************************************
```

Die Hauptschleife hat nur die Aufgabe das **loopPin** PortD.4 ein- und auszuschalten, wie beim ESP32. **rjmp** führt einen relativ jump zum Label **loop** aus. Labels sind Einsprungadressen, stehen am Zeilenanfang und werden mit einem ":" abgeschlossen.

```
loop:
     sbi mPort, loopPin            ; an
     cbi mPort, loopPin            ; aus
     rjmp loop

************************************************************
; * IRQ-Service Routinen
************************************************************
```

Auch die ISR des Timer-IRQs hat die gleiche Aufgabe wie beim ESP32 in der letzten Episode. Das **timerPin**, PortD.6, wird gesetzt, **lpm** holt über den Zeiger des Z-Registers einen Wert aus der Sinus-Tabelle. Ohne weitere Operanden geht der Wert in das Register r0, von wo er am Port B ausgegeben wird.

```
t1ovl:
     sbi mPort, timerPin           ; IRQ-Signal an
     lpm                           ; Signalpegel nach r0 holen
     out portb, r0                 ; an Port B ausgeben
     add ddsp0, ddsd0              ; Phasenwinkel erhöhen
     adc ddsp1, ddsd1
     adc ddsp2, ddsd2
     adc ddsp3, ddsd3
     cbi mPort, timerPin           ; IRQ-Signal aus
     reti
```

DDSp muss jetzt um DDSd erhöht werden. Die beiden LSBs werden einfach addiert (add). Dabei kann sich ein Übertragsbit ergeben, das bei den drei nächsten Additionen jeweils berücksichtigt werden muss (adc = add with carry). Bei der Addition steht im ersten Operanden zunächst der erste Summand und nach der Operation der Summenwert. Danach setzen wir das Bit 6 in Port D auf 0 und kehren vom IRQ zurück. ddsp3 ist der Alias für Register r30 = zl.

