Kompaktkurs
C für Mikrocontroller

Inhalt

Dieses Dokument bietet eine Wiederholung der im Studium bereits erlenten Grundlagen in der Programmiersprache C dar. Ziel dieser Vorbereitung ist die Erlangung eines ausreichenden Wissenstandes zur erfolgreichen Praktikumsdurchführung. Im ersten Praktikumstermin wird der Inhalt dieser Vorbereitung in einem praktischen Programmiertest abgefragt. Das Bestehen des Tests ist Voraussetzung für die Teilname am weiteren Praktikum Busssysteme.

Was ist C?

Trotzdem es heutzutage eine Vielzahl junger Programmiersprachen gibt, ist die Sprache C in der aktuellen Variante C99 am besten für den Einsatz in kleinen, energiesparenden und kostengünstigen eingebetteten Systemen geeignet. Der Grund dafür liegt in der besonders Hardware nahen Struktur. C erlaubt direkte Speicherzugriffe ähnlich Assembler. Im Gegensatz zu Assembler ist C auf verschiedenen Prozessorarchitekturen einsetzbar. C bietet keine Unterstützung gegen unbeabsichtigtes Überschreiben fremder Speicherbereiche. Diese Eigenheit bedeutet eine enorme Fehlerquelle und verlangt große Disziplin vom Programmierer. Der Vorteil eines fehlenden Speicherschutzes liegt in einer schnelleren Programmausführung und geringerem Speicherbedarf. Sprachen mit Speicherschutz müssen vor jedem Speicherzugriff zuerst dessen Rechtmäßigkeit prüfen und dafür zusätzliche Speicherbereiche bereithalten. Die grundlegenden Programme aller UNIX Systeme und die Systemkernel vieler Betriebsysteme sind in C geschrieben. Die Syntax vieler Sprachen wie z.B. C++, Objective-C, C#, D, JAVA, PHP oder Perl lehnt sich an C an.

Basisdatentypen

→ http://de.wikipedia.org/wiki/C_(Programmiersprache)

C bietet 19 basic datatypes, welche sich in Ganzahl- und Fliesskommadatentypen aufteilen. Einfache Mikroprozzessoren und besonders energieeffiziente Systeme bieten jedoch nur Rechenhardware für Ganzzahldatentypen (Integer) an. Eine weitere Gruppe nimmt komplexe Zahlen auf.

Ganzzahlendatentypen in C

Aufgelistet ist jeweils der originäre C Datentyp sowie die Entsprechung in The ToolChain.

- **void**
  Ein leerer Datentyp, welcher keine Daten aufnehmen kann. Eine als void deklarierte Funktion kann keinen Wert zurückliefern.
- **_Bool / BOOL**
  Stellt Wahrheitswerte dar und kann nur die Werte 0 (FALSE) oder 1 (TRUE) annehmen. Oft werden Wahrheitswerte in einem 8-Bit Datentyp mit FALSE = 0 und TRUE != 0 dargestellt.
- **char / s8_t**

Der kleinste in C addressierbare Typ entspricht einem Byte oder 8 Bits. Viele C-Compiler verwenden diesen als vorzeichenbehafte Ganzzahl mit Werten von -128 bis + 127. Der char Datentyp wird hauptsächlich für Zeichenketten (Strings) verwendet.

- **unsigned char / u8_t**
  Eine vorzeichenlose Variante von char die nur positive Werte von 0 bis 255 annimmt.
- **short / s16_t**
  Ein vorzeichenbehafteter 16 Bit (2 Byte) Datentyp für Werte von -32767 bis 32768.
- **unsigned short / u16_t**
  Eine vorzeichenlose Variante von short die nur positive Werte von 0 bis 65535 annimmt
- **long / s32_t**
  Ein vorzeichenbehafteter 32 Bit (4 Byte) Datentyp für Werte von $-2^{31}+1$ bis 2^{31}.
- **unsigned long / u32_t**
  Eine vorzeichenlose Variante von long die nur positive Werte von 0 bis $2^{32}-1$ annimmt

Fließkommadatentypen in C

Die Fließkommadatentypen heißen float, double und long double. Sie unterscheiden sich durch die Speichergröße und damit die Rechengenauigkeit und auch den Rechenaufwand. Fließkommadatentypen können auf Prozessoren, welche über keine Fließkommarecheneinheit (FPU) verfügen durch Softwarebibliotheken (Soft FPU) abgebildet werden. Allerdings läuft die Berechnung dann wesentlich langsamer als bei Einsatz einer Hardware FPU. Vielfach werden dann Bibliotheken für Festkommaarithmetik eingesetzt. Festkommaarithmetik ist schneller als Fließkommaarithmetik in Software.

