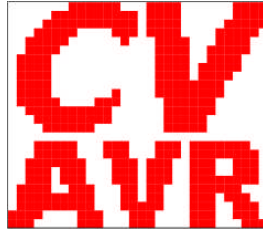


AVR-8-bit-Mikrocontroller
Gruppe 500 - CodeVisionAVR C-Compiler
Teil 505 - Modularer Aufbau der AVR-C-Projekte



Teil 501 - Einführung

- 1 Eine Einführung in **C**
 - 1.1 Warum **C** ?
 - 1.2 Wie entstand **C** ?
 - 1.3 Der AVR-Mikrocontroller in einem eingebetteten System
 - 1.4 Werkzeuge (siehe Gruppe 200)

Teil 502 - Aufbau eines C-Projektes

- 2 Was ist ein **C**-Projekt ?
 - 2.1 Erzeugen eines **C**-Projektes
 - 2.1.1 Ein neues Projekt beginnen
 - 2.1.2 Ein **C**-Projekt generieren
 - 2.2 Dateistruktur eines **C**-Projektes
 - 2.3 Einbindung von AVR Studio in den CVAVR
 - 2.4 AVR Studio Debugger
 - 2.5 **C**-Compiler-Optionen

Teil 503 - Preprozessor

- 3 Preprozessor-Anweisungen
 - 3.1 Struktur der **C**-Quell-Programme
 - 3.2 **#include**-Anweisung
 - 3.3 **#define**-Anweisung (Makro)
 - 3.3.1 Makros ohne Parameter
 - 3.3.2 Makros mit Parametern
 - 3.4 **#undef**-Anweisung
 - 3.5 **#if**-, **#ifdef**-, **#ifndef**-, **#else**- und **#endif**-Anweisungen
 - 3.6 Andere Preprozessor-Anweisungen

Teil 504 - Syntax der C-Programmiersprache

- 4 Die Syntax der **C**-Programmiersprache
 - 4.1 **C**-Quell-Programme
 - 4.1.1 Kommentare
 - 4.1.2 Deklarationen (Vereinbarungen)
 - 4.1.3 Die Funktion **main**
 - 4.1.4 Schlüsselwörter (Keywords) des CodeVisionAVR **C**-Compilers
 - 4.2 Konstanten und Variablen
 - 4.2.1 Zahlensysteme
 - 4.2.2 Datentypen
 - 4.2.3 Konstanten
 - 4.2.4 Variablen
 - 4.3 Operatoren
 - 4.3.1 Arithmetische Operatoren
 - 4.3.2 Relationale Operatoren
 - 4.3.3 Logische und bitweise wirkende Operatoren
 - 4.3.4 Andere Operatoren und Shortcuts
 - 4.4 Komplexe Objekte in **C**
 - 4.4.1 Funktionen
 - 4.4.2 Funktions-Prototypen
 - 4.4.3 Pointers und Arrays
 - 4.4.3.1 Pointers

AVR-8-bit-Mikrocontroller

Gruppe 500 - CodeVisionAVR C-Compiler

Teil 505 - Modularer Aufbau der AVR-C-Projekte

- 4.4.3.1.1 Pointers in Verbindung mit `flash` und `eeprom`
- 4.4.3.1.2 Pointers in Verbindung mit `typedef`
- 4.4.3.2 Arrays
 - 4.4.3.2.1 Ein-dimensionale Arrays
 - 4.4.3.2.2 Zwei-dimensionale Arrays
 - 4.4.3.2.3 Drei-dimensionale Arrays
- 4.4.3.3 Benutzen der Array-Namen als Pointers
- 4.4.3.4 Arrays von Pointers
- 4.4.3.5 Pointers auf Funktionen (Funktionszeiger)
- 4.4.3.6 Funktionen in Verbindung mit `typedef`
- 4.4.4 Strukturen und Unionen
 - 4.4.4.1 Strukturen
 - 4.4.4.2 Unionen
- 4.4.5 Komplexe Typen (eine Zusammenfassung)
- 4.5 Steuerung des Programmablaufs
 - 4.5.1 Anweisungsblöcke { . . . }
 - 4.5.2 Die Anweisung `if`
 - 4.5.3 Die Anweisung `if` in Verbindung mit `else`
 - 4.5.4 Die Fallunterscheidung `switch`
 - 4.5.5 Die Schleife `for`
 - 4.5.6 Die Schleife `while`
 - 4.5.7 Die Schleife `do` in Verbindung mit `while`
- 4.6 Arbeiten mit den Ein-/Ausgabe-Ports

Teil 505 - Modularer Aufbau der AVR-C-Projekte

5 Modularer Aufbau

- 5.1 Das Konzept
- 5.2 Nomenklatur
- 5.3 Die speziellen Header-Dateien
 - 5.3.1 Die spezielle Header-Datei `typedefs.h` (Typ- und Bit-Definitionen)
 - 5.3.2 Die spezielle Header-Datei `iomx.h` (Definitionen aller Register-Bits)
 - 5.3.3 Die spezielle Header-Datei `macros.h` (Definitionen von Makros)
 - 5.3.4 Die spezielle Header-Datei `switches.h` (Definitionen von Schaltern)
- 5.4 Die AVR-C-Module
- 5.5 Anwendung der angepassten AVR-C-Module
 - 5.5.1 Das AVR-C-Modul `application` (Anwendungs-Steuerung)
 - 5.5.2 Das AVR-C-Modul `lcd_2wire` (Ausgabe auf LCD-20x4)
 - 5.5.3 Das AVR-C-Modul `num_conversion` (Typ-Konvertierung nach ASCII)
 - 5.5.4 Das AVR-C-Modul `adc_ref_1_1` (ADC mit interner Referenz 1,1 V)
 - 5.5.5 Das AVR-C-Modul `rc5_decoder` (RC5-IR-Fernsteuerung-Dekoder)
 - 5.5.6 Das AVR-C-Modul `rc5_encoder` (RC5-IR-Fernsteuerung-Encoder)
 - 5.5.7 Das AVR-C-Modul `usart` (USART-Steuerung)
 - 5.5.8 Das AVR-C-Modul `twi_master` (I2C- bzw. TWI-Steuerung)
 - 5.5.9 Das AVR-C-Modul `timer0_pwm` (TIMER0-Steuerung)

Teil 506 - Anhang

6 Anhang

- 6.1 Begriffe und Definitionen
- 6.2 Bibliothek

AVR-8-bit-Mikrocontroller
Gruppe 500 - CodeVisionAVR C-Compiler
Teil 505 - Modularer Aufbau der AVR-C-Projekte

Vorbemerkung

Nichts ist vollkommen - und nichts ist endgültig! So auch nicht dieses Tutorial! Deshalb bitte immer erst nach dem neuesten Datum schauen. Vielleicht gibt es wieder etwas Neues oder eine Fehlerbereinigung oder eine etwas bessere Erklärung. Wer Fehler findet oder Verbesserungen vorzuschlagen hat, bitte melden (info@alenck.de).

Immer nach dem Motto: Das Bessere ist Feind des Guten und nichts ist so gut, dass es nicht noch verbessert werden könnte.

