2 Der Kern des Mikrocontrollers: Die CPU

2.1 Zielsetzung

Im vorliegenden Kapitel wird der Kern des 16-Bit Mikrocontrollers HCS12 beschrieben, der aus der Recheneinheit (CPU, Central Processing Unit), dem Interruptmodul (INT), dem Module-Mapping-Control-Modul (MMC), dem Multiplexed-External-Bus-Interface-Modul (MEBI), dem Breakpoint-Modul (BKP) und dem Background-Debug-Modul (BDM) besteht. ´Da es für den Anwender nur wichtig ist, über Teile Bescheid zu wissen, beschränken wir uns hier auf die Beschreibung des Programmiermodells, der Adressierungsmodi, einer tabellarischen Befehlsübersicht und der Interruptverarbeitung. Dies erscheint uns für den Einsatz des Mikrocontrollers notwendig zu sein. Für weitere detaillierte Informationen sei auf [1] verwiesen. Das im folgenden erwähnte gilt natürlich im wesentlichen auch für die 68HC12-CPU.

2.2 Eigenschaften des HCS12-Kerns

Bei der HCS12-CPU handelt es sich um eine leistungsfähige Recheneinheit, die mit einer Daten- und Adressbreite von 16 Bit arbeitet. Das bedeutet, dass Daten mit einer Wortbreite von 16 Bit in einem Befehl verarbeitet werden können und die internen und externen Datenbusse 16 Leitungen aufweisen. Die maximale Taktfrequenz, mit der der Kern arbeitet, beträgt 33 MHz. Dies bedeutet, dass ein Taktzyklus 30,3 ns dauert. Damit die CPU effizient arbeiten kann, wurde eine 3-stufige Befehlspipeline (Instruction Queue) eingebaut. Viele Adressierungsarten, ein weitestgehend orthogonaler Befehlssatz und effektive Befehle für den Einsatz von Hochsprachen (z. B. C und C++) garantieren dichten und schnellen Programmcode. Der Speicher kann mit Hilfe der Software konfiguriert und an unterschiedliche Speicheradressen im Adressraum gelegt werden (Memory Mapping). Mit Hilfe der externen Busschnittstelle, kann man zusätzlichen Speicher im 8-Bit- oder im 16-Bit-Modus ansprechen (Multiplex- oder Non-Multiplex-Betrieb). Ein in Hardware implementiertes Breakpoint-Modul bietet zwei unterschiedliche Betriebsarten: Im Dual-Address-Modus wird die aktuelle Adresse im Programmzähler (PC) mit zwei einstellbaren Adressen verglichen. Im Full-Breakpoint-Modus kann die Überprüfung auf eine Adresse und ein Datum vorgenommen werden. Das Background-Debug-Modul stellt eine Eindrahtverbindung mit einem Debugger her, mit dem der Chip getestet und der Speicher beschrieben, programmiert und ausgelesen werden kann.

Die Peripheriefunktionen, die auf einem HCS12-Derivat vorhanden sind, werden über einen Standardbus an den HCS12-Kern angebunden. Bei den 68HC12-Derivaten gibt es den IP-Bus Bus in dieser Form noch nicht. Dieser IP-Bus wird auf fast allen neuen Mikrocontrollern von Motorola verwendet, die in 0.25 µm gefertigt werden. Damit können die Peripheriefunktionen innerhalb der Mikrocontrollerfamilien leicht ausgetauscht und

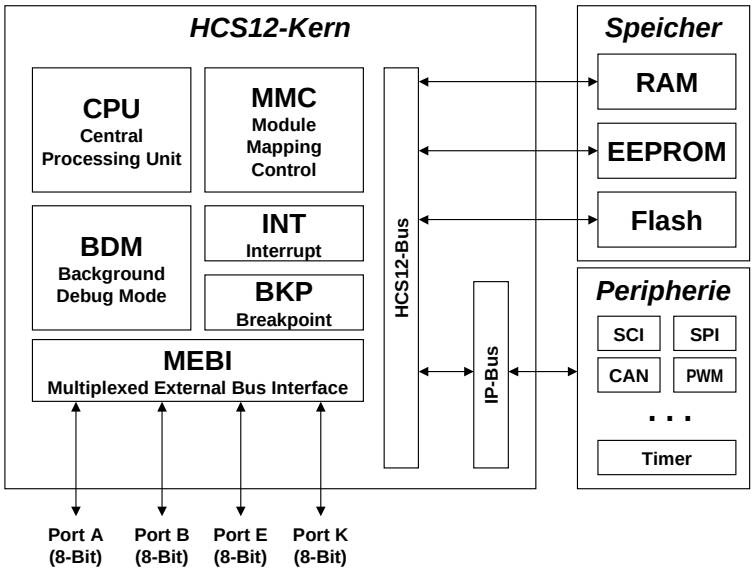

Abbildung 1.1: Blockdiagramm des HCS12-Kerns

erweitert werden. Nur auf den Speicher greift die CPU direkt mit dem HCS12-Bus zu.

Wie in Abb. 1-1 dargestellt, fasst man im HCS12-Kern sechs Module und zwei Busse zusammen. Die CPU (Befehlssatz und Adressierungsmodi) und das Interruptmodul werden im folgenden ausführlicher dargestellt. Zusätzlich zum eigentlichen Kern sind auf dem Mikrocontroller noch verschiedene Speicherarten und Peripheriemodule vorhanden. Die Speichermodule, die bei den HCS12-Derivaten aus RAM, Daten-Flash-EEPROM und Programm-Flash-EEPROM bestehen können, werden über den HCS12-Bus angesprochen. Für den Zugriff auf die internen Speicher sind keine Wartezyklen (Wait States) notwendig. Sowohl 8-, als auch 16-Bit-Zugriffe auf den internen Speicher benötigen einen Taktzyklus. Die Peripherie ist über den IP-Bus an den Kern angebunden. Da der IP-Bus bei allen 8-, 16- und 32-Bit-Mikrocontrollern bis auf die Daten- und Adressbreite gleich ist, können die Peripheriebausteine, die für den IP-Bus entworfen wurden, auf allen Mikrocontrollern eingesetzt werden. Das bietet den Vorteil, dass die Peripherie auf den unterschiedlichen Mikrocontrollern kompatibel sein kann.

2.3 Das Programmiermodell der HCS12-CPU

Das Programmiermodell der 68HC12- und HCS12-CPU ist identisch und beide sind kompatibel zum 68HC11. Das gewährleistet Softwarekompatibilität bei diesen drei

Architekturen. Unter dem Programmiermodell versteht man den Aufbau der Akkumulatoren und der restlichen Register der CPU.

2.3.1 Die Akkumulatoren A und B

Die CPU12 ist im Gegensatz zu RISC-Architekturen eine Akkumulator-Architektur. Das heisst, dass sich Operanden und Resultate der meisten Operationen in den Akkumulatoren befinden. Die CPU12 hat die beiden 8 Bit breiten Akkumulatoren A und B. Beide Akkumulatoren sind bei fast allen Befehlen austauschbar. Einige Befehle verwenden die Kombination von A und B. Dieses Doppelregister wird D genannt und ist 16 Bit breit. Die 8 Bit breiten Akkumulatoren sind noch ein Relikt aus der Zeit des 8-Bit-68HC11-Mikrocontrollers. Aus Kompatibilitätsgründen wurde das Registermodell beibehalten. Man könnte vermuten, dass 8 Bit breite Register in einem 16-Bit-Mikrocontroller ein Nachteil sind. In vielen Fällen jedoch, sind in Embedded-Anwendungen 8-Bit-Operationen ausreichend. Beispiele hierfür sind der Zugriff auf die digitalen Ein- und Ausgänge, die in 8 Bit breiten Ports angelegt sind. Auch die Kommunikation mit seriellen Schnittstellen SPI (Serial Peripheral Interface), SCI (Serial Communication Interface) und CAN findet Byte-weise statt. Dedizierte Befehle für die 8-Bit-Akkumulatoren erzeugen schnellen und dichten Code, da im Opcode auf das zweite Datenbyte verzichtet wird und die Befehle in der Regel einen Zyklus weniger benötigen. Nach dem Reset sind beide Akkumulatoren auf Null gestellt. Sie sind jederzeit beschreib- und lesbar.

