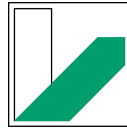


PIC-Programmierung mit PICkit 2 unter Linux

Von Manuel Lippert für den Kurs „Prozessrechner und Elektronik“ an der Universität Bayreuth

31. Januar 2022



1 Einleitung

Die Programmierung eines PIC-Microchip unter Linux hat sich für mich als Linux-Nutzer damals zu einer kleinen Herausforderung entwickelt, weswegen ich versucht habe eine möglichst nahe Programmierung über den Terminal zu suchen. Diese Methode hat den Vorteil, dass man mit einem Editor seiner Wahl Assembler-Code schreiben kann und ihn einfach über Commands im Terminal ausführen kann.

In diesen kleinen Anleitung erkläre ich, wie man dieses Verfahren unter Linux (Ubuntu) aufsetzen lässt. In einem optionalen Schritt zeige ich noch weiterhin, wie man über Visual Studio Code seine Assembler Programmierung aufsetzen kann.

Bei Fragen stehe ich Ihnen gerne über GitHub zur Verfügung. Schreiben sie einfach ein Issue zu Ihrem Problem.

2 Compiler

Zuallererst benötigen wir einen Compiler welcher uns `.asm`-Files in `.hex`-Files kompiliert. Als Compiler benutzen wir das Packages `gputils`, welches über den folgenden Befehl installiert werden kann:

```
sudo apt install gputils
```

Damit ist man nun in der Lage mithilfe von `gpasm` über den Terminal `.asm`-Files zu kompilieren. Bei einem gegebenen File `foo.asm` wäre dann der Befehl im Terminal:

```
gpasm foo.asm
```

Dabei entstehen 3 weitere Files `foo.cod`, `foo.hex` (Dieses File wird benötigt) und `foo.lst`. Damit `foo.hex`-File auf den PIC-Microchip geladen werden kann wird das folgende Tool benötigt.

Wenn der Prozessor, in unserem Fall der PIC16F84A, nicht in der `.asm`-File mit dem Befehl `processor PIC16F84A` definiert worden ist, kann dies in `gpasm` gemacht werden mit:

```
gpasm -p16f84a foo.asm
```

3 pk2cmd

Das pk2cmd Commandline-Tool ermöglicht Linux (Es gibt auch eine Mac und Windows Version) mit dem PICkit 2 PIC-Microchip über den Terminal anzusteuern. Hier wird es dafür benötigt um den compilierten Assembler-Code auf den PIC-Microchip zu laden.

3.1 Installation

Quelle: <https://www.making-sound.co.uk/tech-notes/pickit2-linux.html>

- 1) Zuerst ladet man die nötigen Pakete um pk2cmd zu installieren. Das erfolgt über diesen Befehl im Terminal:

```
sudo apt install build-essential libusb-dev
```

- 2) Lade pk2cmd von GitHub und entpacke Sie den Download in Ihrem Download Ordner. Öffnen Sie dann den Terminal und navigieren zum diesem Ordner-Verzeichnis.

```
cd ~/Downloads/pk2cmd/pk2cmd
```

Alternative mit git über Terminal herunterladen und in das Verzeichnis navigieren:

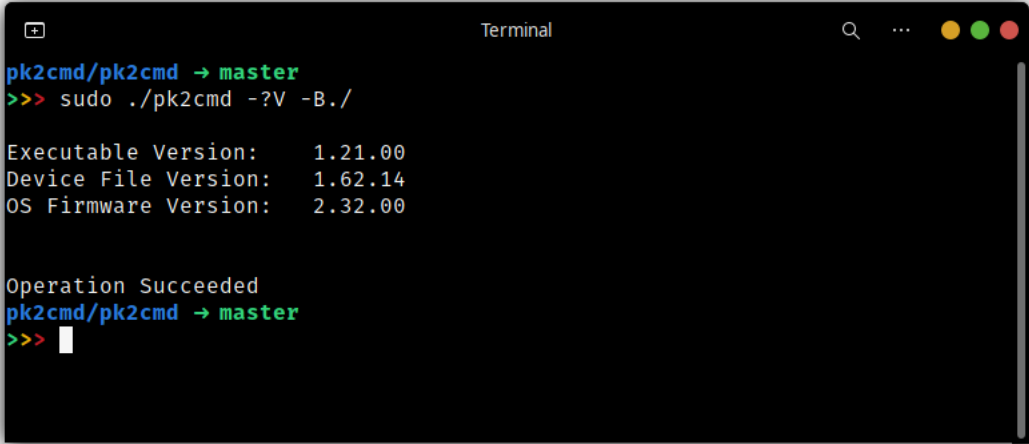
```
git clone https://github.com/psmay/pk2cmd.git  
cd pk2cmd/pk2cmd
```

- 3) Zum installieren führen Sie folgende Befehle aus:

```
make linux
```

- 4) Zum Testen von pk2cmd führen Sie diesen Command im selben Ordner aus:

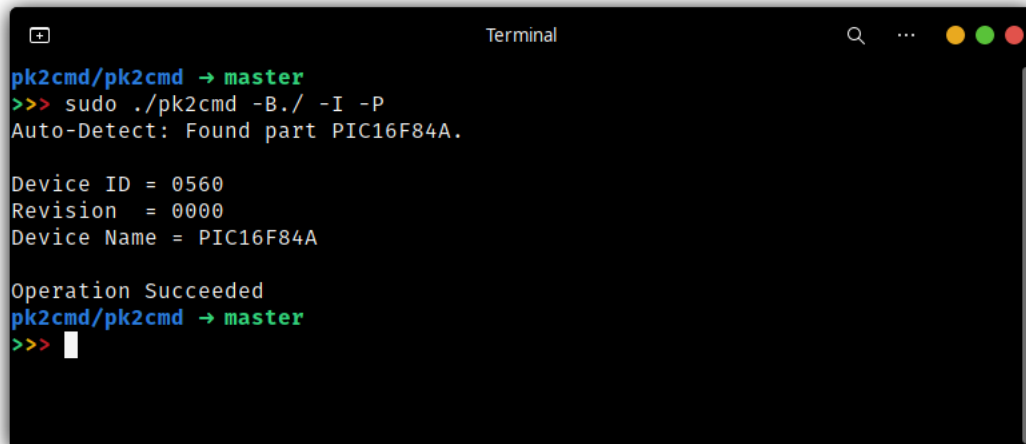
```
sudo ./pk2cmd -?V -B./
```



```
pk2cmd/pk2cmd → master  
>>> sudo ./pk2cmd -?V -B./  
  
Executable Version: 1.21.00  
Device File Version: 1.62.14  
OS Firmware Version: 2.32.00  
  
Operation Succeeded  
pk2cmd/pk2cmd → master  
>>> 
```

- 5) Zum Testen ob der PIC-Microchip von `pk2cmd` angesteuert wird, kann bei angeschlossenen PICKit 2 mit Microchip dieser Command im selben Ordner ausgeführt werden:

```
sudo ./pk2cmd -B./ -I -P
```



```
pk2cmd/pk2cmd → master
>>> sudo ./pk2cmd -B./ -I -P
Auto-Detect: Found part PIC16F84A.

Device ID = 0560
Revision  = 0000
Device Name = PIC16F84A

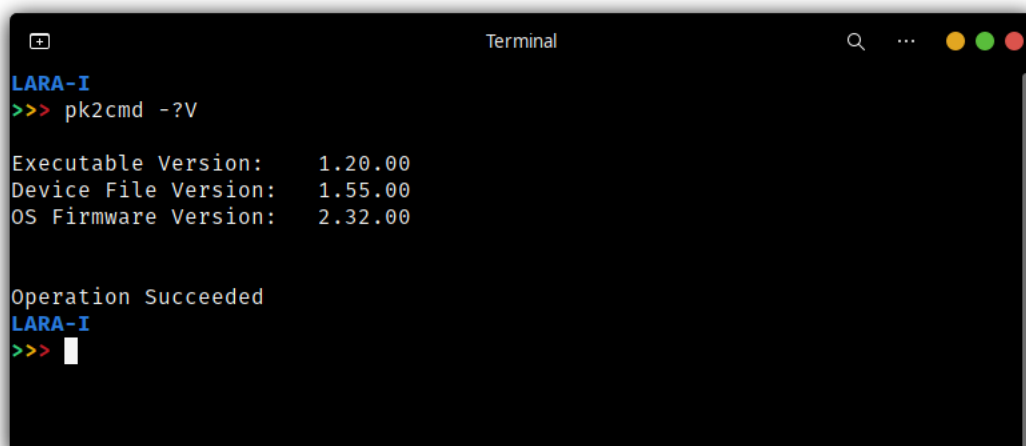
Operation Succeeded
pk2cmd/pk2cmd → master
>>> 
```

- 6) Nun wird der Command „pk2cmd“ global zugänglich gemacht mit diesem Befehl, welcher wieder im selben Ordner ausgeführt werden muss:

```
sudo make install
echo 'export PATH="$PATH:/usr/share/pk2"' >> ~/.bashrc
```

Danach kann man den Ordner verlassen und den globalen Befehl wie folgt testen:

```
cd
pk2cmd -?V
```



```
LARA-I
>>> pk2cmd -?V

Executable Version: 1.20.00
Device File Version: 1.55.00
OS Firmware Version: 2.32.00

Operation Succeeded
LARA-I
>>> 
```

Bei Problemen ist in der Quelle ein Troubleshooting aufgeführt, welches Ihnen bestimmt weiter helfen wird. Dies wird aber hier bewusst weggelassen, da ab nun alles funktionieren sollte.

3.2 Befehlspalette

Quelle: <https://github.com/kvadevack/pk2cmd/blob/master/pk2cmd/release/Readme%20For%20PK2CMD.txt>

PICKit 2 COMMAND LINE HELP		
Options	Description	Default
A<value>	Set Vdd voltage	Device Specific
B<path>	Specify the path to PK2DeviceFile.dat	Searches PATH and calling dir
C	Blank Check Device	No Blank Check
D<file>	OS Download	None
E	Erase Flash Device	Do Not Erase
F<file>	Hex File Selection	None
G<Type><range/path>	Read functions Type F: = read into hex file, path = full file path, range is not used Types P,E,I,C: = output read of Program, EEPROM, ID and/or Configuration Memory to the screen. P and E must be followed by an address range in the form of x-y where x is the start address and y is the end address both in hex, path is not used (Serial EEPROM memory is 'P')	None
H<value>	Delay before Exit K = Wait on keypress before exit 1 to 9 = Wait <value> seconds before exit	Exit immediately
I	Display Device ID & silicon revision	Do Not Display
J<newlines>	Display operation percent complete N = Each update on newline	Rotating slash
K	Display Hex File Checksum	Do Not Display
L<rate>	Set programming speed <rate> is a value of 1-16, with 1 being the fastest.	Fastest
M<memory region>	Program Device memory regions: P = Program memory E = EEPROM I = ID memory C = Configuration memory If no region is entered, the entire device will be erased & programmed. If a region is entered, no erase is performed and only the given region is programmed. All programmed regions are verified. (serial EEPROM memory is 'P')	Do Not Program

N<string>	Assign Unit ID string to first found PICkit 2 unit. String is limited to 14 characters maximum. May not be used with other options. Example: -NLab1B	None
P<part>	Part Selection. Example: -PPIC16f887	(Required)
P	Auto-Detect in all detectable families	
PF	List auto-detectable part families	
PF<id>	Auto-Detect only within the given part family, using the ID listed with -PF Example: -PF2	
Q	Disable PE for PIC24/dsPIC33 devices	Use PE
R	Release /MCLR after operations	Assert /MCLR
S<string/#>	Use the PICkit 2 with the given Unit ID string. Useful when multiple PICkit 2 units are connected. Example: -SLab1B If no <string> is entered, then the Unit IDs of all connected units will be displayed. In this case , all other options are ignored. -S# will list units with their firmware versions. See help -s? for more info.	First found unit
T	Power Target after operations	Vdd off
U<value>	Program OSCCAL memory, where: <value> is a hexadecimal number representing the OSCCAL value to be programmed. This may only be used in conjunction with a programming operation.	Do Not Program
V<value>	Vpp override	Device Specific
W	Externally power target	Power from Pk2
X	Use VPP first Program Entry Method	VDD first
Y<memory region>	Verify Device P = Program memory E = EEPROM I = ID memory C = Configuration memory If no region is entered, the entire device will be verified. (Serial EEPROM memory is 'P')	Do Not Verify
Z	Preserve EEData on Program	Do Not Preserve
?	Help Screen	Not Shown

