Kapitel 4
Modulare Programmierung in C und Assembler

4.1 Einführung

Professionelle Programme im Embedded Bereich werden heute fast immer in C, seltener auch in C++ programmiert, weil die Programme damit schneller entwickelt werden können und besser wartbar sind als in Assembler. Trotzdem muss in diesen Projekten meist auch immer noch ein (kleiner) Teil in Assembler programmiert werden.

Gründe:

- **Algorithmen in Assembler** können Besonderheiten eines Maschinenbefehlsatzes nützen, die vom C-Compiler nicht genutzt werden und werden damit u.U. **schneller oder kleiner**. Einfachere C-Compiler, insbesondere solche, die zu unterschiedlichen CPU-Familien kompatibel sein müssen, nutzen häufig z.B. Bit-Set/Clear-Befehle nicht sondern verwenden AND/OR-Befehle mit Bitmasken, weil letztere im Gegensatz zu den Bit-Befehlen bei praktisch jeder CPU in gleicher Form vorhanden sind. Dasselbe gilt für besondere Arithmetikbefehle, z.B. den EMACS-Befehl des HCS12, der gleichzeitig multiplizieren und addieren kann, in dieser Form aber praktisch auf keinem vergleichbaren Mikrocontroller existiert.
- **Zugriff auf Funktionen des Mikrocontrollers**, für die es keine äquivalenten C-Befehle gibt, z.B. auf das Statusregister CCR.

Für das Zusammenspiel von C- und Assemblerprogrammen existieren zwei Möglichkeiten:

- **Inline-Assembler**: Maschinenbefehle, die in ein C-Programm eingebettet werden.
- **Assembler-Module**: Eigenständige, reine Assembler-Unterprogramme, die von C-Funktionen aus ausgerufen werden.

4.2 Inline-Assembler (siehe [3.13 Abschnitt High Level Inline Assembler])

Einzelne Maschinenbefehle können problemlos innerhalb einer C-Funktion als **`asm befehl`**; eingefügt werden. Für ganze Blöcke von Maschinenbefehlen verwendet man **`asm { ... }`**

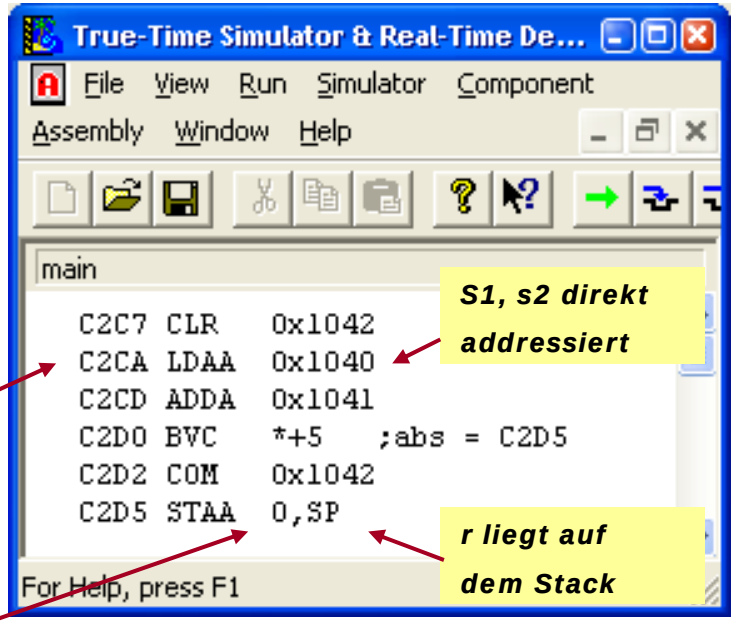
Da der C-Compiler notgedrungen dieselben CPU-Register verwendet wie die Inline-Maschinenbefehle, sind für das reibungslose Zusammenspiel Vorsichtsmassnahmen notwendig:

- **Variablen- und Konstanten** sollten **in C** deklariert werden. Assemblerbefehle können auf diese Variablen unter Verwendung des Variablennamens zugreifen. Globale C-Variable können in allen Befehlen verwendet werden, die die direkte Adressierung zulassen, lokale C-Variable dort, wo die Register-indirekte Adressierung (über **SP**) möglich ist.
- Der **Stack-Pointer** muss am Ende jedes Blocks von Maschinenbefehlen wieder denselben Wert haben wie zu Beginn des Blocks.
- Die Adresse einer C-Variable ist im Assemblerblock als **`@variablenName`** zugänglich.
- Auf die Elemente von Strukturen und Unions wird wie in C zugegriffen, z.B. **`LDD r.a`**
- Elemente von C-Arrays können ebenfalls verwendet werden: **`int a[20]; asm LDD a:24`** greift auf das Element **`a[12]`** zu, wobei der C-Index im Assemblerbefehl als Byte-Offset dargestellt wird, d.h. **`variablenName : C-Index · sizeof(variablenTyp)`**

Wegen möglicher Konflikte bei der Verwendung von Registern sollten Inline-Assembler-Blöcke entweder in separate C-Funktionen ausgelagert oder verwendete **Register auf** dem **Stack gesichert und** am Ende des Assembler-Blocks wieder **restauriert** werden.

Beispielprogramm **InlineAsm.mcp**

- Addition von zwei 8bit Zahlen und Anzeige bei Überlauf




```
char s1, s2;                    // Global variables
char flag;

void main(void)
{   char r;                     // Local variable
     . . .
    for(;;)
    {   . . .
        s1 = . . .              // Input summands from terminal
        s2 = . . .
        . . .
        flag = 0;

        asm
        {       LDAA s1         // Compute r = s1 + s2, if overflow occurs, set flag=1
                ADDA s2         // Compute sum
                BVC  noov       // Check for overflow
                COM  flag       // ... and set flag if overflow occured
          noov: STAA r          // Store result
        }

        sprintf(temp, "\nc =%d + %d = %d   %s\n", s1, s2, r, flag ? "Overflow" : "No overflow");
        PutString(temp);        // Output result to terminal
    }
    asm SWI;                    // Exit program (stop simulator or return to monitor program)
}
```

4.3 Assembler-Unterprogramme in Verbindung mit C-Programmen

(siehe [3.13 Abschnitt Motorola HC12 Backend – Call Protocol and Calling Conventions sowie Stack Frames])

Um Parameter an ein Unterprogramm zu übergeben und von diesem ein Ergebnis an das aufrufende Programm zurück zu liefern, müssen beide Seiten ein einheitliches Konzept für die Datenübergabe verwenden:

```
rückgabeDatentyp funktion(C_Datentyp1 param1, . . ., C_datentypN paramN)
```

Prinzipiell stehen zur Verfügung:

- **Übergabe in Registern** (Vorteil: schnell, Nachteil: Nur wenige Parameter möglich), wird hauptsächlich bei reinen Assemblerprogrammen verwendet
- **Verwendung globaler Variabler** (Vorteil: Datenmenge praktisch beliebig, Nachteil: Verschachtelte Funktionsanrufe (Reentrancy, rekursive Programme) schwierig oder unmöglich
- **Übergabe über den Stack** (Vorteil: Sehr flexibel, Nachteile: Langsam, komplizierte Adressierung der Parameter)

C-Compiler verwenden in der Praxis eine Mischung zwischen der Übergabe in Registern und über den Stack. Leider sind diese Verfahren (sogenannte **Aufrufkonventionen**) nicht durchgängig standardisiert, sondern von Programmiersprache zu Programmiersprache und von Compilerhersteller zu Compilerhersteller unterschiedlich.

Der **Metroworks HCS12-C-Compiler** verwendet folgendes Verfahren:

4.3 C mit Assembler-Unterprogrammen

- Das **Ergebnis** einer Funktion wird **in** einem **Register** (siehe Tabelle nächste Folie) zurückgegeben.
- Wenn eine Funktion eine feste Anzahl **N** von Parametern hat: Die **Parameter 1 bis N-1** werden „**von links nach rechts**“ (sogenannte PASCAL-Reihenfolge) **über den Stack** übergeben, der **letzte Parameter paramN** wird **in** einem **Register** übergeben (siehe Tabelle nächste Folie).