2.3.2 Die Indexregister X und Y

Die Indexregister X und Y werden, wie ihr Name schon andeutet, zur indizierten

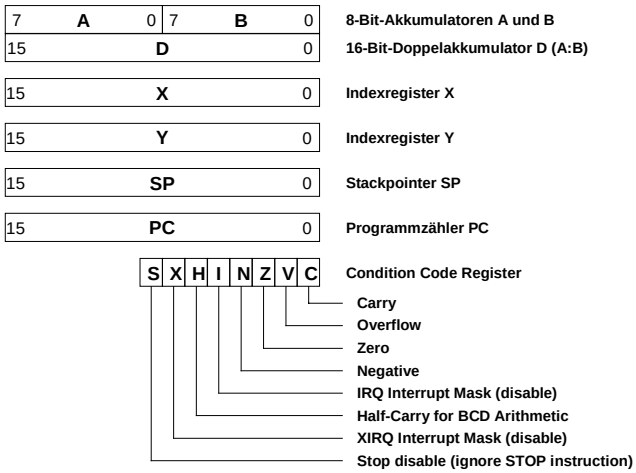

Abbildung 1-2: Das Programmiermodell der 68HC11-, 68HC12- und HCS12-CPU

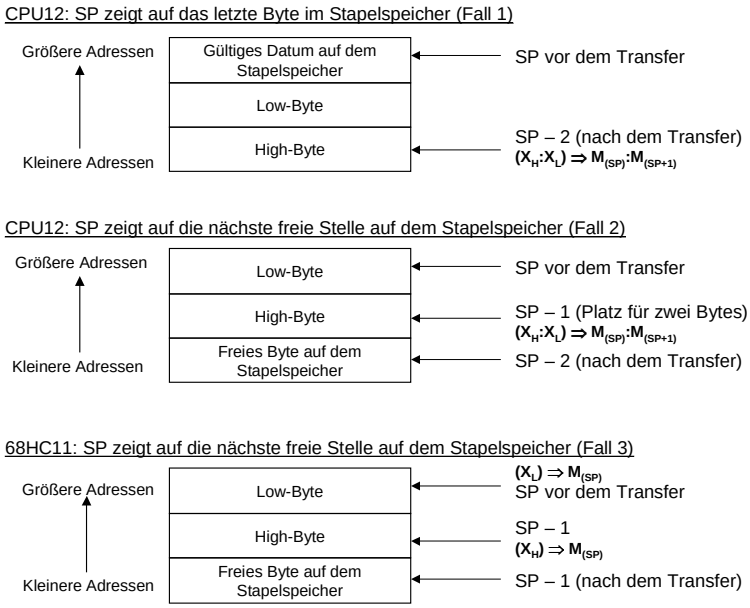

Abbildung 1-3: CPU12 und 68HC11-Stackpointeroperationen

Adressierung verwendet. Bei der indizierten Adressierung wird der Inhalt eines Indexregisters, das die Basisadresse darstellt, zu einer 5-Bit-, 9-Bit- oder 16-Bit-Konstanten oder zum Inhalt eines Akkumulators addiert. Das Resultat wird als effektive Adresse bezeichnet und zeigt auf eine Instruktion oder ein Datum. Die Verwendung von zwei Indexregistern ist besonders bei Speicherkopiervorgängen oder bei Operationen nützlich, bei denen die Operanden aus zwei Tabellen entnommen werden. Zusätzlich werden die Indexregister für einige arithmetische Operationen verwendet, die 16 Bit und 32 Bit breite Größen verwenden. Nach dem Reset befindet sich in den Indexregistern eine Null.

2.3.3 Der Stackpointer SP

Der Stapelspeicher (Stack) wird bei Mikrocontrollern als temporärer Speicher für lokale Variablen verwendet. Beim Aufruf von Unterprogrammen dient er auch als Übergabespeicher der Variablen und die Rücksprungadresse wird ebenfalls dort abgelegt. Bei Interrupts werden automatisch die Register auf den Stack gespeichert. Der Stapelspeicher kann überall im 64 KByte großen Adressraum liegen, benötigt jedoch einen RAM-Speicher. Der Stackpointer wächst bei den Operationen, die ihn verändern nach unten, d. h. er bewegt sich hin zu kleineren Adressen. Immer wenn ein Byte auf dem Stapelspeicher abgelegt wird, dekrementiert sich der Stackpointer automatisch um eine

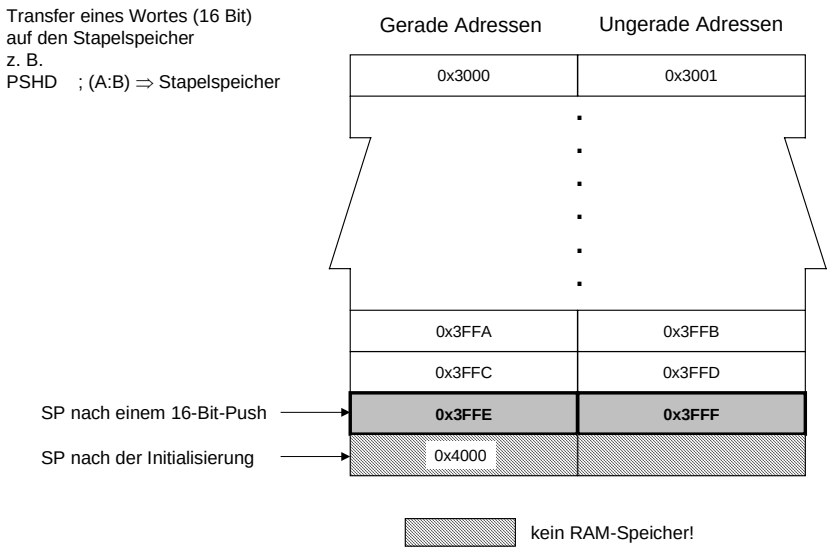

Abbildung 1-4: Transfer eines Wortes (16 Bit) auf den Stapelspeicher bei gerader Stackpointerausrichtung (Even Byte Stack Alignment)

bzw. bei einem Wort um zwei Adressen. Wenn ein Byte vom Stapelspeicher geholt wird, inkrementiert sich der SP automatisch um eine bzw bei einem Wort um zwei Adressen. Der Stackpointer enthält die zuletzt verwendete 16-Bit-Adresse. Üblicherweise ist eine der ersten Aktionen in einem Programm, den Stackpointer mit einem gültigen Wert zu initialisieren. Die Voreinstellung nach dem Reset ist aus Kompatibilitätsgründen zum 68HC11 0x00FF. Diese Defaulteinstellung kann jedoch bei den 68HC12- und HCS12-Derivaten nicht verwendet werden, da in dem Speicherbereich bis 0x00FF Spezialregister der Peripheriemodule liegen und somit kein Platz für den Stapelspeicher ist. Der Stackpointer wird mit dem Befehl „`LDS #$1000`“ oder in C mit „`asm { LDS #$1000 }`“ initialisiert.

Bei Unterprogrammaufrufen wird die Rückkehradresse automatisch auf den Stapelspeicher gelegt. Dabei handelt es sich um die Adresse des Befehls, der dem aufrufenden Sprungbefehl (JSR, CALL) folgt. Am Ende des Unterprogramms bewirkt der RTS-Befehl (Return from Subroutine), dass der Programmzähler mit der Rücksprungadresse, die auf dem Stapelspeicher liegt, geladen wird. Das Programm fährt an der Stelle nach dem Einsprung in das Unterprogramm fort. Bei Interrupts werden zusätzlich zur Rücksprungadresse noch die Register auf den Stapelspeicher gelegt und bei RTI (Return from Interrupt) wieder zurückgespeichert.

