Anleitung

DYPS ONE

Discover **Y**our **P**rogramming **S**kills

http://dyps.gate-ag.ch

GASSMANN
TECHNOLOGIES
GASSMANN TECHNOLOGIES AG
Kreuzsteinstrasse 100
CH-8707 Uetikon

13. Juli 2020
Rev. 200713 00

Dieses Dokument wurde mit LaTeX (2017/01/01) und Tikz (3.0.1a) gesetzt.

Compiled by: pdfTeX-1.40.17

Vorwort

Liebe Leserin, lieber Leser

Es ist bekannt, dass die Kunst des Programmierens nur durch viel Ausdauer und Übung erlernt werden kann. Aus diesem Grund wurde das DYPS ONE entwickelt. Es sollte einen einfachen und gründlichen Einstieg in die Kunst des Programmierens ermöglichen. Dabei steht nicht nur das Programmieren selbst, sondern auch der ganze Prozess im Fokus.

Der Leserin, dem Leser will an dieser Stelle nochmals gesagt sein, dass das Programmieren neben dem lernen der verschiedenen Anweisungen / Strukturen, hauptsächlich durch das Überwinden von Fehlern erlernt wird. Es wird deshalb empfohlen, möglichst viele Fehler zu machen. Dabei trägt auch ein mehrmaliger gleicher Fehler dem Lernfaktor bei. Es soll sich also niemand der lernt zu Programmieren auf Grund eines Fehlers, schlecht fühlen, sondern ganz im Gegenteil.

Manche fragen sich jetzt bestimmt, wie lange es geht, bis das Erlernen einer Programmiersprache abgeschlossen ist. Dabei ist es mit Programmiersprachen so wie mit dem Leben. Man hört nie auf zu lernen! In diesem Sinne wünschen wir viel Ausdauer.

Diese Anleitung dient zur Inbetriebnahme und gleichzeitig als Manual für das DYPS ONE Board. Dies ist die erste Version dieses Dokuments und ist deshalb trotz gebotener Vorsicht höchstwahrscheinlich fehlerbehaftet; die Autoren bitten um Verständnis.

Nun wünschen wir der Leserin, dem Leser viel Vergnügen beim erlernen der Kunst des Programmierens und hoffen, auf ein erfolgreiches und spannendes lernen.

Uetikon, 13. Juli 2020

Das DYPS ONE - Team

Sponsoren

Die hier erwähnten Sponsoren erlauben uns, die (Weiter-)Entwicklung, Produktion und Wartung des DYPS ONE.

Durch ihren Beitrag sichern sie nicht nur die Weiterentwicklung und den Erhalt dieses Boards, sondern ermöglichen den Lehrlingen auch ein kostengünstiges, auf sie zugeschnittenes Mikrocontroller-Board zu kaufen. Dabei sollte es einem Lehrling möglich sein sich in der Kunst des Programmierens weiterzuentwickeln und auch Fehlschläge ohne grosse Konsequenzen zu überwinden.

Falls Ihre Firma hier nicht aufgelistet ist, Sie uns aber gerne Unterstützen möchten, setzen Sie sich einfach mit uns in Verbindung (info@gate-ag.ch). Wir erläutern Ihnen gerne entsprechende Möglichkeiten.

Inhaltsverzeichnis

Kapitel 1

Programmierumgebung

Bevor mit dem Programmieren begonnen werden kann, muss das Entwicklungssystem vorbereitet werden. Das heisst nebst einem Editor[1] sind auch die Toolchain sowie verschiedene Treiber zu Installieren.

Hinweis

Der Editor kann eigentlich frei gewählt werden. Es wird jedoch empfohlen, ein Editor mit Code-Highlighting zu verwenden, da es die Programmierung massiv erleichtert. In dieser Anleitung wird Codeblocks als Editor verwendet, da es nebst einer Projektverwaltung auch noch verschiedene Möglichen für das Übersetzten und Downloaden der Source/Binaries beinhaltet.

Dieses Kapitel befasst sich deshalb mit dem Vorbereiten des Entwicklungssystems. Dabei wird im Detail erklärt wie die sogenannte Toolchain installiert wird.

Achtung

Es wird empfohlen sich genau an diese Anleitung zu halten, ansonsten muss damit gerechnet werden, dass die Installation nicht erfolgreich verläuft und später Probleme beim Arbeiten mit dem DYPS ONE-Board auftreten können!

1.1 Die Toolchain (Werkzeugkette)

Das Programm muss dem Mikrocontroller (Abkürzung: **µC**) in digitaler Form (binärer Code, also nur Nullen und Einsen) übergeben werden. Dies ist die einzige “Sprache”, die der µC versteht. Um den Controller zu programmieren, schreibt man jedoch nicht den binären Code direkt, sondern man benutzt eine sogenannte **Hochsprache**, in unserem Fall die **Programmiersprache C**. Anschliessend wird das Programm mit Hilfe eines **Übersetzungsprogramms**, dem sogenannten **Compiler**, in den binären Code übersetzt. Das C-Programm wird als Text (sog. **Quelltext**) in einer oder mehreren Textdateien (sog. **Quelldateien**) auf dem PC geschrieben. Diese Quelldateien haben die Endung «.c». Um verschiedene Quelldaten zu ”verbinden” werden sogenannte Headerdateien (mit Endung «.h») geschrieben. Der **Compiler übersetzt** die einzelnen Quelldateien zu **Objektdateien**, die dann die binären Codesequenzen enthalten. Ein weiteres Programm, der **Linker**, fügt schliesslich alle Objektdateien (und ggf. auch Librarydatei[2]) zu einer **einzelnen Binärdatei** zusammen. Diese beinhaltet den **ausführbaren Binärcode**, der nun in den µC heruntergeladen werden kann.

[1] Programm um Code zu schreiben/editieren.
[2] Librarydateien sind ebenfalls Ansammlungen von Übersetzten Codesequenzen.

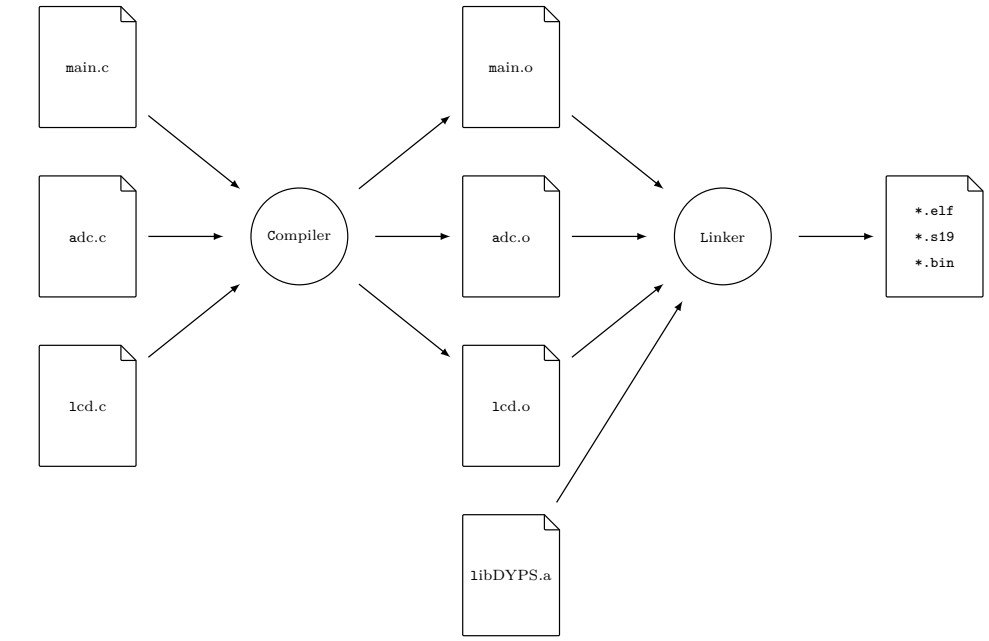


Abbildung 1.1. Übersetzung der Quelldateien zum ausführbaren Code.

In vielen Fällen wird nicht nur der Compiler und der Linker installiert, sondern eine Vielzahl anderer Tools wie zum Beispiel das **Make**, welches diesen **Übersetzungs- und Linkervorgang automatisiert**. Dieses Paket mit all den zusätzlichen Tools wird als **Toolchain** bezeichnet. Eine der gängigsten Toolchains ist die **GNU Toolchain**, welche frei erhältlich ist.

Die GNU Toolchain beinhaltet folgende Pakete:

Make Automatisierung des Kompilier- und Linkervorgangs

GNU Compiler Collection (GCC) Compiler für verschiedene Programmiersprachen

GNU Binutils Sammlung von Programmierwerkzeugen darunter sind der Assembler (as), der Archiv Ersteller (ar), der Linker (ld), der Objektdatei-Kopierer (objcopy) und der Objektdatei-Dumper (objdump) die bekanntesten.

GNU Debugger (GDB) Der Debugger kann verwendet werden, um einen Code schrittweise auszuführen und so Fehler zu finden oder auch den Code besser zu verstehen.

GNU Build System (GNU Autotools) Tools zur Portierung von Quellcode-Paketen auf Unix-Systemen. Es sind folgende tools: Autoconf, Autoheader, Automake, Libtool.

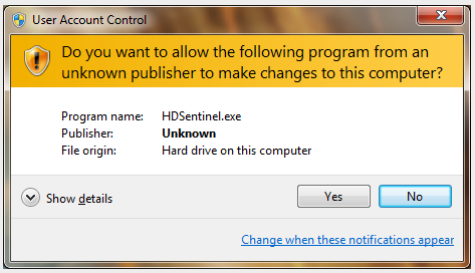
Für sämtliche **ARM-Prozessoren** existiert eine **GNU ARM Toolchain**. Diese ist unter dem Namen **gcc-arm-none-eabi** bekannt. Sie kann unter https://launchpad.net/gcc-arm-embedded heruntergeladen werden, wobei natürlich auf das vom Benutzer eingesetzte Betriebssystem zu achten ist.

Folgend ist die Installationsanleitung unter den verschiedenen Betriebssystemen (Linux, Mac und Windows) erläutert.

1.1.1 Installation unter Windows

Hinweis

Bei Installationen unter Windows wird häufig zu Beginn oder auch während der Installation ein Fenster erscheinen, welches in etwa wie folgt aussieht:



Bei den folgenden Installationen der Programmierumgebung werden diese Fenster auch erscheinen, wichtig ist, dass bei jedem dieser Fenster auf **JA** geklickt wird, ansonsten wird die Installation abgebrochen.

Als erstes wird die ARM Toolchain installiert. Es wird dafür der Installer **gcc-arm-none-eabi-....exe** ausgeführt. Die folgenden Abbildungen führen durch die Installation:

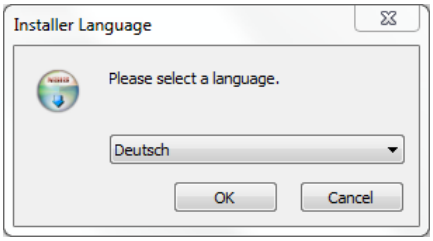


Abbildung 1.2. Schritt 1 → Sprache Deutsch wählen → «OK» anklicken.

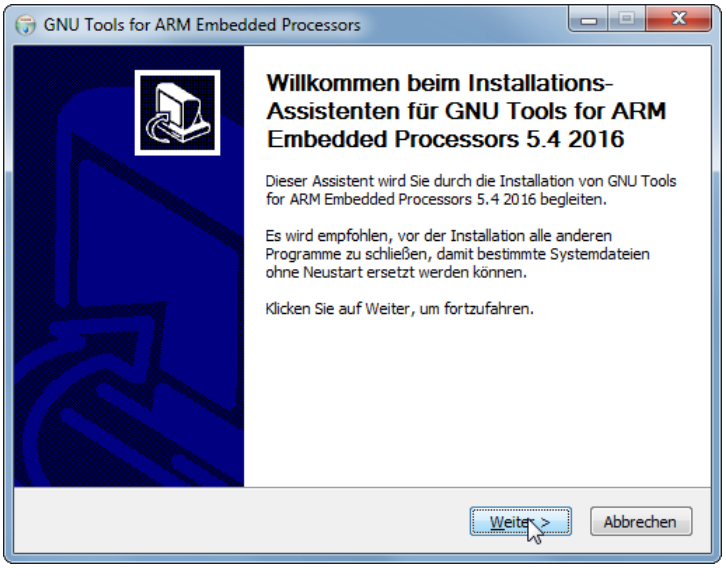


Abbildung 1.3. Schritt 2 → «Weiter» anklicken

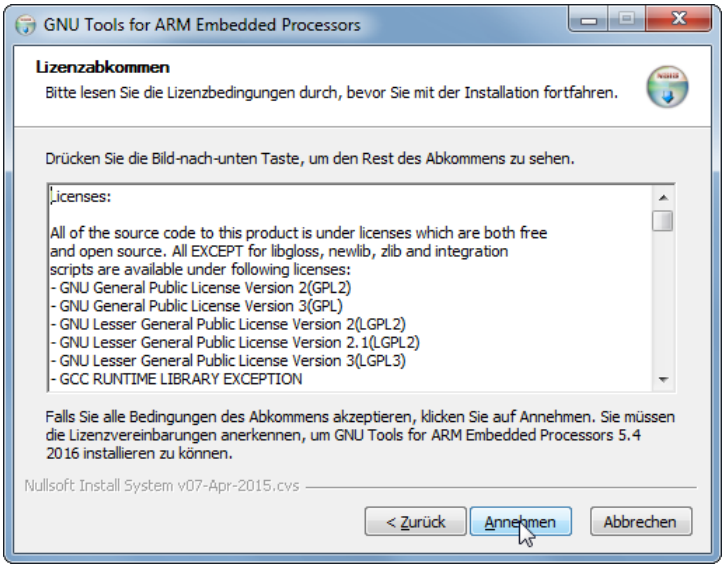


Abbildung 1.4. Schritt 3 → «Annehmen» anklicken

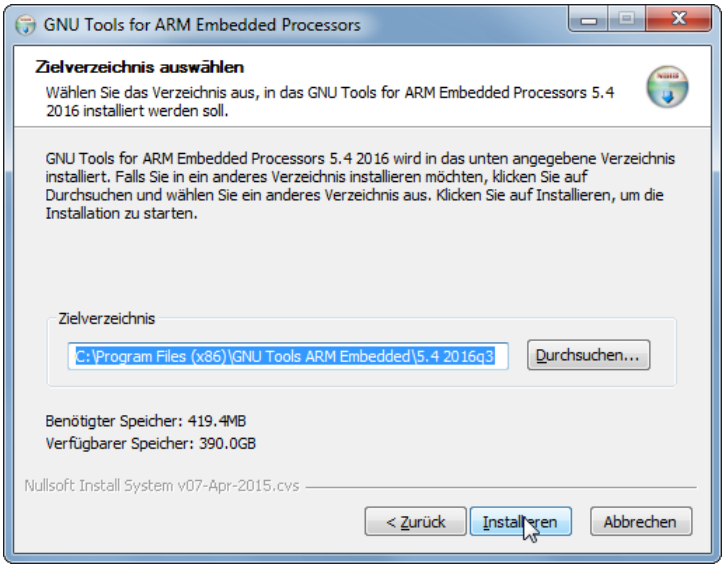


Abbildung 1.5. Schritt 4 → Den Installationspfad merken und «Installieren» anklicken

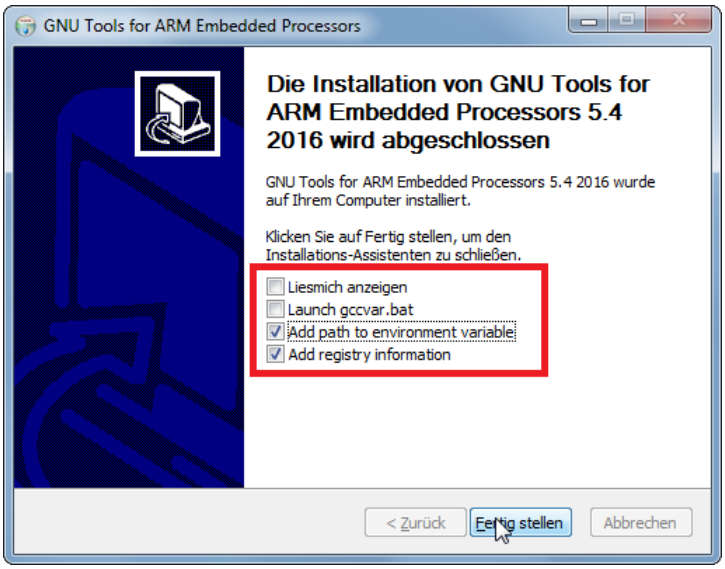


Abbildung 1.6. Schritt 5 → Die Häkchen wie im Bild setzten und auf «Fertig stellen» klicken

Die Toolchain ist damit erfolgreich installiert!

1.2 USBDM

Das USBDM ist das Hardware Debugger Interface welches sich auf dem DYPS ONE befindet (siehe Abschnitt 3.6). Es wird benötigt, um binäre Dateien(also die übersetzten Programme) auf den Mikrocontroller herunter zu laden. Um dabei mit der Hardware zu Kommunizieren, wird eine Software benötigt. Diese ist unter
https://sourceforge.net/projects/usbdm/files/
zu finden.

1.2.1 Installation unter Windows

Für die Installation wird der Installer **USBDM_...._Win.msi** ausgeführt. Die folgenden Abbildungen führen durch die Installation.

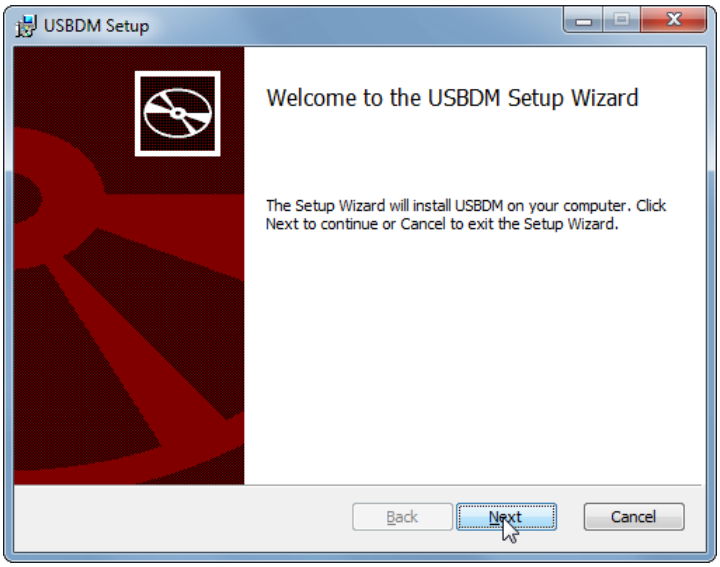


Abbildung 1.7. Schritt 1 → «Next» anklicken

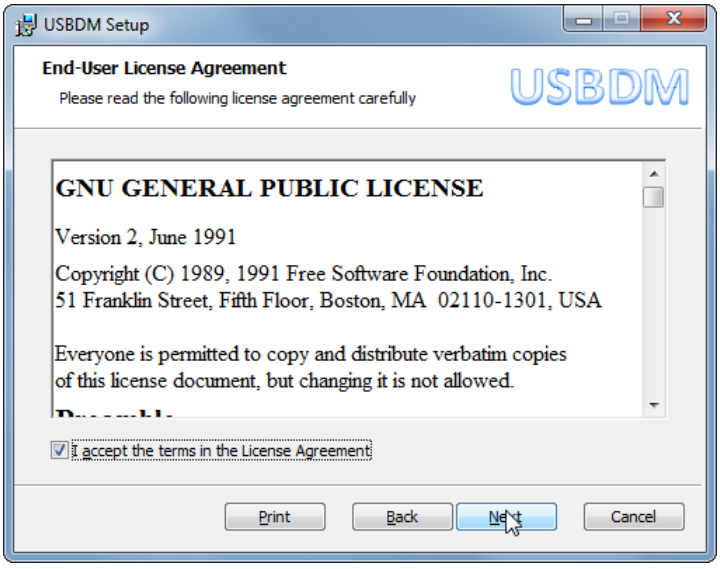


Abbildung 1.8. Schritt 2 → Häkchen setzten und «Next» anklicken

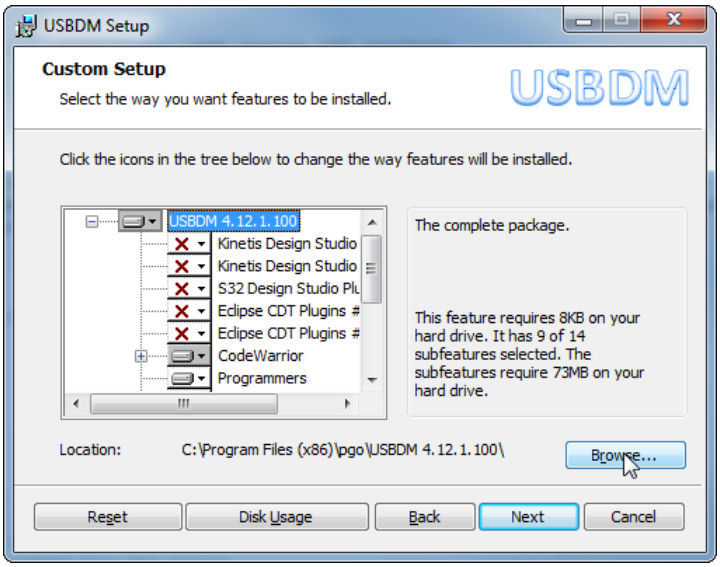


Abbildung 1.9. Schritt 3 → Den Installationpfad (Location:) merken und «Next» anklicken

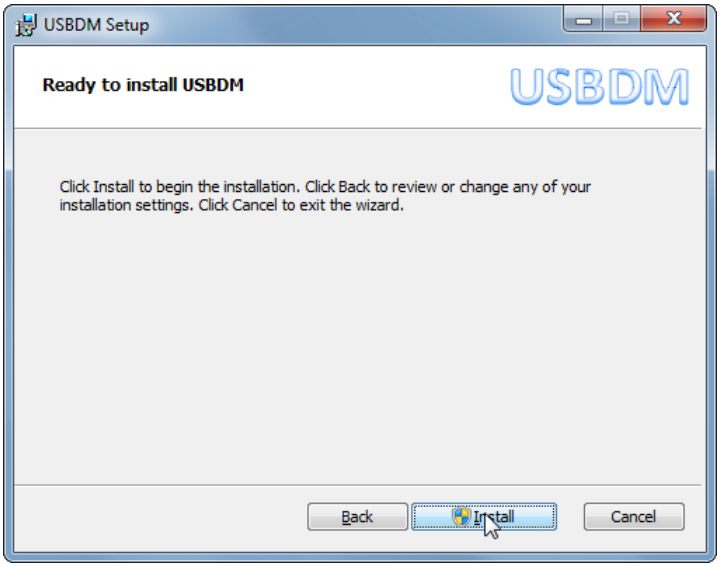


Abbildung 1.10. Schritt 4 → «Install» anklicken

Nachdem die Installation abgeschlossen ist, muss der Installationspfad, in den so genannten System/Umgebungsvariablen eingetragen werden.

1.2.1.1 Umgebungsvariablen unter Windows 9x (< Windows 10)

Bei Systemen unter Windows 10, wird dafür in der Systemsteurung die Systemeigenschaften (System) geöffnet. Es erscheint ein Fenster «Systemeigenschaften». Worin das Register «Erweitert» angewählt wird (siehe Abb. 1.11). Anschliessend wird die Schaltfläche «Umgebungsvariablen...» angewählt. Wiederum erscheint ein Fenster «Umgebungsvariablen» (siehe Abb. 1.12). Hier wird unter den **Systemvariablen** die Linie **Path**(in Abb. 1.12 blau markiert) ausgewählt und auf **Bearbeiten...** geklickt.

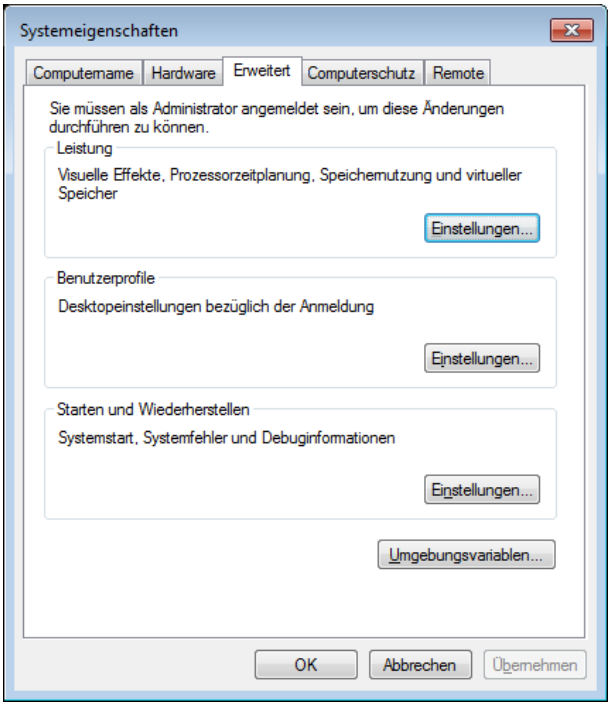


Abbildung 1.11. Systemeigenschaften

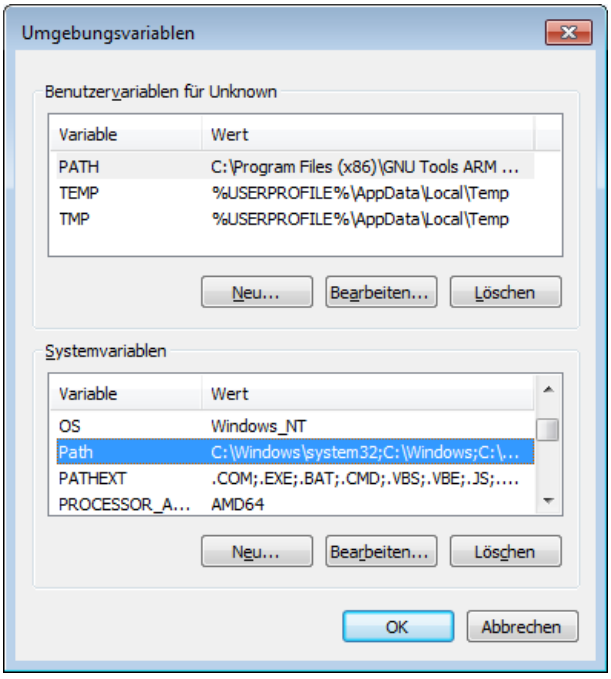


Abbildung 1.12. Umgebungsvariablen

Im neuen Fenster «Systemvariablen bearbeiten» (siehe Abb. 1.13), wird nun unter **Wert der Variablen:** der Path eingefügt.

Achtung

Die einzelnen Pfade müssen mit einem **Semikolon (;) getrennt** werden!

Es wird folglich am Ende des Feldes ein **Semikolon ;** und anschliessend der Pfad eingefügt.

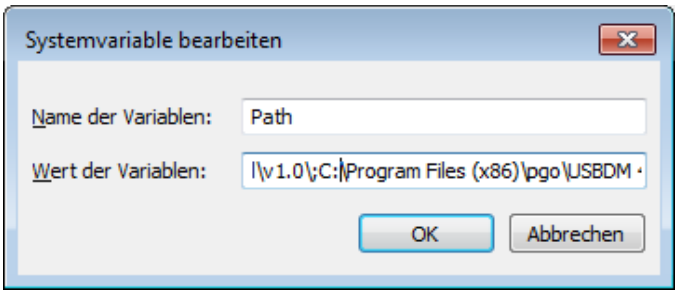


Abbildung 1.13. Neue Umgebungsvariable einfügen

Sobald der Pfad angefügt wurde, können die geöffneten Fenster mit «OK» bestätigt werden. Es ist damit die Installation des USBDMs abgeschlossen.