  Die **Parameter** werden vom **aufrufenden Programm** nach dem Unterprogrammaufruf **wieder** „vom Stack **abgeräumt**“, indem **SP** um die entsprechende Anzahl von Bytes, die als Parameter auf den Stack gelegt wurden, korrigiert wird. D.h.

  param1 → Stack (“Push”)
  . . .
  paramN-1 → Stack
  paramN → Register
  Aufruf des Unterprogramms
  SP + Anzahl Parameter-Bytes → **SP** “Abräumen des Stacks”

- Wenn eine Funktion eine variable Anzahl von Parametern hat, z.B. **printf()**, **sprintf()**, …, werden dagegen **alle** Parameter „von rechts nach links“ übergeben, d.h. der Parameter paramN wird als erster, der Parameter param1 als letzter vor dem Unterprogrammaufruf auf den Stack gelegt (sogenannte C-Reihenfolge).

Datentyp des letzten Aufrufparameters bzw. Rückgabewertes	*Länge in Byte*	*CPU-Register*
`char`	1	**B**
`int` (Near) Pointer **`*beliebiger_C_Datentyp`**	2	**D**
`long`	4	**X** (obere 2Byte), **D** (untere 2Byte)

Größere Datenstrukturen (Arrays, Strukturen, Unions) sollten nicht *by Value*, sondern lediglich *by Reference* , d.h. als Verweis über Pointer, übergeben werden, sonst ist der Compiler gezwungen, besondere „Tricks“ anzuwenden (siehe [3.13])

4.3 C mit Assembler-Unterprogrammen

Damit Compiler, Assembler und Linker die Programme korrekt übersetzen können, müssen die **Schnittstellen eindeutig definiert** werden:

Im aufrufenden C-Programm:

- Funktionsprototyp
  `C_datentyp funktion(C_Datentyp1 param1, . . ., C_datentypN paramN);`
- Verweis auf globale Assembler-Variable (schlechter Stil, aber möglich!)
  `extern C_datentyp assemblerVariable;`

Im aufgerufenen Assembler-Programm:

- Bekanntmachen des Assembler-Unterprogramms für andere Programmmodule (Export)

```
XDEF nameAsmUnterprogramm        // am Dateianfang
nameAsmUnterprogramm:  ...       // eigentliches Programm im Programmcode
```

- Bekanntmachen einer globalen Assemblervariable für andere Programmmodule (Export)

```
XDEF  assemblerVariable          // am Dateianfang für Byte-Variable
assemblerVariable DS.… …         // übliche Definition globaler Variablen im Assemblerprogramm
```

- Verweis auf globale C-Variable (Import)

```
XREF  C_variable                 // am Dateianfang für Byte-Variable
```

Beispielprogramm **CwithASM.mcp**

- Addition von zwei 8bit Zahlen und Anzeige bei Überlauf

Programmteil in C:

```
char asmCompute(char s1, char s2); // Prototype

char s1, s2, r, flag;          // Global variables

void main(void)
{   . . .
    r = asmCompute(s1, s2);    // Call to assembler function
    . . .
}
```

Wird übersetzt in

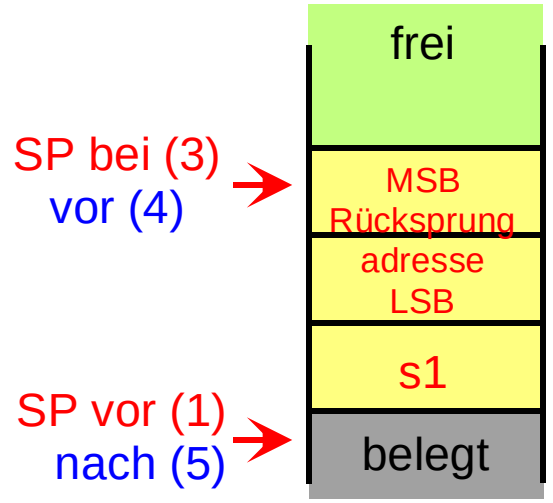
```
LDAB   s1
PSHB                       ;(1)
LDAB   s2
JSR    asmCompute          ;(2)
LEAS   1,SP                ;(5)
STAB   r
```