✖ *CPU12-Unterschiede zum 68HC11 (1)*

Bei der CPU12 zeigt der Stackpointer auf das letzte Byte, welches auf den Stapelspeicher abgelegt wurde. Beim 68HC11 zeigt er jedoch auf die nächste freie Stelle im Stapelspeicher. Dies wurde geändert, da 16-Bit-Operationen auf dem Stapelspeicher schneller sind, wenn der Stackpointer auf das letzte Byte zeigt. Dies wird in Abb. 1-3 veranschaulicht. Im ersten Fall zeigt der Stackpointer bei der CPU12 auf das letzte Byte im Stapelspeicher. Um ein Wort (16 Bits) auf den Stapelspeicher zu legen, müssen nur zwei Operationen ausgeführt werden: Dekrementieren des Stackpointers um zwei Adressen und anschließendes Ablegen des Wortes auf den Stapelspeicher. Im zweiten Fall wird angenommen, dass die CPU12 auf die nächste freie Stelle im Stapelspeicher zeigt. Damit in einer Operation ein Wort abgelegt werden kann, muss zuerst der Stackpointer um eins dekrementiert werden. Danach kann die Speicheroperation ausgeführt werden. Zuletzt muss der Stackpointer noch einmal um eins dekrementiert werden, damit er wieder auf die nächste freie Stelle zeigt. Im Falle des 68HC11's (Fall 3) zeigt der Stackpointer auf die nächste verfügbare Stelle im Stapelspeicher. Um dort ein Wort abzulegen, müssen zwei Byteoperationen durchgeführt werden, da der Speicherdatenbus des 68HC11 nur 8 Leitungen zur Verfügung stellt. Es wird also ein Byte auf den Stapelspeicher gelegt und dann der Stackpointer um eins dekrementiert. Danach wird das zweite Byte auf den Stack gelegt und der Stackpointer noch einmal dekrementiert. Dazu sind vier Aktionen bzw. Taktzyklen notwendig.

✖ *CPU12-Unterschiede zum 68HC11 (2)*

Bei der Initialisierung des Stackpointers am Anfang eines Programms sollte bei der CPU12 die letzte Adresse im verfügbaren Adressraum plus eins geladen werden, da die CPU12 den Stackpointer um eins dekrementiert, bevor sie Daten auf den Stapelspeicher legt. Beim 68HC11 wird der Stackpointer gewöhnlich mit der letzten verfügbaren RAM-Adresse geladen. Allokiert man beispielsweise (MC9S12DP256) den RAM-Bereich von 0x3000 bis 0x3FFF als Stapelspeicher, würde man den Stackpointer bei der CPU12 mit 0x3FFF + 1 = 0x4000 laden. Berücksichtigt man dies nicht, verschwendet man nur ein Byte RAM, was sicherlich bei den RAM-Größen der HCS12-Derivate nicht tragisch wäre.

Zu beachten ist auch, dass der Stackpointer auf eine gerade Adresse gelegt wird, da 16-Bit-Zugriffe auf den Speicher nur dann in einem Zyklus durchgeführt werden können. Zeigt der Stackpointer auf eine ungerade Adresse, wird der 16-Bit-Zugriff in zwei Zyklen abgearbeitet. Dies ist insbesondere hinderlich, wenn der Stapelspeicher im externen RAM angelegt wird, da dann noch Wartzyklen (Wait States) hinzukommen. Grundsätzlich sollte man, falls die CPU-Leistung in der Applikation eine große Rolle spielt, den Stapelspeicher im internen RAM anordnen, damit bei Unterprogrammaufrufen und Interruptbearbeitungen keine Performanceverluste durch Wartezyklen auftreten. Ein Beispiel ist in Abb. 1-4 gezeigt. Wie oben erwähnt, zeigt der SP auf die Adresse 0x4000, bei der sich kein RAM mehr befindet. Wird beispielsweise der Befehl PSHD ausgeführt, der den Doppelakkumulator D auf den Stapelspeicher rettet, kann diese Operation in einem Zyklus ausgeführt werden, wenn der Stackpointer auf eine gerade Adresse im Speicher zeigt. Steht der Stackpointer auf eine ungerade Adresse, wird ein optionaler Zyklus benötigt.

2.3.4 Der Programmzähler PC

Der Programmzähler ist ein 16-Bit-Register, das die Adresse des als nächsten auszuführenden Befehls enthält. Er wird automatisch jedesmal inkrementiert, wenn eine Instruktion ausgeführt wird. Zusätzlich zum PC gibt es bei der CPU12 noch einen Instruktionszeiger IC (Instruction Pointer), der dazu verwendet wird, 16-Bit-Worte aus dem Befehlspeicher in die Instruktionspipeline zu laden. Auf den IC kann vom Programmierer nicht zugegriffen werden. Er ist für ihn nicht sichtbar.

2.3.5 Statusregister CCR (Condition Code Register)

Das Statusregister beinhaltet fünf Statusbits, zwei weitere Bits, um Interrupts zu maskieren und ein Maskenbit, um die STOP-Instruktion zu verhindern. Die Statusbits zeigen das Ergebnis nach einer CPU-Operation an und werden meist automatisch von der CPU verändert oder ausgewertet.

Beim Schreiben von Assemblerprogrammen muss sehr genau beachtet werden, welche Befehle das Statusregister beeinflusst und auswertet, um eine richtige Arbeitsweise des Programms zu garantieren.

STOP-Masken-Bit S

Mit diesem Flag kann die STOP-Instruktion verhindert werden. Die STOP-Instruktion führt den Mikrocontroller in eine Betriebsart über, bei der minimalster Strombedarf herrscht, da große Teile des Mikrocontrollers inaktiv sind und der Oszillator gestoppt wird. Die STOP-Instruktion wird jedoch nur ausgeführt, wenn sie im CCR-Register freigegeben ist. Ist das S-Bit gesetzt, wird der STOP-Befehl wie ein NOP (No Operation) behandelt, d. h. die CPU führt keine Operation durch und führt den nächsten Befehl aus. Bei gelöschtem S-Bit, wird der STOP-Befehl ausgeführt. Nach dem Reset ist das S-Bit gesetzt, d. h. es werden keine STOP-Instruktionen ausgeführt. Das S-Bit kann jederzeit vom Programm gesetzt oder gelöscht werden.

S-Bit = 1 = STOP-Instruktion wird nicht ausgeführt
S-Bit = 0 = STOP-Instruktion wird ausgeführt

XIRQ-Masken-Bit X

Mit dem X-Masken-Bit kann der externe Interrupteingang XIRQ freigegeben oder gesperrt werden. Der XIRQ-Eingang arbeitet als verbesserte Version eines NMI-Eingangs (Non Maskable Interrupt), der ursprünglich nicht gesperrt werden konnte. Nicht maskierbare Interrupteingänge werden dazu verwendet, auf Systemfehler wie z. B. den Verlust der Versorgungsspannung, oder auf eilige Peripherieanforderungen zu reagieren. Da es jedoch beispielsweise beim Systemstart hinderlich sein kann, wenn dieser Interrupt auftritt (Spurious Interrupt), wurde er bei der CPU12 maskierbar eingerichtet. Tritt beispielsweise ein XIRQ-Interrupt auf, wenn der Stackpointer noch nicht initialisiert wurde, stürzt das System unvermeidlich ab.

Nach dem Reset ist der XIRQ-Interrupt gesperrt (X-Bit ist gesetzt) und kann nach dem erfolgreichen Systemstart durch die Software freigegeben, d. h. gelöscht werden. Wurde

der Interrupt einmal freigegeben, kann er nicht mehr wieder gesperrt werden. Nur ein weiterer Reset setzt das X-Bit und sperrt den Interrupt. Damit wird vermieden, dass durch einen Code-Run-Away das X-Bit versehentlich erneut gesetzt und der Interrupt gesperrt wird.

Tritt bei gelöschtem X-Bit ein XIRQ-Interrupt auf, wird von der CPU u. a. das CCR-Register mit dem gelöschten X-Bit auf dem Stapelspeicher gespeichert. Danach setzt sie das X- und das I-Bit. XIRQ und alle anderen Interrupts sind damit gesperrt, während die XIRQ-Interrupt-Service-Routine ausgeführt wird. Die RTI-Instruktion am Ende der XIRQ-ISR restauriert die CPU-Register vom Stapelspeicher und überschreibt das X-Bit im CCR-Register mit der zuvor abgelegten Null. Jetzt können weitere XIRQ-Interrupts auftreten.

