

PIC- Programmierung in C (mit CC5X)

Autor:

Letzte Bearbeitung:

Buchgeher Stefan

23. Juli 2006

Inhaltsverzeichnis

1.	WOZU DIESES DOKUMENT	3
2.	MPLAB UND CC5X INSTALLIEREN.....	3
2.1.	Installation von MPLAB (Version 7.31).....	3
2.2.	Einbindung von CC5X in MPLAB.....	5
3.	ERSTE ERKENNTNISSE.....	16
4.	UMGEHEN DER 1K-GRENZE	18
4.1.	Schritt 1: Aufteilen des Projekts in mehrere C-Dateien.....	18
4.2.	Schritt 2: Kompilieren mit einer .BAT-Datei	22
4.3.	Schritt 3: Zusammenfügen der einzelnen .ASM-Dateien.....	22
5.	BESONDERHEITEN BEIM PROGRAMMIEREN MIT MEHREREN MODULEN .	27
5.1.	Externe Register	27
5.2.	Unterprogramme.....	28
5.3.	Interrupt	29
6.	ALLGEMEINE ERKENNTNISSE ZU MPLAB.....	31
7.	QUELLEN	31

1. Wozu dieses Dokument

Der CC5X-Compiler zur Programmierung der PIC16-Mikrocontroller ist anscheinend sehr beliebt. Nicht zuletzt deshalb weil es eine kostenlose gibt. Diese kostenlose Version hat jedoch einige gravierende Nachteile. Der größte Nachteil ist, dass er nur 1k große Programme kompilieren kann. Diesen scheinbaren Nachteil kann man aber glücklicher Weise sehr elegant umgehen. Denn es besteht ja die Möglichkeit das Programm in mehrere C-Dateien aufzuteilen. Jede diese C-Dateien kann dann unabhängig zu den anderen kompiliert werden. Mit einem Linker (z.B. MPLINK) können dann diese einzelnen kompilierten Dateien zu einem gesamten Projekt zusammengefügt werden.

Dieses Dokument soll zeigen, wie man dabei vorgeht.

Ich setze allerdings voraus, dass der Anwender die Programmiersprache C zumindest in den Grundzügen beherrscht.

Achtung: Dieses Dokument ist **kein** Kurs über die C-Programmierung eines PIC-Mikrocontrollers, sondern zeigt nur wie man die 1k-Grenze umgeht!

Zunächst ist es aber erforderlich die Software für den CC5X Compiler zu installieren und in MPLAB zu integrieren. Dies wird im Abschnitt 2 beschrieben.

2. MPLAB und CC5X installieren

Die Entwicklungsumgebung MPLAB ist die von PIC-Hersteller (Microchip) zur Verfügung gestellte Entwicklungsumgebung. Sie besitzt alle für die Assembler-Programmierung notwendigen Werkzeuge. Weiters Werkzeuge für die Simulation, die Fehlersuche (Debugging) und zum Download des fertigen (getesteten) Programms in den PIC-Baustein. C-Programme (oder andere Hochsprachen wie BASIC oder Pascal) können damit aber nicht direkt in ein für den PIC-Baustein verständliches Programm übersetzt werden. Dazu ist ein zusätzliches Programm notwendig, welches aber in die Entwicklungsumgebung MPLAB integriert werden können. Eines dieser Programme ist CC5X.

Ich beziehe mich hier auf die zum Zeitpunkt des 20. Mai 2006 aktuelle Version 7.31 (MPLAB) bzw. 3.2 (CC5X).

2.1. Installation von MPLAB (Version 7.31)

Die Entwicklungsumgebung MPLAB kann kostenlos von der Microchip-Homepage (www.microchip.com) downgeloadet werden.

PIC-Programmierung in C (mit CC5X)

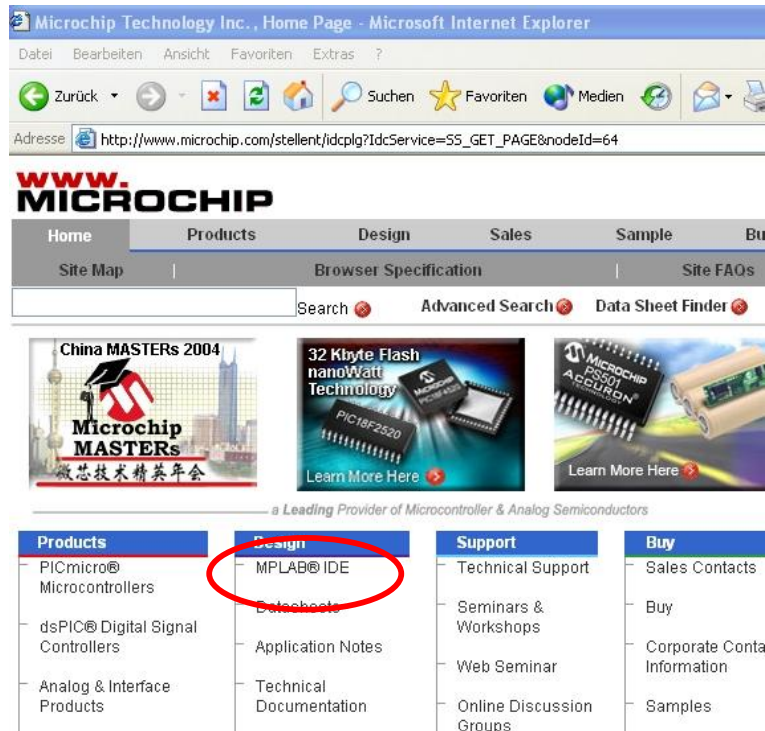


Bild 2.1.

„MPLAB®IDE“ in der Rubrik „Design“ anklicken

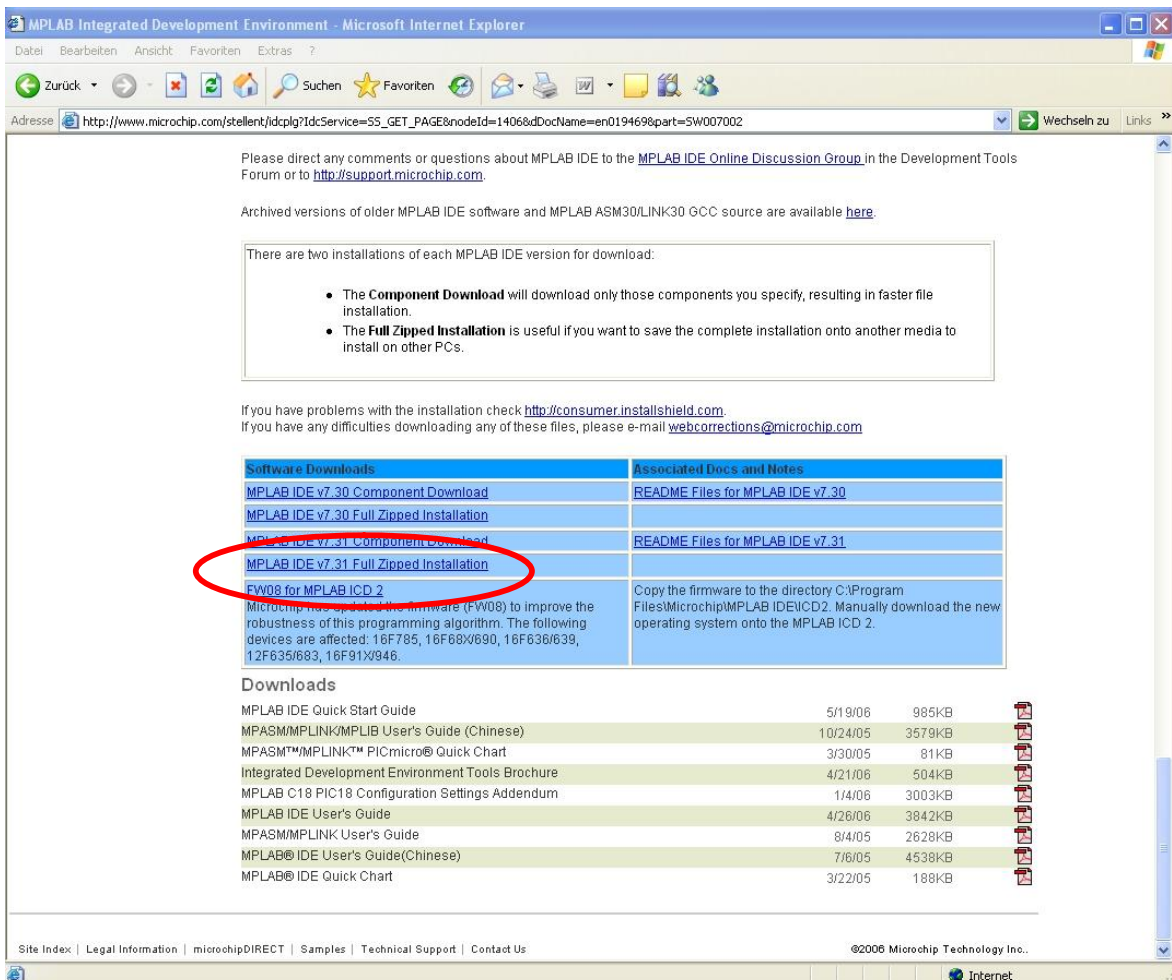


Bild 2.2.

An das Ende der Seite scrollen

„*MPLAB IDE v7.31 Full Zipped Installation*“ anklicken

Download starten.

Ich persönlich speichere diese Datei in einem temporären, lokalen Ordner (z.B.: C:\temp).

Achtung: Die Dateigröße beträgt mehr als 30MB. Der Download dauert daher dementsprechend lange!

Nach erfolgreichem Download diese Datei ausführen (entpacken)

Datei MP731_full_install.exe (im entpackten Ordner) ausführen

License Agreement akzeptieren

Destination Directory: C:\Programme\Microchip

Alle restlichen Fenster können so wie sie sind akzeptiert werden (Taste Next)

Nach der Installation der MPLAB-IDE die downgeladete Datei (bei mir im Ordner C:\temp) löschen, da sie nicht mehr benötigt wird.

2.2. Einbindung von CC5X in MPLAB

Schritt 1: Download von CC5X (Version 3.2):

Eine freie Version des C-Compiler CC5X kann kostenlos unter www.bknd.com/cc5x/index.shtml downgeloadet werden. Link „A [FREE edition of CC5X](#) is available“ anklicken.

Führen Sie die Datei cc5xfree.exe aus und folgen Sie den Anweisungen des Programms

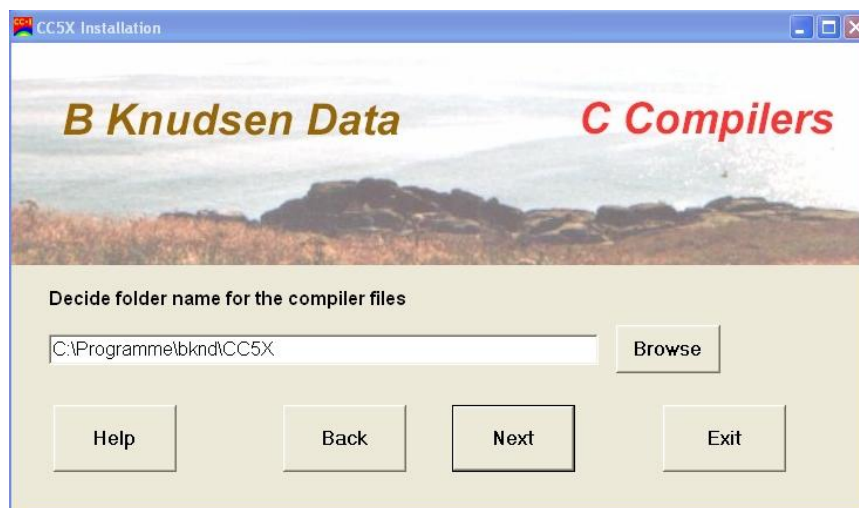


Bild 2.3.

Schritt 2: Der erste Test – Das erste (kleine) Projekt:

Als erstes kleines Projekt sollen alle ungeraden Pins vom Port B (z.B. eines PIC16F628) leuchten.

MPLAB starten à Project à Project Wizard...

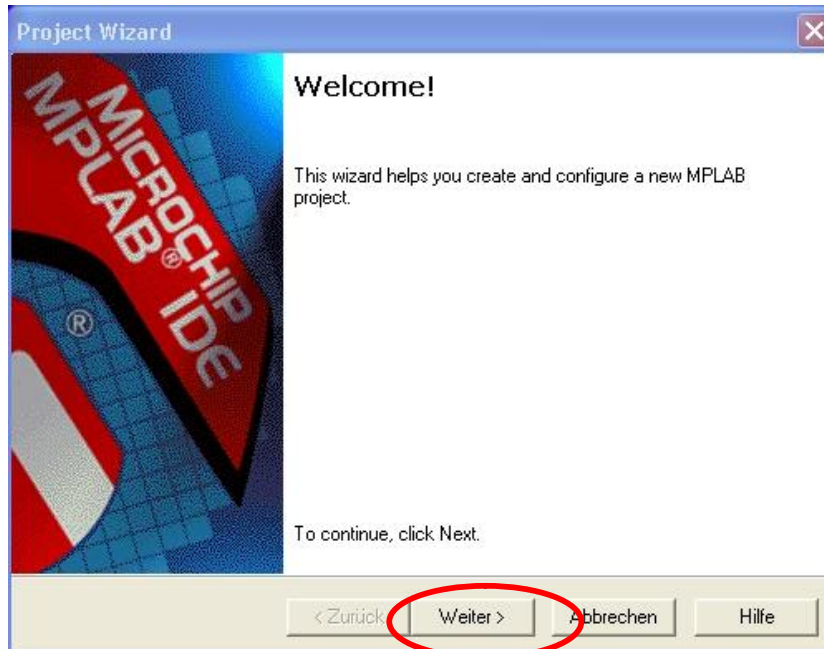


Bild 2.4.

Taste „Weiter >“

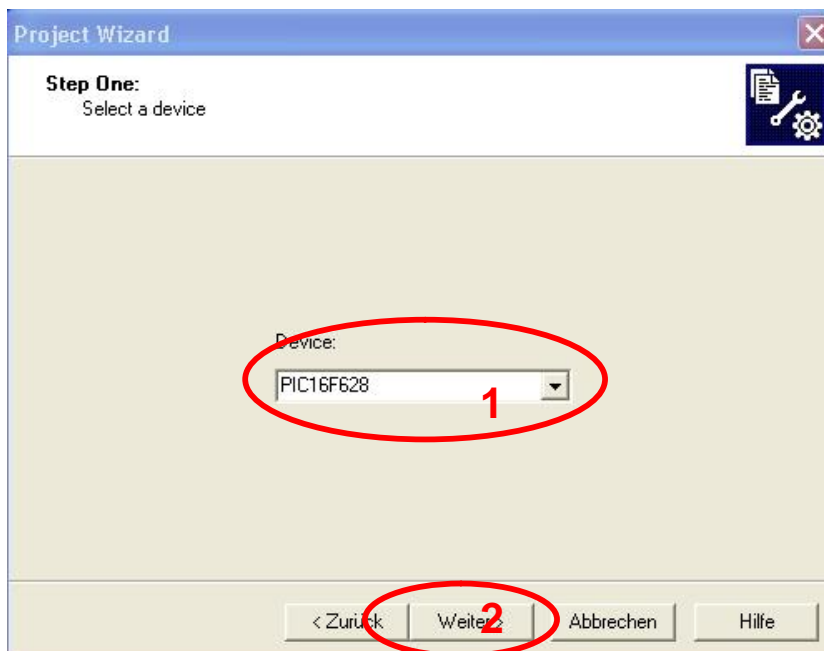


Bild 2.5.

PIC auswählen (z.B. den PIC16F628)

Taste „Weiter >“

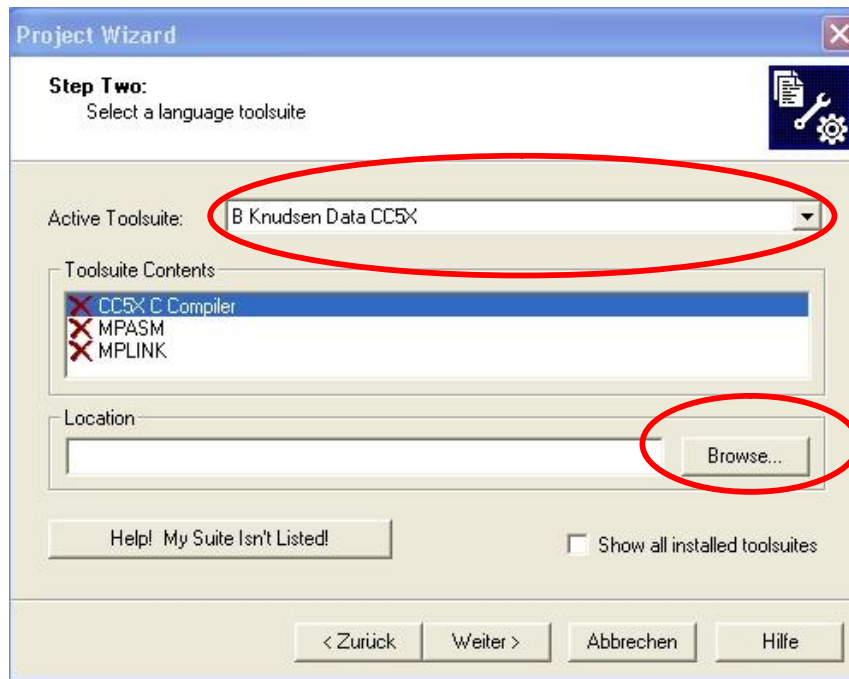


Bild 2.6.

Active Toolsuite: „B Knudsen Data CC5X“ auswählen

Taste „Browse...“

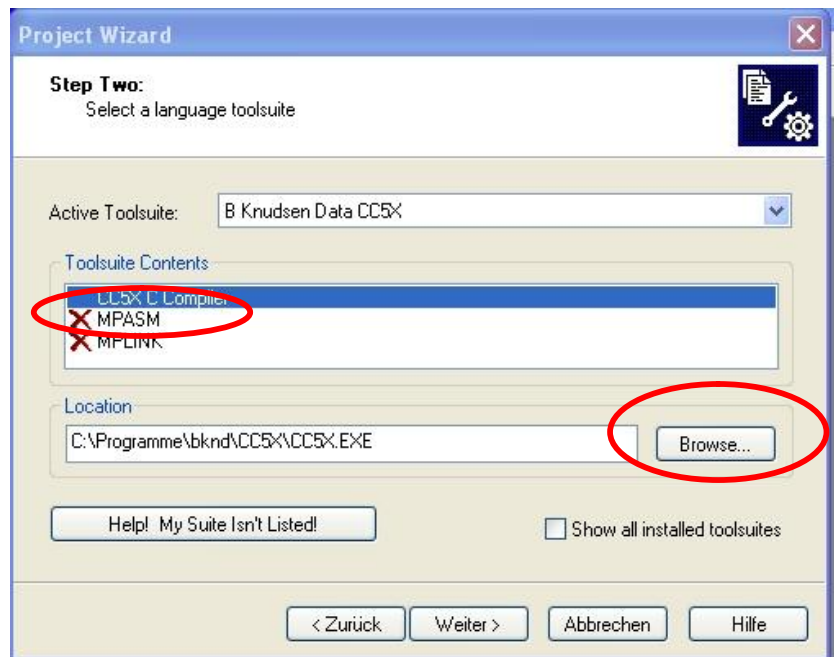


Bild 2.7.

Eintrag „MPASM“ anklicken

Taste „Browse...“

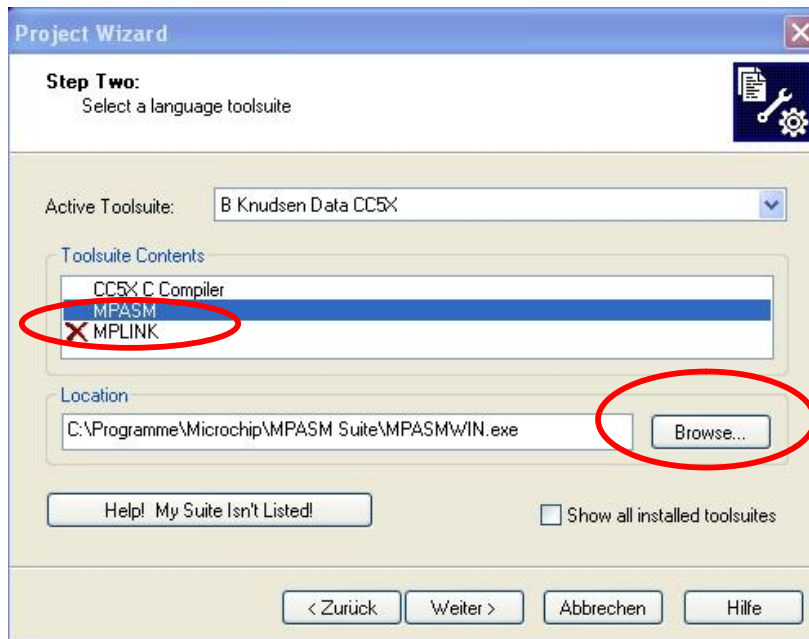


Bild 2.8.

Eintrag „MPLINK“ anklicken

Taste „Browse...“

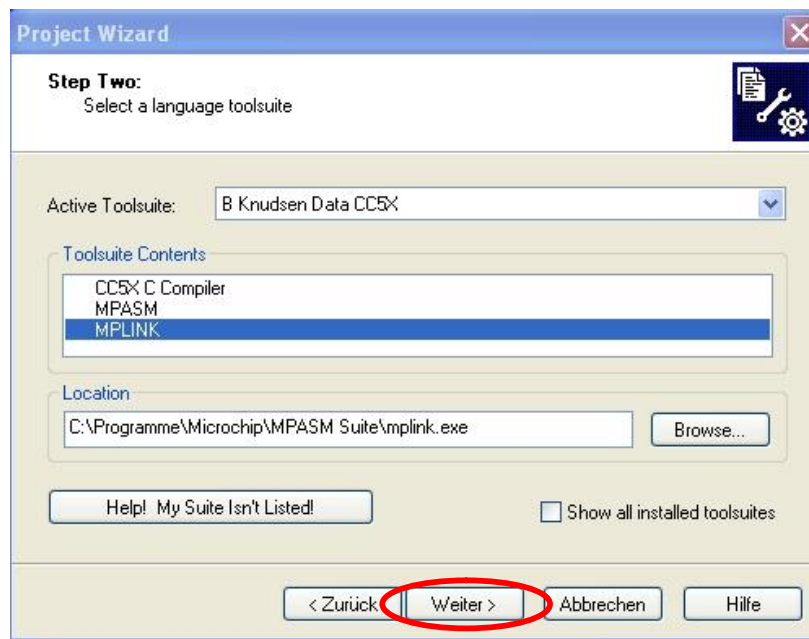


Bild 2.9.