Bild-, Beispiel-, Form- und Tabellen-Nummern sind nach folgendem Schema aufgebaut, damit bei Einfügungen/Löschungen nicht alle Nummern wieder geändert werden müssen (hier bunt dargestellt):

Darstellungsart	Abschnitt-LfdNummer: Beschreibung	allgemeines Schema
•	Bild 5.1.4-02: Daten-Adress-Raum	Benummerung eines Bildes
•	Beispiel 5.1.4-03: EEPROM-Speicherung	Benummerung eines Beispiels
•	Form 5.1.3-01: Die main-Funktion	Benummerung einer Formdarstellung
•	Tabelle 5.1.4-01: Schlüsselwörter vom CVAVR	Benummerung einer Tabelle

Gravierende Änderungen gegenüber der Vorversion

1.

Völlig neue Strukturierung in **Gruppen** und **Teile**, um das Tutorial umfassend ordnen zu können. Die **Abschnitte** in den **Teilen** sind weitgehend erhalten geblieben.

Gruppenbezeichnung	Kurzbezeichnung
Gruppe 100: Technologie der AVR-8-Bit-Mikrocontroller	Technologie
Gruppe 200: Einsetzen von AVR-Tools	Tools
Gruppe 300: Arbeiten mit AVR-Assembler 3xx_Programm_yyyyy	ASM-Programmierung ASM-Programm-Beispiel
Gruppe 400: AVR-ASM-Projekte 4xx_Projekt_yyyyy	ASM-Projekte ASM-Projekt-Bezeichnung
Gruppe 500: CodeVisionAVR C-Compiler 5xx_Programm_yyyyy	C-Programmierung C-Programm-Beispiel
Gruppe 600: AVR-C-Projekte 6xx_Projekt_yyyyy	C-Projekte C-Projekt-Bezeichnung

xx steht für die laufende Nummer innerhalb des **Teils**, in dem das Programm bzw. das Projekt erscheint und **yyyyy** steht für die Programm- bzw. Projekt-Kurz-Bezeichnung.

2.

Notwendige Änderungen auf Grund Neuinstallation von **Windows 7**.

3.

Windows 7 machte eine Installation von **CodeVisionAVR V2.60** als Vollversion notwendig. Daraus leiten sich auch viele Änderungen im Detail für die C-Programmierung (**Gruppe 500**) ab.

4.

Neu-Installation von **AVR Studio Vers. 4.19** unter **Windows 7**

5.

Zur Demonstration des Tools **AVR Studio** ist in **Gruppe 200** eine Trennung in **Teil 205 - Assembler und AVR Studio** und **Teil 206 - C-Compiler und AVR Studio** vorgenommen worden.

5.

ASM- und **C-Projekte** werden jeweils in eigenen Gruppen gesammelt (**Gruppe 400** für Assembler- und **Gruppe 600** für C-Projekte).

5 Modularer Aufbau der AVR-C-Projekte

AVR-C-Projekte sind **C-Projekte**, wie sie im **Teil 206 - C-Compiler und AVR Studio** vorgestellt wurden mit dem Unterschied, dass ihre äußere Struktur einem speziellen modularen Aufbau unterworfen wird. Die **AVR-C-Projekte** werden in der **Gruppe 600** in der Form **6xx_Projekt_yyyyy** gesammelt. **xx** steht für die laufende Nummer innerhalb des **Teils**, in dem das Projekt erscheint und **yyyyy** steht für die Projekt-Kurz-Bezeichnung.

Modulares Denken hat (wie alles im Leben) Vor- und Nachteile!

Ein deutlicher Nachteil ist, dass ein "Newcomer" bei der modularen Vorgehensweise anfänglich erheblichen Verstandes-Schwierigkeiten gegenüber steht. Selbst kleine Programme sehen erschreckend kompliziert aus. Besonders Anfänger werden dadurch immer wieder entmutigt und denken: "Das verstehe ich nie!"

Wenn man anfangs die Scheu überwindet und die passende Einstellung zum modularen Denken findet, dann kann aus dem Nachteil sehr schnell ein Vorteil werden. Wer diese Module zunächst einfach als "Black Boxes" betrachtet, der hat es leichter.

Ein Tipp: Zunächst die Aufgaben aus den verschiedenen Modulen einfach nur nutzen, ihre Wirkung kennen und anwenden ohne genau zu wissen, wie sie funktionieren. Zum Autofahren muss man auch nicht wissen, wie ein 4-Takt-Ottomotor oder ein Dieselmotor funktioniert.

5.1 Das Konzept

Der Hauptgrund für den Einsatz der Modul-Technik geht einher mit der strukturierten Programmierung und dem Einsatz von Funktionen aus sog. Bibliotheken. Im Laufe seiner Befassung mit der Programmierung entwickelt man **allgemein anwendbare** Funktionen, die nicht nur für eine bestimmte Anwendung nützlich sind, sondern sich auch in anderen Projekten einsetzen lassen. So kann man sich einen Bereich für Universalfunktionen anlegen, in dem alles Wiederverwertbare abgelegt wird. Das trifft insbesondere auf Funktionen zu, die den Gebrauch von Hardware-Ressourcen des Mikrocontrollers betreffen. Diese Funktionen bringt man am besten in einer Datei unter, die z.B. einer bestimmten Steuerung dienen. In einem Projekt, das dann aus mehreren Quell-Dateien "gespeist" wird, bezeichnet man die neben dem Quellprogramm **main.c** auftretenden Quell-Code-Dateien als **Module**.

Beispiel 5.1-01: Demonstration einer einfachen Modularität

Ohne auf das Coding einzugehen, soll die nachfolgende Struktur für die Berechnung einer Quadratwurzel die Modularität demonstrieren:

```
// Das Programm liest eine Zahl ein und gibt als Ergebnis eine Zahl,  
// die der Quadratwurzel entspricht, zurück.  
  
#include <mega88.h>      // Header-Datei fuer den Chip ATmega88  
#include <math.h>       // Header-Datei mit mathematischen Funktionen  
  
void main(void)         // Hauptprogramm  
{  
    float x              // Deklaration der positiven Variablen x  
  
    while(1)            // Endlos-Schleife  
    {  
        /* hier steht die Eingabe-Funktion zur Aufforderung zur  
        Eingabe eines Floating-Wertes von dem die Quadratwurzel  
        berechnet werden soll */  
  
        /* es folgt die Ausgabe-Funktion zur Anzeige des eben  
        eingegebenen Zahlenwertes */  
  
        float sqrt(float x); // Aufruf der Quadratwurzel-Funktion  
  
        /* Es folgt die Ausgabe-Funktion mit dem Ergebnis in einer  
        besonderen Zeile mit entsprechendem Text */  
    }  
}
```

AVR-8-bit-Mikrocontroller

Gruppe 500 - CodeVisionAVR C-Compiler

Teil 505 - Modularer Aufbau der AVR-C-Projekte

```
/* Warten auf ein Tastendruck (Tastendruckfunktion) zur Eingabe
   Eingabe der naechsten Berechnung einer Quadratwurzel */
}
}
```

Das **Modul**, das die mathematischen Funktionen für die Quadratwurzel enthält, ist die **globale Header-Datei** `math.h`. Das Modul wird aufgerufen (besser: eingebunden) mit der Preprozessor-Anweisung

```
#include <math.h> // globale Header-Datei mit mathem. Funktionen
```

Dieses Modul enthält neben dem Quadratwurzel-Funktions-Prototyp noch zahlreiche andere mathematische Funktions-Prototypen, z.B. für die Berechnung von Winkelfunktionen. Es reicht, die globale Header-Datei `math.h` einmal im Hauptprogramm mit `#include` einzubinden, um alle Funktions-Prototypen dieser Header-Datei benutzen zu können. Die globalen Funktionen selbst, die durch diese Funktions-Prototypen angesprochen werden, sind im **C-Compiler** verborgen.

