Mikrocontroller-Programmierung am Beispiel eines Intel 8051-kompatiblen Cypress-Chips

Ausarbeitung zu Vortrag im Seminar

C- und Unix-Programmierung

im Fach Informatik
an der Universität Trier

von Franz Brauße

bei Herrn Dr. M. Ley

Trier, September 2010

Zusammenfassung

Diese Arbeit befasst sich mit Besonderheiten bei Speicher, Programmablauf, Hardware und Compilererweiterungen in Bezug auf die Programmierung eines Intel 8051-kompatiblen Mikrocontrollers in der Programmiersprache C.

Inhaltsverzeichnis

1. Mikrocontroller

Unter Mikrocontrollern versteht man auf ein Anwendungsgebiet spezialisierte Prozessoren, deren allgemeines Leistungsniveau in der Regel deutlich unter dem aktueller General-Purpose-Prozessoren liegt, die aber preiswert in der Produktion und einfach integrierbar in ein Produkt sind, da ein Großteil der sonst nötigen Zusatzbausteine bereits in den Chip integriert ist.

Teils nötigen die extremen Anforderungen der Umgebung, in der das System arbeiten soll, zu einer Fertigung in gröberen Strukturen und damit auch zu preiswerter zu produzierenden, älteren Architekturen. Chips feinerer Strukturen sind aufgrund des dort herrschenden, niedrigeren Spannungsniveaus anfälliger für Spannungsschwankungen, ausgelöst durch beispielsweise elektromagnetische Felder oder eine anfällige Stromversorgung in instabiler Umgebung. Ein weiterer Grund für den Einsatz einer robusteren Technik findet sich in extremen Arbeitstemperaturen des Chips, zum Beispiel in der Raumfahrt. Durch Optimierung auf wenige, klar definierte Aufgaben wird eine deutlich geringere Komplexität der Implementierungen erzielt. Dadurch spielen auch Effekte wie kleinere physikalische Ausdehnung des vollständigen Systems, geringerer Strombedarf und letztlich die preiswerteren Produktionsmethoden eine Rolle in der Entscheidung für den Einsatz von spezialisierten Mikrocontrollern anstatt General-Purpose-Prozessoren.

Weiterhin werden diese Controller inklusive eines Code- (ROM und/oder Flash) sowie Arbeitsspeichers hergestellt und enthalten vielfach Schaltlogik für Interruptbehandlung, Watchdogs, A/D-Wandler[1] und in der Regel eine direkte Ansteuerung mindestens eines Peripheriebusses[2] zur Kommunikation und Steuerung.

Demgegenüber stehen Schwierigkeiten in der Programmierung, da sich diese erheblich von der üblicher Programme für vollwertige Betriebssysteme unterscheidet. Die geringe Taktfrequenz und Speichergröße, sowie das Fehlen von Standards (da viele verschiedene Architekturen zum Einsatz kommen) und vor allem eines Betriebssystems stellen eine Herausforderung in den Einstieg die Mikrocontrollerprogrammierung dar.

2. Benutzte Hard- und Software

Diese Arbeit bezieht sich auf die Programmierung des Intel 8051-kompatiblen 8 Bit Cypress-Controllers CY7C68014A[3] der Familie EZ-USB FX2 mittels C und Assembler, kompiliert durch den Open-Source-Compiler SDCC[4]. Das Entwicklerboard (Abb. 3) stammt von der Firma ztex[5].

Das Board verfügt neben dem auch als Stromanschluss fungierenden Mini-USB-Port Typ B über 16,5 KB in den Chip integrierten RAM-Speicher zur Nutzung durch die Firmware sowie weitere 7,5 KB für USB-Puffer, einen µSD-Slot, drei Timer, einen taktgebenden 24 MHz Quarz sowie eine Anzahl programmierbarer I/O-Leitungen (Input / Output).

[1] bspw. die ATmega-Chips von Atmel, verbaut u.a. in Arduino-Boards, `http://arduino.cc/`
[2] CAN im Automobilbereich, I²C, SPI, USB, ...
[3] `http://www.cypress.com/?mpn=CY7C68014A-100AXC`
[4] Small Devices C Compiler, `http://sdcc.sf.net/`
[5] `http://ztex.de/usb-1/usb-1.0.e.html`

2.1. Speicher

Der Speicher eines Mikrocontrollers muss mangels Betriebssystem von der Firmware vollständig selbst verwaltet werden. Dazu ist es erforderlich, alle Speicherklassen zu kennen und dem Compiler mitzuteilen, wo bestimmte Daten liegen sollen. SDCC bietet dafür eigene Bezeichner, die im C-Standard [6] nicht definiert sind. Sie beginnen wie alle inoffiziellen C-Erweiterungen mit zwei Unterstrichen (__). Es folgt ein Überblick des von der 8051-Architektur zur Verfügung gestellten Speichers. Der Übersicht halber werden die Typerweiterungen, die der in Abschnitt 3 vorgestellte Compiler benutzt, mit angegeben.

Der 64 KB große adressierbare Speicherbereich teilt sich in die von [3] so genannten unteren 128 Byte (`0x00-0x7f`), weitere 128 Byte internen RAMs zwischen `0x80` und `0xff` sowie externen Speicher auf. Die Bezeichnung extern impliziert nicht unbedingt, dass ein separater Speicherchip existiert, sondern referiert lediglich auf Adressen größer als `0x00ff` und die dazu benötigten, in Abschnitt 2.1.2 erläuterten Adressierungsvarianten. Der Zugriff auf Daten kann in jedem Fall indirekt erfolgen, das heißt die Speicheradresse wird in ein spezielles 16-Bit-Register `DPTR`, zusammengesetzt aus `DPL` und `DPH`, kopiert, und mit indirekt adressierenden Befehlen kann danach über dieses – im Unteschied zu x86 nicht explizit anzugebende – Register auf den jeweiligen Bereich zugegriffen werden. Es existieren jedoch Modi für direkten Zugriff und auch der indirekte Zugriff lässt sich weiter unterteilen.

2.1.1. Interner Speicher

Für Adressen unterhalb der 256-Byte-Grenze sind zwei weitere Adressierungsmodi vorgesehen: Bit-Adressierung und direkte Adressierung. In beiden Fällen wird die 1 Byte große Adresse direkt in die Instruktion kodiert. Die direkte Adressierung ist nach der der Register `R0` bis `R7` sowie des Akkumulators die schnellste Möglichkeit um auf den Speicher zuzugreifen. Die entsprechende Compiler-Typerweiterung ist **`__data`**. Bit-Adressierung hingegen ermöglicht es, einzelne Bits der an bestimmten Speicheradressen liegenden Daten direkt zu manipulieren oder auszulesen. Für allgemeine Programmdaten sind hierfür die 16 Adressen `0x20-0x2f` im internen RAM vorgesehen, deren je 8 Bits der enthaltenen Daten aufsteigend eine Adresse von `0x00` bis `0x7f` zugeordnet werden. Beispielsweise setzt der Befehl `clrb 0x42` das dritte Bit, $2 = \texttt{0x42} \bmod 8$, (Bits werden von 0 bis 7 nummeriert) des Bytes an Adresse $\texttt{0x28} = \texttt{0x20} + \left\lfloor \frac{\texttt{0x42}}{8} \right\rfloor$ auf 0. Der Typ einer Variable, die vom Compiler in den Bit-adressierbaren Bereich gelegt werden soll, wird um **`__bit`** erweitert um dies mitzuteilen.