1.2.1.2 Umgebungsvariablen unter Windows 10

Bei Systemen mit Windows 10 wird am besten nach «Umgebung» gesucht (siehe Abb. 1.14) und der Eintrag «Systemumgebungsvariablen bearbeiten» gewählt. Im neuen Fenster «Systemeigenschaften» (siehe Abb.1.15) wird «Um-

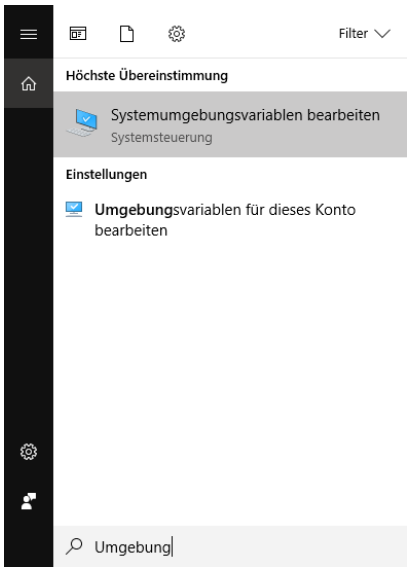


Abbildung 1.14. Suche nach «Umgebung».

gebungsvariablen...» gewählt wodurch das Fenster «Umgebungsvariablen» (siehe Abb. 1.16) geöffnet wird. Unter Systemvariablen wird hier der Eintrag «Path» angewählt und anschliessend auf «Bearbeiten...» gedrückt. Im Fenster «Umgebungsvariablen bearbeiten» (siehe Abb. 1.17) kann nun über die Schaltfläche «Neu» eine neue Variable erzeugt werden welche mit dem Pfad (der USBDM-Installation aus Abbildung 1.9) gefüllt wird. Abschliessend werden alle geöffneten Fester mit «OK» bestätigt.

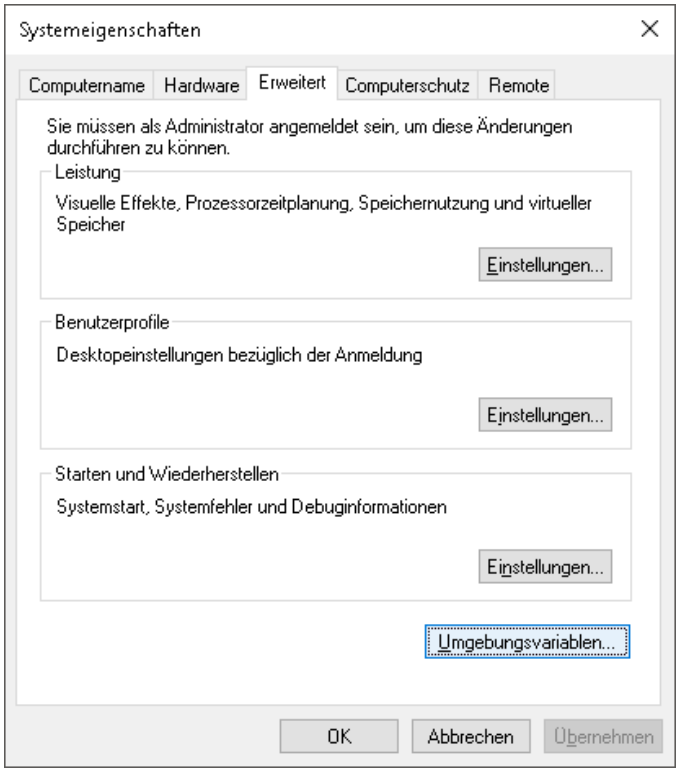


Abbildung 1.15. Systemeigenschaften

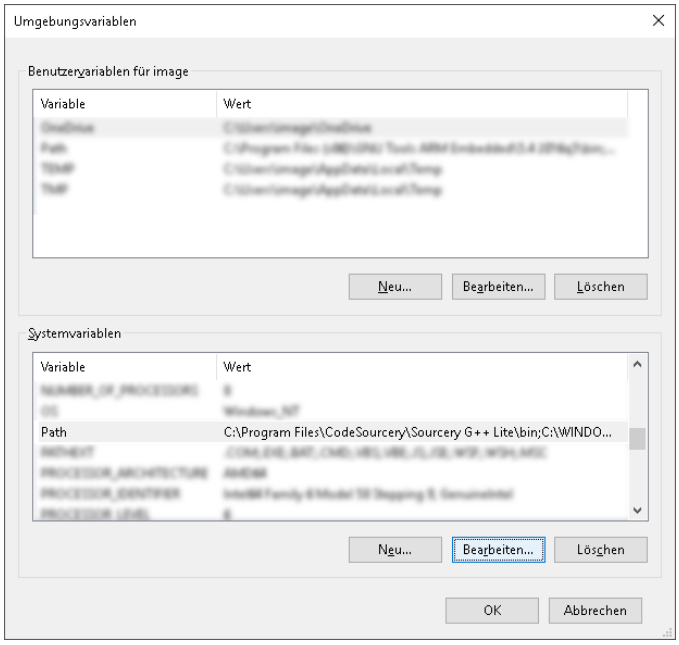


Abbildung 1.16. Umgebungsvariablen

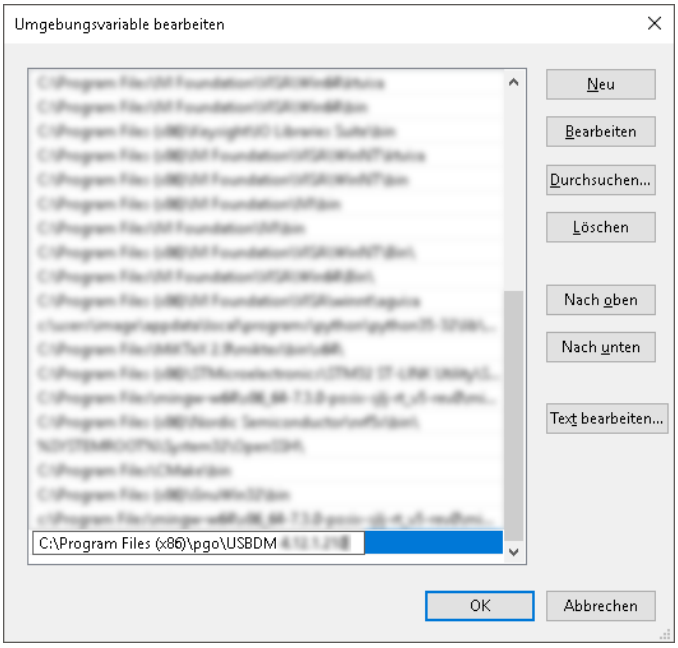


Abbildung 1.17. Umgebungsvariable bearbeiten → den bei Abbildung 1.9 gemerkten Pfad der USBDM-Installation angeben.

1.2.1.3 USBDM Driver

Unter Windows muss zusätzlich ein Treiber für das USBDM installiert werden. Dazu wird der Installer **USBDM_Drivers_....msi** ausgeführt.

Hinweis

Der Installer hierfür ist ebenfalls unter: https://sourceforge.net/projects/usbdm/files/ im Entsprechenden Versions-Ordner unter Drivers zu finden.

Achtung

Es ist wichtig, dass für 64 bit Systeme der x64-Installer und nur für 32 bit Systeme der x86-Installer verwendet wird.

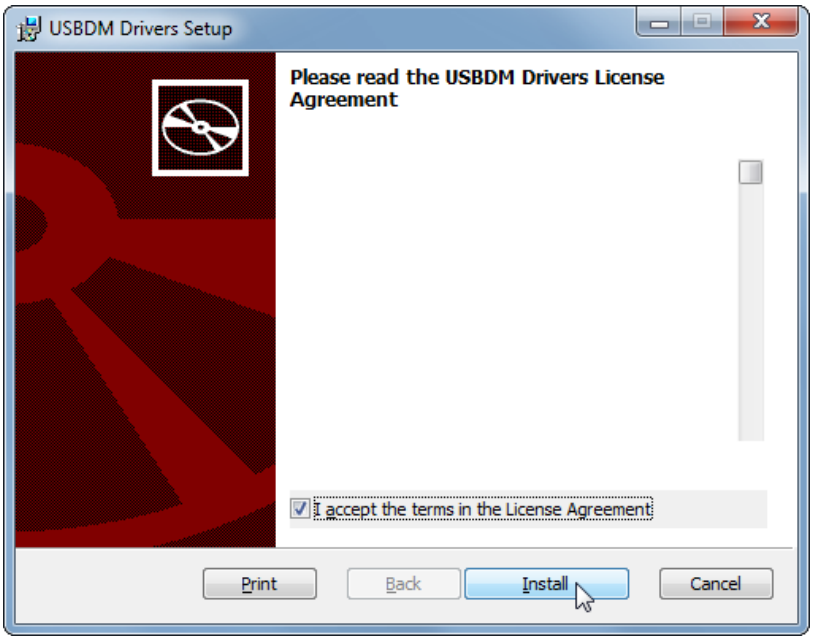


Abbildung 1.18. Schritt 1 → «Install» anklicken

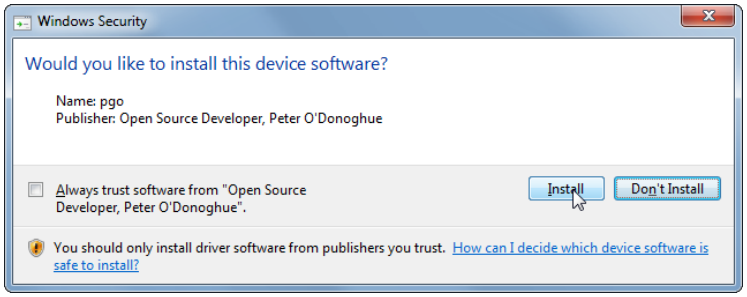


Abbildung 1.19. Schritt 2 → «Install» anklicken

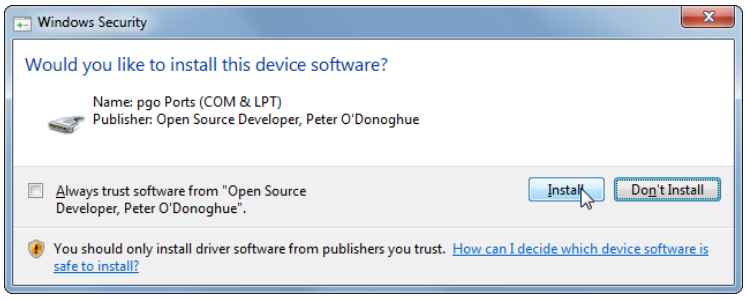


Abbildung 1.20. Schritt 3 → «Install» anklicken

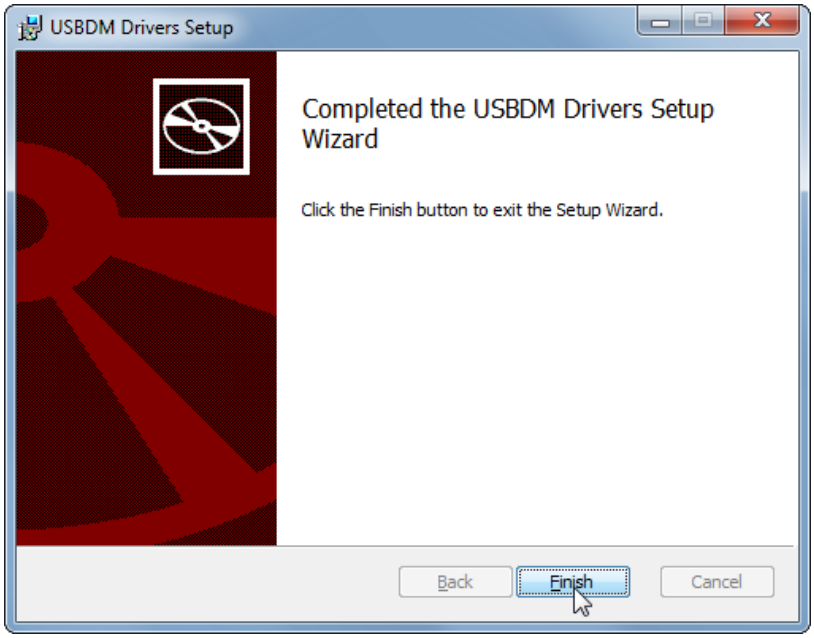


Abbildung 1.21. Schritt 4 → «Finish» anklicken

Der Treiber ist damit installiert.

1.3 Codeblocks

Als letztes wird noch der Editor installiert. Hierzu wird der Installer **codeblocks-...mingw-setup.exe** gestartet.

Abbildung 1.22. Schritt 1 → «Next» anklicken

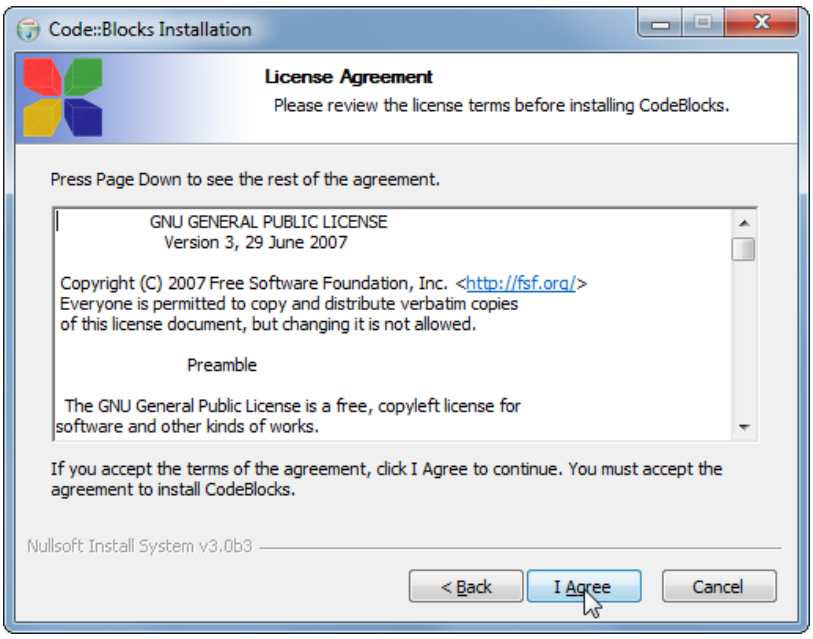


Abbildung 1.23. Schritt 2 → «I Agree» anklicken

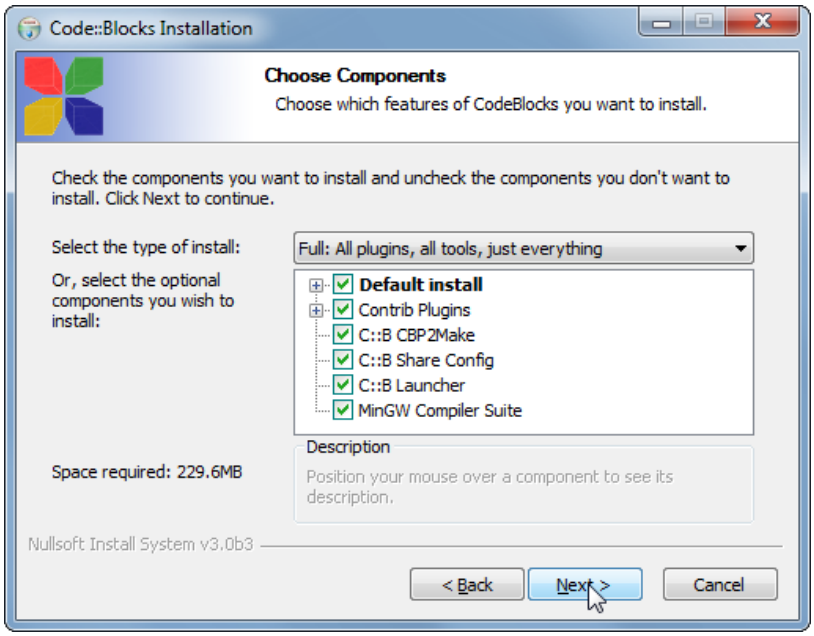


Abbildung 1.24. Schritt 3 → «Next» anklicken

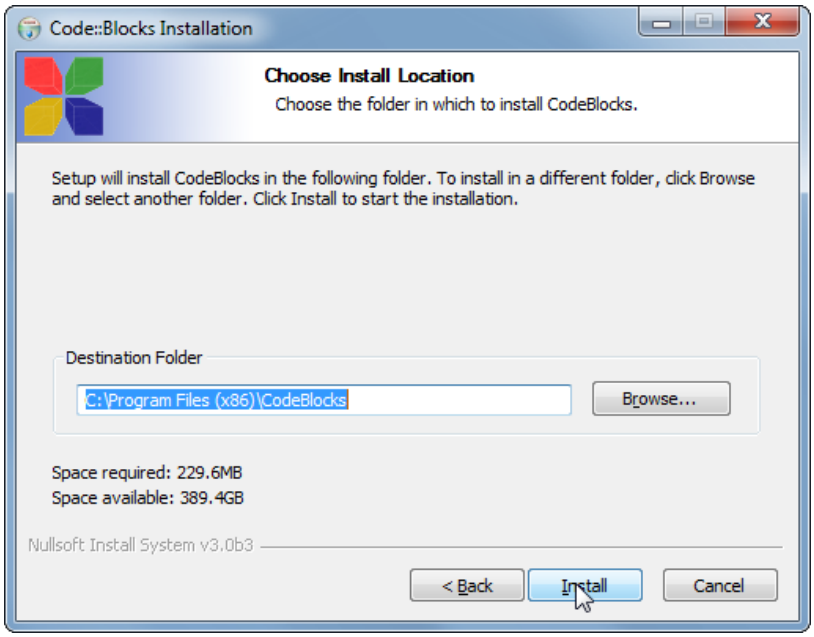


Abbildung 1.25. Schritt 4 → «Install» anklicken

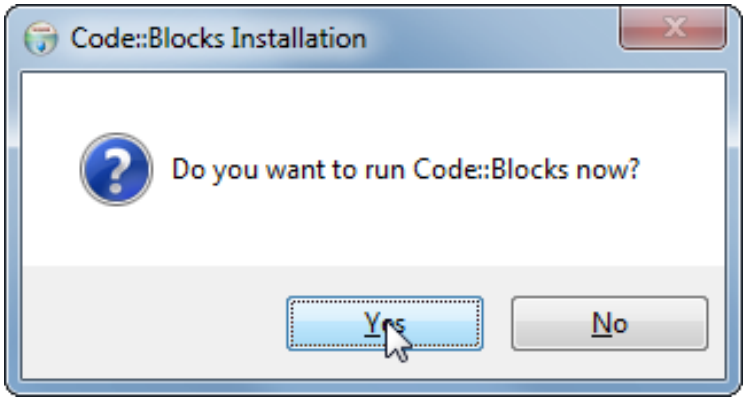


Abbildung 1.26. Schritt 5 → «Yes» anklicken

Codeblocks wird nun gestartet. Bevor jedoch mit dem Programm gearbeitet werden kann, sind noch einige Einstellungen vorzunehmen. Zuvor wird jedoch empfohlen, die Installation noch abzuschliessen (siehe Abb. 1.27 und 1.28).

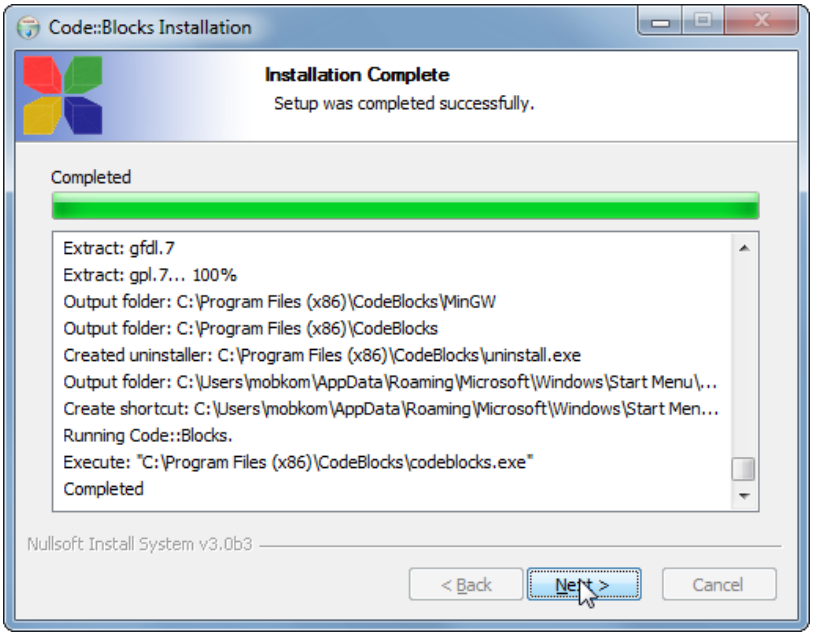


Abbildung 1.27. Schritt 6 → «Next» anklicken

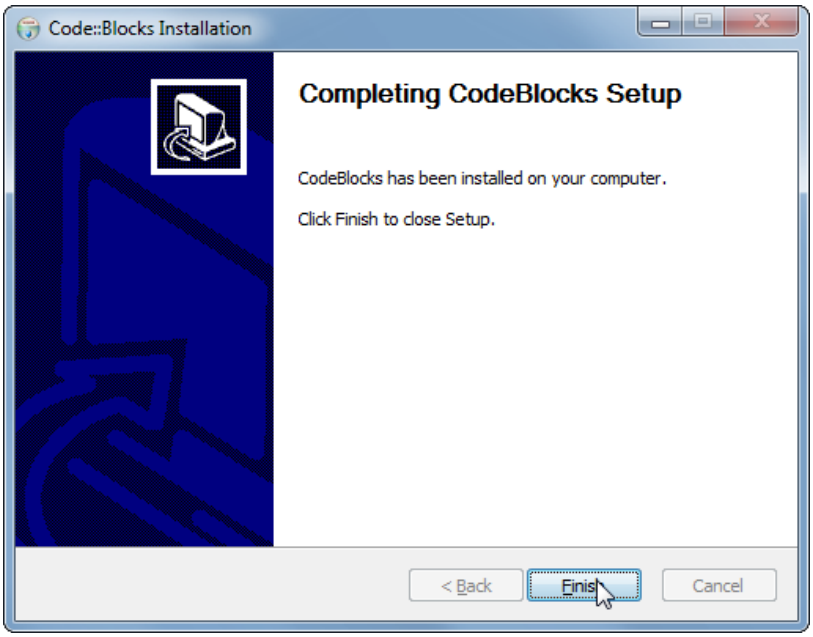


Abbildung 1.28. Schritt 7 → «Finish» anklicken

Beim ersten Aufstarten von Codeblocks erscheint das Fenster «Compiler auto-detection» (siehe Abb. 1.29). In diesem ist der GNU GCC Compiler markiert und es wird mit «OK» bestätigt. Ein weiteres Fenster «File associations» erscheint (siehe Abb. 1.30). Hier kann «**Yes, associate Code::Blocks with C/C++ file types**» gewählt werden, falls eine automatische Öffnung aller C und C++ Dateien mit Codeblocks gewünscht wird. Anschliessend kann auch dieses Fenster mit «OK» bestätigt werden.

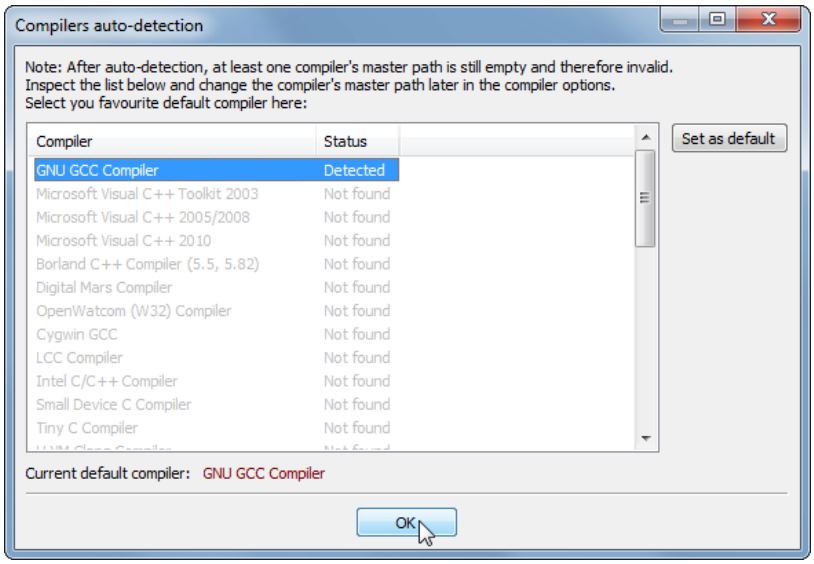


Abbildung 1.29. Schritt 8 → «OK» anklicken

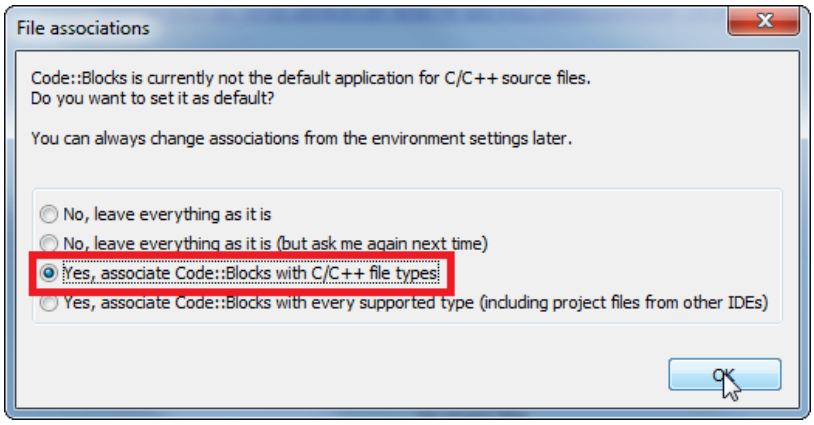


Abbildung 1.30. Schritt 9 → Die gewünschte Datei-Zuordnung wählen und mit «OK» bestätigen

Als nächstes wird im Codeblocks noch der Debugger konfiguriert. Dazu wird im Menü **Settings → Debugger** geöffnet. Es erscheint das Fenster «Debugger settings» auf der linken Seite wird auf « GDB/CDB debugger » geklickt und anschliessend « Create Config » gewählt. Im neuen neuen Fenster « Create config » wird als name **armGDB** gewählt und mit « OK » bestätigt. Auf der linken Seite erscheint nun die neue Config « armGDB ». Diese wird angewählt. Als Executable path wird unter dem Installationspfad der Toolchain (aus Abb.1.5) die Executable « arm-none-eabi-gdb » gewählt. Zudem wird die Konfiguration gemäss Abb. 1.31 vorgenommen.

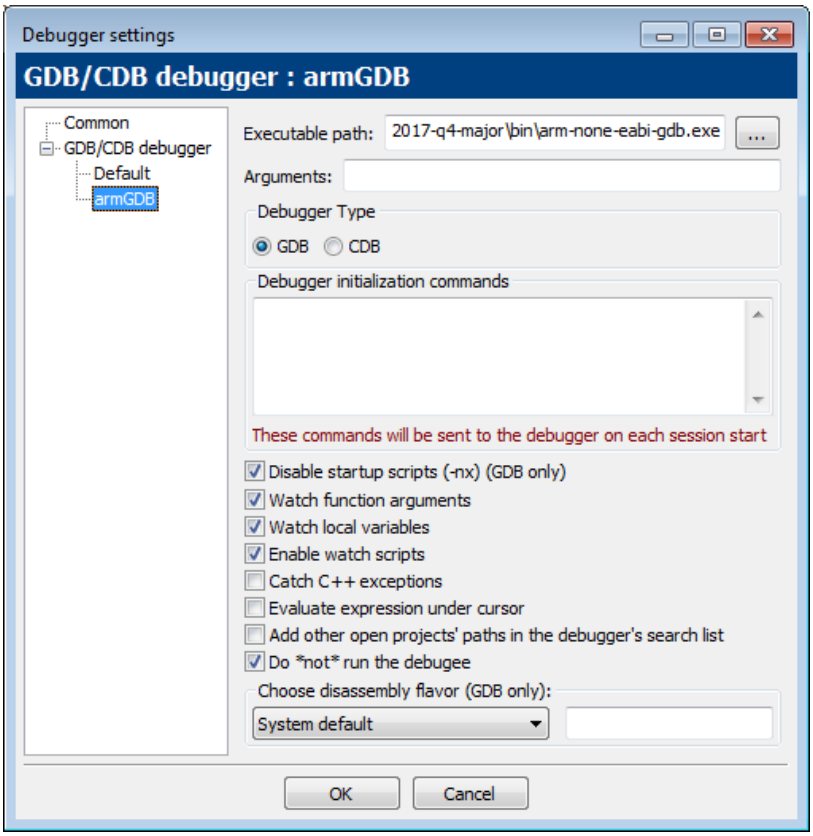


Abbildung 1.31. Schritt 10 → Debugger Konfiguration: Einstellungen gemäss Bild vornehmen und mit « OK » bestätigen.

Nun muss im Codeblocks der Compiler noch konfiguriert werden dazu wird im Menü **Settings → Compiler** geöffnet. Es erscheint das Fenster «Compiler settings» (siehe Abb. 1.32).