Programmteil in Assembler:

```
        XDEF    asmCompute      ; Export of ASM function
        XREF    flag            ; Import of global C variable

.init: SECTION                  ; Assembler program code in ROM
asmCompute:     LDAA  2, SP     ; Get s1 from stack      (3)
                ABA             ; s1 + s2 (in reg. B) --> reg. A
                BVC   noov      ; Check for overflow
                COM   flag      ; ... and set global C variable
                                ;     (bad programming style!)
noov:           TFR   A, B      ; Move result to B
                RTS             ; Return to caller       (4)
```

Stackzustand:



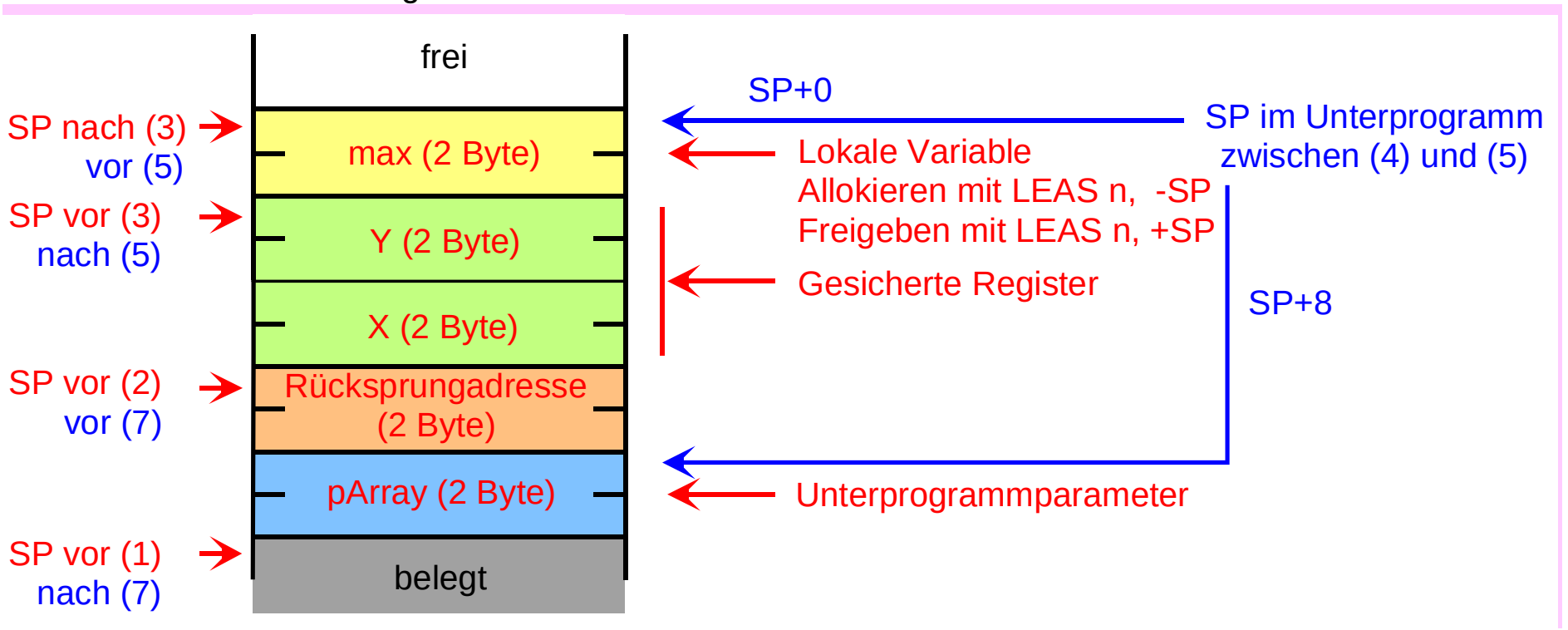
4.4 Lokale Variable in Unterprogrammen (Stack Frame)

Lokale Variablen sind ein bewährtes Konzept in vielen höheren Programmiersprachen. Assembler bietet leider keine direkte Möglichkeit, lokale Variable zu definieren. Man kann allerdings das von C-Compilern verwendete Konzept kopieren. C-Compiler legen lokale Variable (sogenannte Auto-Variable) auf dem Stack in einem sogenannten **Stack-Rahmen** an:

Beispielprogramm **`LocalVar.mcp`** : Suche nach dem Maximalwert in einem Array

→ Programmcode auf der folgenden Seite

- Stack während des Programmablaufs



Außerdem ist es guter Stil und wird von vielen C-Compilern erwartet, dass das Unterprogramm mit Ausnahme des Rückgaberegisters (hier D) keine anderen Register verändert.

```
int asmMaximum(int *pArray, char n);                  // Prototyp der Assemblerfunktion in C
int  val, array[] = { 47, 1600, -4500, 2000, 93, -2010 };  // Array
. . .
val = asmMaximum(array, 6);                              // Aufruf der Funktion          (1)
. . .
```

Assembler-Funktion

```
asmMaximum: PSHX                  ; Save registers                                        (2)
            PSHY
            LEAS  2, -SP          ; Allocate stack space for local variable max           (3)
            MOVW  #-32768, 0, SP  ; Initialize local variable max = -32768

            LDX   8, SP           ; pArray (from stack) -> X                              (4)
                                  ; n is in B
for:        LDY   0, X            ; current array element *pArray -> Y
            CPY   0, SP           ; compare with max
            BLE   next
            STY   0, SP           ; if greater -> max
next:       LEAX  2, +X           ; pArray++

            DECB                  ; n--
endFor:     BNE   for             ; if n > 0 go to next array element

            LDD   0, SP           ; return max
            LEAS  2, +SP          ; Deallocate space for local variable on stack          (5)
            PULY                  ; Restore registers                                     (6)
            PULX
            RTS                   ; Return to caller                                      (7)
```

4.5 Besonderheiten von C für Eingebettete Systeme

Allgemeine Gesichtspunkte bei der Programmierung von Eingebetteten Systemen in C, z.B. die Vorgänge beim Systemstart, werden in der Vorlesung Echtzeitsysteme [1.5] besprochen. Hier werden daher nur einige vor dem Assembler-Hintergrund besser verständliche Themen dargestellt:

- **Speicherplatz** ist bei Eingebetteten Systemen kostbar, am knappsten ist meist RAM
  - **Konstante Daten** grundsätzlich mit **`const ...`** deklarieren, damit der Linker sie im ROM platzieren kann.

- Aus demselben Grund wird der **Stack-Bereich** meist relativ **klein** gewählt
  - Anzahl der lokalen Variablen je Funktion begrenzen, vor allem **keine großen Arrays als lokale Variable**
  - **Keine rekursiven Algorithmen** verwenden, jede Rekursion belastet den Stack
  - **Schachtelung von Unterprogramm- und Interruptebenen** sinnvoll **begrenzen**, jede Unterprogramm- bzw. Interruptebene belastet den Stack
  - **Stack-Belastung** statisch in der Entwicklungsphase abschätzen und dynamisch während der Laufzeit **überwachen**, indem durch ein Hintergrundprogramm oder am Ende von Unterprogrammen der SP überprüft wird (Vergleich mit dem Stack-Anfang/Ende) oder der Stack zunächst mit einem bekannten Muster beschrieben und später überprüft wird, wie groß der Bereich am Ende des Stacks ist, in dem das Muster noch nicht zerstört ist (= Stack-Reserve).

- **Register-Variable**

Zu den geschwindigkeitssteigernden Massnahmen gehört das Halten von häufig verwendeten Variablen in CPU-Registern statt im langsameren RAM-Speicher. Der Compiler optimiert dies in der Regel selbstständig. Der Programmierer kann aber eingreifen, indem er bei der Deklaration von lokalen Variablen das Schlüsselwort **register** verwendet, z.B. **register int a;**

Dies ist (nur!) eine Empfehlung an den Compiler, die Variable **a** in ein Register zu legen. Bei Prozessoren mit sehr wenigen Registern (HCS12, 80x86 u.v.a.) ist die Wirkung dieser Optimierungsmassnahme aber gering.

- **Volatile Variable**

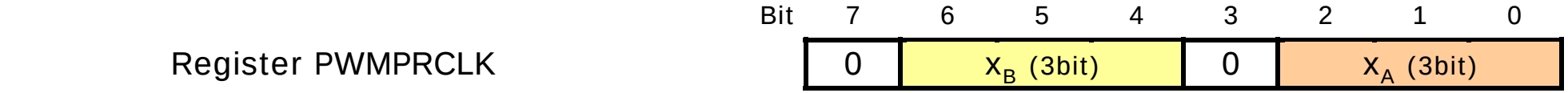
Aus Softwaresicht sind Register der Hardware-Peripheriebausteine globale Variable im Speicher. Der Compiler geht normalerweise davon aus, dass Variable nur durch den Programmablauf selbst verändert werden. Da das Lesen des Speichers langsam ist, versucht der Compiler, Variablen möglichst selten zu lesen. So würde der Compiler bei der folgenden Abfragesequenz das Timer-Zählerregisters **TCNT** nur einmal lesen. Durch **volatile** bei der Variablendeklaration zwingt man den Compiler, die Variable bei jeder Verwendung erneut zu lesen:

```
                        short *pTCNT = 0x0044;      short volatile *pTCNT = 0x0044;
if (*pTCNT > 100)           LDD TCNT                    LDD TCNT
    . . .                   CPD #100                    CPD #100
                            . . .                       . . .
if (*pTCNT > 200)           CPD #200                    LDD TCNT
    . . .                   . . .                       CPD #200
                                                        . . .
```

- **Bitweise Operationen**

Beim Zugriff auf Register der Peripheriebausteine ist es häufig notwendig, einzelne Bits oder Bitgruppen zu setzen oder zu löschen, ohne die anderen Bits zu verändern:

Bsp.: Im Vorteilerregister PWMPRCLK des PWM-Moduls soll der Wert $x_B=5_D=101_B$ gesetzt werden, die anderen Bits sollen nicht verändert werden.



Mögliche Lösung in C:
```
#define PWMPRCLK    (*(char*) 0x00A3)   //Adresszuweisung
PWMPRCLK = (PWMPRCLK & 0x8F ) | 0x50;
```

Die UND-Operation löscht die Bits 6..4, bevor die ODER-Operation sie entsprechend setzen kann. Der C-Compiler übersetzt diese dann in **AND** bzw. **OR**-Maschinenbefehle oder, falls er gut ist, in die effizienteren **BSET** und **BCLR**-Befehle, bei denen **PWMPRCLK** nicht explizit gelesen werden muss. Für den C-Programmierer sind die beiden notwendigen Bitmasken **0x8F** und **0x50** allerdings umständlich und fehlerträchtig. Einfacher zu verwenden ist ein C-**Bitfeld**:

```
typedef struct  {   char xA   : 3;      // Definition des Bitfelds
                    char res3 : 1;
                    char xB   : 3;
                    char res7 : 1;
                } PRCLK;
#define PWMPRCLK (*(PRCLK*) 0x00A3)     // Adresszuweisung
PWMPRCLK.xB = 5;                        // Setzen von xB = 5
```

- **Kompatibilität und Portabilität**

Assembler-Programme sind grundsätzlich nicht zwischen unterschiedlichen CPU-Familien portabel, da jede CPU-Familie ihre eigene Instruction-Set-Architecture (Registersatz, Maschinenbefehle, Adressierungsarten) verwendet.

Aber auch **C-Programme** sind alles andere als problemlos zu portieren:

Peripheriebausteine sind bei praktisch jeder CPU-Familie **unterschiedlich**, manchmal sogar innerhalb derselben CPU-Familie (z.B. ARM7-CPUs unterschiedlicher Hersteller verwenden sogar dieselbe Maschinensprache, aber völlig unterschiedliche Peripheriebausteine)

→ Hardware-abhängige Programmteile müssen so isoliert werden, dass die anzupassenden Programmteile schnell zu überschauen sind.

C enthält - trotz ANSI C89/90 oder ISO C99 Standard – **Spezifikationslücken**, so ist z.B. die Wortbreite des am häufigsten verwendeten Datentyps **`int`** vom Compilerhersteller frei wählbar. In der Regel entspricht sie der Datenwortbreite der CPU, z.B. beim HCS12 16bit, beim 80x86 (Intel/AMD) 32bit. Ähnliches gilt für das Speicherlayout von Strukturen und Bitfeldern, ob der Datentyp **`char`** defaultmässig **`signed`** oder **`unsigned`** ist und vieles andere

→ Datentypen sollten zentral definiert und im Programm nur mit den Anwenderdatentypen gearbeitet werden, z.B.

```
typedef  unsigned char  uint8
typedef  signed char    int8
typedef  unsigned int   uint16
. . .
```

Beim Umstieg auf einen anderen Compiler müssen dann nur die zentralen Typdefinitionen angepasst werden.

Die meisten Compiler-Hersteller sehen **proprietäre Spracherweiterungen** vor, die andere Hersteller in der Regel nicht in der gleichen Form anbieten. Der HCS12-CodeWarrior-C-Compiler kennt ebenfalls verschiedene Erweiterungen:

Die norm-konforme Art herstellerspezifischer Erweiterungen besteht im Schlüsselwort **#pragma**, z.B. als

```
#pragma TRAP_PROC    // Definiert das folgende Unterprogramm als ISR
void myInterruptServiceRoutine(void)
{ . . . }
```

Dieselbe Aufgabe (Definition einer ISR) kann beim CodeWarrior aber einfacher auch nicht norm-konform realisiert werden, wobei man zusätzlich auch die Interrupt-Nummer angeben kann und sich damit einen zusätzlichen Eintrag in der Linker-Steuerdatei spart:

```
void interrupt 23 myInterruptServiceRoutine(void)
{ . . . }
```

Herstellerspezifische Erweiterungen sind auch die beiden Makros zum Freigeben bzw. Sperren von Interrupts in der CPU:

```
EnableInterrupts
DisableInterrupts
```

Um auf Speicheradressen zuzugreifen, kann man standard-konform Pointer verwenden, z.B.

```
char *p = 0x0001;   //Pointer auf PORTB
*p = *p | 0x01;     //Setzt Bit 0 in Port B
```

oder herstellerspezifisch als

```
char PORTB @0x0001;
PORTB = PORTB | 0x01;
```

Dabei wird **PORTB** als Variable definiert und mit @... ihre Adresse im Speicher angegeben.

- **Rechenzeiten in Abhängigkeit vom Datentyp**

Angabe in CPU-Takten, 1 Takt = 42ns bei f_{BUSCLK}=24MHz, gemessen mit dem HCS12 True Time Simulator, C-Programm mit Default-Compiler-Optimierungen, Operanden und Ergebnis als globale Variable, die Ausführungszeit enthält nicht nur die reine Berechnung sondern auch das Laden der Operanden aus dem Speicher in die Register und das Abspeichern des Ergebnisses.

Operation		*Datentyp der Operanden a, b, c*			
		int *(16bit)*	*long* *(32bit)*	*float* *(IEEE 32bit)*	*double* *(IEEE 64bit)*
Addition/Subtraktion	c = a + b	9	21	203	500
Multiplikation	c = a * b	12	83	285	4174
Division	a = c / b	21	143	1049	5324

Schlussfolgerungen:

- Additionen und Subtraktionen sind schnell, Multiplikationen langsamer (falls die CPU kein Hardware-Multiplizierwerk für die volle Datenwortbreite hat), Divisionen eine Katastrophe
- Berechnungen mit Datentypen, die länger sind als die Datenwortbreite der CPU (16bit beim HCS12) werden langsam
- Ganzzahlberechnungen sind um Faktoren schneller als Gleitkommaberechnungen (falls die CPU keine Gleitkomma-ALU hat)

  → Ganzzahlberechnungen und kleinstmögliche Datenwortbreite verwenden, soweit die gewünschte Auflösung erreicht wird und bei der Berechnung noch keine Überläufe auftreten

  → Algorithmen verwenden, bei denen wenig/keine Divisionen notwendig sind