Im Bereich von `0x80-0xff` kann jede Adresse $a = 0 \bmod 8$, d.h. `0x80`, `0x88`, `0x90`, ..., Bit-adressiert werden; $a + b$ mit $b \in \{0, \ldots, 7\}$ ist dann die in den Bit-Maschinenbefehl einzutragende Adresse um auf das Bit b des Werts an Adresse a zuzugreifen. **`__sbit`** ist die Typerweiterung, die so definierten Bits an durch 8 teilbaren Speicheradressen einen Namen zuordnet. Zusätzlich sind die 128 Adressen in diesem Bereich mit einer weiteren Funktion neben dem Ablegen normaler Programmdaten belegt. Der sogenannte SFR-Space[6] liegt hier und wird über direkte Adressierung angesprochen (**`__sfr`**, **`__sfr16`** und **`__sfr32`**), wohingegen der tatsächliche interne RAM indirekt adressiert wird. Hierfür existiert neben der Variante mit dem `DPTR`-Register noch eine

[6] Special Funktion Register, direkter Zugriff auf die Hardware, I/O-Pins, Timer, etc., siehe auch Abb. 5

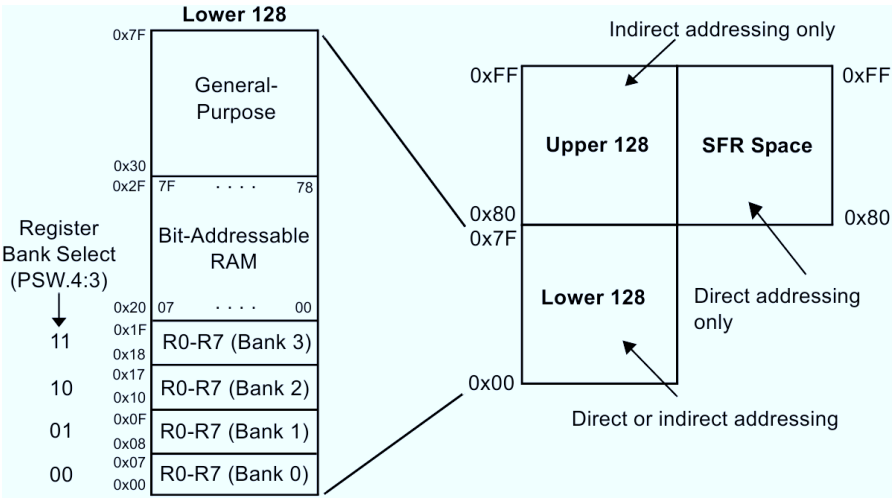

Abbildung 1: Aufteilung der unteren 256 Adressen von `0x00` bis `0xff`

weitere Art der indirekten Adressierung, nämlich über die Register `R0` bzw. `R1`. In diesen kann die zuzugreifende (1 Byte große) Adresse abgelegt und mit einer weiteren Variante der meisten Maschinenbefehle über `R0` bzw. `R1` auf die Adresse zugegriffen werden. Dem Compiler wird eine solche Datenposition mittels **`__idata`** mitgeteilt.

Von den unteren 128 Byte referieren die Adressen `0x00`-`0x07` auf die Register `R0` bis `R7` der Registerbank 0. Der 8051-Chip hält 4 Registerbänke vor, wie in Abb. 1 aufgeschlüsselt ist. Die aktuelle Registerbank wird über ein SFR namens `PSW` gesetzt.

2.1.2. Externer Speicher

Der 8051 besitzt für Programmcode und Speicher eine Harvard-Architektur, das heißt Code und Daten liegen in getrennten Speicherbereichen, beide reichen über den gesamten Adressbereich von `0x0000` bis `0xffff`. Diese Bereiche unterscheiden sich dadurch, dass sie durch unterschiedliche Maschinenbefehle angesprochen werden. Typischerweise nutzt man bei der Standard-8051-Architektur einen Read-Only-Memory-Chip für Programmcode und einen Flash- oder anderen beschreibbaren Speicherchip für Daten. Auf physikalischer Ebene werden beide Bereiche bzw. Chips getrennt durch drei Leitungen, die vom 8051 gesteuert werden: `PSEN` (Program Store Enable) aktiviert den nur lesbaren Programmspeicher, wohingegen `RD` lesenden und `WR` schreibenden Zugriff auf den Datenspeicher herstellen. `PSEN` und `RD`/`WR` werden durch zwei verschiedene Varianten des `MOV`-Maschinenbefehls gesteuert, `MOVC` für Code und `MOVX` für externen Datenspeicher. Das Schlüsselwort **`__code`** teilt dem Compiler mit, dass eine Variable auf Programmcode referiert, wohingegen **`__xdata`** auf den externen RAM zeigt.

Auf externem RAM können keine Rechenoperationen direkt ausgeführt werden. Die Daten müssen zunächst entweder in den Akkumulator (ein spezielles SFR namens `ACC`, siehe Abb. 5 an Adresse `0xe0`) oder in den internen RAM, d.h. an Adressen $\leq$ `0xff` kopiert werden, da die einzige Sorte an Maschinenbefehlen, die mit dem externen Speicher kommunizieren, `MOV`-Befehle sind.

Eine weitere Adressierungsart, die der 8051 bietet, ist der sogenannte Paged External RAM (**`__pdata`**), ein 256-Byte großer Bereich im externen RAM, auf den indirekt

Speicherklasse	Typerweiterung	Adressierung	Speicherbereich
Interner RAM	**__data**	direkt	0x00 - 0x7f
Interner RAM	**__idata**	indirekt über R0, R1	0x00 - 0xff
SFR	**__sfr**	direkt	0x80 - 0xff
Daten-Bits	**__bit**	direkt, Bit-Instruktionen	0x20 - 0x2f
SFR-Bits	**__sbit**	direkt, Bit-Instruktionen	0x80, 0x88, ...
Externer RAM	**__pdata**	indirekt, MOVX mit R0, R1	0xXX00 - 0xXXff
Externer RAM	**__xdata**	indirekt, MOVX mit DPTR	0x0100 - 0xffff
Code ROM	**__code**	indirekt, MOVC mit DPTR	0x0000 - 0xffff

Tabelle 1: Übersicht über Speicherklassen des 8051

über R0 oder R1 zugegriffen werden kann. Dazu müssen die oberen 8 Adressbits der gewünschten Speicherseite in ein SFR[7] kopiert werden. R0 bzw. R1 geben dann die unteren 8 Bit der Adresse an, die der entsprechende MOVX-Befehl benutzt. Über diesen Mechanismus kann der gesamte externe RAM adressiert werden, jedoch werden neben einem der Register R0 und R1 noch das genannte SFR, welches sonst für andere Aufgaben genutzt hätte werden können, sowie der Akkumulator belegt, welcher das einzige Zielregister einer MOVX-Instruktion ist.

2.1.3. Non-8051-kompatible Erweiterungen

Der erwähnte Cypress-Chip FX2 setzt die Architektur des 8051 nur teilweise wie beschrieben um, damit auch schreibbarer Programmspeicher realisiert werden kann, der die Entwicklung einer Firmware und deren Test einfacher macht, siehe Abschnitt 3.1. Der Teil des Adressraums inklusive dem als externen Speicher Angesprochenen, den der Chip mitbringt, ist als Von-Neumann-Architektur realisiert, das heißt Programmcode und Datensegment referieren auf denselben Speicher von 0x0000 bis 0x3fff. Daneben lässt sich allerdings noch externer Speicher anschließen, der über die genannten PSEN- bzw. RD/WR-Leitungen gesteuert werden kann, sodass hier eine Harvard-Architektur möglich ist.

Damit lassen sich bei der 128-Pin-Variante des FX2 bis zu zwei weitere Chips betreiben, in denen 40 KB externen RAMs zwischen 0x4000 und 0xdfff sowie 48 KB Code-ROM von 0x4000 bis 0xffff enthalten sein können. Der Bereich ab 0xe000 ist vom FX2 fast vollständig für USB-Kommunikationsaufgaben reserviert und wird als **__xdata** angesprochen.

2.1.4. Stack

Der Stack der laufenden Firmware wird über einen 1 Byte großen Stack-Pointer (SP) verwaltet, der im SFR-Raum liegt. Die Architektur schreibt vor, dass über ihn indirekt auf den Speicher, also die unteren 256 Bytes, zugegriffen wird. Der Stack wächst von kleinen Adressen in Richtung größerer Adressen. Da der verfügbare Stackbereich relativ klein ist, sollte stark rekursive Progammierung, sowie das Anlegen größerer Arrays auf

[7] Port P2 beim originalen 8051, MPAGE in der Cypress-Dokumentation und _XPAGE bei SDCC in Anlehnung an die Bezeichnung bei Siemens' C500-Chip-Familie genannt [1, 3.18.1.2 und 4.1.1]; das Register befindet sich beim FX2 an Adresse 0x92 im SFR-Space [3, S. 388]

dem Stack, das heißt lokal innerhalb von Funktionen, verzichtet werden. Wenn der Stack-Pointer überläuft, wird kein Signal an die Firmware oder Peripherie des Chips gesendet, sondern mit der Ausführung des Programmcodes fortgefahren, das heißt die Register `R0` bis `R7` der Registerbänke 0 bis 3 überschrieben und die Firmware damit in einen inkonsistenten Zustand gebracht.

Obwohl Daten aus den unteren 128 Byte des Speichers direkt adressiert werden können und ein Zugriff schneller geschieht, ist es bei einer größeren Firmware sinnvoll, den Stack-Pointer initial in diesen Bereich zu legen, damit ein größerer Stack-Bereich zur Verfügung steht. Vorteilhaft dafür ist, dass auf die unteren 128 Byte ebenfalls indirekt zugegriffen werden kann. SDCC verfolgt standardmäßig diese Strategie, siehe Abb. 4, jedoch kann dieses Verhalten durch Kommandozeilenparameter gesteuert werden.

Da der Stack-Pointer für den Speicherzugriff wie Register `R0` oder `R1` bei einer `MOVX`-Instruktion, siehe Abschnitt 2.1.2, verwendet wird, kann durch Setzen des `MPAGE`-Registers der Stack auch außerhalb des internen Speichers gelegt werden und ihm damit eine volle Speicherseite von 256 Byte zur Verfügung stehen. Ein Stack-Overflow ist hier allerdings nicht weniger verheerend als beim Stack in internem Speicher; `MPAGE` wird nicht automatisch inkrementiert.

3. Compiler

Der Small Devices C Compiler ist ein optimierender C-Compiler für 8-Bit-Mikroprozessoren. Er unterstützt über verschiedene Backends eine Reihe von Architekturen, neben MCS51 (zu der der 8051 gehört) auch Chips der Dallas-DS80C390-, Freescale-HC08- und Zilog-Z80-Familien.

SDCC stellt neben dem Standarddatentyp **`char`** der Größe 8 Bit noch die Typen **`bool`** (1 Bit), **`short`** (via Kommandozeile umschaltbar zwischen 8 und 16 Bit), **`int`** (16 Bit) und **`long`** (32 Bit) als Ganzzahldatentypen zur Verfügung. Trotz fehlender nativer Unterstützung von Fließkommazahlen durch den Chip kann SDCC mit Variablen des Typs **`float`** umgehen. Dies ist möglich durch eine Reihe von Unterstützungsfunktionen, die SDCC mitbringt und bei Benutzung von Floating-Point-Berechnungen mit in die Firmware einkompiliert und die Berechnungen durch adäquate Aufrufe dieser Funktionen ersetzt. Die Größe der Firmware wächst dabei allerdings nicht unerheblich, weshalb es ratsam ist, möglichst auf Floating-Point-Berechnungen zu verzichten.

Abweichend vom C99-Standard enthält SDCC keine Unterstützung für den Typ **`long long`** oder **`double`** (beide 64 Bit), eventuell da dafür die verfügbare Registeranzahl mit acht Registern pro Bank zu niedrig ist, um sie auf einem 8-Bit-Controller effektiv zu bearbeiten. Ebenfalls eingeschränkt ist die Benutzung von **`struct`** und **`union`**. Hier fehlt die Unterstützung für die kopierende Zuweisung und damit ebenso die Möglichkeit, Strukturen oder Vereinigungen als Parameter oder Rückgabewerte von Funktionen zu benutzen.

Als Problem stellt sich häufig die durch den C-Standard beispielsweise beim Vergleich von **`signed char`** und **`unsigned char`** oder für Argumente zu `vararg`-Funktionen verlangte Typerweiterung nach **`int`** dar, da dieser 16-Bit-Datentyp nur langsam von einem 8-Bit-Controller – nämlich durch Emulation über zwei 8-Bit-Register – verarbeitet werden kann. Diese Emulation geschieht durch eine vom Compiler **`inline`** eingefügte Hilfsroutine. Um dies zu vermeiden, erlaubt SDCC ein nicht standardkonformes, explizites

Zurückcasten nach **char**. Das Verhalten ist durch Kommandozeilenoptionen allerdings umstellbar.

3.1. Objekt-Code

Der Einstiegspunkt in die Firmware ist die Methode **void** `main()`, die im Gegensatz zu Programmen, die auf Betriebssystemen laufen, weder einen Rückgabewert noch Argumente besitzt. Deshalb unterstützt SDCC die andere von C vorgegebene Signatur **int** `main(`**int** `argc,` **char** `**argv)` nicht.

Der Compiler generiert Objekt-Dateien im 8-Bit Intel-HEX-Format[8] (`.ihx`), einem ASCII-Format, das das binäre Firmware-Kompilat in Hexadezimaldarstellung zeilenweise als Datensätze, jeweils versehen mit einer Prüfsumme und der absoluten Adresse, an die er im Programmspeicher zu schreiben ist, darstellt. Der Cypress FX2 ermöglicht es über einen vorgeschalteten USB-Controller den verfügbaren On-Chip-Speicher (`0x0000 - 0x3fff` und `0xe000 - 0xffff`) über USB-Kontrollpakete direkt zu lesen und schreiben. Darin eingeschlossen sind FX2-spezifische Register wie `CPUCS` [3, S. 216], das Status- und Kontrollbits für die CPU bereithält. Über dieses ist es sowohl möglich, die Ausführung der Firmware zu stoppen, als auch erneut (an Code-Adresse `0x0000`) zu starten.

Für den Transfer der resultierenden `.ihx`-Datei zum Controller existieren einige Programme[9], die – im Fall des FX2 – folgenden Pseudocode umsetzen:

```
Setze Bit in CPUCS, das die CPU anhält
Beschreibe Programmspeicher an den im .ihx-File vorgesehenen Adressen
Lösche Bit in CPUCS, das die CPU anhält
```

Dies lädt die Daten in den Speicher des Controllers und startet ihn an Code-Position `0x0000` neu, die einen absoluten Sprung-Befehl zu der Adresse der vom Compiler bereitgestellten Einstiegsfunktion, welche danach `main()` aufruft und letztlich nach dessen Rückkehr in eine Endlosschleife läuft, beinhaltet. Der eigentliche Code von `main()` startet nicht an Adresse `0x0000`, da drei Bytes weiter (dies entspricht exakt der Größe einer `LJMP`-Instruktion, long jump unconditional) bereits die Interrupt-Tabelle erwartet wird, siehe Abschnitt 3.2.

3.2. Interrupts

Als Interrupt wird eine Unterbrechung des normalen Programmablaufs bezeichnet, um zeitnah auf ein Ereignis reagieren zu können. Diese Ereignisse können beispielsweise das Ankommen von Daten auf einem I/O-Port sein oder ein Timer, der nach einer eingestellten Zeitspanne Signal gibt. Die überwachten Ereignisse sind durchnummeriert, sodass eine spezielle Routine zur Behandlung des Ereignisses aufgerufen werden kann. Eine Liste der verfügbaren Interrupts geben [1, 3.9.2] und [3, Kapitel 4.1]. Den Interrupts sind ebenfalls Prioritäten zugeordnet, sodass bei mehreren gleichzeitig auftretenden Ereignissen oder eines neuen Ereignisses während ein anderes gerade bearbeitet wird, entschieden werden kann, welches Ereignis wichtiger ist und zunächst bearbeitet werden sollte.

[8] `http://microsym.com/editor/assets/intelhex.pdf`

[9] bspw. fx2_programmer (`http://volodya-project.sourceforge.net/fx2_programmer.php`) oder fxload des linux-hotplug-Projekts (`http://linux-hotplug.sourceforge.net/`)

Die Nutzung von Interrupts kann helfen, den Code aufgeräumter zu halten, da die Behandlungen von verschiedenen Ereignissen in verschiedene Funktion ausgelagert wird. Nicht behandelte Interrupts können über das Kontrollregister `IE` maskiert werden, was bewirkt, dass deren Überprüfung ausgesetzt wird, bis sie wieder demaskiert werden.

Intern werden im FX2 Interrupts durch Bits repräsentiert, die vor dem Laden jeder Instruktion überprüft werden. Ist ein solches Bit gesetzt, wird (mit in [3, Abschnitt 4.3.3] beschriebenen Ausnahmen) nicht der folgende Befehl geladen, sondern der aktuelle Program Counter auf dem Stack gespeichert und in eine sogenannte Interrupttabelle gesprungen, die typischerweise selbst aus Sprunganweisungen besteht. Diese Sprunganweisungen führen dann zu der Funktion, auch ISR (Interrupt Service Routine) genannt, die die Behandlung des Ereignisses übernimmt. Ist diese beendet, erfolgt ein Rücksprung an die gespeicherte Code-Adresse mittels der speziellen Maschineninstruktion `RETI`.

Ein Eintrag in der Interrupttabelle ist beim 8051 8 Byte groß, die Tabelle selbst beginnt bei Code-Adresse `0x03`. Ein Interrupt mit Nummer k führt also zu einem Sprung an Adresse $\texttt{0x03} + 8 \cdot k$. SDCC unterstützt die Programmierung von ISRs, indem eine Funktion mittels des Schlüsselworts **`__interrupt`**(k) als zuständig für Interrupt k markiert wird. Dies führt dazu, dass an besagter Adresse in der Interrupttabelle eine Sprunganweisung zu dieser Funktion eingefügt wird. Listing 1 zeigt eine Beispielbehandlung des Timer-1-Interrupts mit $k = 3$, der einen Überlauf des Zählers für Timers 1 signalisiert. Auf die Deklaration des dort benutzten SFRs `TCON` wird in Abschnitt 3.5.1 eingegangen.

```
1 void timer_isr() __interrupt(3) {
2   TCON &= ~(1 << 7);    /* clear timer overflow flag */
3   return;
4 }
```

Listing 1: Interrupt Service Routine für Timer 1

3.3. Pointer

Da auf Daten mit unterschiedlichen Maschinenbefehlen zugegriffen wird, abhängig davon, in welchem Speicherbereich sie liegen, muss für Pointer-Datentypen eine Unterscheidung nach Speicherbereichen ebenfalls existieren. Da der C-Standard keine nach Speicherbereichen typisierten Pointer vorsieht, werden diese sogenannten generischen Pointer von SDCC speziell behandelt und können nicht direkt dereferenziert werden. Daneben bietet der Compiler allerdings die Möglichkeit an, im Code direkt festzulegen, in welchen Speicherbereich ein Pointer zeigt, dann kann eine direkte Dereferenzierung erfolgen.

Speicheradressen haben eine Größe von 2 Byte. Bei einem nach Speicherbereich typisierten Pointer ist zur Dereferenzierung keine Zusatzinformation nötig, weshalb er ebenfalls 2 Byte groß ist. Generische Pointer hingegen werden zur Laufzeit durch vom Compiler eingefügte Funktionen angelegt und ausgewertet. Da die Bestimmung der Speichertypen, die Pointern zur Laufzeit haben, zur Compile-Zeit hinreichend nichttrivial ist, greift hier der Satz von Rice [5], der besagt, dass es dem Compiler nicht für jedes zu kompilierende Programm möglich ist, zu bestimmen, wohin ein Pointer zeigt. Anhang A bietet einen formalen Beweis dieser Aussage.

Ein generischer Pointer ist 3 Byte groß, 2 Byte enthalten die Adresse und ein Byte kodiert den Typ des Speichers, auf den über die Adresse zugegriffen wird, siehe Tabelle 2. Die vom Compiler transparent zur Dereferenzierung und Erstellung eines generischen

1. Byte	Typ	Speicherbereich	Zugriff über
0x00	**__xdata**	externer RAM	MOVX und DPTR
0x40	**__idata**	interner RAM	MOV und R0, R1
0x60	**__pdata**	externer RAM	MOVX und R0, R1
0x80	**__code**	Code-ROM	MOVC und DPTR

Tabelle 2: Kodierung der Speicherbereiche in generischen Pointern

```
/* Pointer in internem RAM, zeigt auf Objekt in externen RAM */
__xdata unsigned char * __data p;

/* Pointer in externem RAM, zeigt auf Objekt in internem RAM */
__data unsigned char * __xdata p;

/* Pointer im Code-ROM, zeigt auf Objekt in externem RAM */
__xdata unsigned char * __code p;

/* Pointer im Code-ROM, zeigt auf Objekt im Code-ROM */
__code unsigned char * __code p;

/* Generischer Pointer in externem RAM */
unsigned char * __xdata p;

/* Generischer Pointer im Standard-Adressraum */
unsigned char * p;

/* Funktionspointer in internem RAM */
char (* __data fp)(void);
```

Listing 2: Beispiele zu typisierten und generischen Pointern

Pointers eingefügten Bibliotheksfunktionen führen je nach kodiertem Speichertyp andere Maschineninstruktionen aus, um den gewünschten Zugriff zu realisieren.

3.4. Speichermodelle

SDCC unterstützt drei Speichermodelle für 8051-kompatible Controller. Sie unterscheiden sich darin, welche Speicherbereiche für Variablen und Funktionsparameter, die ohne Speicherklasse deklariert sind, benutzt werden.

Das Modell `small` legt diese Variablen in den internen RAM, d.h. **__data** oder **__idata**, je nach verfügbarem Platz. Sie können damit entweder direkt oder indirekt über die Register `R0` bzw. `R1` von allen Maschineninstruktionen angesprochen werden. Die Modelle `medium` und `large` legen Variablen ohne Speicherklassenspezifikation in den externen RAM (**__pdata** und **__xdata**), wodurch mindestens eine Indirektionsstufe für den Zugriff hinzukommt, da alle Maschineninstruktionen, ausgenommen Sprünge, `MOVX` und `MOVC`, nur auf internem RAM arbeiten. Tabelle 3 gibt einen Überblick über die dafür benötigten Zeiten und zusätzlichen Register.

Solange die Firmware die Größen der jeweiligen Speicherklassen nicht überschreitet, ist es aus Geschwindigkeitsbetrachtungen ratsam, das kleinstmögliche Speichermodell zu

Modell	Typ	Instruktionen	Zyklen [3]	benutzte Register
`small`	**`__data`**	alle	2	keine
`small`	**`__idata`**	alle	1	`R0` oder `R1`
`medium`	**`__pdata`**	`MOVX`	2	`ACC` und `R0` oder `R1`
`large`	**`__xdata`**	`MOVX`	2	`ACC` und `DPL`, `DPH`

Tabelle 3: Zugriffszeiten und benötigte Register für Variablen nicht deklarierter Speicherklassen abhängig vom gewählten Speichermodell

wählen, da hier der Zugriff mit weniger zusätzlich erforderlichen Registern beziehungsweise Taktzyklen bewerkstelligt werden kann.

3.4.1. Overlaying

Damit weniger interner Speicher durch lokale Variablen von Funktionen benutzt wird, setzt der Compiler eine Technik, die Overlaying genannt wird, ein. Hierbei wird versucht, die Anzahl der benötigten 8-Bit-Speicherplätze, welche initial der Anzahl der Variablen dieser Größe einer Funktion entspricht, dadurch zu minimieren, dass der Speicherplatz der Variablen, die ab einem Punkt in der Funktion nicht mehr verwendet werden, für neue (Zwischen-) Ergebnisse wiederverwendet wird. Dies implementiert eine Optimierung auf Basis des Konzepts der Lebendigen Variablen[10]. Eine Variable gilt an einer Stelle im Kontrollfluss der untersuchten Funktion als lebendig, solange es eine Möglichkeit gibt, dass zu einem späteren Zeitpunkt lesender Zugriff auf sie erfolgt. Ist dies nicht der Fall, so wird der enthaltene Wert (offensichtlich) nicht mehr benötigt und der Speicherplatz kann für andere Zwecke verwendet oder freigegeben werden. Wird die Speicherstelle für andere lokale Variablen wiederverwendet, so bezeichnet Overlaying diese Technik.

SDCC hält einen zur vollständigen Laufzeit der Firmware festen Speicherbereich im internen RAM frei, der nur für lokale Variablen und Parameter einer Funktion, die in diese Overlay-Kategorie fallen, reserviert ist [1, 3.8]. Dies wird nur für das Speichermodell `small` und Variablen, deren Speicherklasse nicht explizit angegeben ist, unterstützt. Probleme bereiten hier Interrupts, da Funktionen mehrfach aufgerufen werden können obwohl sie nicht rekursiv sind, und dadurch Variablen der alten Inkarnationen überschrieben werden können, weshalb diese als **`__reentrant`** markiert werden sollten, siehe Abschnitt 3.5.4.

3.5. Wichtige Ergänzungen zum C-Standard

3.5.1. Deklarationen für physikalische Speicheradressen

Die Firmware arbeitet im Gegensatz zu einem typischen C-Programm mit absoluten physikalischen Speicheradressen, um auf globale Variablen und insbesondere SFRs zuzugreifen. Zwar werden globalen Variablen eines C-Programms beim Prozessstart auf einem Betriebssystem ebenfalls feste Adressen zugeordnet, jedoch können diese durch Benutzung virtuellen Speichers und von ASLR[11] bei moderneren Betriebssystemen zwischen zwei Prozessstarts wechseln. Der C-Standard macht hierzu keine Vorgaben, weswegen er

[10] siehe auch Abschnitt „Live Variable Analysis“ in [8]

[11] Address Space Layout Randomization, randomisiert die Startadressen der Speicherblöcke Text, Heap, Stack und teilweise auch der Bibliotheksfunktionen eines Prozesses bei jedem Start

```
PHYSDECL     ::=  STOCLASSMOD? MEMTYPE "at" NUMBER VARDECL ";";
STOCLASSMOD  ::=  "extern";
MEMTYPE      ::=  "__bit" | "__sbit" | "__sfr" | "__data"
                | "__xdata" | "__idata" | "__pdata";
```

`STOCLASSMOD` Storage-class modifier; benötigt, damit der Compiler keinen Speicherplatz für die Variable im deklarierte Speicherbereich anlegt und initialisiert

`VARDECL` Standardvariablendeklaration, inklusive weiteren Modifiern wie **`const`** oder **`volatile`**; der Typ muss nur bei `MEMTYPE` **`__xdata`** explizit angegeben werden

Tabelle 4: Syntax zur Bindung physikalischer Speicheradressen an C-Identifier

auch keine Schlüsselworte bereithält, um direkt auf physikalische bzw. absolute Adressen im Speicher zu referieren. Dies zeigt sich beispielsweise darin, dass das Verhalten als implementierungsspezifisch und nicht fest definiert ist, wenn Pointern Integerwerte oder umgekehrt zugewiesen werden [6, J.3.7, S. 507].

Um den hier für die Mikrocontrollerprogrammierung bestehenden Mangel auszugleichen, bietet SDCC die in Tabelle 4 angegebene Syntax, in der der Zugriff auf Register und Speicher variabler Größe festgelegt und diesen Speicherstellen damit ein Identifier zugeordnet werden kann.

3.5.2. __critical

Dieses Schlüsselwort markiert einen Bereich oder eine Funktion als kritisch. Damit wird die Ausführung von Interrupt-Requests in dem markierten Code-Bereich unterbunden. Alle aufgetretenen Interrupts werden nach Verlassen des kritischen Bereichs in Reihenfolge ihrer Prioritäten ausgeführt. Intern erreicht der Compiler dieses Verhalten durch Löschen eines Bits im `IE`-SFR (Interrupt-Enable), was global die Bearbeitung aller Interrupts deaktiviert. Das Setzen des Bits startet die Interruptbehandlung wieder.

3.5.3. __naked

Wie bei anderen Architekturen existiert für den 8051 eine Konvention, die besagt welche Registerinhalte beim Aufruf einer Funktion gespeichert und wiederhergestellt werden müssen. Das Speichern erfolgt durch ein Ablegen der Werte auf dem Stack und anschließendes Erhöhen des Stack-Pointers.

Standardmäßig speichert SDCC die Register `R0` bis `R7` der Registerbank, die die aufgerufene Funktion benutzt, sodass die aufgerufene Funktion diese Speicherplätze frei benutzen darf. Ebenfalls benutzt werden können die Register `DPL`, `DPH` und `B` sowie der Akkumulator jedoch unter der Beachtung, dass diese Register für den Rückgabewert und für den ersten übergebenen Parameter (falls er größer als 8 Bit ist, wird in dieser Reihenfolge allokiert) benutzt werden. Weitere Parameter einer Funktion werden entweder auf dem Stack abgelegt oder – abhängig vom gewählten Speichermodell – standardmäßig im **`__data`**- bzw. **`__xdata`**-Bereich zwischengespeichert.

Ist eine Funktion als **`__naked`** markiert, so generiert der Compiler keinen Code um die Registerinhalte vor einem Aufruf dieser Funktion zu speichern und danach wiederherzustellen. Außerdem geht er davon aus, dass der Programmierer die Funktion direkt in Assembler schreibt, damit dieser Kontrolle über die benutzten Register hat. Dies spart

Rechenzeit beim Funktionsaufruf und ist sowohl für ISRs als auch Funktionen, die sehr nah an der Hardware arbeiten und oft aufgerufen werden müssen, zum Beispiel für I/O, nützlich.

3.5.4. __reentrant

Falls es Funktionen gibt, die während ihrer Ausführung durch Interrupts unterbrochen werden, aber durch die entsprechende ISR erneut aufgerufen werden können, sodass es zwei Inkarnationen dieser Funktion auf dem Stack gibt, verlangt SDCC, dass diese Funktion mit **`__reentrant`** markiert wird, damit der Compiler Optimierungen unterlassen kann, die sonst zu Inkonsistenzen im Speicher führen würden.

Speziell betreffen diese Optimierungen die Parameterübergabe an eine so markierte Funktion, welche nicht mehr über Overlays (3.4.1) oder internen beziehungsweise externen RAM (3.5.3) verläuft, sondern über den Laufzeitstack. Damit ist sichergestellt, dass keine Argumente durch Funktionen, die ein Interrupt aufruft, überschrieben werden können.

3.5.5. __using

Mit dem Schlüsselwort **`__using`** kann die aktuelle Registerbank für eine Funktion gewählt werden. Ist es nicht angegeben, wird Bank 0 benutzt. Die verschiedenen Registerbänke erlauben es in einer so markierten Funktion, ohne vorheriges Sichern der alten Register freien Speicher zur Verfügung zu stellen. Sie sind damit ähnlich zu dem Register Window auf SPARC-Architekturen [9], wenn auch mangels einer automatischen Verschiebung des aktiven Fensters und gemeinsam durch das vorige und jetzt aktive Fenster genutzten Registern nicht direkt vergleichbar.

Vor allem bei ISRs, bei denen in der Regel unbekannt ist, welcher Code-Pfad mit welcher Stack-Größe verlassen wurde um die Routine auszuführen, kann es hilfreich sein, eine ansonsten unbenutzte Registerbank zu definieren, um einem eventuellen Stack-Überlauf durch das Speichern der Register vorzubeugen und zusätzlich Rechenzeit bei den häufig zeitkritischen ISRs zu sparen.

3.6. Bibliotheksfunktionen

Mathematik Die im C-Standard definierten Mathematik-Funktionen arbeiten auf Floating-Point-Typen, die der 8051-Controller mangels FPU nicht in Hardware unterstützt. Sie werden deshalb von der Compiler-Bibliothek durch Behandlung der IEEE-754-Darstellung [4] der 32-Bit-**`float`**-Werte als Ganzzahlen implementiert. Die Funktionen wurden für SDCC aus GCCs[12] `floatlib` übernommen, sind deshalb als ANSI-C implementiert und enthalten keine Controller-spezifischen Optimierungen.

I/O Die durch den C-Standard definierten I/O-Funktionen benötigen entweder einen Dateideskriptor für die Ein- und Ausgabe oder implizieren die Standardein-/ausgabe des C-Programms. Diese Deskriptoren existieren mangels Dateisystems und eines Standardkanals bei einem Mikrocontroller nicht, weshalb die Funktionen `getchar()` und

[12] GNU Compiler Collection, `http://gcc.gnu.org/`

`putchar()`, welche die Basis für komplexere I/O-Funktionen bilden, vom Programmierer der Firmware definiert werden müssen. Die von SDCC bereitgestellten Standard-I/O-Funktionen nutzen danach diese Primitive.

printf `printf` ist eine sehr umfangreiche Funktion. Sie formatiert Werte nach selektierbaren Kriterien mit vielen Möglichkeiten, Einfluss auf das Format zu nehmen. So unterstützt sie nach dem C99-Standard 18 Formatcodes[13], eine Reihe Flags und Angaben zu Präzision und Länge der formatierten Ausgabe, sowie 8 Modifier[14], die den Typ des zu formatierenden Arguments angeben [2, S. 139] [6, S. 274-280]. Da bereits einige der Kombinationen durch die fehlende Unterstützung von 64-Bit-Datentypen durch SDCC an Bedeutung verlieren, fehlen diese in der Compiler-Bibliothek. Weiterhin werden selten alle dieser Formatcodes und Modifier von einer Mikrocontrollerfirmware benötigt, weshalb SDCC sechs verschiedene Varianten der `printf`-Funktion bereitstellt, die jeweils unterschiedlichen Funktionsumfang und damit auch Größe im Programmspeicher haben [1, 3.17.2.1.2]. So enthalten nur vier dieser Varianten Floating-Point-Unterstützung, lediglich zwei können mit Byte-Argumenten auf dem Stack umgehen und einer sehr rudimentär gehaltenen Variante fehlt die Unterstützung für Modifier, **`long`** und **`float`**. Diese Vielfalt erlaubt es dem Programmierer, die Variante auszuwählen, die für seine Zwecke ausreichend ist, ohne Platz zu verschwenden.

malloc Die Nutzung dynamischer Speicheranforderung ist eher selten in einer Mikrocontrollerfirmware, da sie die vollständige Kontrolle über den gesamten verfügbaren Speicher bereits besitzt. SDCCs Bibliothek implementiert diese Standardfunktion trotzdem. Mangels Betriebssystem, von dem Speicher anzufordern ist, muss hier die Adresse des Speicherpools, den `malloc()` bzw. `realloc()` verwenden, fest einkompiliert werden. Dies ist ein Speicherblock von zur Compile-Zeit festgelegter Größe (Standard: 1 KB) an einer festen Adresse im externen RAM, aus dem `malloc()` kleinere Speicherblöcke als allokiert markiert und an den Aufrufer zurückgibt. Dabei arbeitet SDCCs `malloc()` ohne Optimierungen wie sie in `dlmalloc()`[15] oder ähnlichen Varianten in Betriebssystemen verwendet werden. So werden beispielsweise keine Cluster von kleinen Blöcken gebildet und mangels Memory Management Unit auch kein Memory-Mapping verwendet. Damit ist SDCCs `malloc()` deutlich anfälliger gegenüber Fragmentierung als die etablierten Varianten.

Ein Aufruf von `malloc(`n`)` führt dazu, dass aus den noch nicht allokierten Bereichen der erste zusammenhängende Block der Größe $n+k$ gesucht wird, wobei k die Größe der für die Verwaltung der einzelnen allokierten Speicherblöcke nötigen Variablen ist. Ist ein solcher Block an Adresse A gefunden, so werden in den ersten k Bytes die Verwaltungsinformationen abgelegt und anschließend die Adresse $A+k$ dem Aufrufer zurückgegeben, an der er jetzt n Bytes Speicherplatz zur Verfügung hat. Abbildung 2 zeigt den Aufbau eines so allokierten Blocks. Die Pointer `prev` und `next` zeigen je auf den vorigen und nächsten allokierten Block, falls vorhanden. Diese Liste ermöglicht die oben genannte (lineare) Suche nach freiem Speicher innerhalb des `malloc`-Arrays.

[13] `%` gefolgt von einem der Zeichen `a`, `A`, `c`, `d`, `i`, `e`, `E`, `f`, `F`, `g`, `G`, `o`, `s`, `u`, `x`, `X`, `p` und `n`
[14] `hh`, `h`, `l`, `ll`, `j`, `z`, `t` und `L`
[15] `http://gee.cs.oswego.edu/dl/html/malloc.html`

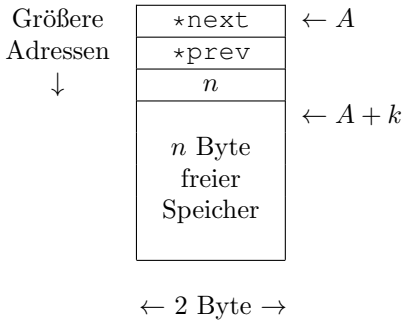

Abbildung 2: Block allokierten Speichers durch SDCCs `malloc(`n`)`

exit Die Funktion `exit(`k`)` bewirkt normalerweise die Beendigung des Programms mit dem Exit-Code k. Eine Beendigung der `main()`-Routine der Firmware führt dazu, dass in den vom Compiler generierten Initialcode zurückgesprungen und eine Endlosschleife ohne Inhalt ausgeführt wird.

4. Beispiele

4.1. GGT

Eine Firmware, die den Euklidsche Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier Zahlen a und b implementiert, sei hier vorgestellt. Die Firmware erwartet an den je 8 I/O-Leitungen, die mit `IOA` bzw. `IOB` bezeichnet sind, je eine binär kodierte Zahl, die sie ausliest und an die Funktion `ggt()` übergibt. Das binäre Ergebnis wird auf die 8 I/O-Leitungen des `IOC`-Ports gelegt.

Die Deklarationen in Zeilen 11 bis 13 teilen dem Compiler mit, dass die 8-Bit-I/O-Register an den angegebenen Adressen als SFR, das heißt per direktem Speicherzugriff, anzusprechen sind.

```
1  char ggt(char a, char b) {
2    while (a != b) {
3      if (a > b) {
4        a -= b;
5      } else {
6        b -= a;
7      }
8    }
9    return a;
10 }
```

```
11 __sfr at 0x80 IOA;
12 __sfr at 0x90 IOB;
13 __sfr at 0xa0 IOC;
14
15 void main() {
16   IOC = ggt(IOA, IOB);
17 }
```

4.2. Timer / Interrupts

Ein komplexeres Beispiel soll eine Funktion vergleichbar mit der inzwischen aus dem POSIX-Standard entfernten Funktion `usleep()` [7, Anhang B.3.1] mittels eines Timers implementieren. `usleep(`k`)` setzte die Ausführung des laufenden Threads beziehungsweise Prozesses für mindestens k Mikrosekunden aus.

Dieses Beispiel implement das Warten mangels eines Betriebssystems, das sonst den Thread für die gegebene Zeit aus seiner Liste der auf eine CPU-Zeitscheibe wartetenden Threads entfernt hätte, mittels Busy-Waiting auf den Interrupt, der auftritt, wenn der 16-Bit-Timer 2 (spezifisch für den FX2, der originale 8051 hatte nur zwei 16-Bit-Timer, nummeriert 0 und 1) überläuft. Schwierigkeiten stellen hierbei die ebenfalls FX2-spezifische, aus $\{12, 24, 48\}$ MHz wählbare Taktfrequenz sowie eine Einstellung im `CKCO`-Register des Controllers dar, die beide die Anzahl der Timer-Triggers pro Zeit beeinflussen. Die folgend dargestellte Funktion berücksichtigt beides. Auf dem Standard-8051 mit 12 MHz und 0,5 bis 1 MHz Instruktionen pro Sekunde, wären die Zeilen 12 bis 16 zur Konvertierung von µs nach Instruktionszyklen unnötig.

```
1  #include "regs.h"
2
3  static BYTE p = 0;            /* timer 2 overflow counter */
4  /* IRQ 5 is asserted when timer 2 overflows */
5  void timer2_isr() __interrupt(5) {
6    T2CON &= ~(1 << 7);         /* clear timer 2 overflow flag */
7    p++;
8  }
9
10 /* us may be maximal floor(2^16 / 12) = 5461 micro seconds */
11 void usleep(WORD us) {
12   /* the CPU may run at 12, 24 or 48 MHz => ffac = 0, 1 or 2 */
13   BYTE ffac = (CPUCS >> 3) & 0x03; /* log2(timer ticks per µs) */
14   us <<= ffac;                 /* clock ticks to sleep */
15   if (CKCO & (1 << 5))         /* if this flag is set, timer 2 */
16     us += us + us;             /*   ticks at thrice the speed */
17   /* timer 2 shall overflow after this number of ticks */
18   us = -us;                    /* us is unsigned */
19   TL2 = us & 0xff;             /* initialize value of timer 2 */
20   TH2 = us >> 8;
21   p = 0;                       /* reset overflow counter */
22   IE |= 1 << 7;                /* enable interrupts globally */
23   IE |= 1 << 5;                /* enable timer 2 overflow IRQ */
24   T2CON = 1 << 2;              /* start timer 2 */
25   while (!p) ;                 /* wait */
26   T2CON &= ~(1 << 2);          /* stop timer 2 */
27 }
```

Der Inhalt von `regs.h` ist in Listing 3 im Anhang wiedergegeben. Die angegebene Funktion `usleep` initialisiert Timer 2 auf die nötige Anzahl an Clock Ticks, damit er in `us` Mikrosekunden überläuft, wartet dann in einer leeren Schleife darauf, dass der Interrupt Request von `timer2_isr()` bearbeitet und damit die globale Variable `p` gesetzt wird, stoppt schließlich Timer 2 und kehrt zurück. Das Technical Reference Manual [3] gibt Auskunft über die benutzten Bits der SFRs und des FX2-spezifischen Registers `CPUCS`.

Zwar könnte in der Schleifenbedingung in Zeile 25 direkt das Overflow-Flag des Timers 2 abgefragt und somit ohne die Benutzung des Interrupts mit dem gleichen Ergebnis ausgekommen werden, jedoch ist diese Variante insofern erweiterbar, als es der Zähler `p` erlaubt, dass mehrere Durchläufe des Timers 2 abgewartet werden und somit für längere Zeiträume als 5461 µs bei 48 MHz Chip-Takt und `CKCO.5` gesetzt gewartet werden kann (maximal $256 \cdot 5461$ µs $\approx$ 1,4 Sekunden).

Nicht berücksichtigt bei der Zeitmessung sind hier die Dauer der Befehle zum Initialisieren und Starten des Timers sowie die der Interrupt Service Routine `timer2_isr()`. Da diese jedoch dokumentiert sind, wäre es kein Problem, diese Funktion vollständig akkurat zu programmieren, indem diese Zeiten von `us` vor Zeile 18 abgezogen werden.

Eine weitaus weniger resourcenlastige und häufiger anzutreffende Implementierung ebenfalls über Busy-Waiting ist eine Schleife, die eine bestimmte Anzahl `NOP`s (no operation, eine Instruktion, die für eine bestimmte Zeit nichts tut) enthält und oft genug wiederholt wird, um die gewünschte Zeit zu warten. Jedoch könnte die hier vorgestellte Funktion auch ohne aktives Warten, sondern mit periodischer Überprüfung des Timer-2-Overflows (oder direkter Benutzung des IRQs), währenddessen die Firmware an anderer Stelle arbeitet, genutzt werden, um nach einer bestimmten Zeit ein Signal zu erhalten. Dies ist mit einer solchen NOP-Warteschleife nicht möglich.

A. Beweis zu Unentscheidbarkeit bei generischen Pointern

Sei zu Alphabet Σ die Menge aller (evtl. ungültigen) SDCC-Programme Σ^*. Für eine Firmware $c \in \Sigma^*$ sei $x \in \{0,1\}^*$ die Kodierung der Eingabendaten vom Controller. Das Simulationsprogramm $sim_c(x)$ gebe 1 aus, falls $c(x)$ einem (nicht-generischen) Pointer an einer Code-Stelle zur Laufzeit Adressen aus ≥ 2 verschiedenen Speicherbereichen zuweist und später auch dereferenziert, sonst 0. In beiden Fällen halte $sim_c(x)$. Ein für eine solche Stelle vom Compiler generierter Maschinencode würde bei mindestens einem Durchlauf auf einen falschen Speicherbereich zugreifen.

Zu einem $w \in \{0,1\}^*$ bezeichne im Folgenden M_w die von w kodierte Turingmaschine (TM). Sei weiter $P = \{\ c \in \Sigma^* \mid \exists x : sim_c(x) = 1\ \}$ die Sprache all solcher Programme, die generische Pointer benötigen. $C(S)$ aus dem Satz von Rice entspricht hier P. Offensichtlich ist P nichttrivial, d.h. $\varnothing \neq P \neq \Sigma^*$, siehe dazu auch Abschnitte 2.1 und 3.3.

Da SDCCs Variante von C turingvollständig ist, existiert $c_U \in \Sigma^*$, welches die universelle Turingmaschine U simuliert, d.h.

$$\forall w \in \Sigma^*, x \in \{0,1\}^* : c_U(w,x) \text{ terminiert } \iff U(w,x) \text{ terminiert}$$

Falls $c_U(w,x)$ terminiert, dann $c_U(w,x) = U(w,x) = M_w(x)$. O.B.d.A. gelte $c_U \notin P$ (falls $c_U \in P$, betrachte im Folgenden $\Sigma^* \setminus P$).

Seien $c_0 \in P$ (existiert, da P nichttrivial) und $w \in \{0,1\}^*$ beliebig. Algorithmus $c_w \in \Sigma^*$ kodiere eine Firmware, die $c_U(w,w)$ gefolgt von $c_0(x)$ unter Eingabe x berechnet. Dann gilt

$$c_w \in P \iff U(w,w) \text{ terminiert } \iff w \in H' = \{\ w \in \{0,1\}^* \mid M_w(w) \text{ terminiert }\}$$

Annahme P entscheidbar $\implies H'$ entscheidbar. Dies ist ein Widerspruch, da H' das spezielle Halteproblem und damit unentscheidbar ist. Also folgt, dass P nicht entscheidbar ist. Der Compiler kann somit nicht für jedes Eingabeprogramm die richtige Entscheidung treffen und terminieren.

B. Code-Listings

```
1  typedef unsigned char BYTE;
2  typedef unsigned int  WORD;
3
4  /* CPU control and status register, FX2-specific */
5  extern __xdata at 0xe600 volatile BYTE CPUCS;
6
7  __sfr at 0x8e CKCO;  /* clock output control register */
8  __sfr at 0xa8 IE;    /* interrupt control register */
9  __sfr at 0xc8 T2CON; /* control register for 16-bit timer 2 */
10 __sfr at 0xcc TL2;   /* timer 2 value, low byte */
11 __sfr at 0xcd TH2;   /* timer 2 value, high byte */
```

Listing 3: `regs.h`

C. Abbildungen

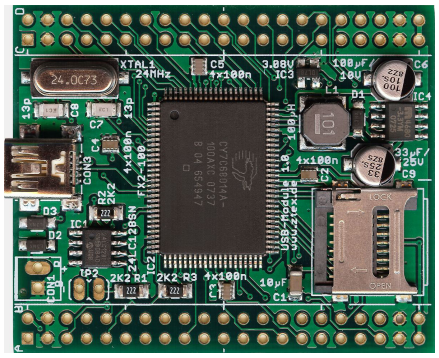

Abbildung 3: FX2-Entwicklerboard

```
      0 1 2 3 4 5 6 7 8 9 A B C D E F
0xf0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xe0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xd0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xc0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xb0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0xa0:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x90:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x80:|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|
0x70:|Q|Q|Q|Q|S|S|S|S|S|S|S|S|S|S|S|S|
0x60:|c|c|c|c|c|c|c|c|c|Q|Q|Q|Q|Q|Q|Q|
0x50:|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|
0x40:|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|
0x30:|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|
0x20:|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|
0x10:|b|b|b|b|b|b|b|b|b|b|c|c|c|c|c|c|
0x00:|0|0|0|0|0|0|0|0|a|a|b|b|b|b|b|b|

0-3: Register Banks      Q: Overlay
a-z: Data (globals,      S: Stack
     parameters and
     contants)
```

Abbildung 4: Layout des internen RAMs einer Firmware durch SDCC

x	8x	9x	Ax	Bx	Cx	Dx	Ex	Fx
0	**IOA**	**IOB**	**IOC**	**IOD**	**SCON1**	PSW	ACC	B
1	SP	**EXIF**	**INT2CLR**	**IOE**	**SBUF1**			
2	DPL0	**MPAGE**	**INT4CLR**	**OEA**				
3	DPH0			**OEB**				
4	**DPL1**			**OEC**				
5	**DPH1**			**OED**				
6	**DPS**			**OEE**				
7	PCON							
8	TCON	SCON0	**IE**	**IP**	**T2CON**	**EICON**	**EIE**	**EIP**
9	TMOD	SBUF0						
A	TL0	**AUTOPTRH1**	**EP2468STAT**	**EP01STAT**	**RCAP2L**			
B	TL1	**AUTOPTRL1**	**EP24FIFOFLGS**	**GPIFTRIG**	**RCAP2H**			
C	TH0		**EP68FIFOFLGS**		**TL2**			
D	TH1	**AUTOPTRH2**		**GPIFSGLDATH**	**TH2**			
E	**CKCON**	**AUTOPTRL2**		**GPIFSGLDATLX**				
F			**AUTOPTR-SETUP**	**GPIFSGLDATLNOX**				

Abbildung 5: Special Function Register des Cypress-Chips aufgeschlüsselt nach Adresse; fett gedruckte Bezeichner sind nicht Teil des ursprünglichen 8051 [3]

Literatur

[1] SDCC 2.9.7 (20-08-2010), *SDCC Usage Guide*, http://sdcc.sourceforge.net/doc/sdccman.html/

[2] Herbert Schildt (2007), *C/C++ Ge-packt*, 3. Auflage. Heidelberg: mitp.

[3] Cypress Semiconductor (2008), *EZ-USB® Technical Reference Manual*, http://www.cypress.com/?docID=17916

[4] IEEE Std 754 (1985), *IEEE Standard for Binary Floating-Point Arithmetic.*

[5] H. G. Rice, *Classes of Recursively Enumerable Sets and Their Decision Problems*, Transactions of the American Mathematical Society, Vol. 74, No. 2 (März, 1953), S. 358-366

[6] ISO/IEC 9899:TC3 Information technology - Programming Language C, WG14/N1256 Committee Draft, C99 + TC1 + TC2 + TC3 (2007-09-07), http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

[7] POSIX.1-2008 bzw. IEEE Std 1003.1™-2008, http://www.opengroup.org/onlinepubs/9699919799/

[8] LLVM Compiler Infrastructure Project, *The LLVM Target-Independent Code Generator*, http://llvm.org/docs/CodeGenerator.html

[9] SPARC International Inc® (1992), *The SPARC Architecture Manual – Version 8*, http://www.sparc.org/standards/V8.pdf