DEV12 Tutorial

22. Oktober 2003

DEV12 Tutorial

Dieses Tutorial soll anhand einiger praktischer Beispiele zeigen, wie die einzelnen Werkzeugkomponenten der DEV12 Toolchain bei der Softwareentwicklung zusammenspielen.

Bei den Bestandteilen des DEV12 Pakets handelt es sich um ausgewählte, kostengünstige Einzelkomponenten, im Gegensatz zu den sonst marktüblichen hochpreisigen, monolithischen Lösungen. Das Hauptproblem der großen, monolithischen Pakete ist deren Komplexität und - damit einhergehend - gesteigerte Fehleranfälligkeit. Der Versuch, Funktionsabläufe der Entwicklungswerkzeuge zu kapseln und die Details möglichst vom Benutzer fernzuhalten, birgt die Gefahr, den Anwender zu "entmündigen" und ihn beim Auftreten von Problemen im Unklaren über deren Ursache zu lassen. Außerdem läßt sich die Amortisation der Anschaffung hochpreisiger, integrierter Toolchains, sowie deren Kosten für Einarbeitung und Wartung, meist nur mit intensivstem Einsatz in einer möglichst großen Zahl von Projekten erreichen.

Die Komponenten des DEV12 Pakets hingegen sind durch ihre rasche Beherrschbarkeit und die geringen Anschaffungskosten bereits nach kürzester Zeit - oft schon nach dem ersten Projekt - bezahlt.

Das Tutorial zeigt, wie ein C-Programm übersetzt, in das Zielsystem geladen und mit dem Debugger untersucht werden kann. Besonderes Augenmerk wird auf die speziellen Anforderungen an Software für Embedded Systeme und die Interaktion der Tools untereinander gelegt.

Die im Tutorial verwendeten Quelltextdateien finden Sie im Internet auf der Elektronikladen Website:

```
http://www.elektronikladen.de/files/manuals/dev12_src.zip
```

Übersicht über die eingesetzten Tools und Komponenten

In diesem Tutorial finden folgende, auf 32-Bit Windows-Plattformen lauffähige Softwaretools Verwendung:

ICC12 ANSI-C Crosscompiler (V6.16 Standard)

Der Compiler besitzt eine eigene Oberfläche (IDE), welche mit Editor, Projektverwaltung und zahlreichen Zusatzfunktionen ausgestattet ist. Eine Demoversion kann von der Imagecraft Website geladen werden, sie ist 30 Tage ab Erstinstallation ohne weitere Einschränkungen lauffähig:

```
http://www.imagecraft.com/software
```

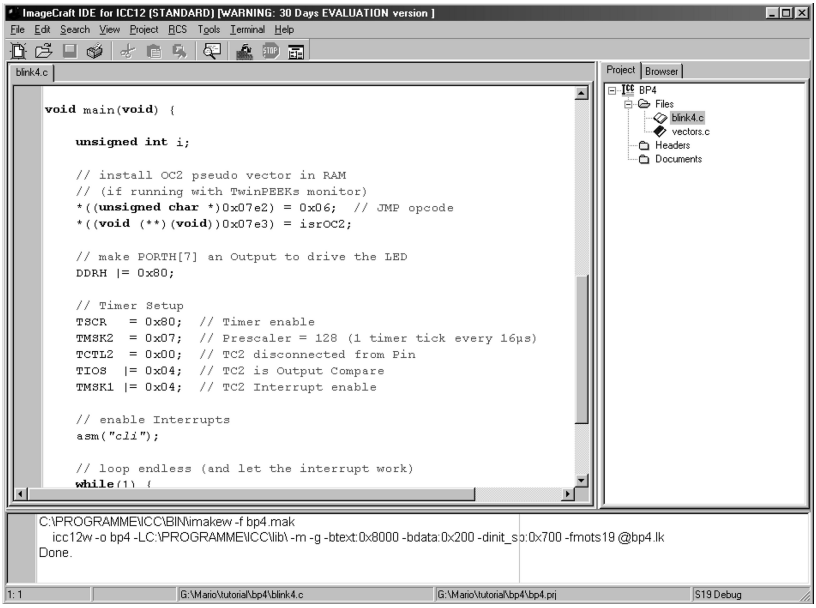

Abb.1: Die Oberfläche des ANSI-C Compilers ICC12

Terminalprogramm OC-Console

Dieses einfache, aber überaus nützliche Terminalprogramm kann kostenlos von der Elektronikladen Website geladen werde:

```
http://www.elektronikladen.de/occonsole.html
```

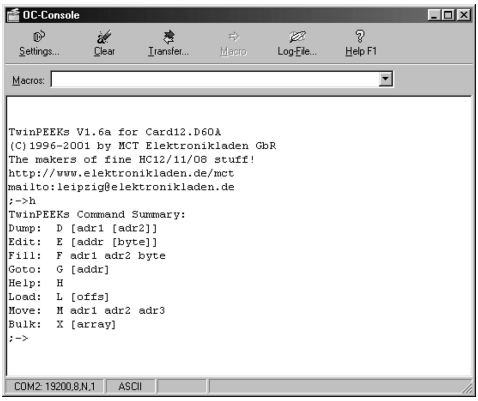

Abb.2: Das Terminalprogramm OC-Console

StarProg In-System Programmer (V2.0)

StarProg ist ein ein Download-Utility zur Programmierung von EEPROM und Flashspeicher der HC(S)12 Mikrocontroller. Die Verbindung zum Zielsystem erfolgt via BDM. Hierzu ist das ComPOD12 BDM-Interface erforderlich (s.u.).

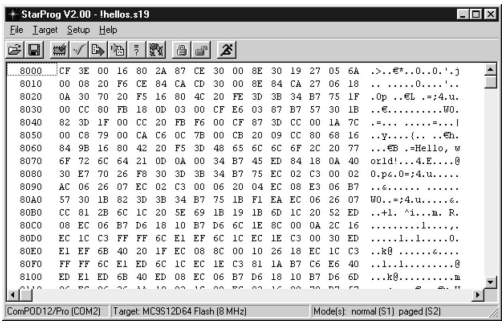

Abb.3: StarProg ist für den Flash-Download zuständig

NoICE12 Remote Debugger (V7.7)

NoICE12 ist ein einfach anzuwendender grafischer Source-Level Debugger, welcher über die serielle Schnittstelle oder über BDM mit dem HC12 Zielsystem verbunden werden kann. Eine Testversion der Debuggersoftware ist von der NoICE Debugger Website erhältlich:

```
http://www.noicedebugger.com
```

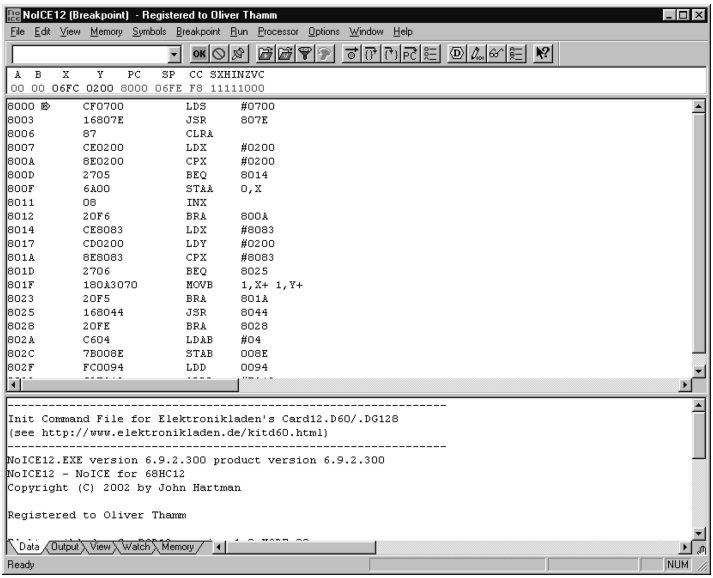

Abb.4: NoICE12 Remote Debugger

Außerdem kommen folgende Hardwarekomponenten zum Einsatz:

ComPod12 BDM-Interface (Firmware V1.10)

Das BDM-Pod stellt die Verbindung zwischen BDM12-Anschluß am Controllerboard und der seriellen Schnittstelle am PC her. Weitere Informationen zu ComPod12 siehe:

```
http://www.elektronikladen.de/compod12.html
```

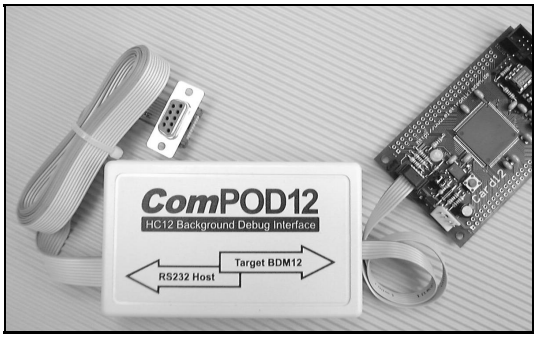

Abb.5: ComPod12, angeschlossen an ein Controllermodul

CardS12.D64 Controller Modul (Hardware Rev. 1.00)

Das scheckkartengroße CardS12 Controllermodul trägt eine HCS12 MCU vom Typ MC9S12D64. Informationen hierzu siehe:

```
http://www.elektronikladen.de/cards12.html
```

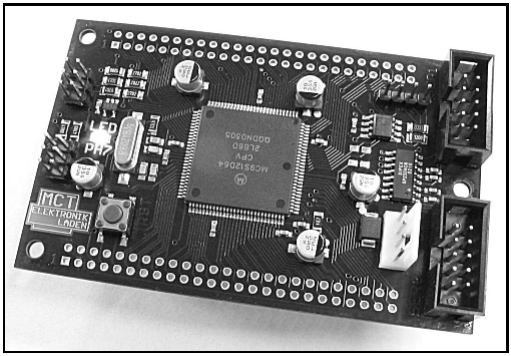

Abb.6: CardS12.D64 Controllermodul

1. Arbeiten mit C-Compiler und Monitor

1.1. Hello World - in der Sprache der Leuchtdioden

Es gibt wohl kaum ein Stück Programmquelltext, das häufiger abgetippt und reproduziert wurde als "hello.c". Es stammt aus dem Standardwerk "Programmieren in C" von Kernighan/Ritchie [1] und dient seither als "das erste Programm" zum Einstieg in die C-Programmierung. Hier ist sie - die Mutter aller C-Programme:

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

Dieser kurze Quelltext enthält alle für ein C-Programm notwendigen syntaktischen Elemente. Er sollte sich mit jedem beliebigen (ANSI-) C-Compiler übersetzen lassen. Je nach Compilereinstellung könnte es sein, dass noch Warnungen zustande kommen, z.B. "Missing Return Value" in der letzten Zeile der Funktion main(). Alles in allem sollte die Übersetzung dieses Programmes aber erfolgreich möglich sein.

Probieren Sie es doch einfach mal aus! Den Quelltext können Sie mit dem C-Compiler ICC12 übersetzen, indem Sie Strg-F9 drücken oder "File/Compile_File/To_Output" aus dem Menü wählen. Das erzeugte Codefile heisst "hello.s19".

Obwohl das Beispiel hello.c sehr einfach aussieht, ist es für den ersten Test eines Mikrocontrollers eigentlich gar nicht recht geeignet. An einem Mikrocontrollerboard ist in der Regel gar kein Bildschirm angeschlossen, demzufolge kann "hello, world" auch nicht auf der

Mattscheibe erscheinen. Man muß sich schon eines anderen, geeigneteren Ausgabekanals bedienen.

Fast alle Mikrocontroller verfügen über die Fähigkeit, über eine asynchrone serielle Schnittstelle zu kommunizieren. Man kann diese Schnittstelle mit dem COM-Port eines PC verbinden und hat eine einfache Möglichkeit, Mitteilungen und Ausschriften des Mikrocontrollers auf dem PC-Bildschirm zu beobachten. In der entgegengesetzten Richtung werden die Eingaben von der PC-Tastatur an den Mikrocontroller weitergeleitet. Neben der normalen Hard- und Software (Betriebssystem) des PC benötigt man lediglich ein so genanntes Terminalprogramm, welches Bildschirm, Tastatur und COM-Port miteinander verknüpft.

Das einfache Prinzip der seriellen Kommunikation erweist sich in der Praxis allerdings manchmal als etwas verzwickt, da es eine Vielzahl von Anschlußvarianten und Softwareoptionen gibt. Aus der Sicht eines Anfängers scheinen die möglichen Kombinationen aus COM-Port, Kabelverbindung, Baudrate, Parität und Protokoll schier unendlich zu sein. Und, zugegeben, auch der erfahrene Anwender verirrt sich zuweilen im RS232-Dschungel. Das führt leicht dazu, daß ein Programmtest mißlingt und die Ursache im Programm selbst gesucht wird. In Wirklichkeit hapert es jedoch nur an der gestörten Beobachtbarkeit - das Programm schickt vielleicht die erwartete Ausschrift ganz korrekt, aber die Daten kommen auf dem Weg zum Terminalfenster unterwegs abhanden.

Aus diesem Grund erscheint es sinnvoll, für das allererste Mikrocontrollerprogramm einen Ausgabekanal zu verwenden, der gleich auf Anhieb funktioniert. Ohne langes Nachdenken über die richtigen Initialisierungsschritte. Was wäre dafür geeigneter als eine Leuchtdiode?!

Auf dem Mikrocontrollerboard CardS12 ist eine solche Leuchtdiode vorhanden. Sie ist an der Leitung PH7 des Mikrocontrollers angeschlossen, d.h. am obersten Bit des Port H. Wir lassen diese LED nun blinken und verwenden dafür das folgende Testprogramm "blink1.c":

```
#include <hcs12dp256.h>

void delay(void) {
    volatile unsigned n = 0;
    while(--n) ;
    }

void main(void) {

    DDRH |= 0x80;
    while(1) {
        PTH ^= 0x80;
        delay();
        }
    }
```

1.2. Compilereinstellungen - und wie man das Resultat überprüft

Bevor Sie dieses Programm mit Ctrl-F9 übersetzen, sollten Sie zunächst einige Einstellungen des Compilers überprüfen (siehe Menüpunkt "Project/Options").

In der Rubrik "Target" müssen zunächst die Startadressen für den Programmcode (im Flash), die Daten (im RAM) und den Stackbereich (ebenfalls im RAM) festgelegt werden. Verwenden Sie hierzu die Einstellung "Custom" und tragen folgende Werte ein:

- Program Memory: 0x8000
- Data Memory: 0x3000
- Stack Pointer: 0x3E00

Kreuzen Sie "Enable" unter "Extended Memory" *nicht* an.

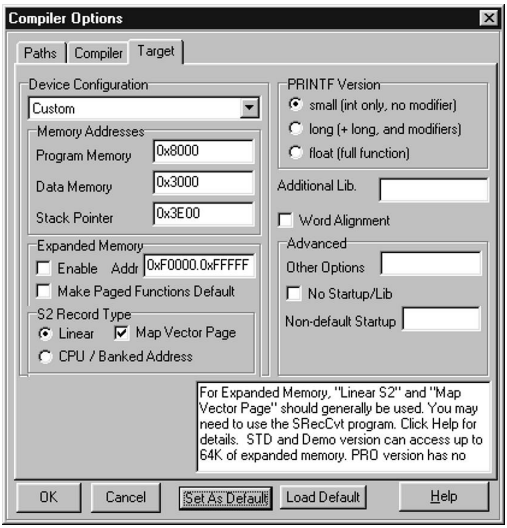

Abb.7: Einstellung der ICC12 Compileroptionen

Der Programmstart liegt üblicherweise innerhalb des internen Flashspeichers (hier: 0x8000), die Daten am Anfang (0x3000) und der Stack am Ende (0x3E00) des internen RAMs. Die Lage des RAMs ergibt sich aus den vom Monitorprogram TwinPEEKs vorgenommenen Initialisierungen.

Weitere Eintragungen sind in diesem Dialogfeld nicht erforderlich. Beachten Sie bitte, daß wir zu diesem Zeitpunkt davon ausgehen, daß *kein* Projekt geladen ist (zur Verwendung von Projekten später mehr).

Bei der Übersetzung erzeugt der Compiler neben dem eigentlichen Output File "blink1.s19" auch ein Map File namens "blink1.mp". In dieser Datei kann man die Adreßlage und die Größe des erzeugten Programms gut kontrollieren. Daher ist bei auftretenden Problemen ein Blick in's Map File stets sehr zu empfehlen.

Auch den Programmcode selbst kann man überprüfen, zumindest ein schneller Plausibilitätstest ist machbar. Das vom Compiler erzeugte Output File blink1.s19 ist eine Datei im Motorola S-Record Format. Da es sich hierbei um Textdateien handelt, kann man sie leicht mit einem

beliebigen Texteditor betrachten. Die erste Codezeile in blink1.s19 sieht etwa so aus (die Leerzeichen wurden nachträglich zur besseren Lesbarkeit eingefügt):

```
S1 0E 8000 CF3E0016805E87CE30008E 5D
```

S1 ist die Kennung für eine "Motorola S-Record Zeile mit 16 Bit Adresse". 0E ist die Anzahl der Bytes in der Zeile (14 Byte dezimal, darin enthalten 3 Byte für Adresse und Prüfsumme, die restlichen 11 Byte sind die eigentlichen Codebytes). 8000 ist die Startadresse, CF ist das erste von 11 Codebytes (ab Adresse 0x1000). Die Prüfsumme 5D steht am Schluß der Zeile.

Man erkennt also am Inhalt der S-Record Datei recht schnell, welche Adressbereiche mit Codebytes bzw. Daten gefüllt sind.

1.3. Wie kommt der Code in den Controller?

Den Compileroutput ausgiebig betrachten zu können, mag allein auf Dauer etwas unbefriedigend sein. Natürlich soll der Code auch in den Controller geladen werden. Der nächste Schritt heißt demzufolge:

Download

An die Innereien eines "fabrikneuen" HCS12 kommt man grundsätzlich nur via Background Debug Mode heran. Auch Downloads in den RAM oder die Programmierung des Flashspeichers erfolgen über diese Schnittstelle. Allerdings benötigt man für diesen Weg zusätzliche Tools. Weiter unten werden wir uns der BDM-Methode noch ausführlich zuwenden.

Es geht aber auch ohne BDM: Das Controllermodul CardS12 wird mit einem Monitorprogramm im Flash des HCS12 ausgeliefert, welches einen zusätzlichen Weg eröffnet, ein Anwenderprogramm in den Controller zu laden.

Verbindet man die serielle Schnittstelle SER0 des Controllerboards mit dem RS232 Anschluß eines PCs, kann man das Monitorprogramm (läuft auf dem HCS12) über ein Terminalprogramm (läuft auf dem PC) fernbedienen. Ein geeignetes, einfach zu handhabendes Terminalprogramm ist OC-Console von Uwe Altenburg, welches kostenlos von der Elektronikladen Website geladen werden kann.

Blick durch's (Terminal-) Fenster

OC-Console wird konfiguriert, indem Sie unter "Settings" die zutreffende PC-Schnittstelle und eine Baudrate von 19200 Bd auswählen. Die korrekten Übertragungsparameter "8N1" sind bereits voreingestellt. Unter "File Transfer" wählen Sie "Wait for Handshake" und tragen unter "Sequence" als Quittungszeichen * oder #42 ein (beides bedeutet Stern bzw. Multiplikationszeichen).

Das Quittungszeichen dient der Synchronisierung beim Download von S-Record Dateien. Das Terminal sendet eine Zeile der Datei und wartet dann zunächst auf das Quittungszeichen, bevor eine weitere Zeile gesendet wird. Die Gegenstelle (hier: das Monitorprogramm) quittiert, sobald die empfangene Zeile fertig verarbeitet wurde. Das Monitorprogramm kann durch diesen Mechanismus die Geschwindigkeit des Downloads steuern, um "in aller Ruhe" Flash- oder EEPROM-Zellen programmieren zu können.

Nach Anlegen der Versorgungsspannung (5 Volt Gleichspannung, natürlich stabilisiert!) sendet das Monitorprogramm eine kurze Begrüßungsmeldung und das Systemprompt (ein anderes Wort für Eingabeaufforderung). Daraufhin ist das Monitorprogramm bereit für die Befehle des Benutzers.

Umwandlung

Bevor der Code in den Controller geladen werden kann ist jedoch ein weiterer Zwischenschritt erforderlich. Dies liegt in der Architektur des im HCS12 integrierten Flashspeichers begründet: er kann nur wortweise beschrieben werden.

Da der C-Compiler allein zur korrekten Codegenerierung solche Besonderheiten des jeweiligen Zielcontrollers nicht unbedingt kennen muss, erzeugt er zunächst S19 Dateien, welche *nicht* aligned (an Wortgrenzen ausgerichtet) sind.

Beim Versuch, eine solche S19 Datei direkt mit dem TwinPEEKs Monitorprogramm zu laden, würde sich folgendes Fehlerszenario ergeben:

Der Monitor verarbeitet S-Record Daten stets zeilenweise. Falls die letzte belegte Adresse in einer solchen S-Record-Zeile gerade ist, fehlt zunächst das für die Wort-Programmierung erforderliche zweite Byte. TwinPEEKs ergänzt in dieser Situation ein $FF-Byte und kann nun das Datenwort programmieren.

Setzt sich der Datenstrom in der folgenden S-Record Zeile mit dem zuvor fehlenden Byte fort, müsste der Monitor an der fraglichen Wortadresse einen erneuten Schreibzugriff vornehmen, was jedoch nicht zulässig ist. Es kommt zu einem Schreibfehler ("not erased").

Es ist daher notwendig, S-Record Daten vor der Programmierung auf gerade Adressen auszurichten. Hierzu kann z.B. das frei erhältliche Motorola Tool SRECCVT verwendet werden:

```
SRECCVT -m 0x00000 0xfffff 32 -o <outfile> <infile>
```

<infile> ist in unserem Beispiel blink1.s19, für <outfile> verwenden wir die Bezeichnung blink1a.s19.

Man kann den Konvertierschritt auch automatisch vom Compiler starten lassen, dazu später mehr.

Die genaue Syntax des Konverter-Tools ist im SRECCVT Reference Guide (PDF) beschrieben.

Load & Go

Nun wollen wir blink1a.s19 in den Flash laden. Nach Eingabe des Ladebefehls "L" (...und natürlich <ENTER>...) erzeugt das Monitorprogramm die Ausschrift "Loading..." und wartet nun auf S-Record Daten.

Wählen Sie nun über "Transfer/Open_file" die Datei blink1a.s19 aus und klicken Sie auf den Knopf "Transmit".

Im Terminalfenster erscheinen ein paar Sternchen und dann wieder das Promptzeichen. Sollte es eine Fehlermeldung wie "Write Error..." und "Invalid S-Record..." gegeben haben, dann war vermutlich der Flash nicht gelöscht. Holen Sie das Löschen des Flash in einem solchen Fall mit dem Monitorkommando "X" nach und laden Sie dann erneut.

Um das geladene Programm zu starten, geben Sie ein: "G 8000". Die Leuchtdiode sollte nun eifrig blinken.

Nichts geht mehr!?

Wundern Sie sich nicht, daß sich das Monitorprogramm nach dem Programmaufruf nicht mehr meldet. Bei einem DOS-Programm (auf einem "normalen" PC) würde sich nach getaner Arbeit irgendwann das Betriebssystem wieder melden. Ein Programm in einem Mikrocontroller kehrt hingegen normalerweise nie zur Betriebssystemebene zurück, denn meistens ist eine solche Ebene gar nicht vorhanden! Um das Programm auf dem CardS12 Modul abzubrechen und den Monitor neu zu starten, drücken Sie einfach den Resetknopf auf dem Modul.

Das Monitorprogramm stellt also gewissermaßen einen Ausnahmezustand für den Start eines Programms dar - im Regelfall kann man bei einer Embedded Applikation nicht davon ausgehen, daß eine solche "Basiseben" vorhanden ist. Welche Anforderungen sind an ein Controllerprogramm (zusätzlich) zu stellen, damit es ohne das Hilfsmittel "Monitor" auskommt?

Dazu muss man zunächst den genauen Ablauf beim Start eines Programms auf dem HC12 Mikrocontroller kennen. Außerdem ist es wichtig zu wissen, wie der C-Compiler die verschiedenen Module (Usercode und Librarykomponenten) zu einem monolithischen "Executable" (das ausführbare Controllerprogramm) kombiniert.

Bevor wir uns jedoch diesen Fragen zuwenden, wollen wir zunächst die Arbeit mit Compiler-Projekten näher in Augenschein nehmen.

1.4. Vom Projekt zum Programm

In der realen (Programmier-) Welt sind die Programme meist etwas länger als nur ein paar Zeilen, daher macht es Sinn, den Quelltext in verschiedene Module aufzuteilen. Der Programmierer behält so den besseren Überblick und der Compiler muß nicht nach jeder Modifikation den gesamten Quelltext neu übersetzen, sondern nur das geänderte Modul.

Natürlich obliegt es dem Compiler, nach jedem Übersetzungslauf die entstandenen bzw. vorhandenen Objektmodule neu zusammensetzen (zu linken). Um dem Compiler klarzumachen, welche Module überhaupt am Projekt beteiligt sind, bedient man sich einer Projektdatei. Die Projektdatei enthält also zunächst eine Liste aller Quelltextdateien. Außerdem wird in der Projektdatei vermerkt, welche Compilereinstellungen für die Übersetzung des Programms verwendet werden sollen.

Das Ergebnis des Compilerlaufes ist das (ausführbare) Controllerprogramm. Dieses Executable wird nun nach dem Projekt benannt, also nicht mehr nach dem Namen einer Quelltextdatei, wie das bei dem "projektlosen" Beispiel blink1.c der Fall war (daraus wurde blink1.s19).

Auch wenn die Einrichtung eines Projektes nach Zusatzaufwand klingt, den man sich am liebsten sparen möchte - es lohnt sich, diese Methode konsequent anzuwenden. Nur so ist die Wiederverwendbarkeit fertiger, getesteter Programmmodule effizient durchführbar.

Ein Projekt anlegen

Genug der grauen Theorie - wir bauen uns nun ein Blink-Projekt. Legen Sie zunächst ein Unterverzeichnis "bp2" an, damit wir den Überblick behalten. In der ICC12 Oberfläche klicken Sie nun auf "Project/New" und wechseln in das neu angelegte Verzeichnis. Verwenden Sie "bp2" zugleich als Projektnamen. Die Endung ".prj" vergibt ICC12 automatisch. Die Projektdatei heißt jetzt also bp2.prj und befindet sich im Verzeichnis bp2. Alle zu dem Projekt gehörenden

Quelltext-, Objekt-, Map-, Listing- und sonstigen Dateien befinden sich normalerweise im selben Verzeichnis.

Kopieren Sie das vorige Beispiel blink1.c in das neue Projektverzeichnis und benennen Sie die Kopie um in blink2.c. Als nächstes klicken Sie auf "Project/Add_File(s)" und fügen blink2.c dem Projekt hinzu. Der Dateiname erscheint daraufhin in der Projektbaum-Ansicht (im rechten Fensterbereich der IDE), ein Doppelklick auf den File-Eintrag öffnet blink2.c im Editor.

Stellen Sie nun noch einmal die Compileroptionen ein, sie werden stets zusammen mit dem Projekt gespeichert. Wählen Sie auch dieses Mal unter "Compiler/Options" innerhalb der Karteikarte "Target" die "Device_Configuration=Custom" aus und stellen Sie die Speicheradressen wie folgt ein:

- Program Memory: 0x8000
- Data Memory: 0x3000
- Stack Pointer: 0x3E00

Die restlichen Eingabefelder auf der Karteikarte "Target" bleiben leer bzw. im Defaultzustand. Auf der Karteikarte "Compiler" sollte in der Auswahl "Output_Format" die Einstellung "S19_with_Source_Level_Debugging" überprüft werden, damit die (später vom Debugger benötigten) Debug-Informationen generiert werden.

Klicken Sie nun auf "Project/Make_Project" oder drücken Sie die Taste F9 und der Compiler wird daraufhin das Projekt (welches bislang nur aus einem einzigen Quelltextmodul besteht) übersetzen. Das Resultat bp.s19 muss noch konvertiert werden. Wir automatisieren diesen Schritt, indem wir unter "Project/Options/Compiler" den Eintrag "Execute_Command_After_Successful_Build" mit folgendem Eintrag belegen (ohne den hier satztechnisch bedingten Zeilenumbruch):

```
c:\programme\icc\bin\SRecCvt -m 0x00000 0xFFFFF 32
-o %pa.s19 %p.s19
```

%p steht hierbei für den Projektnamen, in unserem Beispiel bp2.

Von nun an wird der Compiler nach jedem (erfolgreichen) Durchlauf den von uns benötigten Konvertierungsschritt anschließen.

Das dabei erzeugte File bp2a.s19 kann nun wieder mittels Terminal/Monitor geladen und mit dem Kommando "G 8000" gestartet werden (und vergessen Sie nicht, zuerst den Flash zu löschen :-)

1.5. Der Monitor kann mehr

So weit - so gut. Nach Drücken der Resettaste meldet sich der Monitor wieder. Da das Programm sich ja weiterhin im Flash befindet, man kann es immer wieder mit dem "G"o-Befehl starten. Geht das auch automatisch?

Im praktischen Einsatz wäre es zweifellos irrsinnig, müßte man stets eine magische Formel eintippen, bevor eine Mikrocontrolleranwendung ihre Arbeit aufnähme. Statt dessen müßte das (dauerhaft im Flashspeicher geladene Programm) automatisch starten, sobald die Stromversorgung eingeschaltet wird.

Das Leben nach Reset

Dieser Wunsch nach Automatik führt zu der Frage: Was genau geschieht in jenem ersten Augenblick - beim Einschalten der Stromversorgung bzw. nach einem Reset?

Zunächst kümmert sich der Mikrocontroller um den Resetvektor. Dieser 16 Bit Wert bildet die Startadresse für die CPU. Beim HCS12 befindet sich der Resetvektor auf den letzten beiden Positionen des 64 KB umfassenden Speicheradreßraumes, also auf den Adressen 0xFFFE und 0xFFFF. Die CPU setzt den Programmzähler auf den dort vorgefundenen Wert und beginnt dann, ab dieser Position sequentiell Maschinenbefehle einzulesen und abzuarbeiten.

Auf dem Controllermodul CardS12 ist im obersten, 4KB großen Abschnitt des Flashspeichers das Monitorprogramm untergebracht. Der Monitorcode definiert somit auch den Resetvektor, und dieser verweist

auf den Beginn des Monitorprogramms bei 0xF000. In Folge dessen wird nach jedem Reset automatisch das Monitorprogramm gestartet.

Autostart im Monitor

Der Programmcode des Monitorprogramms kann nicht durch das Monitorprogramm selbst gelöscht werden, diese Vorkehrung ist als Selbstschutz implementiert (via BDM ist das gleichwohl beliebig möglich). Demzufolge kann auch der Resetvektor nicht "verbogen" werden. Um dennoch eine Möglichkeit zu schaffen, statt des Monitors ein Anwenderprogramm zu aktivieren, beinhaltet der Monitor ein "Autostart" Feature. Verbindet man die Portleitungen PH6 und PH7 elektrisch miteinander, erkennt die Initialisierungsroutine des Monitors diese Verbindung und springt zur Adresse 0x8000, statt weiter den Monitorcode abzuarbeiten.

Sie hatten vorhin das Programm bp2a.s19 geladen? Dann verbinden Sie doch mal die beiden Leitungen PH6 und PH7 - zu finden sind diese Signale an den benachbarten Pins 7 und 8 des Steckverbinders X5. Wenn Sie jetzt das Board einschalten, blinkt die LED gleich von Beginn an!

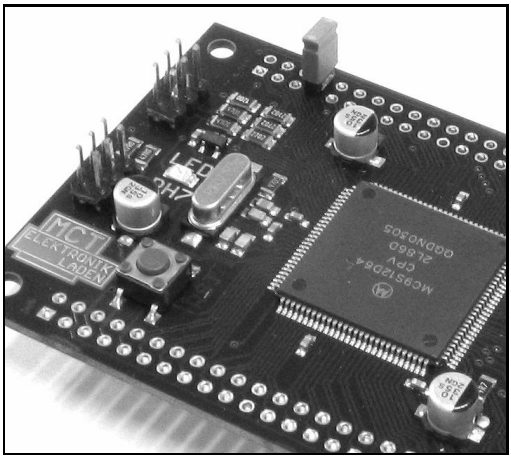

Abb.8: CardS12 mit (nachträglich bestücktem) Autostart Jumper

Noch mehr Vektoren

Natürlich unterstützt der HCS12 - als leistungsfähiger 16 Bit Controller - eine Vielzahl von Interruptquellen und verfügt demzufolge über eine entsprechende Anzahl Interruptvektoren. All diese Vektoren sind im obersten Adreßbereich, gleich unterhalb des Resetvektors angesiedelt. Ebenso wie der Resetvektor, befinden sich die Interruptvektoren in einem durch das Monitorprogramm schreibgeschützten Bereich.

Um dennoch Interruptfunktionen in einem Anwenderprogramm zu ermöglichen, leitet der Monitorcode alle Interruptvektoren in den internen RAM Bereich um. Im RAM ist Platz für einen drei Byte langen "Pseudovektor", der tatsächlich einen Sprungbefehl darstellt.

Ein Beispiel: Der Output Capture 2 (OC2-) Timerinterrupt soll - quarzgenau - nach jeweils 0,5 Sekunden unsere LED umschalten. Ein Blick in den Device Guide des MC9S12D64 verrät, daß der dazu passende OC2-Vektor bei 0xFFEA und 0xFFEB angesiedelt ist. Der Monitorbefehl "D FFEA" gibt schnell Klarheit, wohin der Monitorcode in diesem Fall verweist: nach 0x3FE2 im internen RAM des 'D64. Dort muß das Anwenderprogramm - zur Laufzeit - den Pseudovektor eintragen. Der erste Teil dieses Pseudovektors besteht aus dem Byte 0x06, das ist der Opcode für den Maschinenbefehl JMP. Der zweite Teil ist die Adresse der Interrupt Service Routine des Anwenderprogramms namens isrOC2().

Das folgende Listing des Programms "blink3.c" zeigt, wie alles zusammen kommt. Übersetzt, konvertiert, geladen und gestartet wird das Programm wie zuvor im Beispiel blink1.c.

```
//-------------------------------------------------------------

#include <stdio.h>
#include "hcs12dp256.h"

//-------------------------------------------------------------

#pragma interrupt_handler isrOC2
void isrOC2(void) {

    TFLG1 = 0x04;   // clear OC2 Intr Flag
    TC2 += 31250u;  // 1 interrupt every 31250 timer ticks (0.5s)
    PTH ^= 0x80;    // toggle LED
    }

//-------------------------------------------------------------

void main(void) {

    // install OC2 pseudo vector in RAM
    // (if running with TwinPEEKs monitor)
    *((unsigned char *)0x3fe2) = 0x06;  // JMP opcode
    *((void (**)(void))0x3fe3) = isrOC2;

    // make PTH[7] an Output to drive the LED
    DDRH |= 0x80;

    // Timer Setup
    TSCR1  = 0x80;  // Timer enable
    TSCR2  = 0x07;  // Prescaler = 128 (1 timer tick every 16µs)
    TCTL2  = 0x00;  // TC2 disconnected from Pin
    TIOS  |= 0x04;  // TC2 is Output Compare
    TIE   |= 0x04;  // TC2 Interrupt enable

    // enable Interrupts
    asm("cli");

    // loop endless (and let the interrupt work)
    while(1) ;
    }

//=============================================================
```

2. Background Debugging

Ein Monitorprogramm ist zweifellos sehr praktisch, wie in den vorangegangenen Abschnitten gezeigt wurde. Allerdings belegt es auch Speicherplatz und verwendet Ressourcen des Mikrocontrollers. Und wächst der Speicherbedarf des Anwenderprogramms oder werden spezielle Betriebsbedingungen benötigt, dann ist der Monitor schliesslich irgendwann obsolet. Zum Großreinemachen im Flash benötigen wir den Background Debug Mode des HCS12 - freilich mit den passenden Werkzeugen.

2.1. Wie BDM12 funktioniert

Background Debug Mode, kurz BDM, ist das beim HCS12 (und anderen Motorola Mikrocontrollern) angewandte Kommunikationsverfahren für Debugging und Download. Aus elektrischer Sicht ist BDM, zumindest in der HCS12 Version, eine serielle Verbindung, betrieben über einen einzigen Prozessorpin namens BKGD. Hinzu kommt die Resetleitung, welche ja ohnehin im System vorhanden ist. Obwohl das Verfahren einfach anmutet, ist es sehr leistungsfähig. Für die häufigsten Debuggingaufgaben erübrigt sich die Anschaffung kostspieliger In-Circuit-Emulatoren.

Ganz ohne Hardware-Tools kommt aber auch BDM12 nicht aus. Ein Interface, welches gemeinhin als "BDM-Pod" bezeichnet wird, muß zwischen dem schnellen, proprietären BDM12-Protokoll und den PC-üblichen Kommunikationsschnittstellen vermitteln.

Der Anschluß des Pod erfolgt über einen 6-poligen Steckverbinder, die Anschlußbelegung ist von Motorola wie folgt definiert:

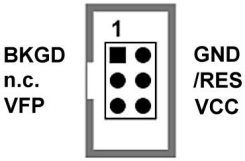

Abb.9: Der von Motorola definierte BDM12 Anschluß

Die Stromversorgung (VCC, GND) wird auf dem Kabel durchgereicht, das Pod kann daher die Betriebsspannung von der Targetplatine erhalten. Natürlich darf das Pod nur im stromlosen Zustand angekoppelt werden. VFP spielt beim HCS12 keine Rolle mehr - hier wurde für die HC12-Typen der ersten Generation eine Programmierspannung für den Flash zugeführt.

Der HCS12 kennt verschiedene Betriebsarten. Wenn ein Anwenderprogramm oder der Monitor aktiv ist, arbeitet der Controller auf dem CardS12 Modul im Normal Single Chip Mode. Ist das BDM-Pod angeschlossen, startet der Controller statt dessen im Special Single Chip Mode. Dies ist der einzige Modus, in dem das Background Debug Modul gleich nach Reset die Kontrolle über den Chip übernimmt und nicht erst explizit aktiviert werden muß. Das Pod legt dazu einfach die BKGD Leitung kurzzeitig auf Low und erzeugt während dessen einen Resetimpuls.

Im Gegensatz zu den Normalbetriebsarten startet nach Reset nicht das Anwenderprogramm, sondern eine spezielle BDM Firmware im Controller-ROM. Der Controller wartet daraufhin auf das Eintreffen von Befehlen über die BKGD Leitung.

Der HCS12 kann nun per Software in eine andere Betriebsart geschaltet werden. Der Debugger muß hierzu lediglich einige Steuerregister der MCU via BDM modifizieren. Dieses Feature ermöglicht es, nahezu beliebige Startbedingungen zu emulieren, auch wenn nach Reset zunächst immer der Special Single Chip Mode aktiv ist.

2.2. Download via BDM

Laden mit StarProg

Die Windowsanwendung StarProg nutzt das seriell angeschlossene ComPod12 zum Download von S-Record Dateien via BDM. Die Handhabung ist sehr einfach: der Targettyp wird aus einer Liste im Menü "Setup/Select_Target" ausgewählt, z.B. "MC9S12D64 Flash" für das CardS12 Board mit dem MC9S12D64. Dann wird das Controller-programm über "File/Open" eingelesen. Nun kann der Targetspeicher mit "Target/Erase" gelöscht und anschließend mit "Target/Programm" programmiert werden. Mit "Target/Verify" kann man das Ergebnis nochmals überprüfen.

Weil die Erase-Funktion stets den gesamten Flash löscht, ist demzu-folge anschließend das Monitorprogramm "weg". Die bisher gezeigten Programmbeispiele bauten aber auf Funktionen des Monitors auf. Da nun Autostart, Pseudovektoren und Systeminitialisierung nicht mehr vorbereitet sind, muß sich das Anwenderprogramm ab sofort selbst um diese Belange kümmern.

Auf eigenen Füßen

Der Quelltext blink3.c soll weiter verwendet und ausgebaut werden, um das Beispielprogramm eigenständig lauffähig zu machen. Zunächst legen wir ein neues Projektverzeichnis mit dem Namen bp4 an. In dieses Verzeichnis kopieren wir den o.g. Quelltext, wir geben ihm dort den Namen blink4.c. Aus dem Unterverzeichnis "examples.12" des C-Compilers ICC12 besorgen wir uns vectors_dp256.c und legen ebenfalls eine Kopie im Projektverzeichnis ab. Nachdem im ICC12 ein neues Projekt namens "bp4.prj" angelegt wurde, können die beiden Quelltextdateien zu dem Projekt hinzugefügt werden.

Nun ändern wir in der lokalen vectors_dp256.c die Zeile

```
  DUMMY_ENTRY,  /*Timer Channel 2*/
```

wie folgt:

```
    isrOC2,       /*Timer Channel 2*/
```

Unter die Externdeklaration für das Symbol _start

```
extern void _start(void); /* entry pt in crt??.s */
```

ergänzen wir eine ähnliche Zeile für die Interrupt Service Routine isrOC2():

```
extern void isrOC2(void);
```

Die modifizierte Datei vectors_dp256.c enthält nun den Resetvektor (zeigt auf _start) und den Interruptvektor für den Timer Kanal 2.

Die Lücke zwischen _start und Start

Offen bleibt die Frage, was es mit dem Symbol _start auf sich hat. Offenbar ist dies der Eintrittspunkt des Programms, denn der Resetvektor verweist auf _start. Dieses Symbol taucht aber im Quelltext gar nicht weiter auf?

Die Lücke zwischen _start und dem eigentlichen Start des C-Programms in der Funktion main() erklärt sich leicht, wenn man sich die Vorgehensweise eines C-Compilers beim Linken eines Programms verdeutlicht. Jeder C-Compiler fügt zusätzliche Objekte zu den Anwendermodulen hinzu. Wenn das Programm Bezug auf Bestandteile der Standardbibliothek nimmt, werden die benötigten Bibliotheksmodule mit in das Programm aufgenommen. In jedem Fall wird aber das so genannte Startup-Modul eingebaut.

Das ICC12 Startup-Modul heißt crt12.o. Den Quelltext hierzu enthält die Assemblerdatei crt12.s; sie beginnt mit der Marke "_start" (aha!). Die wichtigste Aufgabe des Startup-Moduls ist der Aufruf der Hauptfunktion main(). Darüber hinaus zeichnet es verantwortlich für das Setzen des Stackpointers, das Löschen des Daten im RAM (bss-Bereich) und die Initialisierung von statischen Variablen (so vorhanden).

Das Startup-Modul des ICC12 ruft gleich zu Beginn eine Funktion namens _HC12Setup() auf. Diese Funktion schaltet - in der

Defaultversion, wie sie in der ICC12 Standardbibliothek enthalten ist - den Watchdog (COP) ab (das ist beim HCS12 eigentlich redundant). Die Funktion kann beliebig durch Anwendercode ersetzt werden, denn Code in den Anwendermodulen hat stets Vorrang vor gleichartigen Bibliotheksfunktionen. Ein Gerüst zur Implementierung einer eigenen _HC12Setup() Funktion sieht wie folgt aus:

```
void _HC12Setup(void) {
      //...
      }
```

So enthalten in unserem Beispiel blink4.c. In der selben Datei können noch die Zeilen gelöscht werden, die zum Initialisieren der - nunmehr entfallenen - Pseudo-Interruptvektoren dienten.

Bevor das Projekt neu übersetzt wird, müssen noch die Linkereinstellungen angepasst werden. Die Änderungen ergeben sich aus der Lage des RAMs unmittelbar nach Reset (ohne die Initialisierungen des TwinPEEKs Monitors):

- Program Memory: 0x8000
- Data Memory: 0x0400
- Stack Pointer: 0x1000

2.3. BDM-Debugging mit NoICE

Debugging via BDM bietet hervorragende Möglichkeiten, Fehler in Programmen schnell aufzufinden. Der Aufwand des Verfahrens ist gleichzeitig sehr gering. Es gibt viele Entwickler, die den HCS12 alleine schon wegen des Background Debug Mode anderen Controllern vorziehen. Das überrascht nicht, wenn man bedenkt, daß Debugging und Test den größten Zeitanteil bei einer Softwareentwicklung erfordern.

Der größte Vorteil, den die BDM-Schnittstelle bei der Fehlersuche bietet, besteht wohl darin, daß meistens keine Änderungen am zu untersuchenden Anwenderprogramm vorgenommen werden müssen. Der Background Debug Mode blockiert keine Ressourcen des Controllers.

Lese- wie auch Schreibzugriffe kann das BDM-Modul parallel zum normalen Programmablauf ausführen - das Verfahren ist "non-intrusive".

NoICE starten

Nach dem Start überprüft NoICE zunächst die Verbindung zum Zielsystem. Es wird ein Reset ausgelöst und die Target-MCU in den Special Single Chip Mode versetzt. Anschließend sendet die Software erforderlichenfalls eine Initialisierungssequenz zur Target-MCU. Dieses Feature ermöglicht es, hardwarespezifische Initialisierungen direkt nach Reset vorzunehmen, z.B. die Umschaltung des Operating Mode oder die Aktivierung des externen Businterfaces samt Bussignalen, Chip Selects, Waitstates etc. Die Initialisierungssequenz ist in einer Textdatei definiert, die vom Anwender leicht anzupassen ist.

Bei dem CardS12 Modul mit dem MC9S12D64 beschränken sich die notwendigen Initialisierungsschritte auf die Umschaltung in den Normal Single Chip Mode über die Steuerregister MODE und PEAR. Der Watchdog (COPCTL Register) ist beim HCS12 nach Reset ohnehin abgeschaltet. Die Initialisierungssequenz, ein so genanntes PLAY-File, ist eigentlich nichts weiter als eine NoICE Makrodatei:

```
REM --MODE--
EDIT 0x000b 0x80

REM --PEAR--
EDIT 0x000a 0x10
```

Der Name dieses PLAY-Files "cards12.noi" wird in das Feld "Play_this_file_after_Reset" (unter "Options/Target_Communications") eingetragen. NoICE wird diese Datei nun automatisch nach jedem Reset abarbeiten.

In der selben Dialogbox sind auch alle globalen NoICE-Einstellungen zur Kommunikation mit dem Target enthalten. Die Abbildung zeigt die Einstellungen für die Arbeit mit CardS12 und den nachfolgenden Beispielen:

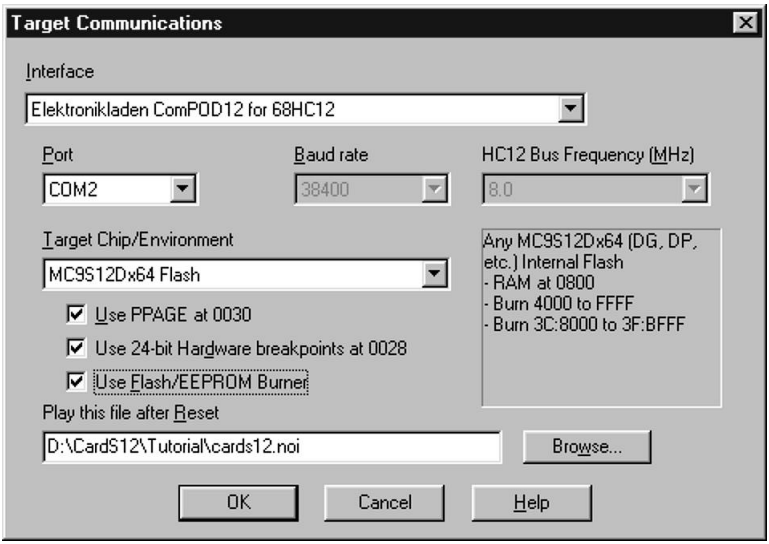

Abb.10: NoICE Einstellungen im Target Communications Dialog

Mit NoICE arbeiten

Die Oberfläche von NoICE gliedert sich in drei Hauptbereiche (vergl. Abb.11). Der zentrale Fensterbereich dient zur Anzeige des Anwenderprogramms. NoICE liest den Inhalt des Programmspeichers in der MCU, disassembliert den dort vorgefundenen Code und stellt die Ergebnisse als Disassemblerlisting dar. Darüber hinaus kann NoICE Symbolinformationen in dieses Listing einbauen. Diese Informationen über symbolische Bezeichner können vom Benutzer eingegeben oder aus einer (vom Assembler bzw. Compiler erzeugten) Symboldatei gelesen werden.

Da die zur Symboldefinition verwendeten Dateien auch Zeileninformationen enthalten können, ist NoICE in der Lage, Assembler- und sogar C-Programme in Sourceform darzustellen. Die Anzeige als Quelle oder Disassembleroutput ist umschaltbar mit dem Button "Toggle_source/disassembly".

Unterhalb der Symbolleiste befindet sich eine Darstellung aller Prozessorregister. Die Werte werden nach jedem Programmschritt bzw. -lauf aktualisiert. Hilfreich ist die farbliche Hervorhebung von Registerinhalten. Werte, die sich durch einen Programmschritt geändert haben, werden rot eingefärbt.

Oberhalb der Registeranzeige und neben der Symbolleiste blendet NoICE eine Eingabezeile ein. Hier kann der Benutzer Befehle an den Debugger per Kommandozeilen-Eingabe erteilen. Alle Befehle, die über Menüeinträge oder Icons erreichbar sind, können ebenso über diese Kommandozeile eingegeben werden. Darüber hinaus ist der Remote Debugger makrofähig, d.h. man kann Befehlsfolgen in Dateien (die oben erwähnten PLAY-Files) speichern und später erneut "abspielen".

Der untere Bereich des Programmfensters ist der Anzeige von Statusinformationen, Watchpoints und Memory-Dumps vorbehalten.

Zum Testen eines Programms lädt man es zunächst mit dem Menübefehl "File/Load" in den Speicher des Zielsystems. NoICE lädt entweder nur den Programmcode als S-Record (S19-) Datei oder berücksichtigt auch gleich die vom Compiler bereitgestellten Debuginformationen.

Die mit dem Debug-File übergebenen Informationen enthalten im wesentlichen Namen, Struktur, physikalische Adressen und Gültigkeitsbereich von symbolischen Bezeichnern sowie Zuordnung von Quelltextzeilen zu Codeadressen. NoICE kann Debuginformationen im einfachsten Falle in Form von PLAY-Files einlesen. Die verschiedenen Compiler erzeugen ganz unterschiedliche Dateiformate für Debuginformationen, die Standardisierung ist in diesem Gebiet nicht sehr fortgeschritten. Es stehen für NoICE verschiedene Konverterutilities zur Verfügung, um Fremdformate in PLAY-Files umzuwandeln. Im Falle des Imagecraft Compilers ICC12 ist NoICE in der Lage, die erzeugten Debug-Files mit der Endung .DBG auch direkt mit der LOAD Funktion einzulesen - zweifellos der einfachste Weg.

Nach dem Laden der DBG-Datei steht das Program Counter-Registers auf der Startadresse des Programms. Im Disassemblerfenster ist

der Startup-Code zu sehen. Weil der Startup-Code nicht auf einer C-Quelle basiert, kann freilich hier noch kein Sourcelisting angezeigt werden. Dies ist erst möglich, wenn der Eintrittspunkt im Anwenderprogramm, die Funktion main(), erreicht ist. Man kann den Startup-Code automatisch abarbeiten und bei main() stoppen, indem im Menü "Run" die Option "Go_until_main_after_load" markiert wird.

Einen Breakpoint platziert man einfach mittels Mausklick (Kontextmenü) auf die gewünschte Zeile im Programmlisting. Ein laufendes Programm (starten mit "Run/Go") wird dann an dieser Stelle gestoppt und die Anzeige automatisch aktualisiert. Dabei ist es nicht immer gewährleistet, dass die Unterbrechung auf einen Maschinenbefehl fällt, welcher der erste Befehl innerhalb der Befehlssequenz einer C-Quellzeile ist. Die Anzeige wechselt ggf. von der C-Sourceansicht zu der gemischten C/Asm Ansicht.

NoICE unterstützt das Debugging weiterhin mit Watchpoints (Anzeige von Variablen bzw. Symbolen), einem Memory-Editor sowie einem Zeilenassembler. Die ausführliche Onlinehilfe gibt viele Tipps und Hinweise!

Im Flash

Zum Laden von Programmen in den Flash bedarf NoICE keiner weiteren Hilfestellung. Die Programmieralgorithmen für den Flash sind seit der NoICE Version 7 integriert, müssen aber im Menü "Options/Target_communications" aktiviert sein. Andernfalls nimmt NoICE an, der Zielspeicher wäre RAM.

Es blinkt wieder

Nach diesen Vorbereitungen können wir nun das zuletzt besprochene Beispielprojekt "bp4" mit NoICE untersuchen.

Laden Sie zunächst mit dem Menübefehl "File/Load" die von ICC12 erzeugte Debuginfo-Datei "bp4.dbg". Das Dateiformat ist "Imagecraft_DBG_files". In der Datei ist ein Verweis auf die zugehörige Codedatei bp4.s19 enthalten. Beim Laden der Debuginfo-Datei wird die Codedatei von NoICE automatisch mit geladen (geflasht).

NoICE zeigt nun das Programm ab der Startadresse 0x8000 in Form eines Disassemblerlistings an:

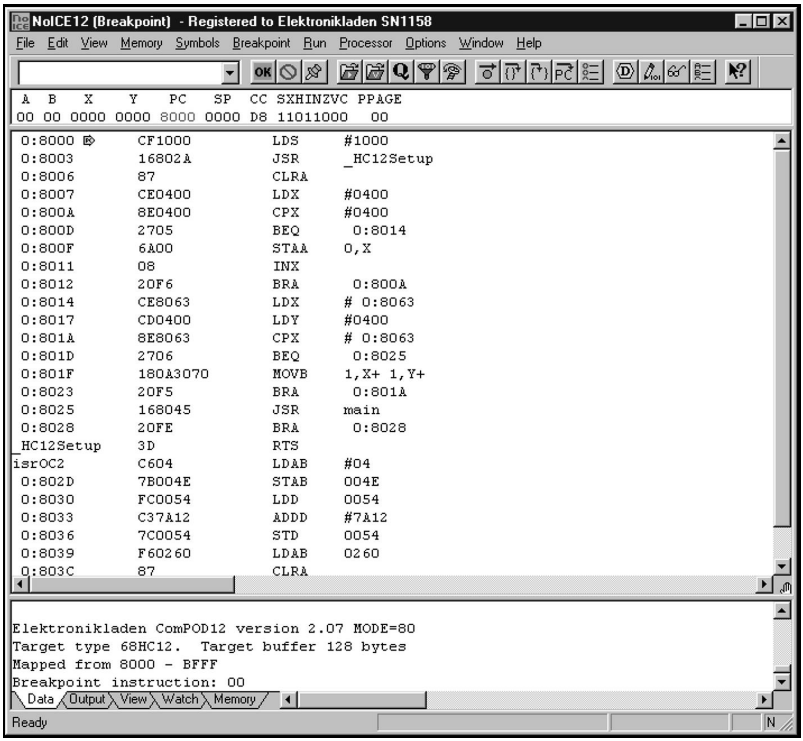

Abb.11: Disassemblerlisting nach dem LOAD Befehl

An dieser Stelle kann noch kein C-Quelltext angezeigt werden, weil wir das C-Startup Modul sehen, welches in Assemblersprache abgefaßt wurde. C-Quelltext gibt es erst ab der main()-Funktion.

Wir setzen mit "Breakpoint/Insert" einen Breakpoint auf "#main" und starten das Programm mit "Run/Go". Gleich darauf stoppt es an der gewünschten Stelle und nun ist es auch möglich, zwischen Quelltext und Assembler-Level umzuschalten (ggf. ist zuvor ein "View/Source_at_PC" notwendig).

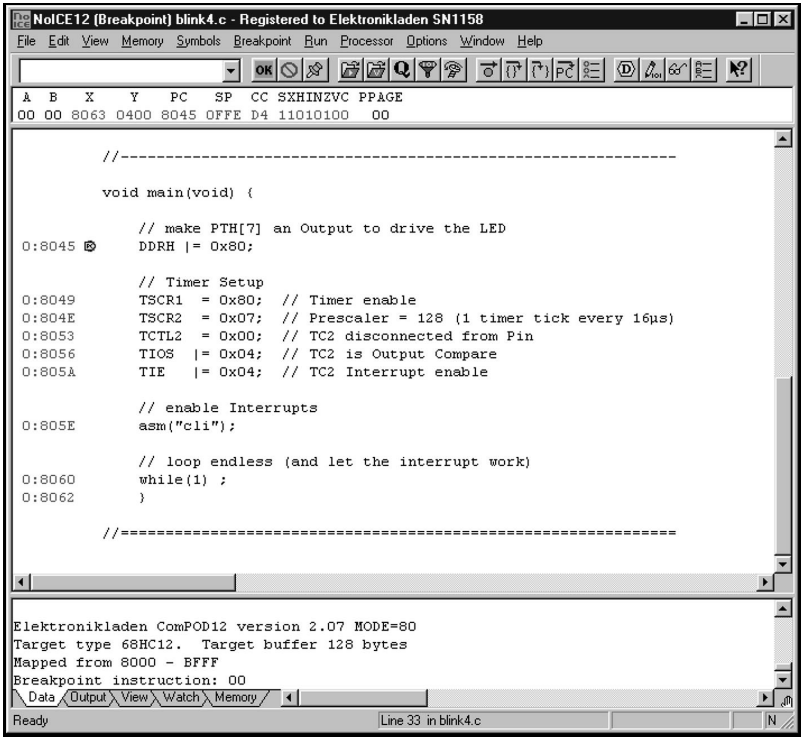

Abb.12: Ab main() ist die Anzeige von C-Quelltext möglich

Von hier aus kann man das Programm schrittweise oder in Echtzeit weiterlaufen lassen, Unterbrechungspunkte setzen, Variablen mittels Watchpoints verfolgen oder jederzeit mittels Memory Dump einen Blick in den Speicher des Target werfen.

Fazit

NoICE ist eine große Hilfe, wenn es darum geht, Fehlfunktionen im Programmcode aufzuspüren. Die wichtigste Fähigkeit für erfolgreiches Debugging kann dieses Tutorial leider nicht vermitteln: der siebte Sinn zum Aufspüren von Programmfehlern. Hierfür ist Erfahrung nötig, welche man am besten durch nachhaltiges Training der eigenen grauen Zellen erwirbt.

Und eines ist sicher: der Fehler hat zu 99,9% seine Ursache nicht in einem defekten Controllerchip, sondern im menschlichen Faktor. Lesen und Nachdenken ist stets wichtiger, als Quelltext im Akkord zu tippen. Getreu dem Motto: Wenn Debugging der Prozess zur Entfernung von Fehlern ist, dann ist Programmieren offenbar das Gegenteil...

Literatur:

[1] Brian W. Kernighan, Dennis M. Ritchie: Programmieren in C, Zweite Ausgabe: ANSI-C; Carl Hanser Verlag

[2] Harald Kreidl, Gerald Kupris, Oliver Thamm: Mikrocontroller-Design; Hardware- und Softwareentwicklung mit dem 68HC12/HCS12; Carl Hanser Verlag