Es gibt jedoch eine Möglichkeit, den XIRQ-Interrupt wieder mit Hilfe der Software zu sperren und das X-Bit zu setzen. In der XIRQ-ISR kann das auf dem Stack abgelegte CCR-Register manipuliert werden, indem das entsprechende Byte auf dem Stapelspeicher verändert wird. Nach dem Ausführen der XIRQ-ISR wird dann vom Stapelspeicher das veränderte Byte mit dem gesetzten X-Bit in das CCR-Register geladen und nun sind weitere XIRQ-Interrupts gesperrt.

Bedeutung des X-Bits:

X-Bit = 1 = XIRQ-Interrupt ist gesperrt
X-Bit = 0 = XIRQ-Interrupt ist freigegeben

Half-Carry-Bit H

Das Half-Carry-Bit zeigt einen Übertrag von Bit 3 auf Bit 4 nach einer Addition an. Die DAA-Instruktion (Dezimale Anpassung von A bei BCD-Operationen) verwendet das H-Bit für die Anpassung des Akkumulators A auf das BCD-Format (Binary Coded Decimal). ABA, ADD und ADC sind die einzigen Instruktionen, die das H-Bit beeinflussen.

Bedeutung des H-Bits:

H-Bit = 1 = Übertrag von Bit 3 auf Bit 4 nach einem ABA-, ADD- und ADC-Befehl
H-Bit = 0 = Kein Übertrag hat von Bit 3 auf Bit 4 stattgefunden

Interrupt-Masken-Bit I

Mit dem I-Bit können alle maskierbaren Interrutps gesperrt werden. Nach dem Reset ist das I-Bit gesetzt und alle Interrupts sind gesperrt. Um die Interrupts wieder freizugeben, muss das I-Bit von der Software gelöscht werden. Treten maskierbare Interrupts auf, während das I-Bit gesetzt und sie damit gesperrt sind, bleiben sie anhängig und kommen zur Ausführung, sobald das I-Bit gelöscht wird. Bei wiederholtem Auftreten des gleichen Interrupts, wenn das I-Bit gesetzt ist, wird nur die zuletzt aufgetretene Interruptanforderung gespeichert und ausgeführt.

Tritt ein Interrupt bei gelöschtem I-Bit auf, wird u. a. das CCR-Register mit dem gelöschten I-Bit auf den Stapelspeicher gerettet und danach das I-Bit gesetzt. Damit sind weitere Interrupts während der Interrupt-Service-Routine (ISR) gesperrt. Erst das RTI-Kommando restauriert das CCR-Register mit dem gelöschten I-Bit vom Stapelspeicher und weitere Interrupts werden ausgeführt.

Das I-Bit kann auch in der ISR durch die Software gelöscht werden. Damit sind weitere Interrupts freigegeben und die ISR kann durch weitere Interruptanforderungen unterbrochen werden. Verschachtelte Interrupts sind jedoch schwierig zu programmieren und zu testen. Deshalb ist bei der Verwendung von verschachtelten Interrupts Vorsicht angesagt. Manchmal ist es sinnvoll, Interrupts während der Programmausführung zu sperren, d. h. während der Ausführung eines begrenzten Programmteils wird verhindert, dass dieses Programmteilstück unterbrochen wird. Dies kann bei zeitlich kritischen Programmen notwendig sein.

Bedeutung des I-Bits:

I-Bit = 1 = Maskierbare Interruptanforderungen sind gesperrt
I-Bit = 0 = Maskierbare Interruptanforderungen sind freigegeben

Negativ-Status-Bit N

Das N-Bit wird von der CPU gesetzt, wenn das höchstwertige Bit (MSB) als Resultat einer Operation gesetzt wird. Das N-Bit wird meist von der Zweier-Kompliment-Arithmetik verwendet, wo das MSB eines Bytes oder Wortes eine negative Zahl anzeigt. Bei einer positiven Zahl ist das MSB Null. Das N-Bit kann auch als allgemeines Statusbit verwendet werden. Beim Laden eines Wertes aus dem Speicher oder eines Ein- und Ausgabeports in den Akkumulator, kann das MSB getestet und ausgewertet werden. Ein zusätzliches Testen des Bits ist nicht mehr notwendig.

Bedeutung des N-Bits:

N-Bit = 1 = MSB ist im Ergebnis gesetzt
N-Bit = 0 = MSB ist im Ergebnis gelöscht

Zero-Bit Z

Das Z-Bit wird gesetzt, wenn das gesamte Resultat Null ist, d. h. alle Bits sind gelöscht. Vergleiche von Zahlen werden in der CPU durch eine Subtraktion realisiert und beeinflussen die CCR-Bits, einschließlich des Z-Bits und reflektieren das Ergebnis der Subtraktion. Die Inkrementier- und Dekrementierbefehle INX, DEX, INY und DEY beeinflussen nur das Z-Bit, jedoch keine weiteren CCR-Flags.

Bedeutung des Z-Bits:

Z-Bit = 1 = Ergebnis ist Null
Z-Bit = 0 = Ergebnis ist ungleich Null

Überlaufbit (Overflow) V

Das Überlaufbit V wird gesetzt, wenn ein Überlauf eines Zweierkompliments als Ergebnis einer Operation auftritt. Beispielsweise wird das V-Bit gesetzt, wenn zwei Zahlen, die kleiner oder gleich 127 sind, zusammenaddiert werden und dabei ein negatives Ergebnis herauskommt (d. h. das MSB ist gesetzt). Das gesetzte V-Bit zeigt an, dass das Ergebnis nicht richtig im Zweierkompliment dargestellt werden kann.

V-Bit = 1 = Überlauf ist aufgetreten
V-Bit = 0 = Kein Überlauf ist aufgetreten

Übertragsbit (Carry) C

Das Übertragsbit C wird gesetzt, wenn während einer Addition ein Übertrag oder während einer Subtraktion ein Unterlauf aufgetreten ist. Das C-Bit arbeitet auch als Fehlerflag bei Multiplikations- und Divisionsbefehlen. Schiebe- und Rotierbefehle verwenden das Carrybit als Zwischenspeicher bei mehreren Argumenten.

C-Bit = 1 = Überlauf oder Unterlauf ist aufgetreten
C-Bit = 0 = Kein Überlauf oder Unterlauf ist aufgetreten

2.4 Der Befehlssatz der CPU12

Dieses Kapitel beschreibt die verfügbaren Befehle und die Adressierungsmodi der CPU12 in tabellarischer Form, da der Befehlssatz sehr umfangreich ist und die CPU12 viele Adressierungsmodi unterstützt. Für weitergehende Informationen sei auf [1] und [2] verwiesen. Der Autor glaubt auch, dass die meisten Anwender die Mikrocontroller in der Hochsprache C programmieren werden, auch wenn die CPU12 über einen leistungsfähigen Befehlssatz verfügt, der leicht und schnell erlernbar ist.

Bei der CPU12 handelt es sich um eine Von-Neumann-Architektur, d. h. es wird nicht zwischen Daten- und Programmspeicher unterschieden. Alle Peripheriefunktionen und alle Kontrollregister der Ein- und Ausgänge werden wie Speicher behandelt. Es gibt also keine dedizierten I/O-Befehle, wie das bei anderen Architekturen der Fall ist. Das vereinfacht die Programmierung erheblich. Zusätzlich zu den üblichen Befehlen eines Mikrocontrollers verfügt die CPU12 noch über spezielle mathematische Befehle, die die Fuzzylogik unterstützten.

2.4.1 Adressierungsmodi

Die CPU12 bietet unterschiedliche Möglichkeiten an, die Operanden für Operationen zu bestimmen, d. h. zu adressieren. Dabei wird während des Holens des Befehls von der CPU die tatsächliche (effektive) Adresse des Operanden bestimmt. Die effektive Adresse ist dabei die Adresse, die auf den tatsächlichen Befehl oder auf das Datum im Speicher zeigt. Sie kann sich aus mehreren Teilen zusammensetzen. Das Berechnen der effektiven Adresse benötigt keine weiteren CPU-Zyklen. Durch das geschickte Verwenden der Adressierungsarten können Programmieraufgaben effizient und elegant gelöst werden. Die Funktionsweise der Adressierungsarten werden auch in [3] ausführlich erläutert. Die CPU12 verfügt über die in Tabelle 1-1 aufgeführten Adressierungsmodi.

