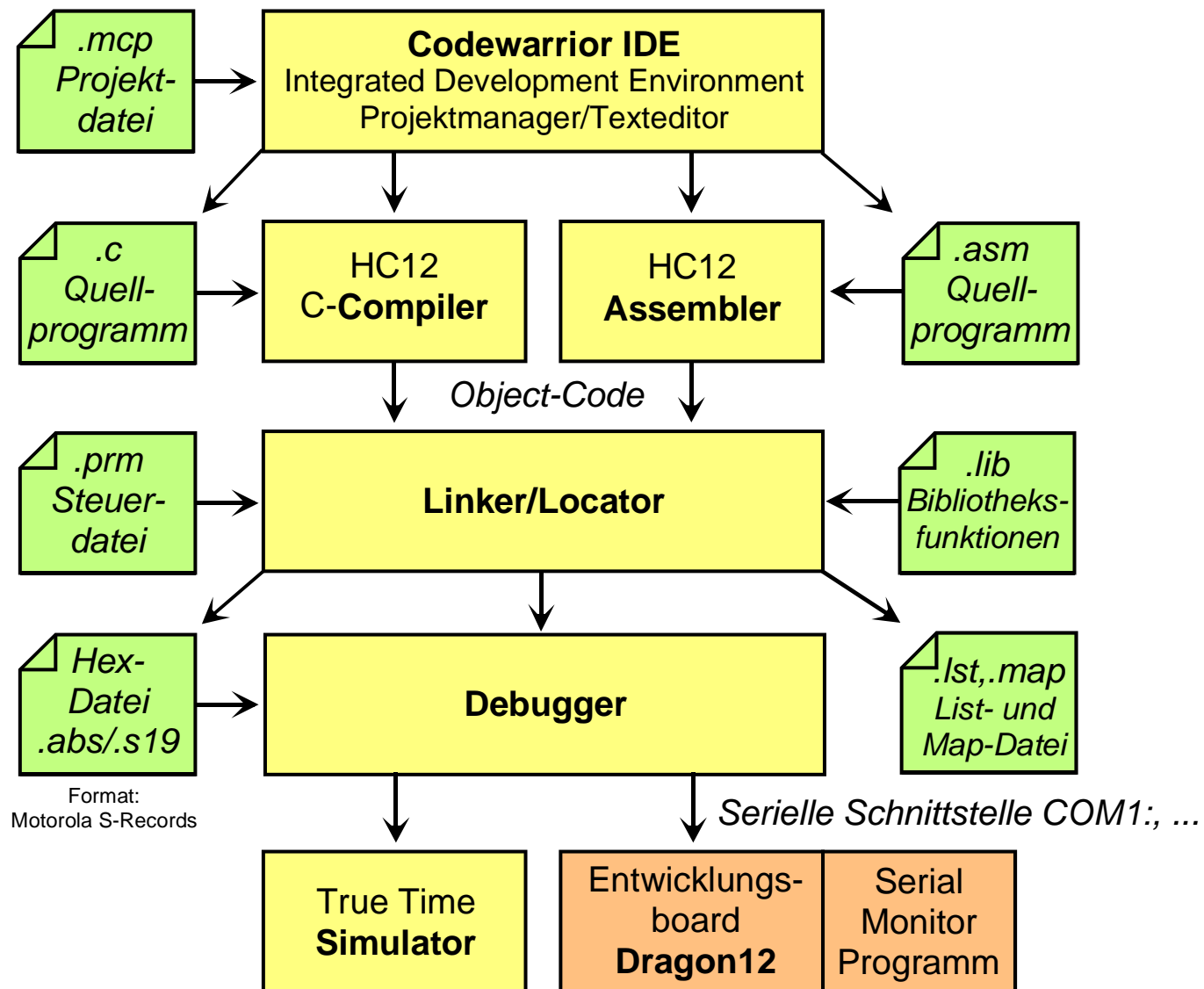

Anhang

Metroworks CodeWarrior HCS12 Entwicklungsumgebung

1.	Überblick über die Entwicklungsumgebung.....	2
2.	Installation	4
3.	Bedienung über Menü-, Tastatur- und Mausbefehle.....	4
4.	Arbeiten mit Projekten	5
5.	Aufbau und Syntax von Assembler-Programmen	15
6.	Debugger	22



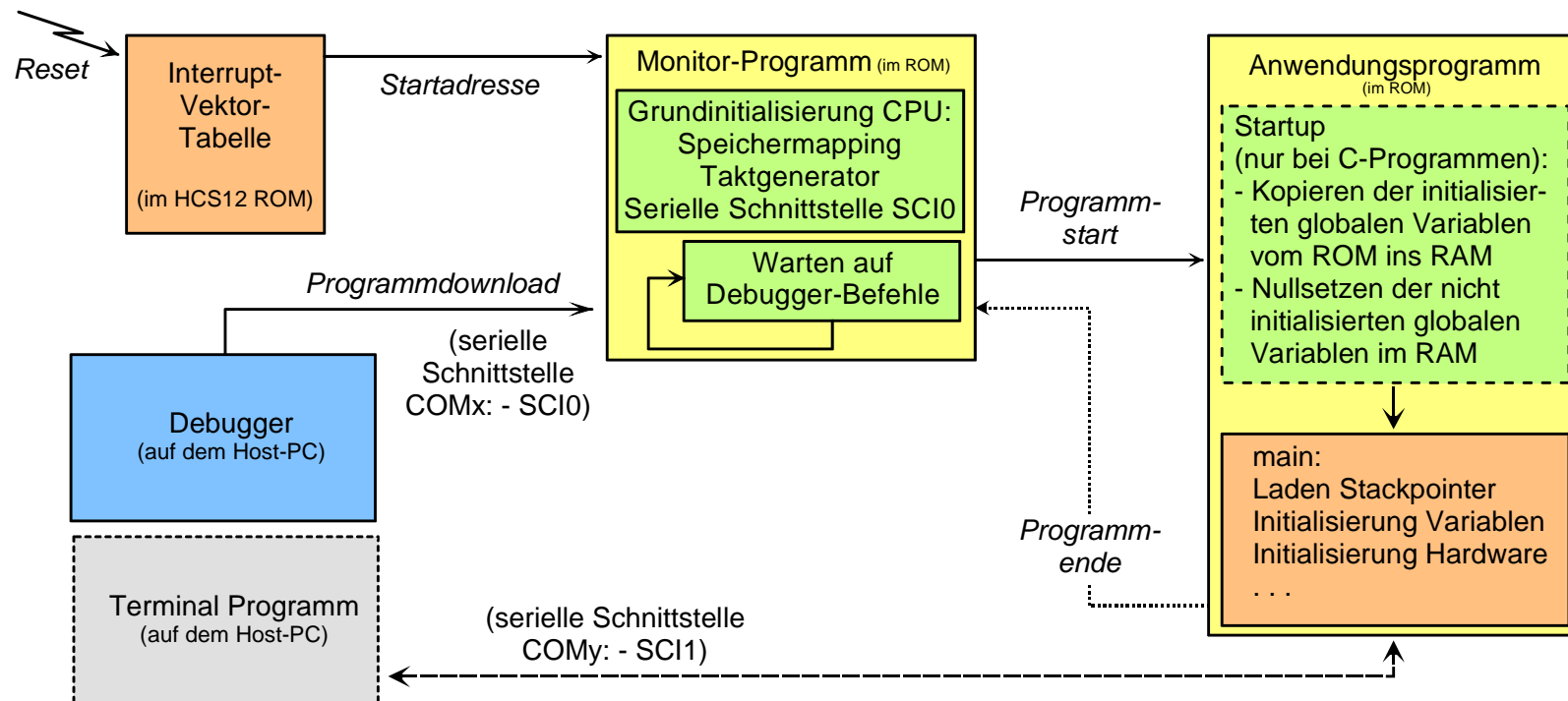
1. Überblick über die Entwicklungsumgebung

Im Labor wird die Entwicklungsumgebung Metroworks CodeWarrior HC12 Special Edition V3.1 mit Service Pack V2.95 von Freescale eingesetzt.

Die Umgebung besteht aus einer graphischen Entwicklungsumgebung (Integrated Development Environment IDE V5.5), die eine Projektverwaltung und einen Editor enthält und bei Bedarf Compiler, Assembler, Linker und Debugger (True Time Simulator and Real Time Debugger) aufruft.

Entwicklungsumgebung

- Die Anwendungssoftware wird vom Entwicklungsrechner (Host PC) mit Unterstützung eines ROM-residenten Monitor-Programms in das Flash-ROM des Dragon12-Entwicklungsboard (Target) geladen und von dort aus über einen Debugger-Befehl gestartet.
- In der Regel läuft das Programm in einer Endlosschleife. Falls es sich dennoch beenden will, muss es zum Monitor-Programm zurückkehren (über den Assemblerbefehl **SWI**).
- Nach dem Reset des Rechners wird (im Dragon12 Evaluation Board Modus EVB) lediglich das Monitor-Programm gestartet. Das Monitor-Programm führt die Grundinitialisierung der CPU durch, die Initialisierung der weiteren Hardware-Peripherie sowie sämtlicher Variablen erfolgt im Anwendungsprogramm.
- Ausgaben kann das Anwendungsprogramm entweder auf das LCD-Display des Dragon12-Boards oder über eine zweite serielle Schnittstelle zu einem Terminal-Programm machen.



2. Installation

Einzelheiten zur Installation finden Sie in [2.1 Anhang A]

3. Bedienung über Menü-, Tastatur- und Mausbefehle

Die Bedienung von CodeWarrior erfolgt über eine Mischung von Menüs, Icons, Dialogboxen und Tastatur. Meist sind mehrere Möglichkeiten vorhanden. Soweit nichts anderes angegeben wird, wird in dieser Beschreibung die Bedienung über Menüs vorausgesetzt und die **Abfolge von Menübefehlen** in Kurzform angegeben, z.B.

File – Open – Projektdatei mit Endung **.mcp** auswählen

Menübefehle sind *kursiv* geschrieben, Benutzeraktionen in Normalschrift. Dass bei der Befehlsfolge eine Dialogbox erscheint und dass am Ende der Aktion auf einen OK bzw. Open-Button geklickt bzw. die Eingabetaste gedrückt werden muss, wird in der Regel nicht zusätzlich angegeben.

In einigen Fällen ist die Bedienung über die Tastatur wesentlich schneller. **Funktionstasten** werden als F1, F2 usw. angegeben. Häufig sind auch **Tastaturkürzel** vorhanden, bei denen mehrere Tasten gleichzeitig betätigt werden müssen. Tastaturkürzel werden in der Regel bei Menübefehlen in Klammern angegeben, z.B.

Search – Find (CTRL+F)

Wenn die Tasten nacheinander gedrückt werden müssen, steht zwischen den Tastenkürzeln ein einfaches Komma, für gleichzeitiges Betätigen ein '+'. Die Abkürzungen für die wichtigsten Tasten sind:

CTRL	Steuerungstaste (häufig auch STRG)	ALT	Alt-Taste
SPACE	Leertaste	SHIFT	Umschalttaste

In der grafischen Schaltungseingabe wird häufig mit der Maus gearbeitet. **Mausaktionen** werden wie folgt abgekürzt:

LCLICK	Einfacher Klick mit der linken Maustaste	RCLICK	Einfacher Klick mit der rechten Maustaste
DKCLICK	Doppelter Klick mit der linken Maustaste	ZIEHEN	Maus bei gedrückter linker Taste bewegen

Häufig gebrauchte Funktionstasten:

F7	Make (Compile/Link)	F5	Debugger starten	CTRL+SHIFT+F7	Disassemble
----	---------------------	----	------------------	---------------	-------------

4. Arbeiten mit Projekten (siehe [3.12])

CodeWarrior verwaltet sämtliche zu einem Programm gehörenden Daten und Dateien in einem Projekt (Metroworks CodeWarrior Project **.mcp**).

4.1 Öffnen eines vorhandenen Projekts

Im Windows-Explorer: DCLICK auf die Projektdatei (Endung **.mcp**)

Im CodeWarrior: *File – Open – Name der Projektdatei (Endung **.mcp**)*

4.2 Anlegen eines neuen Projekts

Am einfachsten verwendet man ein vorhandenes Projekt des gewünschten Projekttyps (reines C-Projekt, reines Assembler-Projekt, gemischtes C/Assembler-Projekt) und kopiert das vollständige Projektverzeichnis.

Alternativ kann man mit Hilfe eines Wizards ein neues Projekt erstellen:

- *File – New – Registerkarte **Project** auswählen – **HC(S)12 New Project Wizard** auswählen – **Project name:** <name des Projekts> - **Location:** <Projektverzeichnis> - **OK***
- *Select Derivative (Prozessortyp):* auswählen – *Weiter*
- *Projekttyp auswählen:* anwählen - *Weiter*

- Nur bei reinen Assemblerprogrammen: *Objektcode-Typ:* auswäh-
len – *Weiter*

Sobald ein Projekt aus mehreren getrennt übersetzten Dateien besteht (C-Programm mit Bibliothek oder gemischtes C/Assemblerprogramm) muss verschiebbarer (relocatable) Objektcode verwendet werden, da die tatsächlichen Speicheradressen (absolute Adresse) erst beim Zusammenbinden der einzelnen Programmteile im Linker bestimmt werden können. Bei einfachen Assemblerprogrammen könnte der Programmierer die Adressen dagegen mit Hilfe von ORG Assemblerdirektiven direkt im Programmcode festlegen.

- Nur bei C-Programmen: *Use Processor Expert: No* auswählen – *Weiter* – *Setup PC-lint: No*
– *Weiter* – *Floating point support: None* – *Weiter* – *Weiter*

Ohne Floating point support können Sie in C-Programmen die Datentypen float und double nicht einsetzen. Falls dies ausnahmsweise notwendig sein sollte, wählen Sie *Floating point support: float is IEEE32, double is IEEE64*. Das Programm wird dann aber wesentlich größer und langsamer.

Im Memory Modell Small darf der Programmcode maximal 48 KB groß werden (max. Adressraum 64KB abzüglich Adressbereich für RAM und Peripherieregister). Für größere C-Programme können die Speichermodelle *Banked* und *Large* verwendet werden. Dadurch werden wahlweise zusätzliche ROM-Speicherblöcke in den Adressraum eingeblendet. In Assembler-Programmen ist die Adressierung in diesem Fall aber sehr unübersichtlich und sollte daher vermieden werden.

- *Connections:* und
auswählen – *Fertigstellen*

Danach erzeugt der Wizard ein vollständiges Projekt (siehe nächste Seite), das Templates für das C- bzw. Assemblerprogramm (.c, .asm), Include-Dateien (.h, .inc) sowie vordefinierte Steuerdateien für den Linker (.prm) und den Debugger (.ini, .cmd) enthält.

Projekte

CodeWarrior mit C/Assembler-Projekt

Target-Auswahl Target Settings Make Debug Meldungsfenster (nur bei Compiler/Linker-Fehlern sichtbar)

Quell-Programme (C bzw. Assembler)

Initialisierung für C-Laufzeit-system (nur C-Programme)

Linker-Steuerdateien .prm

Linker-Resultate .map

Bibliotheken. Header und Include-Dateien

Projektdatei (als Verzeichnisstruktur dargestellt)

Editor-Fenster

Metrowerks CodeWarrior - [Errors & Warnings]

File Edit View Search Project Debug Processor Expert Window Help

TestProjectCASM.mcp

Simulator

Files Link Order Targets

File	Code	Data
readme.txt	n/a	n/a
Sources	164	1
main_asm.h	0	0
main.asm	9	1
main.c	7	0
datapage.c	148	0
Startup Code	57	23
Start12.c	57	23
Prm	0	0
burner.bbl	n/a	n/a
Simulator_linker.prm	n/a	n/a
Monitor_linker.prm	n/a	n/a
Linker Map	0	0
Simulator.map	n/a	n/a
Monitor.map	n/a	n/a
Libraries	7K	2K
mc9s12dp256.inc	0	0
mc9s12dp256.h	0	0
mc9s12dp256.c	0	573
ansisi.lib	7846	2012
Debugger Project File	0	0
Debugger Cmd Files	0	0

26 files 7K 2K

Error : C2801: ';' missing
main.c line 12
Error : Compile failed

```
#include <hidef.h> /* common defines and macros */
#include <mc9s12dp256.h> /* derivative information */

#include "main_asm.h" /* interface to the assembly module */

#pragma LINK_INFO DERIVATIVE "mc9s12dp256b"

void main(void)
{
    EnableInterrupts; /* enable interrupts */
    asm_main() /* call the assembly function */
    for(;;) /* wait forever */
    {
    }
}
```

Aufbau eines C-Programms (siehe [3.13])

```
#include . . .                                     // Standard-CodeWarrior-Include-Datei
                                                    // Definition der CPU- und Peripherieregister

                                                    // CPU-Typ-Information für den Linker

#define . . .                                     // Definition von symbolischen Konstanten

typedef . . .                                     // Definition von anwendungsspezifischen Datentypen

. . . < Globale Variable > . . .                 // Deklaration und Initialisierung globaler Variabler

. . . < Funktionen bzw. Prototypen > . . .        // Definition von Funktionen bzw. Prototypen

void main(void)                                  // Hauptprogramm mit beliebigen Funktionsaufrufen
{
    . . .                                       // Freigabe der Interrupts (für den Debugger notwendig)
                                                    // Initialisierung der Hardware-Peripherie
                                                    // Endlosschleife

}
```

Die Header-Datei enthält symbolische Namen für alle CPU- und Peripherieregister, so dass diese direkt im C-Programm angesprochen werden können.

Soll ein C-Programm mit einem Assembler-Programm kombiniert werden, muss entweder das C-Programm eine Funktion **main()** oder das Assembler-Programm eine Einsprungmarke

main: (die zusätzlich unter **XDEF** exportiert wird) enthalten, aber niemals beide gleichzeitig (siehe Abschnitt 5). Weitere Hinweise zur Kombination von C- mit Assembler-Programmteilen finden Sie im Vorlesungskapitel 4, allgemeine Hinweise zu C-Programmen finden Sie in [3.13] und in [1.5].

4.3 Einfügen und Löschen von Dateien – Ändern von Optionen in Projekten

Am einfachsten verwendet man ein vorhandenes Projekt des gewünschten Projekttyps (reines C-Projekt, reines Assembler-Projekt oder gemischtes C/Assembler-Projekt) als Grundlage.

- Entfernen einer Datei aus einem Projekt:
In der Verzeichnisstruktur im linken Fenster Datei mit RCLICK anwählen – *Remove*
Die Datei wird dabei nur aus dem Projekt entfernt, aber nicht von der Festplatte gelöscht.
- Hinzufügen einer vorhandenen Datei:
In der Verzeichnisstruktur *Sources* mit RCLICK anwählen – *Add Files*
- Erstellen einer neuen Datei
File – New – File – Text File – File name: Name der Datei eingeben - *Location – Set:* Verzeichnis auswählen (Quellprogramme sollten in das Unterverzeichnis *Sources* des Projektes)
C-Programme müssen die Endung **.c** (nicht **.cpp**), Assemblerprogramme die Endung **.asm** erhalten. Neue Dateien werden in der Regel nicht automatisch in das Projekt eingefügt, sondern müssen, wie im vorigen Punkt beschrieben, manuell hinzugefügt werden.

Allgemeine Optionen

Unter *Edit – Preferences* können Sie allgemeine Optionen der Entwicklungsumgebung einstellen. Für die Beispielpprogramme zur Vorlesung empfehlen sich für alle Optionen die voreinges-

tellten *Factory Settings*. Ausnahme: Bei *Editor – Font & Tabs* sollten Sie eine Schriftart mit fester Zeichenbreite, z.B. *Courier New*, einstellen und die Tabulatorschrittweite *Tab size: 8* setzen.

Targets und Target Options

CodeWarrior verwaltet verschiedene Varianten (Targets) eines Projektes. Das **Simulator Target** ist für den Test des Programms mit dem Simulator ohne reale Hardware, das **Monitor Target** für den Test mit dem realen Dragon12-Entwicklungsboard vorgesehen. Die Auswahl eines Targets erfolgt über das Listenfeld im Projektfenster (siehe S. 7). Die zu einem Target gehörenden projektspezifischen Optionen werden über ein Dialogfenster eingestellt, das über das Icon *Target Settings* oder den Menüpunkt *Edit – Monitor Setting* bzw. *Simulator Settings* ausgewählt werden kann.

In der Regel muss an den vom Projekt-Wizard voreingestellten Werten nichts geändert werden. Bei Problemen sollten folgende Einstellungen überprüft werden:

	<i>Command line option</i> für	
	Assembler-Projekt	C- oder C/Assembler-Projekt
<i>Assembler for HCS12</i>		
• Simulator Target	-D_HCS12	-D_HCS12 -Ms
• Monitor Target	-D_HCS12	-D_HCS12 -Ms
<i>Compiler for HCS12</i>		
• Simulator Target	-	-D_HCS12 -Ms
• Monitor Target	-	-D_HCS12 -Ms -D_HCS12_SERIALMON

Um eine List-Datei mit dem übersetzten Programmcode zu erzeugen, kann man als zusätzlichen Parameter **-L** eintragen. Die List-Datei landet im **.\bin**-Verzeichnis des Projekts.

Falls Sie beim Disassemblieren eines C-Programms schwer durchschaubaren Assemblercode oder unerklärbare Laufzeitfehler erhalten, kann das daran liegen, dass der C-Compiler beim Übersetzen versucht, den Programmcode zu optimieren, um die Programmgröße zu verringern bzw. die Ausführungsgeschwindigkeit zu erhöhen. Sie können die verschiedenen Optimierungen über *Target – Compiler for HCS12 – Options – Optimizations* selektiv deaktivieren. In der Regel sollten Sie allerdings die Defaulteinstellungen nicht verändern.

4.4 Programme übersetzen: Compilieren, Assemblieren und Linken

- Wählen Sie das passende Target aus (siehe oben) und klicken Sie zum Übersetzen auf das Icon *Make* bzw. Menüpunkt *Project Make* (F7)
- Treten Fehler auf, erscheint ein Meldungsfenster, in dem die Fehler aufgezählt werden. Wenn Sie einen Fehler im Meldungsfenster anklicken, wird im Editor-Fenster die fehlerhafte Zeile (oder eine dazu benachbarte Zeile) mit einem roten Pfeil markiert. Falls das Meldungsfenster verdeckt ist, kann es mit *View – Errors and warnings* (CTRL+I) geöffnet werden. Falls keine Fehler vorhanden sind, wird es nicht angezeigt.
- Zum Testen können Sie den Debugger mit dem Icon *Debug* bzw. Menüpunkt *Project – Debug* (F5) starten (siehe Abschnitt 6).

4.5 Disassemblieren – List-Dateien

- Sie können sich den vom Compiler bzw. Assembler erzeugten Programmcode jederzeit anzeigen lassen, indem Sie im Editor-Fenster *RCLICK – Disassemble* oder Menüpunkt *Project – Disassembler* (CTRL-Shift-F7) anwählen.

Bei Assembler-Programmen sehen Sie dabei vor Ihrem eigenen Programmcode den Inhalt der Include-Datei **mc9s12dp256.inc**, so dass Sie nach Ihrem eigenen Code etwas suchen müssen.

Projekte

Beispiel: Disassemblieren von `main.c` aus dem Projekt `BlinkingLeds.mcp`

```
30: void main(void)
31: {   EnableInterrupts;           ← C-Befehl mit Zeilennummer im C-Programm

32:
33:     DDRJ_DDRJ1 = 1;

34:     PTJ_PTJ1   = 0;

...
39:     DDRB  = 0xFF;

. . .
```

Beispiel: Disassemblieren von `main.asm` aus dem Projekt `BlinkingLedsAsm.mcp`

```
    43    24i
    44    25i
. . .
12074    32          main:
12075    33          Entry:
12076    34          LDS   #__SEG_END_SSTACK
12077    35          CLI
12078    36
12079    37          BSET  DDRJ, #2
12080    38          BCLR  PTJ,  #2
```

4.6 Linker und Map-Datei (siehe [3.17])

Der Projekt-Wizard erzeugt Steuerdateien für den Linker/Locator, die die Memory Map des Mikrocontrollers beschreiben und damit die Platzierung des Programmcodes und der Variablen im Speicher festlegen. Für jedes Target gibt es eine separate Steuerdatei, Details sh. [3.17].

Beispiel: **Monitor_Linkер.prm** aus dem Projekt **BlinkingLeds.mcp**

```
. . .
SEGMENTS
    RAM      = READ_WRITE 0x1000 TO 0x3FFF;
    ROM_4000 = READ_ONLY  0x4000 TO 0x7FFF;
    ROM_C000 = READ_ONLY  0xC000 TO 0xFEFF;
    . . .
END

PLACEMENT
    STARTUP,      /* startup data structures */
    ROM_VAR,      /* constant variables */
    . . .
    DEFAULT_ROM           INTO  ROM_C000;
    DEFAULT_RAM           INTO  RAM;
    . . .
END

STACKSIZE 0x100

VECTOR 0 _Startup
```

Neben der Objektdatei, die mit dem Debugger in das Flash-ROM des Mikrocontrollers geladen werden kann, erzeugt der Linker/Locator eine Map-Datei, die einen Überblick über die tatsächliche Adresszuordnung und Speicherbelegung gibt. Am interessantesten darin sind die Abschnitte, die die Programmgröße und die Adressen der Funktionen und Variablen angeben.

Beispiel: **Monitor.map** aus dem Projekt **BlinkingLedsAsm.mcp**

```
. . .
*****
SECTION-ALLOCATION SECTION
Section Name                Size  Type    From      To        Segment
-----
.init                       50    R      0xC000    0xC031    ROM_C000
.stack                     256   R/W    0x1000    0x10FF    RAM
.vectSeg0_vect              2     R      0xFFFFE  0xFFFF   .vectSeg0

Summary of section sizes per section type:
READ_ONLY (R):              34 (dec:    52)
READ_WRITE (R/W):          100 (dec:   256)
. . .
*****
OBJECT-ALLOCATION SECTION
      Name                Module                Addr    hSize    dSize    Ref    Section    RLIB
-----
- PROCEDURES:
    Entry (main)          C000         21      33      0      .init
    loop                  C021         6       6       0      .init
. . .
- VARIABLES:
. . .
- LABELS:
    __SEG_END_SSTACK      1100         0       0       1
```

5. Aufbau und Syntax von Assembler-Programmen (siehe [3.14])

<code>; export symbols</code>	<code>; Export von Funktions- und Variablennamen für den Linker</code> <code>; (notwendig für Namen, die von anderen Modulen verwendet werden)</code>
<code>; import symbols</code>	<code>; Import von Funktions- und Variablennamen</code> <code>; (notwendig für Namen, die in anderen Modulen definiert wurden)</code> <code>; hier: Adresse des Endes des Stack-Bereiches</code>
<code>; include derivative specific macros</code>	<code>; Definition der CPU- und Peripherieregister</code>
<code>. . . : . . .</code>	<code>; Definition von symbolischen Konstanten</code>
<code>; RAM: Variable data section</code>	
<code>.data:</code>	<code>; Speicherbereich für Variable im RAM</code>
<code>. . . : . . .</code>	<code>; Deklaration globaler Variabler (keine Initialisierung möglich!)</code>
<code>; ROM: Constant data</code>	
<code>.const:</code>	<code>; Speicherbereich für initialisierte Konstanten im ROM</code>
<code>. . . : . . .</code>	<code>; Deklaration und Initialisierung von konstanten „Variablen“</code>
<code>; ROM: Code section</code>	
<code>.init:</code>	<code>; Speicherbereich für Programmcode</code> <code>; Anfangsadresse des Hauptprogramms (Einsprungpunkt)</code>
	<code>; Initialisierung des Stack-Pointers</code> <code>; Freigabe der Interrupts (für den Debugger notwendig)</code> <code>; Initialisierung der Hardware-Peripherie</code>
<code>. . .</code>	
<code>Loop: . . .</code>	<code>; Endlosschleife</code>
<code> Loop</code>	

Syntax in Assemblerprogrammen

- **Maschinenbefehle**

label **befehl** **Operand(en)** **Kommentar** z.B. `MOVB #$01, PORTB ; Initialisierung`

label (Marke) markiert die Adresse eines Befehls (oder Datums) symbolisch und ist nur für den ersten Befehl einer Funktion sowie für Programmstellen erforderlich, auf die mit einem Sprungbefehl gesprungen werden soll. Jedes **label** muss einen eindeutigen Namen haben.

befehl ist ein beliebiger HCS12 Befehl mit den zugehörigen Operanden.

Kommentar (inklusive des `;`) ist ein optionaler **Kommentar** bis zum Zeilenende

- **Gross-/Kleinschreibung**

Befehle können beliebig groß oder klein geschrieben werden, bei Labels und Variablennamen ist die Schreibweise beliebig, sollte (muss) aber durchgängig gleich sein.

- **Einrückungen**

Ein **label** sollte in der ersten Spalte einer Zeile stehen, ohne Label sollte die erste Spalte leer bleiben (Leerzeichen oder Tabs). Maschinenbefehle und Assembler-Direktiven beginnen in der zweiten Spalte.

- **Einsprungpunkt (Programmbeginn):**

Der erste Befehl des Assembler-Hauptprogramms muss mit den Marken **Entry:** und **main:** markiert werden (siehe auch **XDEF**)

- **Einbinden von Dateien**

dateiname bindet eine Datei mit Definitionen, Variablen oder Funktionen ein.

label z.B. `.init: SECTION`

- **Symbolische Abkürzungen** (ähnlich wie `#define ...` in C)

Anstatt im Programmcode direkt Zahlenwerte zu verwenden, z.B. die hexadezimale Adresse eines Peripherieregisters, verwendet man symbolische Abkürzungen (**label**), die am Programmbeginn oder in einer Include-Datei definiert werden. Falls der Wert geändert werden muss, kann dies damit zentral an einer einzigen Stelle getan werden. Die Namen von Peripherieregistern sind bereits in der Include-Datei `mc9s12dp256.inc` definiert. Der Linker definiert das Symbol `__SEG_END_SSTACK`, das das Ende des Stacks bezeichnet.

16382	...	Dezimalzahl	3FFE	...	Hexadezimalzahl
37776	...	Oktalzahl	1101	...	Binärzahl
A , AB	...	ASCII-Zeichen bzw. Textstring			

Beachten Sie, dass bei der unmittelbaren Adressierung `#...` geschrieben werden muss, also z.B. `MOVB #$20, PORTB`. Wenn `name` eine Variable ist, verwenden Sie `name` als Operand, wenn Sie den Wert der Variable und `#name`, wenn Sie die Adresse der Variable benötigen.

Assembler-Syntax

Achtung: Im Gegensatz zu C werden Strings nicht automatisch durch 0_H abgeschlossen. Die von C bekannten Sonderzeichen `\n`, `\r`, `\t` usw. gibt beim HCS12-Assembler nicht. Falls notwendig, müssen sie als Symbole definiert werden, z.B. `CR: EQU $0D`, `LF: EQU $0A`

Bei der Angabe des Wertes sind auch einfache Berechnungen möglich, z.B. `A EQU 3*5+4`, allerdings dürfen dabei nur konstante, zur Übersetzungszeit bekannte Größen verwendet werden, also keine Register- oder Variableninhalte o.ä.

- **Globale Variable**

.data		Speicherabschnitt für Variable im RAM
name	anzahl	reserviert Speicherplatz für anzahl 8bit Variable
name	anzahl	reserviert Speicherplatz für anzahl 16bit-Variable
name	anzahl	reserviert Speicherplatz für anzahl 32bit-Variable

Diese Deklarationen müssen in einer eigenen SECTION stehen, damit die Variablen im RAM platziert werden.

Falls **anzahl** > 1 ist, beschreibt **name** die Adresse des ersten Bytes des reservierten Speicherplatzes.

- **Lokale Variable** siehe Vorlesung Kapitel 4

- **Konstanten**

.const		Speicherabschnitt für Konstante im ROM
name	wert	definiert und initialisiert eine 8bit-Konstante
name	wert	definiert und initialisiert eine 16bit-Konstante
name	wert	definiert und initialisiert eine 32bit-Konstante

Auf diese Weise können auch Textstrings definiert werden, z.B. `Txt: DC.B "Say yes", 0.`

Mehrere Werte können (mit demselben Label) auch hintereinander angegeben werden, z.B. `array: DC.W 0,1,2,3`

Ein größerer Konstanten-Datenblock kann folgendermassen definiert und initialisiert werden:

name anzahl wert

Dabei werden **anzahl** Byte reserviert und jeweils mit **wert** initialisiert. Statt **DCB.B** kann auch **DCB.W** oder **DCB.L** zum Deklarieren von 16bit oder 32bit-Konstantblöcken verwendet werden.

- **Exportieren und Importieren von Funktionen und Variablen**

name1, name2, ...	Importieren von Funktionen (und 16bit Variablen)
name	Importieren von 8bit Variablen
name	Importieren von 16bit Variablen
name	Importieren von 32bit Variablen

Um auf außerhalb des aktuellen Programmmoduls definierte Variablen oder Funktionen zugreifen zu können, müssen diese mit **XREF** importiert werden (entspricht **extern** in C). Bei Variablen ist dabei eine Größenangabe mit **.B**, **.W** (Default) bzw. **.L** sinnvoll. In jedem Fall notwendig ist die Angabe von **XREF __SEG_END_SSTACK**

name1, name2, ...	Exportieren von Funktionen und Variablen
--------------------------	--

Um Funktionen oder Variable für andere Programmmodule **und den Linker** zugänglich zu machen, müssen diese exportiert werden. In jedem Fall ist dies für den Einsprungpunkt eines Programms notwendig, also in der Regel **XDEF Entry, main**. Auch hier sind nötigenfalls wieder Größenangaben wie **XDEF.B** usw. möglich.

Seltener gebrauchte Direktiven und anderes

- **Fiktiver Adresszähler beim Assemblieren: Current Location Counter CLC**

Während des Assemblierens verwaltet der Assembler einen fiktiven Adresszähler, den Current Location Counter, der auf die fiktive Speicheradresse des übersetzten Programmcodes zeigt. Im Programmlisting bzw. im Disassembler-Fenster des Debuggers erscheint der CLC als „*“ vor allem bei Sprungbefehlen mit relativer Adressierung, z.B. **BRA *-12** ← Sprung um -12 Byte von der aktuellen Adresse aus rückwärts.

- **Vorgabe absoluter Adressen**

adresse

Üblicherweise erfolgt die Zuordnung des Programmcodes und der Daten zu den Speicheradressen durch den Linker/Locator automatisch. Man kann jedoch vor der Definition von Variablen oder Programmcode durch **ORG adresse** deren absolute Speicheradresse vorgeben und damit den CLC auf die angegebene **adresse** setzen. **ORG** wird z.B. im Zusammenhang mit der Interrupt-Vektor-Tabelle (siehe Vorlesung Kapitel 3) verwendet.

- **Bedingte Assemblierung**

Ähnlich wie in C mit **#if**, **#ifdef** ... lässt sich die Übersetzung bestimmter Programmteile auch in Assembler in Abhängigkeit von der Definition bestimmter Symbole, von deren Wert oder von anderen Bedingungen steuern. Zu Einzelheiten siehe [3.14, Abschnitt Assembler Directives – **IF** und **IFcc**].

- **Makros**

Name

. . .

Ähnlich wie Funktionsmakros in C mit **#define** oder in C++ mit Inline-Funktionen lassen sich auch in Assembler Deklarationen oder Programmabschnitte, die mehrfach benötigt werden, zusammenfassen und der Einsetzvorgang beim Assemblieren automatisieren. Solche Makros sind (stark eingeschränkt) beim Übersetzen auch parametrierbar. Zu Details siehe [3.14, Abschnitt Macros].

- **Rücksprung zum Monitor-Programm**

Üblicherweise enden Programme in Embedded-Systemen niemals, sondern laufen bis zum Abschalten der Spannungsversorgung in einer Endlosschleife. Die **startup()**-Funktion des CodeWarrior-C-Compilers z.B. ruft **main()** einfach erneut auf, falls **main()** endet. Bei Assembler-Programmen muss diese Endlosschleife in jedem Fall selbst programmiert werden, sonst führt die CPU am Programmende undefinierten Programmcode aus.

Da auf dem Dragon12-Entwicklungsboard ein Monitor-Programm als Default-Programm installiert ist, dürfen Programme dort ausnahmsweise enden, wenn man durch den Assemblerbefehl **SWI** einen „geordneten“ Rücksprung zu diesem Monitor-Programm bewirkt. Man sollte dies im Regelfall aber vermeiden, da diese Möglichkeit in anderen Embedded Systemen meist nicht besteht.

6. Debugger (siehe [3.15, 3.16])

Source-Code Start F5 Step In F11 Step Over F10 Step Out F11 Step ASM CTRL+F11 Halt F6 Reset CTRL+R Disassemblierter Code

Variable CPU-Register Speicherinhalt (erste Spalte: Adresse, andere Spalten: Daten in hex und ASCII)

- Zum Testen von Programmen wird der Debugger aus der CodeWarrior-Entwicklungsumgebung mit dem Icon *Debug* bzw. Menüpunkt *Project – Debug (F5)* gestartet. Dabei muss zuvor das *Target* gewählt werden (siehe Abschnitt 4.3 und 4.4). Mit dem *Target Simulator* können Programme (teilweise) ohne reale Hardware überprüft werden. Leider wird dabei nur ein kleiner Teil der Hardware simuliert, so dass z.B. die Ansteuerung des realen LCD-Displays oder der Sieben-Segment-Anzeigen, die auf dem Dragon12-Board zu finden sind, mit dem Simulator nicht oder nur unvollständig getestet werden können.
- Falls *Monitor* als *Target* gewählt wurde, muss das Dragon12-Entwicklungsboard eingeschaltet und über die serielle Schnittstelle SCIO mit einer der seriellen Schnittstelle COMx: des Host-PCs verbunden sein. Das Anwendungsprogramm wird dann automatisch in den Speicher des Entwicklungsboards geladen. Eventuell ist es notwendig, das Entwicklungsboard vorher durch kurzen Druck auf die Reset-Taste SW6 (im Bild auf S. 25 unten in der Mitte) zurückzusetzen. Bei Problemen siehe 6.1.
- Im Debugger-Betrieb mit dem Dragon12-Entwicklungsboard kann die serielle Schnittstelle SCIO nicht vom Anwendungsprogramm benutzt werden. Falls das Anwendungsprogramm eine serielle Schnittstelle benötigt, verwenden Sie dafür SCI1 (siehe auch Bild S. 3)

6.1 Probleme beim Anschluss des Dragon12-Entwicklungsboards

der Debugger (bei *Target Monitor*) aufnehmen kann, gehen Sie wie folgt vor:

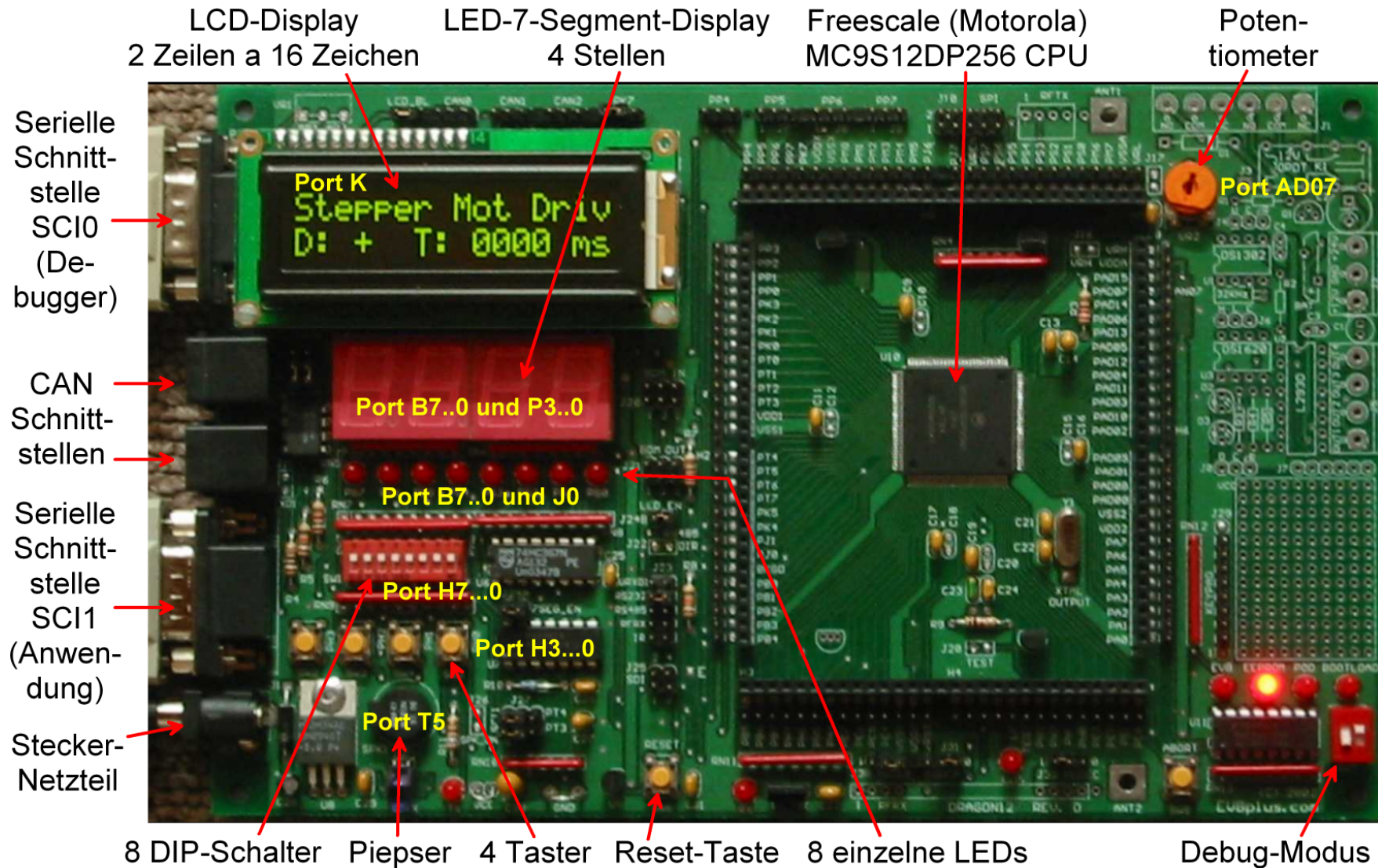
- Prüfen Sie, ob das Dragon12-Board mit Spannung versorgt wird. Die Leuchtdiode EVB rechts unten im Bild auf S. 25 muss leuchten. Beide DIL-Schiebeschalter SW7 müssen in Stellung ON (in Richtung zur Platinenunterkante) stehen (Evaluation Board EVB-Modus).

- Prüfen Sie die Kabelverbindung zwischen der seriellen Schnittstelle COMx: , x=1,2,... des PCs mit der seriellen Schnittstelle SCIO des Mikrocontrollers.
- Setzen Sie das Dragon12-Board durch kurzen Druck auf die Reset-Taste SW6 zurück.
- Wählen Sie im Debugger Menü *Component – Set Target – Processor: HC12 – Target Interface: GDI Target Interface*.
- Wählen Sie im Debugger Menü *GDI – Connect – GDI Driver DLL: hcs12serialmon.dll*. Die Datei **hcs12serialmon.dll** liegt im Unterverzeichnis **prog** der CodeWarrior-Installation.
- Stellen Sie im Debugger Menü *GDI – Monitor Connect – Monitor Communication – Host Serial Communication Port: COMx* ein. Wählen Sie aus der Liste diejenige serielle PC-Schnittstelle COMx aus, an die das Verbindungskabel zum Dragon12-Board angeschlossen ist.
- Beim Verbinden fragt der Debugger die sogenannte Part-ID und die MCU-ID des Mikrocontrollers ab, mit der er die verschiedenen Prozessorvarianten (Derivative) unterscheiden und sich darauf einstellen kann. Falls die automatische Erkennung nicht funktioniert, wird die Dialogbox *Set Derivative* angezeigt. Wir verwenden im Labor leider Boards mit leicht unterschiedlichen CPU-Derivaten, für die meisten CPUs ist die Part-ID **0x0012** und die MCU-ID **0x3C6**.

Erzwingt man den Betrieb mit einer falschen Kombination von Part-ID und MCU-ID, d.h. einer falschen Prozessorvariante, scheint der Programmdownload und das Starten des Anwendungsprogramms eventuell trotzdem zu funktionieren, das Programm wird jedoch u.U. nicht korrekt ablaufen, wenn die Speicherzuordnung falsch ist, und das Setzen von Haltepunkten (Breakpoints) oder der Einzelschrittbetrieb schlagen fehl.

Debugger

Dragon12-Entwicklungsboard



6.2 Programmtest mit dem Debugger

- **Laden eines Programms**

Beim Starten des Debuggers aus der CodeWarrior-Entwicklungsumgebung (Taste F5 oder Icon Debug) wird das Programm automatisch in den Speicher des Mikrocontrollers geladen, das Quellprogramm im Source-Fenster angezeigt und der Programmzähler PC auf den ersten Programmbefehl gesetzt (Bild S.22):

Assemblerprogramme beginnen bei der Marke **main**:

C-Programme beginnen mit der Initialisierungsfunktion **startup()** der C-Laufzeitumgebung. Erst von dort aus wird **main()** als Unterprogramm über einen Funktionszeiger als ***_startupData.main()** aufgerufen. Verwenden Sie *Run to Cursor* (siehe unten), um direkt zum Anfang von **main()** zu gelangen.

Im Assembly-Fenster wird der disassemblierte Programmcode angezeigt, im Registerfenster der aktuelle Inhalt der CPU-Register.

- **Programmänderungen**

Bei Programmänderungen muss das Programm in der CodeWarrior-Entwicklungsumgebung editiert, neu übersetzt und von dort der Debugger erneut gestartet werden.

- **Starten, Anhalten von Programmen, Reset**

Ein geladenes Programm wird ab dem aktuellen Programmzählerstand durch *Run – Start/Continue (F5)* gestartet oder fortgesetzt. Es läuft, bis ein Haltepunkt (siehe unten) erreicht oder das Programm durch *Run – Halt (F6)* wieder angehalten wird. Durch *Simulator* bzw. *Monitor – Reset Target (CTRL+R)* kann die CPU jederzeit zurückgesetzt werden. Falls

dies nicht funktioniert, muss gegebenenfalls zusätzlich die Reset-Taste SW6 auf dem Dragon12-Board kurz gedrückt werden.

- **Setzen und Löschen von Haltepunkten (Breakpoints)**

Das Programm kann an vordefinierten Haltepunkten (Breakpoints) automatisch angehalten werden. Die Haltepunkte müssen vor dem Starten des Programms festgelegt werden: Gewünschte Programmstelle im Source- oder Assembly-Fenster mit *RCLICK* anwählen – *Set Breakpoint*. Haltepunkte werden in der Anzeige durch einen roten Pfeil markiert.

Im Simulator können beliebig viele Breakpoints definiert werden, in der realen CPU dagegen sind gleichzeitig nur 2 Breakpoints möglich, da zur Speicherung der Breakpoints nur 2 Register in der CPU zur Verfügung stehen.

Breakpoints können über *RCLICK – Show breakpoints* angezeigt und über *RCLICK – Delete breakpoint* gelöscht werden.

Zusätzlich können bei Haltepunkten Bedingungen angegeben oder Speicherbereiche (Watchpoints) überwacht werden, zu Einzelheiten siehe [3.15, 3.16].

- **Einzelsschrittbetrieb – Run to cursor**

Zeigt man mit dem Cursor auf einen Befehl im Source- oder Assembly-Fenster und wählt *RCLICK – Run to cursor*, setzt der Debugger einen temporären Haltepunkt auf diese Stelle und startet das Programm. Achten Sie dabei darauf, dass diese Stelle im Programmablauf auch tatsächlich erreicht wird.

Mit *Run – Single Step (F11)* bzw. *Run – Step Over (F10)* kann man (ohne Breakpoints) einen einzelnen Befehl ausführen lassen. Bei *Step Over* wird ein Unterprogrammaufruf dabei als ein einzelner Schritt ausgeführt, bei *Single Step* kann auch das Unterprogramm schrittweise durch-

laufen werden. Mit *Run – Step Out (Shift+F11)* kann man ein Unterprogramm bis zu dessen Ende durchlaufen.

In einem C-Programm wird unter einem Schritt ein C-Befehl verstanden, der typischerweise aus mehreren Assembler-Befehlen besteht. Will man lediglich einen einzelnen Assembler-Befehl in einem C-Programm ausführen, muss man *Run – Assembly Step (CTRL+F11)*, *Run – Assembly Step Over (CTRL+F10)* bzw. *Run – Assembly Step Out (CTRL+Shift+F11)* wählen.

- **Assembly-Fenster**

Die Anzeige im Assembly-Fenster kann mit *RCLICK – Display ...* verändert werden. Mit *Symbolic* werden Peripherieregister und Variablen symbolisch statt als Hexadezimaladressen angezeigt, mit *Address* und *Code* kann die Anzeige der Programmadresse bzw. des Objektcodes aktiviert werden. Mit *RCLICK – Show location* kann man im Source- oder Assembly-Fenster die jeweils zugehörige Stelle im anderen Fenster finden.

Mit *RCLICK – Address* kann man eine Programmadresse angeben, für die der Programmcode angezeigt werden soll.

- **Register-Fenster: Ändern von Registerinhalten**

Durch *DCLICK* auf eines der Register und Eingabe eines Hexadezimalwertes kann man den Inhalt eines CPU-Registers ändern.

Ändert man den Inhalt von PC, kann man eine Programmverzweigung simulieren und ein Programm an einer anderen Stelle fortsetzen.

- **Messen von Programmausführungszeiten**

Im Register-Fenster wird die Anzahl der CPU-Taktzyklen seit dem letzten Reset angezeigt. Wenn man das Programm im Single-Step-Betrieb oder bis zum nächsten Breakpoint laufen

lässt, kann man aus der Differenz des aktuellen Wertes und des Wertes vor dem Programmstart die Befehls- oder Programmausführungszeit in CPU-Taktzyklen bestimmen. Beim Dragon12-Entwicklungsboard beträgt die CPU-interne Taktfrequenz 24MHz (externer 4MHz Quartz, durch interne PLL versechsfacht).

- **Data-Fenster**

Im Datenfenster werden globale und/oder lokale Variable angezeigt. Das Anzeigeformat (binär, dezimal, hexadezimal usw.) lässt sich über *RCLICK – Format ...* einstellen.

Mit *DCLICK* auf einen Wert und Eingabe eines neuen Wertes kann man eine Variable (im RAM) verändern. Konstanten im Flash-ROM werden nicht verändert. Der geänderte Wert wird in der Regel beim Neustart des Programms wieder überschrieben. Der Wert im Originalprogramm wird nicht geändert!

Mit *RCLICK – Show Location* wird der zugehörige Speicherbereich auch im Memory-Fenster angezeigt.

- **Memory-Fenster**

Mit *DCLICK* auf einen Wert und Eingabe einer neuen Zahl lässt sich der Speicherinhalt (im RAM) ändern.

Mit *RCLICK – Format ...* und *RCLICK – Word Size* kann das Anzeigeformat umgestellt werden.

Mit *RCLICK – Address* kann ein beliebiger Adressbereich ausgewählt werden.

Mit *RCLICK – Fill* bzw. *RCLICK – Copymem* kann ein Speicherbereich initialisiert bzw. kopiert werden.

6.3 Simulation von Peripheriebausteinen im Target Simulator

Für den Programmtest ohne Hardware können einige Peripheriekomponenten (*Components*) des Mikrocontrollers bzw. des Dragon12-Boards simuliert werden. Leider steht nur eine begrenzte Zahl von Komponenten zur Verfügung, die sich zudem teilweise anders verhalten als die realen Komponenten auf dem Dragon12-Board.

Eine Komponente wird mit *Component – Open* aus einer Liste ausgewählt, mit *DCLICK* geöffnet und danach in der Regel mit *RCLICK – Setup* konfiguriert. Damit die Komponente beim nächsten Start des Debuggers wieder automatisch geöffnet und konfiguriert wird, sollte man die aktuellen Einstellungen mit *File – Save Configuration* speichern.

Hilfe zu den Komponenten finden Sie in der Online Hilfe des Debuggers unter *Help – Help Topics – Debugger Engine – Framework Components*.

- **Komponente IO_Led:** Simuliert 8 LEDs

Mit dieser Komponente können 8 beliebige Digitalausgänge des Prozessors visualisiert werden. Um die 8 LEDs am Port B der CPU auf dem Dragon12-Board zu simulieren, ist folgende Setup-Einstellung notwendig:

PORT: 1 (hexadezimale Adresse des Datenregisters PORTB)
DDR: 3 (hexadezimale Adresse des Datenrichtungsregisters DDRB)
Target: TargetObject (Simulator-interne Schnittstelle für Komponenten)

Die auf dem Dragon12-Board zusätzlich notwendig Freigabe der LEDs durch Port J0 wird nicht nachgebildet. Achtung: Verwenden Sie die Komponente IO_Led, nicht die Komponente Led!

Anwendungsbeispiel: CodeWarrior-Projekt **BlinkingLed.mcp**.

- **Komponente Push_Buttons:** Simuliert 8 Drucktasten (keine Schalter!)

Mit dieser Komponente können die 4 Taster am Port H auf dem Dragon12-Board simuliert werden. Dazu ist folgende Setup-Einstellung notwendig:

Ports address: 260 (hexadezimal Adresse des Ports PTH)

Berücksichtigen Sie bitte, dass die Wirkungsrichtung der Taster umgekehrt ist wie bei den echten Tastern auf dem Dragon12-Entwicklungsboard:

Wenn Sie beim Dragon12-Board einen der Taster SW2 ... SW5 drücken, liegt am entsprechenden Port eine ‚0‘ an, während beim Drücken einer Taste der Komponente Push_Button eine ‚1‘ geliefert wird. Die Logik bei Abfragen muss also beim Wechsel zwischen Simulator und Dragon12 invertiert werden.

Außerdem löst das Betätigen des Tasters im Simulator keinen Interrupt aus, eignet sich also *nicht* für den Test der Interrupt-Funktionen des Port H. Siehe dazu die Komponente IT_Keyboard.

Anwendungsbeispiel: CodeWarrior-Projekt **Lauflicht.mcp**.

- **Komponente IT_Keyboard:** Simuliert Drucktasten mit Interrupt

Mit dieser Komponente können die 4 Taster am Port H auf dem Dragon12-Board simuliert werden, wenn das Interrupt-Verhalten des Ports H getestet werden soll. Dazu ist folgende Setup-Einstellung notwendig:

Ports address: Lines: 262 (hexadezimale Adresse von DDRH)

Columns: 260 (hexadezimale Adresse von PTH)

Interrupt vector: 19 (19_H = 25_D Interruptnummer von Port H)

Auch diese Simulations-Komponente verhält sich „gewöhnungsbedürftig“. Solange eine „Taste“ mit der Maus gedrückt wird, wird ständig der Interrupt ausgelöst, unabhängig davon, ob der Interrupt in PIEH freigegeben wurde oder nicht. Die Komponente darf also nicht verwendet werden, wenn keine ISR für den Interrupt definiert wurde!

Die Tasten in der Spalte 0,4,8,C liegen alle an Port H.0, die Tasten der Spalte 1,5,9,D an Port H.1, die Tasten 2,6,A,E an H.2 und 3,7,B,F an H.3. Ein Tastendruck entspricht wie auf dem Dragon12-Entwicklungsboard einer logischen ‚0‘ am Port-Pin.

Anwendungsbeispiel: CodeWarrior-Projekt **ButtonInterrupt.mcp**, **ButtonInterruptAsm.mcp**

- **Komponente Terminal:** Simuliert ein Terminal an der seriellen Schnittstelle SCIO

Im wesentlichen wird nur das Sende- und Empfangsregister simuliert, d.h. im Simulator funktioniert die Schnittstelle im Gegensatz zur Hardware auch dann, wenn die Peripherieregister der Schnittstelle, z.B. die Baudrate, falsch konfiguriert wurden.

Testet man auf dem Dragon12-Entwicklungsboard statt mit dem Simulator, muss man im Anwendungsprogramm die serielle Schnittstelle SCI1 des Mikrocontrollers verwenden und diese mit einer weiteren seriellen Schnittstelle COMy: des PCs verbinden. Die Schnittstelle SCIO des Mikrocontrollers und die serielle Schnittstelle COMx: des PCs werden für den Debugger benötigt. Die Komponente Terminal kann wieder als Terminal für Programmein- und ausgaben benutzt werden, muss aber folgendermassen konfiguriert werden:

RCLICK innerhalb Terminal – *Configure Port* – *Use Serial Port* anwählen, *Redirect ...* nicht anwählen – *Communication Device*: COMy – *Baud Rate*: 115200 (bzw. je nach verwendeter Baudrate des HCS12).

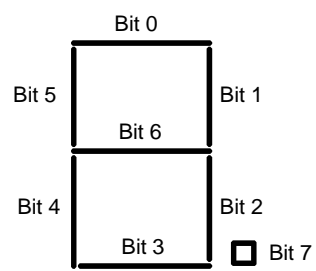
Anwendungsbeispiel: CodeWarrior-Projekt **SerialPolling.mcp** und **SerialInterrupt.mcp**.

- **Komponente Segments_Display:** Simuliert eine mehrstellige 7-Segment-Anzeige
Mit dieser Komponente kann notfalls die vierstellige 7-Segment-Anzeige des Dragon12-Boards an Port B und P nachgebildet werden. Wählen Sie folgende Setup-Einstellungen:

Select a display: 258 (hexadezimale Adresse von Port P)

Segments activation: 1 (hexadezimale Adresse von Port B)

Die Ansteuersignale für die simulierte Komponente unterscheiden sich allerdings von den Signalen auf dem Dragon12-Entwicklungsboard:

	<i>Dragon12</i>	<i>Simulation</i>
Ansteuerung der Segmente	Segmentwert in Port B schreiben, '1' aktiviert Segment 	
Aktivierung der 1er Stelle	Port P.0 = 0	Port P.3 = 1
Aktivierung der 10er Stelle	Port P.1 = 0	Port P.2 = 1
Aktivierung der 100er Stelle	Port P.2 = 0	Port P.1 = 1
Aktivierung der 1000er Stelle	Port P.3 = 0	Port P.0 = 1
Alle Stellen aus	Port P.3...0 = 1111 _B	Port P.3...0 = 0000 _B

D.h. die Signale an Port P sind logisch zu invertieren und die Reihenfolge zu vertauschen.

Anwendungsbeispiel: CodeWarrior-Projekt **SevenSegmentDisplay.mcp**.

- Die **Komponente LCD** auf dem Dragon12-Entwicklungsboard verwendet leider eine andere Hardwarechnittstelle als die LCD Komponente im Simulator. Das Treiber-Programm **LCD.asm** aus dem Laborversuch 1 kapselt die Unterschiede und kann daher sowohl mit dem Dragon12-Board als auch mit dem Simulator verwendet werden. Im Simulator ist die LCD-Komponente folgendermassen zu konfigurieren: RCLICK auf die Komponente – *Setup – Ports Address - Data: 32 – Control: 0*. Damit der Code für den Simulator aktiviert wird, muss am Anfang der Datei **LCD.asm** das folgende Symbol definiert werden:

SIMULATOR: EQU 1

Für den Betrieb mit dem Dragon12-Board muss diese Zeile auskommentiert sein. Alternativ kann das Symbol auch im Target Option Dialog (siehe S. 10) für das Target *Simulator* als

-DSIMULATOR

definiert werden.

- Die Komponente **Visualization Tool** erlaubt die Darstellung von Zeitdiagrammen (*Chart*) ähnlich einem Oszilloskop. Nach dem Öffnen des Visualization Tool Fensters RCLICK – *Add New Instrument – Chart*, dann mit LCLICK positionieren und durch Ziehen die gewünschte Höhe und Breite des Diagramms einstellen. Die Konfiguration eines Zeitdiagramms ist relativ unübersichtlich:

Einstellung des dargestellten Signals und der horizontalen und vertikalen Achse: DCLICK innerhalb des Charts:

- *Kind of Port: Expression – Port to Display: (PTP & 0x80) >> 7*, wählt das dargestellte Signal aus, hier z.B. Bit 7 des Ports P, mit PTP würde man den gesamten Port P als 8 bit Wert darstellen.

- *High Display Value: y_{\max} - Low Display Value: y_{\min}* , stellt den dargestellten Zahlenbereich $y_{\min} \dots y_{\max}$ ein, z.B. für 1bit Signal $y_{\max}=1$ $y_{\min}=0$, für 8bit Betragszahl: $y_{\max}=255$ $y_{\min}=0$ - usw. – *Vertical Index Step: Δy* (Skalierung, Größe eines Teilstrichs der Anzeige)
- *Type of Unit: Target Periodical – Unit size: n - Number of Units: N* , stellt den Zeitbereich $0 \dots n \cdot N$ ein (Einheit für alle Zeitangaben ist 1 CPU-Taktzyklus, d.h. beim Dragon12-Board $1/24\text{MHz}=42\text{ns}$) – *Horizontal Index Step: m* , stellt die Größe eines angezeigten Teilstrichs $\Delta t=n \cdot m$ ein. Bsp.: $n=24$, $N=200$, $m=10 \rightarrow$ Zeitbereich $0 \dots 24 \cdot 200/24\text{MHz} = 200\mu\text{s}$ mit $24 \cdot 10/24\text{MHz}=10\mu\text{s/div}$

Einstellung der Auffrischrate des Diagramms: DCLICK im Visualization Tool Fenster außerhalb des Charts – *Refresh Mode: CPU Cycles – Cycle Refresh Count: 100000* (Einheit: CPU-Taktzyklen), andere Zahlenwerte nach Bedarf.

Anwendungsbeispiel: CodeWarrior-Projekt **PWM1.mcp**

- Die **Komponente ADC** funktioniert leider deutlich anders als die entsprechende Baugruppen auf dem Dragon12-Entwicklungsboard und kann für die Simulation nicht sinnvoll verwendet werden.