Achtung

Vor der Konfiguration ist es äusserst wichtig, dass zuerst der **GNU GCC Compiler for ARM** unter «Selected compiler» gewählt wird.

Anschliessend müssen unter dem Reiter **Toolchain executables** die Felder wie folgt gefüllt werden:

Compiler's Installation directory:	Verweis auf den Installationspfad der Toolchain (aus Abb.1.5) Bsp.: **C:\Program Files (x86)\GNU Tools ARM Embedded\5.4 2016q3**
C Compiler:	arm-none-eabi-gcc.exe
C++ Compiler:	arm-none-eabi-g++.exe
Linker for dynamic libs:	arm-none-eabi-g++.exe
Linker for static libs:	arm-none-eabi-ar.exe
Debugger: GDB/CDB debugger:	GDB/CDB debugger: armGDB
Resource compiler:	
Make program:	arm-none-eabi-nm.exe

Siehe dazu auch Abb. 1.32. Sobald die Einstellungen eingetragen sind, können diese mit einem Klick auf «OK» bestätigt werden.

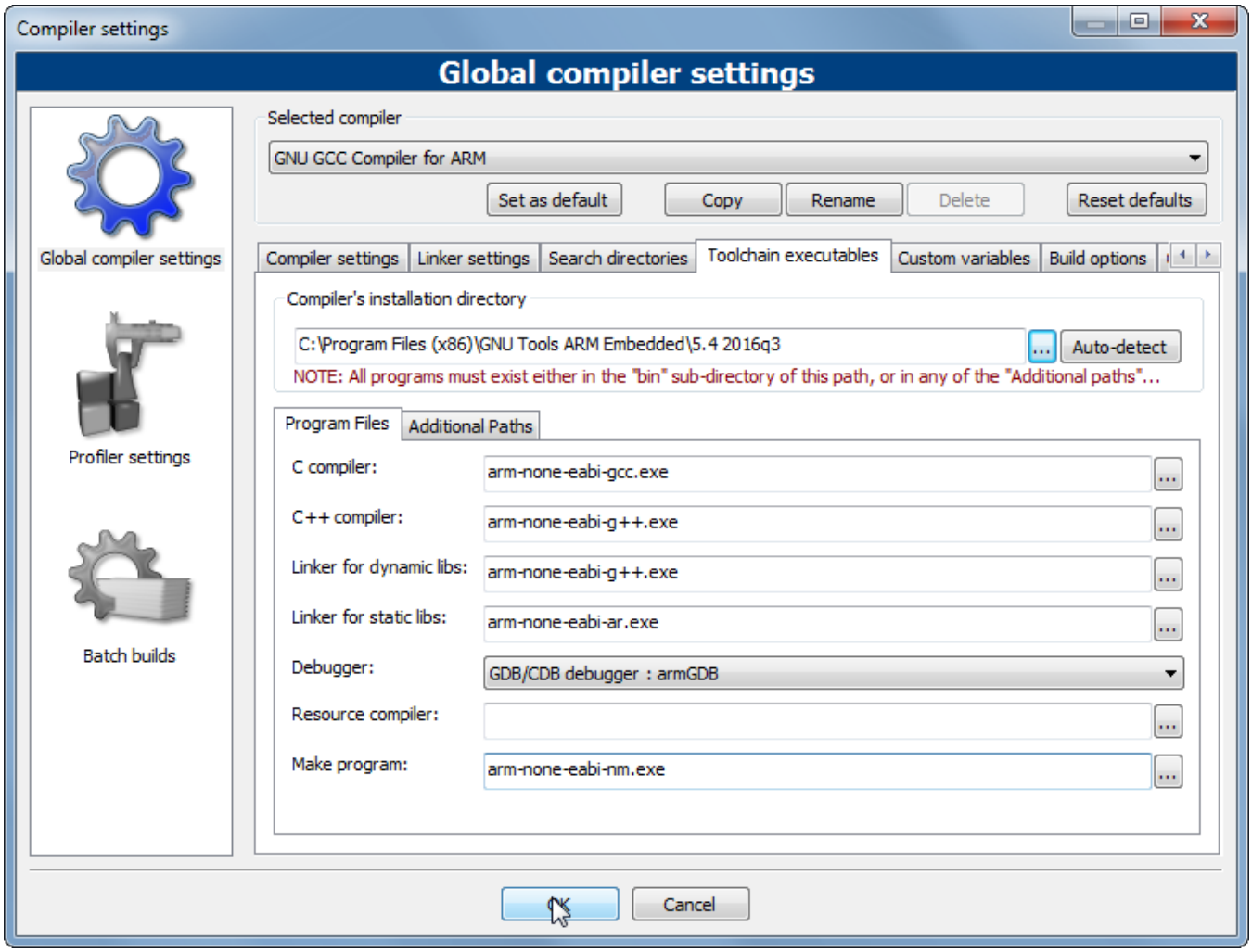


Abbildung 1.32. Schritt 11 → Einstellungen wie im Bild gezeigt vornehmen und mit «OK» bestätigen.

1.3.1 DYPS-Projektwizard

Damit in Zukunft einfach neue DYPS-Projekte erstellt werden können, wird zum Abschluss der Installation noch ein DYPS-Projektwizard ins Codeblocks eingefügt.
Hierzu wird das «DYPS_Wizard.zip» heruntergeladen und unter dem Installationspfad (bei der Standard Installation unter C:\Program Files (x86)\CodeBlocks) zum Ordner \share\CodeBlocks\templates\wizard navigiert. Dort wird der Ordner «dyps» aus dem ZIP-Archiv «DYPS_Wizard.zip» eingefügt. Des weiteren muss in der Datei «config.script» (ebenfalls im Ordner wizard) unter der Sektion «build target wizards» folgende Zeile eingefügt werden (siehe auch Abb.1.33):

```
RegisterWizard(wizProject,     _T("dyps"),          _T("DYPS Project"),          _T("Embedded Systems"));
```

```
RegisterWizard(wizProject,    _T("arduino"),    _T("Arduino Project"),    _T("Embedded Systems"));
RegisterWizard(wizProject,    _T("arm"),        _T("ARM Project"),        _T("Embedded Systems"));
RegisterWizard(wizProject,    _T("dyps"),       _T("DYPS Project"),       _T("Embedded Systems"));
RegisterWizard(wizProject,    _T("avr"),        _T("AVR Project"),        _T("Embedded Systems"));
RegisterWizard(wizProject,    _T("msp430"),     _T("MSP430 Project"),     _T("Embedded Systems"));
```

Abbildung 1.33. Erweiterung der «config.script» Datei, mit dem DYPS-Projekt-Wizard.

Achtung: Administrator

Unter Windows 10 kann es sein, dass für die Bearbeitung der Datei Admin-Rechte erforderlich sind.

Kapitel 2

Projekte erstellen

Dieses Kapitel befasst sich mit dem Erstellen eines neuen DYPS ONE-Projekts. Es werden dazu anschliessend zwei Möglichkeiten erläutert.

Variante 1 Unter Abschnitt 2.1 wird das Erstellen eines μC-Projekts von Grund auf erklärt. Das Vorgehen kann dabei auch für andere μCs verwendet werden, da die Grundlagen die Gleichen sind. Dieses Vorgehen setzt ein Grundverständnis für den Übersetzungsvorgang sowie das Memory-Layout von Programmen voraus. Sind in diesen Gebieten keine Kenntnisse vorhanden, sollten zuerst die Kapitel B und C aus dem Anhang gelesen werden.

Variante 2 Da Variante 1 eher aufwendig und für Einsteiger möglicherweise etwas undurchsichtig ist, wurde ein Projekt-Generator (Wizard) für die IDE Code::Blocks entwickelt. Mit diesem Projekt-Generator (Abschnitt 2.2) reduziert sich das Erstellen eines neuen Projekts auf wenige Klicks. Ein neues Projekt kann somit sehr einfach und innert kürzester Zeit erstellt werden.

Zum Schluss dieses Kapitels wird in Abschnitt 2.3 das Debuggen eines DYPS ONE-Projektes in Code::Blocks beschrieben.

2.1 Ein neues DYPS ONE Projekt erstellen (makefile)

Hinweis: Grundlagen

In diesem Kapitel wird davon ausgegangen, dass die grundlegenden Übersetzungsmechanismen von ”normalen” C-Programmen mit und ohne `makefile` bekannt sind. Ist dies nicht der Fall, so sind diese Grundlagen im Anhang B zu finden. Zudem sollte auch das Memory-Layout von Programmen bekannt sein. Hierzu sind die Grundlagen im Anhang C zu finden.

Im Generellen stellen μC-Hersteller ”Fertig-Entwicklungsumgebungen”[1] für Ihre μC zur Verfügung[2]. Diese beinhalten zwar meistens Startup-Codes, Treiber, Bibliotheken und teilweise sogar ganze OS oder Stacks, was aber leider dazu führt, dass diese SDK’s oft sehr gross und extrem unübersichtlich sind. Zudem wird damit die Entwicklungsumgebung (IDE und Compiler) vorgeschrieben, was auch nicht immer gewünscht wird. Deshalb wird an dieser Stelle ein Beispielprojekt für das DYPS ONE von Grund auf erstellt. Es wird damit gezeigt, welche Komponenten/Informationen für das Ansteuern eines μC benötigt werden und wie diese zusammenhängen.

[1] engl. SDK für Software Development Kits
[2] Beispielsweise TI (Code Composer Studio IDE), ST (MDK-ARM-STM32), Nordic (nRF5 SDK) oder auch Freescale (KDS)

Die folgenden Schritte sind zwar spezifisch auf das DYPS ONE ausgelegt, dennoch kann diese Anleitung auch als Grundlage für die Ansteuerung anderer μC verwendet werden.

2.1.1 Speicherbereiche

Zu Beginn werden die nötigen Informationen über den μC gesucht. Im Falle des DYPS ONE, bei welchem als μC der MK22FN256VLL12 verwendet wird, wird dessen Reference Manual (RM) genauer studiert. Dabei ist besonders die « Memory Map » (siehe Abb. 2.1 bzw. Kapitel *4.2 System memory map* im RM) von Interesse. Diese Tabelle listet die einzelnen Speicherbereiche der MK22-μC-Familie auf.

Table 4-1. System memory map

System 32-bit Address Range	Destination Slave	Access
0x0000_0000–0x07FF_FFFF [1]	Program flash and read-only data (Includes exception vectors in first 1024 bytes)	All masters
0x0800_0000–0x0FFF_FFFF	Reserved	—
0x1000_0000–0x1BFF_FFFF	Reserved	—
0x1C00_0000–0x1FFF_FFFF [2]	SRAM_L: Lower SRAM (ICODE/DCODE)	All masters
0x2000_0000–0x200F_FFFF [2]	SRAM_U: Upper SRAM bitband region	All masters
0x2010_0000–0x21FF_FFFF	Reserved	–
0x2200_0000–0x23FF_FFFF	Aliased to SRAM_U bitband	Cortex-M4 core only
0x2400_0000-0x2FFF_FFFF	Reserved	–
0x3000_0000-0x33FF_FFFF [1]	Program Flash and read-only data	Cortex-M4 core only
0x3400_0000-0x3FFF_FFFF	Reserved	–
0x4000_0000–0x4007_FFFF	Bitband region for peripheral bridge 0 (AIPS-Lite0)	Cortex-M4 core & DMA/EzPort
0x4008_0000–0x400F_EFFF	Reserved	–
0x400F_F000–0x400F_FFFF	Bitband region for general purpose input/output (GPIO)	Cortex-M4 core & DMA/EzPort
0x4010_0000–0x41FF_FFFF	Reserved	–
0x4200_0000–0x42FF_FFFF	Aliased to peripheral bridge (AIPS-Lite) bitband	Cortex-M4 core only
0x4300_0000–0x43FD_FFFF	Reserved	–
0x43FE_0000–0x43FF_FFFF	Aliased to general purpose input/output (GPIO) bitband	Cortex-M4 core only
0x4400_0000–0xDFFF_FFFF	Reserved	–
0xE000_0000–0xE00F_FFFF	Private peripherals	Cortex-M4 core only
0xE010_0000–0xFFFF_FFFF	Reserved	–

Abbildung 2.1. Die Memory Map der MK22-Familie

Zusätzlich werden die Tabellen « Flash Memory Sizes » (siehe Abb. 2.2 bzw. RM *3.5.1.2 Flash Memory Sizes*) und « SRAM sizes » (siehe Abb. 2.3 bzw. RM *3.5.3.1 SRAM sizes*) für die nächsten Schritte benötigt.

Device	Program flash (KB)	Block 0 address range
MK22FN256VLL12	256	0x0000_0000–0x0003_FFFF

Abbildung 2.2. Die Flash Memory Size Tabelle des MK22FN256VLL12. Aus RM Abschnitt *3.5.1.2 Flash Memory Sizes*.

Device	SRAM_L size (KB)	SRAM_U size (KB)	Total SRAM (KB)	Address Range
MK22FN256VLL12	16	32	48	0x1FFF_C000-0x2000_7FFF

Abbildung 2.3. Die RAM Memory Size Tabelle des MK22FN256VLL12. Aus RM Abschnitt *3.5.3.1 SRAM sizes*.

Hierzu einige Punkte zur Erklärung:

ROM Bereich Der ROM/Flash Bereich befindet sich laut Tabelle 2.1 zwischen den Adressen `0x0000_0000` und `0x07FF_FFFF`. Dies würde einer Speicherkapazität von 128 MB entsprechen. Gemäss RM (siehe Abb. 2.2) besitzt der MK22FN256VLL12 aber nur gerade 256 KB Flashspeicher. Aus diesem Grund entspricht der ROM-Bereich lediglich dem fortlaufenden Block ab Adresse `0x0000_0000` bis `0x0003_FFFF`

(256 KB). Von diesen 256 KB sind gemäss RM (siehe Abb. 2.1) die ersten 1024 Bytes durch die exception Vectors[3] und die anschliessenden 16 Bytes durch die Flash Konfiguration[4] belegt.

RAM Bereich Der RAM Bereich ist gemäss Tabelle 2.1 in die zwei Sektoren `0x1C00_0000` bis `0x1FFF_FFFF` und `0x2000_0000` bis `0x200F_FFFF` aufgeteilt. Der genaue Bereich ist abhängig von der Grösse des vorhandenen RAMs. Angaben zur Grösse und zum genaue Bereich des RAMs sind im RM Abschnitt *3.5.3.1 SRAM sizes* oder in Abbildung 2.3 zu finden. Es ist der SRAM_L Bereich 16 KB und der SRAM_U Bereich 32 KB gross, womit diese im Bereich von `0x1FFF_C000` bis `0x1FFF_FFFF` und von `0x2000_0000` bis `0x2000_7FFF` liegen.

2.1.2 Das Linker Script

Für jeden Linkvorgang wird ein Linker Script benötigt, welches das Memory-Layout und die einzelnen Sections darin festlegt (siehe Kapitel C für weitere Informationen zum Thema Memory Layout). Wird beim Linken kein Linker Script angegeben, so wird das Standard-Script[5] des Linkers verwendet. Leider kann hier das Standard-Script aber nicht verwendet werden, da das Memory-Layout μC spezifisch ist. Folglich muss hier das Linker Script entsprechend geschrieben werden und anschliessend mit der Option « `-T` » dem Linker übergeben werden.

Als Erstes wird dazu das Memory-Layout mit den aus Abschnitt 2.1.1 gewonnen Erkenntnissen wie folgt definiert:

```
MEMORY
{
    ROM_VECTORS    (rx) : ORIGIN = 0x00000000, LENGTH = 0x00000400
    ROM_FLASH_CFG  (rx) : ORIGIN = 0x00000400, LENGTH = 0x00000010
    ROM_TEXT       (rx) : ORIGIN = 0x00000410, LENGTH = 256K-0x00000410
    RAM_L          (rw) : ORIGIN = 0x1FFFC000, LENGTH = 16K
    RAM_U          (rw) : ORIGIN = 0x20000000, LENGTH = 32K
}
```

Es wird mit dem `ROM_VECTORS`-Bereich ab Adresse `0x0000_0000` begonnen. Dieser Bereich umfasst 1024 Byte = 1 KB. Entsprechend wird die Länge zu `0x0000_0400`. Es folgen 16 Byte für die Flash Konfiguration (`ROM_FLASH_CFG`) und anschliessend der `ROM_TEXT`-Bereich mit einer Grösse von

$$256\,\text{KB} - 1\,\text{KB} - 16\,\text{Byte} = 255\,\text{KB} \cdot 1024 - 16\,\text{Byte} = 261\,104\,\text{Byte}.$$

Es sind damit die Bereiche `ROM_VECTORS`, `ROM_FLASH_CFG` und `ROM_TEXT` in der Flash-Region (`0x0000_0000` bis `0x0003_FFFF`) enthalten.

Des Weiteren werden die zwei RAM-Bereiche `RAM_L` (`0x1FFF_C000` bis `0x1FFF_FFFF`) und `RAM_U` (`0x2000_0000` bis `0x2000_7FFF`) entsprechend der μC-Spezifikation definiert.

Hinweis: Speicherbereich-Bezeichnungen

Die Bezeichnung `ROM_VECTORS`, `ROM_FLASH_CFG`, etc. sind frei wählbar.

2.1.2.1 Mapping

Im Linker-Script muss weiter definiert werden, welche Input-Section der Source/Object-Dateien in welche Output-Section der Binärdatei (*.elf-Datei) gehören. Man spricht hier vom Mapping. Standardmässig wird der Kompiler (ohne die Verwendung von Standard-Libraries) die folgenden Input-Sections erzeugen:

.text Programmcode

.rodata read-only data (Konstanten)

.data initialisierte globale / statische Variablen

.bss / COMMON uninitialisierte globale Variablen

[3] Die « vector table » (Tabelle aller exception Vectors) beinhaltet die Initialisierung des Stackpointers sowie die Adressen sämtlicher exception handler (Interrupt Handler)
[4] Siehe Abschnitt *29.3.1 Flash Configuration Field Description* im RM.
[5] Das Standard-Script kann beim Aufruf des Linkers mit der Option `--verbose` angesehen werden.

Es können auch weitere benutzerspezifische Sections definiert werden. Für die DYPS ONE-Umgebung, welche den Kinetis als μC verwendet, werden folgende zusätzliche Input-Sections definiert:

.isr_vector für die Speicherung der Vektor-Tabelle

.flash_config für die Flash-Konfiguration

Mit diesen Informationen kann nun mit dem Mapping begonnen werden.
Gestartet wird mit der Input-Section « `.isr_vector` ». Diese muss in die Output-Section « `.vector` » gemappt werden, welche wiederum in den `ROM_VECTORS`-Bereich gemappt werden muss. Dies wird mit folgendem Code erreicht:

```
.vectors :
{
    . = ALIGN(4);
    KEEP(*(.isr_vector))
    . = ALIGN(4);
} > ROM_VECTORS
```

Zur Erläuterung: Mit dem letzten Befehl `> ROM_VECTORS` wird die Output-Section `.vectors` in den zuvor definierten Bereich `ROM_VECTORS` gemappt. Während mit dem Befehl `*(.isr_vector)` die Input-Section `.isr_vector` in der Output-Section `.vectors` platziert wird.

Hinweis: Alignment

Der Befehl `. = ALIGN(4);` platziert den Program-Counter auf die nächste Adresse, welche ohne Rest durch 4 teilbar ist. Man spricht hier von einem « Alignment auf 4 Byte ».

Das Selbe muss für die Input-Section `flash_config` gemacht werden, welche in die Output-Section `.flash_cfg` platziert werden muss. Diese muss anschliessend in den Bereich `ROM_FLASH_CFG` gemappt werden.

```
.flash_cfg :
{
    . = ALIGN(4);
    KEEP(*(.flash_config))
    . = ALIGN(4);
} > ROM_FLASH_CFG
```

Nun sind die Input-Sections `.text` und `.rodata` an der Reihe. Diese sind in der Output-Section `.text` zu platzieren:

```
.text :
{
    . = ALIGN(4);
    *(.text*)
    *(.rodata*)
    . = ALIGN(4);
} > ROM_TEXT
```

Wiederum wichtig ist hier, dass die Output-Section `.text` mit dem Befehl `> ROM_TEXT` in den zuvor definierten Bereich `ROM_TEXT` gemappt wird.

Als Nächstes müsste die Input-Section `.data` entsprechend platziert werden. Da diese Section jedoch bei jedem Start in den RAM-Bereich kopiert werden muss, ist es sinnvoll an dieser Stelle zuerst ein Symbol zu erstellen, welches für das spätere Kopieren der Daten hilfreich ist (siehe Abschnitt 2.1.3). Es wird dazu das Symbol `_sfdata` (für Start-Flash-Data) gesetzt.

```
_sfdata = .; /* Symbol at start of flash data section */
```

Nun kann die Input-Section `.data` platziert werden:

```
.data : AT(ROM_TEXT)
{
    . = ALIGN(4);
    _sdata = .;
    *(.data*)
    _edata = .;
    . = ALIGN(4);
} > RAM_U
```

Da diese Section zu Beginn jedoch im `ROM_TEXT` Bereich ist und beim Start in dem RAM-Bereich kopiert wird, muss dieser Bereich sowohl in den RAM wie auch in den ROM-Bereich gemappt werden. Für das Mapping in den RAM-Bereich wird dazu wie gewohnt der Befehl `> RAM_U` verwendet. Zusätzlich wird aber diese Section mit dem Befehl `: AT(ROM_TEXT)` in den ROM-Bereich umgeleitet. Auch hier werden zur Vereinfachung des Kopierens der

Daten vom ROM ins RAM zwei Symbole `_sdata` (start Data) und `_edata` (end Data) jeweils vor und nach der Input-Section `.data` platziert.

Zum Schluss fehlt noch das Mapping der `.bss` und `COMMON` Sections. Diese gehören ebenfalls in den RAM-Bereich. Auch hier werden für das spätere Löschen zwei Symbole `__bss_start__` und `__bss_end__` definiert.

```
.bss :
{
    . = ALIGN(4);
    __bss_start__ = . ;
    *(.bss*)
    *(COMMON)
    __bss_end__ = . ;
    . = ALIGN(4);
} > RAM_U
```

Nun ist die Konfiguration des Linker Scripts abgeschlossen. Es fehlt nur noch ein Symbol für den Stack-Start. Dieses wird als `_stack_top` bezeichnet und liegt am Ende des RAM-Bereichs. Das komplette Linker Script « k22f.ld » ist im Listing 2.1 zu sehen.

```
MEMORY
{
    ROM_VECTORS   (rx) : ORIGIN = 0x00000000, LENGTH = 0x00000400
    ROM_FLASH_CFG (rx) : ORIGIN = 0x00000400, LENGTH = 0x00000010
    ROM_TEXT      (rx) : ORIGIN = 0x00000410, LENGTH = 256K-0x00000410
    RAM_L         (rw) : ORIGIN = 0x1FFFC000, LENGTH = 16K
    RAM_U         (rw) : ORIGIN = 0x20000000, LENGTH = 32K
}

SECTIONS
{
    .vectors :
    {
        . = ALIGN(4);
        KEEP(*(.isr_vector))
        . = ALIGN(4);
    } > ROM_VECTORS

    .flash_cfg :
    {
        . = ALIGN(4);
        KEEP(*(.flash_config))
        . = ALIGN(4);
    } > ROM_FLASH_CFG

    .text :
    {
        . = ALIGN(4);
        *(.text*)
        *(.rodata*)
        . = ALIGN(4);
    } > ROM_TEXT

    _sfdata = LOADADDR(.data);
    .data : AT(ROM_TEXT)
    {
        . = ALIGN(4);
        _sdata = .;
        *(.data*)
        _edata = .;
        . = ALIGN(4);
    } > RAM_U

    .bss :
    {
        . = ALIGN(4);
        __bss_start__ = . ;
        *(.bss*)
        *(COMMON)
        __bss_end__ = . ;
        . = ALIGN(4);
    } > RAM_U

    _stack_top = ORIGIN(RAM_U) + LENGTH(RAM_U);
}
```

Listing 2.1 Komplettes Linker Script für das DYPS ONE-Board

2.1.3 Startup Code

Jeder ARM-Prozessor führt beim Startup folgende Schritte durch:

Load Stack Pointer Es wird der Stack Pointer mit dem Wert am Anfang der Vector Tabelle geladen.

Execute Resethandler Es wird der Resethandler ausgeführt. Die Adresse zu diesem liegt an der zweiten Stelle der Vector Tabelle.

Für ein minimales Beispiel eines Startup Codes müssen folglich die ersten zwei Vektortabellen-Einträge definiert und ein Resethandler implementiert werden. Zusätzlich muss für Kinetis μCs die `.flash_config` definiert werden.

Listing 2.2 zeigt die minimale « startup.s » Datei, welche die Input-Sections `.isr_vector` und `.flash_config` erzeugt. In der Vektortabelle (`.isr_vector`-Section) werden zudem die ersten zwei Einträge definiert. Diese sind:

StackPointer entspricht der obersten RAM Adresse (siehe Listing 2.1)

Adresse des Resethandlers Der Resethandler wird auf Zeile 17 und folgende definiert. Es wird folglich dessen Label `_reset` als Sprungadresse verwendet.

Auch wird die `.flash_config` definiert. Diese kann mit Hilfe des RM dekodiert werden. Auf weitere Einzelheiten dieses Codes wird an dieser Stelle nicht eingegangen.

```
1  .cpu cortex-m4
2  .thumb
3
4  .section .isr_vector, "a"
5      .long _stack_top
6      .long _reset
7
8  .section .flash_config, "a"
9      .long 0xFFFFFFFF
10     .long 0xFFFFFFFF
11     .long 0xFFFFFFFF
12     .long 0xFFFFFFFE
13
14 .section .text
15 .thumb_func
16 .global _reset
17 _reset:
18     bl init
19     bl main
```

Listing 2.2 Minimale Startup Datei für das DYPS ONE-Board

Achtung: Änderungen an der `.flash_config`

Bei Änderungen an der `.flash_config` ist höchste Vorsicht geboten. Mit der Konfiguration kann der μC gegen das Lesen und Schreiben geschützt werden. Ein weiteres Flashen in einem solchen Fall, würde folglich vom μC verweigert. Ein Spielen mit dieser Section wird also nicht empfohlen ;-).

Hinweis: Vektortabelle

Für einen generellen Einsatz dieses Startup-Skripts in anderen Programmen müsste die Vektortabelle komplett implementiert werden (siehe RM Abschnitt *3.2.2.3 Interrupt channel assignments*).

Im Resethandler werden die zwei Funktionen `init` und `main` aufgerufen. Dabei ist klar, dass die `main` Funktion der Applikation entspricht. Die `init` Funktion hingegen soll sich um den korrekten Aufstart des μC kümmern. Folgende Aufgaben gehören dazu:

Watchdog deaktivieren Für Überwachungszwecke haben μC einen sogenannten Watchdog. Dieser hat die Aufgabe das System zu überwachen. Wird eine mögliche Fehlfunktion erkannt, so wird dies dem System mitgeteilt (WHD_IRQ), sodass das System entsprechend reagieren kann. Ein Totalausfall eines Gerätes durch Softwareversagen wird dadurch massiv verringert. Für diese Überwachung muss die Software in regelmässigen Abständen dem WHD mitteilen, dass es noch ordnungsgemäss

arbeitet. Würde die Software nicht mehr ordnungsgemäss arbeiten, so würden die Mitteilungen an den WHD ausbleiben, was einen Reset zur Folge hätte. Diese Überwachung ist im vorliegenden Beispiel nicht erwünscht. Da der WHD jedoch standardmässig aktiv ist, muss er zu Beginn deaktiviert werden.

`.data` Sektion kopieren Die `.data` Sektion muss vor dem eigentlichen Applikationsstart vom ROM ins RAM kopiert werden.

`.bss` Sektion löschen Die `.bss` Sektion muss vor dem eigentlich Applikationsstart auf null gesetzt werden.