Mit der Taste **F7** startet ATMEL-Studio 7.0 den Assemblierungsvorgang und zeigt das Ergebnis im Output-Fenster an. Wir suchen jetzt die Zeile mit dem Programmstart und klicken in grauen Rand neben der ersten Zeile danach. Der rote Punkt zeigt an, dass wir an dieser Stelle einen Breakpoint gesetzt haben.

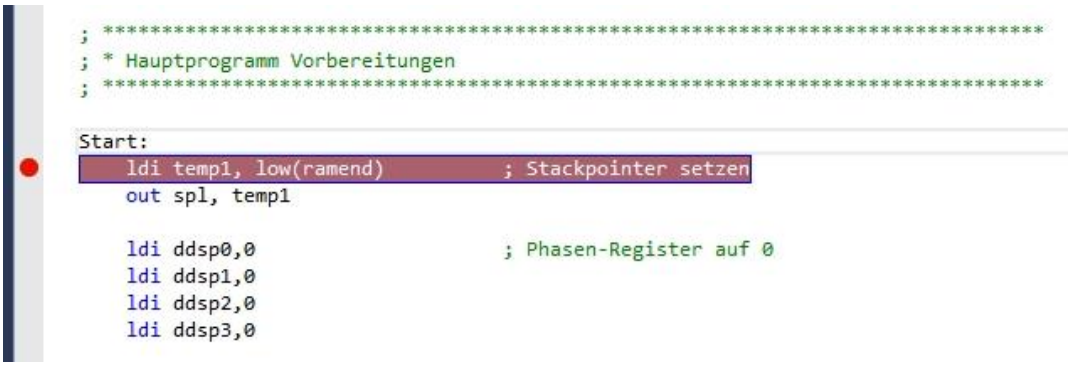

Abbildung 26: Setzen eines Unterbrechungspunktes

Jetzt starten wir mit **F5** eine Debug-Session. Der Simulator startet das Programm und bleibt beim Breakpoint stehen. Mit der Schaltfläche **Prozessor Status** schalten wir die Anzeige der Register ein. Mit der Taste **F11** können wir durch das Programm wandern und im Fenster Prozessor Status die Belegung der Register verfolgen. Mit **Strg+Shift+F5** schalten wir den Debugmodus aus.

Brennen des AT Tiny 2313

Jetzt fehlt nur noch die Übertragung des Programms auf den AT Tiny 2313. Mit dem MK II könnten wir jetzt einfach aus dem ATMEL-Studio 7.0 heraus das Programm übertragen. Um den Nano V3 als Programmer nutzen zu können, brauchen wir ein weiteres Programm, das den AVRISP im Nano V3 mit den korrekten Signalen ansteuern kann und das ist **avrdude.exe**.

Der Aufruf zum Brennen des Nano V3 aus der Arduino IDE heraus, ist, mit Verlaub, horrende lang. Es geht aber auch kürzer und vor allem direkt aus dem Verzeichnis heraus, in dem ATMEL-Studio 7.0 die übersetzte Assembler-Datei ablegt. Dorthin wechseln wir, nachdem wir ein Fenster der Eingabeaufforderung geöffnet haben.

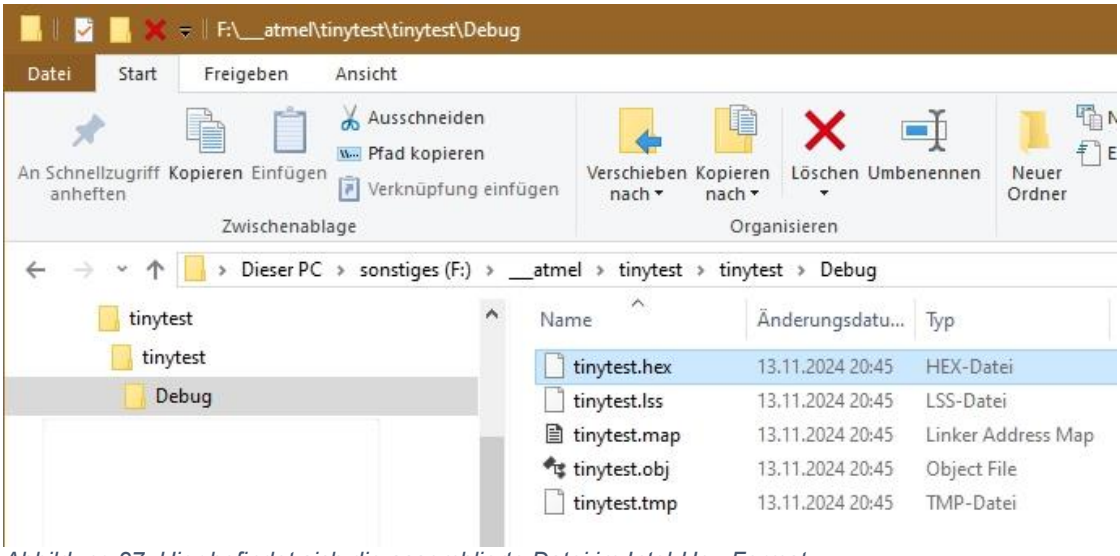

Abbildung 27: Hier befindet sich die assemblierte Datei im Intel-Hex-Format

Unsere Datei heißt **tinytest.hex** und liegt im Intel-Hex-Format vor. Das Intel-Hex-Format dient seit 1975 dazu, Daten in Files so abzulegen, dass sie gesichert auf Speicherbausteine übertragen werden können. Ein Datensatz (eine Zeile, siehe unten) besteht aus dem Startbyte (:), dem Bytezähler (02), der Zieladresse im Chip 0x0000, dem Datentyp (02), den Datenbytes (0x00 0x00) und einer Prüfsumme (FC). Die Zeile endet mit Wagenrücklauf (0x0D) und Zeilenvorschub (0x0A). Jedes Byte wird durch zwei Hexadezimalziffern im ASCII-Format codiert. Genaueres kann man in diesem Wikipedia-Artikel nachlesen. Der Dateiinhalt unserer **tinytest.hex** lässt sich mit dem kostenlosen Hex-Editor HxD betrachten. In Abbildung 27 ist die erste Textzeile markiert:

:02 0000 02 00 00 46 43 0D 0A

Danach folgen die Sinus-Daten ab Byteadresse 0x0400.

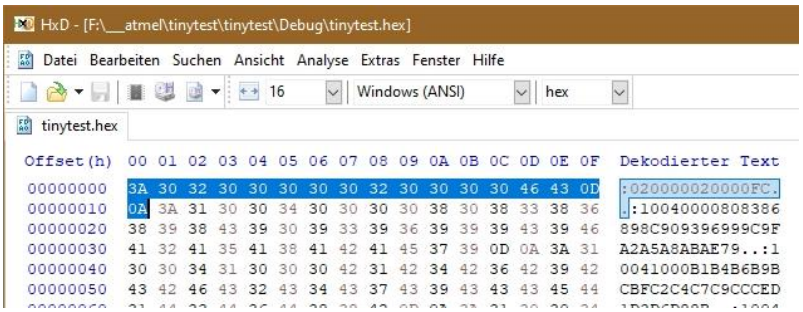

Abbildung 28: Die ersten Bytes aus der Datei tinytest.hex

Ich gehe also jetzt in das Verzeichnis **F:___atmel\tinytest\tinytest\Debug**, wo sich meine Datei befindet.

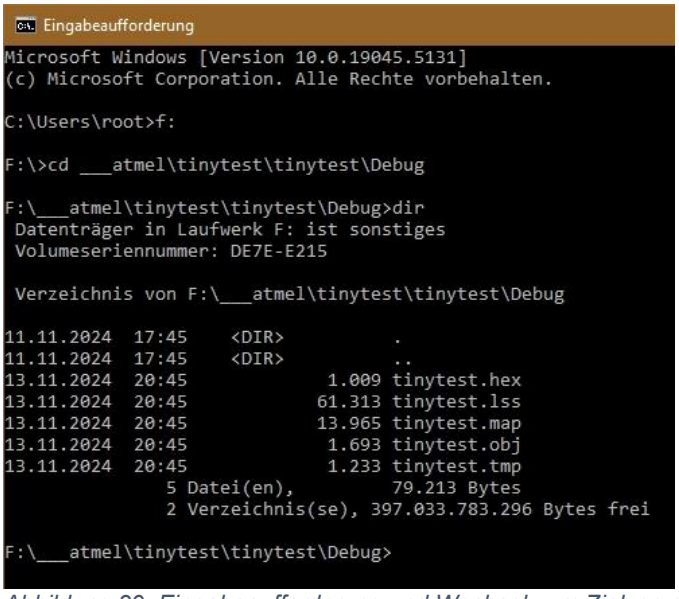

Abbildung 29: Eingabeaufforderung und Wechsel zum Zielverzeichnis

Der AT Tiny 2313 sollte nun wie in Abbildung 1 mit dem Nano V3 und dem Quarzoszillator verdrahtet sein. Zusätzlich schließen wir den Logic Analyzer an, um die Pegel an PortD.4 (Mainloop pulse an Kanal 1) und PortD.6 (IRQ pulse an Kanal2) aufzuzeichnen.

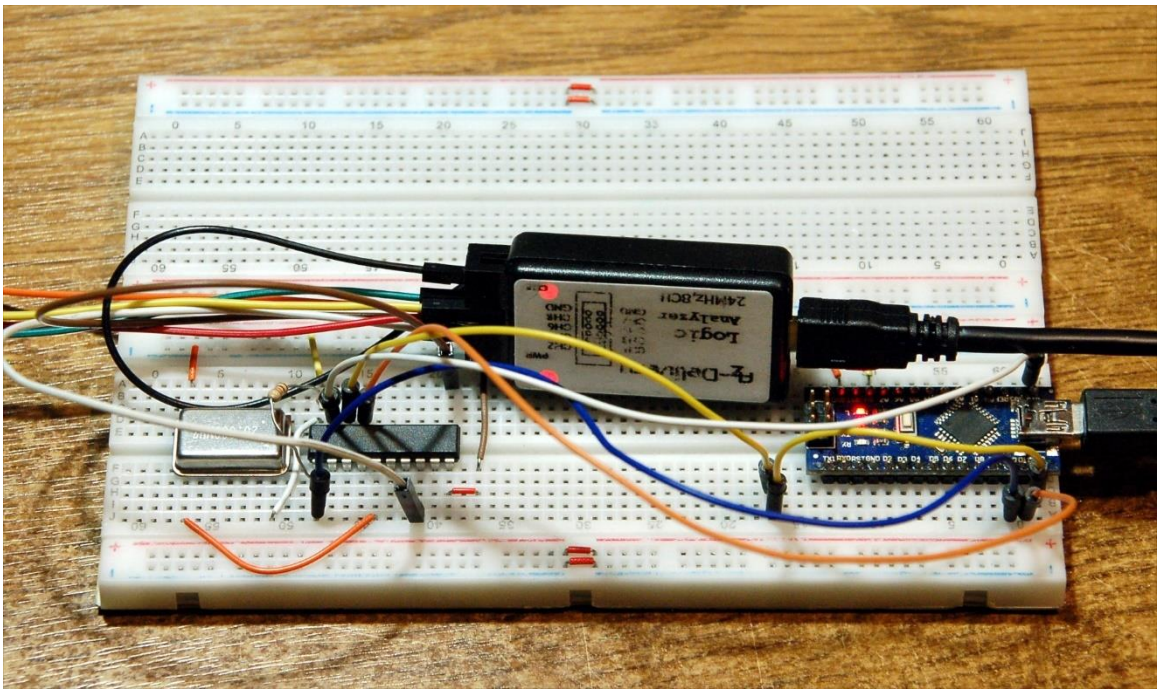

Abbildung 30: IRQ-Timing mit dem Logic Analyzer erfassen

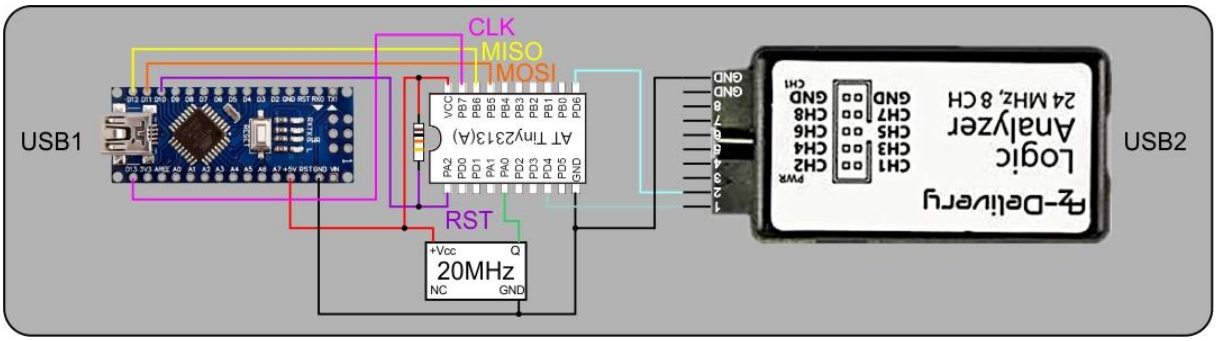

Abbildung 31: Signalabtastung mit dem Logic Analyzer

Bevor wir zum "Brennen" des Programms schreiten, ist noch eine Geschichte zu klären, die oft Unwohlsein und Bauchgrimmen hervorruft, oder gar zum Ableben des Controllers führen kann. Das ist der Umgang mit den **Fuses**. ATMEL-Controller haben neben dem Programm- RAM- und EEPROM-Speicher einen Speicherbereich von vier Bytes, die das Startverhalten und weitere Parameter des Chips einstellen, genannt **Fuses**. Der Begriff **Fuse** ist nicht im Sinn des deutschen Worts Sicherung zu verstehen, vielmehr als Schalter, den man schließen oder öffnen kann. Zwei davon **lfuse** und **hfuse** sind für uns von Interesse, von den anderen sollte man vorerst ganz die Finger lassen. Die einzelnen Bits in den Fuse-Bytes haben folgende Bedeutung.

AT Tiny 2313 -Fuses				
Hfuse = Fuse High Byte				
Bit	Nummer	Beschreibung	Standardwert	mein Wert
DWEN	7	Debugwire enable	1 nicht gebrannt	1
EESAVE	6	EEPROM-Inhalt vor Löschen sichern	1 nicht gebrannt	1
SPIEN	5	Programm und Daten seriell transferieren	0 gebrannt	0
WDTON	4	Watchdog Timer stets an	1 nicht gebrannt	1
BODLEVEL2	3	Brown out Detektor Level Bit 2	1 nicht gebrannt	1
BODLEVEL1	2	Brown out Detektor Level Bit 1	1 nicht gebrannt	1
BODLEVEL0	1	Brown out Detektor Level Bit 0	1 nicht gebrannt	1
RSTDISBL	0	Reset-Eingang deaktivieren !!!!	1 nicht gebrannt	1
			Bytewert	0xDF
Lfuse = Fuse Low Byte				
Bit	Nummer	Beschreibung	Standardwert	mein Wert
CKDIV8	7	Systemtakt durch 8 teilen	0 gebrannt	1
CKOUT	6	Systemtakt an Pin PORTD.2 ausgeben	1 nicht gebrannt	1
SUT1	5	Startzeit Einstellung Bit1	1 nicht gebrannt	1
SUT0	4	Startzeit Einstellung Bit0	0 gebrannt	0
CKSEL3	3	Quelle für Systemtakt wählen Bit3	0 gebrannt	0
CKSEL2	2	Quelle für Systemtakt wählen Bit2	0 gebrannt	0
CKSEL1	1	Quelle für Systemtakt wählen Bit1	1 nicht gebrannt	0
CKSEL0	0	Quelle für Systemtakt wählen Bit0	0 gebrannt	0
			Bytewert	0xE0

Abbildung 32: Fuse-Bitwerte und ihre Bedeutung

Das Fuse Low Byte muss also geändert werden, wenn der AT Tiny 2313 fabrikneu ist. Das liegt daran, dass werkseitig der controllerinterne RC-Oszillator mit 8MHz eingestellt ist und die Taktfrequenz auf 1MHz gedrosselt ist. In diesem Zustand könnte man den AT Tiny 2313 also gänzlich ohne äußere Taktquellen betreiben.

Wir versorgen den Controller aber mit einer externen, viel genaueren Taktquelle (CKSEL3:0 = 0b0000) und wollen eine möglichst hohe Taktgeschwindigkeit erreichen. Also teilen wir nicht durch 8. Damit wird unser Wert von Lfuse zu 0xE0.

Wir fragen zuerst einmal die Fuse-Werte ab.

avrdude -v -pt2313 -cavrisp -PCOM6 -b19200

```
avrdude: AVR device initialized and ready to accept
instructions

Reading | ################################################## |
100% 0.05s

avrdude: Device signature = 0x1e910a
avrdude: safemode: lfuse reads as 62
avrdude: safemode: hfuse reads as DF
avrdude: safemode: efuse reads as FF

avrdude: safemode: lfuse reads as 62
avrdude: safemode: hfuse reads as DF
avrdude: safemode: efuse reads as FF
avrdude: safemode: Fuses OK
```

Die **Lfuse** stellen wir jetzt auf 0xE0 um.

avrdude -v -pt2313 -cavrisp -PCOM6 -b19200 -Ulfuse:w:0xE0:m

```
Reading | ################################################## |
100% 0.06s

avrdude: Device signature = 0x1e910a
avrdude: safemode: lfuse reads as 62
avrdude: safemode: hfuse reads as DF
avrdude: safemode: efuse reads as FF
avrdude: reading input file "0xE0"
avrdude: writing lfuse (1 bytes):

Writing | ################################################## |
100% 0.02s

avrdude: 1 bytes of lfuse written
avrdude: verifying lfuse memory against 0xE0:
avrdude: load data lfuse data from input file 0xE0:
avrdude: input file 0xE0 contains 1 bytes
avrdude: reading on-chip lfuse data:

Reading | ################################################## |
100% 0.01s

avrdude: verifying ...
avrdude: 1 bytes of lfuse verified

avrdude: safemode: lfuse reads as E0
avrdude: safemode: hfuse reads as DF
avrdude: safemode: efuse reads as FF
avrdude: safemode: Fuses OK
```

Wir haben somit die Taktquelle erfolgreich umgestellt. Das Herz unseres AT Tiny 2313 tickt jetzt mit 20MHz. Wenn wir das Lfuse-Bit CKOUT brennen würden, könnten wir den Kanal 3 des Logic Analyzers an PortD.2 anschließen und den Takt aufzeichnen, um ihn am Bildschirm darzustellen. Wie würde das Lfuse-Byte dafür lauten? – Klar: 0b10100000 = 0xA0.

Jetzt wollen wir aber endlich unser Programm zum AT Tiny 2313 schicken.

avrdude -v -pt2313 -cavrisp -PCOM6 -b19200 -Uflash:w:"tinytest.hex":i

```
Writing | ################################################## |
100% 1.02s

avrdude: 1280 bytes of flash written
avrdude: verifying flash memory against tinytest.hex:
avrdude: load data flash data from input file tinytest.hex:
avrdude: input file tinytest.hex contains 1280 bytes
avrdude: reading on-chip flash data:
```

```
Reading | ################################################## |
100% 1.38s

avrdude: verifying ...
avrdude: 1280 bytes of flash verified

avrdude: safemode: lfuse reads as E0
avrdude: safemode: hfuse reads as DF
avrdude: safemode: efuse reads as FF
avrdude: safemode: Fuses OK
```

Der AT Tiny 2313 arbeitet jetzt bereits mit dem neuen Programm. Schauen wir uns die Signale an PortD.4 und PortD.6 an. Der Logic Analyzer hängt ja hoffentlich schon am USB-Bus. Dann starten wir die Anwendung Logic2. Sobald die Verbindung zum Logic Analyzer steht, drücken wir zweimal kurz hintereinander die Taste **R**. Sie startet und stoppt die Aufzeichnung. Mit dem Scroll-Rad der Maus zoomen wir in die Kurven hinein.

Abbildung 33: Ermittlung des Timings mit dem Logic Analyzer

Die Berechnungen, die wir bei der Programmbesprechung aufgestellt haben, werden durch die Messung bestätigt. Mit diesen Werten sind wir auf der sicheren Seite, unseren DDS-Wellenform-Generator doch noch hinzukriegen.

Die nächste Aufgabe, die wir lösen müssen, ist die Umwandlung der Sinuswerte am Port B in Spannungswerte. Leider besitzt der AT Tiny 2313 keinen eingebauten DAC. Aber auch dafür gibt es eine Lösung. Wie die ausschaut und wie es damit weitergeht, das lesen Sie in der nächsten Folge.

Also, bleiben Sie dran!