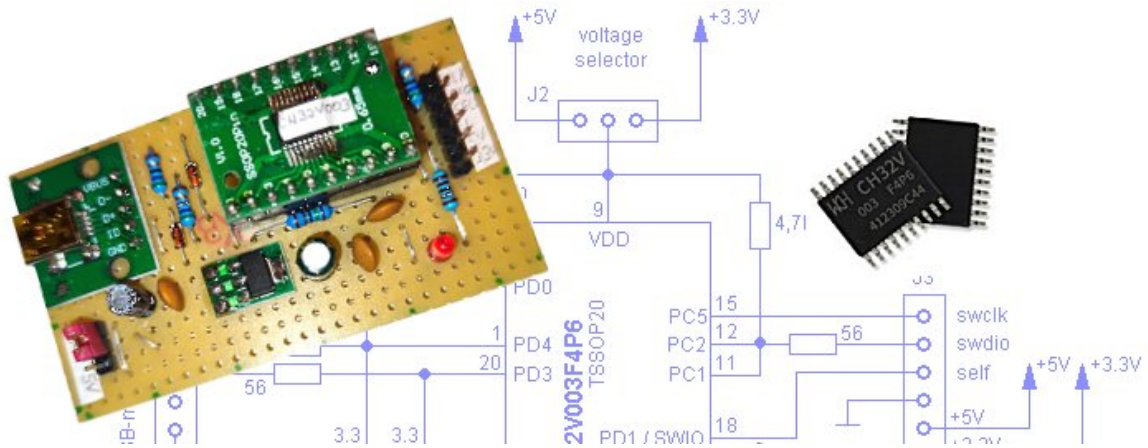
Selbstbauprogrammer für CH32V003
Getting started unter Linux

R. Seelig



Aller Anfang ist schwierig und möchte man mit einem unbekanntem Mikrocontroller starten kommt es unweigerlich zu unterschiedlichen Einstiegshürden.

Heutzutage gibt es für Bastler (heißt mittlerweile ja neudeutsch „Maker“) jedoch viele Produkte die diesen Einstieg erleichtern und so erklärt sich auch der große Erfolg von Arduino: Einstecken, Programm aufspielen, geht.

Für diese Gruppe ist der Selbstbauprogrammer nicht gedacht. Für den, der sich den Einstieg leichter machen und mit einem CH32V003 anfangen möchte gibt es vom Hersteller des Chips ein Entwicklungspaket namens ***MounRiver Studio*** welches hier gedownloadet werden kann:

http://mounriver.com/download

Hinweis: Der hier vorgestellte Selbstbauprogrammer ist mit der IDE ***MounRiver Studio*** NICHT kompatibel, bzw. kann von dieser IDE heraus nicht gestartet werden. Hier müßte auf Konsolenebene ein erstellte Binary- oder Hexdatei manuell in den Controller übertragen werden. Des weiteren benötigt es zum Programmieren der Firmware des Programmers einen Ardudino UNO / nano oder eines AVR-Systems mit zugänglichen GPIO-Pins und einer USB2UART Bridge. ATmega328, ATmega168, ATmega8 und ATtiny2313 sind in der Lage, die Firmware in den Programmer zu flashen.

Übersicht der Schritte zum Aufbau des Selbstbauprogrammers und „erste Schritte“ bei der Programmierung eines CH32V003

- Hardwareaufbau des Programmers
- Installieren des RISCV-Compilers (wird für die Compilierung der Programmerfirmware und der späteren Programmierung von CH32V003 Mikrocontrollern benötigt)
- Installieren des zu diesem Text gehörenden Archivs (alle zum Aufbau des Programmers und späteren Umgang mit CH32V003 benötigten Dateien sind hierin enthalten, exclusive des Compilers)
- flashen der Firmware **ARDULINK** auf einen Arduino um hieraus einen (sehr langsamen) Programmer zu machen (wird nur einmal benötigt um die Firmware des Selbstbauprogrammers zu flashen)
- flashen der Firmware des Selbstbauprogrammers über ARDULINK
- Compilieren des Hostprogramms **minichlink** über den der Selbstbauprogrammer angesprochen wird
- evtl. installieren eines Texteditors (vorzugsweise hier: Geany) mit dem ein Programm für den CH32V003 bearbeitet oder erstellt werden kann
- Demoprogramme des Archivs testen

Programmer

Grundsätzlich gibt es vom Hersteller des Chips auch einen sehr günstigen Programmer / Debuger für den CH32V003 zu kaufen, aber hier gibt es schon den allerersten Fallstrick, auf den man hereinfallen kann (und auf den auch der Autor hereingefallen ist):

Im Internet werden unterschiedliche Programmer angeboten, die auf den ersten Blick identisch aussehen, es jedoch nicht sind:

WCH-LinkE

WCH-Link (ohne "E")

Hier ist zwischen einem Programmer WCH-LinkE (mit einem großem „E“ am Ende) und einem WCH-Link (OHNE einem „E“ am Ende) zu unterscheiden.

Wichtig: Der WCH-Link ohne „E“ ist NICHT in der Lage einen CH32V003 zu programmieren und ist wohl ein älterer Adapter, der gerne als WCH-LinkE verkauft wird (dem Autor wurden insgesamt 3 Stück ohne „E“ verkauft und das ist der Grund, warum der Selbstbauprogrammer aufgebaut worden ist). Unterscheiden kann man die Versionen vordergründig dadurch, dass der Programmer ohne „E“ mit einem 16 pol. IC (CH549) realisiert ist, der mit „E“ einen 20 pol. CH32V305 beherbert.

Dieser Programmer ist natürlich mit dem herstellereigenen ***MounRiver Studio*** kompatibel, jedoch auch mit dem hier später vorgestellten Framework CH32FUN auf Kommandozeilenebene.

Der Selbstbauprogrammer

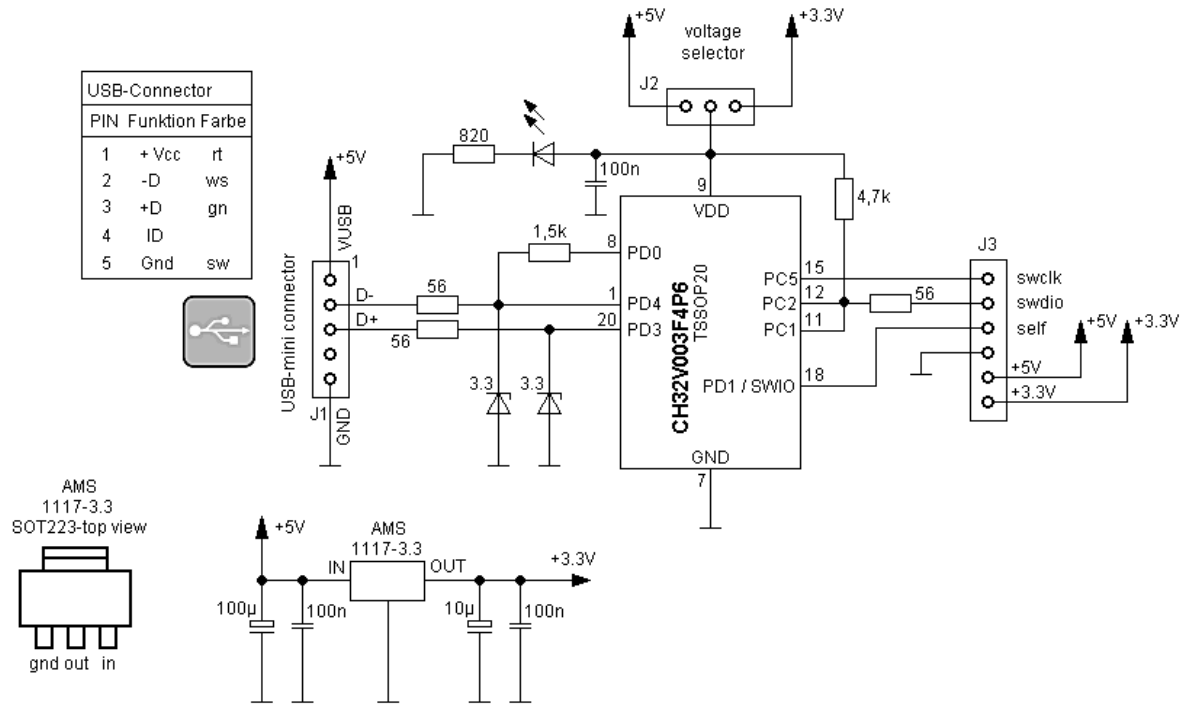
Um es hier deutlich zu machen: Der Selbstbauprogrammer sowie das CH32FUN Framework sind nicht vom Autor dieses Textes. Das Programm des Programmers sowie das Framework selbst sind auf zwei github-Seiten von Christian Lohr zu finden. Für das Arbeiten mit diesem Text ist ein Download von diesen Seiten nicht notwendig, weil bis auf den Compiler selbst alle benötigten Dateien im Archiv zu diesem Text enhalten sind. Die Linkadressen zu Christian Lohr‘s Seiten wird hier für denjenigen aufgeführt, der sich evtl. aus den dort erhältlichen Programmen und Dateien sein eigenes Setup „zusammenbasteln“ oder sich über weitreichende Beispiele informieren mag:

CH32FUN-Framework: https://github.com/cnlohr/ch32fun

RVSWIO-Programmer und V-USB: https://github.com/cnlohr/rv003usb

Dass der Autor den Programmer aufgebaut hat, ist zum Einem dem Umstand geschuldet, dass er „verärgert“ war über die Nichtlieferung eines geeigneten Programmers sowie die sehr lange Wartezeit bis ein solchiger geliefert wurde. Zum anderen arbeitet er gerne mit eigenen Tools, damit er im Falle eines Defektes sich selbst schnell behelfen kann. Ist man zudem unschlüssig, ob man mit dem CH32V003 etwas anfangen mag bedarf es außer dem Zeitaufwand wirklich nur sehr wenig finanziellen Mitteln um diesen Chip zu erkunden. Bestellt man bspw. bei AliExpress oder bei www.lcsc.com sowieso irgendwelche Bauteile, so kann man sich ein paar der Chips gleich mitbestellen, was angesichts von 0,25€ per Chip verschmerzbar ist. Außer PCB-Adaptern für den Chip selbst, der Mini-USB Buchse sowie für einen AMS1117 3.3V Spannungsreglers bedarf es nur absolut üblichen Elektronikbauteilen um mit dem CH32V003 zu starten.

Schaltplan



Um mit dem Programmer Zielsysteme sowohl für 3,3V und 5V unterstützen zu können, wurde der Schaltung ein 3,3V Spannungsregler hinzugefügt, die mittels eines Jumpers (voltage selector) oder eines Schiebeschalters umschaltbar ist. Da ein USB-Pegel nur 3,3V Pegel annehmen darf, sind in die D- und D+ Leitungen des USB in allerbester (okay, eher „dirty“) V-USB Manier im Stile von bspw. eines USBasp 3,3V Zenerdioden eingefügt worden. Hier sind kleine 1/4W Z-Dioden zu verwenden. Bei leistungsstärkeren Z-Dioden kann es vorkommen, dass die Kennlinie im Sperrbereich die Spannung nicht steil genug begrenzt und höhere Pegel (bis hinauf zu 3,9V) auf den Leitungen des USB auftreten. Hier arbeitet dann der Programmer nicht richtig, bzw. wird der Programmer vom PC nicht als solcher erkannt und arbeitet demzufolge dann mit diesem auch nicht zusammen.

Die Connector-Reihe J3 beherbergt alle Anschlüsse die für den Umgang mit einem Zielsystem benötigt werden.

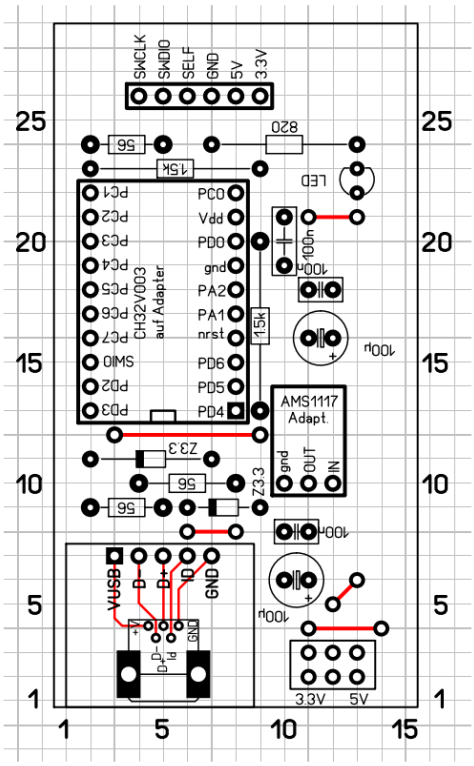
- **swclk / swdio** ist an die Pins des Zielsystems anzuschließen, wobei für das Programmieren eines CH32V003 nur swdio benötigt wird, da dieser über ein SingleWire Protokoll verfügt.
- **self** wird benötigt, um die Firmware in den Programmer zu flashen (wie dieses ohne einen vorhandenen Programmers erfolgen kann wird später erklärt)
- **GND / 5V / 3,3V** sind Versorgungsanschlüsse für das Zielsystem

Der Programmer selbst kann sogar auf einem Steckbrett aufgebaut werden (für einen ersten Versuch), nur leider sind Steckbretter bezüglich der Steckverbindungen nicht sonderlich zuverlässig. Aus diesem Grund heraus hat der Autor (seit Jahren das erste mal wieder) einen Aufbau auf Lochrasterkarte vorgenommen. Dieses ist dann immer noch schneller, als auf die Lieferung eines korrekten WCH-LinkE oder einer entworfenen Platine zu warten. Für denjenigen, der das auf Lochrasterkarte aufbauen mag, wurde ein Layout für Lochrasterkarte und Silberdraht erstellt.

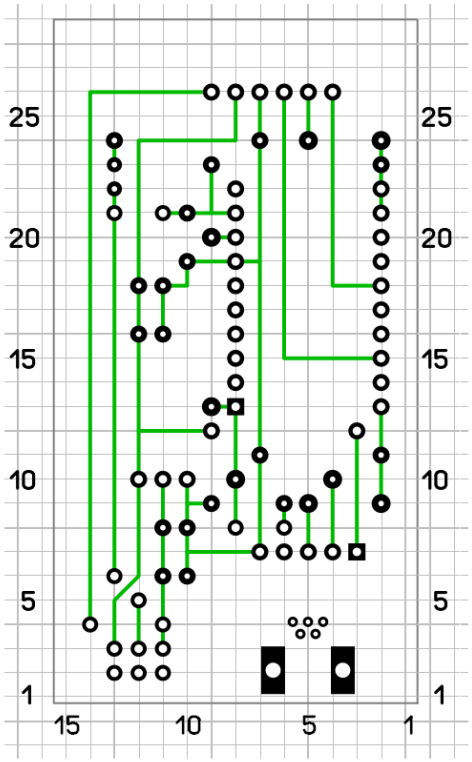
Layout Lochrasterkarte

CH32V003-Programmer

Bestückungsseite



Lötseite



Hinweis: Der CH32V003 im TSSOP-20 Gehäuse, die Mini-USB Buchse sowie der Spannungsregler AMS1117 3.3 sind auf PCB-Adaptern zu verlöten und diese Adapter dann auf der Lochrasterkarte zu verbauen. Beim Aufbau darauf achten, von den niedrigen Bauteilen zu den höhreren Bauteilen zu Bestücken. Am Besten wird mit den Lötbrücken angefangen.

Die Lötseite ist so dargestellt, wie die Verdrahtung aussehen muß, wenn die Platine umgedreht wird.

Inbetriebnahme

Vorraussetzungen, um den Programmer und ein späteres Arbeiten mit CH32V003 zu ermöglichen:

- ein systemweit erreichbarer RISCV- Compiler für den Chip, downloadbar unter:

  https://github.com/xpack-dev-tools/riscv-none-elf-gcc-xpack/releases/
  oder
  https://www.jjflash.de/ch32v003/downloads

  (xpack-riscv-none-elf-gcc-14.2.0-3-linux-arm64.tar.gz auswählen)

- einen Arduino UNO / nano oder ein AVR-System (ATmega328 / 168, ATmega8 oder ATtiny2313)
- ein im System eingerichtetes AVRDUDE
- Texteditor um Programme erstellen / bearbeiten zu können (Empfohlen wird hier **Geany** oder der Konsoleneditor **cide**

Die Reihenfolge für die Inbetriebnahme des Selbstbauprogrammers und der Toolchain für den CH32V003 ist:

- installieren des Compilers
- installieren des zu diesem Text gehörenden Archivs
- compilieren des Hostprogramms **minichlink**
- flashen eines AVR-Controllers um aus diesem einen **ARDULINK** zu machen, der dann seinerseits die Firmware für den Selbstbauprogrammer flashen kann
- flashen des Selbstbauprogrammers
- mit den Demoprogrammen aus dem Archiv, einem CH32V003 und dem Selbstbauprogrammer experimentieren => **getting started**

Installieren des Compilers

Als erstes muß der Compiler aus den oben genannten Quellen gedownloaded und ausgepackt werden. Dieses kann entweder aus dem Desktop heraus mit dem Programm **Engrampa,** in der Konsole mit dem **Midnight Commander** oder manuell mittels Befehlseingabe erfolgen.

Grundsätzlich kann für den Speicherort des Compilers jedes Verzeichnis gewählt werden, da alle vom Compiler benötigten Dateien relativ zu seiner Verzeichnisstruktur vorhanden sind. Der Autor hat als Speicherort **/usr/local** gewählt, damit der Compiler für alle Benutzer verfügbar gemacht werden kann. Da mit Emgrampa und dem Midnight Commander das Entpacken menügesteuert ist bedarf es für diese Variante keine weitere Erklärung. Exemplarisch hier also nur das Entpacken in der Kommandozeile:

In den Ordner wechseln, in den das Compilerarchiv gedownloaded wurde, dort sich als Superuser anmelden:

```
su
Passwort:
```

und in der nachfolgenden Zeile das Passwort hierfür eingeben. Anschließend das Archiv auspacken mit:

```
tar -xvf xpack-riscv-none-elf-gcc-14.2.0-3-linux-x64.tar.gz -C /usr/local
```

Das Archiv wird jetzt ausgepackt und im Ordner **/usr/local** gibt es einen neuen Unterordner mit dem Namen:

```
xpack-riscv-none-elf-gcc-14.2.0-3-linux-x64
```

Innerhalb dieses Ordners ist ein weiterer Ordner namens **bin** enthalten. Dort sind alle Programmteile des Compilers gespeichert und der vollständige Pfad zu diesen Programmen lautet dann:

```
/usr/local/xpack-riscv-none-elf-gcc-14.2.0-3-linux-x64/bin/
```

Dieser Pfad ist wichtig, weil er in den Suchpfad eingetragen werden muß, in dem Linux aufgerufene ausführbare Dateien sucht.

Für die Aufnahme des Compilerpfades in den Suchpfad gibt es 2 Möglichkeiten. Zum einen ist es möglich, den Suchpfad so zu erweitern, dass alle Benutzer des Linuxsystem diesen Compiler nutzen können. Zum anderen kann der Suchpfad nur für den aktuellen Benutzer erweitert werden.

In beiden Fällen sind ist jeweils eine Textdatei zu editieren und um einen Eintrag zu erweitern.

Für die Erweiterung des Suchpfades für alle Benutzer ist die Datei **/etc/profile** als **root** zu bearbeiten. Ist das Programm **Midnight Commander mc** und der dazugehörende Editor **mcedit** auf dem System installiert (auf den meisten System ist das der Fall) kann dieses nach Anmelden auf der Konsole als root mit folgendem Aufruf erfolgen:

```
mcedit /etc/profile
```

Dort muß am Ende der Datei der Eintrag:

```
PATH="$PATH:/usr/local/xpack-riscv-none-elf-gcc-14.2.0-3-linux-x64/bin/"
```

gemacht werden. Hat man den Editor **mcedit** verwendet, ist der Eintrag mittels der F2 Taste zu speichern und der Editor selbst kann mit F10 beendet werden.

Für die Erweiterung des Suchpfades für den aktuellen Benutzer ist die Hidden-Datei **/home/benutzername.bashrc** zu bearbeiten. Wäre der angemeldete Benutzer bspw. **mcu,** würde der Aufruf lauten:

```
mcedit /home/mcu/.bashrc
```

Dort muß am Ende der Datei der Eintrag:

```
export PATH="$PATH:/usr/local/xpack-riscv-none-elf-gcc-14.2.0-3-linux-x64/bin/"
```

gemacht werden. Hat man den Editor **mcedit** verwendet, ist der Eintrag mittels der F2 Taste zu speichern und der Editor selbst kann mit F10 beendet werden.

Es ist sinnvoll, zu überprüfen, ob der Compiler richtig arbeitet. Das kann jetzt oder zu einem späteren Zeitpunkt erfolgen, aber es ist empfehlenswert es gleich nach dieser Installation zu erledigen, damit bei einem späteren eventuellen Fehlverhalten gut ausgeschlossen werden kann, dass der Fehler nicht an einem nicht arbeitenden / auffindbaren Compilerprogramm liegt. Zu diesem Zweck startet man den Rechner am besten neu, damit die gemachten Änderungen aktiv sind.

Nach dem Neustart des Rechners sollte jetzt der Compiler von jedem Verzeichnisort aufgerufen werden können. Hierzu lässt man sich die Ausgabe der Versionsnummer des Compilers geben und damit ist dann geklärt, dass die Suchpfade richtig gesetzt sind:

```
riscv-none-elf-gcc --version
```

Der Compiler antwortet mit:

```
riscv-none-elf-gcc (xPack GNU RISC-V Embedded GCC x86_64) 14.2.0
Copyright (C) 2024 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Installieren des Archivs

Das Archiv hat den Dateinamen **ch32v003.tar.gz** und enthält alle benötigten Dateien und kann in ein (fast) beliebiges Verzeichnis entpackt werden.

Auch hier kann das Entpacken mittels **Engrampa**, dem **Midnight Comander** oder per Befehlseingabe erfolgen.

Nachfolgende Angaben in Bezug auf ein Verzeichnis beziehen sich im Text hier immer auf einen Benutzer mcu. Bei Befehlseingaben ist der entsprechende Benutzername anstelle von mcu einzugeben!

Beim Entpacken des Archivs wird ein Ordner ch32v003 in dem Verzeichnis angelegt, in den das Archiv ausgepackt wird. Im Beispiel hier ist es das Homeverzeichnis des Benutzers **mcu:**

```
tar -xvf ch32v003.tar.gz -C /home/mcu/
```

Das Archiv ist somit entpackt.

Compilieren des Hostprogramms minichlink

minichlink ist das Programm, mit dem der PC über einen Programmer (ARDULINK, der Selbstbauprogrammer oder auch ein Debuger/Programmer von WCH) den Mikrocontroller anspricht und flasht. Dieses Programm liegt im Sourcecode vor und muß compiliert werden.

Hierfür wechselt man in das Verzeichnis **/home/mcu/ch32v003/minichlink** und startet dort **make**:

```
cd /home/mcu/ch32v003/minichlink
make
```

minichlink spricht die Hardware an (USB) und damit ein normaler Benutzer diese Verwenden kann, muß dem Linux gesagt werden, dass ein Benutzer dieses tun darf (nur ein Admin, root oder superuser kann ohne Einschränkung alle Hardware des PC‘s benutzen). Linux verwendet hierfür sogenannte „Regeln“ die im zentralen Konfigurationsverzeichnis **/etc/udev/rules.d** abgelegt werden. In diesen Regeldateien wird vereinbart, welche Benutzergruppen welche Hardware verwenden dürfen. Im Ordner minichlink ist hierfür die Regeldatei **99-minichlink.rules** enthalten, die Hardware für den Selbstbauprogrammer, einen USB-Bootloader für CH32V003 und die originalen Debuger / Programmer von WCH für die Benutzer in den Gruppen **„plugdev“** und **„dialout“** freigibt. Diese Regeldatei muß als root nach **/etc/udev/rules.d** kopiert werden:

```
su
Passwort:
cp 99-minichlink.rules /etc/udev/rules.d
exit
```

Beim nächsten Reboot des PC hat der Benutzer nun die Rechte, Programmer für den Mikrocontroller zu benutzen.

Hinweis: Jedes USB-Gerät besitzt eine Produktidentifikationsnumer, PID oder auch idProduct und eine Herstelleridentifikationsnummer, VID oder auch idVendor. Die Regeldatei beinhaltet eine Auflistung genau dieser Identifikationsnummern zum Zugriff auf die Hardware.

Arduino ARDULINK erstellen

Um die Firmware in den Selbstbauprogrammer zu flashen benötigt es einen Programmer, den wir (noch) nicht haben. Hierfür kann man sich mit einem Arduino UNO/nano oder eines AVR-Systems behelfen, welches über einen der Controller ATmega328p / 168 / 88 oder ATtiny2313 sowie einer USB2UART-Bridge verfügt.

Hierfür gibt es das AVR-Programm ARDULINK. Dieses Programm ist bereits compiliert und kann mittels eines Scripts in den AVR-Controller geflasht werden. Hierzu wechselt man in das Verzeichnis ardulink und startet dort das Script. Im Script selbst sind Angaben über den zu verwendeten Programmer des AVR, den Anschlußport des Controllers und den zu angeschlossenen Controller selbst zu tätigen. Jede der Eingaben im Script ist mit der Return-Taste zu bestätigen.

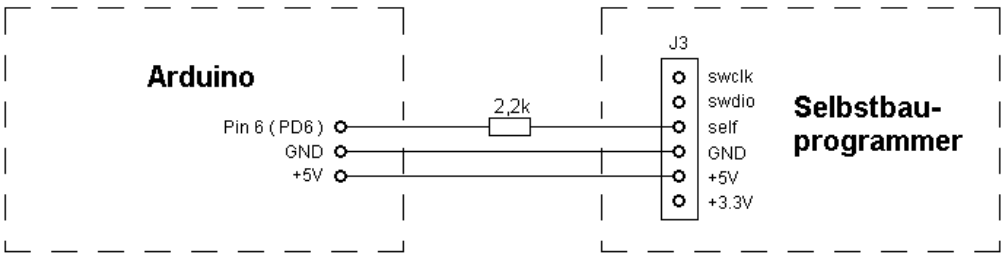
Hinweis: Originale Arduino‘s verwenden den Port ttyACM0, die China-Clones haben in aller Regel einen CH340G Chip als USB2UART-Bridge verbaut und verwenden den PortUSB0. Desweiteren werden originale Arduino nano mit einer Baudrate von 57600 Baud, manche Clone‘s mit 115200 Baud betrieben.

```
cd /home/mcu/ch32v003/ardulink
./flash_ardulink
```

Der Arduino ist jetzt ein Programmer für einen CH32V003, der SWIO-Anschluss ist hier PD6 Arduino Pin 6.

Mit diesem könnte man jetzt schon arbeiten und grundsätzlich Programme für den CH32V003 flashen (wie es der Autor des Textes zu Beginn mit dem CH32V003 auch getan hat). Allerdings hat der ARDULINK-Programmer einen entscheidenden Haken: er ist schier unerträglich langsam. Für das Flashen eines Programms in der max. Größe von 16 kByte benötigt ARDULINK ca. 80 Sekunden , bis das Programm geflasht ist. Für ein flüssiges Arbeiten ist dieses viel zu langsam und aus diesem Grunde wurde der Selbstbauprogrammer aufgebaut.

Firmware des Selbstbauprogrammers flashen



Um die Firmware zu flashen verbindet man Pin 6 / PD6 des Arduino‘s über einen 2,2kΩ Serienwiderstand mit dem self-Anschluß des Selbstbauprogrammers. Der 5V und Masseanschluß GND des Arduinos ist ebenfalls mit dem Selbstbauprogrammer zu verbinden.

Um die Firmware für den Programmer zu flashen wechselt man in das Verzeichnis **/home/mcu/ch32v003/rvswdio_programmer** und startet dort **make**:

```
cd /home/mcu/ch32v003/rvswdio_programmer
make
```

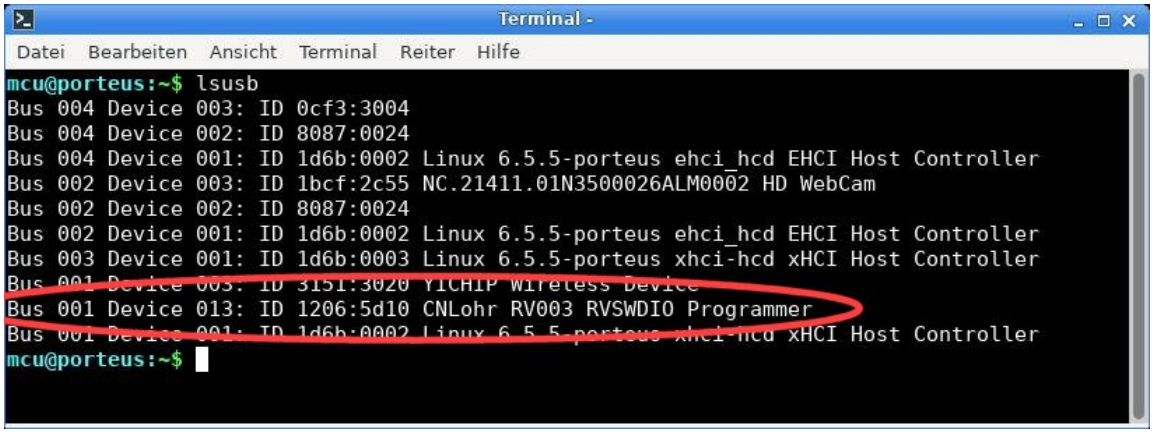
Da der Arduinoprogrammer sehr sehr langsam ist, wird der Flashvorgang ca. 40 Sekunden dauern. Während dieser Zeit werden (leider) keine Bildschirmausgaben gemacht und es muß einfach auf das Ende des Flashvorgangs gewartet werden.

Nach Beendigung des Flashvorgangs ist es geschafft und der Programmer ist betriebsbereit, sofern keine Fehler aufgetreten sind und der Hardwareaufbau ebenso fehlerfrei ist.

Nach dem Anschließen des Programmers an einen USB-Port kann überprüft werden, ob der Programmer sich auch am USB anmeldet. Hierfür gibt man in der Konsole ein:

```
lsusb
```

Der PC listet nun alle am Computer angeschlossenen USB Devices auf. Je nach Version von lsusb werden zu den PID und VID Adressen auch Klartextnamen mit angezeigt. Wichtig hier ist jedoch nur, dass die Adressen des Programmers (ID 1206:5D10) auch aufgelistet sind:



Getting started

Mit CH32FUN von Christian Lohr (Webadresse der Github-Seiten wurden bereits auf Seite 2 dieses Textes genannt) liegt ein Framework vor, dass das Erstellen oder Bearbeiten von Programmen für den CH32V003 sehr erleichtert.

Um dieses tun zu können bedarf es eines Texteditors, der reine Ascii-Texte erstellt. Hierfür empfiehlt der Autor dieses Textes zum Einen **Geany** für den Desktop und zum anderen **CIDE** für die Konsole. Geany kann entweder aus dem Web heruntergeladen und compiliert werden, oder es kann von hier genauso wie **CIDE** als Binary gedownloaded werden:

https://www.jjflash.de/ch32v003/downloads

Sollte das Binary heruntergeladen werden, so muß das Geany-Archiv als Benutzer root in das Stammverzeichnis des Computers entpackt werden.

```
su
Passwort:
tar -xvf geany.tar.gz -C /
exit
```

Letztendlich wird die Geany-Binärdatei in **/usr/bin/** abgelegt. Um Geany vom Desktop aus zu starten muß manuell eine Desktopverknüpfung von **/usr/bin/geany** erstellt werden.

Warum Geany und / oder CIDE ?

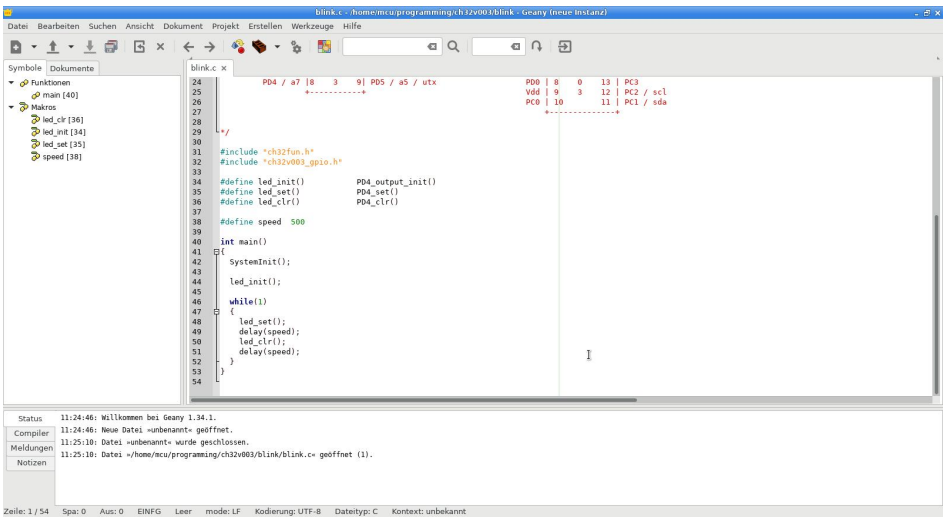
Die Beispielprogramme in diesem Archiv bauen auf ein auf einen CH32V003 reduziertes Framework **CH32FUN** auf. Dieses Framework (und viele andere Programme für auch andere Controller) verwenden zur Erzeugung eines Programms ein sogenanntes **Makefile**. Selbst PC-Programme werden über so ein **Makefile** erstellt. Hier bedarf es dann keiner Entwicklungsumgebung, keines „Studios“, keiner Projektdatei, sondern nur den oder die Quelltexte und eben eines **Makefile**‘s das die Programmerzeugung steuert und im Falle von Mikrocontrollern ein Flashen in den Controller ausführen kann.

Das CH32FUN Framework wurde im Makefile dahingehend modifiziert, dass es:

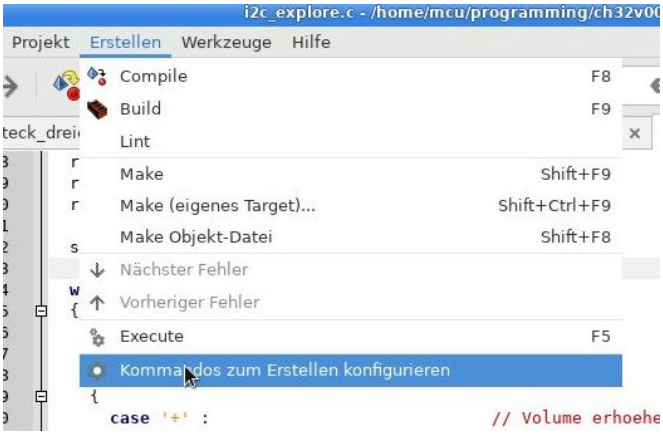
- grundsätzlich das System für 48 MHz mit internem Takt initialisiert wird
- ein einfacher Aufruf von **make** nicht mehr compiliert, linkt und flasht , sondern ein **make** nur noch compiliert und linkt und somit eine Binär- Elf- und Hexdatei für den CH32V003 erzeugt, das Flashen jedoch noch nicht startet (oft möchte man ein Programm nur compilieren um zu sehen, ob es syntaktisch korrekt ist).
- ein **make flash** überträgt bei angeschlossenem Programmer die zuvor mit make erzeugte Binärdatei in den Controller

Geany und **CIDE** sind in der Lage, aus ihrem Programm heraus ein make, make clean oder make flash aufzurufen ohne das Programm selbst zu verlassen. Somit kann ein Programm komfortabel geschrieben, compiliert und geflasht werden.

Geany



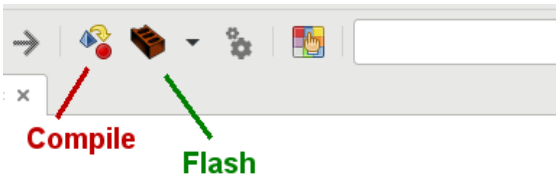
Um Programme von **Geany** heraus zu compilieren und eine erzeugte Binärdate zu flashen muß / sollte man **Geany** für einen Build-Prozess konfigurieren. Hierfür startet man **Geany** und lädt eine *.c Datei in den Editor, damit **Geany** den Dateityp kennt.



Im Menüpunkt „Erstellen“ => „Kommandos zum Erstellen konfigurieren“ (siehe Bild) sind folgende Einstellungen zu machen:

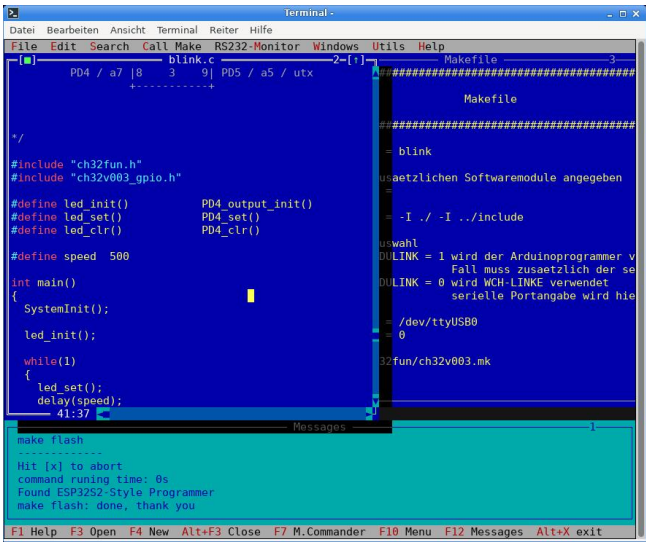
Für:

Compile	make all
Build	make flash
Lint	make clean
Make:	make
Make (eigenes Target)	make
Make Objekt-Datei	make %e.o
Execute:	„./%e“



Geany kann jetzt aus dem Programm heraus eine Quelldatei übersetzen und eine Binärdatei in den CH32V003 übertragen. In der Symbolleiste von **Geany** gibt es 2 Symbole (siehe Bild links). Ein Klick auf das erste Symbol compiliert die Datei (alternativ kann zum Compilieren auch die Taste F8 gedrückt werden), ein Klick auf das zweite Symbol flasht den Chip (alternativ kann zum Flashen die Taste F9 gedrückt werden).

CIDE

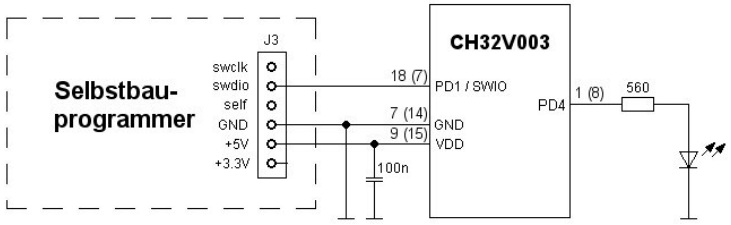


CIDE ist ein Kommandozeileneditor der sehr stark an die IDE der alten Borland-Produkte angelehnt ist und dieselben Fenstertechniken wie diese verwendet. Mit **CIDE** können beliebig viele Quelltexte gleichzeitig editiert werden, Fenster vergrößert, verkleinert oder verschoben werden.

Ebenso wie **Geany** ist es möglich, aus dem Editor heraus eine Datei zu compilieren oder zu flashen. Dieses kann entweder über die Menüzeile oder, im Gegensatz zu **Geany**, mit der Taste F9 für compilieren, der Taste F8 für flashen erfolgen

Hallo Welt

```
Pinnummer sind Nummer für TSSOP-20 Gehäuse,
Nummern in Klammern sind Pinnummern für SOP16 Gehäuse
```



Ein traditionelles „Hallo Welt“-Programm für Mikrocontroller ist eine blinkende Leuchtdiode. Mit dieser Tradition wird auch hier nicht gebrochen.

Verbinden sie wie im Schaltbild links den Programmer mit dem Mikrocontroller, evtl. auch auf einem Steckbrett.

Starten sie **Geany** (wahlweise auf der Konsole auch **CIDE**) und laden sie dort das Programm blink.c im Ordner blink. Drücken sie die Taste F8 oder klicken wahlweise auf das Icon zum Compilieren eines Programms. Nach Beendigung des Compiler- Linkergangs betätigen sie die Taste F9 oder wahlweise ein Klicken auf das Icon zum Flashen.

Die LED sollte nun blinken !

Viel Spaß beim Ausprobieren der anderen Beispielprogramme und beim Experimentieren mit CH32V003

Die Sache mit dem Makefile

Im Text wurde bereits hingewiesen, dass beim Erstellen einer Software das sogenannte Makefile Verwendung findet. Hier soll jetzt hinsichtlich einer Programmerstellung für CH32V003 erläutert werden, wie die aus dem Archiv entpackten Programme mit einem Makefile zu einem lauffähigen Binary compiliert und geflasht werden.

Grundsätzlich gibt es zu jedem einzelnen Projekt ein eigenes Makefile. Makefiles können so geschrieben sein, dass sie zusätzlich zum Aufruf mittels **make** ein weiteres Argument beinhalten können. Im hier vorliegenden Fall kann ein Makefile mit folgendermaßen aufgerufen werden:

- **make**
- **make all**

  make / make all (beide Aufrufe bewirken dasselbe) compiliert alle zu einem Programm gehörenden Quelldateien und linkt diese zu einem Programm mit dem Format **.elf, .hex** und **.bin** zusammen, welches hier ein auf einem CH32V003 lauffähiges Programm darstellt. Je nach Programmer wird einer dieser Dateien benötigt um damit einen CH32V003 flashen zu können. Der Selbstbauprogrammer verwendet das .bin Format

- **make flash**

  make flash programmiert einen Controller mit einem mit make zuvor erstellen .bin Programme

- **make clean**

  make clean löscht alle die Dateien, die mit make / make all erstellt werden. Das ist vor allen Dingen dann hilfreich, wenn eine Datei die nicht die Main-Funktion des Programms beinhaltet bearbeitet worden ist. Durch das Löschen aller Dateien wird sicher gestellt, dass bei einem Compiler- / Linkvorgang die editierte Quelldatei ebenfalls neu übersetzt wird.

Hinweis: make ist ein im Linux vorhandenes Programm das bei Aufruf die Angaben, die in **Makefile** gemacht werden auswertet und ausführt.

Da die Syntax eines Makefile sich bisweilen (zumindest für den Autor) sehr kryptisch anmutet und er bisweilen auch nachlesen muß wie genau etwas funktioniert, wurde das Makefile „aufgetrennt“. Im Makefile selbst, welches in jedem Projektordner liegen muß das compiliert werden soll, müssen nur Angaben über den Projektnamen, zusätzlich zu verwendenden Quelldateien und den zu verwendenden Programmer gemacht werden (die zu machenden Angaben hier sind sehr überschaubar). Dieses Makefile bindet (inkludiert) eine Datei **makefile.mk** im übergeordneten Verzeichnis (das Stammverzeichnis CH32V003) ein. Jedes Projekt inkludiert diese Datei, deshalb sollte diese Datei, wenn überhaupt, nur mit sehr großer Vorsicht bearbeitet werden. **makefile.mk** stellt hier den funktionalen Teil des Compilierens dar und erstellt CH32V003 Programme für ein System mit 48 MHz interner Taktfrequenz und schaltet mit Systemstart alle GPIO Anschlüsse ein.

Für ein neues Projekt ist es eine gute Idee, einen neuen Ordner anzulegen und das Makefile aus einem bestehenden Projekt in diesen neuen Ordner zu kopieren.

Aufbau eines Makefile für CH32V003 an einem Beispiel

Am Beispiel für das Programm uart_demo (im gleichnahmigen Verzeichnis) soll ein Makefileaufbau erläutert werden:

```
############################################################
#
#                         Makefile
#
############################################################

PROJECT       = uart_demo

# hier alle zusaetzlichen Softwaremodule angegeben
SRC           = ../src/uart.c
SRC          += ../src/my_printf.c
SRC          += ../src/term_colpos.c

INC_DIR       = -I ./ -I ../include

# Programmerauswahl
# fuer USE_ARDULINK = 1 wird der Arduinoprogrammer verwendet, in diesem
#                       Fall muss zusaetzlich der serielle Port angegeben werden
# fuer USE_ARDULINK = 0 wird WCH-LINKE verwendet wird die
#                       serielle Portangabe hier ignoriert

PROGPORT      = /dev/ttyUSB0
USE_ARDULINK  = 0

include ../ch32v003.mk
```

Ein Makefile arbeitet sich ähnlich einer Programmiersprache mit Variable / Konstanten (der Autor weiß nicht, ob dieses die korrekte Bezeichnung ist aber glaubt, dass das verständlich ausgedrückt ist).

In diesem „Benutzerteil“ des Makefile gibt es Variable, die von der später inkludierten Datei ../ch32v003.mk ausgewertet werden. Diese sind:

- **PROJECT**
- **SRC**
- **INC_DIR**
- **PROGPORT**
- **USE_ARDULINK**

Für **PROJECT** ist der Name der Datei OHNE die Erweiterung .c anzugeben, die die Main-Funktion des Programms beinhaltet. Von daher ist es also zwingend vorgegeben, dass diese Datei zwingend auf dem Datenträger mit der Namensendung .c gespeichert vorliegt. Hier wird also das Programm uart_demo.c compiliert und gelinkt werden.

Mit **SRC** werden alle zusätzlichen Quelldateien angegeben, die übersetzt und in das Programm mit eingebunden werden sollen. Eine erste einzubindende Datei wird nur mit einem „=“ Zeichen zugewiesen, jede weitere einzubindende Datei mit „+=“.

Im vorliegenden Fall werden also im übergeordneten Verzeichnis und dort im Verzeichnis src die Dateien uart.c, my_printf.c und term_colpos.c mit eingebunden. Die Angaben der Dateinamen sind MIT der Dateinamensendung .c anzugeben.

Hier ist jetzt zu erkennen, dass im übergeordneten Verzeichnis ein Verzeichnis src existiert, in dem alle zusätzlich vorhandenen Quelldateien gespeichert sind.

Mit **INC_DIR** wird der Suchpfad angegeben, nachdem der Compiler einzubindende Dateien, meistens Header, suchen soll. Im vorliegenden Beispiel sucht der Compiler also zuerst in dem Projektverzeichnis (-I ./) nach der Datei, wird er dort nicht fündig, sucht er im übergeordneten Verzeichnis und dort im Verzeichnis include (-I ../include). Wird der Compiler im Projektverzeichnis fündig, wird eine eventuell im ../include Ordner ignoriert. Das Speichern eines Headers im Projektordner macht dann Sinn, wenn bspw. GPIO-Pins, die Hardware steuern nur für dieses eine Projekt gelten soll.

USE_ARDULINK gibt an, welcher Programmer bei einem Flash-Vorgang zu verwenden ist. Mit der Benutzung des Selbstbauprogrammers oder eines originalen WCH-LinkE ist hier der Wert 0 anzugeben. Eine 1 gibt an, dass der Arduino basierende Programmer (der zum Flashen des Selbstbauprogrammers benötigt wurde) zu verwenden ist. In aller Regel wird man diesen ARDULINK jedoch nicht verwenden, sobald eben ein deutlich besserer Programmer vorliegt.

PROGPORT gibt an, welcher Anschluss verwendet werden soll, sollte der Arduino-Programmer zum Einsatz kommen. Ist in **USE_ARDULINK** der Wert 0 angegeben hat die Angabe in PROGPORT keinerlei Wirkung.

Quelldateien für den CH32V003

Um mit den CH32V003 programmieren zu können sind dem Archiv einige (wenige) Sourcedateien beigefügt, die ein Kennenlernen deutlich erleichtern, und werden hier beschrieben. Eine der wichtigsten Quelldatei besteht lediglich aus einem Header:

ch32v003_gpio.h

Der Autor ist ein großer Freund davon, Hardwaredeklarationen und -definitionen in einem Main-Programm zu „verstecken“, damit die Funktion eines Hauptprogramms klarer und strukturiert bleibt. Eine der wichtigsten Funktionen im Umgang mit Mikrocontrollern ist die Möglichkeit, frei programmierbare Anschlußpins, die als sogenannte GPIO (general purpose input output) bezeichnet werden, schalten zu können. Je komplexer ein Mikrocontroller ist, umso größer sind in aller Regel Funktionsaufrufe diesen zum Einen zu initialisieren und zum Anderen einen solchen Pin anzusprechen um ihn ein bzw. aus zu schalten. Aus diesem Grund existiert für den CH32V003 (und beim Autor für jeden anderen Mikrocontroller auch) eine Headerdatei, die einen GPIO einheitlich ansprechen kann und hier alle verfügbaren GPIO‘s nach dem gleichen Schema funktionieren. Für das Initialisieren eines GPIO‘s existieren Makros für die wichtigsten Betriebsarten:

- GPIO als Ausgang (es kann vom Programm aus ein Pin gesetzt bzw. gelöscht werden)
- GPIO als Eingang mit eingeschaltetem PopUp- / PopDownwiderstand (intern im Chip geschaltetem Widerstand gegen +Vdd oder GND)
- GPIO als Eingang ohne eingeschaltetem PopUp- / PopDownwiderstand (dieser Zustand wird auch als „float“ bezeichnet, ist aber nicht zu verwechseln mit dem Variablentyp float)

Für das Ansprechen eines GPIO‘s gibt es Makros für:

- das Setzen eines GPIO-Pins auf logisch 1
- das Löschen / reseten eines GPIO-Pins auf logisch 0
- das Einlesen des digitalen logischen Pegels, der an einem GPIO anliegt.

Für jeden einzelnen GPIO-Pin die der CH32V003 besitzt wurden Makros deklariert, die alle demselben Schema folgen.

Schemata für die GPIO Anschlüsse

- pinbezeichnung_output_init();
- pinbezeichnung_set();
- pinbezeichnung_clr();

- pinbezeichnung_input_init();
- pinbezeichnung_float_init();
- is_pinbezeichnung();

Hierbei ist „pinbezeichnung“ durch den Anschlußnamen des Pin‘s in Großschreibweise zu ersetzen. Soll bspw. der Portpin PD4 als Ausgang benutzt werden, so kann dieser mit

```
PD4_output_init();
```

als Ausgang initialisiert werden.

Ein

```
PC3_input_init();
```

initialisiert den Anschluss PC3 als einen Eingang mit angeschlossenem internen PullUp- oder PullDown-Widerstand.

Beispielprogramm für die Benutzung von GPIO‘s

```
#include "ch32fun.h"
#include "ch32v003_gpio.h"


#define led_init()              PD4_output_init()
#define led_set()               PD4_set()
#define led_clr()               PD4_clr()

#define key_init()              PC4_input_init()
#define key_pullup()            PC4_set()
#define is_key()                (is_PC4())


int main(void)
{
  SystemInit();

  led_init();
  key_pullup();
  key_init();

  while(1)
  {
    if (is_key()) { led_set(); }
              else { led_clr(); }
  }
}
```

In diesem Beispielprogramm wird angenommen, dass an PD4 eine Leuchtdiode gegen GND und ein Taster ebenfalls gegen GND geschaltet ist. Das Makro key_pullup() schaltet durch das Setzen des Portpins bei Funktion als Eingang einen internen PullUp-Widerstand gegen +Vdd. Hierdurch wird beim Lesen des Tastenpins an PC4 bei einem unbetätigten Taster eine logische 1 zurück geliefert und die LED leuchtet bei unbetätigtem Taster (bei betätigtem Taster wird eine 0 zurück geliefert und die LED ist aus).

Wollte man, dass die LED bei betätigtem Taster leuchtet, müßte die Deklaration für is_key lauten (man beachte die Invertierung durch das „!“):

```
#define is_key()                (!is_PC4())
```

systick.h / systick.c

CH32V003 besitzt einen sogenannten Systemticker. Im Prinzip ist dieses ein Timerinterrupt, der ein Programm periodisch unterbricht um den Programmteil im Interrupthandler auszuführen. Im hier vorliegenden Fall hat systick eine einzige Funktion, die vom Benutzer aufrufbar ist:

systick_init

Prototyp: `void systick_init(void);`

Durch Aufruf von systick_init wird der Systemticker so initialisiert, dass der Interrupthandler hierfür jede Millisekunde aufgerufen wird. Innerhalb dieses Interrupthandlers werden Varbiablen durch den Systemticker gezählt, auf die der Benutzer, da global, innerhalb seines Programms Zugriff hat. Diese sind:

```
volatile uint32_t systick_millis;
volatile uint32_t system_zsec;
volatile uint32_t system_halfsec;
volatile uint32_t system_sec;
```

Während die Variablen **system_millis** (jede Millisekunde) und **system_sec** (jede Sekunde) bis zu einem Überlauf hochgezählt werden (was im Falle von system_sec nie passieren wird, denn 2^{32} Sekunden sind

über 130 Jahre) zählen die Variable **system_halfsec** alle 1/10 Sekunde periodisch nur von 0 bis 9. **system_halfsec** toggelt im Halbsekundentakt zwischen 1 und 0 hin und her:

```
#include "ch32fun.h"
#include "ch32v003_gpio.h"
#include "systick.h"

#define led_init()              PD4_output_init()
#define led_set()               PD4_set()
#define led_clr()               PD4_clr()


int main(void)
{
  SystemInit();

  led_init();
  systick_init();

  while(1)
  {
    if (system_halfsec) { led_set(); }
                   else { led_clr(); }
  }
}
```

uart.h / uart.c

uart beinhaltet rudimentäre Funktionen für den Umgang mit einer seriellen Schnittstelle. Die Anschlüsse der seriellen Schnittstelle sind PD5 für transmit data (RxD) und PD6 für receive data (RxD). Zudem kann über einen Define-Schalter eine Funktion zum Einlesen eines dezimalen Integerwertes aktiviert werden.

Vorhanden Funktionen sind:

- `void uart_init(uint32_t baudrate);`
- `void uart_putchar(uint8_t ch);`
- `uint8_t uart_getchar(void);`
- `uint8_t uart_ischar(void);`

wenn aktiviert:

- `int16_t readint(void);`

uart_init

Prototyp: `void uart_init(uint32_t baudrate);`

initialisiert die serielle Schnittstelle mit dem Protokoll 8 Datenbits, keine Parität, 1 Stopbit (8N1).

Übergabe:
baudrate : einzustellende Baudrate

uart_putchar

Prototyp: `void uart_putchar(uint8_t ch);`

sendet ein Zeichen auf der seriellen Schnittstelle

Übergabe:
ch : zu sendendes Zeichen

uart_putchar

Prototyp: `void uart_putchar(uint8_t ch);`

sendet ein Zeichen auf der seriellen Schnittstelle

Übergabe:
ch : zu sendendes Zeichen

uart_getchar

Prototyp: `uint8_t uart_getchar(void);`

wartet solange, bis auf der seriellen Schnittstelle ein Zeichen eingegangen ist und liest dieses ein.

Rückgabe: gelesenes Zeichen

uart_ischar

Prototyp: `uint8_t uart_ischar(void);`

testet, ob auf der seriellen Schnittstelle ein Zeichen eingegangen ist, liest dieses aber nicht ein. **uart_ischar** wartet auch nicht auf ein eingehendes Zeichen und blockiert von daher den Programmablauf nicht.

Rückgabe:
1 : es ist ein Zeichen eingetroffen
0 : eis ist kein Zeichen verfügbar

readint

Prototyp: `int16_t readint(void);`

Da **readint** per default nicht verfügbar ist, muß diese Funktion in der Header-Datei durch einen Define-Schalter freigeschaltet werden:

```
#define readint_enable     1
```

readint liest einen 16-Bit signed Integer auf der seriellen Schnittstelle ein, der Eingabebereich reicht hier allerdings nur von -32767 .. +32767. Eine Korrektur ist mit der Backspacetaste (löschen nach links) möglich.

Rückgabe: eingelesener dezimaler Integer Wert

my_printf.h / my_printf.c

my_printf ist ein in der Funktionalität reduzierter Ersatz für **printf**, speziell zur Benutzung in Verbindung mit Mikrocontrollern. Er macht dann Sinn, wenn es darum geht, (Flash)speicher zu sparen.

Ein vom Standard abweichendes Umwandlungszeichen %k ermöglicht es, Pseudo-Kommazahlen auszugeben.

my_printf benötigt zur Ausgabe irgendwo in den zu einem Programmprojekt gehörenden Dateien (vorzugsweise in der Datei, die die main-Funktion beinhaltet) eine Funktion namens:

```
void my_putchar(char ch);
```

my_printf bedient sich dieser Funktion zur Zeichenausgabe. Soll bspw. ein Programm die Ausgaben auf der seriellen Schnittstelle vornehmen und ist die Software in uart.c eingebunden, dann gibt es dort die Funktion:

```
void uart_putchar(char ch);
```

Um diese Ausgabefunktion nutzen zu können, kann ein einfaches Programm folgendermaßen aussehen:

```
#include "ch32fun.h"
#include "ch32v003_gpio.h"

#include "uart.h"
#include "my_printf.h"

#define  printf      my_printf

/* ------------------------------------------------------------
                        my_putchar

     wird von my_printf aufgerufen. Hierauf gibt my_printf
     den Datenstream aus
   ------------------------------------------------------------ */
void my_putchar(char ch)
{
  uart_putchar(ch);
}

int main(void)
{
  SystemInit();

  uart_init(115200);
  printf("\n\r Hallo Welt \n\r ");
  while(1);
}
```

Eine wichtige Zeile im obigen Programm ist:

```
#define  printf      my_printf
```

Mit diesem Define wird jede printf-Anweisung auf die eigene abgespeckte Version **my_printf** umgeleitet.

Umwandlungszeichen von my_printf

Zeichen	Datentyp und Darstellung
%d	Integer als dezimale Zahl ausgeben int ist 16-Bit Wert auf 8-Bit Systemen int ist 32-Bit Wert auf 32-Bit Systemen
%k	(Pseudo-Kommazahl) Integerwert mit eingesetztem Kommapunkt ausgeben In der globalen Variable printfkomma wird angegeben, an welcher Position ein Dezimalpunkt ausgegeben wird.
%c	Ausgabe eines char als (Ascii)-Zeichen
%s	Zeichenkette (String) ausgeben
%x	Integer als hexadezimale Zahl ausgeben
%%	Ausgabe des Prozentzeichens

Beispiel:

```
#define printf    my_printf

int a, a2, b, c;

printf("\n\r ASCII-Zeichen '%c' = dezimal %d = hexadezimal 0x%x\n\r", 'M', 'M', 'M');

a= 42; b= 13;

// Variable a um 2 Stellen nach links schieben, um spaeter
//2 Nachkommastellen anzeigen zu koennen
a2= a*100;
c= a2 / b;

printfkomma= 2;
printf(" %d / %d = %k", a, b, c);
```

Ausgabe:

```
ASCII-Zeichen 'M' = dezimal 77 = hexadezimal 0x4D
42 / 13 = 3.23
```

term_colpos.h / term_colpos.c

term_colpos beinhaltet Funktionen und Deklarationen, um die Zeichenausgabe in einem seriellen Terminal hinsichtlich Farben und Ausgabeposition steuern zu können.

Für die Ausgabe in einem seriellen Terminal wird zusätzlich die Software aus my_printf.h / my_printf.c und natürlich uart.h / uart.c benötigt.

Die Nummerierung der darstellbaren 16 Farben erfolgt nach dem Schema der (ur)alten EGA-Grafikkarte:

```
#define black           0
#define blue            1
#define green           2
#define cyan            3
#define red             4
#define magenta         5
#define brown           6
#define grey            7
#define darkgrey        8
#define lightblue       9
#define lightgreen      10
#define lightcyan       11
#define lightred        12
#define lightmagenta    13
#define yellow          14
#define white           15
```

Innerhalb von **term_colpos** werden Farbattribute verwendet. Ein Farbattribut bedeutet hier, dass in einem Byte (uint8_t) die Farben für Hintergrund und Vordergrund gespeichert ist.

Hierbei gilt: die oberen 4 Bits (oberes Nibble) repräsentieren die Farbnummer für den Hintergrund, wobei keine Darstellung von intensiven Farben möglich ist.

Die unteren 4 Bits (unteres Nibble) repräsentieren die Textfarbe.

Beispiel:

```
settextattr(0x1e);        // setzt für den Hintergrund die Farbe 1
                          // (blau) mit Textfarbe 15 (0x0e=15=gelb)
```

Funktionen:

void clrscr(void);

löscht den Bildschirminhalt des seriellen Terminals

void gotoxy(uint8_t x, uint8_t y);

Positioniert den Textcursor auf die Position der nächsten Textausgabe im seriellen Terminal.

Übergabe:

x,y : Textkoordinate auf die der Cursor positioniert wird. Die Koordinate 1,1 repräsentiert hierbei die linke obere Ecke

void settextattr(uint8_t attr);

setzt die Farben für Hintergrund und Text. Hierbei repräsentieren die oberen 4 Bit die Hintergrund-, die unteren 4 Bit die Textfarbe. Die Farbnummerierung erfolgt nach dem EGA-Farbschema

Übergabe:
attr : Farbattribut für Hinter- und Vordergrundfarbe

Makros:

textcolor(col)

legt die Textfarbe nach dem EGA-Farbschema fest

bkcolor(col)

legt die Hintergrundfarbe nach dem EGA-Farbschema fest

Beispielprogramm:

```
#include <stdio.h>
#include <stdlib.h>

#include "ch32fun.h"
#include "ch32v003_gpio.h"

#include "uart.h"
#include "my_printf.h"
#include "term_colpos.h"

#define  printf      my_printf

/* ------------------------------------------------------------
                        my_putchar

     wird von my_printf aufgerufen. Hierauf gibt my_printf
     den Datenstream aus
   ------------------------------------------------------------ */
void my_putchar(char ch)
{
  uart_putchar(ch);
}

int main(void)
{
  SystemInit();

  uart_init(115200);

  clrscr();
  textcolor(yellow);
  bkcolor(blue);
  printf("\n\r Hallo Welt \n\r ");
  textcolor(white);
  bkcolor(black);
  while(1);
}
```

i2c_sw.h / i2c_sw.c

Um einen einfachen ersten Zugang zu I2C-Funktionen zu haben, wurden diese mittels Bitbanging realisiert. Dieses erleichtert ein Portieren von einem anderen Mikrocontrollersystem oder auf ein anderes Mikrocontrollersystem ungemein. Es sind lediglich entsprechende Header zu ändern und das Ansprechen von GPIO-Pins zu implementieren.

Bei diesem Bitbangingfunktionen hier geschieht ein Setzen und Rücksetzen eines GPIO-Pins folgenderweise:

- eine logische 1 wird realisiert, in dem der entsprechende Pin als floatender Eingang (Eingang ohne Pull-Up Widerstand) geschaltet wird. Die logische 1 komm durch den externen „I2C-Pullup Widerstand“ von 2,2 kΩ zustande und ist u.a. deswegen wichtig, damit ein Slave die SDA-Leitung bei einem Acknowledge auf 0 legen kann
- eine logische 0 wird realisiert, in dem der entsprechende Pin als Ausgang geschaltet und eine logische 0 angelegt wird.

Zum Verständnis der I2C-Adresse:

Es „streiten“ sich die Gelehrten, ob die Adresse eines I2C-Gerätes nun eine 7-Bit oder eine 8-Bit Adresse besitzt. Technisch gesehen ist dieses sogar klar definiert: Es hat eine 7-Bit Adresse, gefolgt von einem read- (logisch1) oder einem write-Bit (logisch 0). Jetzt kommt das große **ABER**:

Da die 7-Bitadresse um eine Bitposition innerhalb eines Bytes um eine Position nach links verschoben werden muß, kann man innerhalb eines Bytes die Adresse so „interpretieren“, dass das read/write Bit Bestandteil der Adresse ist (was in vielen Fällen den Umgang erleichtert). Am Beispiel des HT16K33A WÄRE die Adresse in Wirklichkeit 0x70, die um eine Stelle nach links geschoben werden muß (was dann 0xE0 ergibt).

0xE0 wäre demzufolge ein Schreibzugriff auf die Adresse 0x70,
0xE1 wäre ein Lesezugrif auf die Adresse 0x70.

Innerhalb dieser I2C Bitbangingfunktionen wird die I2C-Adresse als ein 8-Bit Wert begriffen bei dem das niederwertigste Bit mittels logischer Verknüpfung AND / OR zwecks Zugriff zum Schreiben oder Lesen gelöscht oder gesetzt wird.

Der Header i2c_sw.h

Innerhalb dieser Headerdatei kann folgendes eingestellt werden:

Die GPIO-Pins mit denen eine I2C-Kommunikation vorgenommen wird (hier Default PA1 für SDA und PA2 für SCL)

```
#define i2c_sda_hi()      PA1_float_init()
#define i2c_sda_lo()      { PA1_output_init(); PA1_clr(); }
#define i2c_is_sda()      is_PA1()

#define i2c_scl_hi()      PA2_float_init()
#define i2c_scl_lo()      { PA2_output_init(); PA2_clr(); }
```

Manchen I2C-Devices ist selbst das I2C-Bitbanging noch zu schnell (bspw. dem HT16K33A) und über die aufgeführten defines kann das Timing verlangsamt werden. Der angegebenen Verzögerungszeiten hinter den #define entspricht etwas mehr als 1 Mikrosekunde mal dem angegebenen Wert.

```
#define short_puls     1           // Einheiten fuer einen langen Taktimpuls
#define long_puls      1           // Einheiten fuer einen kurzen Taktimpuls
#define del_wait       1           // Wartezeit fuer garantierten 0 Pegel SCL-Leitung
```

Die Funktionen von i2c_sw

Folgende Funktionen sind in i2c_sw.c definiert:

i2c_master_init

void i2c_master_init(void);

setzt die Pins die für den I2C Bus realisieren auf logisch 1

i2c_sendstart

void i2c_sendstart(void);

erzeugt die Startcondition des I2C Buses

i2c_start

uint8_t i2c_start(uint8_t addr);

erzeugt die Startcondition auf dem I2C Bus und schreibt anschließend eine 8-Bit Deviceadresse auf den Bus

Rückgabe: 0 wenn ein I2C-Slave geantwortet hat (Acknowledge)
1 wenn kein I2C-Slafe geantwortet hat

i2c_startaddr

uint8_t i2c_startaddr(uint8_t addr, uint8_t rwflag);

Diese Funktion ist eine „Kompatibilitätsfunktion“ für eine Startfunktion, bei der die I2C-Deviceadresse als 7-Bit Adresse angegeben wird und das read/write Flag als Argument gesondert mit angegeben wird.

Übergabe: addr : 7-Bitadresse des I2C-Slaves
rwflag : 0 wenn auf den Bus geschrieben werden soll
1 wenn vom Bus gelesen werden sollt

i2c_stop

void i2c_stop(void);

erzeugt die Stopcondition auf dem I2C Bus

i2c_write_nack

void i2c_write_nack(uint8_t data);

schreibt den Wert in ***data*** auf dem I2C Bus OHNE ein Acknowledge einzulesen

i2c_write

uint8_t i2c_write(uint8_t data);

schreibt den in ***data*** Wert auf dem I2C Bus und liest ein Acknowledge ein.

Rückgabe: > 0 wenn Slave ein Acknowledge gegeben hat
== 0 wenn kein Acknowledge vom Slave

i2c_write16

uint8_t i2c_write16(uint16_t data);

schreibt den 16 Bit Wert (2Bytes) in ***data*** auf dem I2C Bus und liest ein Acknowledge ein

Rückgabe: > 0 wenn Slave ein Acknowledge gegeben hat
== 0 wenn kein Acknowledge vom Slave

i2c_read

uint8_t i2c_read(uint8_t ack);

liest ein Byte vom I2c Bus

Übergabe: 1 : nach dem Lesen wird dem Slave ein Acknowledge gesendet
0 : es wird kein Acknowledge gesendet

Rückgabe: vom I2C-Bus gelesenes Byte

to be continued ?

Hier könnte jetzt einige weitere Beschreibungen folgen, auch zu Software, die im Archiv enthalten ist. Jedoch sind die Kommentare in den Quelldateien an sich Erklärung genug (hoffentlich). Zudem ist der Autor unschlüssig, wieviele Beispiele diesem Archiv insgesamt hinzugefügt werden sollen evtl. auch für Hardware, die an den cH32V003 angeschlossen werden kann (bspw. ein SPI-Grafikdisplay).

Hier soll jedoch für das Erste Schluss mit der Beschreibung von Quelldateien sein (der Text soll ja ein „getting started“ in Verbindung mit dem Selbstbauprogrammer sein), damit es in der nächsten Beschreibung um etwas gehen kann, mit dem der Autor selbst nicht arbeitet, von dem er jedoch weiß, dass dieses einige bis viele tun und er fasziniert ist, dass das auch mit dem Billig-Chip CH32V003 funktioniert. Die Rede ist hier von Arduino !

Arduino

Dank eines Arduino-Cores für den CH32V003 auf Github und etwas gutem Zureden ist es sogar möglich, den CH32V003 aus Arduino heraus zu programmieren. Dieser Arduino-Core liegt auf:

https://github.com/openwch/arduino_core_ch32

Leider ist der Autor dieses Cores nicht namentlich genannt und deshalb geht von hier aus ein Dank an den unbekannten Ersteller des Cores.

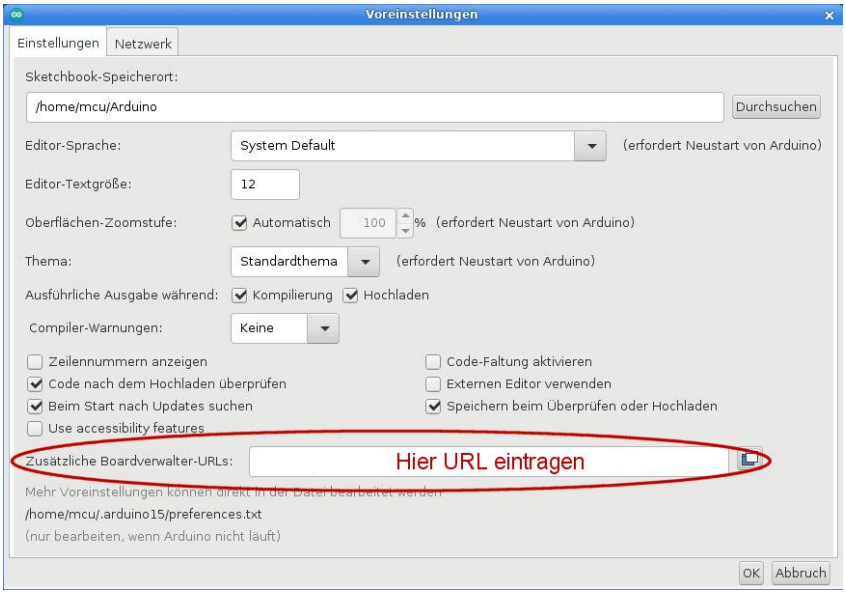
Installieren des Cores für Arduino Legacy IDE 1.8.x

Zum Installieren des Cores starten sie die Arduino-IDE und wählen Sie unter Datei den Punkt „Voreinstellungen“ aus. Ein Fenster öffnet sich und dort müssen sie unter

„Zusätzliche Boardverwalter URLs:“

Die Adresse des Cores für den CH32V003 eintragen:

https://github.com/openwch/board_manager_files/raw/main/package_ch32v_index.json



Nachdem dieses geschehen ist, öffnen sie unter dem Menüpunkt „Werkzeuge => Board => Boardverwalter“ das Dialogfenster des Boardverwalters und geben im Suchfeld „ch32“ ein:

Klicken sie auf „Installieren“. Dieser Vorgang kann etwas dauern, da Arduino nicht nur den Core, sonden die gesamte Toolchain-Kette (inklusive Compiler) installiert.

Nachdem der Core installiert ist, können wir zwar Programme für einen CH32V003 erstellen und kompilieren, dieses jedoch mit Einschränkungen:

- der Core ist für die originalen Evaluationboards gemacht, die auf dem Board einen externen Quarz als Taktgeber haben. Aus diesem Grund initialisiert Arduino das Programm eben für einen externen Quarz. Da dieser an einem blanken Chip nicht vorhanden ist, läuft der Chip nur mit einem reduziertem Takt und u.a. stimmen bspw. Verzögerungszeiten bei „delay“ dann nicht.
- Arduino kann nach dieser Installation den Chip nur mit einem originalen WCH-LinkE Programmer flashen, aber arbeitet nicht mit unserem Selbstbauprogrammer zusammen.

Um diese beiden Einschränkungen zu beheben, reden wir der Arduino-IDE gut zu und patchen diese.

Öffnen sie hierzu auf dem Desktop ihren persönlichen Ordner und aktivieren sie unter „Ansicht“ den Punkt „**Verborgene Dateien**“ anzeigen. Arduino hat seine Konfiguration in einem versteckten Ordner namentlich ./arduino15 abgelegt).

Dort klicken sie sich zu dem Pfad:

/home/benutzername/.arduino15/packages/WCH/hardware/ch32v/1.0.4/

vor (oder geben diesen manuell ein).

Hier müssen 2 Dateien bearbeitet werden: **platform.txt** und **boards.txt**.

Klicken sie hierzu mit der rechten Mousetaste auf **platform.txt** und wählen sie im aufgehenden Fenster „**Mit <<geany**>> öffnen“ aus (oder dem von ihnen präferierten Texteditor). Fügen sie am Ende der Datei folgende Einträge hinzu:

```
## minichlink
tools.minichlink.path={runtime.tools.openocd-1.0.0.path}/bin/
tools.minichlink.cmd=minichlink
tools.minichlink.upload.params.verbose=
tools.minichlink.upload.params.quiet=
tools.minichlink.upload.config=
tools.minichlink.upload.pattern="{path}{cmd}" -w {build.path}/{build.project_name}.bin flash -b
```

Speichern sie die Datei ab und öffnen sie die Datei „**boards.txt**“ ebenfalls in Geany. Dort suchen sie nach dem Text „Upload menu“. Die Datei hat zwar für unterschiedliche Chips mehrere „Upload menu“ Zeilen, dass sie an der richtigen Stelle sind erkennen sie daran, dass die Zeilen mit

```
CH32V00x_EVT.menu.upload_method.
```

beginnen. Geben sie unterhalb von

```
CH32V00x_EVT.menu.upload_method.swdMethod.upload.tool=WCH_linkE
```

folgende Zeilen ein:

```
CH32V00x_EVT.menu.upload_method.minichlink=minichlink
CH32V00x_EVT.menu.upload_method.minichlink.upload.protocol=
CH32V00x_EVT.menu.upload_method.minichlink.upload.options=
CH32V00x_EVT.menu.upload_method.minichlink.upload.tool=minichlink
```

Die IDE von Arduino hat jetzt Menüeinträge zur Auswahl unseres Flasherprogramms für den Selbstbauprogrammer. Das Flasherprogramm selbst muß jetzt in den Ordner:

/home/benutzername/.arduino15/packages/WCH/tools/openocd/1.0.0/bin/

kopiert werden.

Klicken sie sich zu diesem Verzeichnis vor und öffnen sie ein zweites Dateifenster. Wechseln sie in das Verzeichnis, das sie beim Auspacken des Archives gewählt haben und dort in den Ordner **minichlink**.

Von dort kopieren sie die beiden Dateien:

- minichlink (die ausführbare Datei)
- minichlink.so

in den oben aufgeführten Ordner. Arduino kann nun über den Selbstbauprogrammer einen Chip flashen.

Als letzten Schritt muß Arduino mitgeteilt werden, dass es sich beim Mikrocontroller um einen Chip ohne externen Quarz handelt. Hierfür muß im Ordner

/home/benutzername/.arduino15/packages/WCH/hardware/ch32v/1.0.4/system/CH32V00x/USER/

die Datei

system_ch32v00x.c

bearbeitet werden.

In dieser Datei werden Defines deklariert, die die Takteinstellung des Chips vorgeben. Eine der Zeilen lautet:

```
//#define SYSCLK_FREQ_48MHZ_HSI   48000000
```

Diese Zeile ist zu „entkommentieren“ nach (die beiden Schrägstriche entfernen):

```
#define SYSCLK_FREQ_48MHZ_HSI   48000000
```

Die Zeile

```
#define SYSCLK_FREQ_48MHz_HSE   48000000
```

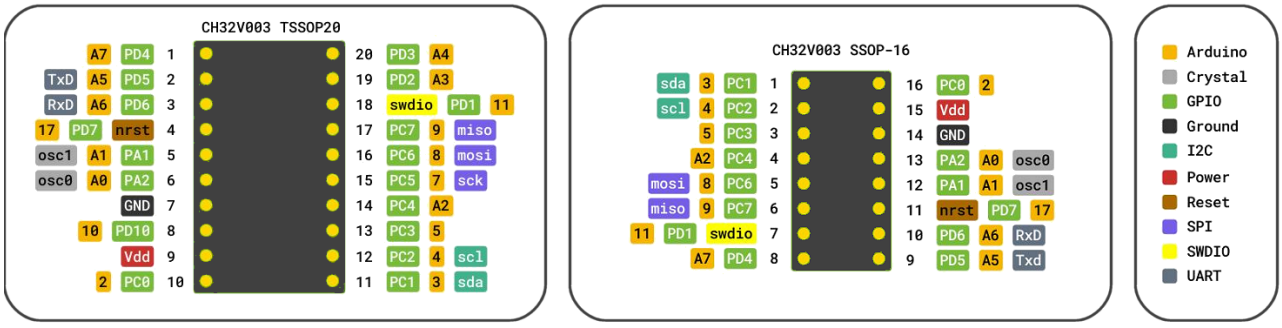
kommentieren aus

```
//#define SYSCLK_FREQ_48MHz_HSE   48000000
```

Speichern sie die Datei ab, beenden sie (falls gestartet) Arduino und starten sie Arduino neu.

Geschafft, sie können nun einen CH32V003 aus Arduino heraus programmieren und flashen.

Damit sie wissen, welcher Pin wo zu finden ist und unter welchem Namen Arduino die Pins kennt, gibt es hier noch das Pinout zu den Chips. Sie können die Pins unter dem Arduinonamen ansprechen (orangene Farbe) oder auch mit dem Namen des Chips, bspw. PD5 (grüne Farbe)



Text von R. Seelig, 22.05.2025