Generell ist die `init`-Funktion unabhängig von der Applikation und wird deshalb in der separaten Datei « startup.c » gespeichert.

```
/* Titel:       minimal startup code for the DYPS ONE Board
 * Datei:       startup.c
 * Ersteller:   R.Gassmann
 * Funktion:    Initialises the K22 - uC
 *              1. disable WHD
 *              2. copy data-section
 *              3. clear bss-section
 */

#define WDOG_STCTRLH_REG    (*(volatile short *)0x40052000u)
#define WDOG_UNLOCK_REG     (*(volatile short *)0x4005200Eu)

#define WDOG_UNLOCK_MASK    0xFFFFu
#define WDOG_UNLOCK_SHIFT   0
#define WDOG_UNLOCK_WIDTH   16
#define WDOG_UNLOCK(x)      (((short)(((short)(x))<<WDOG_UNLOCK_SHIFT))&WDOG_UNLOCK_MASK)

#define WDOG_STCTRLH_WAITEN_MASK 0x80u
#define WDOG_STCTRLH_STOPEN_MASK 0x40u
#define WDOG_STCTRLH_ALLOWUPDATE_MASK 0x10u
#define WDOG_STCTRLH_CLKSRC_MASK 0x2u

#define WDOG_STCTRLH_BYTESEL_MASK 0x3000u
#define WDOG_STCTRLH_BYTESEL_SHIFT 12
#define WDOG_STCTRLH_BYTESEL(x) (((short)(((short)(x))<<WDOG_STCTRLH_BYTESEL_SHIFT))&
    WDOG_STCTRLH_BYTESEL_MASK)

extern unsigned int _sfdata;
extern unsigned int _edata;
extern unsigned int _sdata;
extern unsigned int __bss_start__;
extern unsigned int __bss_end__;

void init()
{
    unsigned int *src, *dst;
    // Disable WHD
    WDOG_UNLOCK_REG =  WDOG_UNLOCK(0xC520);
    WDOG_UNLOCK_REG =  WDOG_UNLOCK(0xD928);
    WDOG_STCTRLH_REG = WDOG_STCTRLH_BYTESEL(0x00) |
                       WDOG_STCTRLH_WAITEN_MASK |
                       WDOG_STCTRLH_STOPEN_MASK |
                       WDOG_STCTRLH_ALLOWUPDATE_MASK |
                       WDOG_STCTRLH_CLKSRC_MASK |
                       0x0100U;
    // Copy .data section
    src = &_sfdata;
    for ( dst = &_sdata ; dst < &_edata ; ) {
        *(dst++) = *(src++);
    }
    // Clear .bss section
    for ( src = &__bss_start__ ; src < &__bss_end__ ; ) {
        *(src++) = 0;
    }
}
```

Listing 2.3 Minimaler Startup Code für das DYPS ONE-Board

In der « startup.c »-Datei aus Listing 2.3 sind besonders die als `extern` definierten Variablen in den Zeilen 27 bis 31 hervorzuheben. Diese haben die selben Bezeichnungen wie die Symbole im Linker Script, wodurch der Linker beim Linken eine Zuordnung der Symbole zu diesen Variablen vornehmen wird. Damit stehen die Informationen des Linker Scripts im Code zur Verfügung. Das Kopieren der `.data` Section vom ROM-Bereich (`_sfdata`) in den RAM-Bereich (`_sdata`) kann daher mittels einer einfachen for-Schleife gelöst werden. Selbes gilt natürlich auch für das Löschen der `.bss` Section mit den Symbolen `__bss_start__` und `__bss_end__`.

2.1.4 Applikation

Nun sind die nötigen Grundlagen gelegt und so kann mit dem Schreiben der Applikation begonnen werden. Nachfolgend wurde eine einfache Blink-Applikation geschrieben. Da der Applikations-Code jedoch nichts mit dem eigentlichen Erstellen eines Projekts zu tun hat, wird dieser Code nicht genauer erläutert.

```
1  /* Titel:       minimal blink-application
2   * Datei:       main.c
3   * Ersteller:   R.Gassmann
4   * Funktion:    lets D0 blink
5   */
6
7  #define SIM_SCGC5   (*(volatile int *)0x40048038)
8  #define SIM_SCGC5_PORTB 10
9
10 #define PORTB_PCR16 (*(volatile int *)0x4004A040)
11 #define PORTB_PCR16_MUX 8
12
13 #define GPIOB_PSOR  (*(volatile int *)0x400FF044)
14 #define GPIOB_PCOR  (*(volatile int *)0x400FF048)
15 #define GPIOB_PDDR  (*(volatile int *)0x400FF054)
16 #define PIN_N 16
17
18 int main()
19 {
20     unsigned long i;
21     // Enable clocks.
22     SIM_SCGC5 |= 1 << SIM_SCGC5_PORTB;
23     // Configure pin 16 as GPIO.
24     PORTB_PCR16 |= 1 << PORTB_PCR16_MUX;
25     // Configure GPIO pin 16 as output.
26     GPIOB_PDDR |= 1 << PIN_N;
27     GPIOB_PSOR |= 1 << PIN_N;
28
29     while(1) {
30         for ( i = 500000 ; i > 0 ; i-- ) ;
31         GPIOB_PCOR |= 1 << PIN_N;
32         for ( i = 500000 ; i > 0 ; i-- ) ;
33         GPIOB_PSOR |= 1 << PIN_N;
34     }
35     return 0;
36 }
```

Listing 2.4 Einfache Blink-Applikationmit dem DYPS ONE-Board

Hinweis

Für weitere Informationen bezüglich Funktion/Aufbau/Logik dieser Applikation wird auf das RM verwiesen.

2.1.5 Makefile

Für die Übersetzung, der mit dieser Anleitung erstellten Dateien, ist es sinnvoll, noch ein Makefile zuschreiben. Dies könnte in etwa wie folgt aussehen:

```
1  CC=arm-none-eabi-gcc
2  OBJCPY=arm-none-eabi-objcopy
3  SIZE=arm-none-eabi-size
4  CFLAGS=-Wall -Wextra -mthumb -mcpu=cortex-m4 -nostdlib -g
5
6  all:
7      $(CC) startup.s startup.c main.c $(CFLAGS) -T k22f.ld -o simple.elf
8      $(SIZE) simple.elf
```

```
9       $(OBJCPY) -O srec simple.elf  simple.srec
10      UsbdmFlashProgrammer -device=MK22FN256M12 -erase=Mass -program simple.srec -execute
11
12  clean:
13      rm simple.*
```

Listing 2.5 Makefile für das minimale DYPS ONE-Projekt

Einige Erläuterungen dazu:

Das Target « all » löst den Übersetzungsvorgang aus. Als Kompiler wird dazu der `arm-none-eabi-gcc` verwendet. Für die Übersetzung sind die `CFLAGS` gesetzt, welche spezifizieren, dass es sich beim vorliegenden μC um einen cortex-m4 handelt, der lediglich den `thumb`-Mode unterstützt. Zudem werden sämtliche Warnings eingeschaltet und es wird eine Übersetzung ohne zusätzliche Bibliotheken verlangt.
Nach der Übersetzung werden die Grössen der einzelnen Sektionen mittels `arm-none-eabi-size` ausgegeben, bevor die erstellte *.elf-Datei mittels `arm-none-eabi-objcopy` in eine *.srec-Datei gewandelt und anschliessend mit dem `UsbdmFlashProgrammer` auf das DYPS ONE-Board geladen wird.

Das Target « clean » säubert das Projekt von alten Übersetzungsdateien.

Eine Übersetzung kann nun in der Konsole mit dem Befehl `make` oder `make all` ausgelöst werden.

```
$ make
arm-none-eabi-gcc startup.s startup.c main.c -Wall -Wextra -mthumb -mcpu=cortex-
    m4 -nostdlib -g -T k22f.ld -o simple.elf
arm-none-eabi-size simple.elf
   text    data     bss     dec     hex filename
    300       0       0     300     12c simple.elf
arm-none-eabi-objcopy -O srec simple.elf  simple.srec
UsbdmFlashProgrammer -vdd=3V3 -device=MK22FN256M12 -erase=Mass -program simple.
    srec -execute
```

2.2 Ein neues Projekt erstellen (Code::Blocks Wizard)

Um ein neues Projekt zu erstellen wird über das Menu File→New→Projekt... oder auf dem « Start here »-Schirm über « Create a new project » der Projektwizard geöffnet (siehe Abb. 2.4).

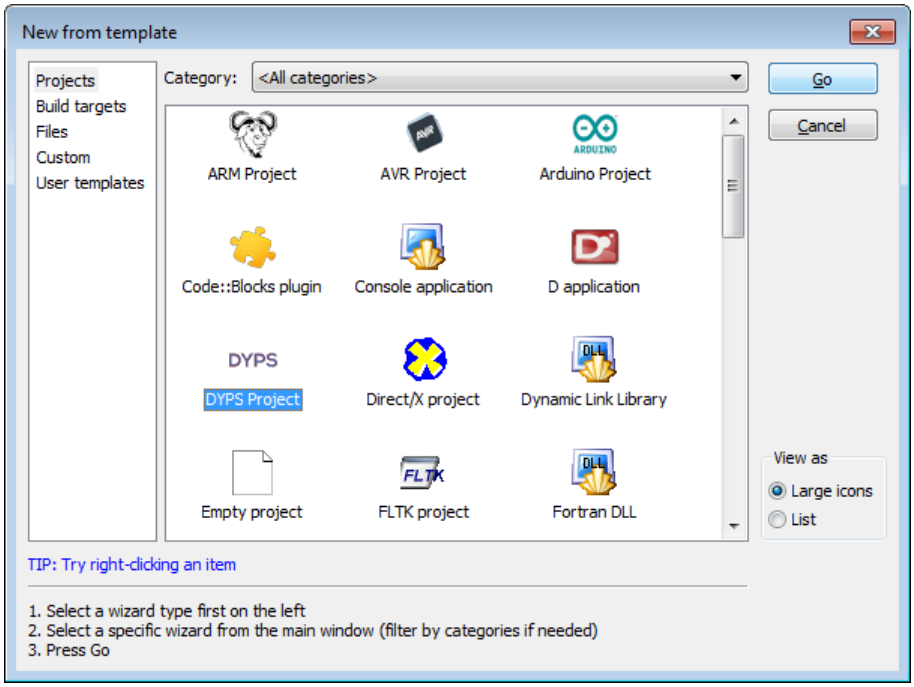


Abbildung 2.4. Im Fenster « New from template » ist das « DYPS Projekt » anzuwählen und anschliessend mit « Go » zu bestätigen.

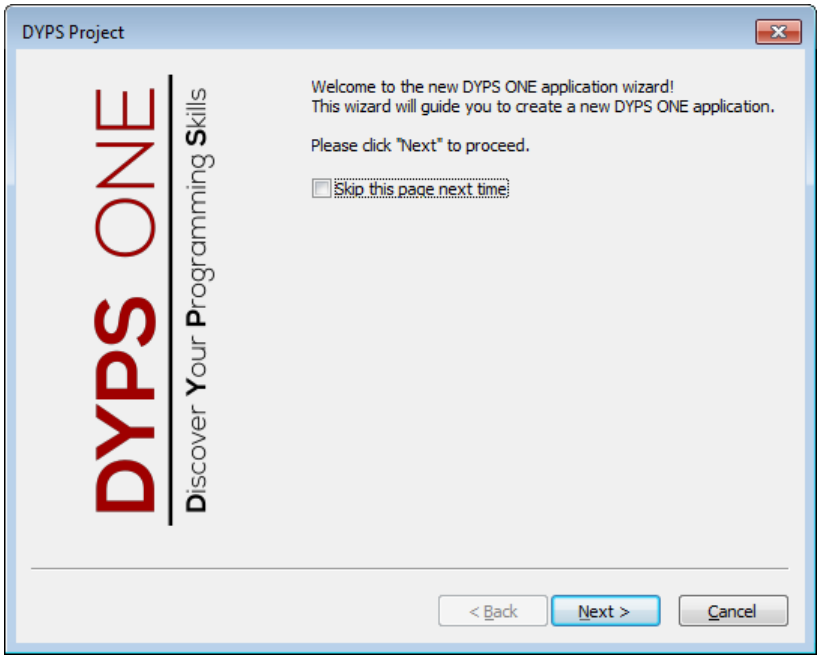


Abbildung 2.5. Das « Welcome-Fenster » kann mit « Next > » bestätigt werden (Achtung! « Skip this page next time » sollte nicht angewählt sein).

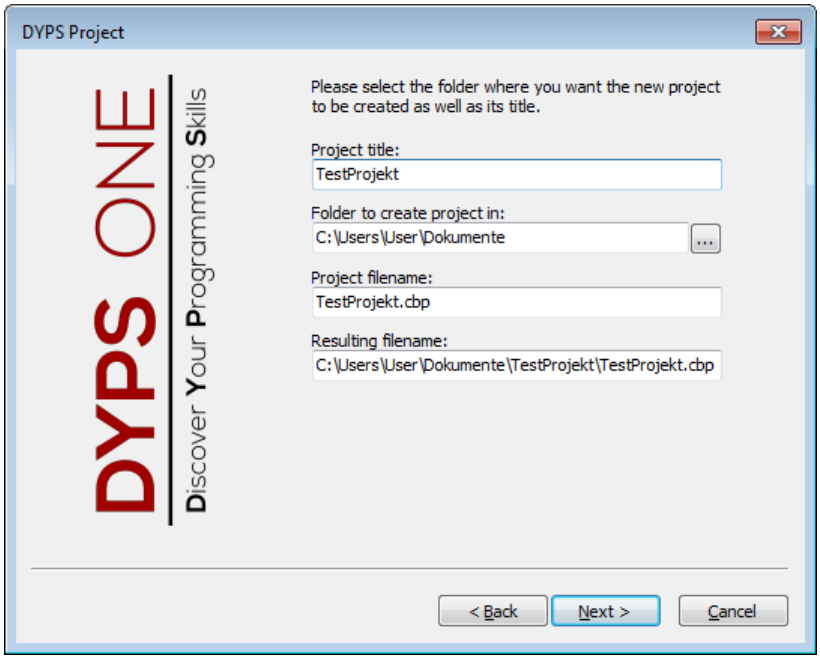


Abbildung 2.6. Anschliessend muss ein Projekttitel sowie ein Speicherort festgelegt werden (Achtung! Es gilt jedes mal den Pfad zu kontrollieren). Auch dieses Fenster kann mit « Next > » bestätigt werden.

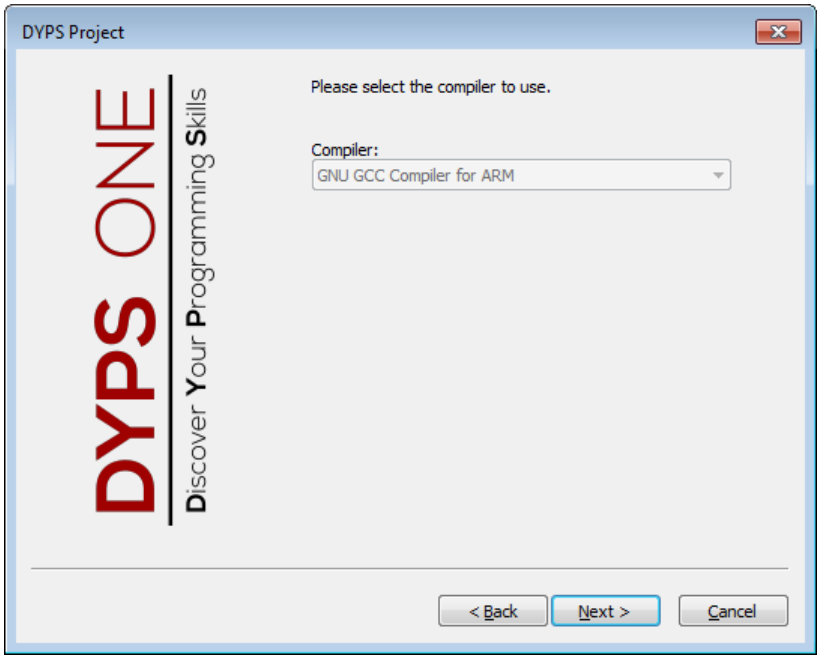


Abbildung 2.7. Als Compiler ist nur der GNU GCC Compiler for ARM zugelassen d.h. auch hier mit « Next > » bestätigen.

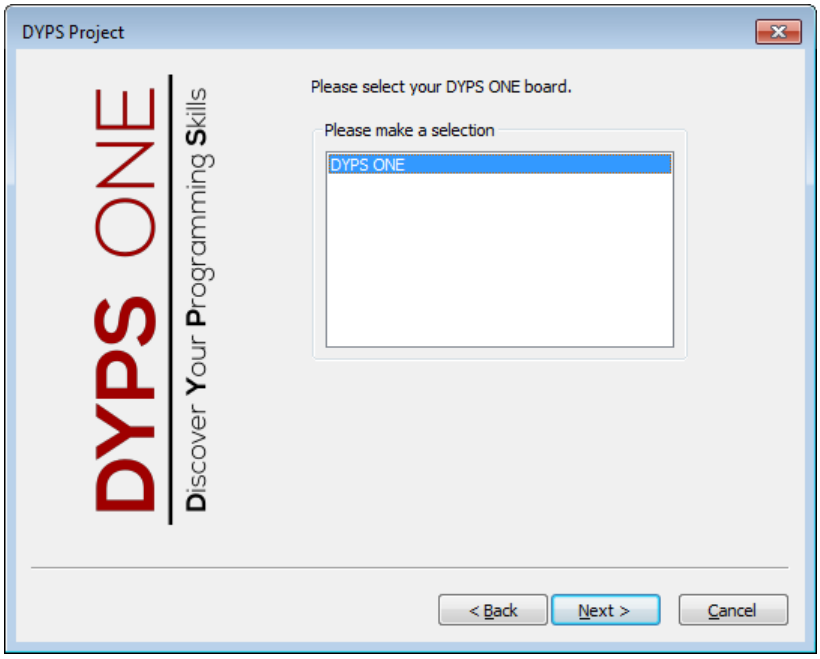


Abbildung 2.8. Es gilt hier das DYPS ONE auszuwählen und mit « Next > » zu bestätigen.

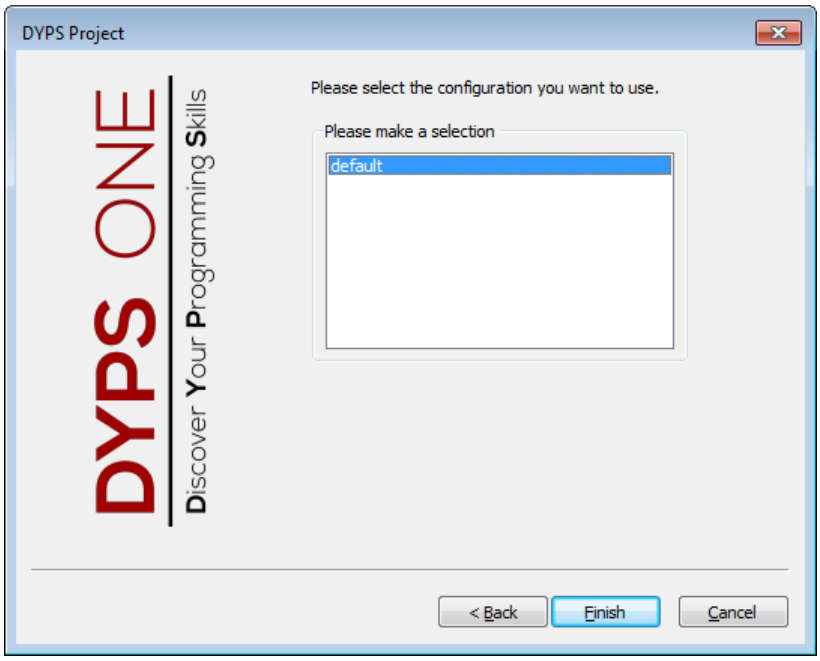


Abbildung 2.9. Abschliessend muss die « default » Konfiguration noch mit « Finish » bestätigt werden.

2.2.1 Debugger-Options setzten

Um das vorliegende Projekt zu Debuggen müssen unter den Projekt-Einstellungen die Debugger-Options noch eingestellt werden. Es wird dazu mit der rechten Maustaste auf das Projekt geklickt und im Menu der Punkt « Properties... » gewählt. In den « Project/targets options » wird in den Register Debugger gewechselt. Hier kann in der unteren Hälfte des Fensters unter « Select target: » Das « <Project> » angewählt werden. Rechts daneben wird nun als Connection type: « TCP », als IP address: « localhost » und als Port: « 1234 » eingetragen (siehe Abb. 2.10). Anschliessend wird ins Register « Additional GDB commands » gewechselt wo im unteren Feld (« After connection: ») « monitor reset halt » eingetragen wird (siehe Abb. 2.11). Damit ist die Konfiguration des Debuggers abgeschlossen.

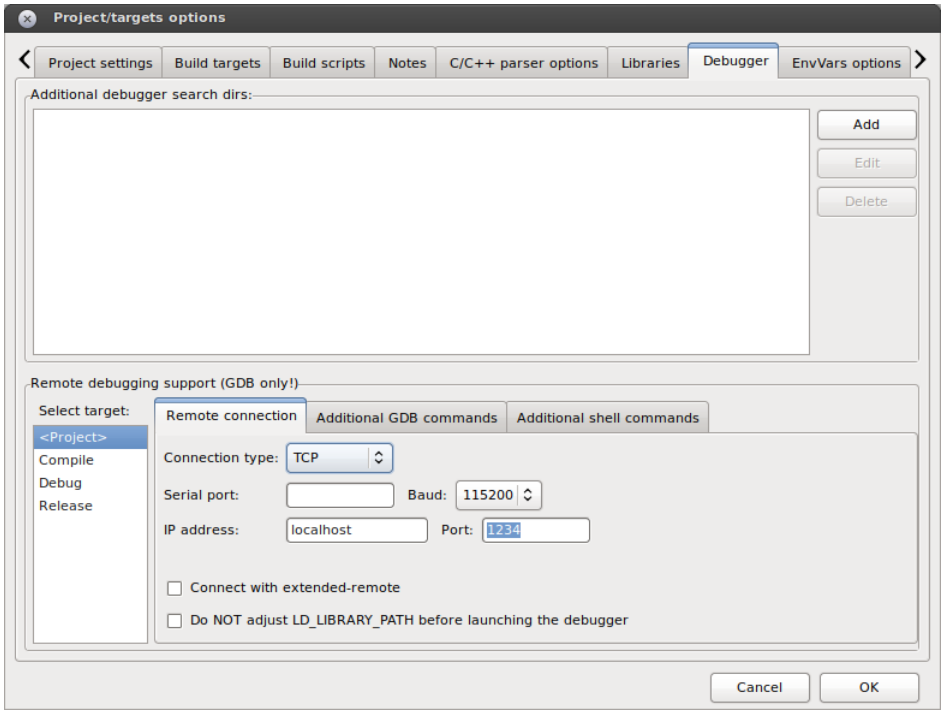


Abbildung 2.10. Debugger Remote connection Einstellungen.

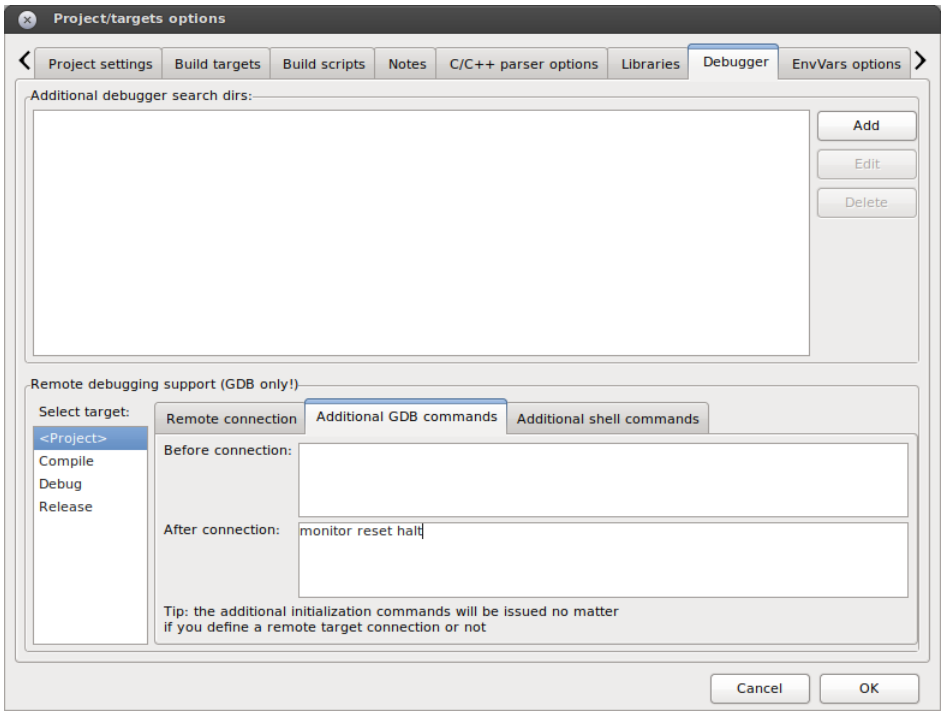


Abbildung 2.11. Debugger Additional GDB commands Einstellungen.

2.2.2 Projekt übersetzten und downloaden

Nach dem erfolgreichen Erstellen und Konfigurieren eines neuen Projekt, kann dieses nach der Wahl des « Build targets » (siehe Abb. 2.12) mittels der Taste « Rebuild » ( ) übersetzt und auf das DYPS-Board geladen werden.

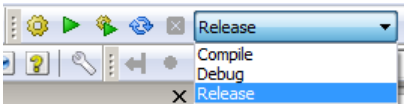


Abbildung 2.12. Build target Wahl.

Generell stehen dazu 3 verschiedene Builds zur Verfügung

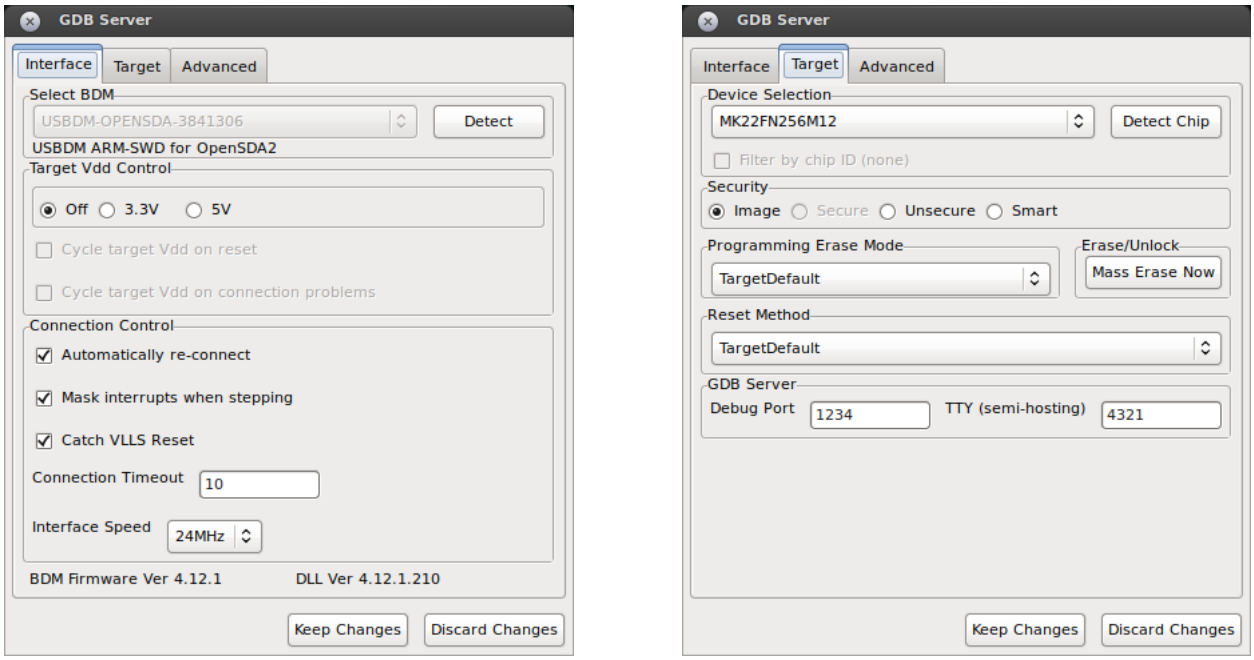
Compile Kompiliert das vorliegende Projekt lediglich.

Debug Kompiliert das vorliegende Projekt inklusive allen Debug-Symbolen. Nach erfolgreichem Kompiliervorgang wird das Image (*.elf) in ein *.srec gewandelt und anschliessend auf die Hardware geladen. Es ist an dieser Stelle nun möglich eine Debug-Session zu starten.

Release Kompiliert das vorliegende Projekt. Entfernt alle Debug-Symbole und Optimiert das Programm auf Geschwindigkeit. Bei erfolgreichem Kompilieren wird das Image auf die Hardware geladen und ausgeführt.

2.3 Ein Projekt mit Code::Blocks Debuggen

Zu Beginn wird kontrolliert, ob die Schritte aus Abschnitt 2.2.1 durchgeführt wurden. Gegebenenfalls werden diese Schnitte noch nachgeführt. Anschliessend muss eine DEBUG-Verbindung mit dem DYPS Board hergestellt werden. Es muss dazu der GDB-Server des USBDM-Interfaces gestartet, indem der « ARM_GDBServer[6] » ausgeführt wird. Im GDB Server wird der Reiter « Interface » gemäss Abbildung 2.13a und der Reiter « Target » gemäss Abbildung 2.13b konfiguriert. Anschliessend kann mit dem DYPS ONE-Board eine Verbindung aufgebaut werden, indem auf « Keep Changes » geklickt wird. Der GDB Server ist somit bereit für eine Debug-Session.
Sobald der Server Bereit ist, kann das Projekt im « Debug »-Mode mittels der Taste « Rebuild » ( ) übersetzt



(a) Interface Einstellungen

(b) Target Einstellungen

Abbildung 2.13. GDB-Server Einstellungen

werden. Anschliessend wird unter dem Menu « Debug→Active debuggers » der « GDB/CDB debugger: armGDB[7] » ausgewählt. Anschliessend kann der Debug-Vorgang mittels gestartet werden.

2.3.1 Die Debug Steuerelemente

Breakpoint Ein Breakpoint kann mit einem einfachen Links-Klick, rechts neben der gewünschten Zeilennummer, gesetzt werden. Das Symbol zeigt an, das an dieser Stelle ein Breakpoint gesetzt wurde. Dieser kann mit einem erneuten Links-Klick wieder entfernt werden.

Ist ein Breakpoint gesetzt und es wird die Taste erneut gedrückt, so springt der Debugger an diese Stelle (markiert mit dem Symbol).

Nächste Zeile Mittels der Taste kann zur nächsten Zeile gesprungen werden.

Springe in Funktion Will in die aktuelle Funktion gesprungen werden, kann die Taste gedrückt werden.

Springe aus Funktion Eine Funktion kann mittels der Taste verlassen werden (der Debugger springt ans Ende dieser Funktion).

Debugging Stoppen Um den Debugger zu Beenden steht die Taste zur Verfügung.

[6] Wird mit dem Packet USBDM mit installiert.
[7] Sollte dieser nicht vorhanden sein, so wurde im Abschnitt 1.3 der Debugger nicht Konfiguriert

2.4 Debug-Umgebung

Bei aktivem Debugger können über die Taste  verschiedene Debug-Fenster geöffnet werden. Diese werden hier kurz erläutert.

2.4.1 Variablen Fenster

Um Variablen zu debuggen steht das Fenster « Watches » zur Verfügung. Es listet automatisch alle lokalen Variablen auf und verfolgt sie (d.h. bei Veränderungen werden diese automatisch aktualisiert). Sollte eine Variable nicht aufgelistet werden, so kann diese mit einem Rechtsklick im Programmcode auf die entsprechende Variable über den Eintrag « Watch '...' » hinzugefügt werden.

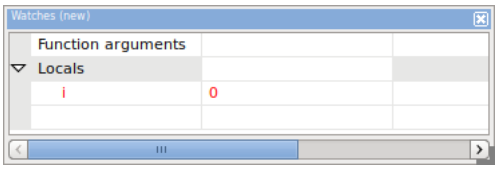


Abbildung 2.14. Debug Variablen Fenster. Es werden alle lokalen Variablen automatisch aufgelistet und verfolgt.

2.4.2 Memory Fenster

Will der speicher an einer bestimmten Stelle angesehen werden, so kann dies über das Memory Fenster erfolgen. Es muss dazu die Adresse der Speicherstelle sowie die Anzahl zu lesender Bytes angegeben werden. Anschliessend können diese mittels der Taste « Go » gelesen werden.

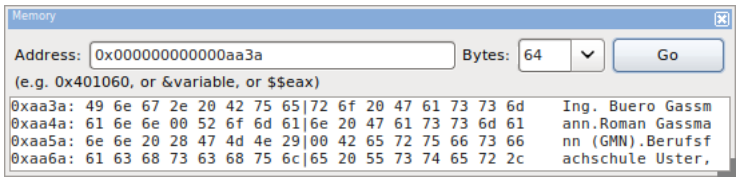


Abbildung 2.15. Debug Memory Fenster.

2.4.3 CPU-Registers

Auch könnten die CPU-Register beim Debuggen von Interesse sein. Dazu kann das Fenster CPU Registers verwendet werden. Die Register werden dabei automatisch aktualisiert und stehen damit während der gesamten Debug-Session jederzeit zur Verfügung.

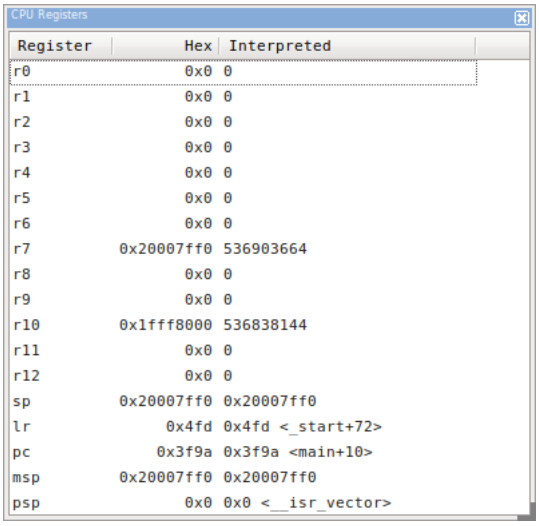


Abbildung 2.16. Debug CPU-Register Fenster

2.4.4 Assembler Fenster

Für erfahrene Programmierer steht dann auch noch das Disassembly Fenster zur Verfügung. Es zeigt den aktuellen Assembler-Code.

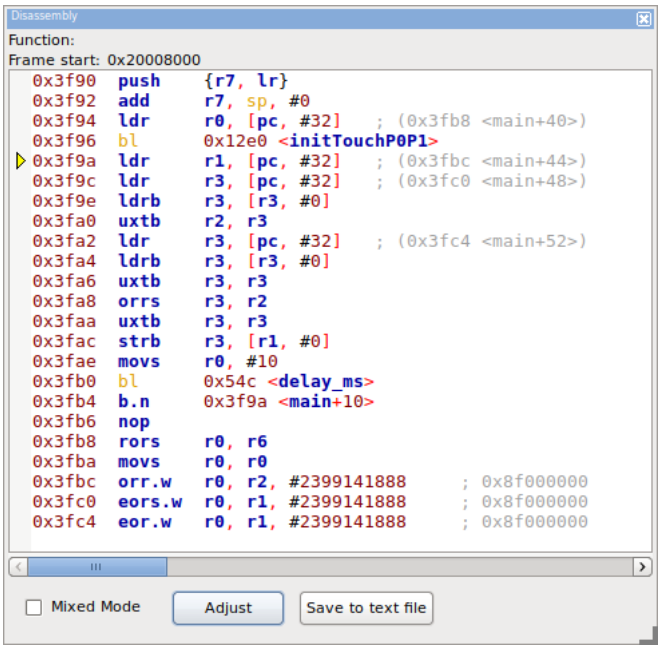
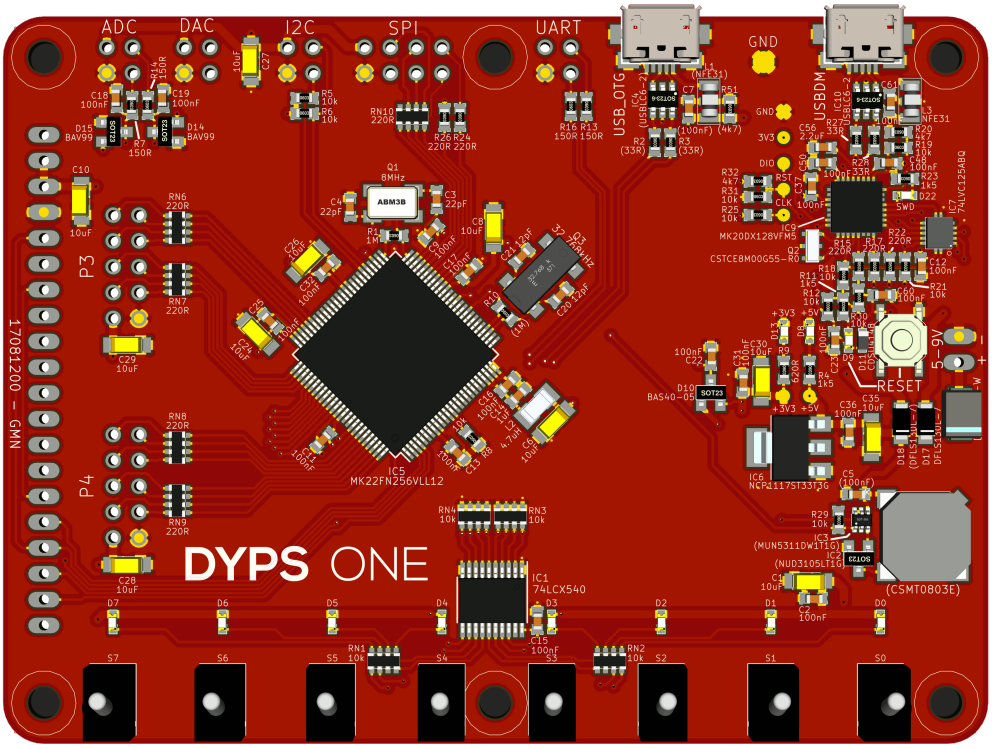


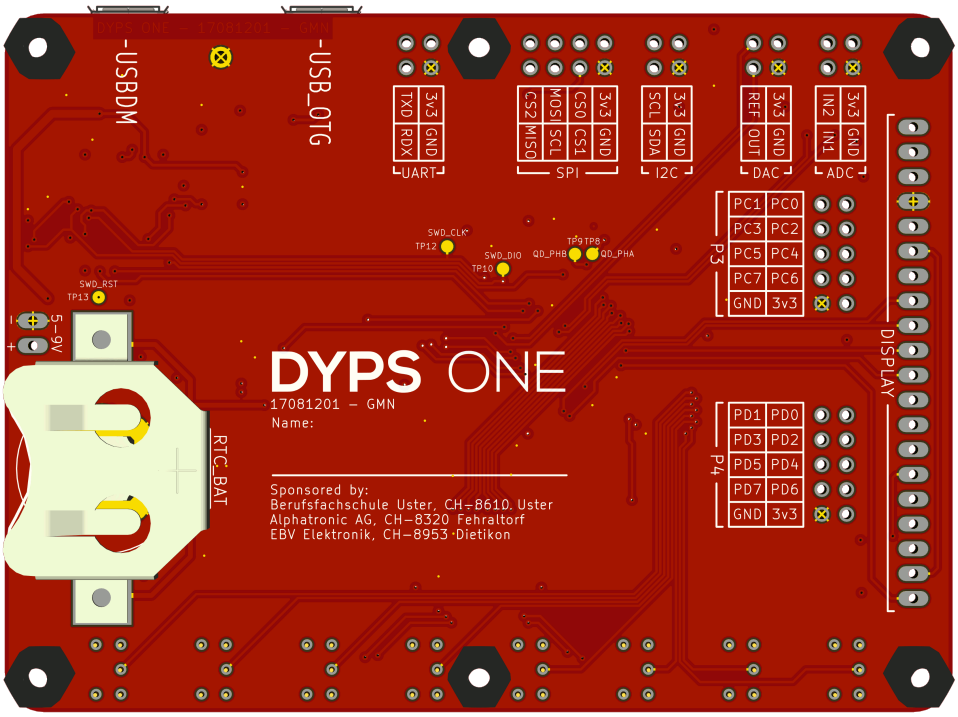
Abbildung 2.17. Debug Assembler Fenster

Kapitel 3

DYPS-ONE Hardware

Das Herzstück der DYPS ONE Hardware bildet der Mikrocontroller (MK22FN256VLL12). Für die Programmierung dieses μC, ist zudem ein Download-Interface (das UBDM-Interface, siehe Abschnitt 3.6) integriert. Auch sind verschiedene Funktions-Module wie LED's, Schalter, Interfaces etc. vorhanden, welche vom μC angesteuert werden können. Dazu werden für verschiedene Module in der `libDYPS.a` Interfaceroutinen bereit gestellt, welche in der `libDYPS.h` Headerdatei beschrieben sind. Zudem werden die einzelnen Module in diesem Kapitel genauer behandelt und deren Interface allenfalls mittels einem Beispielcode erläutert. Generell wird für die Verwendung der Module jedoch auf das Reference Manual (K22P121M120SF8RM.pdf) des μCs hingewiesen.





3.1 MCU-Spezifikationen

MCU	32 Bit ARM Cortex-M4 aus der Kinetis Familie. Genaue Bezeichnung MK22FN256VLL12.
Performance	120 MHz
Memory	256 kB Flash und 48 kB RAM
ADC	2×16 Bit SAR
DAC	12 Bit SAR
Interface	USB-OTG, 2× SPI, 3×UART, low-power UART, 2× I2C, I2S
Timers	8-Channel PWM-Timer, PI-Timer, LP-Timer, RTC
Speisung	1.71 V-3.6 V

3.2 LEDs - P1

Für einfachste Ausgaben stehen 8 LED's (D0-D7) zur Verfügung. Die LED's sind an den Pins PTB16 - PTB23 angeschlossen und werden über einen oktalen Buffer / line Driver (74LCS540) angesteuert. Nebst der 5 V Verträglichkeit der Ein- und Ausgänge, erlaubt dieser Treiberbaustein zusätzlich einen Strom von 50 mA pro Kanal.

Hinweis

Die LED's können bei bedarf ausgetauscht werden, solange die Eingesetzten LED's nicht mehr als 50mA benötigen. Es sollten dabei natürlich auch die Vorwiderstände **RN1** und **RN2** entsprechend angepasst werden.

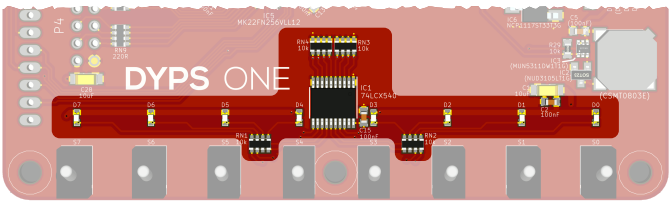


Abbildung 3.1. LEDs und Treiber auf dem Board

3.2.1 Ansteuerung

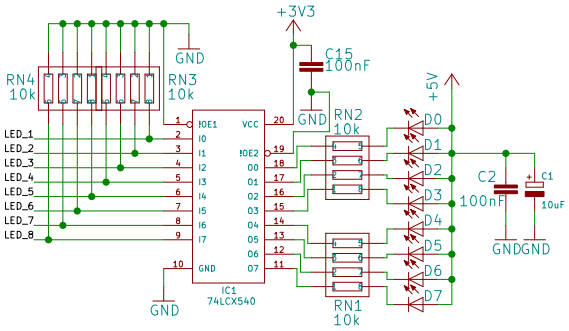


Abbildung 3.2. Schema der LED-Ansteuerung

Hinweis: Compiler Optimierung

Beim Schalten von Ausgängen ohne Verzögerungen ist bei einer Übersetzung mit «Release» Vorsicht geboten, da in diesem Fall die Compiler-Optimierung aktiv ist. Bei Programmen wie

```
int main(void) {
    initP0P1(); // LED und Schalter initialisieren
    while (1) {     // Endlosschleife
        P1 = 0x01;  // Einschalten der ersten LED
        P1 = 0x02;  // Einschalten der zweiten LED
        P1 = 0x04;  // Einschalten der dritte LED
    }
}
```

bei denen die Ausgänge mit voller Geschwindigkeit geschaltet würden, würde die Optimierung dazu führen, dass die Ausgänge 1 und 2 für gerade mal knapp 10 ns eingeschaltet würden. Die Ausgangsstufen sind jedoch nicht für diese Zeiten ausgelegt. Die LEDs (1+2) können damit nicht angesteuert werden. Anders sieht dies für LED 3 aus. Diese wird nämlich eingeschaltet (ca. 10 ns) und anschliessend muss die while-Schlaufe abgearbeitet werden dies führt zu einer zusätzlichen Verzögerung von rund 40 ns. Es bleibt damit die LED 3 für insgesamt 50 ns eingeschaltet was dazu führt, dass diese als einzige leuchtet (**Achtung:** nicht zu 100% da sie für ca. 2×10 ns $= 20$ ns ausgeschaltet wird).

Will das Schalten sichtbar gemacht werden so sind folgende Möglichkeiten vorhanden:

1. Verzögerungen einbauen. Es reicht hier nach jedem setzten der Ausgänge ein `delay_us(1);` auszuführen.
2. Software mit «Debug» übersetzten.
3. Es könnte das Attribute `__attribute__((optimize("O0")))` vor die Funktion mit entsprechenden Aufrufen (hier die `main`-Funktion) gesetzt werden. Dies würde dann wie folgt aussehen:

```
int __attribute__((optimize("O0"))) main(void) {
    initP0P1(); // LED und Schalter initialisieren
    while (1) {     // Endlosschleife
        P1 = 0x01;  // Einschalten der ersten LED
        P1 = 0x02;  // Einschalten der zweiten LED
        P1 = 0x03;  // Einschalten der dritte LED
    }
}
```

Es wird damit die Optimierung für diese Funktion ausgeschaltet.

Für die Ansteuerung der LEDs steht das Macro `P1` (aus `libDYPS.h`) zur Verfügung, wobei vor der Verwendung eine Initialisierung durch `initTouchP0P1()` oder `initP0P1()` nötig ist.

```
1 /*
2 Titel:      BspProgramm Ausgangs-Port
3 Datei:      P0toP1.c
4 Autor:      R.Gassmann
5 Funktion:   Ausgabe an P1
6 */
7
8 // Einbindung der Bibliotheken
```

```
9  #include "libDYPS.h"         // Header P0P1-defintionen
10
11 // Hauptprogramm
12 int main(void) {
13     initP0P1(); // LED und Schalter initialisieren
14     while (1) {      // Endlosschleife
15         P1 = 0x01;   // Einschalten der ersten LED
16     }
17 }
```

Listing 3.1 Beispielprogramm: LED-Ausgabe über P1

3.3 Schalter - P0

Zur Steuerung von Programmen stehen 8 Schalter (S0-S7) zur Verfügung. Diese sind an den Pins PTC8 - PTC15 des μCs angeschlossen, wobei sie bei der Initialisierung als Inputs mit PullDown-Widerständen Konfiguriert werden.

Hinweis

Es muss an dieser Stelle erwähnt werden, dass bei den Schaltern gezielt auf eine hardwaremässige Entprellung verzichtet wurde, damit dieser Effekt gezeigt und softwaretechnisch gelöst werden kann.

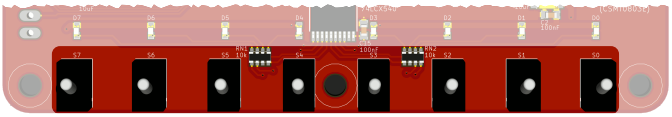


Abbildung 3.3. Schalter auf dem Board

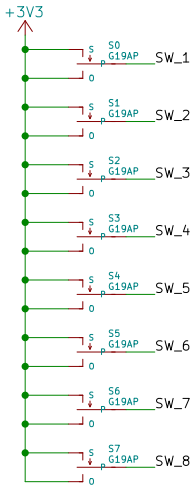


Abbildung 3.4. Schema der Schalter

3.3.1 Ansteuerung

Auch hier steht zur Ansteuerung ein Macro `P0` (aus `libDYPS.h`) zur Verfügung, wobei vor der Verwendung eine Initialisierung durch `initTouchP0P1()` oder `initP0P1()` nötig ist.

```
1  /*
2  Titel:      BspProgramm Ausgangs-Port
3  Datei:      P0toP1.c
4  Ersteller:  R.Gassmann
5  Funktion:   Liest die Schalter an Port 0 ein und gibt diese an Port 1 aus
6  */
```

```
7
8  // Einbindung der Bibliotheken
9  #include "libDYPS.h"        // Header P0P1-defintionen
10
11 // Hauptprogramm
12 int main(void) {
13     initP0P1(); // LED und Schalter initialisieren
14     while (1) {      // Endlosschleife
15         P1 = P0;     // Ausgabe der Schalter auf den LEDS
16     }
17 }
```

Listing 3.2 Beispielprogramm: Ausgangs-Port

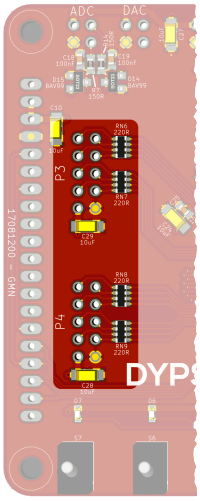
3.4 P3 und P4

Erweiterungen können einfach an das DYPS ONE über die Port P3 und P4 angeschlossen werden.

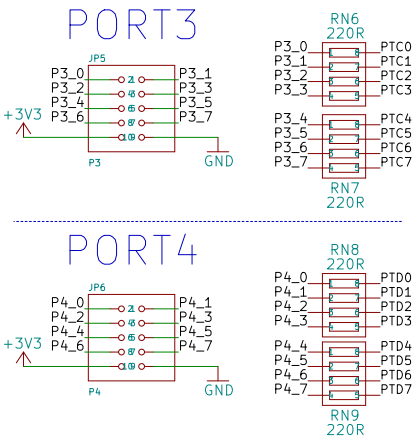
Dabei können an diesen Ports verschiedene Controller angeschlossen werden. Bei der Verwendung der verschiedenen Controller ist jedoch Vorsicht geboten, da einige der Controller bereits für andere Funktionen benötigt oder allenfalls konfiguriert werden. Genauere Informationen sind in den Abschnitten 3.4.1 und 3.4.2 zu finden.

Hinweis: Kurzschluss-Schutz

Aus dem Datenblatt (Kapitel "Voltage and current operating ratings") ist zu entnehmen, dass jeder digitale PIN ein Stromlimit von 25 mA aufweist.
Um den μC bzw. dessen PINs zu schützen, besitzt jeder PIN der PORTE 3 und 4 ein 220 Ω Widerstand welcher bei Kurzschlüssen den Strom auf 15 mA limitiert.



(a) Position auf dem Board

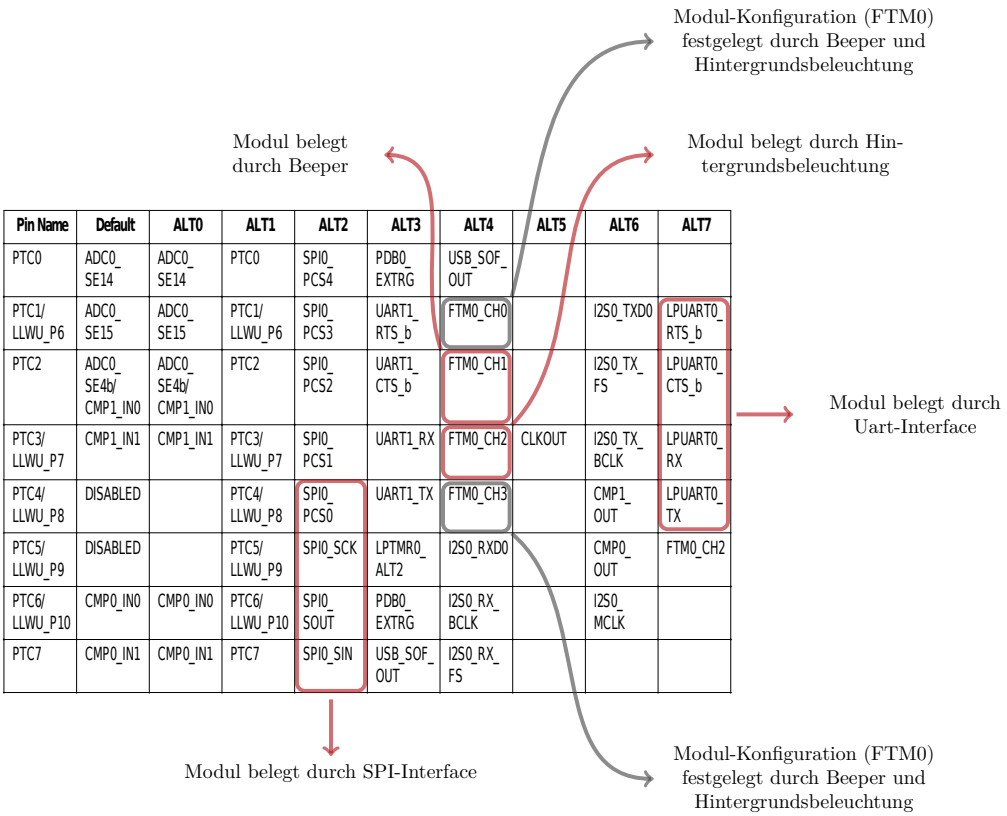


(b) Schema

Abbildung 3.5. GPIO-Ports P3 und P4

3.4.1 P3

Port P3 ist an den Pins PTC0 - PTC7 des μCs angeschlossen. Dabei sind die möglichen Konfigurationen der einzelnen PINs in der Übersicht (siehe Abb. 3.6) zusammengetragen. Auch sind die, durch mögliche andere Anwendungen, belegten (rot) oder konfigurierten (grau) Module in dieser Übersicht eingezeichnet.



Pin Name	Default	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7
PTC0	ADC0_SE14	ADC0_SE14	PTC0	SPI0_PCS4	PDB0_EXTRG	USB_SOF_OUT			
PTC1/LLWU_P6	ADC0_SE15	ADC0_SE15	PTC1/LLWU_P6	SPI0_PCS3	UART1_RTS_b	FTM0_CH0		I2S0_TXD0	LPUART0_RTS_b
PTC2	ADC0_SE4b/CMP1_IN0	ADC0_SE4b/CMP1_IN0	PTC2	SPI0_PCS2	UART1_CTS_b	FTM0_CH1		I2S0_TX_FS	LPUART0_CTS_b
PTC3/LLWU_P7	CMP1_IN1	CMP1_IN1	PTC3/LLWU_P7	SPI0_PCS1	UART1_RX	FTM0_CH2	CLKOUT	I2S0_TX_BCLK	LPUART0_RX
PTC4/LLWU_P8	DISABLED		PTC4/LLWU_P8	SPI0_PCS0	UART1_TX	FTM0_CH3		CMP1_OUT	LPUART0_TX
PTC5/LLWU_P9	DISABLED		PTC5/LLWU_P9	SPI0_SCK	LPTMR0_ALT2	I2S0_RXD0		CMP0_OUT	FTM0_CH2
PTC6/LLWU_P10	CMP0_IN0	CMP0_IN0	PTC6/LLWU_P10	SPI0_SOUT	PDB0_EXTRG	I2S0_RX_BCLK		I2S0_MCLK	
PTC7	CMP0_IN1	CMP0_IN1	PTC7	SPI0_SIN	USB_SOF_OUT	I2S0_RX_FS			

Abbildung 3.6. Pinout P3 mit den durch mögliche andere Anwendungen belegten (rot markiert) oder konfigurierten (grau markiert) Modulen.

3.4.1.1 GPIO-Port

Für die Initialisierung als GPIO's kann die in der DYPS Library vorhandene Funktion `initP3()` verwendet werden. Es muss dabei eine 8-Bit-Maske (unsigned char) übergeben werden, welche definiert, welcher PIN als Input (entsprechendes Bit ist 1) oder als Output (entsprechendes Bit ist 0) verwendet wird.

```
1 initP3( 0x05 ); // Configure P3_0 and P3_3 as inputs
```

Listing 3.3 Beispielprogramm: P3 als GPIO-Port

Anschliessend können über die folgenden Makros Einfluss auf die GPIO's genommen werden.

Hinweis

Um diese Makros nutzen zu können muss zuvor `initP3()` aufgerufen werden, da nur damit die GPIO's aktiviert werden.

`P3DIR` P3 - Output-Direction-Register
Über das Makro `P3DIR` können die GPIO's als Input (1) oder Output (0) konfiguriert werden.

```
P3DIR = 0x03; // Configure P3_0 and P3_1 as inputs (all others as
    outputs)
```

`P3IN` P3 - Input-Register
Über das Makro `P3IN` können die Zustände der Inputs abgefragt werden. GPIO's welche als Ausgänge geschaltet sind, sind 0.

```
unsigned char inputs = P3IN; // Read status of P3-Inputs
```

P3OUT P3 - Output-Register
Über das Makro P3OUT können Ausgänge gesetzt und gelöscht werden. GPIO's welche als Eingänge geschaltet sind ignorieren Schreibzugriffe mittels P3OUT.

```
P3OUT = 0x45; // Set P3_1, P3_3 and P3_6 clear others
```

P3SET P3 - Output-Set-Register
Wollen lediglich Ausgänge gesetzt (aber nicht gelöscht) werden so kann dies über das Makro P3SET gemacht werden. Alle Ausgänge welche auf 1 gesetzt sind, werden gesetzt.

```
P3SET = 0x45; // Set P3_1, P3_3 and P3_6
```

P3CLR P3 - Output-Clear-Register
Wollen lediglich Ausgänge gelöscht (aber nicht gesetzt) werden so kann dies über das Makro P3CLR gemacht werden. Alle Ausgänge welche auf 1 gesetzt sind, werden auf 0 gesetzt.

```
P3CLR = 0x45; // Clear P3_1, P3_3 and P3_6
```

P3TOG P3 - Output-Toggle-Register
Wollen lediglich Ausgänge getoggelt (invertiert) werden so kann dies über das Makro P3TOG gemacht werden. Alle Ausgänge welche auf 1 gesetzt sind, werden invertiert.

```
P3TOG = 0x45; // Toggle P3_1, P3_3 and P3_6
```

3.4.1.2 Spezial-Port

Für die Initialisierung der Port-Pins zu anderen Modulen wie SPI, UART, I2C, FTM etc. gilt folgender Ablauf:

1. PORT mit Takt versorgen
2. Modul initialisieren (gemäss Reference Manual)
3. PIN Zuweisung tätigen

Für Schritt eins stehen die Makros USE_PORTA ... USE_PORTE zur Verfügung. Schritt zwei ist stark vom jeweiligen Modul abhängig und so wird an dieser Stelle auf das Reference Manual verwiesen. Für Schritt 3 steht dann wiederum ein Makro (INIT_ALT_PIN bzw. INIT_ALT_PIN_EXT) zur Verfügung. Zur weiteren Vereinfachung kann das Makro PIN_ALT verwendet werden, um PIN-Konfigurationen zu erstellen.

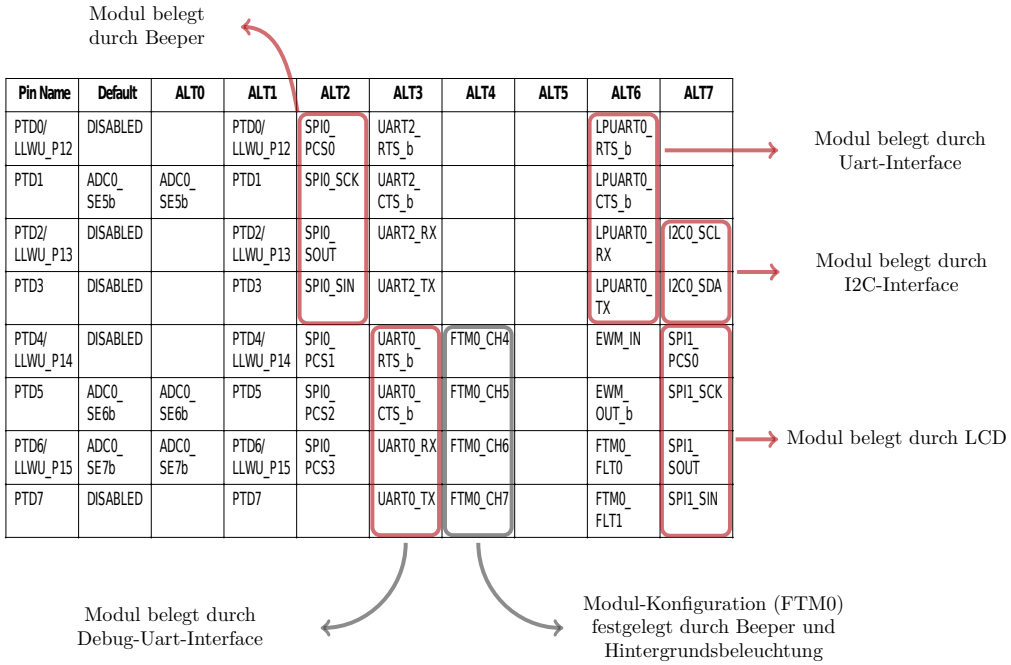
In Listing 3.4 ist die Initialisierung des PIN P3_1 zum PWM-Pin des FTM0 Moduls (Channel 0) mit Erklärung aufgeführt.

```
1  #define PWM_TEST_PIN     PIN_ALT( PIN_C1 , 4 )  // Pin Configuration (P3_1 als FTM0_CH0)
2
3  initP0P1(); // init P0 P1 and Buzzer (FTM0-Modul)
4  USE_PORTC; // Port3 mit Takt versorgen
5  // INIT FTM0 (bereits geschehen in iniP0P1())
6  // FTM0->MOD register will be set to 60000000/2350 = 25531
7
8  // Configure Channel 0
9  FTM0->CONTROLS[0].CnSC = FTM_CnSC_MSB_MASK | FTM_CnSC_ELSB_MASK;
10 FTM0->CONTROLS[0].CnV  = (25531)*50/100; // => PWM with 2350Hz and duty cycle of 50%
11
12 INIT_ALT_PIN(PWM_TEST_PIN);  // set P3_1 as FTM0_CH0 PIN
```

Listing 3.4 Beispielprogramm: P3 Spezial Pin Ansteuerung

3.4.2 P4

Port P4 ist an den Pins PTD0 - PTD7 des μCs angeschlossen. Dabei sind die möglichen Konfigurationen der einzelnen PINs in der Übersicht (siehe Abb. 3.7) zusammengetragen. Auch sind die, durch mögliche andere Anwendungen, belegten (rot) oder konfigurierten (grau) Module in dieser Übersicht eingezeichnet.



Pin Name	Default	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7
PTD0/ LLWU_P12	DISABLED		PTD0/ LLWU_P12	SPI0_ PCS0	UART2_ RTS_b			LPUART0_ RTS_b	
PTD1	ADC0_ SE5b	ADC0_ SE5b	PTD1	SPI0_SCK	UART2_ CTS_b			LPUART0_ CTS_b	
PTD2/ LLWU_P13	DISABLED		PTD2/ LLWU_P13	SPI0_ SOUT	UART2_RX			LPUART0_ RX	I2C0_SCL
PTD3	DISABLED		PTD3	SPI0_SIN	UART2_TX			LPUART0_ TX	I2C0_SDA
PTD4/ LLWU_P14	DISABLED		PTD4/ LLWU_P14	SPI0_ PCS1	UART0_ RTS_b	FTM0_CH4		EWM_IN	SPI1_ PCS0
PTD5	ADC0_ SE6b	ADC0_ SE6b	PTD5	SPI0_ PCS2	UART0_ CTS_b	FTM0_CH5		EWM_ OUT_b	SPI1_SCK
PTD6/ LLWU_P15	ADC0_ SE7b	ADC0_ SE7b	PTD6/ LLWU_P15	SPI0_ PCS3	UART0_RX	FTM0_CH6		FTM0_ FLT0	SPI1_ SOUT
PTD7	DISABLED		PTD7		UART0_TX	FTM0_CH7		FTM0_ FLT1	SPI1_SIN

Abbildung 3.7. Pinout P4 mit den durch mögliche andere Anwendungen belegten (rot markiert) oder konfigurierten (grau markiert) Modulen.

3.4.2.1 GPIO-Port

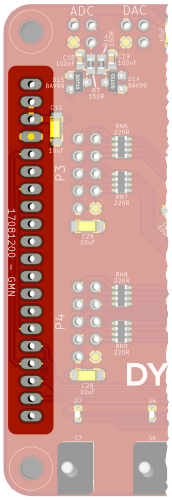
Die Konfiguration ist analog der Konfiguration von P3 (siehe Abschnitt 3.4.1.1).

3.4.2.2 Spezial-Port

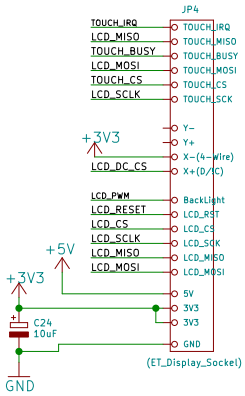
Die Konfiguration ist analog der Konfiguration von P3 (siehe Abschnitt 3.4.1.2).

3.5 TFT-Display

Für erweiterte Funktionen und Anzeigen kann der Display-Port (siehe Abb. 3.8) mit einem DYPS-TOUCH genutzt werden. Das DYPS-TOUCH kann dabei optional erworben werden.



(a) Position auf dem Board



(b) Schema

Abbildung 3.8. Displayport

Hinweis: Weitere Informationen

Weitere Informationen über die Verwendung sind in Kapitel 4 zu finden.

3.6 USBDM - Uart-Debug

USBDM ist ein Hardware Debugger Interface für NXP (früher Freescale) Mikrocontroller. In erster Linie wurde es für das Arbeiten mit Codewarrior unter Windows und Linux designed. Für weitere Arbeiten ohne Codewarrior stehen sogenannte stand-alone Programmers zur Verfügung, durch die es auch von anderen Programmen möglich ist, neue Software auf einen Mikrocontroller zu laden. Für die Programmierung mit der DYPS-Umgebung (CodeBlocks) wird der stand-alone Programmer via Kommandozeilen aufrufe verwendet.

Als Zusatz wurde in in dieses Interface auch eine USB-UART-Bridge implementiert. Es ist damit sehr einfach möglich, eine serielle Verbindung mit dem PC herzustellen, und damit Daten auszutauschen. In der DYPS-Library stehen hierfür bereits die Routinen `initDebugInterface()` (Initialisierung auf 115200-8-N-1) und `writeDebugMsg()` zur Verfügung.

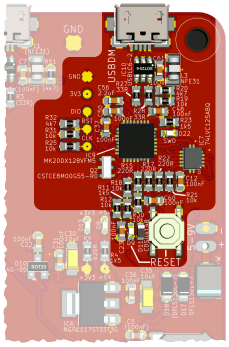


Abbildung 3.9. Das USBDM-Interface auf dem DYPS ONE

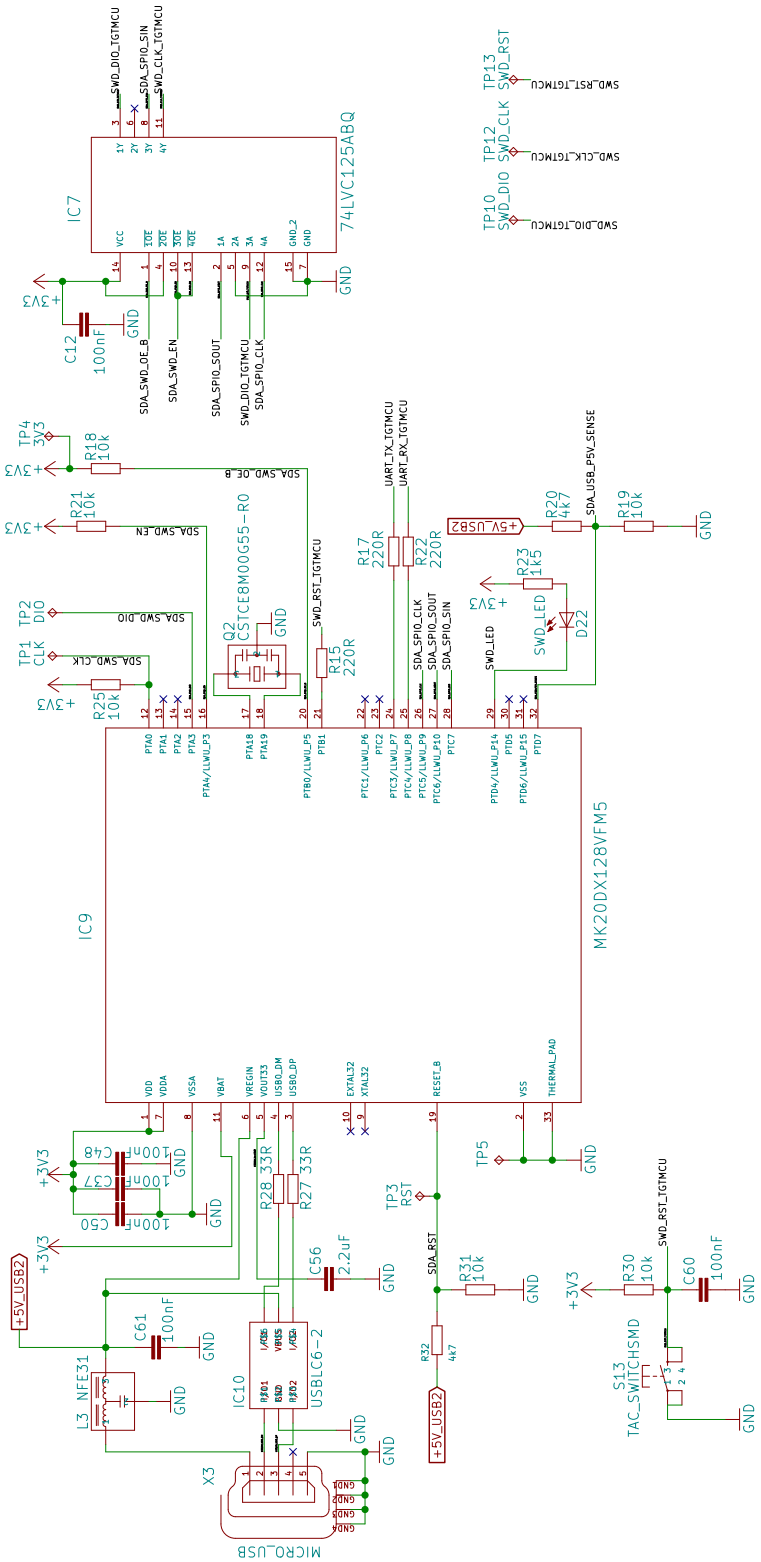
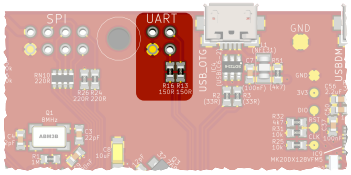


Abbildung 3.10. Schema des USBDM-Interfaces

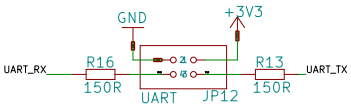
```
initDebugInterface();                // initialize debug interface
writeDebugMsg("Hello world\n\r");    // Write "Hello world" through debug interface
```

Listing 3.5 Beispielprogramm: Debug-Message

3.7 UART



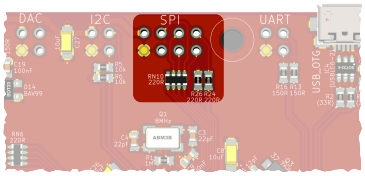
(a) Position auf dem Board



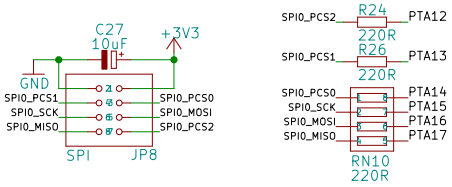
(b) Schema

Abbildung 3.11. UART-Interfaces

3.8 SPI



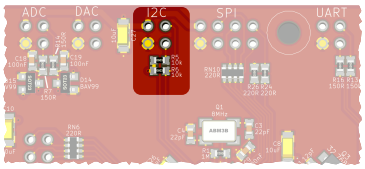
(a) Position auf dem Board



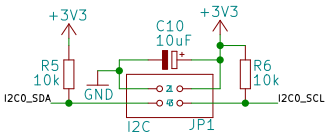
(b) Schema

Abbildung 3.12. SPI-Interfaces

3.9 I2C



(a) Position auf dem Board



(b) Schema

Abbildung 3.13. I2C-Interfaces

Hinweis

Die I2C-Interface-Funktionen sind ab Version 19050600 in der DYPS Library verfügbar.

```
/** @brief  Initialize the I2C-Interface
*
*  Initializes the I2C-Interface with the given Baud-rate.
*  See RM 45.3.2 I2C Frequency Divider register
*
*  @param I2Cx_F - the Frequency Divider value
*
*  @return None
*/
void driver_i2c_init ( unsigned char I2Cx_F );

/** @brief  Write a single register
*
*  Write a byte of Data to specified register
*
*  @param slaveID - ID of the slave (7Bit)
*  @param u8RegisterAddress - is Register Address
*  @param u8Data - is Data to write
*
```

```
20  *  @return None
21  */
22  void driver_i2c_writeRegister ( unsigned char SlaveID,
23                                  unsigned char u8RegisterAddress,
24                                  unsigned char u8Data );
25
26  /** @brief  Write multiple registers
27  *
28  *  Write n bytes of Data to the slave at the specified register
29  *
30  *  @param slaveID - ID of the slave (7Bit)
31  *  @param u8RegisterAddress - is Register Address
32  *  @param u8Data - is Array of Data to write
33  *  @param len - is length of Data to write
34  *
35  *  @return None
36  */
37  void driver_i2c_write ( unsigned char SlaveID,
38                          unsigned char u8RegisterAddress,
39                          unsigned char* u8Data,
40                          unsigned char len );
41
42  /** @brief  Read a single register
43  *
44  *  Read a register
45  *
46  *  @param slaveID - ID of the slave (7Bit)
47  *  @param u8RegisterAddress - is Register Address
48  *
49  *  @return Data stored in Register
50  */
51  unsigned char driver_i2c_readRegister ( unsigned char SlaveID,
52                                          unsigned char u8RegisterAddress );
53
54  /** @brief  Read multiple registers
55  *
56  *  Read n bytes of Data from slave at the specified register
57  *
58  *  @param slaveID - ID of the slave (7Bit)
59  *  @param u8RegisterAddress - is Register Address
60  *  @param u8Data - is Array of Data to write to
61  *  @param len - is length of Data to read
62  *
63  *  @return None
64  */
65  void driver_i2c_read ( unsigned char SlaveID,
66                         unsigned char u8RegisterAddress,
67                         unsigned char* u8Data,
68                         unsigned char len );
```

Listing 3.6 Library-Funktionen: I2C-Ansteuerung

3.10 ADC

Hinweis

Die Dioden D14 und D15 sind zum Schutz der μC PINs, vor Überspannung oder Verpolung. Es handelt sich dabei um einen groben Schutz, der keine Gewährleisten mit sich bringt.

Achtung: Tiefpassfilter

Die Widerstände R7 bzw. R14 (150 Ω) bilden mit den Kondensatoren C18 bzw. C19 (100 nF) je ein Tiefpassfilter. Diese sollen HF (Hochfrequenzen) aus den Signalen filtern, und so das Rauschen verringern. Die Grenzfrequenz (Englisch: cutoff frequency) eines Tiefpassfilter kann dabei wie folgt berechnet werden:

$$fc = \frac{1}{2\pi RC}$$

Bei den vorliegenden Tiefpassfiltern werden folglich Frequenzen welche grösser sind als:

$$fc = \frac{1}{2\pi RC} = \frac{1}{2\pi\, 150\,\Omega\, 100\,\mathrm{nF}} = 10.6\,\mathrm{MHz}$$

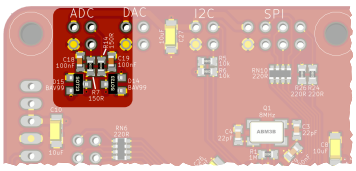
gedämpft, während Frequenzen darunter möglichst nicht beeinflusst werden.

Hinweis: 3.3 V Referenzspannung

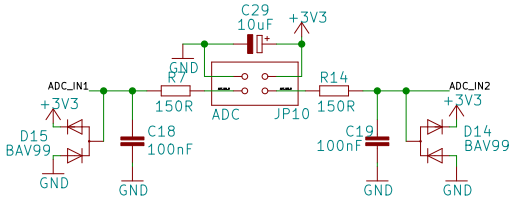
Die Referenzspannung des ADC ist auf 3.3 V festgelegt.

```
#define ADC_IN1_PIN           PIN_ALT(PIN_E24,0)
#define ADC_IN2_PIN           PIN_ALT(PIN_E25,0)
#define USE_ADC_PIN_PORT      USE_PORTE

void driver_ADC_init()
{
    SIM_SCGC6 |= SIM_SCGC6_ADC0_MASK;
    ADC0->CFG1 = ADC_CFG1_ADIV(3) | ADC_CFG1_MODE(3) | ADC_CFG1_ADICLK(0) ;
    ADC0->CFG2 = 0;
    ADC0->SC2 = 0;
    ADC0->SC3 =  0;
    ADC0->SC1[0] = ADC_SC1_ADCH(0x11);
    (void)(ADC0->R[0]);

    USE_ADC_PIN_PORT;
    INIT_ALT_PIN(ADC_IN1_PIN);
    INIT_ALT_PIN(ADC_IN2_PIN);
}

uint16_t driver_ADC_get(uint8_t CH)
{
    uint16_t res;
    ADC0->SC1[0] = ADC_SC1_ADCH(CH);
    while( !(ADC0->SC1[0] & ADC_SC1_COCO_MASK) ) __NOP(); // wait 4 conversion-complete
    res = (ADC0->R[0]);
    return res;
}

uint16_t driver_ADC_get_IN1()
{
    return driver_ADC_get( 0x11 ); // get channel SE17
}

uint16_t driver_ADC_get_IN2()
{
    return driver_ADC_get( 0x12 ); // get channel SE18
}
```

Listing 3.7 Beispielprogramm: ADC-Ansteuerung



(a) Position auf dem Board



(b) Schema

Abbildung 3.14. ADC-Interface

3.11 DAC

Achtung: Ungeschützte PINs

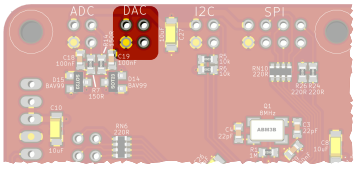
Die DAC-Pins sind aus messtechnischen Gründen nicht geschützt. Kurzschlüsse, Verpolung etc. führen zur Zerstörung dieser PINs und ggf. des μC.

Hinweis: Referenzspannung

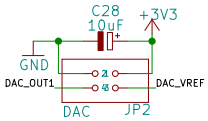
Für die Verwendung des DAC's muss dem DAC zwingend eine Referenzspannung V_{ref} oder REF (siehe DYPS ONE-Rückseite DAC) angehängt werden.

```
void driver_DAC_init()
{
    SIM_SCGC6 |= SIM_SCGC6_DAC0_MASK;
    DAC0_C0 = DAC_C0_DACEN_MASK;
    DAC0_C1 = 0;
    DAC0_C2 = 0;

    // No Pin declaration necessary as we have special DAC Pins
}

void driver_DAC_set( uint16_t value )
{
    DAC0_DAT0L = value & 0xFF ;
    DAC0_DAT0H = (value & 0x0F00)>>8 ; // 12bit DAC so take just lower nibble of 2 byte
}
```

Listing 3.8 Beispielprogramm: DAC-Ansteuerung



(a) Position auf dem Board



(b) Schema

Abbildung 3.15. DAC-Interface

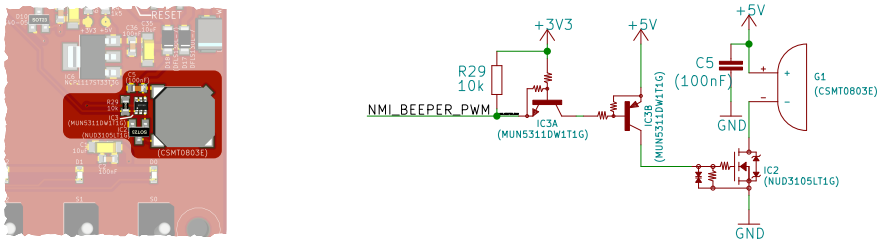
3.12 BEEPER

Optional kann ein Buzzer/Beeper auf dem DYPS ONE bestückt werden. Für die Ansteuerung wurde dabei der NMI-Pin[1] (PIN_A4) verwendet. Weil der μC eine 3.3 V Speisung aufweist und der NMI-Pin beim Start des μCs 'High' sein muss, wird eine Basis-Transistorschaltung (IC3A) als Spannungswandler auf 5 V eingesetzt[2]. Anschliessend folgt eine normale PNP-Emitter Schaltung (IC3B) welche als Inverter- und Bufferstufe dient. Sie liefert dem FET (IC2) die nötige U_{GS} für ein Schalten grösserer Lasten. Das Herzstück bildet hier der Buzzer (G1), bei dem es sich um ein CMST0803E handelt.

Hinweis

Der Buzzer kann natürlich ersetzt werden. Es gilt dann zu Beachten, dass der FET (IC2) zwar für induktive Lasten ausgelegt (Freilaufdioden) ist, jedoch einen $R_{DS(ON)}$ von ca. 500 mΩ bei $I_D = 0.5$ A aufweist.

[1] Non-Maskable-Interrupt
[2] R29 wird nur benötigt, um bei nicht bestücktem Beeper ein 'High'-Signal am NMI-Pin zu erzeugen.



(a) Position auf dem Board (b) Schema

Abbildung 3.16. Beeper

3.12.1 Ansteuerung

Da der verwendete Buzzer kein Oszillator aufweist, muss er mit einem PWM-Signal angesteuert werden. Es wird dazu der FTM0_CH1 (FlexTimer Modul 0 - Channel 1) verwendet. Da jedoch die Hintergrundbeleuchtung des TFT-Displays (PIN_A5) ebenfalls am FlexTimer Modul 0 (aber Channel 2) angeschlossen ist, muss das FTM0-Modul für beide Funktionen Konfiguriert werden.

```
/* buzzer.h */
#define BUZZER_PIN          PIN_ALT(PIN_A4,3)
#define BACKLIGHT_PWM_PIN   PIN_ALT(PIN_A5,3)
#define BACKLIGHT_DISABLE   INIT_INPUT_PULLDOWN(PIN_A5,1,0) // Release GPIO

#define INIT_BUZZER_TFT_PORT     USE_PORTA

#define BUZZER_DISABLE      INIT_INPUT_PULLUP(PIN_A4,1,0) // Release GPIO
#define BUZZER_ENABLE       INIT_ALT_PIN(BUZZER_PIN)

/* buzzer.c */
#define BUZZER_FRQ  2350

bool tpmInitialized = false;

void TPM_init(bool forced)
{
    if ( tpmInitialized & !forced )
        return;
    SIM->SCGC6 |= SIM_SCGC6_FTM0_MASK ;     // Enable FTM0
    SIM->SOPT4 |= SIM_SOPT4_FTM0CLKSEL( 1 );  // Set Clock Source

    FTM0->MOD = ( BUS_CLK_kHz * 1000 / BUZZER_FRQ );  // for 2350Hz

    //BUZZER
    FTM0->CONTROLS[1].CnSC = FTM_CnSC_MSB_MASK | FTM_CnSC_ELSB_MASK;
    FTM0->CONTROLS[1].CnV  = ( BUS_CLK_kHz*1000 / BUZZER_FRQ ) >> 1; // 50% pulse width

    //LCD_PWM
    FTM0->CONTROLS[2].CnSC = FTM_CnSC_MSB_MASK | FTM_CnSC_ELSB_MASK;
    FTM0->CONTROLS[2].CnV  = ( BUS_CLK_kHz*1000 / BUZZER_FRQ );

    FTM0->SC   = FTM_SC_CLKS(1) | FTM_SC_PS(0); // Edge Aligned PWM from BUSCLK / 1

    BACKLIGHT_DISABLE;
    BUZZER_DISABLE;
    tpmInitialized = true;
}

void driver_BUZZER_Init() {
    INIT_BUZZER_TFT_PORT;
    TPM_init(false);
    BUZZER_DISABLE;
}

void driver_Backlight_Init(unsigned char percent)
{
    INIT_BUZZER_TFT_PORT;
    TPM_init(false);
    driver_Backlight_Set(percent);
}

void driver_Backlight_Set(unsigned char percent)
{
```

```
55      FTM0->CONTROLS[2].CnV   = ( BUS_CLK_kHz*1000 / BUZZER_FRQ ) * MIN(percent, 100)/100;
56      INIT_ALT_PIN( BACKLIGHT_PWM_PIN );
57  }
```

Listing 3.9 Beispielprogramm: Beeper-Ansteuerung

3.13 RTC

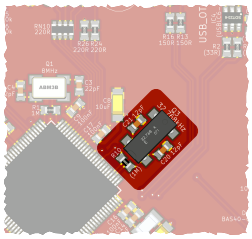


Abbildung 3.17. RTC-Komponenten auf dem Board (ohne Batteriehalter auf der Rückseite)

3.14 USB-OTG

Für eine etwas schnellere Verbindung kann die USB-OTG Schnittstelle in Betrieb genommen werden. Es muss an dieser Stelle jedoch darauf hingewiesen werden, dass die Bare-Metal-Programmierung des USB-Stacks eine sehr Zeitintensive Angelegenheit ist. Aus diesem Grund wird hier nur zur Vollständigkeit auf die Position der USB-OTG Schnittstelle und deren Schema hingewiesen.

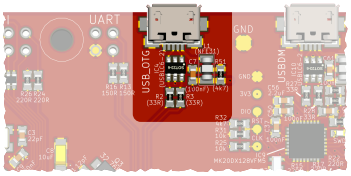


Abbildung 3.18. Die USB-OTG-Schnittstelle auf dem DYPS ONE

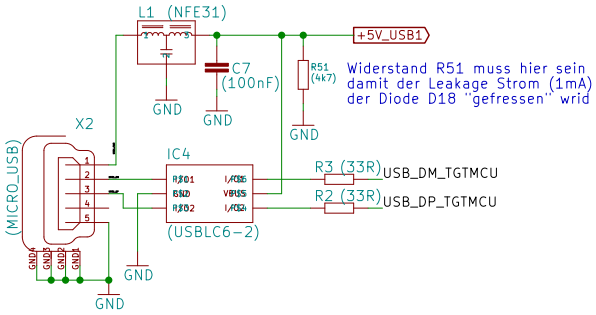


Abbildung 3.19. Schema der USB-OTG-Schnittstelle

3.15 Speisung

Das DYPS ONE wird im Normalfall über die USB-Verbindung des USBDM-Interfaces oder der USB-OTG-Verbindung mit 5 V versorgt. Zusätzlich kann es aber auch über den Anschluss JP11 mit einer Spannung von 5 V versorgt werden. Die Dioden D16, D17 und D18 erlauben es dabei mehrere Speisungen gleichzeitig anzuschliessen.

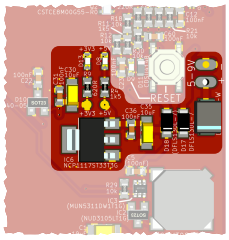


Abbildung 3.20. Speisung des DYPS ONE

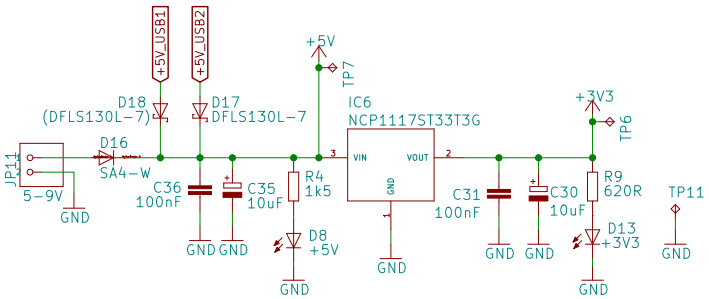


Abbildung 3.21. Schema der Speisung

Achtung: Spannungen > 5V

Sofern keine Erweiterung am DYPS angeschlossen ist und die Beeper-Umgebung auf dem Board nicht bestückt ist, können auch Spannungen von bis zu 9 V angeschlossen werden. Generell ist dies jedoch nicht zu empfehlen, da es sich beim nachfolgendem Spannungsregler IC6 um einen Linear-Regler handelt.

Kapitel 4

DYPS-TOUCH

4.1 Hardware

Als Option kann das DYPS-TOUCH für Ein- und Ausgaben benutzt werden. Dazu wird es gemäss Abbildung 5.1 auf das DYPS ONE gesteckt.

Achtung

Das DYPS-TOUCH wird von der Rückseite gemäss Abbildung 5.1 auf das DYPS ONE gesteckt.

Das DYPS-TOUCH wurde aus Kostengründen gezielt als Erweiterung entwickelt und kann zusätzlich erworben werden.

Obschon der Display-Treiber sehr effizient geschrieben wurde, entstehen durch den Update des Displays Verzögerungen. Diese Verzögerung ist natürlich abhängig von der Aktualisierungs-Menge welche zwischen 0.2 ms (kein Update) bis 160 ms (Update des kompletten Displays) variieren. Da jedoch meistens nur wenige Änderungen vorhanden sind, dürfte sich eine mittlere Update-Geschwindigkeit von <=30 ms ergeben. Änderungen der Anzeige welche dabei unter dieser Zeit liegen, werden ggf. nicht korrekt durchgeführt bzw. können zu falschen Anzeigen führen.

Hinweis

Es sollten Änderungen mit Zeiten <=30 ms vermieden werden.

Hinweis: Single-Touch

Bei der Touchfläche handelt es sich um einen resistiven Touchsensor welcher nur Single-Touch-Fähig ist. Ein gleichzeitiges Drücken / Auslösen mehrerer Tasten ist demzufolge nicht möglich.

Für die Auswertung dieses Touchsensors wird der TSC2046 verwendet. Dieser wird jeweils bei jedem zweiten Aktualisierungsvorgang abgefragt.

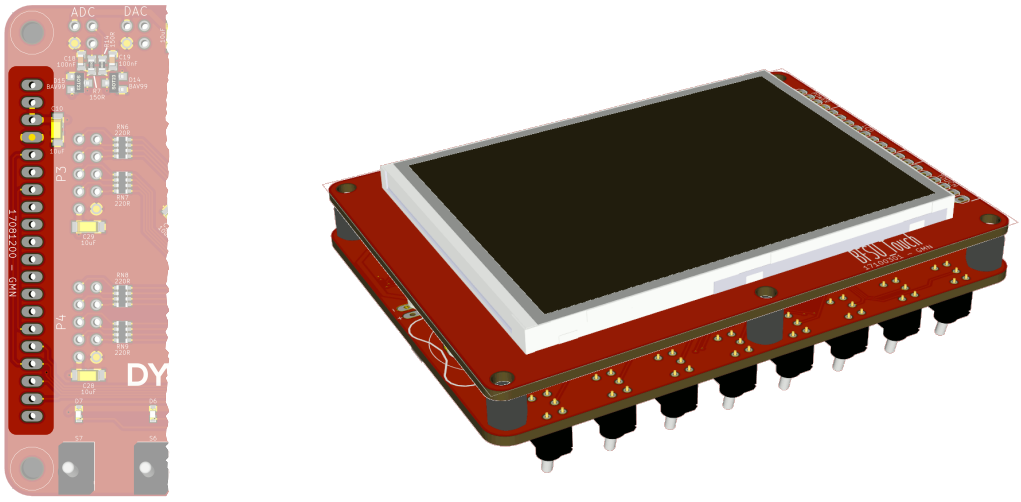
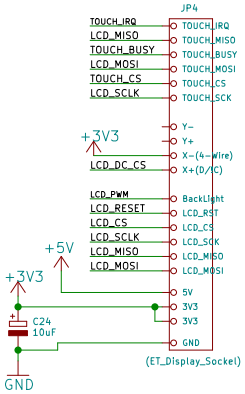


Abbildung 4.1. Display-Port auf dem DYPS ONE und 3D-Ansicht des DYPS ONE mit aufgestecktem DYPS-TOUCH-Display



4.2 Software

4.2.1 P0P1-Ansteuerung

Damit das TFT-Display benutzt werden kann, muss es zuvor durch `initTouchP0P1()` initialisiert werden. Dabei können, zur Konfiguration, bei der Initialisierung verschiedene Parameter übergeben werden. Folgend eine Zusammenfassung der Parameter:

d [Debug] Initialisiert die USBDM-UART-Schnittstelle (115200-8-N-1) für Debug Meldungen. (Siehe Abschnitt 3.6).

c [Calibrate] Führt beim Start eine Kalibration des Touchscreens durch.

p [Portexchange] Position von P0 und P1 austauschen.

r [Rotate] Rotiert die Anzeige bei jedem aufkommen dieses Zeichens um jeweils 90 Grad.

m [Mirrored] Die Schalter und LEDs werden in umgekehrter Reihenfolge angezeigt.

1-4 [Schaltertyp] 1 = OFF-ON , 2 = ON-OFF , 3 = OFF-ON-OFF , 4 = ON-OFF-ON

Es ist dabei erlaubt mehrere Parameter hintereinander zu übergeben.

```
/*
Titel:      BspProgramm Ausgangs-Port
Datei:      P0toP1.c
Ersteller:  R.Gassmann
Funktion:   Liest die Schalter an Port 0 ein und gibt diese an Port 1 aus
*/

// Einbindung der Bibliotheken
#include "libDYPS.h"

// Hauptprogramm
int main(void) {
    initTouchP0P1("mp3");    // Display, P0 und P1 initialisieren
    while (1) {              // Endlosschleife
        P1 = P0;             // Ausgabe der Schalter auf den LEDS
    }
}
```

Listing 4.1 Beispielprogramm: Ausgangs-Port

4.2.2 Erweiterte Funktionen

4.2.2.1 Tasten-/LED-Bezeichnungen

Mittels der DYPS Library-Funktion `setP0Caption()` bzw. `setP1Caption()` können die Tasten-Bezeichnungen (T0...T7) bzw. die LED-Bezeichnungen (0...7) abgeändert werden. Die Bezeichnung kann dabei aus maximal 3 Zeichen bestehen.

Hinweis

Diese Funktion ist ab Version 18092400 verfügbar.

```
setP0Caption("++", "+", "-", "--", "", "", "ERR", "EIN");
```

Listing 4.2 Beispielprogramm: Tastenbezeichnung

4.2.2.2 Tasten-/LED-Farben

Mittels der DYPS Library-Funktion `setP0Colors()` bzw. `setP1Colors()` können die Tasten-Farben bzw. LED-Farben abgeändert werden. Der erste Parameter entspricht dabei der Maske, wobei jedes Bit dieser Maske der entsprechenden Taste / LED entspricht. Es ist damit möglich die selben Farben mehreren Tasten/LEDs gleichzeitig zu zuordnen.

Hinweis

Diese Funktion ist ab Version 18092400 verfügbar.

```
setP0Colors( 0x09 , COLOR_CYAN_DARK , COLOR_CYAN_BRIGHT); // T0 + T3 CYAN
```

Listing 4.3 Beispielprogramm: Tastenfarbe

4.2.2.3 Tasten Typ

Mittels der DYPS Library-Funktion `setP0SwitchType()` bzw. `setP0SwitchTypeMask()` kann der Typ jeder Taste bzw. einer Tasten-Auswahl bestimmt werden.

Hinweis

Diese Funktionen sind ab Version 19050600 verfügbar.

```
setP0SwitchType ( switch_pos, switch_neg , taster_pos , taster_neg ,
                  switch_pos, switch_neg , taster_pos , taster_neg );

setP0SwitchTypeMask ( 0x05, switch_neg ); // Set S0 and S3 as negative switch
```

Listing 4.4 Beispielprogramm: Eigene Ansteuerung

4.2.3 Eigene Ansteuerung

Es ist denkbar, dass das DYPS-TOUCH auch für andere Zwecke verwendet wird. Dazu steht die Funktion `initTouch()` zur Verfügung. Sie initialisiert das Display mit Touch-Interface und startet je nach Vorgabe einen Timer welcher dann periodisch, die bei der Initialisierung übergebene Funktion, aufruft.

Hinweis

Diese Funktion ist für fortgeschrittene Programmierer. Diese Funktion ist ab Version 18092400 verfügbar.

```
#include "../system/lib/libDYPS.h"

void updateFrame(){
    static uint16_t color = 0x0000;
    color+=100;
    clearScreen(color);
}

int main(void) {
  initTouch("",&updateFrame);
  while(1){
  }
}
```

Listing 4.5 Beispielprogramm: Eigene Ansteuerung

Kapitel 5

DYPS - TRAFFIC LIGHT

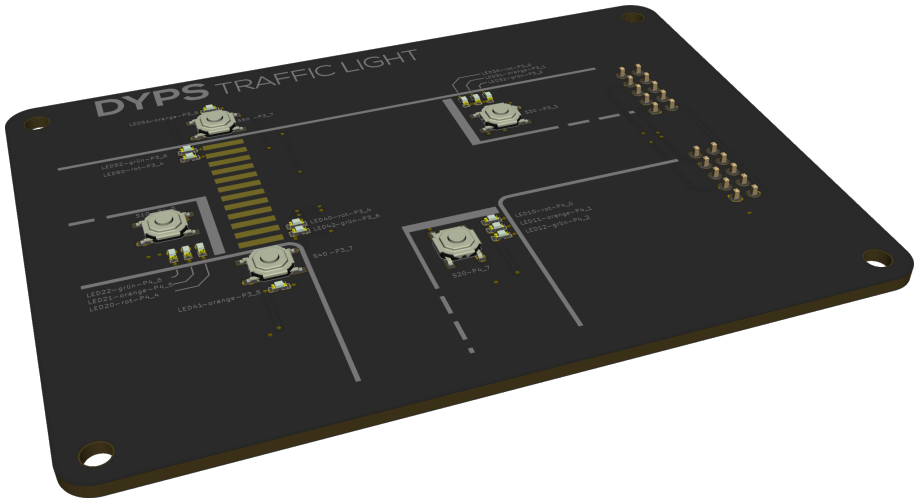


Abbildung 5.1. DYPS TRAFFIC LIGHT Erweiterung

In vielen μC-Anwendungen sind strikte Abläufe vorhanden welche gezielt umgesetzt werden müssen. Essentiell dabei ist der Aufbau / die Struktur des Programms. Die Traffic Light Erweiterung sollte genau dies Aufzeigen.

Achtung

Das DYPS TRAFFIC LIGHT wird wie schon das DYPS-TOUCH von der Rückseite auf das DYPS ONE gesteckt.

Das DYPS TRAFFIC LIGHT wurde gezielt als Erweiterung entwickelt und kann zusätzlich erworben werden. Es wird über die GPIO-Ports `P3` und `P4` des DYPS angesteuert (siehe Abschnitt 3.4.1 und 3.4.2).

5.1 Hardware

5.1.1 Schema

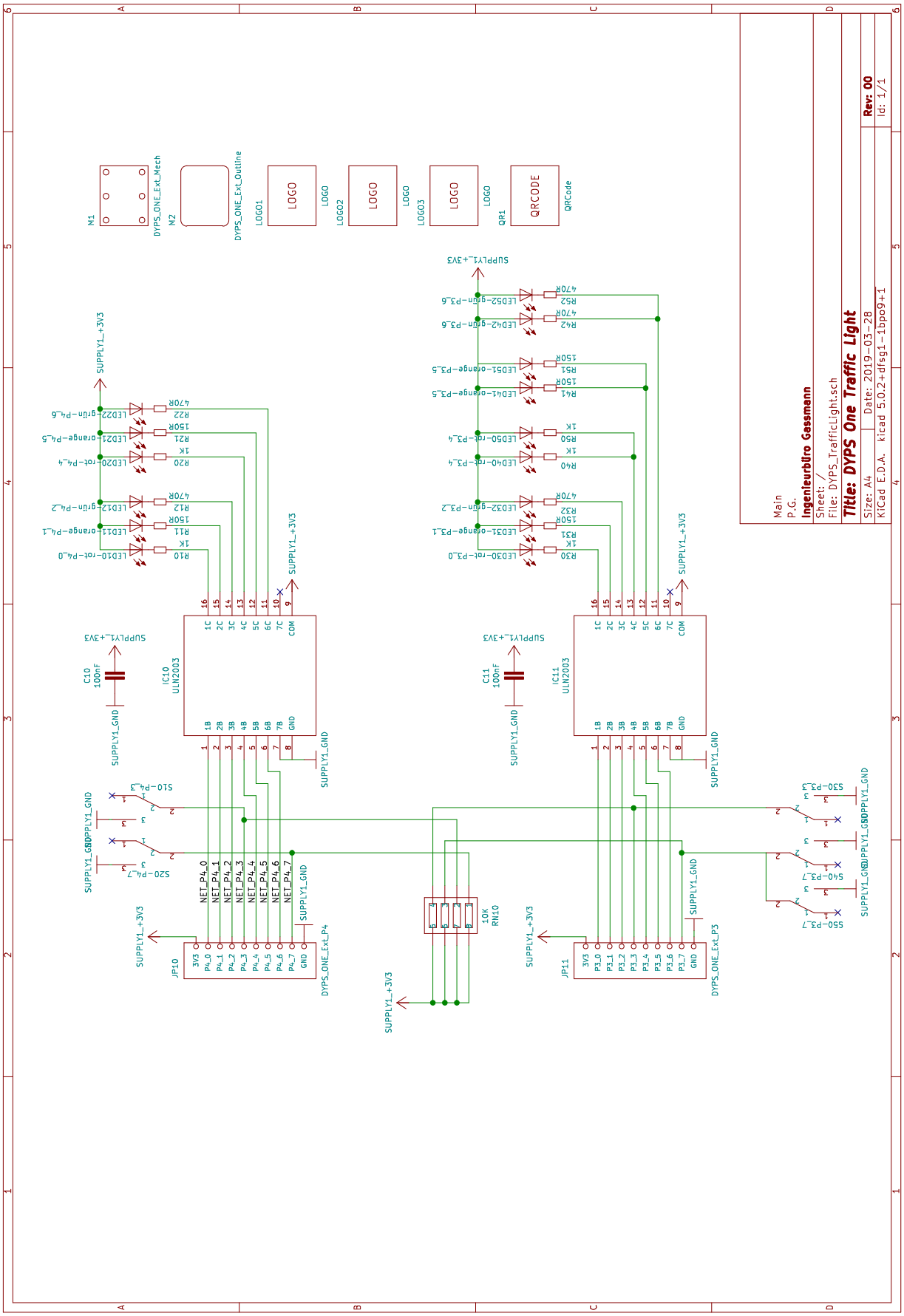


Abbildung 5.2. DYPS - TRAFFIC LIGHT Schema

5.2 Ansteuerung

```
/*
Titel:      BspProgramm Traffic Light
Ersteller:  R.Gassmann
Funktion:   Initialisiert P3 + P4 für die Traffic Light Extension und schaltet alle LED's auf rot.
*/

// Einbindung der Bibliotheken
#include "libDYPS.h"

// Hauptprogramm
int main(void) {

    initP0P1();          // initialisiert MCU (und die nicht benötigten Ports P0 und P1)
    initP3 ( 0x88 ) ;    // initialisiert P3 wobei die obersten Pins jeder Tetrade als Input
                         // gesetzt werden (für die Taster-Eingänge).
    initP4 ( 0x88 ) ;    // initialisiert P4 wobei die obersten Pins jeder Tetrade als Input
                         // gesetzt werden (für die Taster-Eingänge).

    while (1) {      // Endlosschleife
        P3OUT = 0x11;    // östliche und Fussgänger-Ampel auf rot
        P4OUT = 0x11;    // südliche und westliche Ampel auf rot
    }
}
```

Listing 5.1 Beispielprogramm 1: Traffic Light

Hinweis

Da die Verdrahtung der einzelnen Ampeln einheitlich Aufgebaut ist (Anschluss entweder P3/4_0-2 oder 4-6 und identische Farbreihenfolge) ist die Benutzung der Definitionen `P3_#_OUT` bzw. `P4_#_OUT` hier klar nicht sinnvoll. Es sollte vielmehr überlegt werden, wie dies für eine bessere Programm-Übersicht ausgenutzt werden könnte. Mehr dazu in Abschnitt 5.4.

5.3 Aufgaben

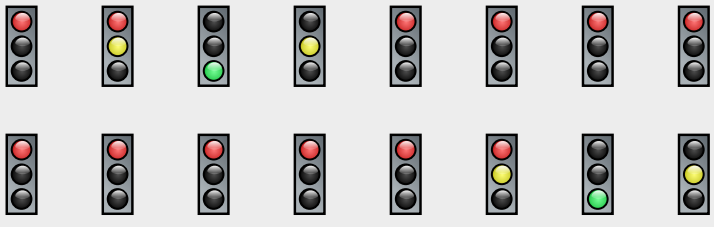
Die folgenden Aufgaben bauen auf einander auf. Es wird daher empfohlen sie nach einander zu lösen.

Aufgabe 5.3.1: Einzelne Ampel

Steuern Sie die östliche Ampel gemäss folgendem Ablauf.

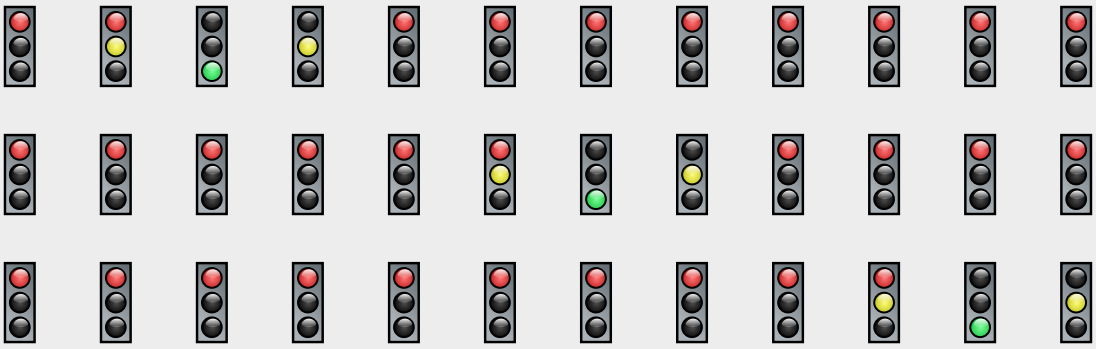
Aufgabe 5.3.2: Zweifach Ampel

Erstellen Sie ein Programm welches die östliche und westliche Ampel wie folgt bedient.



Aufgabe 5.3.3: Dreifach Ampel

Erstellen Sie ein Programm welches die östliche, westliche und südliche Ampel wie folgt bedient.

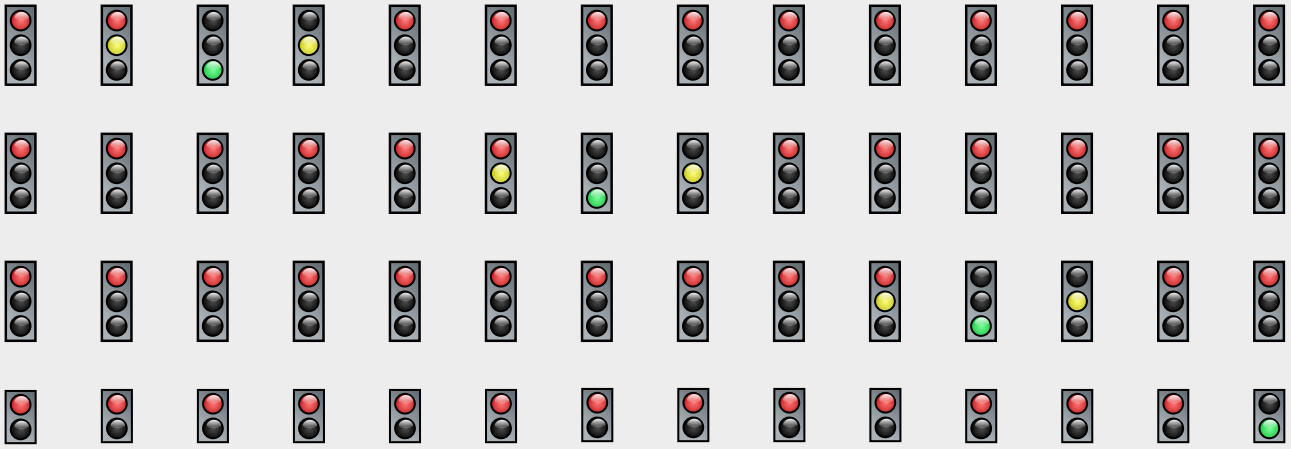


Aufgabe 5.3.4: Dreifach Ampel Erweitert

Erstellen Sie ein Programm welches im Normalfall die östliche und westliche Ampel bedient (gemäss Aufgabe "Zweifach Ampel"). Sollte jedoch ein Fahrzeug an der südlichen Ampel stehen (Taster gedrückt), soll die südliche Ampel in den Ablauf aufgenommen werden (gemäss Aufgabe "Dreifach Ampel").

Aufgabe 5.3.5: Vierfach Ampel

Erstellen Sie ein Programm welches die östliche, westliche, südliche und Fussgänger (nur rot und grün) Ampel wie folgt bedient.



Aufgabe 5.3.6: Vierfach Ampel Erweitert 1

Erstellen Sie ein Programm welches alle vier Ampel bedient wobei die Fussgänger immer zur selben Zeit über die Strasse dürfen in der die Fahrzeuge aus dem Süden fahren dürfen.

Hinweis: Orange Blinken
Denken Sie daran, in diesem Falle sollte die südliche Ampel nicht mehr auf grün gehen, sondern orange blinken.

Aufgabe 5.3.7: Vierfach Ampel Erweitert 2

Erstellen Sie ein Programm welches im Normalfall die östliche, westliche und südliche Ampel bedient (gemäss Aufgabe "Dreifach Ampel"). Sollte jedoch ein Fussgänger anmelden (über eine der beiden Fussgänger-Tasten an Port `P3_7`) wird dies sofort registriert und mittels der Orangen LED der Fussgänger-Ampel bestätigt. Sobald der Fussgänger nun in den Ablauf passt. Wird dieser Bedient (gemäss Aufgabe "Vierfach Ampel"). Sobald der Fussgänger bedient wurde, muss dessen Registrierung natürlich wieder gelöscht werden.

5.4 Beispielcode

5.4.1 Ampelansteuerung

Da wie bereits erwähnt alle Ampeln einheitlich verdrahtet sind, sollte etwas Zeit investiert werden um zu überlegen ob eine einheitliche Ansteuerung für alle Ampeln gefunden werden kann. Dies könnte die Programm-Übersicht massiv steigern und damit die Entwicklungszeit und auch die Zeiten für spätere Anpassungen massiv verringern. Eine Art "Farb-Angabe" pro Ampel wäre folglich anzustreben.

Aus Schema und Layout wird ersichtlich, dass bei allen Ampeln die rote LED dem LSB und die Grüne dem MSB entspricht. Mit dieser Information und bekanntem Datentyp der Aufzählung (`enum`), könnte versucht werden die einzelnen Zustände einer einzelnen Ampel à 3 Lichtern (⇒ 8 Zustände) durch einen `enum` zu definieren. Einzig die richte Farbzuordnung zu den einzelnen Werten ist zu definieren. Dies ist mit all den Informationen aus Schema und Layout relativ einfach möglich. Der enum-Typ `trafficlight_t` würde dann wie folgt aussehen:

```
typedef  enum {
    off,                // 0x00 => keine LED
    red,                // 0x01 => rote LED
    orange,             // 0x02 => orange LED
    red_orange,         // 0x03 => rot + orange LED
    green,              // 0x04 => grüne LED
    red_green,          // 0x05 => rote + grüne LED
    orange_green,       // 0x06 => orange + grüne LED
    red_orange_green    // 0x07 => alle LED
} trafficlight_t;
```

Listing 5.2 Enum für die Ampel-Farben

Es können anschliessend die 4 Ampeln der Traffic Light Erweiterung (vom Typen `trafficlight_t`) wie folgt deklariert und initialisiert werden:

```
trafficlight_t lightEast  = red;    // östliche Ampel
trafficlight_t lightPed   = red;    // Fussgänger Ampel
trafficlight_t lightWest  = red;    // westliche Ampel
trafficlight_t lightSouth = red;    // südliche Ampel
```

Listing 5.3 Deklaration und Initialisierung der 4 Ampeln

Einzig die richtigen Zuweisungen an den Ports `P3` und `P4` sind jetzt noch zu erledigen. Auch hier muss das Schema/Layout zur Hand genommen werden. Es ist aus ihnen ersichtlich, dass die östliche Ampel an Port `P3_0-P3_2` und die Fussgänger-Ampel an Port `P3_4-P3_6` angeschlossen sind. Es kann folglich das `lightEast` direkt und das `lightPed` um 4 stellen geschoben an Port `P3` übergeben/gesetzt werden. Analog sieht dies für die südliche und westliche Ampel an Port `P4` aus. Für die Ausgabe sind daher noch folgende Programmzeilen nötigen:

```
P3OUT = lightEast | ( lightPed << 4 );
P4OUT = lightSouth | ( lightWest << 4 );
```

Listing 5.4 Ausgabe an den Ports P3 und P4

5.4.2 Taster

Genau wie bei der Ansteuerung der Ampel sollte bei den Tasteingängen auch überlegt werden, wie diese möglichst einfach in einen Programm Ablauf aufgenommen werden könnten. Da die 4 Tasten über 2 Bytes "verstreut" sind, könnte es hilfreich sein, diese zuerst zu einer Tetrade zusammen zu führen und anschliessend mit dieser Tetrade zu arbeiten. Dies könnte wie folgt aussehen:

```
tst.act = (!P3_3_IN) | ( !P3_7_IN << 1 ) | ( !P4_3_IN << 2 ) | ( !P4_7_IN << 3 );
```

Listing 5.5 Zusammenzug der Tasteingänge

Mit den nachstehenden Definitionen können anschliessend durch Maskierungen die einzelnen Taster abgefragt werden.

```
#define TST_CAR_EAST    0x01
#define TST_PED         0x02
#define TST_CAR_WEST    0x04
#define TST_CAR_SOUTH   0x08
```

Listing 5.6 Definitionen für die Maskierungen

Eine Maskierung für die Fussgänger-Tasten würde wie folgt aussehen:

```
if ( tst.act & TST_PED ){
...
}
```

Listing 5.7 Taster Maskierung

Anhang A

Portierung

Dieses Kapitel befasst sich mit der Portierung des Codes. Dabei sind (zur Zeit) zwei Szenarien denkbar.

A.1 MCB32 → DYPS

Um einen Code der auf dem MCB32 geschrieben wurde auf das DYPS zu portieren sind folgende Schritte nötig:

Neues DYPS-Projekt Zu Beginnn wird ein neues DYPS-Projekt gemäss Anleitung Abschnitt 2.2 erstellt. Achten Sie auf die bereits jetzt auf die richtige Namensgebung des Projekts. Dies kann zwar später noch angepasst werden führt aber zu unnötigem Aufwand.

Code Kopieren Kopieren Sie nun den Code des MCB32-Projekts in das DYPS-Projekt. Wahlweise durch das Kopieren des Inhaltes der main.c Datei oder durch das Kopieren der ganzen Datei. Besteht das Projekt aus mehreren C-Dateien, so sind diese und deren H-Dateien ebenfalls zu Kopieren. Des weiteren müssen diese im Codeblocks ins DYPS-Projekt aufgenommen werden.

Includes Anpassen Das DYPS-Board arbeitet mit einer anderen Bibliothek im Hintergrund. Es ist daher wichtig, die Include-Anweisungen entsprechend anzupassen. Ändern Sie die `include`-Anweisungen von

```
1 #include <stm32f10x.h>
2 #include "TouchP0P1.h"
```

Listing A.1 MCB32-Include

auf

```
1 #include "../system/lib/libDYPS.h"
```

Listing A.2 DYPS-Include

Code Anpassen Bei den Bibliotheksfunktionen wurden die Interface-Routinen neu geschrieben. In diesem Schritt wurde auch gleich eine einheitliche Namensgebung eingepflegt. Neu sind daher alle Name der Bibliotheksfunktionen im camelCase (auch camelStyle genannt) geschrieben oder besitzen bei erweiterten Funktionen "_" als Wort-Trenner. Bei kleineren Programmen, bei denen lediglich `InitTouchP0P1(...);` bzw. `InitP0P1();` aus der Bibliothek aufgerufen werden, sind lediglich diese entsprechend durch `initTouchP0P1(...);` resp. `initP0P1();` zu ersetzten.

Bei grösseren Programmen sind evt. noch weitere Funktionen anzupassen. Diese werden aber durch den Compiler angegeben und müssen entsprechend angepasst / neu geschrieben werden.

A.2 DYPS → MCB32

Um einen Code der auf dem DYPS geschrieben wurde auf das MCB32 zu portieren sind folgende Schritte nötig:

Neues MCB32-Projekt Zu Beginnn wird ein neues MCB32-Projekt gemäss Anleitung MCB32-Anleitung erstellt. Achten Sie auf die bereits jetzt auf die richtige Namensgebung des Projekts. Dies kann zwar später noch angepasst werden führt aber zu unnötigem Aufwand. Vergessen Sie dabei nicht die Bibliothek also TouchP0P1.h und TouchP0P1.lib einzubinden.

Code Kopieren Kopieren Sie nun den Code des DYPS-Projekts in das MCB32-Projekt. Wahlweise durch das Kopieren des Inhaltes der main.c Datei oder durch das Kopieren der ganzen Datei. Besteht das Projekt aus mehreren C-Dateien, so sind diese und deren H-Dateien ebenfalls zu Kopieren. Des weiteren müssen diese im Kile ins MCB32-Projekt aufgenommen werden.

Includes Anpassen Das MCB32-Board arbeitet mit einer anderen Bibliothek im Hintergrund. Es ist daher wichtig, die Include-Anweisungen entsprechend anzupassen. Ändern Sie die `include`-Anweisungen von

```
1 #include "../system/lib/libDYPS.h"
```

Listing A.3 DYPS-Include

auf

```
1 #include <stm32f10x.h>
2 #include "TouchP0P1.h"
```

Listing A.4 MCB32-Include

Code Anpassen Bei den Bibliotheksfunktionen des DPYS wurden die Interface-Routinen neu geschrieben. In diesem Schritt wurde auch gleich eine einheitliche Namensgebung eingepflegt. In der DYPS-Umgebung sind daher alle Name der Bibliotheksfunktionen im camelCase (auch camelStyle genannt) geschrieben oder besitzen bei erweiterten Funktionen "_" als Wort-Trenner. Bei kleineren Programmen, bei denen lediglich `initTouchP0P1(...);` bzw. `initP0P1();` aus der Bibliothek aufgerufen werden, sind lediglich diese entsprechend durch `InitTouchP0P1(...);` resp. `InitP0P1();` zu ersetzten.

Bei grösseren Programmen sind evt. noch weitere Funktionen anzupassen. Diese werden aber durch den Compiler angegeben und müssen entsprechend angepasst / neu geschrieben werden.

A.3 Beispiel

Der MCB32 Code A.5 wird zum DYPS Code A.6 und umgekehrt.

```
1 #include <stm32f10x.h>
2 #include "TouchP0P1.h"
3
4 int main (void) {
5     InitTouchP0P1("mp3");
6     while (1) {
7         P1 = P0;
8     }
9 }
```

Listing A.5 MCB32-Code

⇔

```
1 #include "../system/lib/libDYPS.h"
2
3 int main (void) {
4     initTouchP0P1("mp3");
5     while (1) {
6         P1 = P0;
7     }
8 }
```

Listing A.6 DYPS-Code

Anhang B

Der Übersetzungsvorgang

Dieses Kapitel befasst sich mit den unterschiedlichen Arten des Übersetzungsvorgangs von C-Programmen mit und ohne `makefile`. Wichtig dabei ist, zu verstehen wie die einzelnen Schnitte voneinander abhängen.

B.1 Einfaches C-Programm

Bei einfachen C Programmen wie beispielsweise:

```
1  /* Titel:        Hallo World
2   * Datei:        main.c
3   * Ersteller:    R.Gassmann
4   * Funktion:     Gibt "Hello World" auf der Kommandozeile aus.
5   */
6
7  #include<stdio.h>
8
9  int main() {
10     printf("Hello World\n");
11     return 0;
12 }
```

Listing B.1 Beispielprogramm: Hello World

ist bereits bekannt, dass sie mit dem Compiler mittels

```
gcc -c main.c
```

übersetzt werden können.

Achtung: Case-Sensitivity

Im Gegensatz zu Windows ist Linux Case-Sensitiv, d.h. es wird zwischen Gross- und Kleinschreibung unterschieden. Grundsätzlich wird daher empfohlen auf die Gross- und Kleinschreibung zu achten, um damit die Portierbarkeit der Programme sicherzustellen.

Hinweis: Option -c

Die Option `-c` gibt lediglich an, dass die Datei nur kompiliert nicht aber gelinkt werden soll. Würde diese Option weggelassen würde die Datei auch gelinkt und es würde die ausführbare Datei `a.out` entstehen.

Die durch die Kompilierung entstandene Objektdatei `main.o` kann noch nicht ausgeführt werden. Es fehlt dazu noch der Link-Vorgang, welcher alle nötigen Funktionen zusammen, in eine ausführbare Datei, linkt. Dieser Vorgang wird unter Windows mittels

```
gcc -o main.exe main.o
```

oder unter Linux mittels

```
gcc -o main main.o
```

durchgeführt. Die entstehende ausführbare Datei "main.exe" bzw. "main" entspricht nun dem gewünschten Programm.

Hinweis: Option -o

Mit der Option `-o` wird dem Linker mitgeteilt, dass die ausführbare Datei den Namen `main.exe` bzw. `main` erhalten soll.

Beide Schritte, also das Kompilieren und Linken, kann dabei auch in einem Schritt zusammengefasst werden. Dieser würde dann wie folgt aussehen:

```
gcc -o main.exe main.c
```

bzw.

```
gcc -o main main.c
```

B.2 Programm mit zwei Dateien

Nachdem der Übersetzungsvorgang eines einfachen C-Programms gezeigt wurde, wird ein Fall mit zwei Source-Dateien `main.c` (Listing B.4) und `func.c` (Listing B.2) betrachtet.

Es wird dazu die Datei `func.c` (Listing B.2) angeschaut. Sie beinhaltet lediglich die Funktion `helloWorld()`, welche von der Applikation (main) aufgerufen wird.

```
1  /* Titel:   Func-Source
2   * Datei:   func.c
3   * Ersteller:   R.Gassmann
4   * Funktion:    Datei mit Funktion
5   */
6
7  #include<stdio.h>
8
9  void helloWorld() {
10     printf("Hello World\n");
11 }
```

Listing B.2 Beispielprogramm: Hello World 2, func.c

Damit die Funktion `helloWorld()` jedoch in der Datei `main.c` bekannt ist, muss noch eine passende Header-Datei (`func.h`) dazu erstellt werden.

```
1  /* Titel:   Func-Header
2   * Datei:   func.h
3   * Ersteller:   R.Gassmann
4   * Funktion:    Datei mit Funktion
5   */
6
7  void helloWorld();
```

Listing B.3 Beispielprogramm: Hello World 2, func.h

Diese muss anschliessend in der Hauptsource-Datei mittels `#include` eingebunden werden.

```c
/* Titel:       Hallo World
 * Datei:       main.c
 * Ersteller:   R.Gassmann
 * Funktion:    Gibt "Hallo World" auf der Kommandozeile aus.
 */

#include "func.h"

int main() {
    helloWorld();
    return 0;
}
```

Listing B.4 Beispielprogramm: Hello World 2, main.c

Soll nun dieses Programm übersetzt werden, so muss der Kompilier-Schritt aus Abschnitt B.1 für jede Datei (main.c und func.c) und anschliessend der Link-Schritt mit allen Objektdateien durchgeführt werden. Dies würde also unter Linux wie folgt aussehen:

```
gcc -c main.c
gcc -c func.c
gcc -o HelloWorld main.o func.o
```

Listing B.5 Kompilier und Link Befehle

Achtung: Reihenfolge der Parameter

Ein häufiger Fehler beim Linkbefehl ist eine falsche Reihenfolge der Parameter. Es gilt dabei immer folgender Aufbau:

```
gcc -o ProgrammName Objektfile_1.o Objektfile_2.o ... Objektfile_n.o
```

Da die Kompilier-Befehle mit jeder zusätzlichen Source-Datei länger und komplexer werden, wäre eine Art "Automatismus" schön. Hierzu kann ein sogenanntes `makefile` geschrieben/verwendet werden, welches den ganzen Übersetzungsvorgang koordiniert. Mehr dazu im nächsten Abschnitt.

B.3 Das Makefile

Ein Makefile ist im Grunde nichts anderes als eine Textdatei, welche eine Reihe von Konsolen-Befehlen beinhaltet. Es können dabei Abhängigkeiten definiert werden, so dass die einzelnen Aufrufe in einer bestimmten Reihenfolge ablaufen. Sollten die Abhängigkeiten aus Dateien bestehen (beispielsweise eine Objektdatei), wird geprüft ob diese bereits existiert. Falls dies der Fall ist, wird weiter geprüft, ob die Datei aktuell ist oder ob sich die Sourcedatei inzwischen verändert hat und damit eine erneute Kompilierung nötig ist. Überflüssige Übersetzungen lassen sich somit vermeiden.

Die Interpretation und Ausführung des `makefile`s wird durch das Programm `make`[1] erledigt. Das `make`-Programm sucht bei dessen Aufruf selbständig nach einer Datei mit dem Namen `makefile` oder `Makefile` und führt die Schritte aus.

```
make
```

Hinweis: Dateiname

In dieser Anleitung wird das Makefile immer klein geschrieben. Generell ist aber jeder Dateiname erlaubt. Er muss dann jedoch bei der Ausführung von `make` mit der Option `-f` angegeben werden.

```
make -f MakefileName
```

Das `makefile` wird im Wesentlichen mit vier verschiedene Elementen aufgebaut, wobei diese jeweils mehrmals verwendet werden können:

[1] Es gibt auch Umgebungen in denen hierfür das `nmake` oder `nm` bereit gestellt werden.

Kommentare sind für Erläuterungen u.ä.
Beispiel:`# Nur zur Info`

Definitionen von Variablen und Funktionen
Beispiel:`CC = gcc -Wall`

Includes Beispiel:`-include Makefile.local`

Regeln bilden die Kernteile des `makefile`. Eine Regel ist wie folgt aufgebaut:

```
Target [weitere Targets]:[:] [Vorbedingungen] [; Kommandos]
[<Tab> Kommandos]
[<Tab> Kommandos]
```

Achtung: Tabulatoren

Ab der zweiten Zeile muss vor dem Kommando immer ein Tabulator(`<Tab>`) stehen. Dieser darf vom Editor nicht durch Leerschläge ersetzt werden.

Ein kleines Beispiel:

```
test:
    @echo Dies ist ein Test.
test2:
    @echo Dies ist ein anderer Test.
```

Es wurden hier die zwei Regeln `test` und `test2` definiert. Diese können nun gezielt aufgerufen werden:

```
$ make test2
Dies ist ein anderer Test.
```

oder

```
$ make test
Dies ist ein Test.
```

Es bildet dabei die erste Regel im makefile automatisch die Hauptregel. Das heisst, wird nur `make` aufgerufen, wird im makefile automatisch die erste Regel (oder eben die Hauptregel) durchgeführt.

B.3.1 Beispiel Makefile

In diesem Abschnitt wird ein Makefile für das Beispiel aus Abschnitt B.2 mit drei Files (main.c, func.c und func.h) erstellt und erläutert. Gestartet wird dazu mit der Hauptregel `HelloWorld` welche als Startbedingung die Dateien `main.c` und `func.c` benötigt. Darunter sind die Befehle für das Kompilieren und Linken zu finden, welche analog zu den Befehlen aus Listing B.5 sind.
Zusätzlich wird die Regel `clean` definiert. Diese erleichtert lediglich das Löschen von den erzeugten Objektdateien und dem Programm.

```
# Makefile für Beispielprogramm

HelloWorld: main.c func.c
    gcc -c main.c
    gcc -c func.c
    gcc -o HelloWorld main.o func.o

clean:
    rm -f *.o HelloWorld
```

Listing B.6 Erstes Makefile

Ein Übersetzten der Source kann nun mittels dem Befehl `make` ausgelöst werden.

```
$ make
gcc -c main.c
gcc -c func.c
gcc -o HelloWorld main.o func.o
```

Da das makefile von Listing B.6 jedoch alle Befehle untereinander aufgelistet hat, würde im Falle eines Fehler in Datei main.c die weiteren Befehle `gcc -c func.c` und `gcc -o HelloWorld main.o func.o` dennoch ausgeführt, was natürlich besonders bei grossen Programmen, bei denen ein Übersetzten mehrere Minuten oder sogar Stunden dauert, nicht viel bringen. Das makefile wird deshalb mit den Regeln `main.o` und `func.o` erweitert. Neu hängt dabei die Hauptregel nicht mehr von den Dateien `main.c` und `func.c` sondern von deren Objektdateien `main.o` und `func.o` ab. Des weiteren soll diese Regel bloss noch den Link-Befehl (Befehl mit `-o`) beinhalten. Die Kompiler-Befehle (Befehle mit `-c`) werden in den neuen Regeln definiert. Es wird also `main.o` aus der Datei `main.c` mittels dem darunter stehenden Befehl (mit -c für das Compilieren) erstellt. Die Regel für die Datei `func.o` ist dabei identisch aufgebaut.

```
# Makefile für Beispielprogramm

HelloWorld: main.o func.o
    gcc -o HelloWorld main.o func.o

main.o: main.c
    gcc -c main.c

func.o: func.c
    gcc -c func.c

clean:
    rm -f *.o HelloWorld
```

Ein Übersetzten der Source kann mit dem selben Befehl `make` ausgelöst werden.

```
$ make
gcc -c main.c
gcc -c func.c
gcc -o HelloWorld main.o func.o
```

Der grosse Vorteil liegt nun darin, dass bei einem Fehler der Übersetzungsvorgang an der Stelle des Fehlers abgebrochen wird. Dies kann gezeigt werden indem in der Datei `main.c` anstelle der Funktion `helloWorld` die nicht existierende Funktion `helloWorlds` aufgerufen wird. Ein Übersetzten würde dann folgende Ausgabe erzeugen.

```
$ make
gcc -o HelloWorld main.o func.o
main.o: In function ‘main’:
main.c:(.text+0xa): undefined reference to ‘helloWorlds’
collect2: error: ld returned 1 exit status
makefile:5: die Regel für Ziel „HelloWorld" scheiterte
make: *** [HelloWorld] Fehler 1
```

Der Übersetzungsvorgang wird dann wie gewünscht beim Kompilieren der Datei `main.c` abgebrochen.
Wird das vorhandene Makefile erneut angeschaut, fällt auf, dass der Programmname `HelloWorld` sowie die Funktion `gcc` mehrmals aufgeführt werden. Das heisst wollen diese Angepasst werden, so hat die Anpassung an allen Stellen zu erfolgen. Dies könnte mittels einer Definition vereinfacht werden. Es wird dazu das makefile erneut angepasst, indem die zwei Definitionen `PROGNAME` und `CC`, vor den Regeln, definiert werden. Diese können anschliessend in Regeln mit `$(DEFINITION)` verwendet werden.

```
# Makefile für Beispielprogramm

PROGNAME=HelloWorld
CC=gcc

$(PROGNAME):    main.o func.o
    $(CC) -o $(PROGNAME) main.o func.o

main.o: main.c
    $(CC) -c main.c
```

```
func.o: func.c
    $(CC) -c func.c

clean:
    rm -f *.o $(PROGNAME)
```

Sicherlich ist auch aufgefallen, dass die Regel für `main.o` und `func.o` identisch ist. Um das ganze weiter zu Vereinfachen können sogenannte « Wildcards » verwendet werden. Mit diesen können Befehle auf ganze Gruppen von Dateien angewendet werden. Es werden dazu die Operatoren `%` und `$<` verwendet.
Folgend wurden die zwei Regeln `main.o` und `func.o` mit der Regel `%.o` ersetzt. Diese Regel wird immer dann aufgerufen, wenn keine spezifische Regel für den Aufruf einer `.o`-Datei vorhanden ist. Der %-Operator ist folglich der Platzhalter für den Dateinamen. Wird die Datei gefunden, so kann sie mittels dem Operator `$<` im Befehl verwendet werden.

```
# Makefile für Beispielprogramm

PROGNAME=HelloWorld
CC=gcc

$(PROGNAME):    main.o func.o
    $(CC) -o $(PROGNAME) main.o func.o

%.o:    %.c
    $(CC) -c $<

clean:
    rm -f *.o $(PROGNAME)
```

Als letztes stört noch, dass die Objektdateien jeweils zweimal (einmal als Vorbedingungen und einmal als Objektdateien beim Link-Befehl) aufgelistet werden müssten. Aus diesem Grund wird für die Objektdateien zusätzlich noch die Definition `OBJ` angelegt, und entsprechend verwendet. Es können damit nun beliebig weitere Sourcedateien hinzugefügt werden, indem sie einfach der Definition `OBJ` angehängt werden.

```
# Makefile für Beispielprogramm

PROGNAME=HelloWorld
CC=gcc
OBJ=main.o func.o   # add more sources here

$(PROGNAME):    $(OBJ)
    $(CC) -o $(PROGNAME) $(OBJ)

%.o:    %.c
    $(CC) -c $<

clean:
    rm -f *.o $(PROGNAME)
```

Anhang C

Das Speicherlayout von μC-Programmen

Da μCs meistens sehr wenig Speicher-Ressourcen aufweisen, ist es von Vorteil, die verschiedenen Speicherteile und deren Eigenschaften zu kennen. Zudem ist es gerade bei der Inbetriebnahme eines neuen μCs wichtig, zu verstehen was mit den einzelnen Speicher-Elementen bei der Ausführung / dem Start eines μC passiert. Dieses Kapitel soll daher zu einen groben Überblick der einzelnen Speicherelementen führen. Das Speicherlayout eines Programms besteht typischerweise aus folgenden Segmenten (engl. Sections):

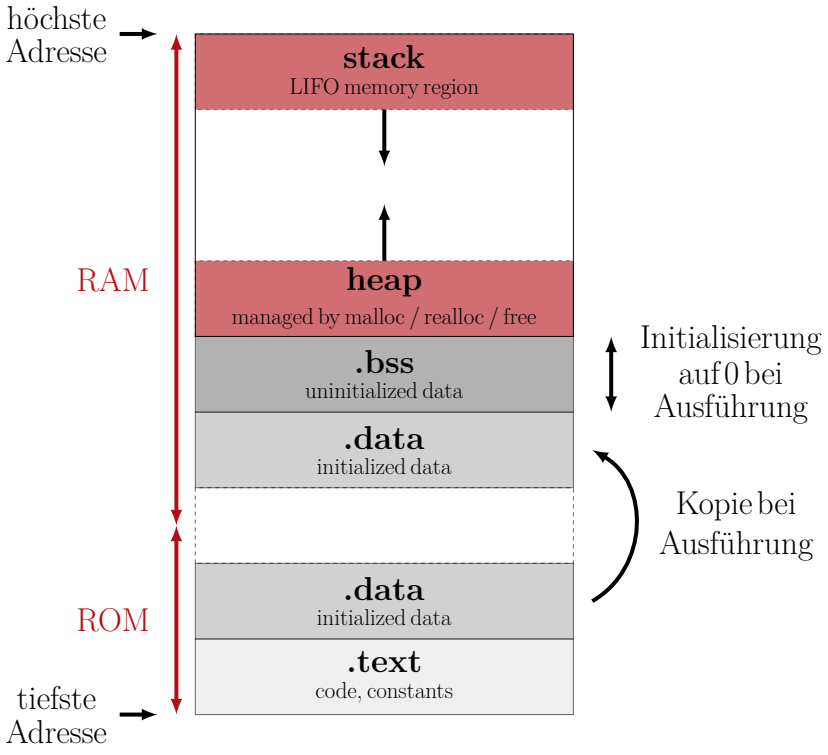


Abbildung C.1. Memory Layout eines Programms

.text (Textsegment) Das Textsegment, auch bekannt als Codesegment oder einfach als « .text », ist jenes Segment, welches ausführbare Instruktionen (Opcodes) und Konstanten [1] (.rodata[2]) beinhaltet.

[1] Lookup-Tables oder allgemein mit `const` bezeichnete Variablen sind Konstanten.
[2] rodata steht für **r**ead-**o**nly-**data**

Beispiele:

```
const unsigned short sin4LUT[8] = { 0x8000,0x98f8, 0xb0fb,0xc71c,
                                    0xda82,0xea6d, 0xf641,0xfd89}

if ( eingabe > minimum ){ return; }
```

.data (Datensegment (initialisiert)) Das .data-Segment beinhaltet globale und statische Variablen. Diese müssen beim Start bereits bekannt sein. Zudem ist ihr Speicherplatzbedarf bereits zur Übersetzungszeit bekannt. Das .data-Segment wird beim Start ins RAM geladen und steht dort zur Verfügung.

Beispiele:

```
char s[] = "hello world";
short i = 10;
```

.bss (Daten-Segment (uninizialisiert)) Das uninizialisierte Datensegment oder auch bss-Segment[3], enthält statische Variablen, die mit « Null » initialisiert werden. In Objektdateien wird üblicherweise lediglich die Grösse dieses Segments gespeichert. Beim Start wird dann ein entsprechender Speicherbereich mit Null initialisiert und als "bss" verwendet.

Beispiele:

```
short values[10];
short i;
```

Stack Der Stack wird für die Speicherung von Daten wie zum Beispiel Rücksprungadressen oder lokale Variablen mit beschränkter Gültigkeit (temporäre Variablen) verwendet. Es handelt sich dabei um ein LIFO (Last-In-First-Out) Buffer, welcher typischerweise bei der höchsten Adresse im RAM beginnt und dann nach unten wächst. Für die Koordination des Stacks ist der der Stackpointer zuständig. Dieser zeigt auf den jeweils nächsten freien Platz im Stack.

Heap Der Heap wird für die dynamische Allozierung von Speicher im RAM-Bereich benötigt. Weil dieser Bereich sehr dynamisch ist, muss er von einer Instanz verwaltet werden. Damit dies vernünftig möglich ist, ist dieser Bereich nach der `.bss`-Sektion platziert und wächst dem Stack entgegen. Für die Verwaltung stehen die Befehle `malloc`, `realloc` und `free` zur Verfügung, welche wiederum die Systemaufrufe `brk` oder `sbrk` verwenden können.

C.1 Beispiele

Das ganze Speicherlayout wird in diesem Abschnitt anhand eines kleinen PC-Programmen aufgezeigt. Dazu wird der Befehl `size` des entsprechenden Kompilers verwendet.

Hinweis: man page

Für weitere Informationen wird auf die « man page » von `size` verwiesen.

Das folgende C-Programm wird in weiteren Schritten jeweils nur wenig angepasst und danach die Grössen der Sektionen überprüft.

1. Grund-Programm

```
#include <stdio.h>

int main( void ) {
    return 0;
}
```

Nach dem Übersetzen der Source werden durch den Aufruf von `size Test`, die Grössen (in Bytes) der einzelnen Segmente `.text`, `.data` und `.bss` dezimal ausgegeben. Zusätzlich wird die Summe der Segmente dezimal (`dec`) und hexadezimal (`hex`) angegeben.

[3] Der Name bss steht für « Block Started by Symbol » was einer Pseudo-Operation eines Assemblers der 1950er Jahre entspricht.

```
$ gcc -o Test main.c
$ size Test
   text    data     bss     dec     hex filename
   1521     544       8    2073     819 Test
```

2. Es wird eine globale aber uninitialisierte Variable hinzugefügt.

```
#include <stdio.h>

long global; // Uninitialized variable stored in bss

int main( void ) {
    return 0;
}
```

⇒ die .bss-Sektion (rot) wird grösser.

```
$ gcc -o Test main.c
$ size Test
   text    data     bss     dec     hex filename
   1521     544      16    2081     821 Test
```

Hinweis: 64 Bit **System / Compiler**

Durch die Vergrösserung der Sektion um 8 Bytes beim Anlegen einer « long »-Variable kann darauf geschlossen werden, dass im vorliegenden Beispiel mit einem 64 Bit System / Compiler gearbeitet wurde.

3. Es wird zusätzlich eine statische, uninitialisierte Variable angelegt.

```
#include <stdio.h>

long global; // Uninitialized variable stored in bss

int main( void ) {
    static long i; // Uninitialized static variable stored in bss
    return 0;
}
```

⇒ Wiederum wird die .bss-Sektion (rot) grösser.

```
$ gcc -o Test main.c
$ size Test
   text    data     bss     dec     hex filename
   1521     544      24    2089     829 Test
```

4. Es wird nun die statische Variable bei der Deklaration bereits initialisiert.

```
#include <stdio.h>

long global; // Uninitialized variable stored in bss

int main( void ) {
    static long i = 100; // Initialized static variable stored in data
       section
    return 0;
}
```

⇒ Die Variable wird nun in der .data-Sektion (rot) gespeichert wodurch die .bss-Sektion (rot) wieder kleiner wird.

```
$ gcc -o Test main.c
$ size Test
   text    data     bss     dec     hex filename
   1521     552      16    2089     829 Test
```

5. Zum Schluss wird auch noch die globale Variable initialisiert.

```
#include <stdio.h>

long global = 10; // Initialized global variable stored in bss

int main( void ) {
    static long i = 100; // Initialized static variable stored in data
       section
    return 0;
}
```

⇒ Auch diese wird nun in der `.data`-Sektion (rot) gespeichert und die `.bss`-Sektion (rot) wird erneut kleiner.

```
$ gcc -o Test main.c
$ size Test
   text    data     bss     dec     hex filename
   1521     560       8    2089     829 Test
```