Abgeleitete Datentypen

Aus Basisdatentypen und allen bereits deklarierten Datentypen lassen sich weitere Datentypen ableiten. Die wesentlichen Methoden zur Ableitung sind:

- **Feldtyp** (Array)
  Ein Feld von n Elementen vom Typ T wird als T[n] geschrieben.
- **Zeigertyp** (Pointer)
  Der Zeiger (die Speicheradresse) auf einen Typ T wird als T* geschrieben.
- **Funktion** (Function)
  Die Funktion, welche für einen Typ T ein T zurückgibt wird T() geschrieben. In den Klammern können Funktionsargumente angegeben werden.
- **Strukturen** (Structures)
  Mehrere Elemente unterschiedlicher oder gleicher Typen können zu einem neuen Typ zusammengefasst werden. Das Schlüsselwort struct bündelt die einzelnen Elemente in geschweiften Klammern { }.
- **Typdefinition** (typedef)
  Abgeleitete Datentypen können mit einem neuen Namen versehen werden zur einfacheren Verwendung.

Deklarationen

Um Typen, Variablen oder Funktionen einsetzen zu können bedarf es der Deklaration. Dabei handelt es sich um eine statische Typisierung welche dem deklarierten Element einen Namen zuweist.

- **Typdeklaration**
  Mittels Schlüsselwort typedef können existierende Datentypen und Strukturen von existierenden Datentypen als neuer benannter Datentyp deklariert werden.
- **Variablendeklaration**
  Jede deklarierte Datentyp kann als Variable deklariert werden. Für die Variable wird Speicherplatz in Größe des Datentyps belegt.
- **Funktionsdeklaration**
  Damit eine Funktion aufgerufen werden kann, muss diese deklariert werden. Die Deklaration besteht aus Rückgabedatentyp (Return Type), Funktionsname und einer Liste von Datentypen die als Argumente der Funktion übergeben werden müssen.

Beispiele für Deklarationen:

```
typedef unsigned char u8_t; // vorzeichenlose 8-Bit Werte heißen auch u8_t
char C;                     // Variable C vom Typ char belegt 1 Byte

// Funktion calc mit Argumenten A,B und Rückgabetyp u8_t
u8_t calc(long A, short B);

char A[3];       // Array mit drei Elementen vom Typ Char
short B[5][6];   // Array mit 5 Elementen. Jedes Element enthält 6 Elemente
long *C[4];      // Array mit 4 Elementen vom Typ Pointer auf long
long (*D)[4];    // Zeiger auf Array mit 4 Elementen vom Typ long
void (*D)(char); // Zeiger auf Funktion mit Argument vom Typ char

// Typ AB ist Struktur mit zwei Elementen
typedef struct { char A; unsigned long B } AB;
```

Präprozessordirektiven

→ http://de.wikipedia.org/wiki/C-Präprozessor

C hat von verschiedenen Assemblersprachen eine Form der Textersetzung übernommen. Spezielle Direktiven, die allesamt mit einem Hashsymbol (#) beginnen werden in einer Präprozessorphase noch vor dem eigentlichen Übersetzungsvorgang interpretiert. Präprozessordirektiven enden am nächsten Zeilenende und nicht am nächsten Semikolon wie C Anweisungen. Neben den hier aufgeführten Direktiven existieren noch spezielle weitere die in der Referenz aufgelistet sind.

- **#include**
  Fügt den Inhalt einer anderen Quelldatei in den aktuellen Quelltext ein.
  Je nachdem, ob der Pfad der einzubindenden Datei in spitzen Klammern (< >) oder Anführungszeichen (" ") steht, wir in anderen Verzeichnissen nach der Datei gesucht.
- **#if ... #endif**
  Prüft eine Bedingung und blendet den zwischen #if und #endif liegenden Quelltext in den zu übersetzenden Quelltext ein oder aus.

- **#ifdef ... #endif**
  Prüft ob eine bestimmte Konstante per #define definiert wurde und blendet den zwischen #ifdef und #endif liegenden Quelltext in den zu übersetzenden Quelltext ein oder aus.
- **#define**
  Definiert eine Textersetzung. Wird eine Argumentliste in runden Klammern angegeben, so wird von einem Makro gesprochen. Wichtig bei Makrodefinitionen ist, dass ausreichend viele Klammern gesetzt werden, damit beim Aufruf immer das gewünschte Ergebnis berechnet wird.
- **#undef**
  Nimmt eine zuvor per #define festgelegte Definition wieder zurück.
- **Mehrzeilige Zeilen**
  Ein Backslash (****) am Zeilenende setzt die aktuelle Zeile in der nächsten fort.

Beispiele für Präprozessordirektiven:

```
#include "calc.h" // Inhalt von Datei calc.h in aktuelle Datei einbinden

// Vergleicht die Konstante SOME_CONSTANT mit 1. Bei einem erfolgreichen
// Vergleich wird die Zeile (1) in den Quelltext eingeblendet. Andernfalls
// wird die Zeile (2) eingeblendet

#if SOME_CONSTANT == 1
    u8_t Result = calc(3,4); // (1)
#else
    u8_t Result = 3;         // (2)
#endif

// Falls die Konstante SOME_NAME zuvor per #define definiert wurde, wird
// Zeile (1) in den Quelltext eingeblendet. Falls SOME_NAME nicht definiert
// wurde, wird Zeile (2) eingeblendet.

#ifdef SOME_NAME
    u8_t Result = calc(3,4); // (1)
#else
    u8_t Result = 3;         // (2)
#endif

#define PI 3.141592653589793 // eine Konstante

#undef PI                    // PI ist jetzt wieder undefined

// Makro zur Umrechnung von Inch in Millimeter
#define Inch2Millimeter(Inch) ( (Inch) * 25.4 )

// Makro über mehrere Zeilen
#define calc(A, B) \
        compute(A, B)
```

Rechenoperationen in C

C unterstützt mathematische, boolesche und logische Rechenoperationen. Passend zu den binären Operationen existieren auch unäre Operationen, welche dieselbe Variable als Quelle und Ziel haben. Die in C verfügbaren Rechenoperationen sind in Abbildung 1 dargestellt.

```
C = A + B;  // Addition         C = A | B;  // boolesch OR     C = A || B;  // logisch Oder
C = A - B;  // Subtraktion      C = A & B;  // boolesch AND    C = A && B;  // logisch Und
C = A * B;  // Multiplikation   C = A ^ B;  // boolesch XOR
C = A / B;  // Division         C = ~A;     // Komplement

A += B;     // A = A + B        A |= B;     // A = A | B
A -= B;     // A = A - B        A &= B;     // A = A & B
A *= B;     // A = A * B        A ^= B;     // A = A ^ B
A /= B;     // A = A / B
```

Abbildung 1: Binäre und unäre Rechenoperationen

Aufbau eines Programms in C

C-Programme können komplett in einer einzigen Datei zusammengefasst werden. Allerdings ist es praktischer Programmteile in mehreren Quelltexten (Dateiendung .c) und Headerdateien (Dateiendung .h) zu bündeln. Zu jeder Quelltextdatei muster.c gehört typischerweise eine passende Headerdatei muster.h. In muster.c befindet sich der Programmcode und globale Variablen. In muster.h ist die Schnittstelle zur Benutzung aus anderen Quelltexten deklariert. In muster.c wird muster.h per #include eingebunden. Die beiden Dateien synchron zu halten ist Aufgabe des Programmierers. Abbildung 2 zeigt den Aufbau eines typischen Musterprogramms.

Die Funktion main()

Der Programmfluss beginnt in der Funktion main(). Jeder Typname darf insgesamt nur einmal im gesamten Programm vorkommen. Eine Quelltextdatei, welche main() enthält kann daher nicht für andere Programe wiederverwendet werden. Es ist üblich diese Datei main.c zu nennen.

Zeiger / Pointer

Ein wesentlicher Grund für die Geschwindigkeit und den niedrigen Speicherbedarf von C-Programmen ist ihre Fähigkeit Zeiger zu verwenden. Ein Zeiger ist dabei ein Verweis auf ein anderes Element. Vereinfacht können Zeiger als die Speicheradresse des Zielobjektes angesehen werden. Die vom Compiler erzeugte Maschinensprache kennt keine Zeiger sondern nur Speicheradressen und deren Inhalte. Zeiger in C tragen jedoch außer der Zieladresse auch noch einen Datentyp. Dieser Datentyp legt fest, welchen Typ das Zielelement hat. Dieses Feature hilft viele, ansonsten schwer zu findende, Fehler zu vermeiden.

Für Zeiger existieren drei Symbole deren Bedeutung vom Programmierer jederzeit klar sein muss:

& Liefert die Referenz/ den Zeiger (Pointer) auf eine Variable.

−> Dereferenziert einen Zeiger.

***** Die Bedeutung des Sterns ist in C leicht miss zu verstehen. Es ist ein großer Unterschied ob der Stern in einer Deklaration oder bei einem Variablenzugriff steht.
Deklaration: **Typ *Variable;**
Bei einer Deklaration bedeutet jeder Stern „Zeiger auf“.
Es können beliebig viele Sterne hintereinander geschaltet werden.
Zuweisung: **Variable1 = *Variable2;** // Zeiger Variable2 dereferenzieren + auslesen
***Variable2 = Variable1;** // Zeiger Variable2 dereferenzieren + schreiben
Bei der Zuweisung bedeutet jeder Stern „Wert von“.
Auch hier können mehrere Sterne aufeinander folgen.

Zeigerzuweisungen und -vergleiche

Zuweisungen von Zeigern sind oft eine Quelle von Compilerwarnungen und schwierig zu findender Fehler. Dabei ist das Grundkonzept sehr einfach:

> Auf beiden Seiten eines Gleichheitszeichens muss derselbe Datentyp stehen!

Abbildung 3 zeigt ein Beispiel für die Verwendung von Zeigern in C. Im blau hinterlegten Kasten befindet sich der Quelltext. Die grün hinterlegten Kästen zeigen die Situation im Arbeitsspeicher. Die 3 PadBytes ab Adresse 0x20000001 werden vom Compiler eingefügt, damit der nächste Zeiger an einer durch vier teilbaren Adresse beginnt.

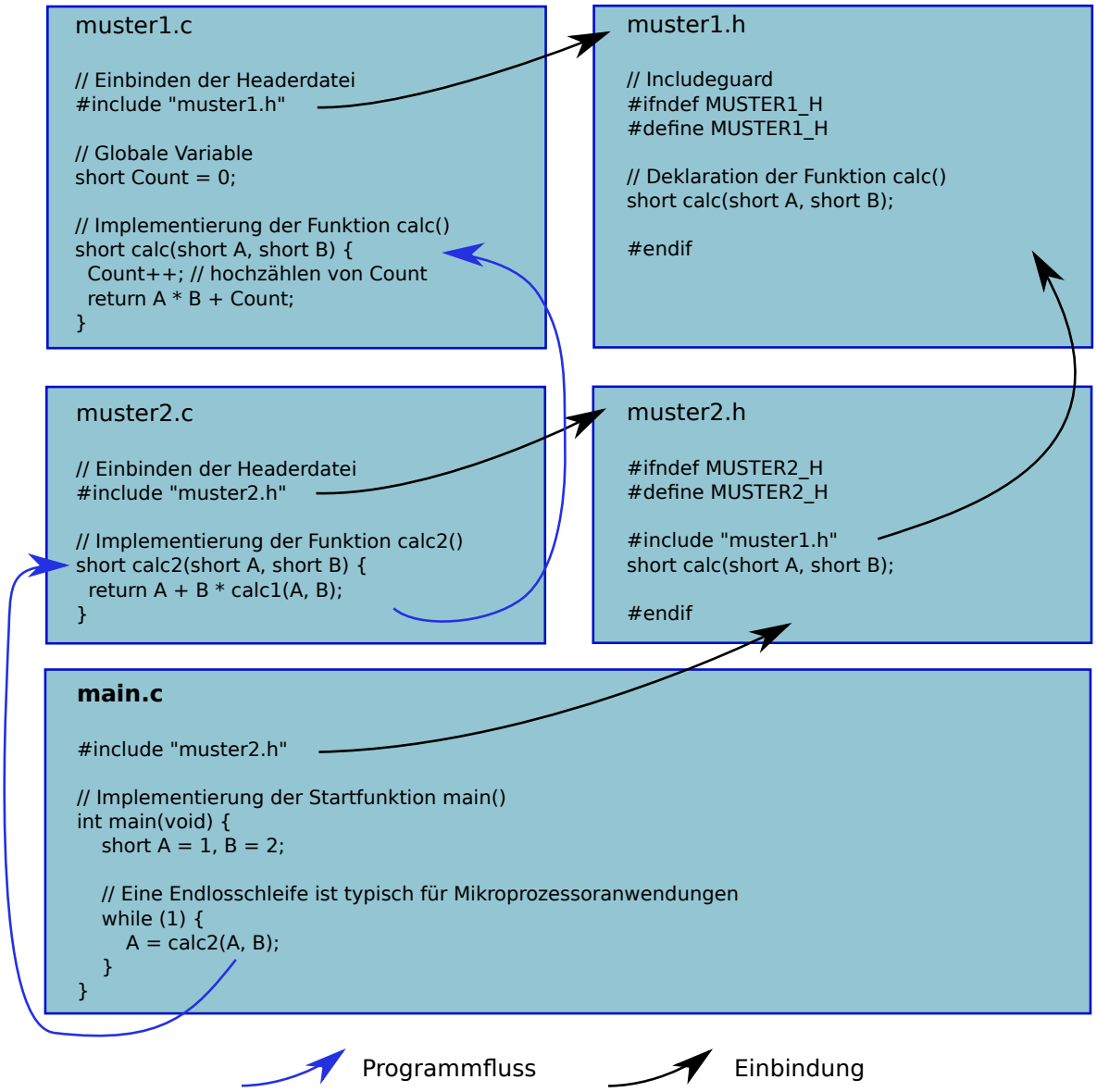

Abbildung 2: Musterprogramm in C

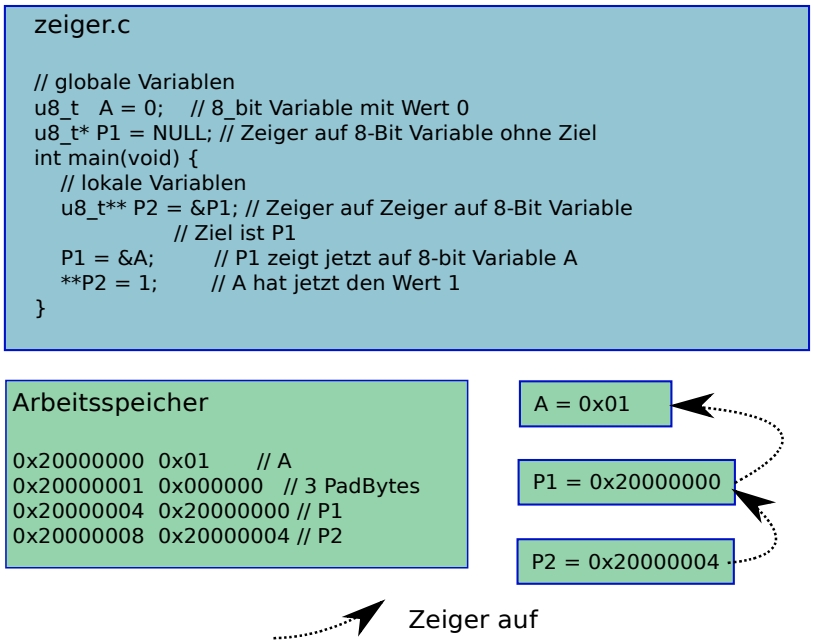

Abbildung 3: Zeiger in C

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:30, 27:26, 23:22, 19:18, 15:14, 11:10, 7:6, 3:2 **CNFy[1:0]:** Port x configuration bits (y= 0 .. 7)
These bits are written by software to configure the corresponding I/O port.
Refer to *Table 20: Port bit configuration table on page 156*.
In input mode (MODE[1:0]=00):
00: Analog mode
01: Floating input (reset state)
10: Input with pull-up / pull-down
11: Reserved
In output mode (MODE[1:0] > 00):
00: General purpose output push-pull
01: General purpose output Open-drain
10: Alternate function output Push-pull
11: Alternate function output Open-drain

Bits 29:28, 25:24, 21:20, 17:16, 13:12, 9:8, 5:4, 1:0 **MODEy[1:0]:** Port x mode bits (y= 0 .. 7)
These bits are written by software to configure the corresponding I/O port.
Refer to *Table 20: Port bit configuration table on page 156*.
00: Input mode (reset state)
01: Output mode, max speed 10 MHz.
10: Output mode, max speed 2 MHz.
11: Output mode, max speed 50 MHz.

Abbildung 4: Registerbeschreibung aus STM32F107 Datenblatt RM0008

```
// Strukturdeklaration
typedef struct {
  unsigned MODE0    : 2; // Bits  0..1
  unsigned CNF0     : 2; // Bits  2..3
  unsigned MODE1    : 2; // Bits  4..5
  unsigned CNF1     : 2; // Bits  6..7
  unsigned MODE2    : 2; // Bits  8..9
  unsigned CNF2     : 2; // Bits 10..11
  unsigned MODE3    : 2; // Bits 12..13
  unsigned CNF3     : 2; // Bits 14..15
  unsigned MODE4    : 2; // Bits 16..17
  unsigned CNF4     : 2; // Bits 18..19
  unsigned MODE5    : 2; // Bits 20..21
  unsigned CNF5     : 2; // Bits 22..23
  unsigned MODE6    : 2; // Bits 24..25
  unsigned CNF6     : 2; // Bits 26..27
  unsigned MODE7    : 2; // Bits 28..29
  unsigned CNF7     : 2; // Bits 30..31
} __attribute__((__packed__)) GPIOx_CRL_t;

int main(void) {
  // Zeiger auf Register gemäß Datenblatt
  GPIOx_CRL_t* CRL_Pointer;
  CRL_Pointer = (GPIOx_CRL_t*) 0x40010814;

  // Registerinhalt in lokale Variable
  GPIOx_CRL_t  CRL_Value = *CRL_Pointer;

  // Felder setzen (Werte gemäß Datenblatt)
  CRL.MODE0 = 2;
  CRL.CNF0  = 1;

  // Neuen Wert in Register schreiben
  *CRL_Pointer = CRL_Value;

  // Einzelnes Feld von Register auslesen
  u8_t Mode1 = CRL_Pointer->MODE1;
}
```

Abbildung 5: Registerzugriff per Strukturzeiger

```
int main(void) {
  // Zeiger auf Register gemäß Datenblatt
  u32_t* CRL_Pointer = (u32_t*) 0x40010814;

  // Registerinhalt in lokale Variable
  u32_t  CRL_Value = *CRL_Pointer;

  // Neue Feldwerte gemäß Datenblatt
  u8_t MODE0 = 2;
  u8_t CNF0  = 1;

  // Zu setzende Felder leeren
  CRL_Value &= 0xfffffff0;

  // Neue Feldwerte setzen
  CRL_Value |= CNF0 << 2 | MODE0;

  // Neuen Wert in Register schreiben
  *CRL_Pointer = CRL_Value;

  // Einzelnes Feld von Register auslesen
  u8_t Mode1 = ((*CRL_Pointer) >> 4) & 0x3;
}
```

Abbildung 6: Registerzugriff per Shifting

Memory Mapped IO

Die Fähigkeit in C direkt auf Speicherbereiche zugreifen zu können ist bedeutend für den Einsatz in Mikrocontrollern. Die verwendete Technik heißt speicherabgebildete Ein-/Ausgabe (memory mapped IO). Dabei steuern einzelne Bits von Speicheradressen direkt Hardware im Mikrocontroller. In einem einfachen Fall kann z.B. eine 1 in einem Bit bedeuten, dass ein Anschlusspin des Mikrocontrollers seine Spannung von 0V auf 3,3V ändert. Die Speicheradressen mit Steuerfunktion werden Register genannt. Die Register eines Mikrocontrollers sind in seinem Datenblatt genau beschrieben.

Ein Beispiel ist das Reference Manual RM0008 für die Mikrocontroller der STM32f1xx Reihe. → RM0008 ab Seite 166 werden die Register für generelle Ein-/Ausgabe (GPIO) beschrieben.

Besonders praktisch für das Setzen einzelner Bitfelder in Registern sind entsprechende Strukturzeiger.

Abbildung 4 Zeigt einen Auszug aus der Registerbeschreibung einer GPIO Bank im STM32F1xx Mikrocontroller. Die Bitfelder sind jeweils 2 Bits groß und können vier verschiedene Werte annehmen. Jeweils ein CNFx und MODEx Feld kontrollieren einen GPIO Pin. Der zugehörige

Quelltext unter Verwendung von Strukturen ist in Abbildung 5 zu sehen.

Oftmals werden Register jedoch unter Verwendung von Shift-Operationen und booleschen UND-/ ODER-Verknüpfungen gesetzt. Dieser Zugriff kann schneller sein, wenn mehrere Felder aufeinmal gesetzt/ ausgelesen werden sollen. Allerdings ist dieser Zugriff mühsam und fehleranfällig. Der Compiler kann den Zugriff auf die richtigen Bits hier nicht überprüfen. Ein Beispiel ist in Abbildung 6 zu sehen.

Textausgabe per printf(), snprintf()

Die Funktion printf() wird auf POSIX Systemen in einer Bibliothek namens Standard Ein-/ Ausgabe (stdio) bereitgestellt. Printf() ist in C die zentrale Funktion zur Ausgabe von Text gemischt mit Zahlenwerten. Das Besondere ist die Kombination von beliebig vielen Wertargumenten und einer Formatzeichenkette. Für jedes Wertargument enthält diese eine Formatdirektive:

Gängige Formatdirektiven:

%i / %d	Dezimalzahl (Integer)
%f	Fließkommazahl
%s	Zeichenkette (String)
%c	Einzelnes Zeichen (Character)
%x	Hexadezimalzahl

Nach dem Prozentzeichen (%) kann eine Zahl die feste Länge eines Feldes angeben. Eine vorangestellte Null füllt kleine Zahlen am Anfang mit Nullen auf.

Eine ausführliche Beschreibung von printf() ist in Abschnitt 3 der Man Pages zu finden.

> man 3 printf

Für Mikrocontroller bedeutsamer ist die snprintf() Funktion welche in eine Speicherpuffer schreibt. Der Puffer kann per serieller Schnittstelle (UART, SPI, I²C) oder grafisch ausgegeben werden.

Beispiele:

```
#include <stdio.h>
// printf ist in der Systembibliothek stdio implementiert

printf("Hallo Welt\n");
// konstanter Text mit Zeilenumbruch (Newline)
→ "Hallo Welt\n"

printf("Zähler=%i", Count++);
// Text + Integerwert ausgeben (anschließend erhöhen)
→ "Zähler=3"

#define BUFFER_SIZE 100
u8_t Buffer[BUFFER_SIZE];
u8_t A = 1; u8_t B = 10; float C = 3.141;
snprintf(Buffer, BUFFER_SIZE, "A=%i, B=0x%04x, C=%f", A, B, C);
// Integer, Hexadezimal, Fließkommawert
→ Buffer[]="A=1, B=0x000a, C=3.141"

// Formatstring mit Stringdirektive
const char* Format = "Hallo %s!";

const char* Welt = "Welt";  // Zeichenkette (String)
snprintf(Buffer, BUFFER_SIZE, Format, Welt);
→ Buffer[]="Hallo Welt!"
```

C-Compiler GCC

Der wohl am meisten eingesetzte C-Compiler ist die GNU Compiler Collection (GCC). Der GCC ist für praktisch jede Prozessorarchitektur portiert worden. Die große Verbreitung bietet den Vorteil dieselbe Sprachsyntax auf verschiedenen Architekturen zu nutzen.

Unter Unix artigen Betriebsystemen wie z.B. Linux stehen Man Pages zur Dokumentation bereit. Hierzu wird in einer Shellkonsole einfach folgender Befehl eingegeben (> ist der Eingabeprompt):

> **man gcc**

Das komplette Manual umfasst über 12000 Zeilen. Der Löwenteil davon beschreibt Kommandozeilenargumente für verschiedene Zielarchitekturen. Kommandozeilenargumente geben dem Compiler zusätzliche Informationen wie z.B. die Namen von Ein- und Ausgabedateien. Jedes Argument beginnt mit einem Minuszeichen unmittelbar gefolgt vom Argumentnamen. Bei Argumenten mit Wert folgt dem Argumentnamen direkt der Wert.

Aufrufsyntax (Ausschnitt):

gcc [**-c**] [**-I**<INCLUDE_VERZEICHNIS>**] [-L**<QUELLTEXT_VERZEICHNIS>]
[-D<NAME>[=<WERT>]] [**-o** <AUSGABEDATEI>] <EINGABEDATEI>

Metazeichen: <...> = zu ersetzender Teil, […] = optionaler Teil, ***fett*** = einzugebender Text

Die meisten Argumente für gcc sind optional. Das einzige nicht optionale Argument ist die Eingabedatei <EINGABEDATEI>.

Die wichtigsten Kommandozeilenargumente:

- **-c** Kompiliert die Eingabedatei in eine Objektdatei.
- **-I** Fügt ein Verzeichnis in dem nach Headerdateien gesucht werden soll, hinzu.
- **-l** Fügt eine Binärbibliothek zum Linken hinzu.
- **-D** Setzt eine Konstante als wenn diese im Quelltext mit #define gesetzt wurde
  Fehlt =defn so wird die Konstante als 1 definiert.
- **-o** Die Ausgabedatei für den aktuellen Vorgang.

Der Übersetzungsprozess

Der GCC übersetzt C-Programme im Wesentlichen in zwei Schritten

1. **gcc -c <DATEI.c>**
   Eine einzelne .c Quelldatei wird in eine .o Objektdatei übersetzt (kompiliert).
   Dabei sind nur die Quelldatei und alle per #include eingebundenen Headerdateien beteiligt.
   Funktionsaufrufe in andere Quelltextdatein tauchen in der Objektdatei nur symbolisch auf, da die Speicheradressen der anderen Funktionen noch nicht festliegen.
2. **gcc <DATEI1.o> [<DATEI2.o> […]] -o <DATEI>**
   Alle .o Objektdateien werden in eine Ausgabedatei zusammen gebunden (gelinkt).
   Der Linker sucht in allen angegebenen Objektdateien nach allen bisher nicht aufgelösten Symbolen wie globale Variablen und Funktionen. Wenn in dieser Phase ein Symbol nicht gefunden werden kann, dann fehlt vermutlich die entsprechende Objektdatei.

Ein großer Vorteil dieses zweistufigen Übersetzungsprozesses ist, dass alle C-Dateien parallel

übersetzt werden können. Dadurch kann die Übersetzungszeit auf modernen Mehrkernprozessoren deutlich reduziert werden. Wenn Fehler auftreten ist es wichtig zu erkennen, in welcher der zwei Phasen der Fehler auftritt.

Steuerung größerer Übersetzungsprozesse mittels makefile

Wenn C-Programme altern dann wachsen sie, ähnlich Menschen, nicht nur in die Länge sondern auch in die Breite. Dass heißt, es werden Funktionen hinzugefügt die nicht immer benötigt werden. Ein Grafiktreiber zum Beispiel ist überflüssig wenn das aktuelle Board über kein Display verfügt.

Im Bereich der Mikrocontroller gibt es wesentlich mehr verschiedene Boards als bei Desktoprechnern. Gleichzeitig sind die Programmspeicher so klein, dass möglichst nur unbedingt gebrauchte Funktionen in die Firmware (die übersetzte Binärdatei) übernommen werden sollen.

Die Lösung: Zu erzeugende Objektdateien (.o) werden in einer Steuerdatei für das Programm make eingetragen. Die Steuerdatei namens makefile enthält verschiedene Regeln. Die folgene Liste zeigt die wichtigsten Direktiven und Variablennamen welche in The ToolChain verwendet werden:

- Objektdatei soll erzeugt werden (kompiliert aus .c oder .s Datei)
  Direktive: **MAIN_OBJS** += <DATEINAME>**.o**
- In welchen Ordnern liegen Headerdateien (.h)
  Direktive: **INCLUDE_DIRS** += **-I**<VERZEICHNISPFAD>
- In welchen Ordnern liegen C-Quelltexte (.c)
  Direktive: **vpath %.c** <VERZEICHNISPFAD>
- In welchen Ordnern liegen Assemblerquellen (.s)
  Direktive: **vpath %.s** <VERZEICHNISPFAD>
- Definition einer Konstanten (in allen Quelltexten sichtbar)
  Direktive: **COMPILE_OPTS** += **-D**<NAME>=<WERT>
- Definition eines Übersetzungszieles
  Direktive: <TARGET>**:** <QUELLE1> [<QUELLE2>]

Das wesentliche beim Lesen eines makefiles ist, dass ein Übersetzungsziel immer das Ende einer Operation darstellt. Um ein Ziel zu erstellen, erstellt make zunächst alle dafür benötigten Teile. Das können auch wieder Ziele sein. Die Erstellung beginnt bei den Aufgaben die keine Unteraufgaben mehr enthalten. Um z.B. eine Ausgabedatei main.bin zu erzeugen muss zunächst main.elf erzeugt werden, dafür müssen alle Dateien in MAIN_OBJS erstellt werden usw.

The ToolChain verwendet zwei makefiles:

1. <PROJEKTORDNER>**/makefile**
   Stellt das Grundgerüst für das aktuelle Projekt und bindet das zweite makefile ein.
2. <PROJEKTORDNER>**/extensions.active/makefile**
   Ist eine Zusammenfassung aller im selben Ordner liegenden makefile Fragmente. Die Fragmente werden in der Aktivierungsstufe von activate_project.sh durch Aufruf von Aktivierungsskripten erstellt. Nach erfolgreicher Kompilierung können in diesem makefile alle Einstellungen des aktuellen Übersetzungsprozesses nachgelesen werden:
   - Der zu verwendende **Compiler**

- Das verwendete Board mit **Mikrocontroller**, **Schnittstellen** und **GPIOs**
- Größe und Lage von Arbeitsspeicher (**RAM**) und Programmspeicher (**FLASH**)
- Der zu verwendende **Programmieradapter** (350_*)
- Aktivierte **Treiber** (500_ttc_* sind hardwareunabhängig und 450_* sind hardwareabhängig)
- Aktivierte **Beispielprogramme** (600_example_*)
- Die projektbezogene Steuerdatei **extensions.local/makefile.700_extra_settings**

Übungsaufgaben C für Mikrocontroller

Zur Durchführung der Übungsaufgaben wird ein C-Compiler unter einem beliebigen Betriebsystem benötigt. Es wird empfohlen ein aktuelles Kubuntu von http://www.kubuntu.org/ herunterzuladen und entweder im Dualbootmodus oder unter einer virtuellen Maschine zu installieren. Für beide Installationsarten existieren viele Anleitungen im Netz.

Es sei hier insbesondere auf die Präsentation Softwareentwicklung_unter_Linux.pdf verwiesen. Dort wird die Installation des Linux Betriebsystems und der benötigten Anwendungen beschrieben

Unter Linux muss dann noch der gcc installiert werden. Das funktioniert unter Debian basierten Distributionen wie z.B. Ubuntu, Xubuntu oder Kubuntu in einer Shellkonsole mit folgender Eingabe:

> sudo apt-get install gcc

Hinweis: Das abgefragte Benutzerkennwort wird nicht auf dem Bildschirm ausgegeben (Linuxer sind überzeugt, daß das sicherher ist).

Die Quelltexte können mit einem beliebigen Texteditor (kate, gedit, …) erstellt werden.

Aufgabe 1a

Erstellen Sie ein Programm in der Quelldatei a1.c welches

1. eine Konstante PI definiert
2. eine int main() Funktion enthält die
   PI * 2 per printf() Funktion ausgibt

Übersetzen Sie per gcc a1.c in a1.o und linken Sie diese anschließend in die Ausgabedatei aufgabe1. Das erzeugte ausführbare Programm kann dann per ./aufgabe1 ausgeführt werden.

Aufgabe 1b

Kommentieren Sie in a1.c die Zeile mit der Konstanten PI aus (// an den Zeilenanfang).

Jetzt übergeben Sie die Konstante per GCC Argument -D beim Kompilieren und linken Sie in die Ausgabedatei aufgabe2.

Kombinieren Sie Kompilieren Linken und Auführen in eine einzelne Kommandozeile indem Sie einzelne Befehle mit Semikolon trennen.

Probieren Sie verschiedene Werte für PI aus.

Aufgabe 2

Erstellen Sie ein Programm in der Quelldatei a2.c welches

1. Eine Struktur S_t definiert die zwei vorzeichenlose 8-Bit Werte mit Namen A und B enthält
2. ein globales Array A mit 100 Einträgen vom Typ S_t aufnimmt.
3. eine int main() Funktion enthält die
   Für alle Einträge A[i], i = 0..99 jeweils A mit i und B mit 99-i beschreibt
4. Einen Pointer P1 vom Typ Pointer auf unsigned long anlegt
5. P1 auf die Adresse von A[0] setzt
6. Pointer P2 vom Typ Pointer auf Pointer auf unsigned long anlegt
7. P2 auf Adresse von P1 setzt
8. Per printf() ausgibt:
   P1, *P1, P1->A, *P2, (**P2).B, (*P2)->B, P2[0]->B
9. P1 um 12 erhöht
10. Per printf() ausgibt:
    P1, *P1, *P2, **P2

Übersetzen Sie a2.c in a2.o und linken Sie diese anschließend in die Ausgabedatei aufgabe2.

Welche Werte werden ausgegeben?
Von welchem Typ ist *P2?

Aufgabe 3

Erstellen Sie ein Programm mit drei Quelldateien a3.c, b3.c, c3.c und zugehörigen Headerdateien a3.h, b3.h, c3.h.

a3.c

Bindet a3.h ein.

Funktion int main()
- berechnet Result = c_calc(b_calc())
- gibt Result per printf() aus

a3.h

Bindet b3.h, c3.h ein.

b3.c

Bindet b3.h ein.

Funktion int b_calc() gibt 3 zurück.

b3.h

Deklariert Funktion int b_calc().

c3.c

Bindet c3.h ein.

Funktion int c_calc(int A) gibt A mal 5 zurück.

c3.h

Deklariert Funktion int c_calc(int A).

Übersetzen Sie a3.c, b3.c, c3.c in a3.o, b3.o, c3.o und linken Sie diese anschließend in die Ausgabedatei aufgabe3.