In einem Befehl können auch mehrere Adressierungsarten vorkommen. Ein Beispiel ist

```
BRSET    FLAG,#$03,THERE.
```

In diesem Beispiel wird die erweiterte absolute Adressierung (Extended) verwendet, um auf den Operanden FLAG zuzugreifen. Über unmittelbare Adressierung (Immediate) greift die CPU auf die Maske $03 zu. Die Zieladresse des Verzweigungsbefehls, falls die Bedingung im Befehl wahr ist, wird mit Hilfe der PC-relativen Adressierung bestimmt.

In der folgenden Tabelle werden folgende Abkürzungen verwendet:

INST	Instruktion, Befehl
opr8i	Unmittelbarer 8-Bit-Wert
opr16i	Unmittelbarer 16-Bit-Wert
opr8a	8-Bit-Wert bei absoluter Adressierung
opr16a	16-Bit-Wert bei absoluter Adressierung
oprx3	Jede positive Ganzzahl (Integer) von 1 bis 8 für Pre- und Post-Inkrement und -Dekrement
oprx5	Jede positive Ganzzahl (Integer) von -16 bis +15
oprx9	Jede positive Ganzzahl (Integer) von -256 bis +255
oprx16	Jede positive Ganzzahl (Integer) von –32 768 bis 65 535
rel8	Sprungmarke (Label) eines Sprungziels innerhalb eines Bereichs von –256 bis +255
rel16	Sprungmarke (Label) eines Sprungziels innerhalb des 64 KByte Adressraumes
xysp	Bezeichner für die Register X, Y oder SP
abd	Bezeichner für die Akkumulatoren A, B oder D
d	Bezeichner für den Akkumulator D

Adressierungsart	**Urform**	**Abk.**	**Beschreibung**	**Beispiele**
Registeradressierung (Inherent, Register direct)	INST	INH	Die Operanden befinden sich in den CPU-Registern. Die Registeradresse befindet sich im OpCode des Befehls.	`NOP ; kein Operand` `INX ; Operand ist ein ; CPU-Register`

Unmittelbare Adressierung (Immediate)	INST #opr8i
	INST #opr16i
		IMM	Der Operand ist eine Konstante und ist im OpCode enthalten. Der Operand kann Byte- oder Wort-Größe haben.	`LDAA #$55`
				`LDX #$1234`
				`LDY #$67`

Absolute Adressierung (Direct, Absolute)	INST opr8a	DIR	Die effektive 8-Bit-Adresse des Operanden befindet sich als absolute Adresse im OpCode. Der Adressierungsbereich geht von 0x0000 bis 0x00FF.	`LDAA $55 ; $55 ist die ; Adresse` `LDX $20`
Erweiterte absolute Adressierung (Extended)	INST opr16a	EXT	Der Operand ist eine 16-Bit-Adresse.	`LDAA $F03B ; $F03B ist ; die ; Adresse`
PC-relative Adressierung (Relative)	INST rel8 INST rel16	REL	Die effektive Adresse wird aus dem Wert des PC's bestimmt, zu dem entweder ein 8-Bit-Wert oder ein 16-Bit-Wert (= Offset) addiert wird. Diese Adressierungsart wird nur in Verzweigungsbefehlen (Branch) verwendet.	`BRSET` `BRCLR`

Indizierte Adressierung mit Offset (Indexed)	INST oprx5,xysp
	INST oprx9,xysp
	INST oprx16,xysp
		IDX
		IDX1
		IDX2
			Die effektive Adresse wird aus der variablen Basisadresse in den Registern X, Y, SP oder PC und dem konstanten Offset gebildet, der 5 Bit (IDX), 9 Bit (IDX1) oder 16 Bit (IDX2) groß sein kann.	`LDAA 0,X`
				`STAB -8,Y`
				`LDAA $FF,X`
				`LDAB -20,Y`
				`LDAA $2000,X`

Adressierungsart	Urform	Abk.	Beschreibung	Beispiele
Indizierte Adressierung mit Prädekrement (Indexed predecrement)	INST oprx3,-xysp	IDX	Die effektive Adresse errechnet sich aus dem Wert in X, Y oder SP, der vorher um einen Wert von 1 bis 8 dekrementiert wurde.	`STAA 1,-SP; = PSHA` `STX 2,-SP ; = PSHX`
Indizierte Adressierung mit Präinkrement (Indexed preincrement)	INST oprx3,+xys	IDX	Die effektive Adresse errechnet sich aus dem Wert in X, Y oder SP, der vorher um einen Wert von 1 bis 8 inkrementiert wurde.	
Indizierte Adressierung mit Postdekrement (Indexed postdecrement)	INST oprx3,xys-	IDX	Die effektive Adresse errechnet sich aus dem Wert in X, Y oder SP, der nachher um einen Wert von 1 bis 8 dekrementiert wurde.	

Indizierte Adressierung mit Postinkrement (Indexed postincrement)	INST oprx3,xysp+	IDX	Die effektive Adresse errechnet sich aus dem Wert in X, Y oder SP, der nachher um einen Wert von 1 bis 8 inkrementiert wurde.	`CPX 1,X+`
				`LDX 2,SP+ ; = PULX`
				`LDAA 1,SP+ ; = PULA`

Indizierte Adressierung mit Akkumulatoroffset (Indexed, accumulator offset)	INST abd,xysp	IDX	Die effektive Adresse errechnet sich aus dem Wert in dem Register X, Y, SP oder PC plus dem Wert im Akkumulator A, B oder D.	`LDAA B,X`

Adressierungsart	**Urform**	**Abk.**	**Beschreibung**	**Beispiele**
Registerindirekte Adressierung mit Offset (Indexed-indirect)	INST [oprx16,xysp]	[IDX2]	Der Wert in X, Y, SP oder PC plus eine 16-Bit-Konstante zeigen auf die effektive Adresse im Speicher.	`LDAA [10,X]`

Registerindirekte Adressierung mit D-Offset (Indexed-indirect)	INST [d,xysp]	[D,IDX]	Der Wert in X, Y, SP oder PC plus der Wert in D zeigt auf die effektive Adresse im Speicher.	`JMP [D,PC]`
				`GO1 DC.W PLACE1`
				`GO2 DC.W PLACE2`
				`GO3 DC.W PLACE3`

Tabelle 1-1: Adressierungsarten der CPU12

2.4.2 Befehlsübersicht

Die Übersicht der Befehle der CPU12 wird tabellarisch wiedergegeben. Sie sind nach Funktionen sortiert und es wird jeweils nur die Grundform des Befehls beschrieben, ohne die anwendbaren Adressierungsmodi für den jeweiligen Befehl zu erwähnen. Eine ausführliche Beschreibung der Befehle ist in [1, 2] zu finden.

Lade- und Speicher-Befehle

Die LD- und ST-Befehle speichern Werte vom Speicher (M) in das jeweilige Register A, B, D, X, Y und SP oder holen Werte vom Register in den Speicher. Das ist notwendig, da die CPU viele Operationen nur mit Hilfe der Register ausführen kann.

Tabelle 1-2: Lade-Instruktionen (Load from Memory M)		
Mnemonic	**Funktion**	**Operation**
LDAA	Lade den Akkumulator A	(M) → A
LDAB	Lade den Akkumulator B	(M) → B
LDD	Lade den Akkumulator D (A:B)	(M:M+1) → D
LDS	Lade den Stackpointer SP	(M:M+1) → SP
LDX	Lade das Indexregister X	(M:M+1 → X
LDY	Lade das Indexregister Y	(M:M+1 → Y
Speicher-Instruktionen (Store into Memory M)		
Mnemonic	**Funktion**	**Operation**
STAA	Speichern des Akkumulators A	(M) → A
STAB	Speichern des Akkumulators B	(M) → B
STD	Speichern des Akkumulators D (A:B)	(M:M+1) → D
STS	Speichern des Stackpointers SP	(M:M+1) → SP
STX	Speichern des Indexregisters X	(M:M+1 → X
STY	Speicheren des Indexregisters Y	(M:M+1 → Y

Kopieren von Speicher zu Speicher

Die MOV-Befehle kopieren einen Wert von einer Speicherzelle direkt in eine andere Speicherzelle, ohne dass dazu CPU-Register benötigt werden.