Taste „Weiter >“

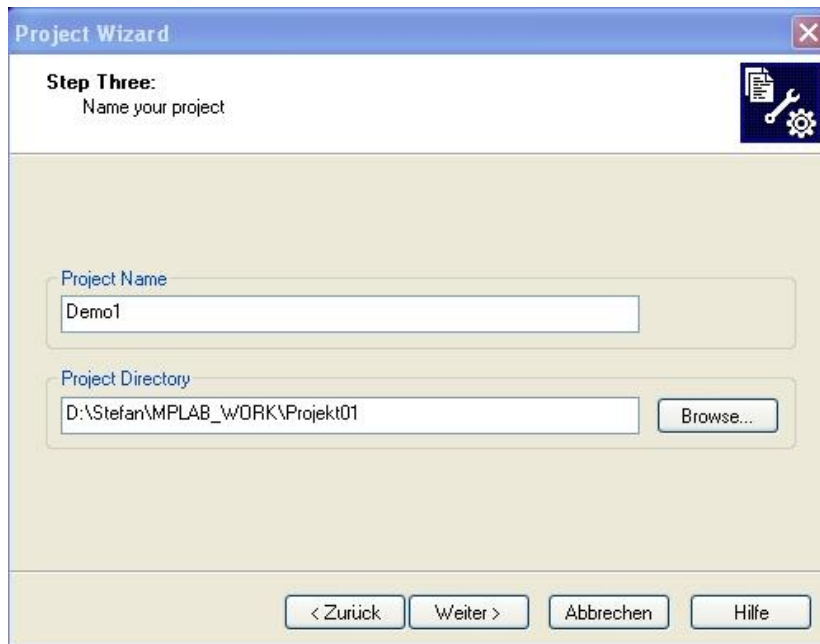


Bild 2.10.

Den Projektordner auswählen, wo dieses Beispiel-Projekt abgelegt werden soll. Dieser Ordner muss schon existieren!
Projektname eingeben (z.B. *Demo1*). Anmerkung: der Projektname muss **nicht** identisch mit dem Ordnername sein!

Taste „*Weiter >*“

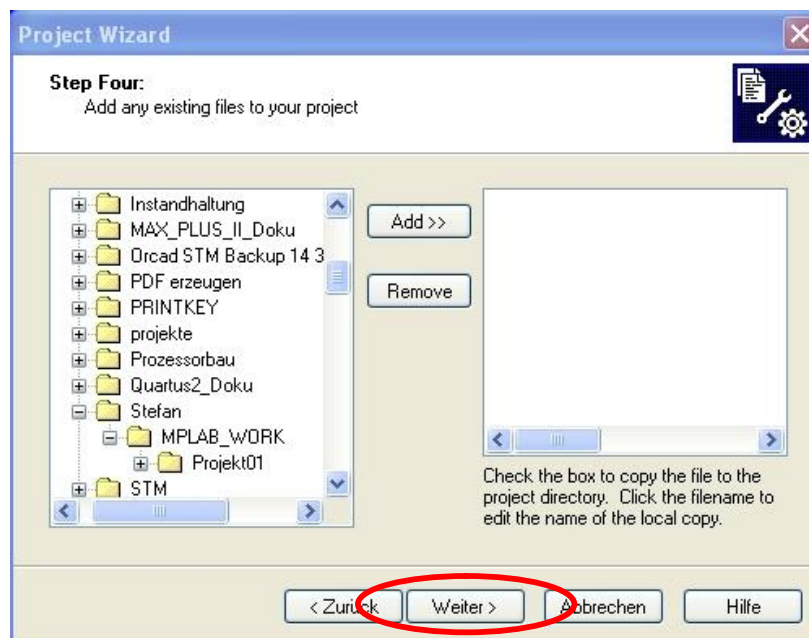


Bild 2.11.

Taste „*Weiter >*“

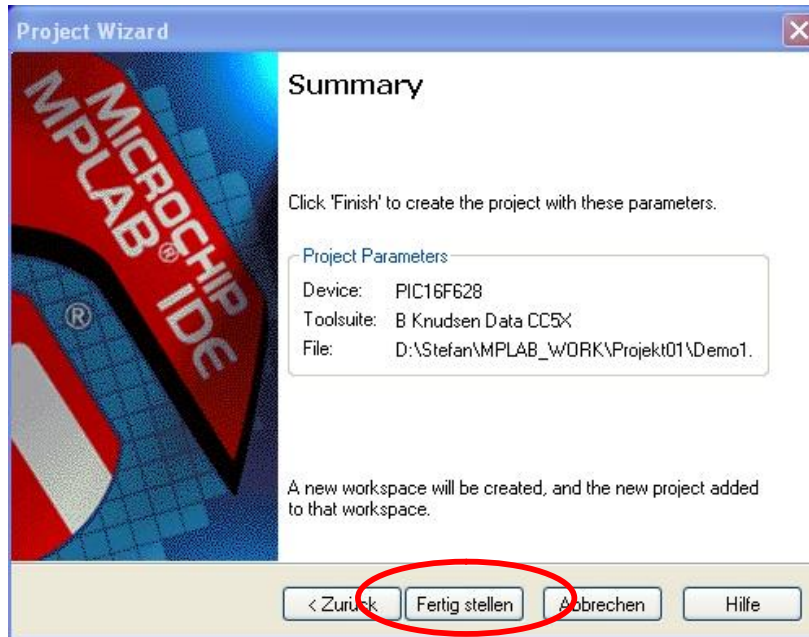


Bild 2.12.

Taste „Fertig stellen“

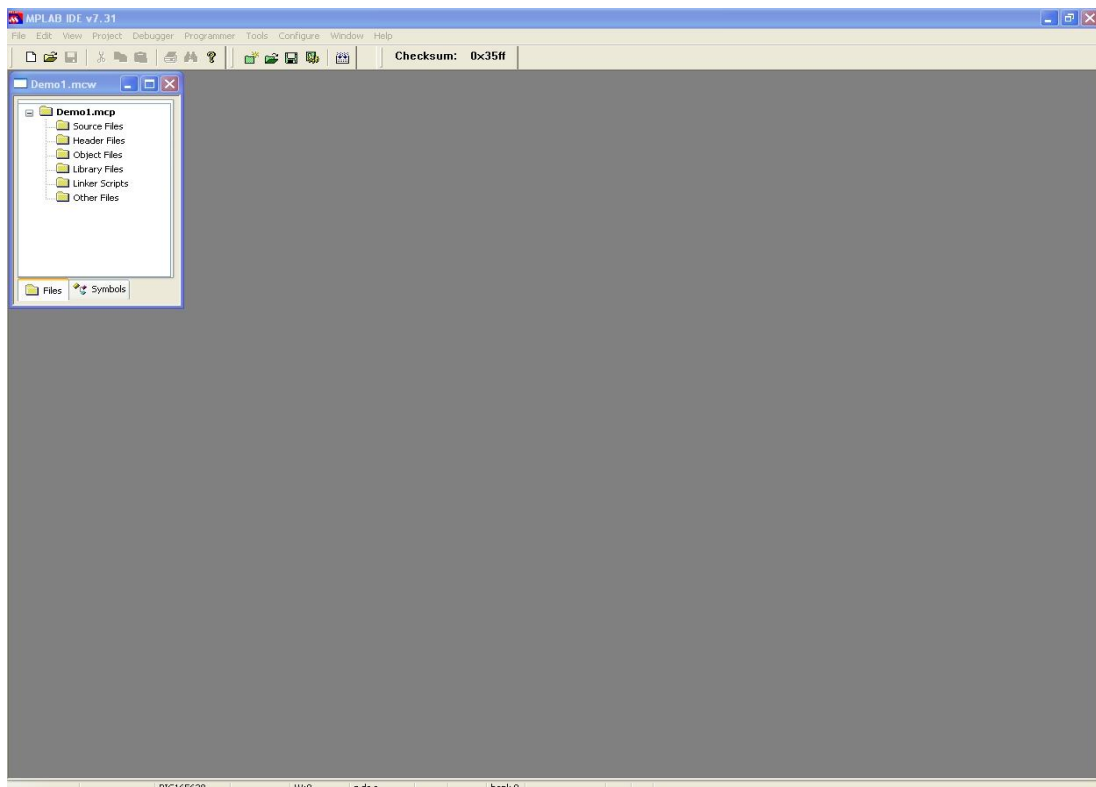


Bild 2.13.

Project à Build Options... à Project

PIC-Programmierung in C (mit CC5X)

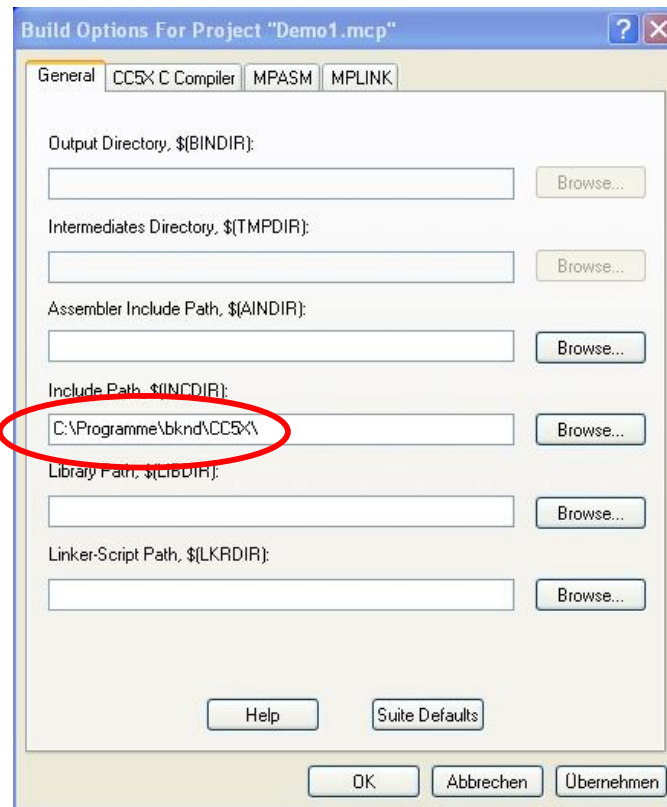


Bild 2.14.

Taste „OK“

File à New

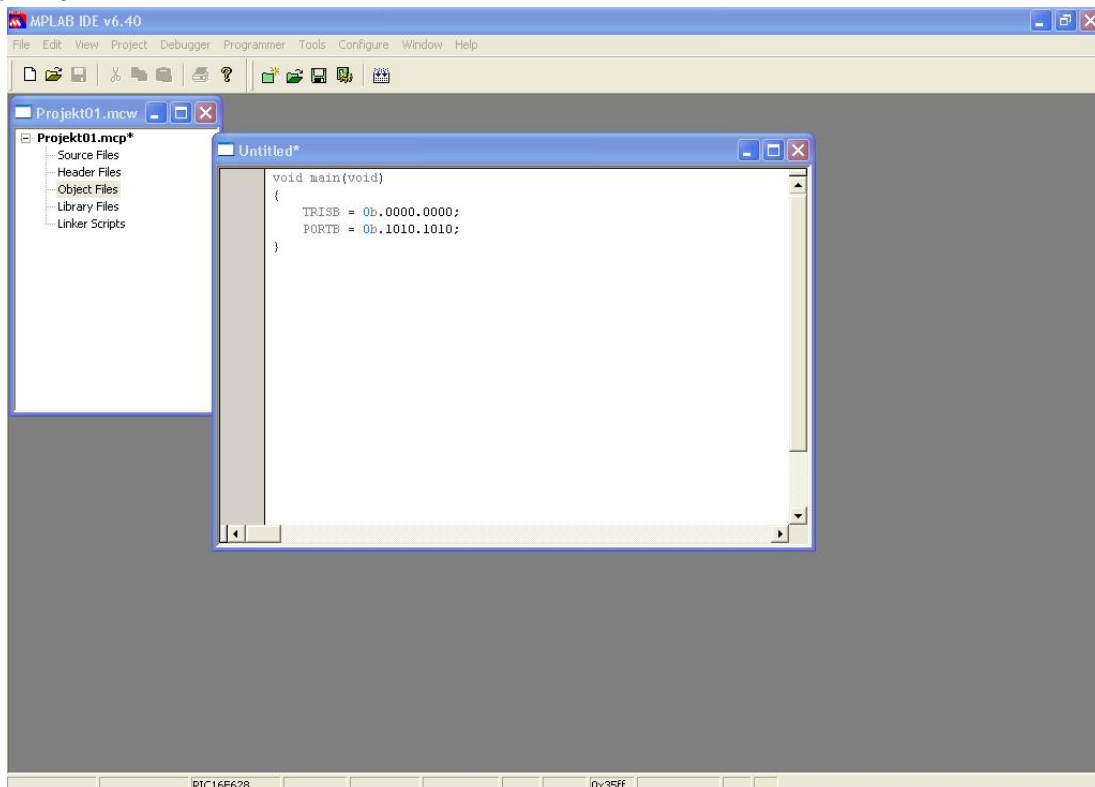


Bild 2.15.

In das sich nun öffnende Fenster geben Sie folgende Zeilen ein:

```
void main(void)
```

```
{  
    TRISB = 0b.0000.0000;  
    PORTB = 0b.1010.1010;  
}
```

Dies ist das erste C-Programm mit welchem wir nun die Funktion der Entwicklungsumgebung testen wollen.

(Dieses Programm initialisiert zunächst den Port B als Ausgang. Anschließend werden nur die ungeraden Portpins vom Port B gesetzt.)

Dieses Programm nun speichern...

File à Save As...

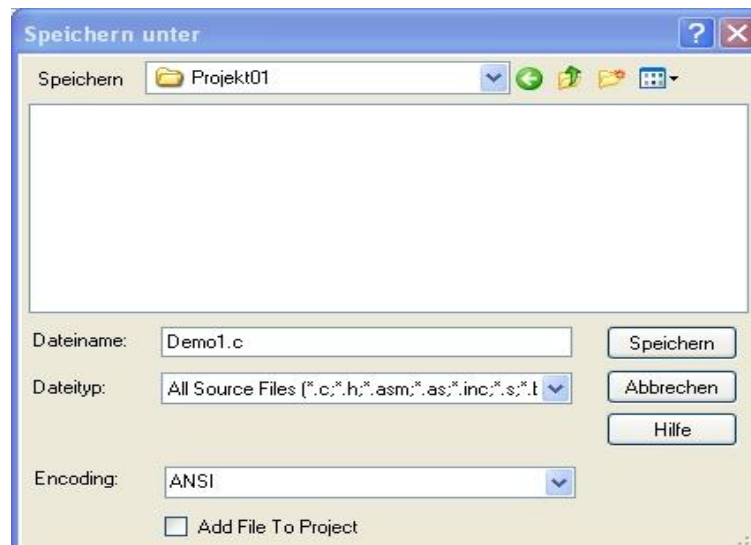


Bild 2.16.

(Anmerkung: Dieses C-File muss im Projektordner abgelegt werden)

Taste „*Speichern*“ nicht vergessen!

... dieses File dem Projekt zuweisen ...

PIC-Programmierung in C (mit CC5X)

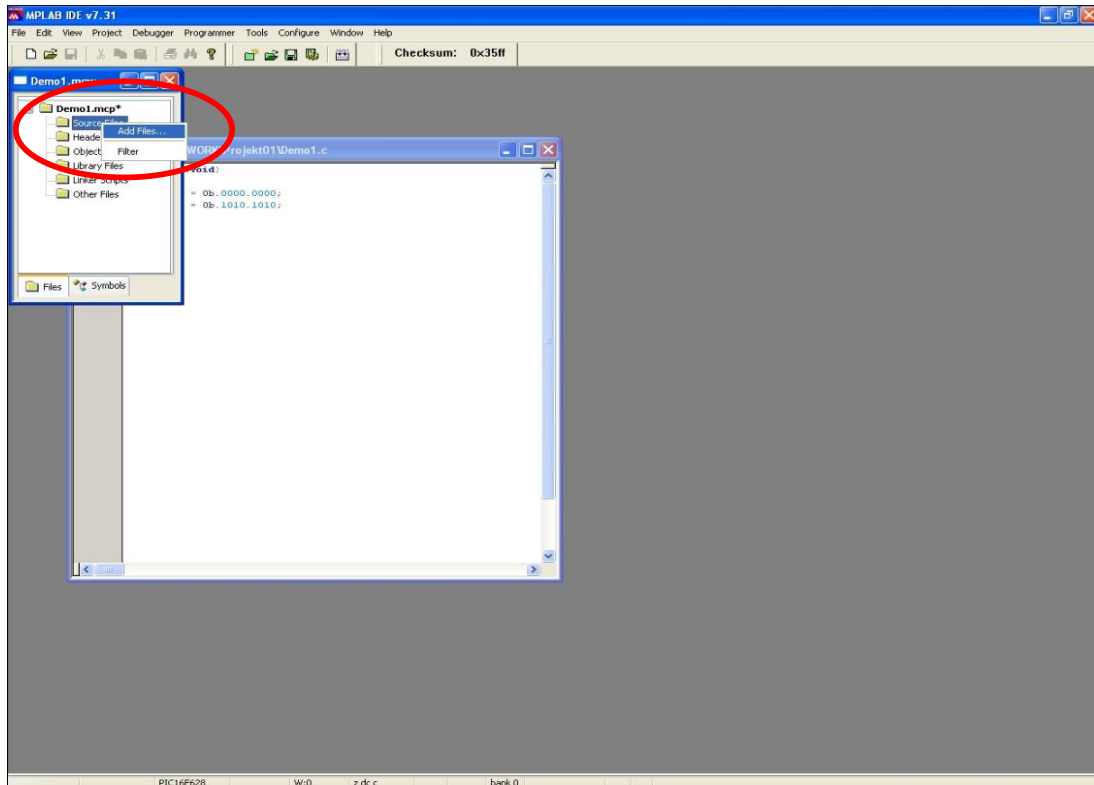


Bild 2.17.

Eintrag „Source Files“ anklicken à rechte Maustaste à „Add Files...“

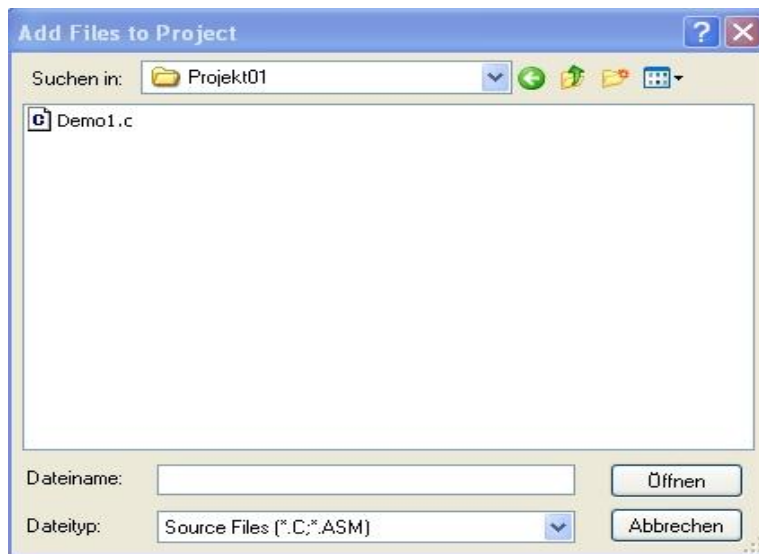


Bild 2.18.

Eintrag „Demo1.c“ auswählen à Taste „Öffnen“

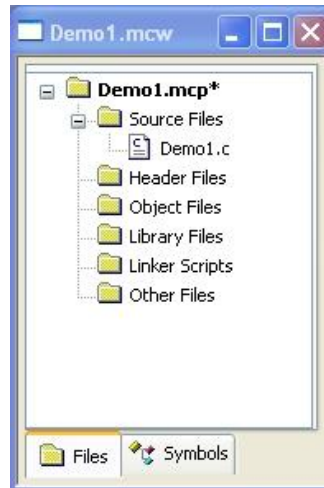


Bild 2.19.

Die Datei „Demo1.c“ gehört nun zum Projekt.

... nun das Programm kompilieren...

Project à Build

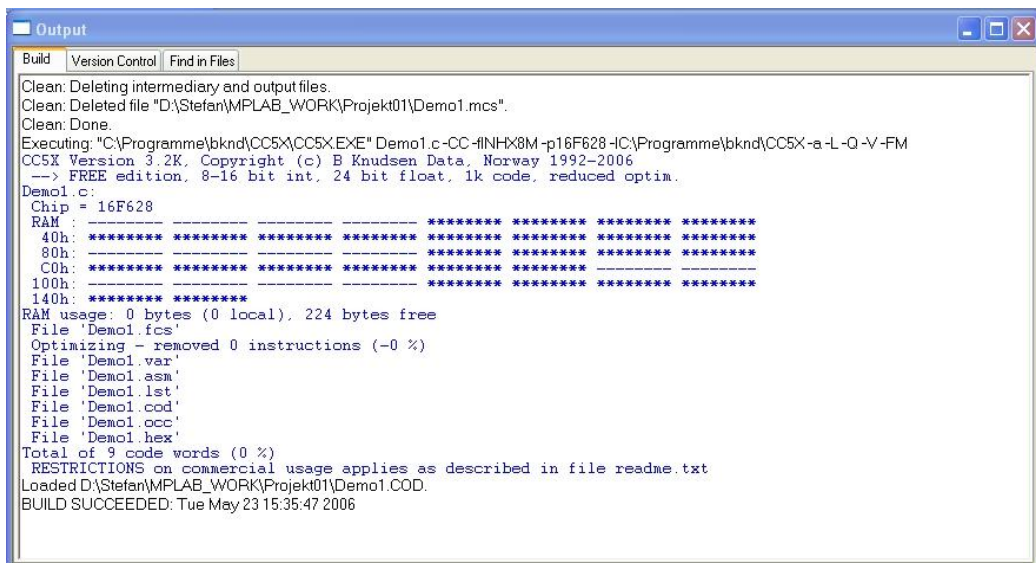


Bild 2.20.

Wurde das Programm erfolgreich kompiliert, so sollte das Ergebnis in etwa Bild 2.20. entsprechen.

... und das fertige HEX-File in den PIC brennen.

Dies erfolgt genau so, als ob Sie ein Programm in Assembler geschrieben hätten. Dazu gibt es verschiedene Möglichkeiten. Ich benutzte hier als Brenner PICSTART-Plus:

Programmer à Select Programmer à PICSTART Plus

Programmer à Enable Programmer

PIC-Programmierung in C (mit CC5X)

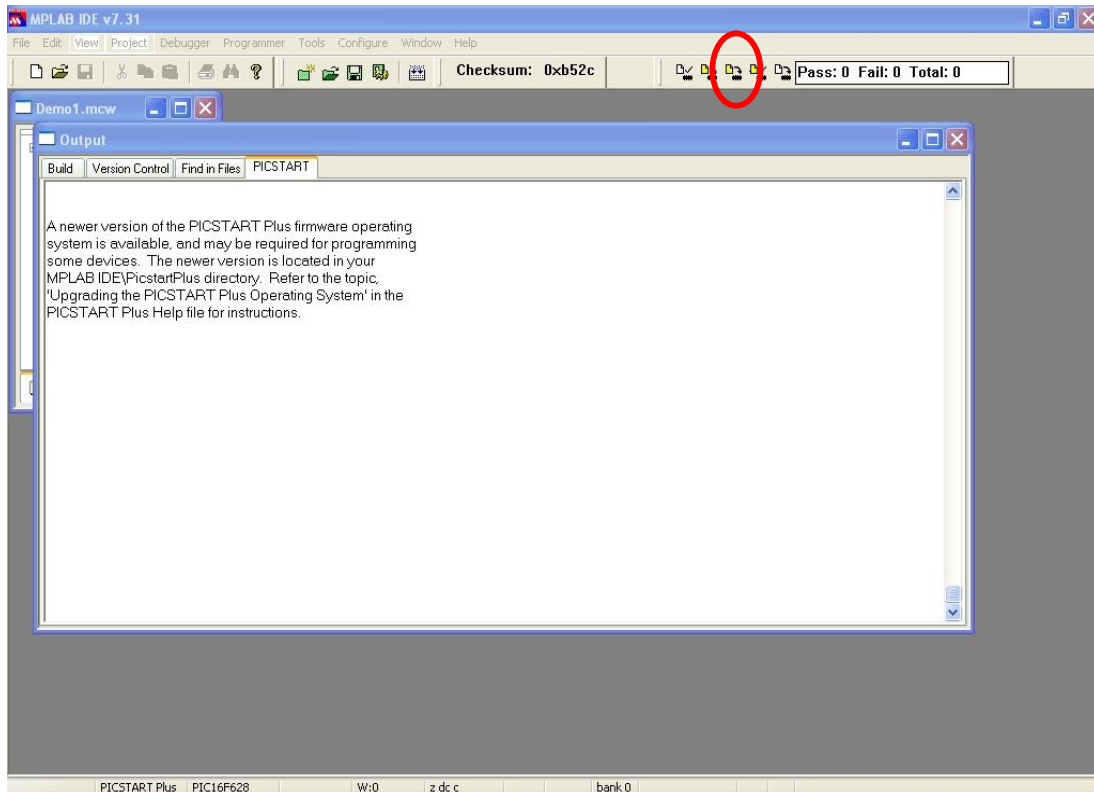


Bild 2.21

Symbol zum Brennen anklicken

Nach dem Brennen den PIC aus dem Brenner nehmen und in die Testschaltung nach Bild 2.22 stecken.

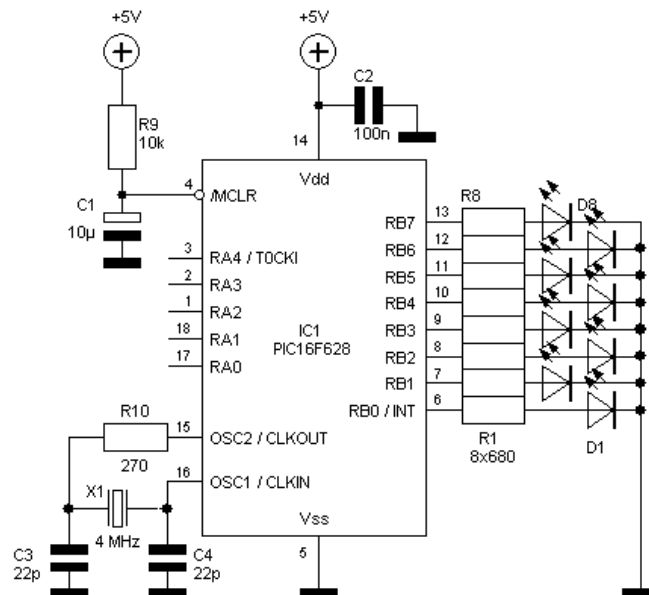


Bild 2.22

Nun wird's spannend. Betriebsspannung anschließen. Und - leuchten die ungeraden Leuchtdioden? – Bei mir schon, und bei Dir?

3. Erste Erkenntnisse

Unterprogramme:

Externe Variablen sind oft besser als die Übergabe von mehreren Parametern beim Aufruf eines Unterprogramms.

Strukturen:

Wird die Adresse (Referenz) einer Struktur an ein Unterprogramm übergeben, so kann auf deren Elemente nur mit dem „Pfeil-Operator“ zugegriffen werden:

Richtig: struktur->element

Falsch: (*struktur).element (Dies funktioniert nicht!!)

Weiters müssen die Elemente einer Struktur lokal zwischengespeichert werden.

z.B.

```
void Unterprogramm (struct struktur *beispiel)
{
    char    elem1;
    char    elem2;
    char    elem3;
    // usw.

    elem1 = beispiel->element1;
    elem2 = beispiel->element2;

    // irgendwelche Anweisungen

    elem3 = .....;

    beispiel->element3 = elem3;
}
```

Aber auch hier ist es besser, wenn man (größere) Strukturen als externe Variablen definiert, und in den Unterprogrammen auf diese zugreift.

Multiplikation mit 2, 4, 8, 16 usw.:

```
a = a << 1;           // Multiplikation mit 2
a = a << 2;           // Multiplikation mit 4
a = a << 3;           // Multiplikation mit 8
a = a << 4;           // Multiplikation mit 16
```

Division durch 2, 4, 8, 16 usw.:

```
a = a >> 1;           // Division durch 2
a = a >> 2;           // Division durch 4
a = a >> 3;           // Division durch 8
a = a >> 4;           // Division durch 16
```

Division einer int16 –Variable mit 2 (oder 4, 8, 16 usw.):

```
int16  a;           //Vorzeichenbehaftete 16-Bit-Zahl

a = a / 2;         // Fehlermeldung

a = a / (uns16)2;  // keine Fehlermeldung, aber falsche Ausführung
                  // ASM-File zeigt zwei Schiebebefehle, was aber bei
                  // einer vorzeichenbehafteten Zahl nicht stimmen kann!
```

Einfachste Variante mit Schiebebefehl:

```
a = a >> 1;        // erfolgt bei int16-Variablen auch vorzeichenrichtig
                  // Übersetzte Assembleranweisung:
                  // bsf    0x03,Carry
                  // btfsc  a+1,7
                  // bcf    0x03,Carry
                  // rrf    a+1,1
                  // rrf    a,1
```

Over- und Underflow bei Addition und Subtraktion erkennen (bei vorzeichenbehafteten Ganzen Zahlen)

Das Problem des Überlaufs tritt dann auf wenn entweder zwei positive oder zwei negative Zahlen addiert werden. Bei einer Addition von einer positiven mit einer negativen Zahl kann hingegen nie ein Überlauf auftreten. Bei einer Subtraktion verhält es sich anders herum. Hier kann nur ein Überlauf auftreten, wenn eine positive Zahl von einer negativen Zahl abgezogen wird, oder eine negative von einer positiven Zahl.

Einen Überlauf erkennt man folgendermaßen:

Ist das Ergebnis einer Addition zweier positiver Zahlen **negativ**, so trat ein Überlauf auf.
Ist das Ergebnis einer Addition zweier negativer Zahlen **positiv**, so trat ein Überlauf auf.

z.B:

```
int8 a;
int8 b;
int8 summe;

a = 57;
b = 109;

summe = (int8)(a + b);

if ((a > 0) && (b > 0) && (summe <= 0))
{
    // positiver Überlauf (Overflow)
}

if ((a < 0) && (b < 0) && (summe >= 0))
{
    // negativer Überlauf (Underflow)
}
}
```

4. Umgehen der 1k-Grenze

Zum umgehen der 1k-Grenze sind folgende drei Schritte notwendig:

- Schritt 1: Aufteilen des Projekts in mehrere C-Dateien
- Schritt 2: Kompilieren mit einer .BAT-Datei
- Schritt 3: Zusammenfügen der einzelnen .ASM-Dateien

Bild 4.1 zeigt diesen prinzipiellen Vorgang

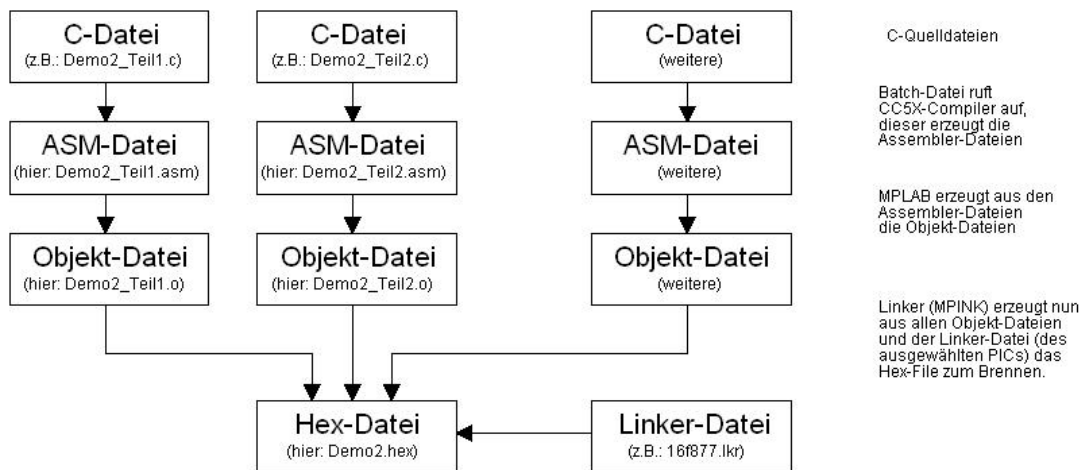


Bild 4.1.

4.1. Schritt 1: Aufteilen des Projekts in mehrere C-Dateien

Wird also eine C-Datei so groß, dass es beim Kompilieren die 1k-Grenze überschreitet, so muss es in mehrere C-Dateien aufgeteilt werden, wobei auch hier gilt: Jede C-Datei darf beim kompilieren die 1k-Grenze nicht überschreiten!

Im Prinzip sieht dann das gesamte Projekt zum Beispiel folgendermaßen aus:

Hier bestehend aus zwei C-Dateien (Demo2_Teil1.C und Demo2_Teil2.C) und einer Header-Datei (PROJEKT.H).

(Achtung: Es geht hier nur um das Prinzip, daher ist der folgende Code-Ausschnitt kein voll funktionsfähiges Projekt!)

Erste C-Datei (z.B. Demo2_Teil1.C)

```

/*****
/* Projekt zur Demo (Umgehung der 1k-Grenze) */
/*
/* Entwickler: Buchgeher Stefan */
/* Entwicklungsbeginn der Software: 20. Mai 2006 */
/* Funktionsfaehig seit: 20. Mai 2006 */
/* Letzte Bearbeitung: 20. Mai 2006 */
*****/

/***** Include-Dateien *****/
#include "PROJEKT.H"
#include <int16CXX.H> // Ist fuer die Interrupts notwendig
#include <INLINE.H> // Ist fuer Assembleranweisungen notwendig
    
```

PIC-Programmierung in C (mit CC5X)

```
/****** Konfigurations-Bits *****/
#pragma config |= 0b.11110100111010

/****** ISR - Timer0 *****/

/* Interrupt Service Routine (ISR): */
/****** */
#pragma origin 4

interrupt ISR(void) // Interruptroutine
{
    int_save_registers // W, STATUS (und PCLATH) retten

    // Beginn der eigentlichen ISR-Routine

    int_restore_registers // W, STATUS (und PCLATH) Wiederherstellen
}

/****** INIT: */
/* */
/* Aufgabe: */
/* Initialisierung des Prozessor: */
/****** */
void INIT(void)
{
    // Timer-0-Interrupt
    TMRO = 0; // Timer 0 auf 0 voreinstellen
    OPTION = 0b.1000.0011; // Pull-Up-Widerstaende am Port B deaktivieren
    // Timer-ISR (Vorteiler = 16)

    // Ports
    TRISA = 0xFF; // Port A und Port E muessen bei verwendung als
    TRISE = 0xFF; // ADC als Eingang konfiguriert werden
    TRISB = 0; // Port B als Ausgang definieren
    TRISC = 0b.1111.1001; // Port C: Bits 1 und 2 als Ausgang definieren
    // Port C: Bits 0 und 3 bis 7 als Eingang definieren
    TRISD = 0; // Port D als Ausgang definieren

    // ADC
    ADON = 1; // ADC global einschalten
    ADCON1 = 0b.1000.0010; // Ganzer Port A als Analog; Vref+ = Vdd, Vref- = Vss
    // und linksbuendig
    ADCS0 = 0; // Geschwindigkeit des ADC mit ADCS0 und ADCS1
    ADCS1 = 1;

    // usw.
}

void delay_us(char mikro)
{
    #asm
    wdh9 nop
    nop
    decfsz mikro,f
    goto wdh9
    #endasm
}

/****** Unterprogramm zur Analog-Digital-Wandlung *****/

/****** */
/* ADC: */
/* */
/* Aufgabe: */
/* Analog-Digital-Wandlung mit dem PIC-interne ADC fuer den ausgewaehlten ADC-Kanal */
/* starten, auf das Wandlungsergebnis warten und dieses dem aufrufenden Programm */
/* zurueckgeben. */
/* */
/* Uebergabeparameter: */
/* kanal: */
/* */
```

PIC-Programmierung in C (mit CC5X)

```
/* */
/* Rueckgabeparameter: */
/* 6-Bit-Ergebnis der Analog-Digital-Wandlung im EXTERNEN UEBERGABEREGISTER */
/* (tmp_uebergabe_uns16) */
/* */
/* Vorgehensweise: */
/* + ADC-Kanal auswaehlen (Bits <3:5> im Register ADCON0). Dazu muss die Kanal-Nummer */
/* an diese Position geschoben werden und mit dem Register ADCON0 verknuepft werden */
/* + ca. 50 Mikrosekunden (us) warten. Diese Zeit benoetigt der PIC um den internen */
/* Kondensator mit der zu messenden Analogspannung zu laden. */
/* + Analog-Digital-Wandlung starten. Dazu das Flag GO (im Register ADCON0) setzen. */
/* + Warten, bis der PIC mit der Wandlung fertig ist. Der PIC setzt das Flag GO auto- */
/* matisch zurueck, wenn er mit der Analog-Digital-Wandlung fertig ist. */
/* + Aus den Registern ADRESL und ADRESH das Ergebnis zusammensetzen und an das auf- */
/* rufende Unterprogramm (oder Hauptprogramm) zurueckgeben */
/* */
/* Anmerkung: */
/* Das Ausgabeformat der ADC-Wandlung, also wie die 10 Ergebnisbits in den Registern */
/* ADRESL und ADRESH abgelegt werden wird an anderer Stelle (z.B. im Unterprogramm */
/* INIT) konfiguriert. */
/* */
/*****
void ADC(char kanal)
{
    ADCON0 = ADCON0 & 0b.1100.0111;
    kanal = kanal << 3;
    ADCON0 = ADCON0 | kanal;

    delay_us(50);

    GO = 1;
    while(GO);

    tmp_uebergabe_uns16.low8 = ADRESL; // externes Uebergaberegister
    tmp_uebergabe_uns16.high8 = ADRESH;
}

/***** Hauptprogramm *****/
void main(void)
{
    char Wert1;

    INIT(); // Controller initialisieren

    INTCON = 0b.1010.0000; // Timer0 freigeben durch Setzen von
                          // GIE und T0IE im Register INTCON

    while(1)
    {
        // aktuelle Werte einlesen
        Soll_Istwerte_einlesen();

        // Test-Routine
        UNTERPROGRAMM2();
        Wert1 = UNTERPROGRAMM3(123);
    }
}

```

Zweite C-Datei (z.B. Demo2_Teil2.C)

```

/*****
/* Unterprogramme zur Demo (Umgehung der 1k-Grenze) */
/* */
/* Entwickler: Buchgeher Stefan */
/* Entwicklungsbeginn der Software: 20. Mai 2006 */
/* Funktionsfaehig seit: 20. Mai 2006 */
/* Letzte Bearbeitung: 20. Mai 2006 */
/*****

/***** Projekt-Header einbinden *****/
#include "PROJEKT.H"

/***** externe Register *****/
uns16 tmp_uebergabe_uns16;

uns16 sollwert;
uns16 istwert;

```

PIC-Programmierung in C (mit CC5X)

```
/****** weitere Unterprogramme *****/
/******
/* Unterprogramm 2
/*
/******
void UNTERPROGRAMM2(void)
{
    // irgendwelche Anweisungen
}

/******
/* Unterprogramm 3
/*
/******
char UNTERPROGRAMM3(char Parameter)
{
    char wert;

    // irgendwelche Anweisungen

    return (wert);
}

/****** Unterprogramm zur Ist- und Sollwerteingabe *****/
/******
/* ADC:
/*
/* Aufgabe:
/* Sollwert und Istwert von den analogen Eingaengen AN0 und AN1 einlesen und in den
/* externen Registern istwert und sollwert sichern
/******
void Soll_Istwerte_einlesen (void)
{
    // Istwert von ADC-Eingang AN0 (Kanal 0) einlesen und in der externen Variable
    // istwert sichern
    ADC(0);
    istwert = tmp_uebergabe_uns16;

    // Sollwert von ADC-Eingang AN2 (Kanal 2) einlesen und in der externen Variable
    // sollwert sichern
    ADC(1);
    sollwert = tmp_uebergabe_uns16;
}
```

Gemeinsame Header-Datei (z.B. PROJEKT.H)

```
/******
/* Header zur Demo (Umgehung der 1k-Grenze)
/*
/*
/* Entwickler: Buchgeher Stefan
/* Entwicklungsbeginn der Software: 20. Mai 2006
/* Funktionsfaehig seit: 20. Mai 2006
/* Letzte Bearbeitung: 20. Mai 2006
/******

#ifndef __PROJEKT
#define __PROJEKT

/****** Pragma-Anweisungen *****/
#pragma chip PIC16F877 // PICmicro Device

/****** Externe Register *****/
extern bank0 uns16 tmp_uebergabe_uns16;

extern bank0 uns16 sollwert;
extern bank0 uns16 istwert;
```

```

/***** Funktionsprototypen *****/
/* Initialisierung des Mikrocontroller */
extern page0 void INIT(void);

/* Unterprogramm zur Analog-Digital-Wandlung */
extern page0 void ADC(char kanal);

/* Unterprogramm zur Ist- und Sollwerteingabe */
extern page0 void Soll_Istwerte_einlesen(void);

/* Weitere Unterprogramme */
extern page0 void UNTERPROGRAMM2(void);
extern page0 char UNTERPROGRAMM3(char Parameter);

#endif

```

4.2. Schritt 2: Kompilieren mit einer .BAT-Datei

Im nächsten Schritt wird nun jede C-Datei (mit Hilfe des CC5X-Compilers) in eine ASM-Datei kompiliert. Diese Aufgabe übernimmt eine BAT-Datei. Dazu geht man wie folgt vor:

1) Im Projektordner eine neue Textdatei erzeugen. Der Name der Datei ist egal, wichtig ist nur die Endung `.bat`. (Also z.B. `Demo2.bat`). Es entsteht also eine Datei vom Typ „Stapelverarbeitungsdatei für MS-DOS“.

2) Diese Datei (`Demo2.bat`) mit der rechten Maustaste anklicken und im Pop-Up-Menü „bearbeiten“ auswählen. Im Editor nun die folgenden Einträge einfügen:

```

C:\Programme\bknd\CC5X\CC5X.EXE Demo2_Teil1.C -IC:\Programme\bknd\CC5X\ -u -r -a -r2
C:\Programme\bknd\CC5X\CC5X.EXE Demo2_Teil2.C -IC:\Programme\bknd\CC5X\ -u -r -a -r2

```

Hinweise:

- Der **blau** hinterlegte Teil gibt an wo sich der CC5X-Compiler befindet
- Der **rot** hinterlegte Teil ist der Name der C-Datei
- Für jede C-Datei muss eine entsprechende Zeile eingefügt werden

3) Editor schließen (Datei natürlich vorher speichern)

4) Batch-Datei (hier `Demo2.bat`) mit einem Doppelklick ausführen. Wenn in den C-Dateien alles richtig ist (also die Syntax) werden die dazugehörigen `.ASM`-Dateien erzeugt (hier `Demo2_Teil1.ASM` bzw. `Demo2_Teil2.ASM`). Bei einem Syntaxfehler (in der C-Datei) wird die eventuell schon vorhandene `.ASM`-Datei gelöscht, und die `.OCC`-Datei gibt Auskunft über die aufgetretenen Syntaxfehler.

4.3. Schritt 3: Zusammenfügen der einzelnen .ASM-Dateien

Die im vorhergehenden Schritt erzeugten Assembler-Dateien müssen nun zu einem gesamten Projekt zusammengefügt werden. Und schließlich soll daraus die zum Programmieren des PIC notwendige `.HEX`-Datei erzeugt werden. Dazu geht man wie folgt vor:

MPLAB starten -> Project -> Project Wizard...



Bild 4.2.

Taste „Weiter >“



Bild 4.3.

PIC auswählen (hier den PIC16F877)

Taste „Weiter >“

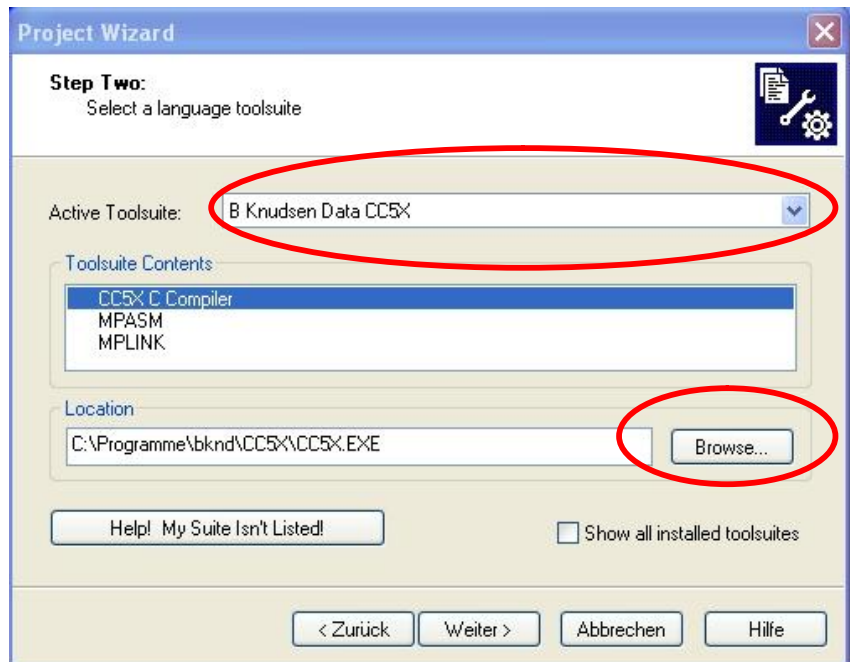


Bild 4.4.

Active Toolsuite: „*B Knudsen Data CC5X*“ auswählen
Taste „*Weiter >*“

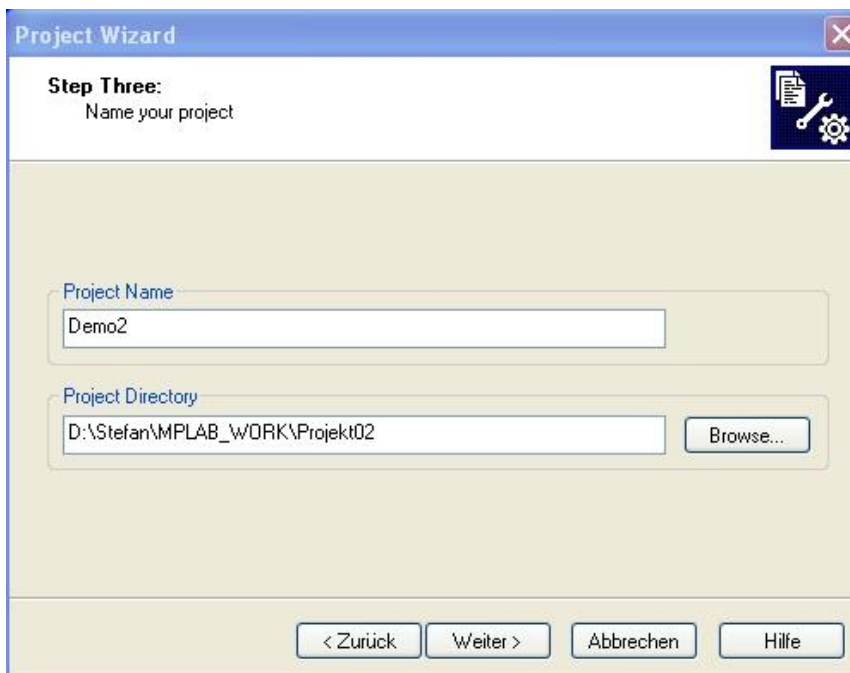


Bild 4.5.

Den Projektordner auswählen, wo dieses Beispiel-Projekt abgelegt werden soll. Dieser Ordner muss schon existieren!
Projektname eingeben (hier. *Demo2*). Anmerkung: der Projektname muss **nicht** identisch mit dem Ordernamen sein!

Taste „*Weiter >*“

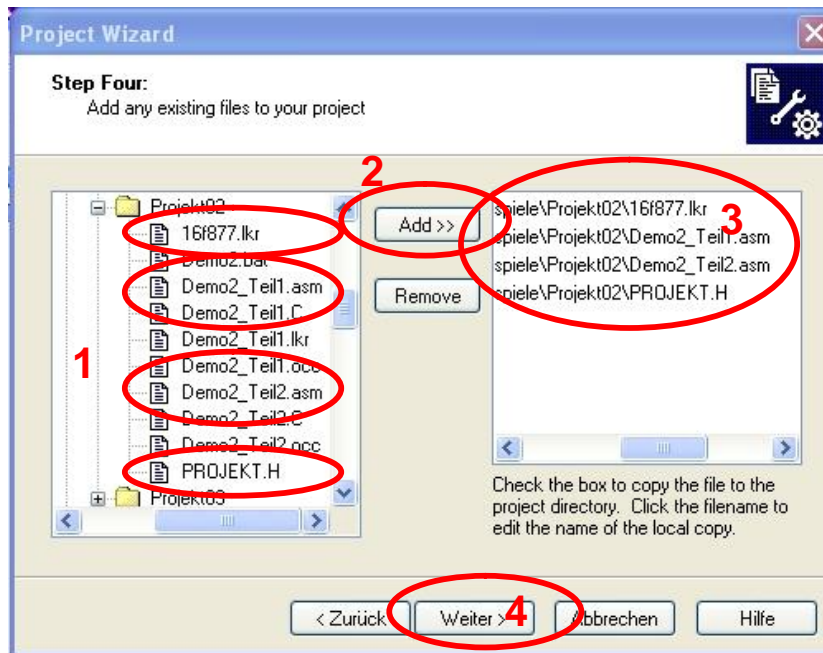


Bild 4.6

Zunächst in der rechten Spalte die Linker-Datei 16f877.lkr auswählen und die Taste „Add >>“ anklicken. Die ausgewählte Datei erscheint nun in die rechte Spalte. Diesen Vorgang für alle Assembler-Dateien (hier also Demo2_Teil1.asm und Demo2_Teil2.asm) und die Header-Datei PROJEKT.H wiederholen.

Taste „Weiter >“

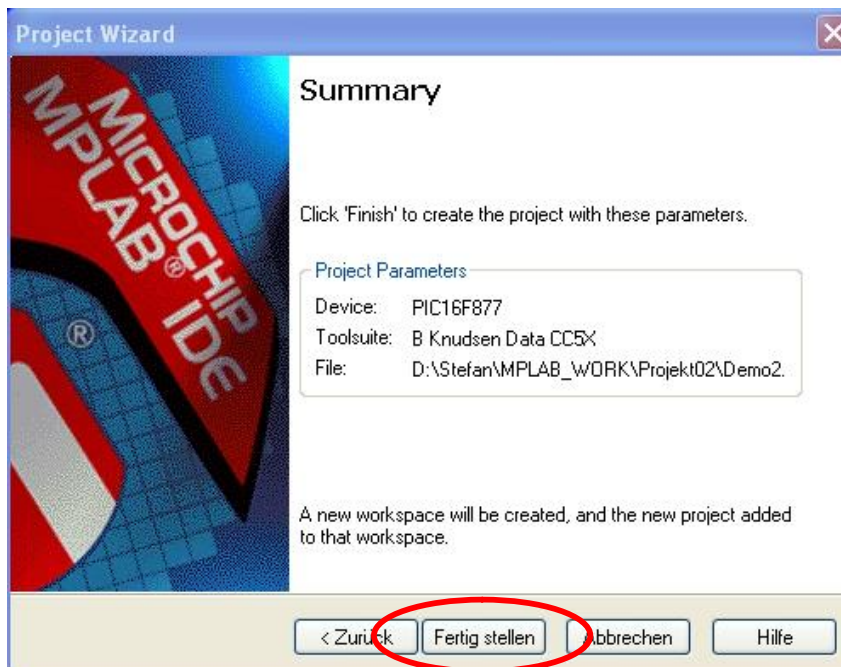


Bild 4.7.

Taste „Fertig stellen“

PIC-Programmierung in C (mit CC5X)

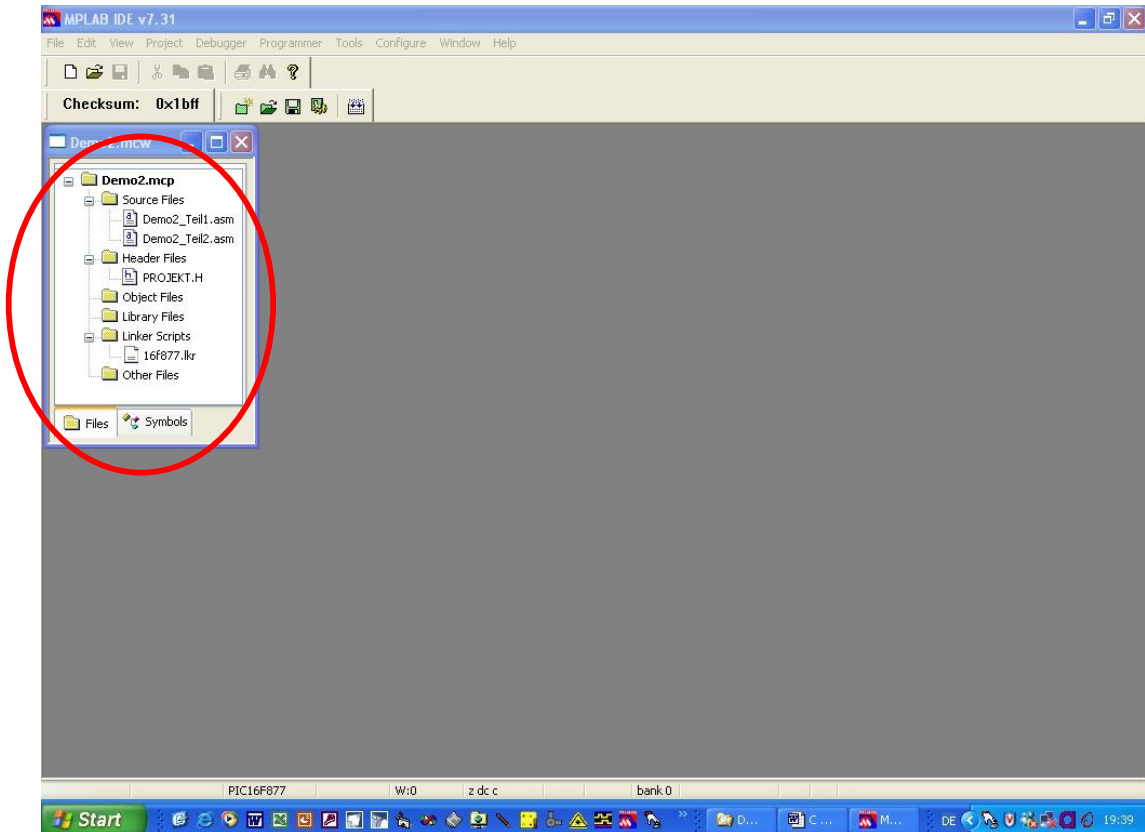


Bild 4.8.

Project -> Build

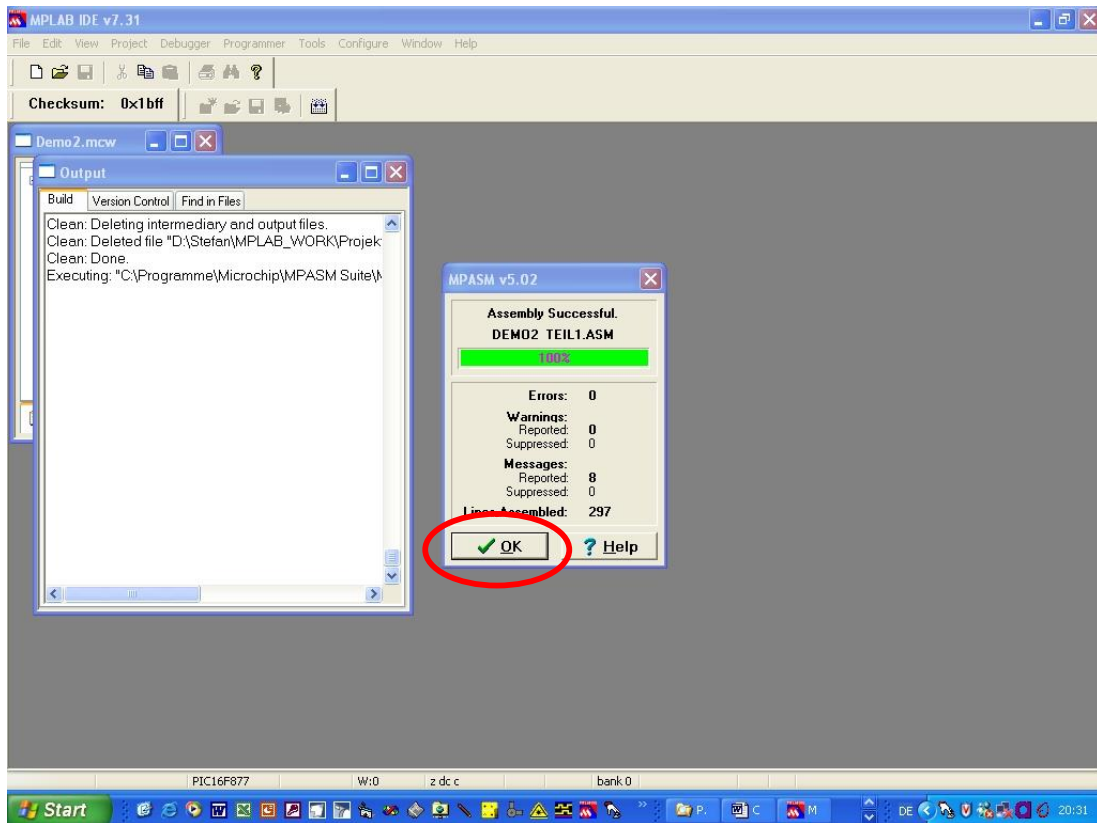


Bild 4.9.

Für jede Assembler-Datei sollte nun nacheinander die Box „*Assembly Successful*“ erscheinen. Diese muss jedes Mal mit der *OK*-Taste bestätigt werden. Wichtig ist die Meldung „*BUILD SUCCEEDED*“ am Ende der Assemblierung (Bild 4.10.).

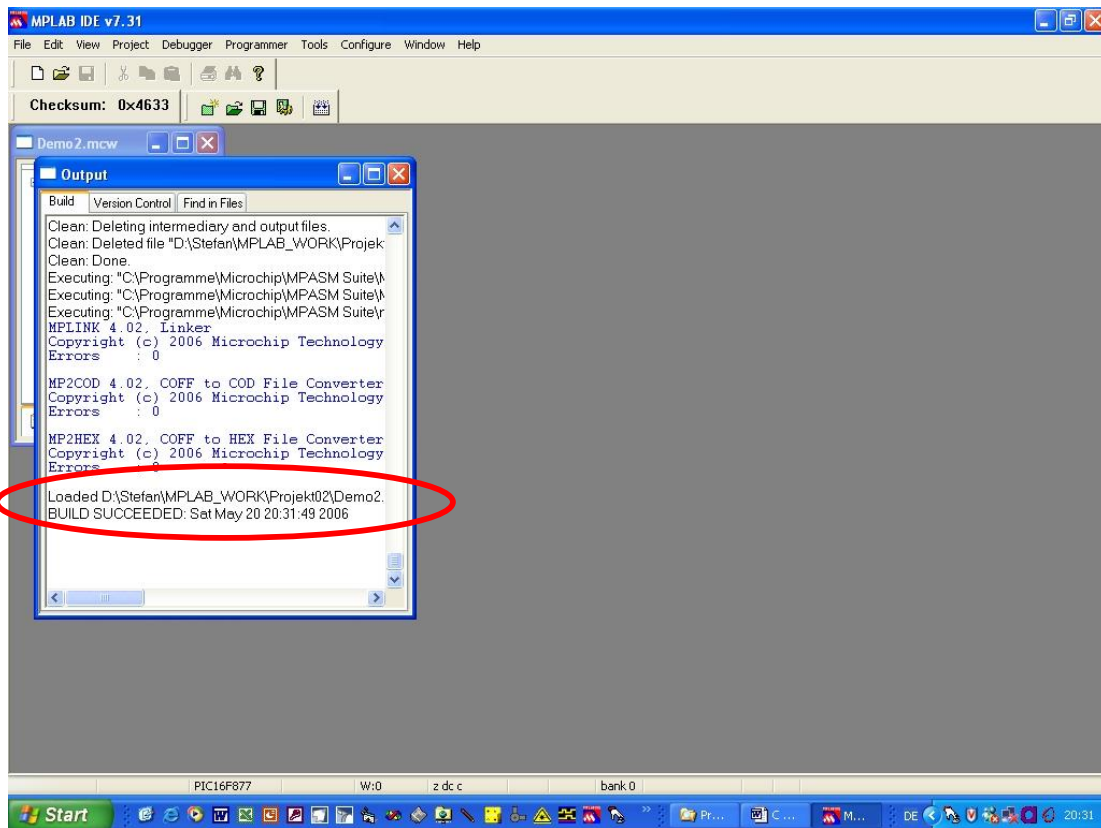


Bild 4.10.

5. Besonderheiten beim Programmieren mit mehreren Modulen

5.1. Externe Register

Wird eine Variable (= Register) in mehreren Unterprogrammen verwendet, so muss diese Variable als externes Register definiert werden.

Dabei gilt:

- 1) Dieses Register darf nur in **einer** C-Datei (wo dieses Register verwendet wird) definiert werden. Zum Beispiel

```
uns16    sollwert;
char     zahl1;
```

- 2) Zusätzlich muss diese externe Variable auch noch in „PROJEKT.H“ als externe Variable definiert werden. Zum Beispiel:

```
/* Externe Register */
extern bank0 uns16 sollwert;
extern bank0 char zahl1;
```

Wichtig ist hier das Schlüsselwort `extern` und die Angabe `bank0` gibt an in welcher Registerbank (bank0 bis ... je nach PIC unterschiedlich) diese Variable gesichert wird.

5.2. Unterprogramme

Befindet sich aufgerufene Unterprogramm **nicht** in derselben Datei wie das aufrufende Unterprogramm oder aufrufende Hauptprogramm so gelten folgende Einschränkungen:

- 1) Es ist **nur ein** Übergabeparameter möglich
- 2) Dieser eine Übergabeparameter kann nur **8bit breit** sein
- 3) Strukturen können **nicht** übergeben werden
- 4) Für einen eventuellen Rückgabeparameter gelten die selben Einschränkungen

Es sind daher nur möglich:

- `void Unterprogramm (void)`
- `void Unterprogramm (char Parameter)`
- `char Unterprogramm (void)`
- `char Unterprogramm (char Parameter)`

wobei, anstelle von `char` auch `unsigned char`, `int8` oder `uns8` verwendet werden kann.

Alle anderen sind so nicht möglich
z.B.:

- `void Unterprogramm (char Parameter1, char Parameter2)`
- `void Unterprogramm (int16 Parameter)`
- `uns16 Unterprogramm (uns16 Parameter)`
- usw.

Ausweg:

Externe Parameter (Variablen) verwenden!

Bei mir lauten diese z.B.

- `tmp_uebergabe_uns16`, `tmp_uebergabe_uns16_1` für 16bit vorzeichenlose Zahlen
- `tmp_uebergabe_int16`, `tmp_uebergabe_int16_1` für 16bit Zahlen mit Vorzeichen

Wichtig:

Dabei müssen natürlich die schon gemachten Hinweise für externe Register (Abschnitt 5.1) beachtet werden!

Weiters:

Zu jedem Unterprogramm muss es auch einen Funktionsprototyp geben. Dieser befindet sich bei mir immer in der Datei „PROJEKT.H“. Für die Funktionsprototypen gilt:

```
extern page0 void Unterprogramm1(void);
extern page0 char Unterprogramm2(char parameter);
```

Wichtig ist hier das Schlüsselwort `extern` und die Angabe `page0` gibt den Programmspeicherbereich (`page0` bis ... je nach PIC unterschiedlich) an.

Anmerkung:

Das Hauptprogramm und die Interrupt-Routine benötigen keinen Funktionsprototyp!

5.3. Interrupt

Für Interrupts gilt:

1) Datei „int16CXX.h“ einbinden (z.B. am Beginn der Datei, wo sich die ISR befindet)

```

/***** Include-Dateien *****/
#include "PROJEKT.H"
#include <int16CXX.H>           // Ist fuer die Interrupts notwendig
    
```

2) Die Interrupt-Routine (z.B.)

```

/***** ISR - Timer0 *****/
/*****
/* Interrupt Service Routine (ISR): */
/*****
#pragma origin 4

interrupt ISR(void)           // Interruptroutine
{
    int_save_registers        // W, STATUS (und PCLATH) retten

    // Beginn der eigentlichen ISR-Routine

    int_restore_registers    // W, STATUS (und PCLATH) Wiederherstellen
}
    
```

Anmerkung: ISR ist dabei ein beliebiger Name!

3) .LKR-File des verwendeten PIC anpassen und im Projektordner speichern (Hier: für den PIC16F877, diese befindet sich z.B. unter C:\Programme\Microchip\MPASM Suite\LKR\):

Die rot markierte Stelle muss entfernt werden, und die blau markierten müssen hinzugefügt werden.

```

// Sample linker command file for 16F877
// $Id: 16f877.lkr,v 1.4.16.1 2005/11/30 15:15:29 curtiss Exp $

LIBPATH .

CODEPAGE    NAME=vectors    START=0x0      END=0x3      PROTECTED
//CODEPAGE  NAME=page0      START=0x5      END=0x7FF
INCLUDE Demo2_Teil1.lkr
CODEPAGE    NAME=page1      START=0x800    END=0xFFFF
CODEPAGE    NAME=page2      START=0x1000   END=0x17FF
CODEPAGE    NAME=page3      START=0x1800   END=0x1FFF
CODEPAGE    NAME=.idlocs    START=0x2000   END=0x2003    PROTECTED
CODEPAGE    NAME=.config    START=0x2007   END=0x2007    PROTECTED
CODEPAGE    NAME=eedata     START=0x2100   END=0x21FF    PROTECTED

DATABANK    NAME=sfr0       START=0x0      END=0x1F     PROTECTED
DATABANK    NAME=sfr1       START=0x80     END=0x9F     PROTECTED
DATABANK    NAME=sfr2       START=0x100    END=0x10F    PROTECTED
DATABANK    NAME=sfr3       START=0x180    END=0x18F    PROTECTED

DATABANK    NAME=gpr0       START=0x20     END=0x6F
DATABANK    NAME=gpr1       START=0xA0     END=0xEF
    
```

Achtung: Dies muss bei jedem Projekt angepasst werden! (Hier: Demo2_Teil1)

PIC-Programmierung in C (mit CC5X)

```
DATABANK NAME=gpr2 START=0x110 END=0x16F
DATABANK NAME=gpr3 START=0x190 END=0x1EF

SHAREBANK NAME=gprnobnk START=0x70 END=0x7F
SHAREBANK NAME=gprnobnk START=0xF0 END=0xFF
SHAREBANK NAME=gprnobnk START=0x170 END=0x17F
SHAREBANK NAME=gprnobnk START=0x1F0 END=0x1FF

SECTION NAME=STARTUP ROM=vectors // Reset vector
SECTION NAME=ISERVER ROM=intserv // Interrupt routine
SECTION NAME=PROG1 ROM=page0 // ROM code space - page0
SECTION NAME=PROG2 ROM=page1 // ROM code space - page1
SECTION NAME=PROG3 ROM=page2 // ROM code space - page2
SECTION NAME=PROG4 ROM=page3 // ROM code space - page3
SECTION NAME=IDLCS ROM=.idlocs // ID locations
SECTION NAME=CONFIG ROM=.config
SECTION NAME=DEEPROM ROM=eedata // Data EEPROM

SECTION NAME=SHRAM RAM=gprnobnk
SECTION NAME=BANK0 RAM=gpr0
SECTION NAME=BANK1 RAM=gpr1
SECTION NAME=BANK2 RAM=gpr2
SECTION NAME=BANK3 RAM=gpr3
```

6. Allgemeine Erkenntnisse zu MPLAB

Ändern des PIC-Mikrocontrollers während der Projektentwicklung (z.B. Vom PIC16F874 auf den PIC16F877 wegen des größeren Programmspeichers)

- Problem: Im .LST-File wird der Mikrocontroller-Typ nicht geändert, folglich auch eine falsche Kompilierung!
- Lösung des Problems: Alle Projektdateien mit Ausnahme der Quellcodedateien (ASM, C, H, INC usw.) löschen und ein neues Projekt mit „*Project à Project Wizard*“ erstellen. Die Quellcodes (ASM, C, H, INC usw.) können bzw. müssen natürlich wieder eingefügt werden.

7. Quellen

- CC5X Version 3.2 User's Manual
- www.cc5x.de
- Buch „Das PICmicro-Hochsprachenbuch“ (ISBN: 3-7723-4264-7, Autoren: Anne König / Manfred König)