Modul

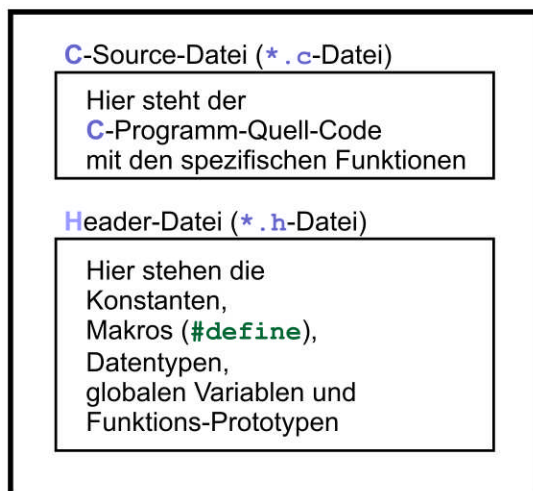


Bild 5.1-01: Das Modul

Neben der (Haupt-)Datei `main.c` existieren - je nach Anwendung - ein oder mehrere Module in einem Projekt. Die Module bestehen in der Regel aus einer `*.h`-Datei und aus einer weiteren `*.c`-Datei. In der `*.c`-Datei (**c** steht für **C**-Source) steht der Programm-Quell-Code mit den Funktionsaufrufen und in der `*.h`-Datei (**h** steht für Header) werden alle Konstanten, Makros, Datentypen, globalen Variablen und Funktions-Prototypen definiert. `*.h`-Dateien können auch in anderen Modulen eingefügt werden, so dass diese dann ebenfalls alle wichtigen Informationen, die in den Header-Dateien festgelegt sind, nutzen können.

Header-Dateien können auch allein für sich stehen, wenn ihr Einsatz als globale oder spezielle Header-Dateien benötigt werden.

Die namensgleichen `*.c`-Dateien der Module sind grundsätzlich in den verschiedenen AVR-C-Projekten nicht nur in ihrem Namen identisch, sondern auch inhaltlich. Im **ATM18-Elektor-CC2-AVR-Projekt** kann es jedoch vorkommen, dass sie bei nachfolgenden Anwendungen erweitert - im Sinne von ergänzt - worden sind. Die `*.c`-Dateien der Module bleiben aber in diesem Projekt auch dann noch zu älteren Anwendungen abwärts kompatibel.

Die namensgleichen Header-Dateien der Module sind dagegen in der Regel von AVR-C-Projekt zu AVR-C-Projekt unterschiedlich, da in ihnen nur die für das Projekt individuell benötigten Definitionen und Funktions-Prototypen festgelegt werden.

5.2 Nomenklatur

Besonders wenn man in einem Team arbeitet ist es wichtig, gemeinsame Regeln für Quelltexte festzulegen. **C** unterscheidet (leider) zwischen der Groß- und Kleinschreibung. So sind

OneFunction und oneFunction

für den **C-Compiler** unterschiedliche Funktionen. Aus diesem Grund werden für alle AVR-C-Projekte folgende Regeln vereinbart:

- Alle Variablen und Funktionsnamen werden generell kleingeschrieben,
- nur Konstante, Makros und Datentypen treten in Großschrift in Erscheinung und
- zusammengesetzte Namen werden mit einem oder mehreren Unterstrichen (`_`) verbunden.

AVR-8-bit-Mikrocontroller

Gruppe 500 - CodeVisionAVR C-Compiler

Teil 505 - Modularer Aufbau der AVR-C-Projekte

Innerhalb eines Moduls haben alle Bezeichnungen einen "Vornamen". So beginnen z.B. alle Funktionsnamen in dem Modul **Typ-Konvertierung** `num_conversion.c` / `num_conversion.h` mit dem Vornamen `nc`. Eine Funktion zur Formatierung von numerischen Ausgaben heißt folglich `nc_format`. Im Quelltext kann man so direkt erkennen, in welchem Modul ein Name definiert ist.

Die englischen Kommentare im **ATM18-Elektor-CC2-AVR-Projekt** sind nicht etwa ein Ausdruck von besonderer Weltläufigkeit, sondern waren eine ursprüngliche Forderung von **Elektor** aufgrund der Veröffentlichungen der Projekte in verschiedenen Ländern. Zug um Zug werden vom Autor die englischen Kommentare durch deutsche Kommentare ersetzt oder zusätzliche deutsche Kommentare eingestreut.

5.3 Die speziellen Header-Dateien

Die speziellen Header-Dateien (`*.h`-Dateien) sind selbst generierte Module ohne `*.c`-Datei. Sie sind speziell für den ATmega88 konzipiert und werden, auch wenn sie nicht gebraucht werden, in allen AVR-C-Projekten eingebunden. Sie stehen neu kommentiert in folgendem besonderen Ordner zur Verfügung:

505_Spezielle_Header_Dateien

- `typedefs.h`
- `iomx8.h`
- `macros.h`
- `switches.h`

Sie sind ebenfalls von Projekt zu Projekt identisch, d.h. sie sind quasi konstant und besitzen keine assoziierte `*.c`-Datei und bilden somit auch kein Modul in der oben geschilderten Form. Wenn man sie einmal in ihrer Funktion begriffen hat, braucht man sie bei späteren Projekten also "nur noch zur Kenntnis" zu nehmen. Auch sollten die vielen Definitionen nicht "irre" machen, sie sind halt wie eine Tabelle zu werten, aus der sich der Preprozessor von Fall zu Fall bedient. Sie kosten später keinen besonderen Speicherplatz, da - wie schon gesagt - sie lediglich von dem Preprozessor benutzt werden.

5.3.1 Die spezielle Header-Datei `typedefs.h` (Typ- und Bit-Definitionen)

Neben ihren folgenden Hauptaufgaben sorgt die Header-Datei `typedefs.h` für die Einfügung der globalen Header-Dateien aus dem Ordner des CVAVR `C:\cvavr_atm18_eval\inc` sowie der speziellen Header-Dateien `iomx8.h`, `macros.h` und `switches.h`.

In `typedefs.h` werden eigene Datentypen definiert. Warum eigene Datentypen? Dafür gibt es mehrere Gründe:

Der erste Grund ist ein rein praktischer. Es ist viel kürzer und einfacher den Datentypen `U32` zu schreiben als `unsigned long`. Besonders bei Funktionsaufrufen mit mehreren Parametern ist das deutlich kürzer. Der zweite Grund liegt in der von der Plattform abhängigen Interpretation von Datentypen. So ist der Datentyp `int` auf 8- und 16-Bit-Controllern ein 16-Bit-Datentyp mit Vorzeichen. Bei 32-Bit-Controllern ist `int` dagegen ein 32-Bit-Datentyp mit Vorzeichen. Wenn in einem Programm konsequent Datentypen aus `typedefs.h` verwendet werden, dann braucht man für die unterschiedlichen Plattformen nur die Header-Datei `typedefs.h` anzupassen. Dann ist z.B. der Datentyp `S16` auf allen Plattformen auch tatsächlich ein 16-Bit-Datentyp mit Vorzeichen.

Darüber hinaus werden in der Header-Datei `typedefs.h` die Werte der "Bit-Namen" definiert. Zum Beispiel wird das gesetzte `BIT7` (Bit mit der Nummer 7) immer den Wert `0x80` (oder binär: `10000000`) erhalten - unabhängig davon, in welchem Register es gesetzt werden soll. Siehe auch **Beispiel 5.3.2-02: Erzeugen der Bit-Werte aus den Bit-Namen.**

5.3.2 Die spezielle Header-Datei `iomx8.h` (Definitionen aller Register-Bits)

In `iomx8.h` wird zunächst die globale Header-Datei `mega88.h` aus dem `inc`-Ordner des CVAVR hinzugefügt. Diese enthält alle spezifischen Definitionen des ATmega88 und macht die Anwendung

AVR-8-bit-Mikrocontroller

Gruppe 500 - CodeVisionAVR C-Compiler

Teil 505 - Modularer Aufbau der AVR-C-Projekte

mit diesem Controller bekannt. Dass hier ATmega88 steht, macht deutlich, dass diese Datei nur für diesen Chip konzipiert ist. Wenn ein anderer Chip verwendet werden soll (also eine andere Plattform gebildet wird), so ist diese Datei zwingend anzupassen! Auch wenn bei den verschiedenen Mikrocontrollern von ATMEL häufig dieselben Register- und Register-Bit-Namen verwendet werden, so sind ihre Hardware-Adressen doch selten identisch.

In `iomx8.h` werden alle Bits zu den gängigen Registern definiert. Ein entscheidender Unterschied ist die Definition als Bit-Wert und nicht als Bit-Nummer.

Beispiel 5.3.2-01: Schreibweise

In vielen C-Programmen kann man Anweisungen wie folgt lesen:

```
TIMSK &=~((1<<TOIE2) | (1<<OCIE2)); // Disable TC2 interrupt
```

Durch die Verwendung von Bit-Werten sieht es dann so aus:

```
TIMSK &=~(TOIE2 | OCIE2); // Disable TC2 interrupt
```

Da man fast ausschließlich Bit-Werte braucht, werden Anweisungen kürzer und übersichtlicher.

Beispiel 5.3.2-02: Erzeugen der Bit-Werte aus den Bit-Namen

Es sollen die Bits `ADEN` (= `BIT7`), `ADATE` (= `BIT5`), `ADIE` (= `BIT3`), `ADPS2` (= `BIT2`), `ADPS1` (= `BIT1`) und `ADPS0` (= `BIT0`) im `ADC Control and Status Register A` auf `1` gesetzt werden. Dazu wird einfach im C-Quellprogramm geschrieben:

```
ADCSRA = ADEN | ADATE | ADIE | ADPS2 | ADPS1 | ADPS0; // ADCSRA = 10101111
```

Gemäß den Definitionslisten der Header-Datei `iomx8.h` werden für die Bit-Namen, die den Originalnamen der ATmega-Register-Summary entsprechen, die zugehörigen Bit-Nummern ausgewählt (rosa markiert):

```
...
// Register ADCSRA
#define ADEN BIT7 // ADC Enable
#define ADSC BIT6 // ADC Start Conversion
#define ADATE BIT5 // ADC Auto Trigger Enable
#define ADIF BIT4 // ADC ADC Interrupt Flag
#define ADIE BIT3 // ADC ADC Interrupt Enable
#define ADPS2 BIT2 // ADC Prescaler Select Bit2
#define ADPS1 BIT1 // ADC Prescaler Select Bit1
#define ADPS0 BIT0 // ADC Prescaler Select Bit0
...
```

Anhand der Bit-Nummern werden dann aus der Header-Datei `typedefs.h` die zugehörigen Bit-Werte (rosa markiert) ermittelt und sie ersetzen den Namen. Dass hier eine umgekehrte Reihenfolge gewählt wurde ist unerheblich!

```
#define BIT0 0x01 // binaer: 00000001
#define BIT1 0x02 // binaer: 00000010
#define BIT2 0x04 // binaer: 00000100
#define BIT3 0x08 // binaer: 00001000
#define BIT4 0x10 // binaer: 00010000
#define BIT5 0x20 // binaer: 00100000
#define BIT6 0x40 // binaer: 01000000
#define BIT7 0x80 // binaer: 10000000
```

Ergebnis des Preprozessors ist:

```
ADCSRA = 0x80 | 0x20 | 0x08 | 0x04 | 0x02 | 0x01; // ADCSRA = 10101111
```

AVR-8-bit-Mikrocontroller
Gruppe 500 - CodeVisionAVR C-Compiler
Teil 505 - Modularer Aufbau der AVR-C-Projekte

5.3.3 Die spezielle Header-Datei `macros.h` (Definitionen von Makros)

Diese Header-Datei ist eine Ansammlung verschiedenster - immer wieder benötigter - Makros, wie zum Beispiel das Einbinden von Assembler-Befehlen und Bit-Manipulationen.

5.3.4 Die spezielle Header-Datei `switches.h` (Definitionen von Schaltern)

Hier werden die Makros definiert, die den Gebrauch der Tast-Schalter vereinfachen sollen. Dazu gehören Makros für die Initialisierung der Port-Pins, an denen die Taster angeschlossen werden sollen, die Definitionen, ob ein Taster betätigt wurde oder nicht und ggf. ob er immer noch gedrückt ist.

5.4 Die AVR-C-Module

Hier wird noch einiges angepasst werden müssen. Wegen der Vereinheitlichung und Übernahme/Neuerstellung weiterer AVR-C-Projekte, können einige AVR-C-Module noch wegfallen, zusammengeführt oder ergänzt werden. Folgende AVR-C-Module wurden in dem **ATM18-Elektor-CC2-AVR-Projekt** bisher verwendet bzw. treten besonders häufig auf (alphabetisch sortiert):

Analog Comparator	<code>ac.c</code>	<code>ac.h</code>
ADC - Auto Trigger	<code>adc.c</code>	<code>adc.h</code>
ADC - Externe Referenz	<code>adc_polled.c</code>	<code>adc_polled.h</code>
ADC - Interne Referenz 1,1 V	<code>adc_ref_1_1.c</code>	<code>adc_ref_1_1.h</code>
ADNS5020-Steuerung	<code>adns5020.c</code>	<code>adns5020.h</code>
Anwendungs-Steuerung	<code>application.c</code>	<code>application.h</code>
Sounderzeugung	<code>beeper.c</code>	<code>beeper.h</code>
Kommando-Ausführung	<code>cmd_func.c</code>	<code>cmd_func.h</code>
Kommando-Erkennung	<code>cmd_parser.c</code>	<code>cmd_parser.h</code>
Zyklische Redundanzprüfung	<code>crc.c</code>	<code>crc.h</code>
DCF77-Steuerung	<code>dcf77.c</code>	<code>dcf77.h</code>
Verzögerungs-Steuerung	<code>delay.c</code>	<code>delay.h</code>
DS1820-Sensor-Steuerung	<code>ds1820.c</code>	<code>ds1820.h</code>
Tastatur-Steuerung	<code>keyboard.c</code>	<code>keyboard.h</code>
Ausgabe auf LCD 20x4	<code>lcd_2wire.c</code>	<code>lcd_2wire.h</code>
LED-Steuerung	<code>led.c</code>	<code>led.h</code>
Magnet-Schwebe-Steuerung	<code>mlc.c</code>	<code>mlc.h</code>
Maus-Steuerung	<code>mouse.c</code>	<code>mouse.h</code>
Typ-Konvertierung nach ASCII	<code>num_conversion.c</code>	<code>num_conversion.h</code>
1-Wire-Steuerung	<code>one_wire.c</code>	<code>one_wire.h</code>
1-Wire-Emulator	<code>one_wire_emulator.c</code>	<code>one_wire_emulator.h</code>
Port-Erweiterung (Relais-Steuerung)	<code>pe_2wire.c</code>	<code>pe_2wire.h</code>
Puls-Phasen-Modulation	<code>ppm_decoder.c</code>	<code>ppm_decoder.h</code>
Puls-Breiten-Modulation	<code>pwm.c</code>	<code>pwm.h</code>
RC5-IR-Fernsteuerung-Dekoder	<code>rc5_decoder.c</code>	<code>rc5_decoder.h</code>
RC5-IR-Fernsteuerung-Enkoder	<code>rc5_encoder.c</code>	<code>rc5_encoder.h</code>
Drehgeber-Steuerung	<code>rotary_encoder.c</code>	<code>rotary_encoder.h</code>
Tastatur-Zeichen-Erkennung	<code>scancodes.c</code>	<code>scancodes.h</code>
Servo-Steuerung	<code>servo.c</code>	<code>servo.h</code>
Song-Steuerung	<code>song.c</code>	<code>song.h</code>
Geräusch-Steuerung	<code>sound.c</code>	<code>sound.h</code>
SPI-Steuerung	<code>spi_master.c</code>	<code>spi_master.h</code>
SPI-Steuerung-Test	<code>spi_master_test.c</code>	<code>spi_master_test.h</code>
Stoppuhr	<code>stop_watch.c</code>	<code>stop_watch.h</code>
Zeit-Basis	<code>t0_timebase.c</code>	<code>t0_timebase.h</code>
Timer0-Steuerung	<code>timer0_pwm.c</code>	<code>timer0_pwm.h</code>
IC2-(TWI)-Steuerung	<code>twi_master.c</code>	<code>twi_master.h</code>
USART-Steuerung	<code>usart.c</code>	<code>usart.h</code>
Wii-Sensor-Steuerung	<code>wii_ir_sensor.c</code>	<code>wii_ir_sensor.h</code>
Wii-Nunchuk-Steuerung	<code>wii_nunchuk.c</code>	<code>wii_nunchuk.h</code>

AVR-8-bit-Mikrocontroller

Gruppe 500 - CodeVisionAVR C-Compiler

Teil 505 - Modularer Aufbau der AVR-C-Projekte

5.5 Anwendung der angepassten AVR-C-Module

Um die Module sinnvoll zu nutzen, muss man wissen, was sie bewirken sollen und wie sie in die Projekte eingebunden werden müssen. Manche Module stellen sehr Mikrocontroller-bezogene Initialisierungs-Funktionen dar (z.B. das Modul `application`), andere stellen eine große Anzahl zusammengehöriger Funktionen zur Verfügung (z.B. das Modul `num_conversion`), die häufig benutzt werden. Viele Module sind für spezielle Steuerungsaufgaben erstellt worden, von denen angenommen wird, dass sie für gleiche Zwecke immer wieder benötigt werden (z.B. das Modul `lcd_2wire`).

Einige der Module, die nachfolgend beschrieben werden, stehen neu kommentiert zum Download in folgendem Ordner zur Verfügung:

505_AVR_C_Modul_Dateien

- `application.c` und `application.h` => `application_general.h`
- `lcd_2wire.c` und `lcd_2wire.h`
- `num_conversion.c` und `num_conversion.h`

5.5.1 Das AVR-C-Modul `application` (Anwendung) (`application.c` / `application.h`)

Die Header-Datei `application.h` dieses Moduls spielt eine besondere Rolle. Alle in einer Anwendung sinnvoll veränderlichen Parameter sind hier einstellbar. Mit dem Funktionsaufruf `app_init()` in der Datei `main.c` werden die Parameter dann initialisiert.

1. Einstellung: Der in der Projekt-Konfiguration eingestellte System-Takt wird mit dem vom CAVR vordefinierten Makro `_MCU_CLOCK_FREQUENCY_` in die Anwendung übernommen. Mit dem Aufruf `app_init()` in der C-Datei `main.c` wird die Funktion in der C-Datei `application.c` angesprochen und der ausgewählte Takt unabhängig von der Einstellung der **Fusebits** eingestellt. Mögliche Frequenzen sind 1, 2, 4, 8 und 16 MHz. Die Definitionen befinden sich in der Header-Datei `application.h`.

Beispiel 5.5.1-01: Einstellung des System-Taktes

```
#define F_CPU          (_MCU_CLOCK_FREQUENCY_)
#define F_CPU_KHZ     (F_CPU / 1000)
#define F_CPU_MHZ     (F_CPU_KHZ / 1000)
```

2. Einstellung: Alle nicht verwendeten Hardware-Eigenschaften bleiben einfach in der `application.h` auskommentiert. Mit dem Funktionsaufruf `app_init()` werden dann nur die benutzten Merkmale, d.h. die Zeilen **ohne** `//`, eingeschaltet.

In dieser Datei ist die Auswahl mancher Definitionen zwingend notwendig, da sie als logische Schalter dienen. Andere Definitionen - besonders auch die in den anderen Header-Dateien - werden wie eine Tabelle genutzt, aus der nur individuell das Passende herausgegriffen wird. In der Header-Datei `application.h` werden manche Ressourcen mehrfach angeführt, d.h. sie werden für mehrere Projekte angeboten - es kann aber jeweils **nur eine Definition gültig sein!**

Beispiel 5.5.1-02: Es wird nur die serielle Schnittstelle `USART0` verwendet

```
//-----
// Definitionen der Hardware-Module, die von den Applikationen benutzt
// werden sollen.
// Es werden nur die wirklich benutzten Statements durch Beseitigung
// der Zeilen-Kommentierung // aktiviert
//-----
//#define USE_TWI          // Soll das TWI- bzw. I2C-Modul benutzt werden?
//#define USE_TIMER2      // Soll der Timer2 benutzt werden?
//#define USE_TIMER1      // Soll der Timer1 benutzt werden?
```

AVR-8-bit-Mikrocontroller

Gruppe 500 - CodeVisionAVR C-Compiler

Teil 505 - Modularer Aufbau der AVR-C-Projekte

```
//#define USE_TIMER0 // Soll der Timer0 benutzt werden?
//#define USE_SPI // Soll das SPI benutzt werden?
#define USE_USART0 // Soll der USART0 benutzt werden?
//#define USE_ADC // Soll der Analog Digital Converter benutzt werden?
//#define USE_ACO // Soll der Analog Comparator benutzt werden?
//#define USE_WATCHDOG // Soll der watchdog timer benutzt werden?
//#define USE_CRYSTAL_CLOCK // Soll der Quarz-Systemtakt benutzt werden?
```

3. Einstellung: Die Verwendung von abstrakten Port-Definitionen anstatt der direkten Verwendung von Register-Namen hat ebenfalls Vorteile. Ein Beispiel zur Ansteuerung der **LED1** mag das verdeutlichen.

Beispiel 5.5.1-03: Ansteuerung der LED1

Die abstrakten Definitionen von `LED2_PRT` und `LED2_BIT` in der Header-Datei `application.h` machen es möglich, dass man bei einer notwendigen Änderung nur an dieser Stelle zu modifizieren braucht, unabhängig davon, an wie vielen verschiedenen Stellen im Projekt z.B. die LED2 ein- oder ausgeschaltet werden soll:

```
//-----
// Definitionen fuer benutzte LED's jeweils für DDR, Port und Bit-Nummer
//-----
//#define LED1_DDR DDRC
//#define LED1_PRT PORTC
//#define LED1_BIT PINC2

#define LED2_DDR DDRC
#define LED2_PRT PORTC
#define LED2_BIT PINC3

//#define LED3_DDR DDRC
//#define LED3_PRT PORTC
//#define LED3_BIT PINC4
```

Diese Methode sollte für (fast) alle Komponenten verwendet werden, die mit den Port-Pins des Mikrocontrollers verbunden werden. Beispiele sind Taster, LC-Display usw. Damit kann man schnell mal das LC-Display an andere Portpins anschließen.

Zusammenfassung:

In der `application.h` können viele grundlegende Einstellungen für die jeweilige Anwendung vorgenommen werden. Um Schreibaufwand zu reduzieren, wurde eine `application_general.h` gebildet, die **alle Applikations-Definitionen** aus allen bisher publizierten **ATM18-Elektor-CC2-AVR-Projekten** enthält.

In dem Ordner `505_AVR_C_Modul_Dateien` ist das `_general` bereits gestrichen worden, so dass hier die neue generalisierte Form - als `application.h` - ausführlich kommentiert erscheint.

`application.h` wird bei einem neuen Projekt einfach in den Projekt-Ordner kopiert und dann werden die benötigten Definitionen von den Kommentierungs-`//` befreit.

5.5.2 Das AVR-C-Modul `lcd_2wire` (LCD-20x4-Steuerung) (`lcd_2wire.c` / `lcd_2wire.h`)

Die Anwendung des Moduls `lcd_2wire` wird im AVR-C-Projekt `602_2_Draht_LCD` ausführlich behandelt. Darüber hinaus ist das Quell-Programm `main.c` dieses Projektes ausführlich kommentiert.

Das komplexe Zusammenspiel der beiden Module `lcd_2wire` und `num_conversion` wird in der Grafik **Bild 5.5.2-01: Zusammenspiel der Module `lcd_2wire` und `num_conversion`** deutlich,

AVR-8-bit-Mikrocontroller

Gruppe 500 - CodeVisionAVR C-Compiler

Teil 505 - Modularer Aufbau der AVR-C-Projekte

wobei zu erkennen ist, dass das Modul `lcd_2wire` ohne das Modul `num_conversion` kaum existieren kann, denn kein Zahlen-Typ kann ohne Konvertierung nach ASCII auf dem LCD angezeigt werden.

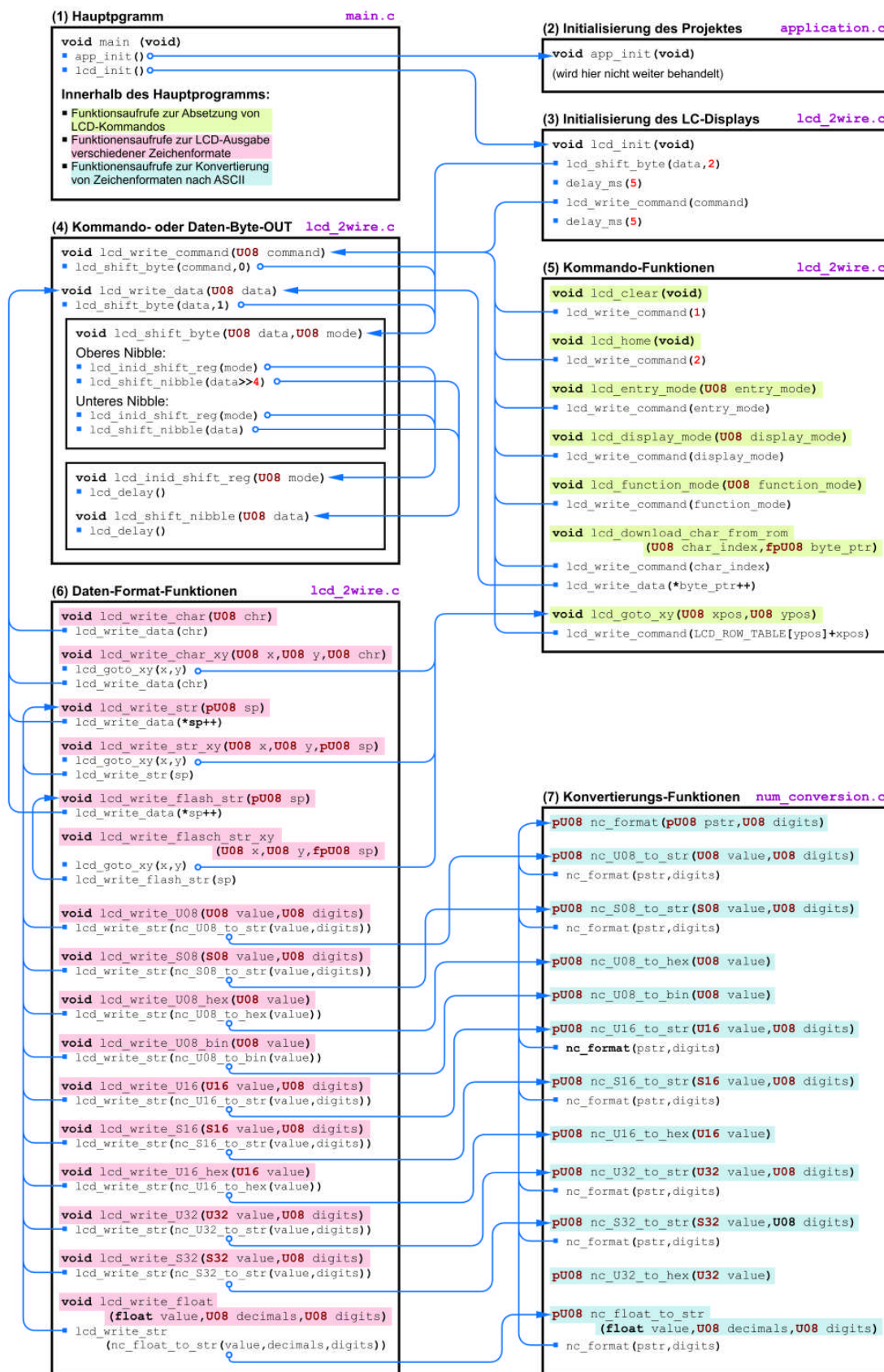


Bild 5.5.2-01: Zusammenspiel der Module `lcd_2wire` und `num_conversion`
(Bildvergrößerung)

AVR-8-bit-Mikrocontroller

Gruppe 500 - CodeVisionAVR C-Compiler

Teil 505 - Modularer Aufbau der AVR-C-Projekte

In globaler Betrachtung stellen beide Module für die verschiedensten Daten-Typen eine große Anzahl von Konvertierungs- und Daten-Format-Funktionen zur Verfügung, um (fast) alle möglichen Daten auf dem LCD anzuzeigen. Natürlich können die Konvertierungs-Funktionen auch für interne Konvertierungsaufgaben herangezogen werden.

FUNKTIONEN ZUM ERZEUGEN VON DISPLAY-KOMMANDOS (eine Auswahl - weitere Funktionen im Listing von `lcd_2wire.c`)

2.6 Aufruf: `lcd_download_char_from_rom(x, FlashPointer);`

Wirkung: Bit6 gesetzt zum Schreiben in das CGRAM: Kommando SET CGRAM ADDRESS
x besteht aus 3 Bits für den CGRAM-Adressteil des Sonder-Zeichens, d.h.
x = 0 bis *x* = 7 ist die Ansteuerungs-Nummer des Sonder-Zeichens (statt ASCII) mit dem Kommando WRITE DATA (Es können bis zu 8 Sonder-Zeichen moduliert werden).
FlashPointer => Pointer auf Punkt-Matrix des Zeichens aus 8 Byte.

2.7 Aufruf: `lcd_goto_xy(Spalte x, Zeile y);`

Wirkung: Bit7 wird in der LCD_ROW_TABLE gesetzt: Kommando SET CURSOR POSITION / SET DDRAM ADDRESS
Setze den Cursor auf die durch *x* und *y* angegebene Position im LCD. Die *Spalte x* gibt die Spalte von 0 bis 19 und die *Zeile y* gibt die Zeile von 0 bis 3 an.

FUNKTIONEN ZUM ERZEUGEN VON ASCII-DATEN-STRINGS

3.1 Aufruf: `lcd_write_char(ASCII-Zeichen);`

Wirkung: Schreibe ein einzelnes ASCII-Zeichen (`unsigned char`) auf die laufende Position im LCD.

3.2 Aufruf: `lcd_write_char_xy(Spalte x, Zeile y, ASCII-Zeichen);`

Wirkung: Schreibe ein einzelnes ASCII-Zeichen (`unsigned char`) auf die durch *x* und *y* angegebene Position im LCD.

3.3 Aufruf: `lcd_write_str(Pointer auf ASCII-String);`

Wirkung: Schreibe ASCII-Zeichen eines Strings Zeichen für Zeichen bis `NULL`-Zeichen erreicht ist auf die laufende Position im LCD.

3.4 Aufruf: `lcd_write_str_xy(Spalte x, Zeile y, Pointer auf ASCII-String);`

Wirkung: Schreibe ASCII-Zeichen eines Strings Zeichen für Zeichen bis `NULL`-Zeichen erreicht ist auf die durch *x* und *y* angegebene Anfangs-Position im LCD.

3.5 Aufruf: `lcd_write_flash_str(Pointer auf String im FLASH-Speicher);`

Wirkung: Schreibe ASCII-Zeichen aus dem Array im FLASH-Speicher Zeichen für Zeichen bis `NULL`-Zeichen erreicht ist auf die laufende Position im LCD.

3.6 Aufruf: `lcd_write_flash_str_xy(Spalte x, Zeile y, Pointer auf String im FLASH-Speicher);`

Wirkung: Schreibe ASCII-Zeichen aus dem Array im FLASH-Speicher Zeichen für Zeichen bis `NULL`-Zeichen erreicht ist auf die durch *x* und *y* angegebene Anfangs-Position im LCD.

3.7 Aufruf: `lcd_write_U08(8-Bit-Dual-Zahl, n-Zeichen);`

Wirkung: Schreibe eine 8-Bit-Dual-Zahl als n-ziffrige Dezimal-Zahl
Wertebereich von 0 bis 255
n-Zeichen: Anzahl darzustellender Stellen insgesamt

3.8 Aufruf: `lcd_write_S08(±8-Bit-Dual-Zahl, n-Zeichen);`

Wirkung: Schreibe eine ±8-Bit-Dual-Zahl als n-ziffrige Dezimal-Zahl
Wertebereich von -128 bis +127
n-Zeichen: Anzahl darzustellender Stellen insgesamt

AVR-8-bit-Mikrocontroller

Gruppe 500 - CodeVisionAVR C-Compiler

Teil 505 - Modularer Aufbau der AVR-C-Projekte

- 3.9 Aufruf:** `lcd_write_U08_hex(8-Bit-Dual-Zahl);`
Wirkung: Schreibe eine *8-Bit-Dual-Zahl* als 2-ziffrige Hexadezimal-Zahl
Wertebereich: 0 bis 255 => 0x00 bis 0xFF
- 3.10 Aufruf:** `lcd_write_U08_bin(8-Bit-Wert);`
Wirkung: Schreibe ein *8-Bit-Wert* in binärer Schreibweise
Wertebereich: 0 bis 255 => 0b00000000 bis 0b11111111
- 3.11 Aufruf:** `lcd_write_U16(16-Bit-Dual-Zahl, n-Zeichen);`
Wirkung: Schreibe eine *16-Bit-Dual-Zahl* als n-ziffrige Dezimal-Zahl
Wertebereich von 0 bis 65535
n-Zeichen: Anzahl darzustellender Stellen insgesamt
- 3.12 Aufruf:** `lcd_write_S16(±16-Bit-Dual-Zahl, n-Zeichen);`
Wirkung: Schreibe eine *±16-Bit-Dual-Zahl* als n-ziffrige Dezimal-Zahl
Wertebereich von -32768 bis +32767
n-Zeichen: Anzahl darzustellender Stellen insgesamt
- 3.13 Aufruf:** `lcd_write_U16_hex(16-Bit-Dual-Zahl);`
Wirkung: Schreibe eine *16-Bit-Dual-Zahl* als 4-ziffrige Hexadezimal-Zahl
Wertebereich: 0 bis 65535 => 0x0000 bis 0xFFFF
- 3.14 Aufruf:** `lcd_write_U32(32-Bit-Dual-Zahl, n-Zeichen);`
Wirkung: Schreibe eine *32-Bit-Dual-Zahl* als n-ziffrige Dezimal-Zahl
Wertebereich von 0 bis 4294967295
n-Zeichen: Anzahl darzustellender Stellen insgesamt
- 3.15 Aufruf:** `lcd_write_S32(±32-Bit-Dual-Zahl, n-Zeichen);`
Wirkung: Schreibe eine *±32-Bit-Dual-Zahl* als n-ziffrige Dezimal-Zahl
Wertebereich von -2147483648 bis +2147483647
n-Zeichen: Anzahl darzustellender Stellen insgesamt
- 3.16 Aufruf:** `lcd_write_float(Gleitkomma-Zahl, d-Bruchstellen, n-Zeichen);`
Wirkung: Schreibe eine *Gleitkomma-Zahl* als Dezimal-Bruch
Wertebereich: ±1.175e-35 bis ±3.402e38
ABER nur 7 Stellen aufeinander folgender Ziffern sind garantiert.
d-Bruchstellen: maximal 5 darzustellende Stellen nach dem Komma
n-Zeichen: Anzahl darzustellender Stellen insgesamt

5.5.3 Das AVR-C-Modul `num_conversion` (Typ-Konvertierung nach ASCII) (`num_conversion.c` / `num_conversion.h`)

Die Anwendung des Moduls `num_conversion` wird ebenfalls im AVR-C-Projekt `602_2_Draht_LCD` behandelt.

FUNKTIONEN ZUR WANDLUNG VERSCHIEDENER FORMATE NACH ASCII

- 4.1 Aufruf:** `nc_format(Pointer auf das Quell-Array, n-Zeichen);`
Wirkung: Formatierung des Übergabe-Strings durch Auffüllen mit vorlaufenden 'Space'.
Der *Pointer* verweist auf die vorderste Ziffer oder das Vorzeichen im Array.
n-Zeichen ist die Anzahl auszugebender Zeichen auf dem LCD
Dies ist eine Hilfsfunktion div. Konvertierungs-Funktionen.
- 4.2 Aufruf:** `nc_U08_to_str(8-Bit-Dual-Zahl, n-Zeichen);`
Wirkung: Wandle eine *8-Bit-Dual-Zahl* in ein ASCII-String um.
Wertebereich von 0 bis 255
n-Zeichen ist die Anzahl darzustellender Stellen insgesamt

AVR-8-bit-Mikrocontroller

Gruppe 500 - CodeVisionAVR C-Compiler

Teil 505 - Modularer Aufbau der AVR-C-Projekte

- 4.3 Aufruf:** `nc_S08_to_str(± 8 -Bit-Dual-Zahl, n-Zeichen);`
Wirkung: Wandle eine ± 8 -Bit-Dual-Zahl in ein ASCII-String um.
Wertebereich von **-128** bis **+127**
n-Zeichen: Anzahl darzustellender Stellen insgesamt
- 4.4 Aufruf:** `nc_U08_to_hex(8-Bit-Dual-Zahl);`
Wirkung: Wandle eine 8 -Bit-Dual-Zahl als 2-ziffrige Hexadezimal-Zahl um und lege ihre Ziffern als ASCII-Zeichen in einem Array ab.
Wertebereich: **0** bis **255** => **0x00** bis **0xFF**
- 4.5 Aufruf:** `nc_U08_to_bin(8-Bit-Wert);`
Wirkung: Wandle einen 8 -Bit-Wert in ein Binär-String um und lege seine Nullen und Einsen als ASCII-Zeichen in einem Array ab.
Wertebereich: **0** bis **255** => **0b00000000** bis **0b11111111**
- 4.6 Aufruf:** `nc_U16_to_str(16-Bit-Dual-Zahl, n-Zeichen);`
Wirkung: Wandle eine 16 -Bit-Dual-Zahl in ein ASCII-String um.
Wertebereich von **0** bis **65535**
n-Zeichen: Anzahl darzustellender Stellen insgesamt
- 4.7 Aufruf:** `nc_S16_to_str(± 16 -Bit-Dual-Zahl, n-Zeichen);`
Wirkung: Wandle eine ± 16 -Bit-Dual-Zahl in ein ASCII-String um.
Wertebereich von **-32768** bis **+32767**
n-Zeichen: Anzahl darzustellender Stellen insgesamt
- 4.8 Aufruf:** `nc_U16_to_hex(16-Bit-Dual-Zahl);`
Wirkung: Wandle eine 16 -Bit-Dual-Zahl in eine Hexadezimal-Zahl um und lege ihre Ziffern als ASCII-Zeichen in einem Array ab.
Wertebereich: **0** bis **65535** => **0x0000** bis **0xFFFF**
- 4.9 Aufruf:** `nc_U32_to_str(32-Bit-Dual-Zahl, n-Zeichen);`
Wirkung: Wandle eine 32 -Bit-Dual-Zahl in ein ASCII-String um.
Wertebereich von **0** bis **4294967295**
n-Zeichen: Anzahl darzustellender Stellen insgesamt
- 4.10 Aufruf:** `nc_S32_to_str(± 32 -Bit-Dual-Zahl, n-Zeichen);`
Wirkung: Wandle eine ± 32 -Bit-Dual-Zahl in ein ASCII-String um.
Wertebereich von **-2147483648** bis **+2147483647**
n-Zeichen: Anzahl darzustellender Stellen insgesamt
- 4.11 Aufruf:** `nc_U32_to_hex(32-Bit-Dual-Zahl);`
Wirkung: Wandle eine 32 -Bit-Dual-Zahl in eine Hexadezimal-Zahl um und lege ihre Ziffern als ASCII-Zeichen in einem Array ab.
Wertebereich: **0** bis **4294967295** => **0x00000000** bis **0xFFFFFFFF**
- 4.12 Aufruf:** `nc_float_to_str(Gleitkomma-Zahl, d-Bruchstellen, n-Zeichen);`
Wirkung: Wandle eine 32 -Bit-Gleitkomma-Zahl in ein ASCII-String um.
Wertebereich: **$\pm 1.175e-35$** bis **$\pm 3.402e38$**
ABER nur 7 Stellen aufeinander folgender Ziffern sind garantiert.
d-Bruchstellen: maximal 5 darzustellende Stellen nach dem Komma
n-Zeichen: Anzahl darzustellender Stellen insgesamt

5.5.4 Das AVR-C-Modul `adc_ref_1_1` (ADC mit interner Referenz 1,1 V)
(`adc_ref_1_1.c` / `adc_ref_1_1.h`)

Noch nicht beschrieben.

5.5.5 Das AVR-C-Modul `rc5_decoder` (RC5-IR-Fernsteuerung-Dekoder)
(`rc5_decoder.c` / `rc5_decoder.h`)

Noch nicht beschrieben.

5.5.6 Das AVR-C-Modul `rc5_encoder` (RC5-IR-Fernsteuerung-Encoder)
(`rc5_encoder.c` / `rc5_encoder.h`)

Noch nicht beschrieben.

5.5.7 Das AVR-C-Modul `usart` (USART-Steuerung)
(`usart.c` / `usart.h`)

Noch nicht beschrieben.

5.5.8 Das AVR-C-Modul `twi_master` (IC2- bzw. TWI-Steuerung)
(`twi_master.c` / `twi_master.h`)

Noch nicht beschrieben.

5.5.9 Das AVR-C-Modul `timer0_pwm` (TIMER0-Steuerung)
(`timer0_pwm.c` / `timer0_pwm.h`)

Noch nicht beschrieben.