Tabelle 1-3: MOV-Instruktionen (Move Memory to Memory)		
Mnemonic	**Funktion**	**Operation**
`MOVB`	Kopiere ein Byte von Quelle nach Ziel	Quelle → Ziel
`MOVW`	Kopiere Wort von Quelle nach Ziel	Quelle → Ziel

Register-Transfer und Register-Austauschbefehle

Diese Befehle dienen dazu, die Daten von einem Register in ein anderes Register zu schreiben oder die Inhalte von Registern auszutauschen. Die CPU12 verwendet, im Gegensatz zum 68HC11 jeweils nur eine Mnemonic für den Transfer und den Austausch.

Talelle 1-4: Register-Transfer- und -Austauschbefehle		
Mnemonic	**Funktion**	**Operation**
`TFR`	Transferiert den Inhalt eines Registers in ein anderes Register	A,B,CCR,D,X,Y,SP → A,B,CCR,D,X,Y,SP
`EXG`	Tauscht den Inhalt eines Registers mit dem eines anderen Registers aus	A,B,CCR,D,X,Y,SP → A,B,CCR,D,X,Y,SP
`TAB`	Transferiert den Inhalt von Akkumulator A nach B	A → B
`TBA`	Transferiert den Inhalt von Akkumulator B nach A	B → A

Tabelle 1-4: Register-Transfer- und -Austauschbefehle

Sprungbefehle

Mit Hilfe von Sprungbefehlen, kann der Programmfluss verändert werden. Bei der CPU12 gibt es drei verschiedene Arten von Sprungbefehlen: Unbedingte, einfache und bedingte Sprungbefehle. Die bedingten Sprungbefehle, die nur Sprünge durchfühen, wenn eine bestimmte Bedingung erfüllt ist, können in durch ein Vorzeichen bedingte und durch den Zustand eines Bits bedingte Sprungbefehle eingeteilt werden.

Tabelle 1-5: Sprungbefehle		
Unbedingte Sprungbefehle		
Mnemonic	**Funktion**	**CCR-Test**
BRA, LBRA	Springe immer zur Adresse	Keiner
BRN	Springe nie (= NOP mit einer Länge von drei Bytes und einer Zyklenzahl von drei)	
LBRN	Springe nie (= NOP mit einer Länge von vier Bytes und einer Zyklenzahl von drei)	
Einfache Sprungbefehle, die die CCR-Flags auswerten		
Mnemonic	**Funktion**	**CCR-Test**
BCC, LBCC	Springe, wenn das Carry-Bit gelöscht ist	C == 0
BCS, LBCS	Springe, wenn das Carry-Bit gesetzt ist	C == 1
BEQ, LBEQ	Springe, wenn der Wert identisch ist	Z == 1
BNE, LBNE	Springe, wenn der Wert ungleich ist	Z == 0
BMI, LBMI	Springe, wenn negativ	N == 1
BPL, LBPL	Springe, wenn positiv	N == 0
BVC, LBVC	Springe, wenn das Überlaufbit gelöscht ist	V == 0
BVS, LBVS	Springe, wenn das Überlaufbit gesetzt ist	V == 1
Sprungbefehle, die auf Werte ohne Vorzeichen angewendet werden		
Mnemonic	**Funktion**	**CCR-Test**
BHS, LBHS	Springe, wenn gleich oder größer (Register = Speicher oder Register > Speicher)	C == 0
BLO, LBLO	Springe, wenn kleiner (Register < Speicher)	C == 1
BHI, LBHI	Springe, wenn größer (Register > Speicher)	(C | Z) == 0
BLS, LBLS	Springe, wenn gleich oder kleiner (Register = Speicher oder Register < Speicher	(C | Z) == 1
Sprungbefehle, die auf Werte mit Vorzeichen angewendet werden		
Mnemonic	**Funktion**	**CCR-Test**
BGE, LBGE	Springe, wenn gleich oder mehr als (Register = Speicher oder Register > Speicher)	(N^V) == 0
BLT, LBLT	Springe, wenn weniger als (Register < Speicher)	(N^V) == 1
BGT, LBGT	Springe, wenn mehr (Register > Speicher)	Z | (N^V) == 0
BLE, LBLE	Springe, wenn gleich oder weniger als (Register = Speicher oder Register < Speicher)	Z | (N^V) == 1

Befehle für Unterprogramme

Tabelle 1-6: Unterprogramm-Befehle		
Mnemonic	**Funktion**	**Operation**
`BSR`	Ruft ein Unterprogramm UP auf. Die Rückkehradresse (RTN_H, RTN_L) wird auf den Stapelspeicher gelegt.	SP – 2 → SP RTN_H → (SP) RTN_L → (SP+1) UP-Adresse → PC
`JSR`	Ruft ein Unterprogramm UP auf. Die Rückkehradresse (RTN_H, RTN_L) wird auf den Stapelspeicher gelegt.	SP – 2 → SP RTN_H → (SP) RTN_L → (SP+1) UP-Adresse → PC
`JMP`	Springt zur einer Adresse	Adresse → PC
`CALL`	Ruft ein Unterprogramm UP im erweiterten Speicher (Expanded Memory) auf. Die Rückkehradresse (RTN_H, RTN_L) wird auf den Stapelspeicher gelegt.	SP – 2 → SP RTN_H → (SP) RTN_L → (SP+1) SP – 1 → SP PPAGE → (SP) Page# → PPAGE UP-Adresse → PC
`RTS`	Rückkehr von einem Unterprogramm	SP → PC_H SP+1 → PC_L SP+2 → SP
`RTC`	Rückkehr von einem Unterprogramm, das sich im erweiterten Speicher (Expanded Memory) befindet	(SP) → PPAGE SP+1 → SP (SP) → PC_H (SP+1) PC_L SP+2 → SP

Schleifenbefehle

Tabelle 1-7: CCR-Befehle		
Mnemonic	**Funktion**	**Operation**
DBEQ	Zähler = Zähler – 1 Springe, wenn Zähler == 0	Zähler – Zähler → Zähler Wenn Zähler == 0 springe ansonsten führe nächste Instruktion aus
DBNE	Zähler = Zähler – 1 Springe, wenn Zähler != 0	Zähler – Zähler → Zähler Wenn Zähler != 0 springe ansonsten führe nächste Instruktion aus
IBEQ	Zähler = Zähler + 1 Springe, wenn Zähler == 0	Zähler + Zähler → Zähler Wenn Zähler == 0 springe ansonsten führe nächste Instruktion aus
IBNE	Zähler = Zähler + 1 Springe, wenn Zähler != 0	Zähler + Zähler → Zähler Wenn Zähler != 0 springe ansonsten führe nächste Instruktion aus
TBEQ	Springe, wenn Zähler == 0	Wenn Zähler == 0 springe ansonsten führe nächste Instruktion aus
TBNE	Springe, wenn Zähler != 0	Wenn Zähler != 0 springe ansonsten führe nächste Instruktion aus

Spezielle Tabellenbefehle

Tabelle 1-8: Tabellenbefehle		
Mnemonic	**Funktion**	**Operation**
TBL	8-Bit-Wert aus einer Tabelle holen und interpolieren	(M) + (B x [(M+1) – MM)] → A
ETBL	16-Bit-Wert aus einer Tabelle holen und interpolieren	(M:M+1) + (B x [(M+2:M+3) – (M:M+1)] → D

Mathematische Befehle

Tabelle 1-9: Mathematische Befehle: Addition und Subtraktion		
Addition		
Mnemonic	**Funktion**	**Operation**
ABA	Addiere B zu A	B + A → A
ADDA	Addiere den Inhalt einer Speicherzelle zu A ohne Carrybit	A + (M) → A
ADCA	Addiere den Inhalt einer Speicherzelle zu A mit Carrybit	A + (M) + CCR:C → A
ADDB	Addiere den Inhalt einer Speicherzelle zu B ohne Carrybit	B + (M) = B
ADCB	Addiere den Inhalt einer Speicherzelle zu B mit Carrybit	B + (M) + CCR:C → B
ADDD	Addiere den Inhalt einer Speicherzelle zu D ohne Carrybit	D + (M:M+1) → D
DAA	Dezimalanpassung von A	A + Korrekturfaktor → A
Subtraktion		
SBA	Subtrahiere B von A	A – B → A
SUBA	Subtrahiere Inhalt von einer Speicherzelle von A ohne Carrybit	A – (M) → A
SBCA	Subtrahiere Inhalt von einer Speicherzelle von A mit Carrybit	A – (M) – CCR:C → A
SUBB	Subtrahiere Inhalt von einer Speicherzelle von B ohne Carrybit	B – (M) → B
SBCB	Subtrahiere Inhalt von einer Speicherzelle von B mit Carrybit	B – (M) – CCR:C → B
SUBD	Subtrahiere Inhalt von einer Speicherzelle von D ohne Carrybit	D – (M:M+1) → D
Befehl zur Vorzeichenerweiterung		
SEX	Erweitern des Vorzeichens eine 8-Bit-Registers auf ein 16-Bit-Register	A,B,CCR → D,X,Y,SP

Tabelle 1-10: Mathematische Befehle: Multiplikation und Division		
Multiplikation		
Mnemonic	**Funktion**	**Operation**
MUL	8-Bit x 8-Bit Multiplikation ohne Vorzeichen	A x B → d
EMUL	16-Bit x 16-Bit Multiplikation ohne Vorzeichen	D x Y → Y:D
EMULS	16-Bit x 16-Bit Multiplikation mit Vorzeichen	D x Y → Y:D
EMACS	16-Bit x 16-Bit Multiplikation mit anschließender 32-Bit-Addition	M_X:M_{X+1}) x (M_Y:M_{Y+1}) + M:M+3 → M:M+3
Division		
Mnemonic	**Funktion**	**Operation**
IDIV	16-Bit ÷ 16-Bit Division ohne Vorzeichen	D ÷ X → X; Rest → D
FDIV	16-Bit ÷ 16-Bit Fractional-Division	D ÷ X → X; Rest → D
IDIVS	16-Bit ÷ 16-Bit Division mit Vorzeichen	D ÷ X → X; Rest → D
EDIV	32-Bit ÷ 16-Bit Division ohne Vorzeichen	Y:D ÷ X → Y; Rest → D
EDIVS	32-Bit ÷ 16-Bit Division mit Vorzeichen	Y:D ÷ X → Y; Rest → D

Tabelle 1-11: Arithmetische Befehle für Adressberechnungen		
Mnemonic	**Funktion**	**Operation**
INX	Inkrementiere das Indexregister X	X + 1 → X
INY	Inkrementiere das Indexregister Y	Y + 1 → Y
INS	Inkrementiere den Stackpointer SP	SP + 1 → SP
DEX	Dekrementiere das Indexregister X	X – 1 → X
DEY	Dekrementiere das Indexregister Y	Y – 1 → Y
DES	Dekrementiere den Stackpointer SP	SP – 1 → SP
ABX	Addiere den Akkumulator B zum Indexregister X	X + B → X
ABY	Addiere den Akkumulator B zum Indexregister Y	Y + B → Y
LEAX	Lade eine effektive Adresse (EA) in das Indexregister X	EA → X
LEAY	Lade eine effektive Adresse (EA) in das Indexregister Y	EA → Y
LEAS	Lade eine effektive Adresse (EA) in den Stackpointer	EA → SP

Minimum- und Maximum-Befehle

Tabelle 1-12: Minimum- und Maximumbefehle		
Mnemonic	**Funktion**	**Operation**
MINA	Minimum von zwei nicht vorzeichenbehafteten 8-Bit-Werten finden. Das Resultat befindet sich im Akkumulator A	Min(A, (M)) → A
MINM	Minimum von zwei nicht vorzeichenbehafteten 8-Bit-Werten finden. Das Resultat befindet sich in der Speicherzelle M	Min(A, (M)) → M
EMIND	Minimum von zwei nicht vorzeichenbehafteten 16-Bit-Werten finden. Das Resultat befindet sich im Akkumulator D	Min(D, (M:M+1)) → D
EMINM	Minimum von zwei nicht vorzeichenbehafteten 16-Bit-Werten finden. Das Resultat befindet sich in den Speicherzellen M:M+1	Min(D, (M:M+1)) → M
MAXA	Maximum von zwei nicht vorzeichenbehafteten 8-Bit-Werten finden. Das Resultat befindet sich im Akkumulator A	Max(A, (M)) → A
MAXM	Maximum von zwei nicht vorzeichenbehafteten 8-Bit-Werten finden. Das Resultat befindet sich in der Speicherzelle M	Max (A, (M)) → M
EMAXD	Maximum von zwei nicht vorzeichenbehafteten 16-Bit-Werten finden. Das Resultat befindet sich im Akkumulator D	Max (D, (M:M+1)) → D
EMAXM	Maximum von zwei nicht vorzeichenbehafteten 16-Bit-Werten finden. Das Resultat befindet sich in den Speicherzellen M:M+1	Max (D, (M:M+1)) → M

Fuzzy-Logikbefehle

Die CPU12 stellt die vier Fuzzy-Logikbefehle MEM, REV, REVW und WAV zur Verfügung, die hier nicht weiter beschrieben werden.

Lesen, Modifizieren und Zurückschreiben

Die folgenden Befehle lesen einen Wert aus einer Speicherzelle oder aus einem Register, verändern ihn und schreiben ihn an dieselbe Stelle zurück.

Tabelle 1-13: Befehle, die Lesen-Modifizieren-Schreiben

Mnemonic	Funktion	Operation
INCA	Inkrementiere den Akkumulator A	A + 1 → A
INCB	Inkrementiere den Akkumulator B	B + 1 → B
INC	Inkrementiere den Inhalt der Speicherzelle M	(M) + 1 → M
DECA	Dekrementiere den Akkumulator A	A – 1 → A
DECB	Dekrementiere den Akkumulator B	B – 1 → B
DEC	Dekrementiere den Inhalt der Speicherzelle M	(M) – 1 → M
NEGA	Bilde das Zweierkomplement von A	0 – A → A
NEGB	Bilde das Zweierkomplement von B	0 – B → B
NEG	Bilde das Zweierkomplement des Inhalts der Speicherzelle M	0 - (M) → (M)
COMA	Bilde das Einerkomplement von A	$FF – A → A
COMB	Bilde das Einerkomplement von B	$FF – B → B
COM	Bilde das Einerkomplement von dem Inhalt der Speicherzelle M	$FF – (M) → (M)
CLRA	Lösche den Akkumulator A	0 →A
CLRB	Lösche den Akkumulator B	0 → B
CLR	Lösche den Inhalt der Speicherzelle M	0 → (M)
TSTA	Teste den Akkumulator A auf eine negative Zahl oder auf Null	A – 0
TSTB	Teste den Akkumulator B auf eine negative Zahl oder auf Null	B – 0
TST	Teste den Inhalt der Speicherzelle M auf eine negative Zahl oder auf Null	(M) – 0

Schiebe- und Rotationsbefehle

Es gibt logische und arithmetische Schiebebefehle. Die logischen Schiebebefehle schieben jedes Bit um eine Stelle. Das MSB wird in das Carrybit geschoben und in das LSB wird eine Null geschrieben. Die logischen und arithmetischen Linksschiebebefehle sind identisch, d. h. sie führen die gleiche Operation aus. Lediglich der logische und arithmetische Rechtsschiebebefehl unterscheidet sich . Der logische Rechtsschiebebefehl schiebt jedes Bit um eine Stelle nach rechts. Das LSB wird in das Carrybit geschoben und in das MSB wird eine Null geschrieben. Beim arithmetischen Rechtsschiebebefehl bleibt das MSB jedoch erhalten.

Tabelle 1-14: Schiebebefehle (Shift)

Mnemonic	**Funktion**	**Operation**
`LSLA, ASLA`	Schiebe den Inhalt des Akkumulators A nach links	MSB → Carrybit 0 → LSB
`LSLB, ASLB`	Schiebe den Inhalt des Akkumulators B nach links	MSB → Carrybit 0 → LSB
`LSL, ASL`	Schiebe den Inhalt der Speicherzelle M nach links	MSB → Carrybit 0 → LSB
`LSLD, ASLD`	Schiebe den Inhalt des Akkumulators D nach links	MSB → Carrybit 0 → LSB
`LSRA`	Schiebe den Inhalt des Akkumulators A nach rechts	0 → MSB LSB → Carrybit
`LSRB`	Schiebe den Inhalt des Akkumulators B nach rechts	0 → MSB LSB → Carrybit
`LSR`	Schiebe den Inhalt der Speicherzelle M nach rechts	0 → MSB LSB → Carrybit
`LSRD`	Schiebe den Inhalt des Akkumulators D nach rechts	0 → MSB LSB → Carrybit
`ASRA`	Schiebe den Inhalt des Akkumulators A nach rechts	MSR → MSR LSB → Carrybit
`ASRB`	Schiebe den Inhalt des Akkumulators B nach rechts	MSR → MSR LSB → Carrybit
`ASR`	Schiebe den Inhalt der Speicherzelle M nach rechts	MSR → MSR LSB → Carrybit

Bei den Rotationsbefehlen wird bei der Linksrotation das MSB in das Carrybit geschoben und das Carrybit in das LSB kopiert. Bei der Rotation nach rechts wird das LSB in das Carrybit geschoben und das Carrybit in das MSB kopiert.

Tabelle 1-15: Rotationsbefehle (Rotate)

Mnemonic	Funktion	Operation
ROLA	Rotiere den Akkumulator A durch das Carrybit nach links	MSB → Carrybit Carrybit → LSB
ROLB	Rotiere den Akkumulator B durch das Carrybit nach links	MSB → Carrybit Carrybit → LSB
ROL	Rotiere den Inhalt der Speicherzelle M durch das Carrybit nach links	MSB → Carrybit Carrybit → LSB
RORA	Rotiere den Akkumulator A durch das Carrybit nach rechts	LSB → Carrybit Carrybit → MSB
RORB	Rotiere den Akkumulator B durch das Carrybit nach links	LSB → Carrybit Carrybit → MSB
ROR	Rotiere den Inhalt der Speicherzelle M durch das Carrybit nach rechts	LSB → Carrybit Carrybit → MSB

Logische Befehle und Vergleichsbefehle

Die logischen Befehle führen die Operationen UND, ODER oder EXKLUSIV-ODER bitweise durch. Das Resultat der Operation befindet sich im Akkumulator A oder B. Bei den BIT-Befehlen wird lediglich das CCR-Register modifiziert. Das Ergebnis der Operation wird verworfen.

Tabelle 1-16: Logische Befehle

Mnemonic	Funktion	Operation
ANDA	Logische UND-Verknüpfung des Akkumulators A mit dem Inhalt einer Speicherzelle	A & (M) → A
ANDB	Logische UND-Verknüpfung des Akkumulators B mit dem Inhalt einer Speicherzelle	B & (M) → B
ORAA	Logische ODER-Verknüpfung des Akkumulators A mit dem Inhalt einer Speicherzelle	A | (M) → A
ORAB	Logische ODER-Verknüpfung des Akkumulators B mit dem Inhalt einer Speicherzelle	B | (M) → B
EORA	Logische EXKLUSIV-ODER-Verknüpfung des Akkumulators A mit dem Inhalt einer Speicherzelle	A ^ (M) → A
EORB	Logische EXKLUSIV-ODER-Verknüpfung des Akkumulators B mit dem Inhalt einer Speicherzelle	B ^ (M) → B
BITA	Logische UND-Verknüpfung des Akkumulators A mit dem Inhalt einer Speicherzelle. Das Ergebnis wird verworfen	A & (M) CCR wird beeinflusst
BITB	Logische UND-Verknüpfung des Akkumulators B mit dem Inhalt einer Speicherzelle. Das Ergebnis wird verworfen	B & (M) CCR wird beeinflusst
ANDCC	Logische UND-Verknüpfung von CCR und einer Maske	CCR & Maske → CCR
ORCC	Logische UND-Verknüpfung von CCR und einer Maske	CCR & Maske → CCR

Die Vergleichsoperationen führen eine Operation durch, die einen Wert aus dem Speicher von einem Register subtrahiert. Dabei werden der Inhalt der Speicherzelle und des beteiligten Registers nicht verändert. Lediglich das CCR-Register wird entsprechend modifiziert.

Tabelle 1-17: Vergleichsbefehle

Mnemonic	Funktion	Operation
CMPA	Vergleiche den Akkumulator A mit dem Inhalt der Speicherzelle M	A – (M)
CMPB	Vergleiche den Akkumulator B mit dem Inhalt der Speicherzelle M	B – (M)
CBA	Vergleiche den Akkumulator A mit dem Akkumulator B	A – B
CPD	Vergleiche den Akkumulator D mit dem Inhalt der Speicherzelle M	D – (M:M+1)
CPX	Vergleiche das Indexregister X mit dem Inhalt der Speicherzelle M	X – (M:M+1)
CPY	Vergleiche das Indexregister Y mit dem Inhalt der Speicherzelle M	Y – (M:M+1)
CPS	Vergleiche den Stackpointer SP mit dem Inhalt der Speicherzelle M	SP – (M:M+1)

Befehle für den Stapelspeicher

Tabelle 1-18: Stackbezogene Befehle		
Mnemonic	**Funktion**	**Operation**
PSHA	Speichere den Akkumulator A auf den Stapelspeicher	SP – 1 → SP A → (SP)
PSHB	Speichere den Akkumulator B auf den Stapelspeicher	SP – 1 → SP B → (SP)
PSHC	Speichere das Condition-Code-Register CCR auf den Stapelspeicher	SP – 1 → SP CCR → (SP)
PSHD	Speichere den Akkumulator D auf den Stapelspeicher	SP – 2 → SP D → (SP:SP+1)
PSHX	Speichere das Indexregister X auf den Stapelspeicher	SP – 2 → SP X → (SP:SP+1)
PSHY	Speichere das Indexregister Y auf den Stapelspeicher	SP – 2 → SP Y → (SP:SP+1)
PULA	Hole den Akkumulator A vom Stapelspeicher zurück	(SP) → A SP + 1 → SP
PULB	Hole den Akkumulator B vom Stapelspeicher zurück	(SP) → B SP + 1 → SP
PULC	Hole das Condition-Code-Register CCR vom Stapelspeicher zurück	(SP) → CCR SP + 1 → SP
PULD	Hole den Akkumulator D vom Stapelspeicher zurück	(SP:SP+1) → D SP + 2 → SP
PULX	Hole das Indexregister X vom Stapelspeicher zurück	(SP:SP+1) → X SP + 2 → SP
PULY	Hole das Indexregister Y vom Stapelspeicher zurück	(SP:SP+1) → X SP + 2 → SP

Bit-Manipulationsbefehle

Tabelle 1-19: Bit-Manipulationsbefehle		
Mnemonic	**Funktion**	**Operation**
BSET	Setzt in einem Byte im Speicher Bits, abhängig von einer Maske	(M) | Maske → M
BCLR	Löscht in einem Byte im Speicher Bits, abhängig von einer Maske	(M) & ~Maske → M
BRSET	Verzweigt, wenn Bits in einer Speicherzelle gesetzt sind. Überprüfung erfolgt mit einer Maske	Verzweigt, wenn ~(M) & Maske == 0
BRCLR	Verzweigt, wenn Bits in einer Speicherzelle gelöscht sind. Überprüfung erfolgt mit einer Maske	Verzweigt, wenn (M) & