

Mikrocontrollerprogrammierung in C

Erste Schritte



MCB32 - Embedded Programmierung Grundlagen

Version: 1.1436a

1 Inhaltsverzeichnis

1	Inhaltsverzeichnis	2
3	Die Hardware	4
3.1	ARM Architektur	4
3.3	Das erste Programm	5
3.4	Aufgaben INOUT	5
3.5	Übersichtstabelle Bit - Manipulationen	6
3.6	Zusammenfassung Bit-Operatoren	7
3.6.1	Kombinierte Zuweisungen zur Erinnerung	7
3.7	Beispiele rund um die Bitmanipulation	8
4	Aufgaben Bitman: Eingabe Port P0, Ausgabe Port P1	9
4.1	Bitman1: Wechselblinker High/Low-Nibble	9
4.2	Bitman2: Einzelbit-Blinker	9
4.3	Bitman3: Einzelbit-Blinker mit Funktion Port1	9
4.4	Bitman4: Logikschaltung	9
4.5	Bitman5: Logikschaltung, Lösung mit bit-Zwischenvariablen	9
4.6	Bitman6: 2-Bit-Komparator	10
4.7	Bitman7: D-Flipflop	10
4.8	Bitman8: Lauflicht Links-Rechts	10
4.9	Bitman9: Lauflicht Start-Stop, Links-Rechts, Speed	10
4.10	Datentypen, Operationen in 'C' für Kontroller MCB32	11
4.10.1	Datentypen	11
4.10.2	Operationen und Portzuweisungen	11
5	Verzögerungen und Laufzeiten im µC-Board MCB32	12
6	Einbit- Definitionen, -Variablen und -Operationen	13
7	Programmstrukturen in 'C' für Kontroller	15
7.1	Define Statement und "Header file" für MCB32	16
7.1.1	#define	16
7.1.2	Konfigurationsfile ...stm32f10x_cl.h	16
8	Anhang Entprellen und Flankendetektion in C	17
8.1	Einleitung	17
8.1.1	Entprellen	17
8.1.2	Flankenerkennung	17
9	Anhang Aufbau Mikrocontrollerboards MCB32	18
9.1	Port 1	19
9.1.1	Schema / Beschaltung Port 1	19
9.1.2	Basis Struktur eines IO-Standard Pins	20
9.2	Übersicht GPIO auf 10pol Steckern	21
9.3	Port-Stecker-Pinbelegung Port E	21
9.4	Button 0 und Button1	22
9.5	Einsatz MCB32	22
10	Anhang: Entwicklungsumgebung KEIL aufsetzen	23
10.1	Neues Projekt einrichten mit Keil µVision 5	23
10.2	Weitere Projekt- und Prozessor (Target)-Einstellungen	24
11	Anhang: Umstellung von C51-Code auf ARM32-Code	28
11.1	Wichtig für das Funktionieren neuer Projekte	28
12	Anhang Touchscreen Kontrolle am µC-Board MCB32	29
13	Anhang Anschlüsse am µC-Board MCB32	30
14	Anhang Debuggen	31
14.1	Methodik des Debuggens (entlausen)	31
14.2	Verfolgen des Programmlaufs	32
14.3	Überprüfen der Daten	32
14.4	Programmtest durch Zielsystemdebugging	33
14.5	Anhang: Debuggen mit Jtag Interface ST Link V2	34
14.5.1	DEBUG FLASH DOWNLOAD	35

15	Anhang ARM M3	36
15.1	Blockschema	36
15.2	ARM Bit-Banding	37
16	Anhang Code Beispiele	38
17	Anhang: Referenzen	39
18	Anhang Literaturverzeichnis und Links	39
19	Anhang Wichtige Dokumente	39
20	Index	40
21	Anhang Notizen	41

3 Die Hardware

Hier eine vereinfachte Übersicht:

- CPU mit ARM 32-bit M3-Cortex
- 72MHz Maximum Freq.
- 256kByte Flash Speicher
- 64KByte SRAM.
- Vielfältiger Timer Block mit 32kHz Oszillator:
- Komplexe Timer Unit.
- 16Bit Motor PWM Timer
- 2 Watchdog Timers
- SysTick Timer: 24Bit Down-Counter.
- LowPower Sleep Modus.
- 2 x 12-bit D/A-Konverter.
- 2 x 12Bit, 1µs A/D Konverter (16 Kanäle) mit S/H.
- Temperatur Sensor.
- Bis zu 80 schnelle I/O-Ports. CRC Unit und
- 96-Bit UID.
- 2 * I2C Interface, 3 * SPI und 5 USARTs.
- CAN und Ethernet Interface sowie USB-Controller.



3.1 ARM Architektur

Die ARM-Architektur ist ein ursprünglich 1983 vom britischen Computerunternehmen Acorn entwickeltes 32-Bit-Mikroprozessor-Design, das seit 1990 von der aus Acorn ausgelagerten Firma ARM Limited weiterentwickelt wird. ARM steht für Advanced RISC Machines. Obwohl der Name außerhalb der IT-Fachwelt wenig bekannt ist, gehören Chips dieses Typs weltweit zu den meistverbreiteten Mikroprozessoren. (Wiki)

Das Unternehmen ARM Limited stellt keine eigenen Elektronikchips her, sondern vergibt unterschiedliche Lizenzen an Halbleiterhersteller. Fast alle derzeitigen Smartphones und Tablet-Computer benutzen einen oder mehrere lizenzierte ARM-Prozessoren, darunter das Apple iPhone und die meisten Geräte der Galaxy-Serie von Samsung.

Der beim MCB32 verwendete Prozessor Cortex-M3 ist ein Modell aus der Architektur-Familie ARMv7. Diese Architektur wurde ab 2004 eingeführt. Das M beim M3 steht für Mikrocontroller-Core.

Herzstück des Cortex-M3-Prozessors ist der M3-Kern mit dreistufiger Pipeline, basierend auf der *Harvard-Architektur*. Das heisst zwei getrennte Bussysteme (und zwei getrennte Speicher) zum Laden von Daten und Befehlen. Der Prozessor kann gleichzeitig sowohl Daten als auch Befehle lesen (bzw. Daten in den Speicher zurückschreiben). Mehr Information siehe auch Anhang.

•

Basis-E/A

3.3 Das erste Programm

```

/*-----
// Titel      : Ein-Ausgabe MCB32
// Datei      : P0P1Touch.c
// Erstellt   : 12.7.14 / rma
// Funktion   : Schalterstellungen am Eingabeport P0 lesen
//              und an Ausgabeport P1 (LED) ausgeben
//-----*/

#include <stm32f10x.h>           // Mikrokontrollertyp
#include "TouchP0P1.h"           // Library mit P0-, P1-Definition

int main(void)                  // Hauptprogramm
{
    InitTouchP0P1 ("1");         // Touchscreen aktiv (siehe Anhang 0 für Erklärung)
    while(1)                     // Endlosschleife
    {
        P1 = P0;                 // Portdurchschaltung. Die Eingänge P0 (Schalter)
                                // werde auf die Ausgänge P1 (LED) geschaltet
    }
}

```

Wichtig: Bei der Erstellung eines neuen Projektes im Schulbereich, also Vorbereitung für 8Bit-Programme „Elektroniker“ mit Port P0, P1 und Touchscreen ist folgendes zu beachten.

Kopieren Sie in jedes neue Projektverzeichnis diesen zwei Dateien:

- P0P1Touch.h
- P0P1Touch.lib

3.4 Aufgaben INOUT

InOut1 Geben Sie das obige Programm ein und testen Sie es.

InOut2 Erweitern Sie das Programm so, dass der eingelesene Wert vor der Ausgabe invertiert wird und wählen Sie dazu eine

- a) arithmetische Operation [8]
- b) logische Operation

Hinweis: Mit logischen Operationen können im Bereich eines Bytes einzelne Bits gesetzt (1), gelöscht (0), invertiert ($0 \leftrightarrow 1$) und auf ihren Zustand getestet werden.

Die logischen Operationen verknüpfen Bit für Bit mit OR, AND, EXOR.

InOut3 Mithilfe der folgenden Verzögerungsschleife (als Unterprogramm codiert)

```

void delay()
{
    long td;
    for (td =12000; td>0; td--);
}

```

sind folgende Programme zu realisieren (Ausgabe an P1) und zu testen

- a) Binärzähler 0 .. 255
- b) Lauflicht
- c) Blinker

3.4.1

3.5 Übersichtstabelle Bit - Manipulationen

Bit setzen: Für jedes zu setzende Bit muss die Maske eine 1 haben!

Wert	0	0	0	0	1	1	1	1
Maske	0	0	0	1	0	0	1	0

OR

Wert | Maske =

Bit löschen: Für jedes zu löschende Bit muss die Maske eine 0 haben!

Wert	0	0	0	0	1	1	1	1
Maske	0	0	0	1	0	0	1	0

AND

Wert & Maske =

Bit invertieren: Für jedes zu invertierende Bit muss die Maske eine 1 haben!

Wert	0	0	0	0	1	1	1	1
Maske	0	0	0	1	0	0	1	0

EXOR

Wert ^ Maske =

Bit testen: Für jedes zu testende Bit muss die Maske eine 1 haben!

Wert	0	0	0	0	1	1	1	1
Maske	0	0	0	0	0	0	1	0

AND

Wert & Maske =

Beispiele:

Blink = Blink	_____	; Lower-Nibble kippen
if ((Schalter	_____	; Wenn Schalter 3 ₍₀₎ ein
while ((Schalter	_____	; Solange Schalter 6 ₍₀₎ aus
Alarm = Alarm	_____	; Alarmbit 7 ₍₀₎ Ein
Control = Control	_____	; Heizungsbit 3 ₍₀₎ Aus

Wichtig: 3₍₀₎ die (0) deutet an dass die Zählung der Schalter bei 0 beginnt.
Also ist Schalter 3 der 4te Schalter. Der 3te Schalter belegt das 2³ Bit eines 8Bit Ports.

3.6 Zusammenfassung Bit-Operatoren

Operator	Bezeichnung	Beispiel
		unsigned int a = 0xBC; /* 188 = 1011'1100 */ unsigned int b = 0x89; /* 137 = 1000'1001 */ unsigned int c = 0;
&	AND ; log. UND	c = a & b; /* 88 = 1000'1000 */
&=	AND ; log. UND	a &= b wird zu a = a & b
 	OR ; log. ODER	c = a b /* 189 = 1011'1101 */
 =	OR ; log. ODER	a = b wird zu a = a b
^	XOR ; log. Exklusiv-ODER	c = a ^ b; /* 53 = 0011'0101 */
~	Einerkomplement	c = ~ a; /* 43 = 0100'0011 */
<<	Shift Left ; Linksschieben	c = a << 0x02; /* 240 = 1111'0000 */
>>	Shift Right ; Rechtsschieben	c = a >> 0x02; /* 47 = 0010'1111 */
<<=	Shift Left ; Linksschieben	a <<= 0x02; /* a = 240 = 1111'0000 */
>>=	Shift Right ; Rechtsschieben	a >>= 0x02; /* a = 47 = 0010'1111 */

3.6.1 Kombinierte Zuweisungen zur Erinnerung

Kombinierte Zuweisungen setzen sich aus einer Zuweisung (=) und einer anderen Operation zusammen.

Der Operand `a += b` wird zu `a = a + b` erweitert.

Es existieren folgende kombinierte Zuweisungen:

`+=` , `-=` , `*=` , `/=` , `%=` , `&=` , `|=` , `^=` , `<<=` , `>>=`

3.7 Beispiele rund um die Bitmanipulation

```
//-----  
// Titel   : Einbit Ein-Ausgabe und Verarbeitung MCB32  
// Datei   : 0405_bitman.c  
// Erstellt : xx.xx.20xx /Mal  
// Funktion : Zeigt das Lesen und Schreiben einzelner  
//            Portbits und die logischen Operationen  
//-----  
  
#include <stm32f10x.h>      // Mikrokontrollertyp  
#include "TouchP0P1.h"      // Library mit P0-, P1-Definition  
  
/* definiere IN-Output Bits für Ein und Ausgabe */  
#define E0      P0_0        // Eingabeport P0_x  
#define E1      P0_1        // Eingabeport P0_x  
#define A0      P1_0        // Ausgabeport P1_x  
#define A1      P1_1        //  
#define Alarm    P1_2        //  
char bTemp = 0;             // Beliebige Bitvariable via Char  
long ltvar =0;              //  
int main (void)             // Hauptprogramm  
{  
    InitTouchP0P1 ("1");     // Touch aktiv, Horizontal gedreht, LSB rechts  
    while(1)                 // Endlosschleife  
    {  
        A0 = 1;              // Konstante Ausgabe 0/1  
        Alarm = 1;           // Konstante Ausgabe 0/1  
        A1 = E1;             // Bitdurchschaltung  
        A1 = !E1;            // Invertieren  
        A1 = E0 & E1;         // bitweise AND-Verknüpfung  
        A1 = E0 | E1;         // bitweise OR-Verknüpfung  
        A1 = E0 ^ E1;         // bitweise EXOR-Verknüpfung  
        /*      Fragen zum beantworten      */  
        A0 = (E0==0);        // Z1  
        while (!E1);          // Z2  
        if ((P0 & 8)==0) A1 = 1; // Z3  
        P1 = P0 | 128;        // Z4  
        for (ltvar=12000; ltvar>0; ltvar--); // Z5  
    }  
}
```

Beschreiben Sie die Funktionen der Zeilen Z1 . . Z5!

Z1

Z2

Z3

Z4

Z5

4 Aufgaben Bitman: Eingabe Port P0, Ausgabe Port P1

4.1 Bitman1: Wechselblinker High/Low-Nibble

Ein Wechselblinker mit High/Low-Nibble Ansteuerung ist in 5 Varianten zu codieren.

4.2 Bitman2: Einzelbit-Blinker

Ein Blinklicht an Port1.3 ist zu realisieren ohne dass die übrigen Bit verändert werden. Lösung mit Maskierung.

4.3 Bitman3: Einzelbit-Blinker mit Funktion Port1

Grundfunktion gemäss Bitman2. Lösen Sie die Aufgabe mit einer Funktion `Port1(...)`, die wie folgt aufgerufen werden kann:

```
Port1(3,on);
```

```
delay(20000);
```

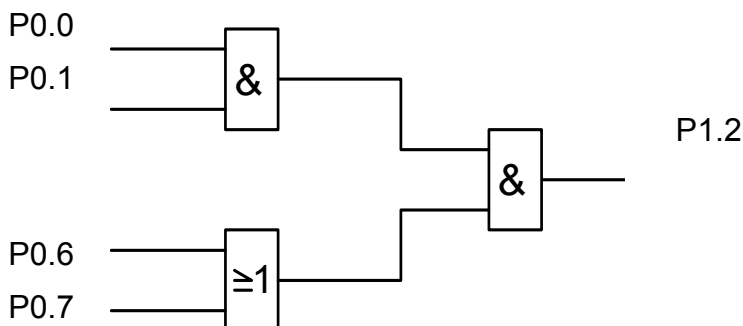
```
Port1(3,off);
```

```
delay(20000);
```

*Hinweis: Für **on** und **off** je eine Variable vom Typ **char (bit)** verwenden.*

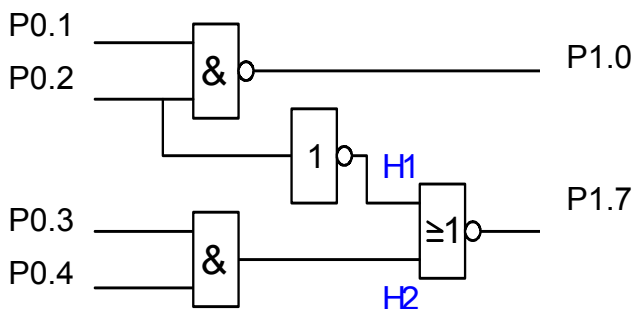
4.4 Bitman4: Logikschaltung

Die folgende logische Verknüpfung ist in einer Programmzeile zu lösen.



4.5 Bitman5: Logikschaltung, Lösung mit bit-Zwischenvariablen

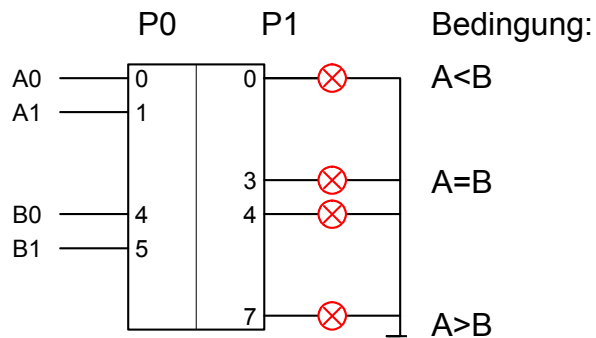
Die folgende logische Verknüpfung ist mit Bit-Zwischenvariablen zu lösen. Schreiben Sie zuerst die WT welche Sie dann zu Testzwecken für die Überprüfung der Gleichungen einsetzen. Zeigen Sie die Hilfsvariablen auf Port 2 und 3 an. (Debugging)



4.6 Bitman6: 2-Bit-Komparator

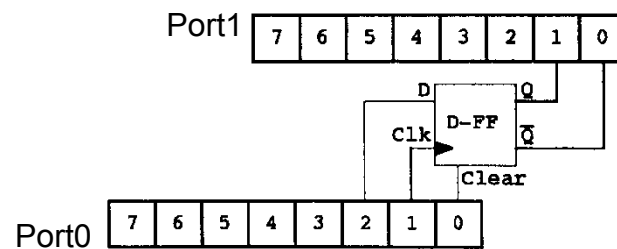
Funktion gemäss Schaltung.

Die übrigen Eingangsbits sind zu ignorieren!



4.7 Bitman7: D-Flipflop

Ein D-FF mit Clear-Eingang ist zu simulieren.

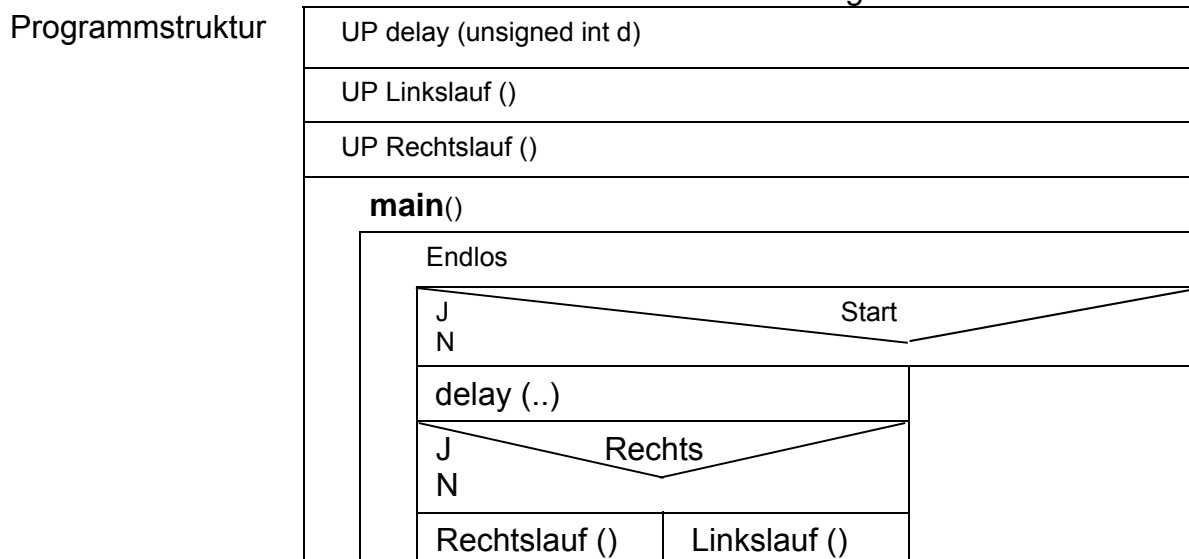


4.8 Bitman8: Lauflicht Links-Rechts

Funktion //Bit 0 Links-Rechts 0: Links 1: Rechts
//

4.9 Bitman9: Lauflicht Start-Stop, Links-Rechts, Speed

Funktion //Bit 0 Start-Stop 0: Stop 1: Start
// 1: Start
//Bit 1 Links-Rechts 0: Links 1: Rechts
// 1: Rechts
//Bit 2..7 Speed 0000'00 schnell
// 1111'11 langsam



4.10 Datentypen, Operationen in 'C' für Kontroller MCB32

4.10.1 Datentypen

Die Datentypen beschreiben wieviel Byte eine Zahl zum speichern benötigt und wie gross der Wertebereich ist. Jeder Datentyp kann mit **unsigned** ein nur positiver Werte-Bereich zugeordnet werden.

// Datentypen

```
char      Wahl, c;           // Byte: -128 .. 127
unsigned char Nr;           // Byte: 0 .. 255
int       i, k, m, n;       // 2 Byte: -32768 .. 32767
unsigned int u;             // 2 Byte: 0 .. 65536
long      L;                // 4 Byte: +- 2.14 Mia
long long sehrL;            // 8 Byte: +- 9.223.372.036.854.755.807
unsigned   long positiv;    // 4 Byte: 0 .. 4.29 Mia
unsigned long long spositiv; // 4 Byte: 0 .. 18446744073709551615
float      fzahltag;        // 4 Byte:  $1.2 \cdot 10^{-38}$  -  $3.4 \cdot 10^{+38}$  ; 6-stellig
double     dsgzahl;        // 8 Byte:  $2.3 \cdot 10^{-308}$  -  $1.7 \cdot 10^{+308}$  ; 15-stel.

char      *Text;            // 1..3 Byte: Zeiger;
char      bTemp = 0;        // 'Bit' Variablentyp char

// Input oder Output Bits als Variablen mit eindeutigen Namen
#define Einschalten P0_0
#define Ausschalten P0_1    // Port0 Bit 1 wird zum Einschalten genutzt

alle pointer *paufwert;     // 4 Byte
```

4.10.2 Operationen und Portzuweisungen

```
void main ()                // Hauptprogramm
{
    // Portzuweisungen
    P1 = 64;                // dezimal
    P1 = 0x40;              // hexadezimal
    P1 = P0;                // Port 0 --> Port 1

    // Arithmetische Operationen

    i = 3 - (8 + 12);
    f = 3 * (-2) / 4;

    // Bitweise Verknüpfung
    // „WERT operator MASKE“
    k = P0 & 0x0F;          // AND
    k = i | 128;            // OR
    k = m ^ 1;              // EXOR (invertiere einzelne Bits gemäss Maske)
    k = ~ n;                // NOT
    k = i << 2;              // 2 Bit links schieben
    k = i >> 4;              // 4 Bit rechts schieben

    // Logische Verknüpfung
    if ((P0==1) && (m!=0)); // AND
    while ((i<10) || (k>0)); // OR
    if ( ! (P0 == 4));       // NOT
}
```

5 Verzögerungen und Laufzeiten im µC-Board MCB32

1) InitTouchP0P1 ("0"); ohne Touchscreen

```
int main(void)                // Hauptprogramm
{
    long t;
    InitTouchP0P1("0");       // P0P1-TScreen OFF
    while(1)                  // Endlosschleife
    {
        P1_0=1;               // Pin0: ON
        for(t=120000;t>0;t--); // Delay ca.10ms
        P1_0=0;               // Pin0: OFF
        for(t=1200000;t>0;t--); // Delay ca.100ms
    }
}
```

Laufzeit in der while-Schleife:

0 ms

10 ms

0 ms

100 ms

Total 110 ms

ON

OFF

```
long t;                        // Immer long und auf 0 prüfend
for ( t = 12; t > 0; t--);     // 1.0µs
for ( t = 12000; t > 0; t--);  // 1.0ms
for ( t = 120000; t > 0; t--); // 10.0ms
for ( t = 1200000; t > 0; t--); // 100ms
for ( t = 12000000; t > 0; t--); // 1.00s
```

```
P1 = 0;                        // Durchschnittliche Operationszeit:
P1 = 1;                        // 0.17µs bis 0.5µs
```

2) InitTouchP0P1 ("xxx"); Touchscreenaufrischung erfordert zusätzliche Laufzeit

```
int main(void)                // Hauptprogramm
{
    long t;
    InitTouchP0P1("1");       // P0P1-TScreen ON
    while(1)                  // Endlosschleife
    {
        P1_0=1;               // Pin0: ON
        for(t=120000;t>0;t--); // Delay ca.10ms
        P1_0=0;               // Pin0: OFF
        for(t=1200000;t>0;t--); // Delay ca.100ms
    }
}
```

Laufzeit in der while-Schleife:

0 oder 10 ms

10 ms

0 oder 10 ms

100 ms

110 bis 130 ms

ON

OFF

Total

Umzeichnen einer In-/Out-Fläche	// 10ms pro Fläche
Somit kürzester Impuls an P1	// 0.25µs, evtl. + 10ms
Touchcheck per Interrupt	// alle 25ms + 3µs
	// bei Touch + 10ms (da 1 Fläche)

6 Einbit- Definitionen, -Variablen und –Operationen

Ref [3]

```

/*****
* Titel:      Einbit- Ein-/Ausgabe und Verarbeitung
* Datei:      EinBit.c / 14.1.14 / Version 1.0
* Ersteller:  R. Weber / BSU    / Keil ARM-Compiler / MAL / TBZ
* Funktion:   Zeigt das Lesen und Schreiben einzelner Portbits
*            und die logischen Operationen.
*****/
#include <stm32f10x.h>           // Mikrocontrollertyp
#include "TouchP0P1.h"          // P0-, P1-Definition

```

```

#define Start      P0_0          // Inputbits an
#define LichtOn    P0_1          // Port0 benennen
#define Alarm      P1_0          // Outputbits an
#define Motor      P1_1          // Port1 benennen

char    bTemp = 0 ;             // Beliebige 'Bit'-variable

```

Der 32 Bit µC hat kein Bitfeld mehr, der Compiler kennt damit **kein sbit und bit!**

```
void main(void)                // Hauptprogramm
{
    InitTouchP0P1 ("1");        // Touchscreen aktiv
    while(1)                    // Endlosschleife
    {

        P1_0 = 0;               // Konstante Ausgabe 0/1 an vordefinierte Portleitung
        Alarm = 1;              // Konstante Ausgabe 0/1 an umbenannte Portleitung
        Motor = Start;          // Bit- Durchschaltung
        bTemp = ! LichtOn;      // Invertiert in Bitvariable schreiben

        Alarm = P0_0 & P0_1;     // Einbit - AND - Verknüpfung
        Motor = Start | LichtOn; // Einbit - OR - Verknüpfung
        Alarm = Start ^ LichtOn; // Einbit - EXOR - Verknüpfung

        // AUFGABEN
        if (P0_7 ==1) P1_7=0;    // 1:a) Welche logische Verknüpfung entsteht hier?
        else P1_7=1;            // 1:b) Wie kann sie einfacher programmiert werden?

        while (!Start);         // 2a) Was für eine Wirkung hat diese Zeile?
                                   // 2b) Schreibe die Bedingung !Start ausgeschrieben!

        if ((P0 & 8)==0)         // 3a) Was für eine Wirkung hat diese Zeile?
            P1 = P1 | 128;       // 3b) Löse sie mit Einzelbits!

        P1 = P0 & 254;           // 4a) Was für eine Wirkung hat diese Zeile?
                                   // 4b) Löse sie mit Einzelbits!
    }
}
```

7 Programmstrukturen in 'C' für Controller

Die bekannten Kontrollstrukturen können auch bei einem C-programm für u-Kontroller (uP , MCU [1]) eingesetzt werden.

```
#include <stm32f10x.h>           // Mikrokontrollertyp, hier ARM STM32
#include "TouchP0P1.h"           // Library mit P0-, P1-Definition. Für MCB32
#define uchar unsigned char      // 0..255

//----- Funktionsdeklarationen (ev. nur Prototypen)
void Ausgabe (uchar w)
{
    // Schreibe hier einen Code
}

//----- Hauptprogramm
void main ()
{
    int i;                       // Variablen immer am Anfang
    uchar Wert;                  // der Funktion deklarieren!

    if (P0 == 0x1)               // Verzweigung einseitig
        P1++;
    else                          // und zweiseitig
        P1--;

    switch (P0)                  // Mehrfachverzweigung. In P0 steht der Wert
    {
        case 1 : P1 = 0x7;       // Wenn P0 = 1: Ausgabe P1 = 0000 0111 (0x07)
            break;               // und Abbruch
        case 2 : P1 = 0x80;      // Wenn P0 = 2: Ausgabe P1 = 1000 0000 (0x80)
            break;               // und Abbruch
        case 4 : P1 = 0xFF;
        default : P1 = 0;        // Wenn nichts zutrifft dann P1=0 und fertig
    }

    for (i=0; i<7; i++)          // Zaehlschleifen
    {
    }

    for (i=20000; i>0; i--);     // ca. 100ms Verzoeigerung. Relativ präzise

    while (P0)                   // Wiederholung mit Eintrittstest
    { }                           // Wenn Eintrittstest TRUE dann {} sonst weiter

    Do                           // Wiederholung mit mindestens einmal {}
    {                             //
    } while ((P0 & 4) != 0);      // mache weiter wenn

    while (1)                    // Endlosschleife
        Ausgabe(Wert);          // Funktionsaufruf mit Parameter
    }
```

7.1 Define Statement und “Header file” für MCB32

7.1.1 #define

Die Anweisung #define ermöglicht dem Programmierer via den Präprozessor Konstanten und Macros zu definieren.

Beispiel:

```
#define PI      3.14159265      // Pi wird als Konstante definiert (der Wert 3.14... ist eine
                                // Konstante)

#define uchar   unsigned char   // Makro um nicht immer unsigned char zu schreiben

#define uint    unsigned int    // Makro für unsigned int (nur positive Zahlen)
```

Das folgende Beispiel berechnet die Gewichtskraft einer Masse m. Der Präprozessor merkt dass keine Konstante vorhanden ist und folgert daraus, dass er dafür bei jedem Aufruf die Berechnung ausführen soll. (= Macro).

```
#define GEWICHTSKRAFT(masse)    9.81*masse

main()
{
    int masse=3;
    printf("%d \n", GEWICHTSKRAFT(masse));
}
```

7.1.2 Konfigurationsfilestm32f10x_cl.h

Bitte beachten Sie, dass dieses File je nach Anwendung und Stand der Software von Keil leicht anders aussehen kann.

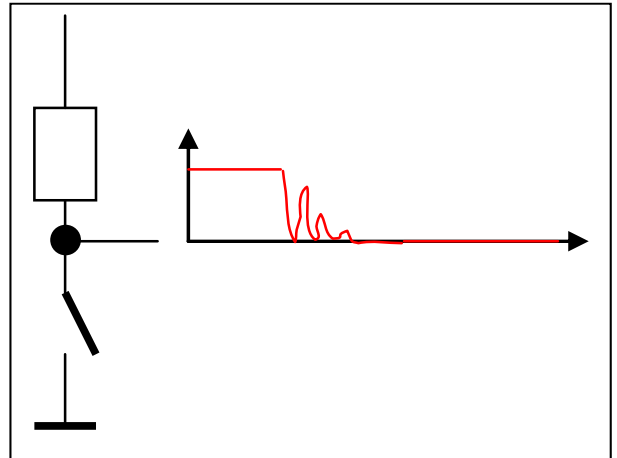
8 Anhang Entprellen und Flankendetektion in C

8.1 Einleitung

Taster können infolge der mechanischen Konstruktion prellen. Das heisst der Schalter schliesst aus sich der Elektronik nicht sofort. Bei schneller Verarbeitung der Signale ist dies ein störender Effekt. In der Elektronik stehen uns verschiedene Massnahmen zur Verfügung um das Problem zu entschärfen. In der Software müssen wir dies, sofern die Elektronik das Problem nicht löst, mit der Programmierung lösen.

Grundsätzlich stehen uns 2 Möglichkeiten zur Verfügung:

- Entprellen der Schalter mit Software
- Flankenerkennung



8.1.1 Entprellen

Der Schalter wird nicht nur einmal gelesen sondern nach einer geeigneten Verzögerungszeit noch einmal und dann mit dem letzten Wert verglichen.

Sofern die Werte gleich sind haben hat sich der Schalter nicht verändert oder ist unterdessen stabil. Wenn die Werte nicht gleich sind wurde geschaltet oder der Schalter ist noch am Prellen.

```
// Muster für mögliche Schalterentprellung
while(1) {
    schalter = P0.1;    // Lese Schalter 1
    delay_ms(2);        // warte 2ms
    if(schalter == P0.1) {
        // mache etwas, Wert hat sich nicht geändert
    }
}
```

8.1.2 Flankenerkennung

Der Schalter wird mit einem Zustand welcher vorher bestimmt wird verglichen. Wenn er von 0 auf geht wird geprüft ob dieser Vorgang wahr ist, was dann einer Änderung von 0Volt auf z.Bsp 5Volt entspricht.

Hier muss die Verarbeitungszeit und die Prellzeit **nicht** aufeinander abgeglichen werden, da der Schalter irgendeinmal sicher auf dem geprüften Wert P0.1==1 landet (beim Beispiel mit positiver Flanke).

```
// Muster für mögliche Flankenerkennung
pegel_vorher =0;    // Signalpegel vor der Betätigung
                  // des Schalters auf 0 (Volt)

while(1) {
    if(P0.1==1)&& (pegel_vorher ==0)
    { // positive Flanke => mache etwas

        pegel_vorher =1; // Setze Pegel auf den
                        // neuen Zustand des Schalters
    }

    if(P0.1==0)&& (pegel_vorher ==1)
    { // negative Flanke => mache etwas

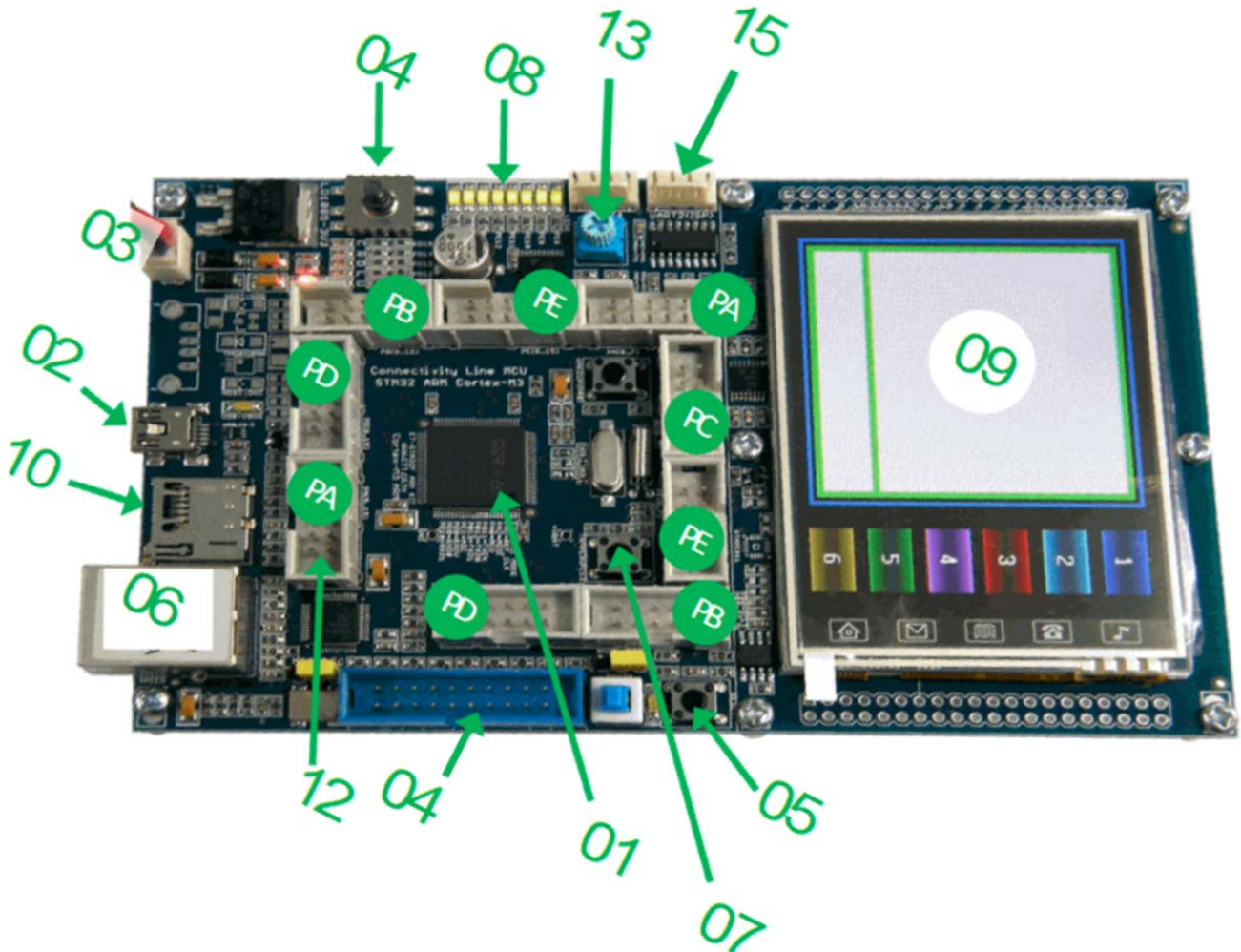
        pegel_vorher =0; // Setze Pegel auf den
                        // neuen Zustand des Schalters
    }
}
```

Die obigen Programme sind als Muster zu verstehen und müssen ja nach Hardware und Compiler angepasst werden.

9 Anhang Aufbau Mikrocontrollerboards MCB32

Das Mikrocomputerboard MCB32 enthält den Mikrocontroller STM32F107VC von ST aus der Familie der ARM MCUs.

Hardwareübersicht



01	ARM STM32 F107VC von ST.	08	8 LED (Port P1)
02	USB-Schnittstelle: Speisung	09	TFT LCD Color + Touch Screen 240x320 Pixel mit C-Bibliothek
03	Externe Speisung: Anschluss für Batterie oder Netzgerät	10	Mini SD-Card: max. 2G Speicher
04	JTAG-Schnittstelle: Download / Debugging	11	Uhrenquarz 32.768KHz für RTC (mit Batterie)
05	Reset: Für einen sauberen Neustart des Systems	12	72 GPIO Ports A-E (0..15). Auf 10pol Stecker verteilt
06	Ethernet Schnittstelle	13	Potentiometer für AD-Wandler
07	Taster 1: (mehr Taster via PA..PE) Verbunden mit Interrupt fähigen Pins	14	Temperatur- und Feuchtesensor.I2C
		15	UART-Schnittstelle: Kommunikation mit dem PC

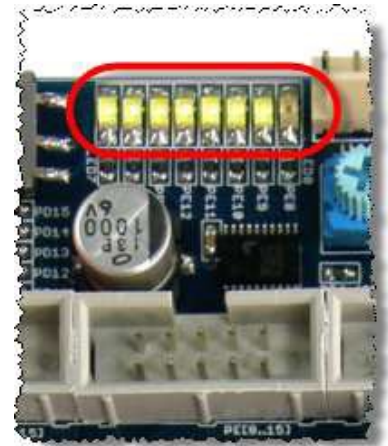
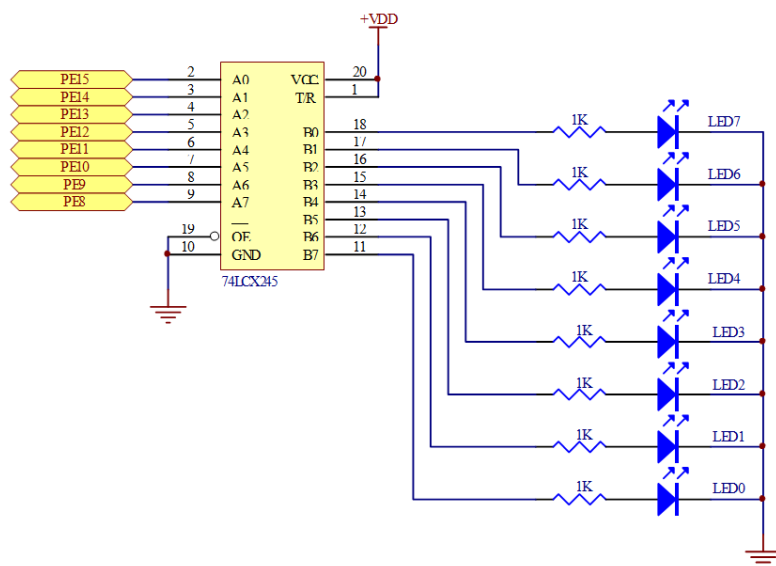
9.1 Port 1

Port 1	Digitale Ein- und Ausgabe. 8 On Board LEDS
Pin 1 ... Pin 8	Input High/Low +/- 50µA ; Output High/Low +/-1mA

9.1.1 Schema / Beschaltung Port 1

Leds welche via Port P1 resp. aus Sicht des ARM PE [8..15] angesteuert werden.

Ein „Logik 1“-Signal schaltet die LED ein. Eine „0“ schaltet sie ab.



Musterprogramm für Port E, Vorlage STM

Das Programm zeigt exemplarisch die hohe Abstraktion wenn keine vereinfachende Bibliotheken (P0P1Touch.lib) Solch ein Code muss, vor allem an Anfang mit dem Handbuch und den Block-schemas des Prozessors gelesen werden. Einfacher geht es mit der Lib: *P0P1Touch.lib* zum MCB32.

```
#include "stm32f10x.h"
int main(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    SystemInit();
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE, ENABLE);
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;

    // GPIO_InitStructure.GPIO_Pin = GPIO_Pin_x. Pins als GPIO initialisieren
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_10 | GPIO_Pin_11 |
                                   GPIO_Pin_12 | GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;

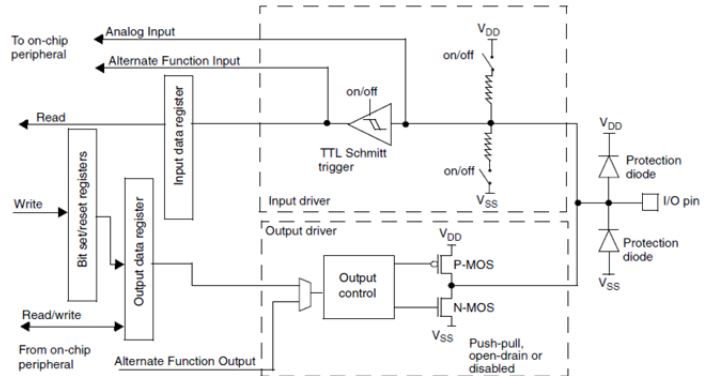
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;           // Speed setzen
    GPIO_Init(GPIOE, &GPIO_InitStructure);                     // Port E initialisieren gem. Vorberei-
teung

    GPIO_WriteBit(GPIOE, GPIO_Pin_8, Bit_SET); // setze Bit 8 und fertig
}
```

9.1.2 Basis Struktur eines IO-Standard Pins

Schaltungskonzept eines IO Pins

Die Pins sind auch für 5VOLT Anwendungen vorgesehen und haben eine entsprechende Schutzschaltung.

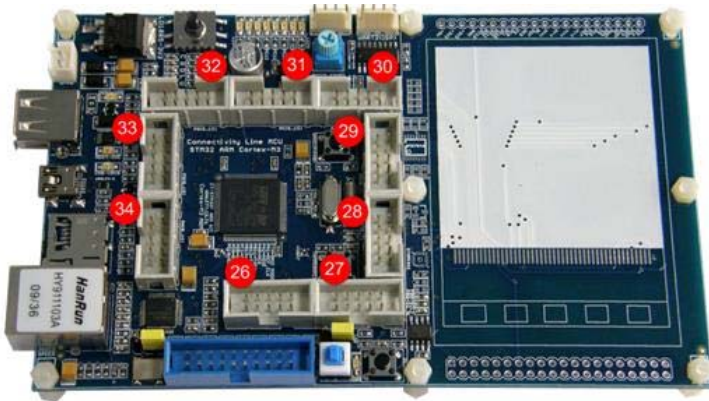


Port 1	Alternative Funktion
Pin 1	T2: Timer/Counter 2 Ext. Count/Inp.Clock
Pin 2	T2Ex: Timer/Counter 2 Reload/ Capture/Direct.Control
Pin 3	ECL: Ext. Clock for PCA
Pin 4	CEX0: Capture/Compare Ext.I/O for PCA module 0
Pin 5	CEX1: ... PCA module 1
Pin 6	CEX2: ... PCA module 2
Pin 7	CEX3: ... PCA module 3
Pin 8	CEX0: ... PCA module 4

Port 3	Alternative Funktion
Pin 1	RXD: Serial input port
Pin 2	TXD: Serial output port
Pin 3	Int0: External Interrupt 0
Pin 4	Int1: External Interrupt 0
Pin 5	T0: Timer 0 external Input
Pin 6	T1: Timer 1 external Input
Pin 7	WR: Ext memory write strobe
Pin 8	RD: Ext memory read strobe

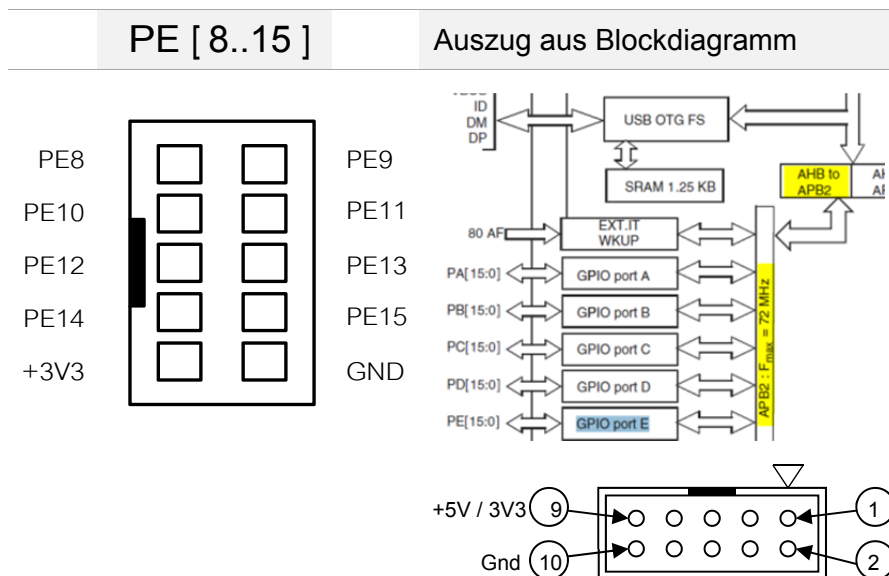
9.2 Übersicht GPIO auf 10pol Steckern

Übersicht über alle Port auf den 10poligen 3M Steckern



26	Stecker zu: GPIO PD[0..7].
27	Stecker zu: GPIO PB[0..7].
28	Stecker zu: GPIO PE[0..7].
29	Stecker zu: GPIO PC[0..7].
30	Stecker zu: GPIO PA[0..7].
31	Stecker zu: GPIO PE[8..15].
32	Stecker zu: GPIO PB[8..15].
33	Stecker zu: GPIO PD[8..15].
34	Stecker zu: GPIO PA[8..15].

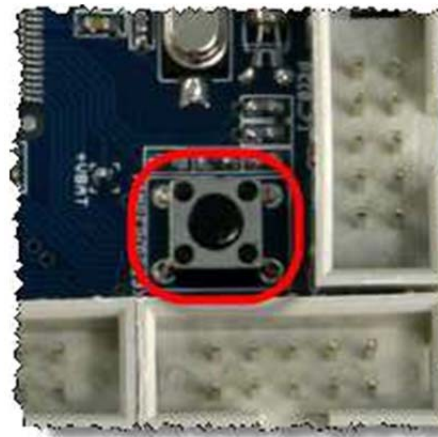
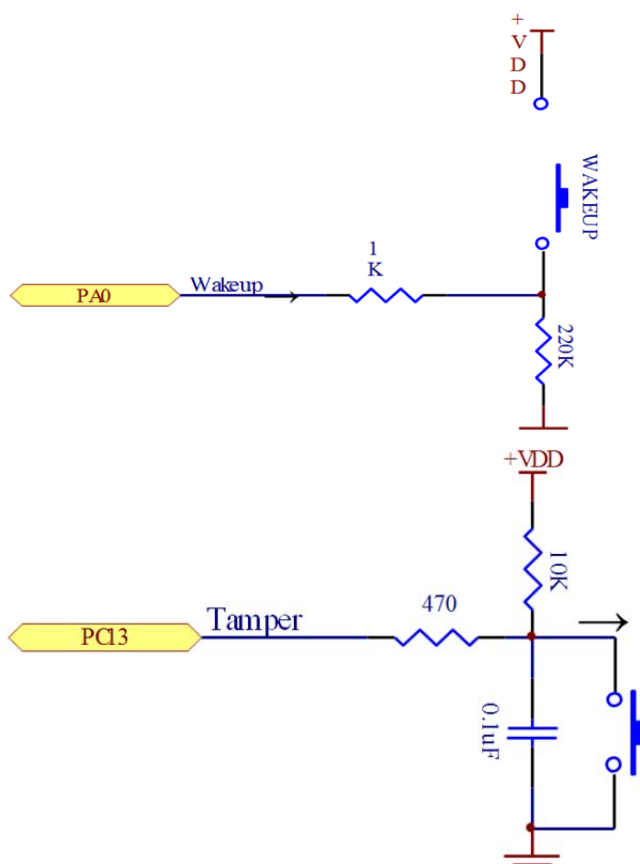
9.3 Port-Stecker-Pinbelegung Port E



9.4 Button 0 und Button1

Die beiden Schalter heissen im Schema „Switch Wakeup“ und „Switch Tamper“. Button 0 ist „Switch Tamper“ und entsprechend Button_1 „Switch Wakeup“. Sofern die Library, welche die Schalter bedient die Polarität nicht dreht haben die beiden Schalter entgegengesetzte Polarität „opposite Logic“.

Wenn Button_1 gedrückt ist wird Pin PA0_High, ansonsten Low. Button_0 heisst im Schema Tamper und geht auf PC13. Das heisst wenn Button_0 gedrückt ist wird der Pin PC13 LOW.

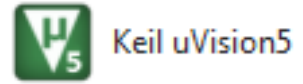


9.5 Einsatz MCB32

- Als Lernkit zum Erlernen der Arbeitsweise von Mikrocontrollern, speziell der ARM Familie.
- Als Evaluationskit zum Test neuer Hardware.
- Als Zielboard im Entwicklungssystem, um uC-Programme zu testen.
- Als autonom einsetzbare Prozesssteuerung in industriellen Anlagen.

10 Anhang: Entwicklungsumgebung KEIL aufsetzen

Wir arbeiten mit der IDE der Firma Keil. Damit können wir die Programme bis zu 32kB Programmcode schreiben. Das genügt für die Ausbildung.



IDE: Integrated Development Environment von <http://www2.keil.com/mdk5/install>

ARM: Advanced RISC Machines

10.1 Neues Projekt einrichten mit Keil µVision 5

1. Schritt

µVision 5 starten:

Neues Projekt erstellen:

Start / Programme / Elektronik / KeilArm µVision 5

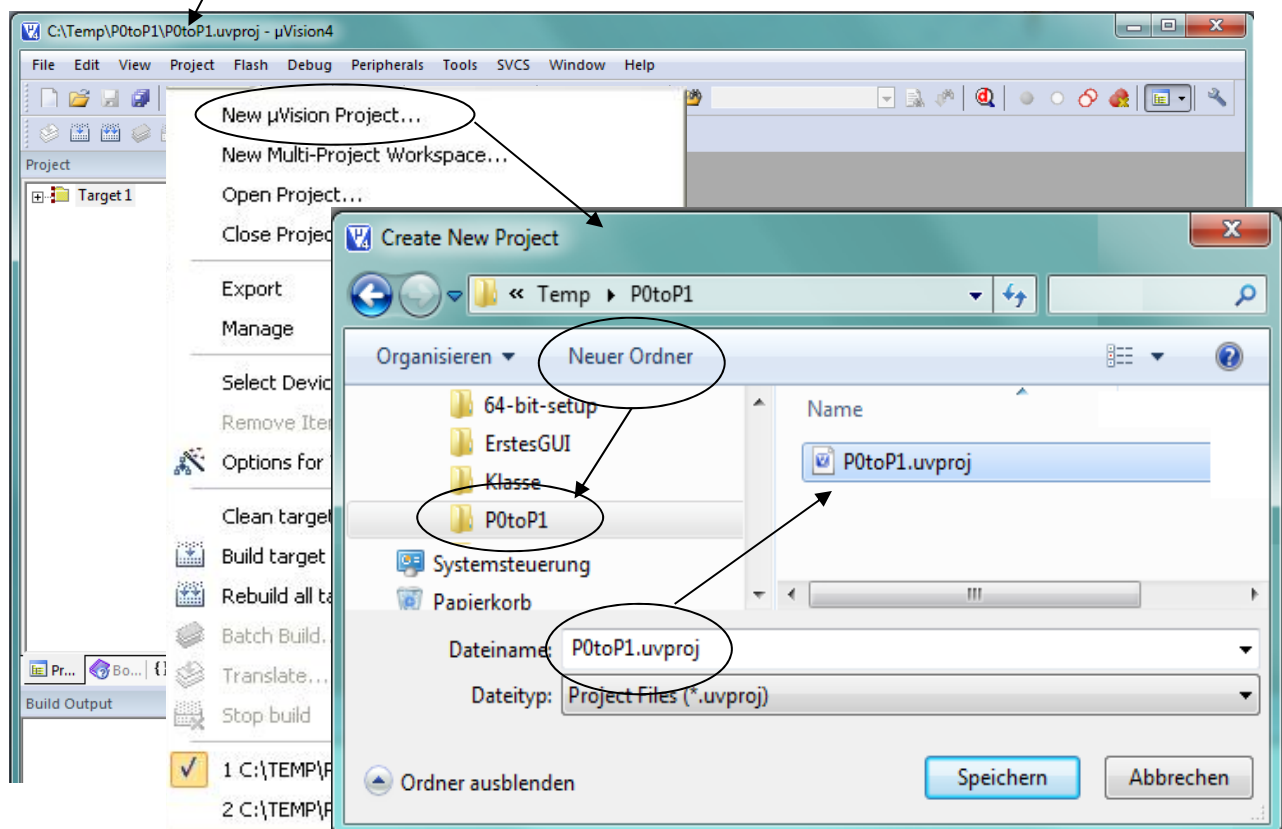
Project / New Project /

Für jedes Projekt

- einen neuen Ordner zB: P0ToP1

- dann Projektname zB: P0ToP1.uvproj

- Speichern



Weitere Menüpunkte der IDE welche es zu beachten gibt.

Altes Projekt öffnen: [Project](#) / Open Project

Projekt schliessen: [Project](#) / Close Project

µVision 4/5 verlassen: [File](#) / Exit

Wichtig:

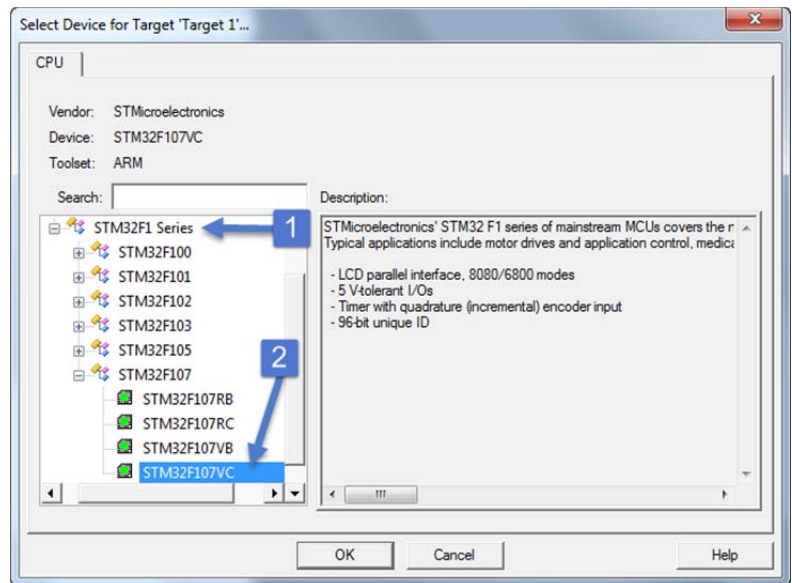
In jedem Projekt müssen die beiden Files: **Touch P0P1.lib** und **TouchP0P1.h** vorhanden sein. Kopieren Sie diese vom Server oder von einem bekannten Ort ins Projekt.

10.2 Weitere Projekt- und Prozessor (Target)-Einstellungen

2. Schritt: Target auswählen

Nun fragt die IDE nach dem Ziel-Controller:

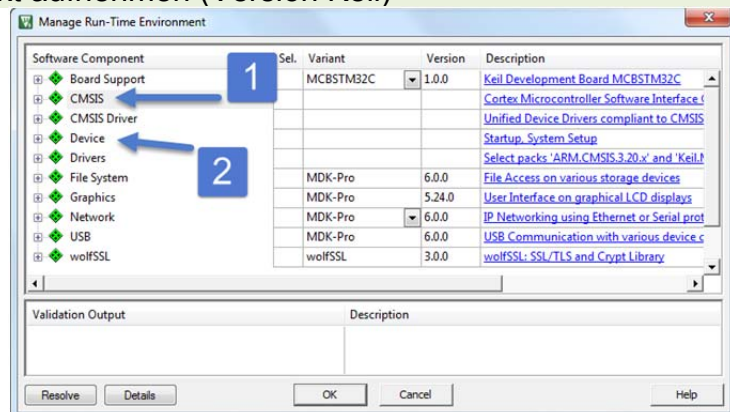
- Ziel-Controller wählen. In unserem Fall der **STM32F107VC**



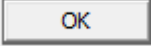
3. Schritt: Startup Code ins Projekt aufnehmen (Version Keil)

Nun benötigt die IDE mehr Informationen und auch „Files“ um nachher während der Kompilation alle Verknüpfungen herstellen zu können:

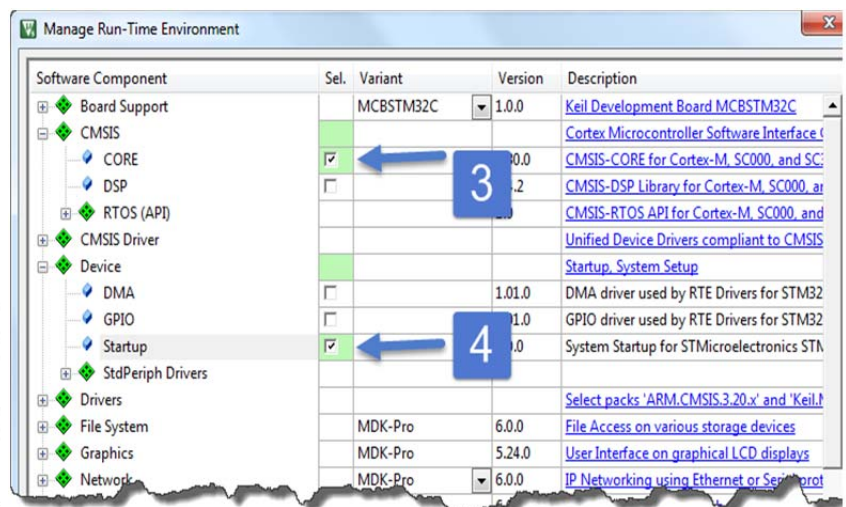
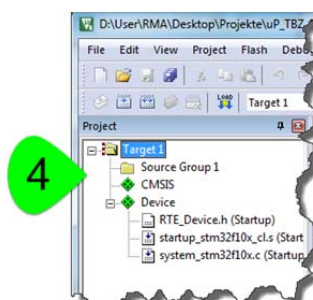
- CMSIS – Basics laden
- Device – Files laden



- CMSIS: Core auswählen
- Device – Startup auswählen

Und  klicken.

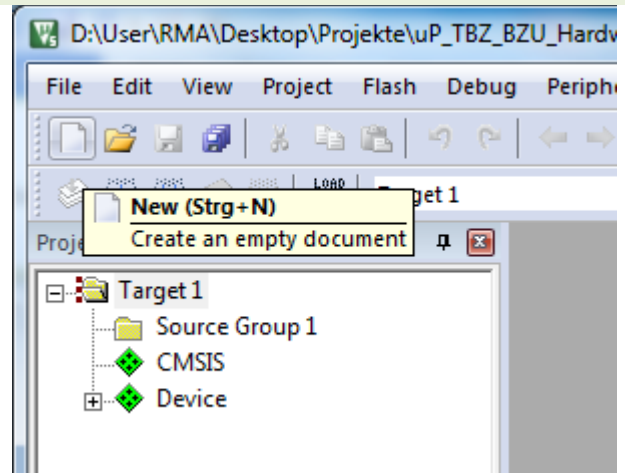
Die Projektumgebung sollte nun so aussehen:



Im Prinzip ist das Projekt bezüglich der grundlegenden Einstellungen bereit. Wir müssen nun die eigentlichen Programmfiles dazufügen.

4. Schritt: Programm Files aufnehmen.

- Klicke auf NEW (oder STRG-N) und speichere das File mit dem Namen P0toP1.c ab.



5. Schritt: Programm schreiben.

- Schreibe nun folgenden Code.
- Speicher alles ab.



Dieses File können wir mit anderen Files nun noch dem Projekt hinzufügen.

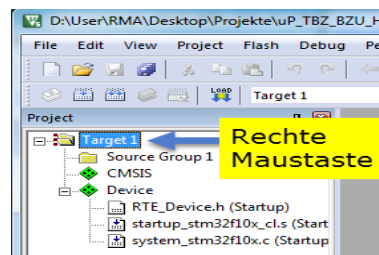
```

1 //-----
2 // MCB32 (BZU / TBZ)
3 // Autor: P0toP1.c / 2.1.14 / rw, mal / BZU,TBZ
4 // Thema: Schaltet am uC-Board MCB32 die Schalter an P0
5 // (GPIOC0..7) zu den LEDs an P1 (GPIOE8..15) durch. Mit Touchbedienung
6 //-----
7 #include <stm32f10x.h> // Mikrokontrollertyp
8 #include "TouchP0P1.h" // Library mit P0-, P1-Definition und Touch
9
10 int main(void) // Hauptprogramm
11 {
12     InitTouchP0P1("1"); // P0,P1 auf Touchscreen ON
13     while(1) // Endlosschleife
14     {
15         P1 = P0; // Portdurchschaltung
16     }
17 }

```

6. Schritt: Alle Programmfiles zum Projekt hinzufügen.

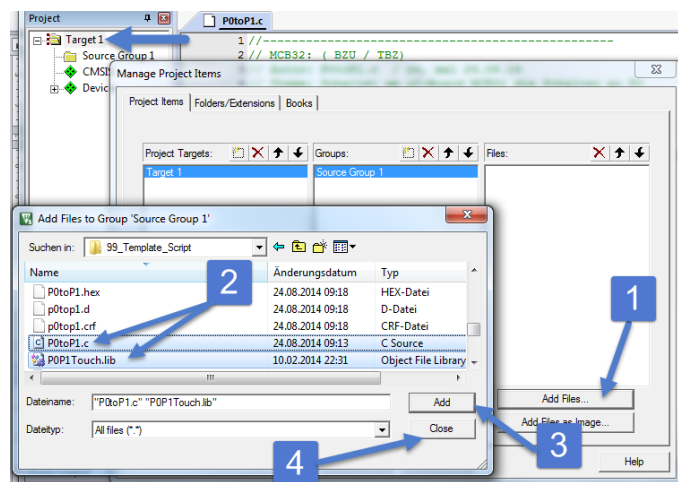
Mit der rechten Maustaste auf „Target1“ klicken.



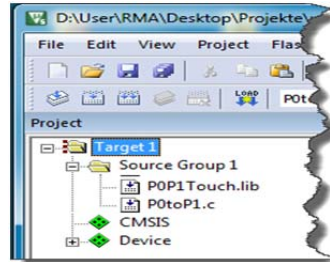
- Nun die Schritte 1-4 abarbeiten:

Das File: P0toP1.c und die Library Touch P0P1.lib werden dem Projekt hinzugefügt.

Wichtig: Die Files können mit CTRL-Click alle zusammen ausgewählt werden.



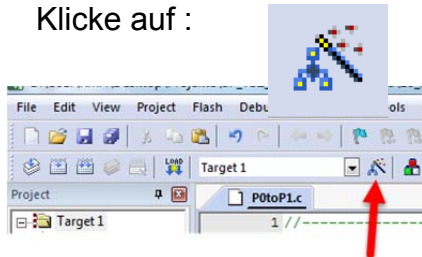
So sollte das Projekt nun aussehen:



7. Schritt: Setup vervollständigen (Debugger und JTAG)

Damit das Programm nach dem Kompilieren in den ARM geladen werden kann sind weitere Einstellungen nötig.

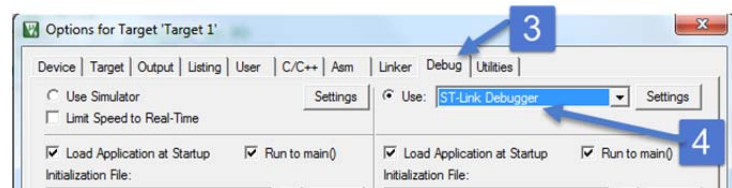
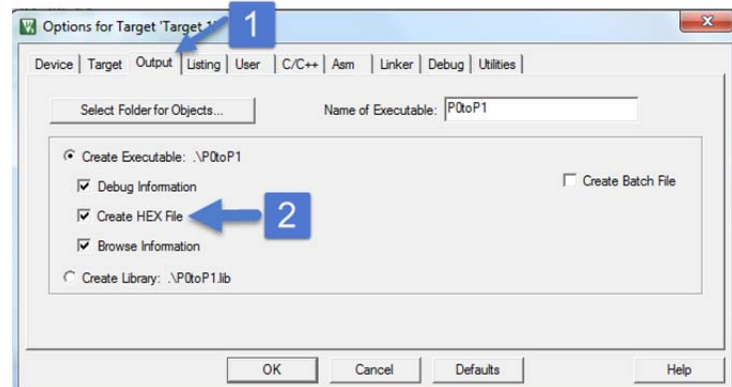
Klicke auf :



Damit werden die „Options for Target1“ ausgewählt.

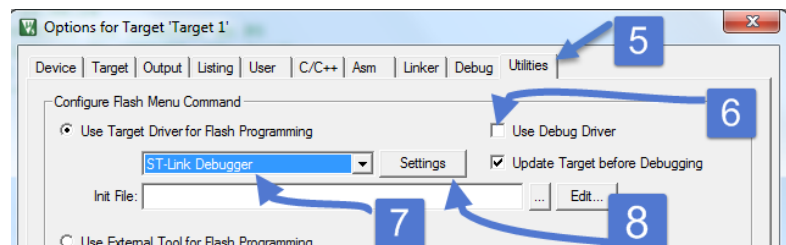
Wähle: „Debug“ und selektiere „STLink Debugger“ aus dem Menü aus.

Wähle: „Output“ und selektiere „Create HEX File“.



Wähle: „Utilities“, Klicke „Use Debug Driver“ **aus** und selektiere noch „STLink Debugger“.

Am Schluss [8] die Settings auswählen.



8. Schritt: Settings für ST-Link Download vornehmen

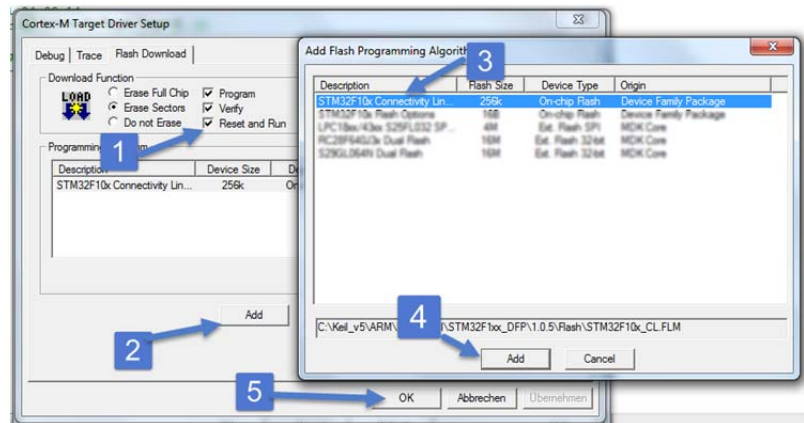
C – Programmieren mit uP-Board

Bei den Settings nun „Reset and Run“ auswählen.

Mit „Add“ [2] sagen wir dem Treiber welchen Programmieralgorithmus gewählt wird. Also den für den STM32F10x [3].

Mit [4] abschliessen und mit OK [5] das Menü „Driver Setup“ schliessen.

Das Fenster schliessen und fertig.

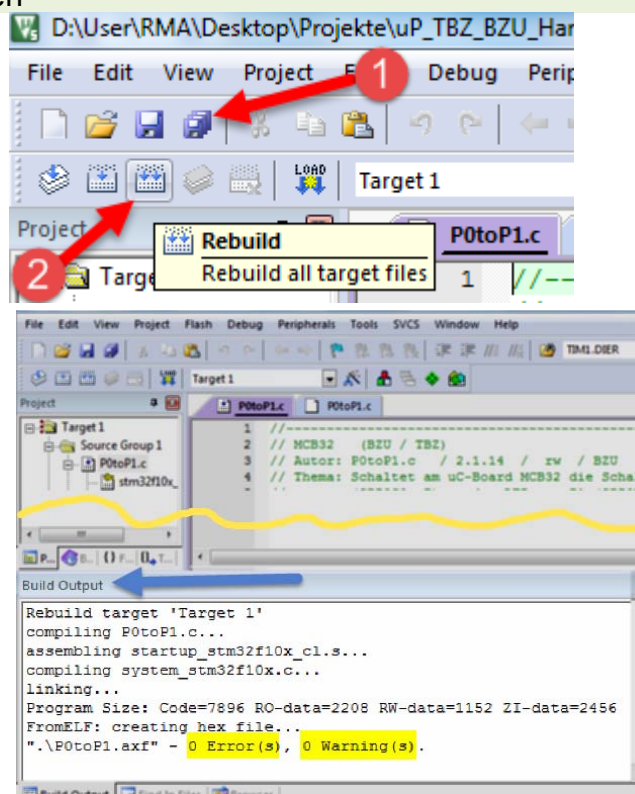


9. Schritt: Programm kompilieren

Sichere das Projekt [1] und kompiliere [2] mit dem Knopf „Rebuild all target files“

Im Fenster „Build Output“ sollte in etwa nebenstehende Meldung erscheinen.

Wenn keine Fehler vorkommen kann das erzeugte HEX-File in den Prozessor geladen werden.

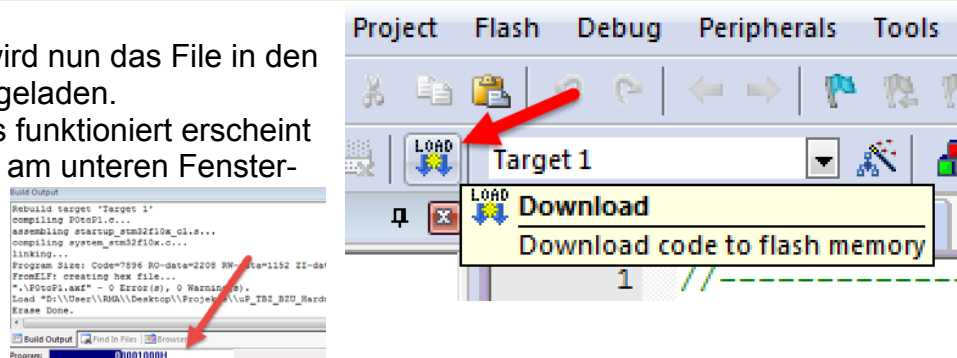


10. Schritt: Programm in den Prozessor laden.

Mit Load wird nun das File in den Kontroller geladen.

Wenn alles funktioniert erscheint ein Balken am unteren Fenster-rand.

Dieser zeigt den Download an.



11 Anhang: Umstellung von C51-Code auf ARM32-Code

```

/*****
* Titel:      Umstellung von C51-Code auf ARM32-Code
* Datei:      C51toARM32.c / 14.1.14 / Version 1.0
* Ersteller:   R. Weber (BSU); E. Malacarne (TBZ)
* Funktion:    Die wichtigsten Umstellungen sind in den Kommentaren dokumentiert
*****/

```

```

#include <stm32f10x.h>           // Mikrokontrollertyp
#include "TouchP0P1.h"          // P0-, P1-Definition

#define Start P0_0              // Input und Outputbits
#define Alarm P1_7              // an Ports benennen
char bTemp = 0;                 // 'Bit'-Variablentyp char
long t;                         // Zeitvariable

```

neue #includes

Keine sfr und
sbit mehr!

```

int main ( void )              // Hauptprog., ohne return bei Keil
{
    InitTouchP0P1 ("1");       // Touchscreen aktiv,
                                // horizontal gedreht
                                // LSB rechts

```

Main verlangt int

InitTouchP0P1 ("0"),
wenn nur P0,P1 und
ohne Touchscreen

```

while(1)                       // Endlos-schleife
{
    P1_0 = 0;                   // Bitverarbeitung wie bisher
    Alarm = 1;                 // Zuweisung, Invertierung,
    bTemp = !Start;            // &, &&, |, ||, ^, !, ==, !=

    while ( P1 < 100 )          // Byteverarbeitung wie bisher
        P1 += 2;               // Kurzformen wie bisher
    P1 = P0 & 0x0F;             // Maskierungen wie bisher

    for(t=120000; t>0; t--);    // Verzögerung 10ms
}

```

Verzögerung

vom Typ long mit
Wert 12 / µs

11.1 Wichtig für das Funktionieren neuer Projekte

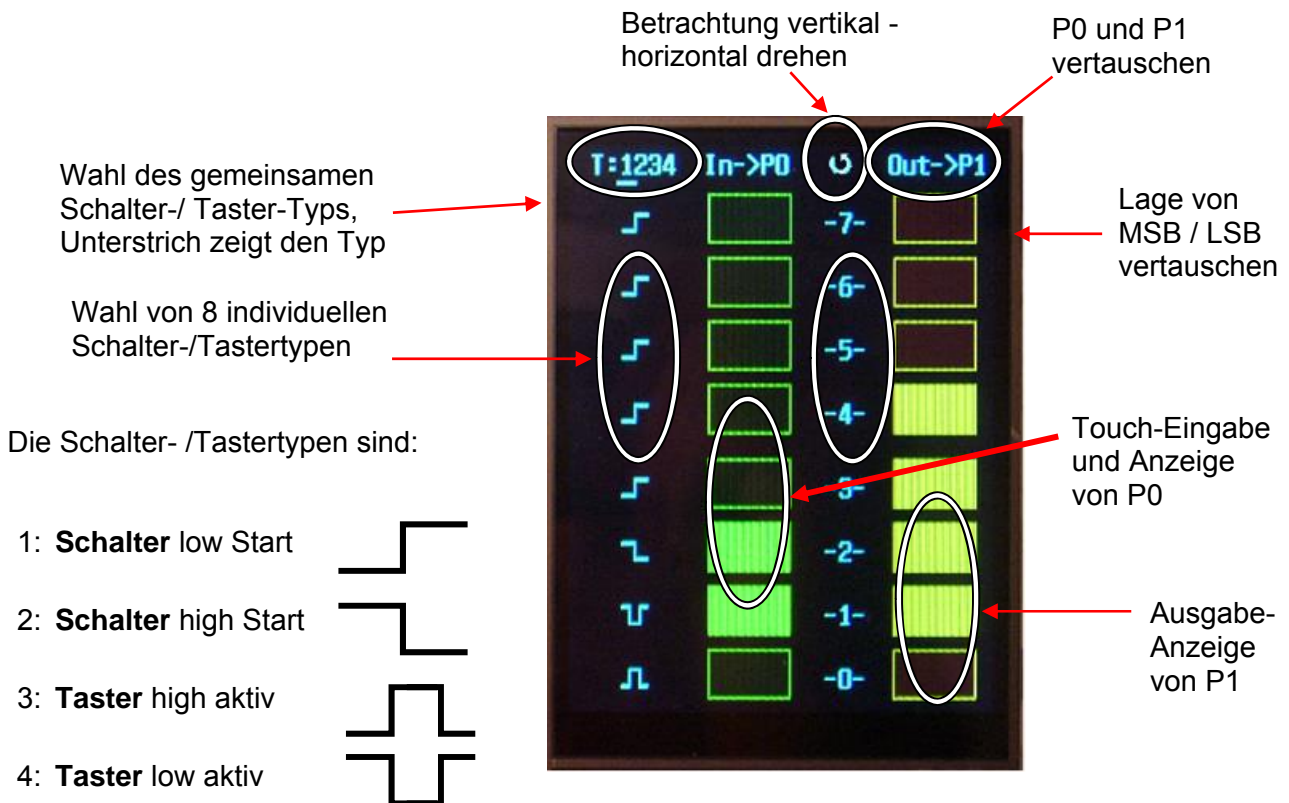
Wichtig: Bei der Erstellung eines neuen Projektes im Schulbereich, also Vorbereitung für 8Bit-Programme „Elektroniker“ mit Port P0, P1 und Touchscreen ist folgendes zu beachten.

Kopieren Sie in jedes neue Projektverzeichnis aus diesen zwei Dateien:

- P0P1Touch.h
- P0P1Touch.lib

12 Anhang Touchscreen Kontrolle am μ C-Board MCB32

Beschreibung der Touchscreen Oberfläche



Touchscreen Kontrolle aus dem Quellcode

Der Projekt-Ordner muss P0P1Touch.h und P0P1Touch.lib enthalten:

Im Projekt-Manager sind zum startupxx.s die Quelldatei.c und P0P1Touch.lib aufzunehmen.

```
/* Beschreibung der 8 Bit P0- und P1-Kontrolle über den Touchscreen des MCB32
*****/
#include <stm32f10x.h>           // Mikrokontrollertyp
#include "TouchP0P1.h"          // P0-, P1-Definition

void main(void)                 // Hauptprogramm
{
    InitTouchP0P1 ("1");        // Touchscreen aktivieren
    while(1) { }                // Benutzerprogramm
}
```

- 1) **InitTouchP0P1 ("0");** Der Touchscreen bleibt ausgeschaltet
P0 ist als Input, P1 als Output konfiguriert
- 2) **InitTouchP0P1 ("1") .. ("1 r m p");** Der Touchscreen wird aktiviert und konfiguriert, einfachste Konfiguration mit InitTouchP0P1 ("1").
 - 1...4: Gemeinsamer Schalter-/Tastertyp
 - p: P0 aussen, sonst mittig.
 - m: MSB oben/rechts, sonst unten/links.
 - r: Rotiert horizontal, sonst vertikal.

13 Anhang Anschlüsse am μ C-Board MCB32

Entwicklungsanschlüsse

Fremdspeisung
genau 5V! oder ...

USB Speisung

USB - ST-Link
Zielsystemdebugging,
Boardspeisung
wie oben

Bootloadschalter:

Ungedrückt lassen **LED OFF**

Reset-Taster:

Programmrestart

Digitale Ein- und Ausgaben am μ C-Board MCB32

P1: 8x onboard LEDs
parallel zu 8x extern LEDs

Pot auf Anschlag Uhrzeigersinn
drehen, da gleichzeitig P0_4

P0: 8 externe Schalter

ControlStick

Button 1

Button 0

// In P0P1Touch.h definierte Pin-Bezeichnungen PA_0 .. PD_11, ohne Bezeichner wie Button .. !

```
char Button0      = PA_0;           // Bitwert 1/0, aktiv low, prellt wenig
char Button1      = PC_13;

char Stick        = PD_High;
char StickSelect  = PD_15;
char StickDown    = PD_14;
char StickLeft    = PD_13;
char StickUp      = PD_12;
char StickRight   = PD_11;

// als Byte 0xF8 open, aktiv low, alle entprellt
// Bitwert 1/0; Bytewert 0x80
//          1/0;          0x40
//          1/0;          0x20
//          1/0;          0x10
//          1/0;          0x08
```

14 Anhang Debuggen

14.1 Methodik des Debuggens (entlausen)

Prinzip: Das Debugging bezweckt das Auffinden von logischen Fehlern, die sich erst bei Laufzeit mit falschem Verhalten bemerkbar machen. (Semantik)

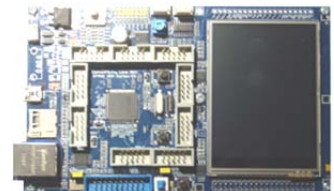
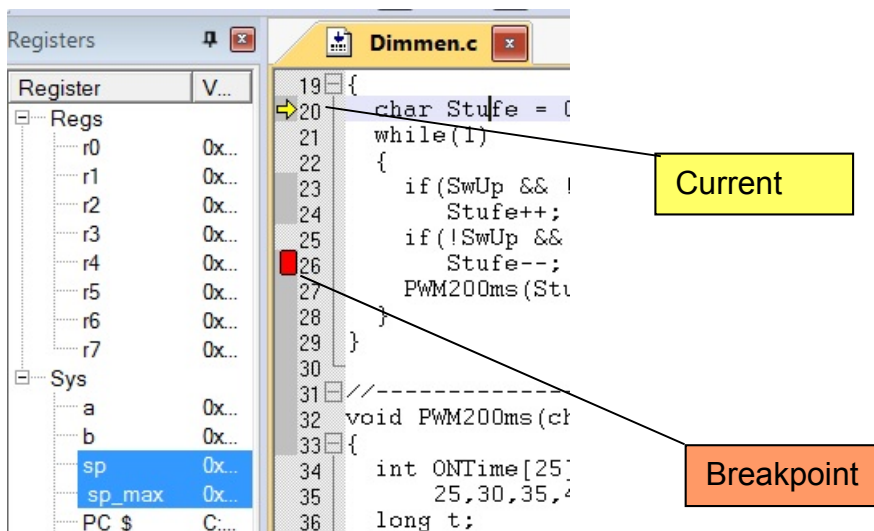
Hochsprachendebugging

verfolgt Daten und Programmablauf im Hochsprachen-Sourcecode auf dem PC. Der uController und seine Peripherie werden auf dem PC simuliert.

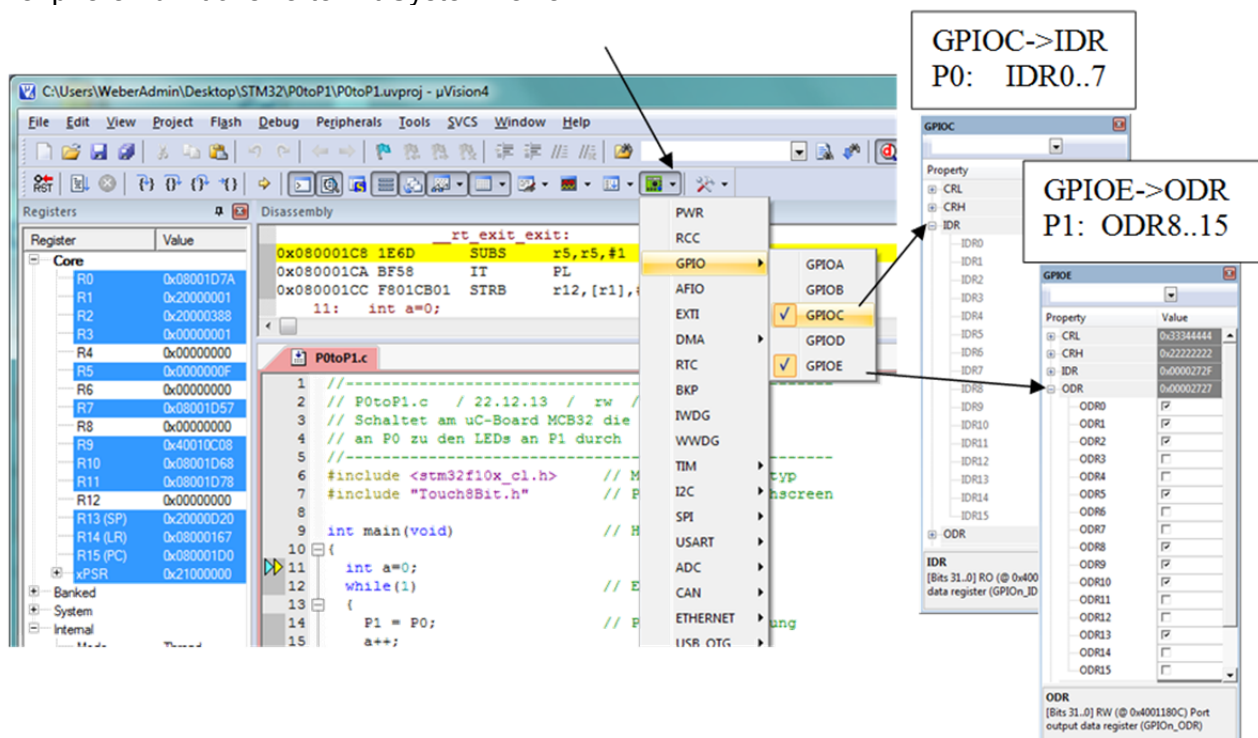


Zielsystemdebugging

lässt jeden Programmschritt im real angeschlossenen Mikrocontroller ausführen. Der PC steuert und zeigt die realen Daten des Controllers und seiner Peripherie.

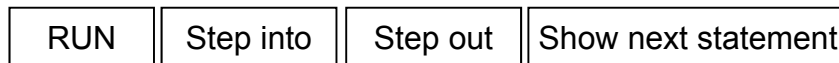
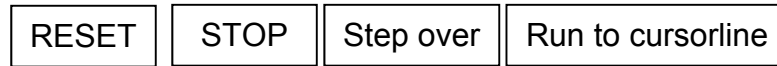


Periphere Funktionswerte mit Systemviewer



14.2 Verfolgen des Programmlaufs

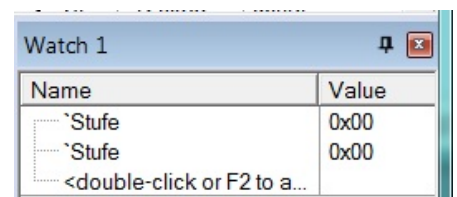
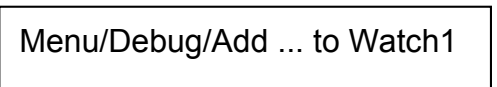
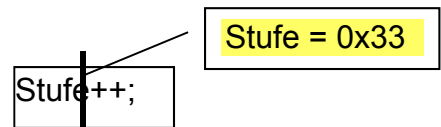
Werden die **Programmschritte** in richtiger Reihenfolge erreicht und ausgeführt, **Verzweigungen** ausgeführt, **Schleifen** wiederholt und **Unterprogramme** erreicht und ausgeführt?



14.3 Überprüfen der Daten

Im Stepbetrieb am Cursor und im Watchfenster beobachten:

- Variablenwerte: Rechenresultat, Bereichsüberschreitung,
- Überlauf, erreichbare Bereichsgrenzen
- Logische Entscheidung True/False,
- logische Bitverknüpfungswerte 00...FF
- Übergabe- und Rückgabewerte von Unterprogrammen
- Ein-/Ausgabewerte am richtigen Ort mit richtigem Wert.



Im Runbetrieb mit Breakpoints auf den Problemzeilen

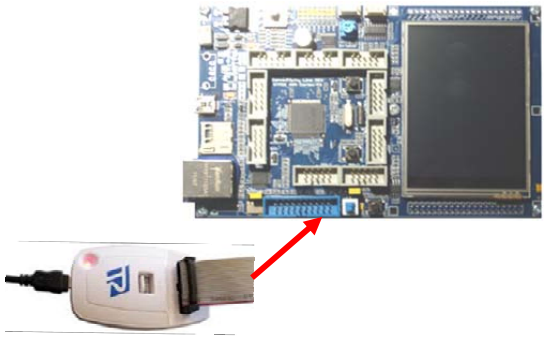
- Korrekte Verzweigung,
- Anzahl Wiederholungen,
- Unterprogrammeinsprung

14.4 Programmtest durch Zielsystemdebugging

Prinzip: Das Programm wird unter Kontrolle des PC's auf dem Zielsystem ausgeführt mit SingleStep, FunctionStep, Run, Variablenbetrachtung

Beachte:

- 1) Das Programm läuft bis. 1000x langsamer als in Echtzeit, also ewige Warteschlaufen!
- 2) Während RUN werden keine Bildschirminfos aufgefrischt, zu grosse Datenmenge!

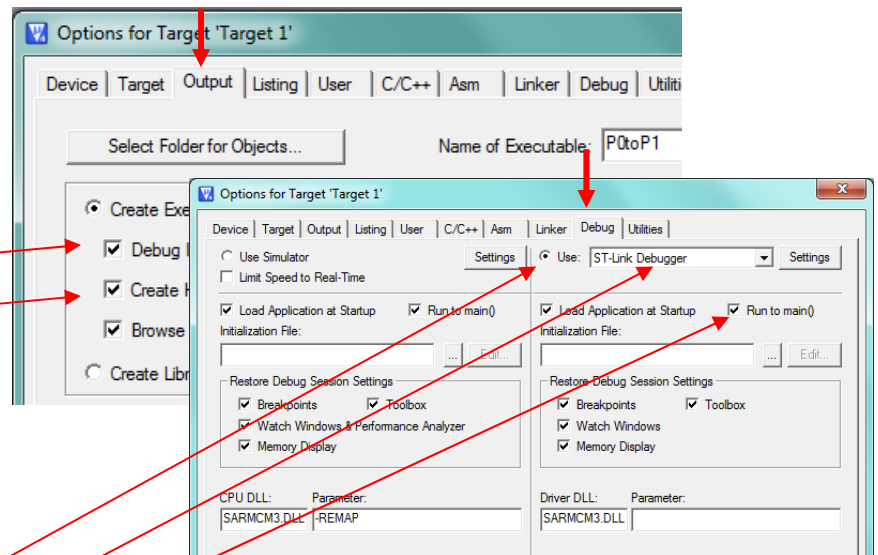


Projekt-Einstellungen für Zielsystemdebugging:



Project/Options for **Target/Output**:

- Debug Information
- Create HEX-File



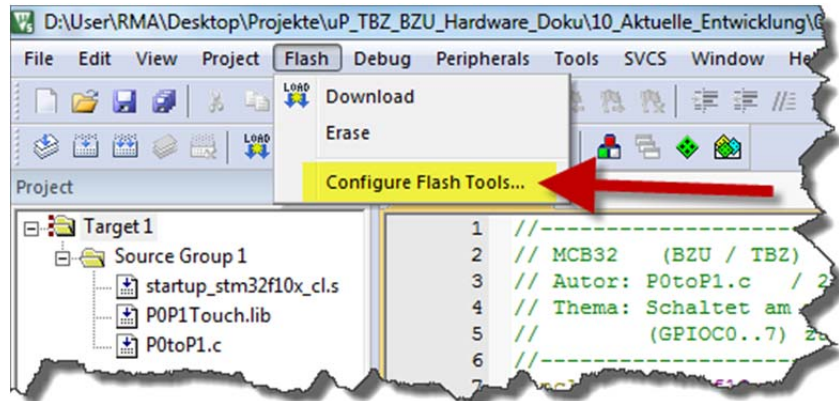
Project/Options for **Target/Debug**:

- Debugging aktivieren
- **ST-Link Debugger** auswählen
- Run to main()

14.5 Anhang: Debuggen mit Jtag Interface ST Link V2

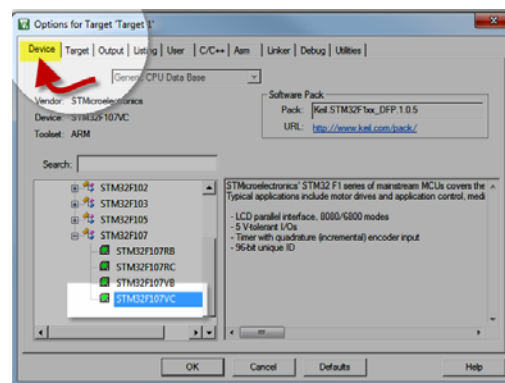
Nachfolgend sind die nötigen Schritte für die Bereitstellung des ST-Link-V2 gezeigt.

1. Programm laden und ohne Fehler kompilieren. Dann die Flashtools auswählen.



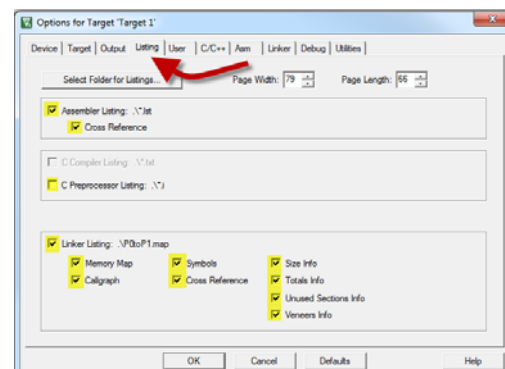
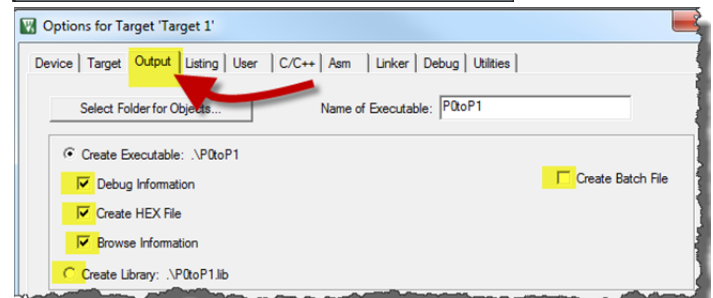
2. Device auswählen und prüfen ob der Chip ausgewählt

STM32F107VC



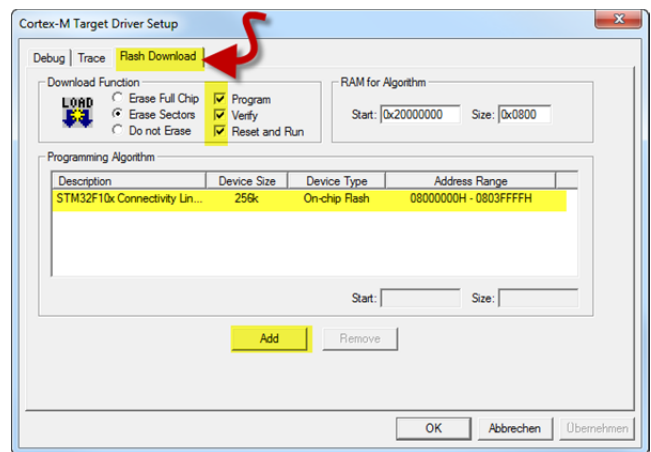
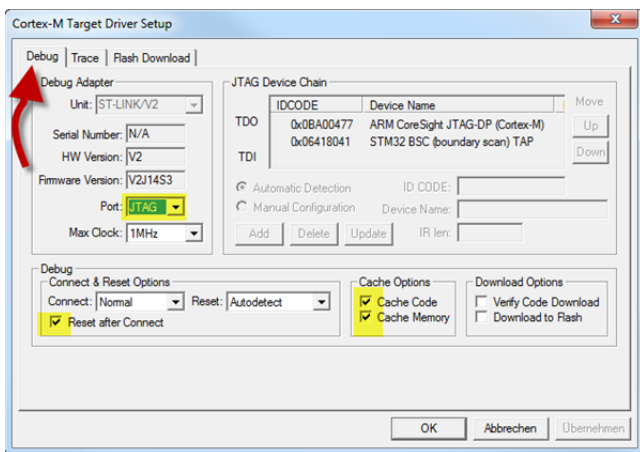
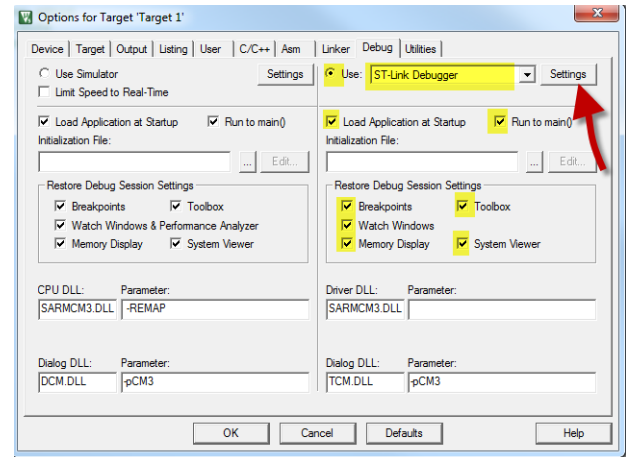
richtige ist.

3. Danach die Output- und Listing- Einstellungen kontrollieren.



4. Debugger Einstellungen kontrollieren. Dazu gehören auch die Settings beim Debugger.

Siehe dazu die beiden Bilder unten für DE-BUG und Flash-Download.

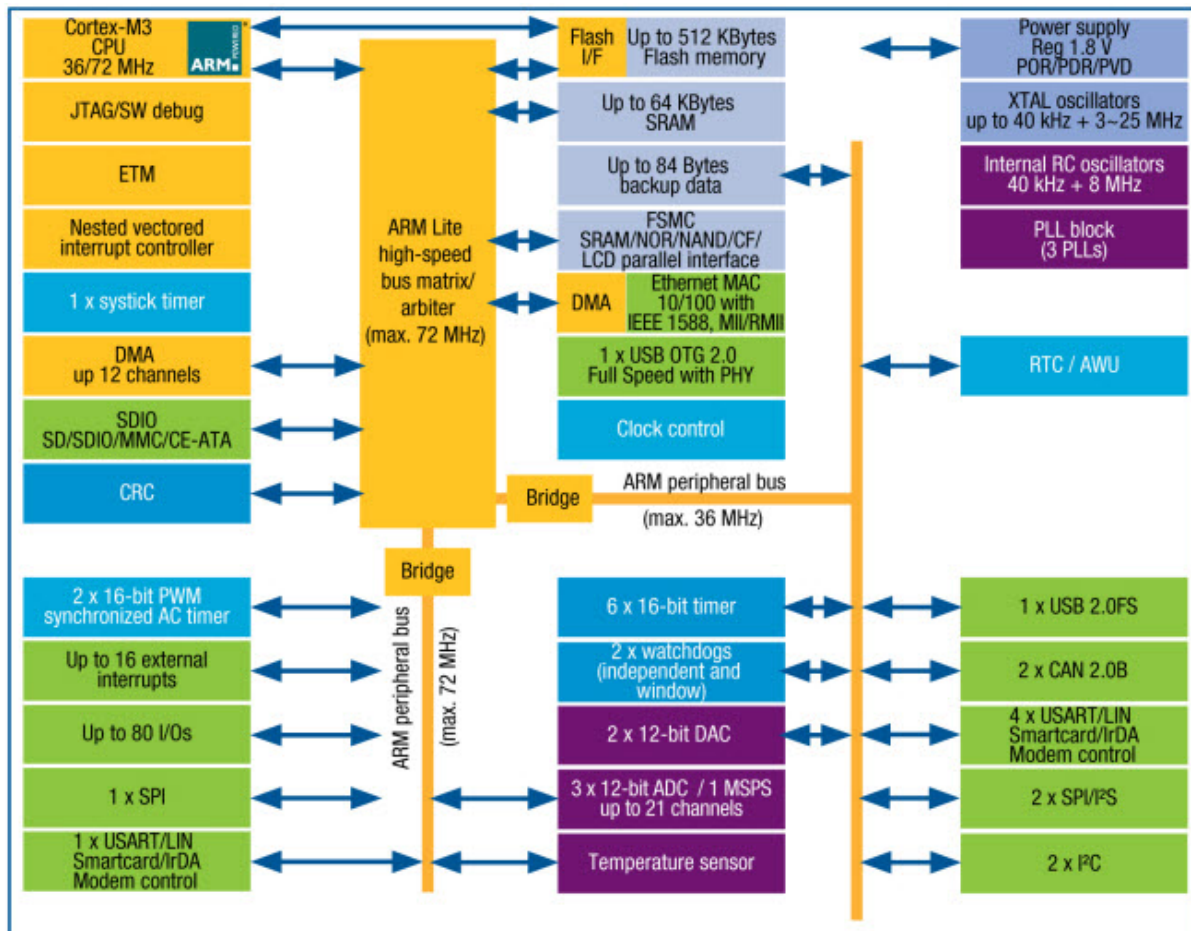


14.5.1 DEBUG FLASH DOWNLOAD

- Wenn bei den FLASH-Download Einstellungen kein Chip erscheint mit dem ADD Button den richtigen Chip auswählen.
- Wenn alle Einstellungen nun stimmen kann der HEX Code via JTAG LINK auf den Prozessor geladen werden.

15 Anhang ARM M3

15.1 Blockschema



AWU: Auto wake-up capability with RTC alarm
CAN: Controller area network
CF: CompactFlash
CRC: Cyclic redundancy check
DMA: Direct memory access
ETM: Embedded Trace Macrocell
IrDA: Infrared Data Association

I²S: Inter-IC sound
LIN: Local interconnect network
MII: Media independent interface
MMC: MultiMediaCard
PDR: Power-down reset
POR: Power-on reset
PVD: Programmable voltage detector

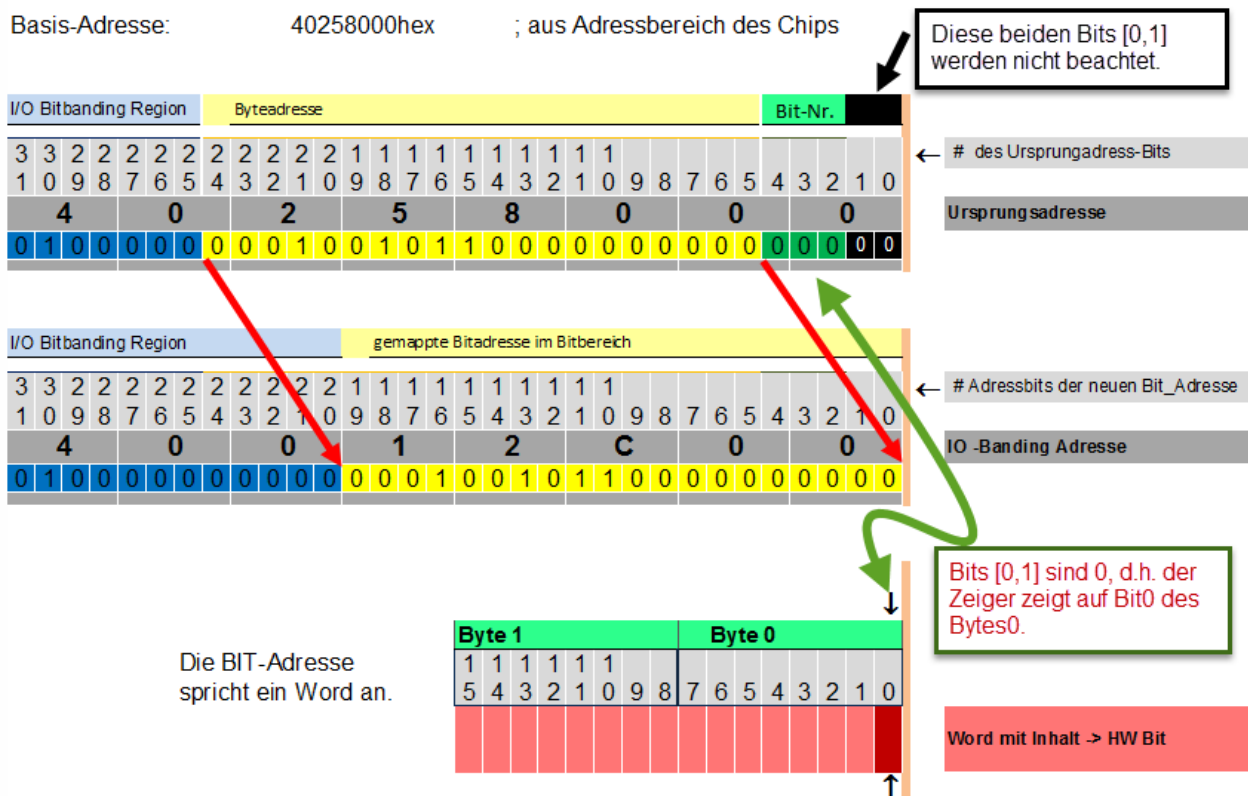
RMII: Reduced media independent interface
RTC: Real-time clock
SDIO: Secure digital input output
SD: Secure digital
USART: Universal sync/async receiver transmitter

15.2 ARM Bit-Banding

Bitmanipulationen bei einem 32Bit System mit einem 1G-Adressbereich sind im Prinzip nicht vorgesehen. Anstatt ein ganzes Wort einzulesen, es mit dem entsprechenden Bit über UND zu verknüpfen und es danach zurückzuschreiben, absolviert Bit-Banding dies mit einem einzigen Speicherbefehl. Dieser Einzelbefehl hat den wichtigen Vorteil: diese Operation wird „atomar“ ausgeführt. Auftretende Interrupts können einen solchen Befehl nicht unterbrechen. Es sind folgende Möglichkeiten vorgesehen:

1. Port lesen, Bit manipulieren und zurückschreiben (auch für nur Schreiben)
2. Bit Banding

Beim Bit-Banding wird via eine Ursprungs-Adresse ein einzelnes Bit angesprochen. Beim ARM-M3 sind dafür Adress-Bereiche reserviert worden. Die angesprochene Adresse wird dabei in einen Adress-Block und die dazugehörige Bitadresse sowie das angesprochene Bit eines Bytes unterteilt. Siehe nachfolgendes Bild für ein Beispiel wie aus der realen Adresse „40258000_{hex}“ die Bitadresse berechnet wird.



16 Anhang Code Beispiele

17 Anhang: Referenzen

- [1] E. Schellenberg, E. Frei / TBZ. Vorlage Script für diese Formatausgabe
- [2] Unterlagen MCB32 / Cityline / E. Malacarne
- [3] Robert Webewr / Berufsschulzentrum Uster / Projektvorlagen (div)
Danke Robi für die tollen Vorlagen.
- [4] Link zu Wikipedia. http://de.wikipedia.org/wiki/ARM_Cortex-M3 (1.9.14)
- [5] D. Schoch ; TBZ ; « C-Grundlagen.doc 29.5.2001 » ; Neu erstellt
- [6] E. Frei ; TBZ ; Erweiterungen in [1] « C-Grundlagen.doc 29.5.2001 »
- [7] Dirk Louis ; C++ « Programmieren mit Beispielen », MT-Verlag
- [8] Link zu Wikipedia: <http://de.wikipedia.org/wiki/Einerkomplement>

18 Anhang Literaturverzeichnis und Links

- [1] R. Jesse, Arm Cortex M3 Mikrocontroller. Einstieg und Praxis, 1 Hrsg., www.mitp.de, Hrsg., Heidelberg: Hütigh Jehle Rehm GmbH, 2014.
- [2] J. Yiu, The definitive Guide to ARM Cortex-M3 and M4 Processors, 3 Hrsg., Bd. 1, Elsevier, Hrsg., Oxford: Elsevier, 2014.

Weblinks <http://www.mikrocontroller.net/topic/158108> // für fehlendes volatile statement

19 Anhang Wichtige Dokumente

Die folgende Liste zeigt auf die wichtigsten Dokumente welche im WEB zu finden sind. Beim Suchen lassen sich noch viele nützliche Links finden.

- Datenblatt ([STM32F107VC](#)) Beschreibung des konkreten Chips für Pinbelegung etc.
- Reference Manual ([STM32F107VC](#)) (>1000Seiten in Englisch)
Ausführliche Beschreibung der Module einer Familie. Unter Umständen sind nicht alle Module im eingesetzten Chip vorhanden – siehe Datenblatt.
- Programming Manual ([Cortex-M3](#))
Enthält beispielsweise Informationen zum Interrupt Controller (NVIC).
- Standard Peripheral Library ([STM32F10x](#))
Im Gegensatz zu anderen MCUs sollen die Register der STM32 nicht direkt angesprochen werden. Dafür dienen die Funktionen der Standard Peripheral Library.
Sie ist auf <http://www.st.com/> zusammen mit einer Dokumentation (Datei: stm32f10x_stdperiph_lib_um.chm) herunterladbar.

20 Index

B

Bit-Banding 37

D

Datenblatt 39

M

MCU 15

N

NVIC 39

P

P0P1Touch.h 5, 28

P0P1Touch.lib 5, 28

Programming Manual 39

S

STM32 39

21 Anhang Notizen