3.3 Verwendung

Um nun ein gegebenes `foo.hex`-File auf einen PIC-Microchip in unseren Fall der PIC16F84A zu laden wird folgender Befehl im Terminal ausgeführt:

```
pk2cmd -A5 -PPIC16F84A -F foo.hex -M -T -R
```

Nach dem ausüben dieses Befehls sollte der Microchip das Programm ausführen. Was die jeweiligen Attribute des Befehls festlegen, kann im Kapitel 3.2 nachgelesen werden. Ab hier können Sie nun über einen Editor Ihrer Wahl `.asm`-File erstellen mit `gpasm` kompilieren und mit `pk2cmd` auf den PIC-Microchip laden.

4 Visual Studio Code

Visual Studio Code (VS Code) ist ein sehr stark über Extensions modifizierbarer Open-Source Code-Editor auf der Basis von Electron-Framework. Der Vorteil an der Nutzung von VS Code liegt darin, dass man mit diesem Editor so ziemlich jede Programmiersprache nutzen kann. Dafür muss man zwei Dinge machen:

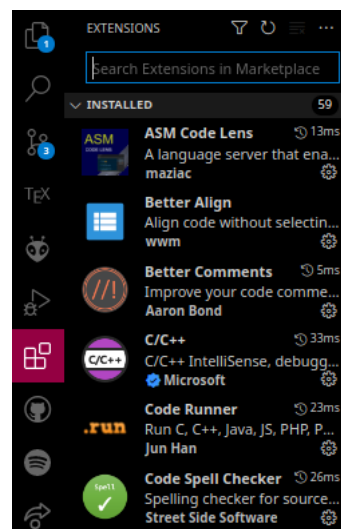
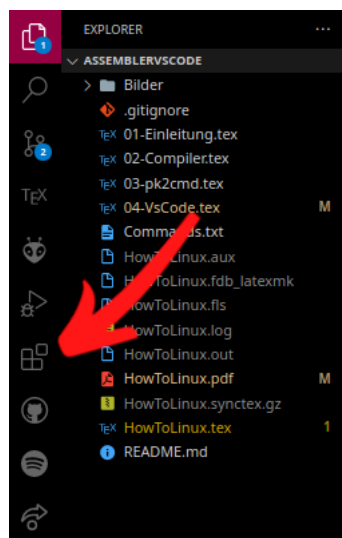
- 1) Die Programmiersprache global über den Terminal zugänglich machen (z.B. man tippt `python` in den Terminal und dieser öffnet die `python-console`)
- 2) In VS Code einen Language-Support über eine Extension aktivieren, damit VS Code weiß welche Programmiersprache gemeint ist (Das schließt auch Syntax-Highlighting ein)

In diesem Kapitel zeige ich Ihnen wie sie das für Assembler machen können und wie sie mit der Extension „Code Runner“ alles automatisch über einem `.asm`-File ausführen können.

Wie man VS Code installiert zeige ich hier nicht, verlinke aber die Seite mit dem Download und den Erklärung dafür hier. Für Ubuntu benötigen sie den Download für Debian.

4.1 Extensions

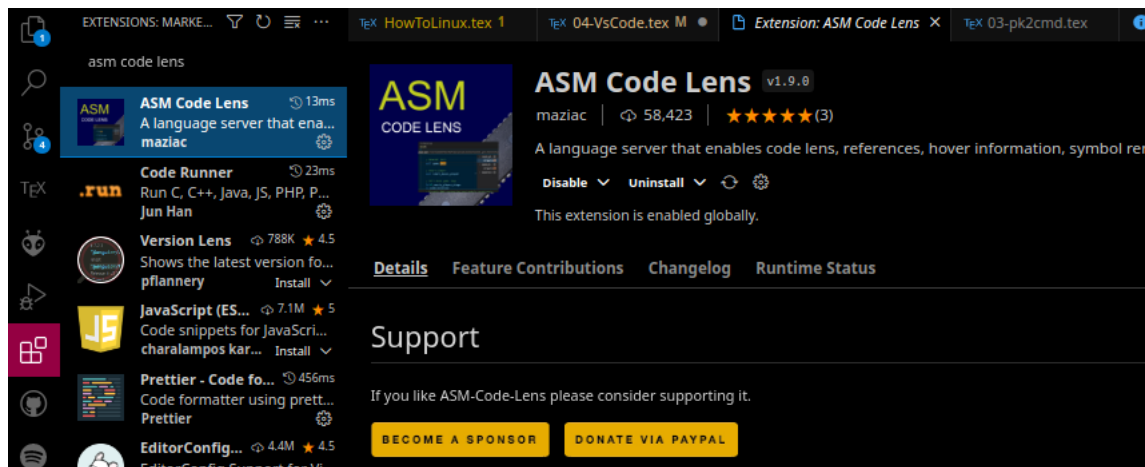
Wenn sie VS Code starten erwartet Sie ein windows-typisches Interface. Auf der linken Seite sind mehrere Menü-Icons aufgeführt, hier wählen sie das Icon, welches in der linken Abbildung gezeigt wird. Wenn der Extensionbereich geöffnet ist, sehen Sie oben die Suchleiste (bei mir blau umrahmt in der rechten Abbildung), darin suchen Sie dann folgende Extensions.



- „Code Runner“
Diese Extension macht uns möglich die Terminal-Befehle die in den vorherigen Kapitel erklärt habe über ein paar Klicks auszuführen.



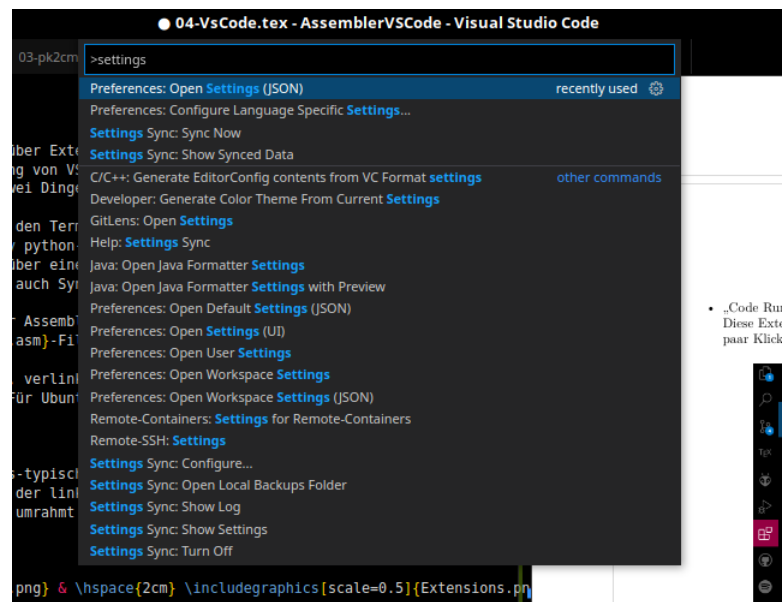
- „ASM Code Lens“
Diese Extension ist für das Syntax-Highlighting und den Language-Support verantwortlich. Damit lassen sich .asm-Files als „Assembler file“ erkennen lassen. Weiterhin hat sie noch ein paar nützliche Feature wie Referenzen.



4.2 Einstellungen

Nun wollen wir VS Code nur sagen, was es zu tun hat. Dies kann man entweder in einer GUI in den Einstellungen machen oder in der `settings.json`. Ich zeige den zweiten Weg, da dieser einfacher ist zum ausführen.

- 1) Führen Sie die Tastenkombination **STRG+SHIFT+P** aus oder **View -> Command Palette** in der Menüleiste. Es sollte sich nun oben eine Suche aufgemacht haben. Dort geben sie „settings“ ein und wählen die Option „Preferences: Open Settings (JSON)“ auch nochmal gezeigt in der Abbildung.



- 2) In diese Datei fügen sie folgende Code-Zeilen innerhalb der geschweiften Klammern ein (Ich habe in Kommentaren gekennzeichnet mit `//` reingeschrieben was welche Zeile macht)

```
// Sagt VS Code als was er .asm-Files und .INC-Files behandeln soll
"files.associations": {
    "*.asm": "asm-collection",
    "*.INC": "asm-collection"
},

// Code Runner
// Play-Button zum Ausfuehren des Codes in TitleMenu
"code-runner.showRunIconInEditorTitleMenu": true,
// Fuehrt Code in Terminal aus
"code-runner.runInTerminal": true,
// Befehle um .asm-File oder .hex-File auszufuehren mit dem PIC16F84A als
// Standard PIC
"code-runner.executorMap": {
    "asm-collection": "cd $dir && gpasm -p16f84a $fileNameWithoutExt.asm &&
        pk2cmd -A5 -PPIC16F84A -F $fileNameWithoutExt.hex -M -T -R && cd
        $workspaceRoot",
    "hex": "cd $dir && pk2cmd -A5 -PPIC16F84A -F $fileNameWithoutExt.hex -M
        -T -R"
},
```

- 3) Save mit **STRG+S**

4.3 Erstes .asm-File ausführen

Als letzten Punkt in dieser Anleitung zeige ich Ihnen wie Sie nun ein .asm-File ausführen können. Dafür drücken sie einfach **STRG+N** in VS Code. Es soll sich ein Tab aufgemacht haben mit dem Namen „Untitled-1“. In die frei Fläche fügen sie den Code `lauf.asm` den ich unten aufführe. Achten Sie auf die Formatierung!

```
;Von Reinhard Richter aus dem Kurs Prozessrechner und Elektronik

processor    PIC16F84A
include     p16f84a.inc

;Configurationword*****
__CONFIG _CP_OFF & _XT_OSC & _WDT_OFF & _PWRTE_ON ;config: 0x3ff1

;INITIALISIERUNG*****
org 0x000      ;schreibe den naechsten Befehl an die Speicherstelle 0x000

    bsf      STATUS,RP0          ;BANK1
    clrf     TRISA                ;PORTA als AUSGANG
    clrf     TRISB                ;PORTB als AUSGANG
    bcf      OPTION_REG,PSA       ;Prescaler auf TMR0
    bsf      OPTION_REG,PS0       ;
    bsf      OPTION_REG,PS1       ;Prescaler auf 1:256
    bsf      OPTION_REG,PS2       ;
    bcf      OPTION_REG,TOCS      ;Internal Clock
    bcf      STATUS,RP0          ;BANK0
    clrf     PORTA                ;PORTA loeschen
    clrf     PORTB                ;PORTB loeschen
    movlw    0x01                ;00000001 in W
    movwf    0x0C                ;lade W in Register 0x0C
    bcf      STATUS, C           ;loesche das Carry-Bit

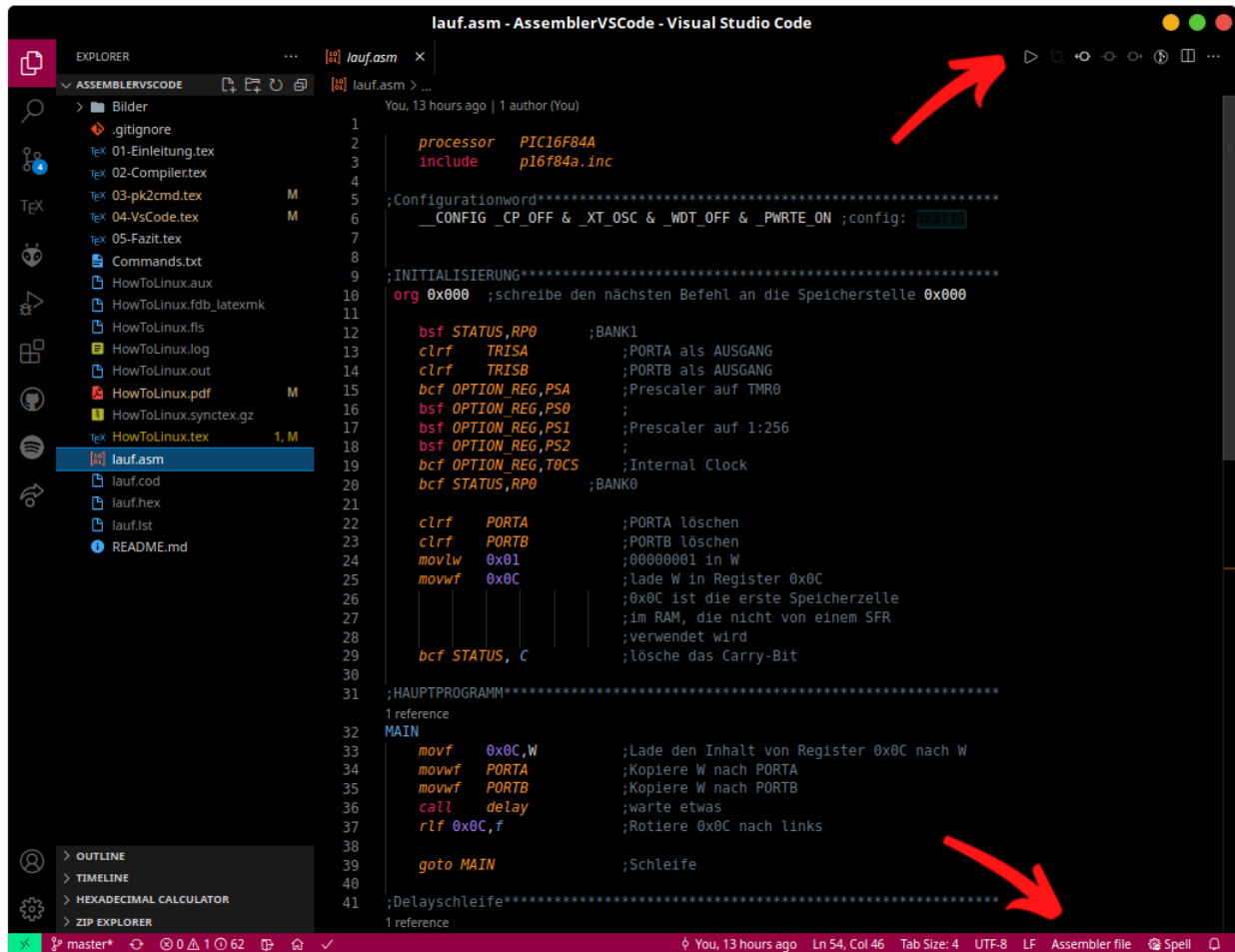
;HAUPTPROGRAMM*****
MAIN
    movf     0x0C,W               ;Lade den Inhalt von 0x0C nach W
    movwf    PORTA               ;Kopiere W nach PORTA
    movwf    PORTB               ;Kopiere W nach PORTB
    call     delay               ;warte etwas
    rlf      0x0C,f              ;Rotiere 0x0C nach links
    goto     MAIN               ;Schleife

;Delayschleife*****
delay
    clrf     TMR0                ;loesche TMR0
    bcf      INTCON,TOIF         ;TMR0 overflow interrupt flag loeschen

delay_loop
    btfss    INTCON,TOIF         ;springe wenn TOIF gesetzt
    goto     delay_loop
    return

end
```

Speichern Sie den Code mit **STRG+S** und geben Sie ihm den Namen `lauf.asm`. Wenn alles geklappt sollte es bei Ihnen aussehen wie unten in der Abbildung. Wichtig ist dabei, dass unten rechts in der Status-Leiste (Bei mir rote Leiste) „Assembler file“ steht. Wenn dies der Fall ist können Sie das PICKit 2 mit dem Microchip anschließen und oben in der rechten Ecke den grauen Play-Button drücken um den Code ausführen zu lassen.



Nach dem Drücken des grauen Play-Button sollte sich ein Terminal aufmachen, indem der Code, wie zuvor in CodeRunner spezifiziert wurde, ausgeführt wird. Der Output sollte am Ende dann so aussehen:

```
Device Type: PIC16F84A

Program Succeeded.

Operation Succeeded
```

Wenn alles geklappt hat, sollte dann sollten die LEDs in ein Lauflicht (Takt für Takt leuchtet eine andere LED auf) erzeugt worden sein. Damit sind Sie auch bereit absofort Assembler-Code in VS Code zu schreiben.

5 Fazit

Nun habe ich Ihnen viel zugemutet. Die Nutzung von VS Code kann ich Ihnen dennoch ans Herz legen, da dieses Programm sehr nützlich ist in vielerlei Hinsicht. Ansonsten kann ich Ihnen nur viel Spaß im Kurs „Prozessrechner und Elektronik“ wünschen.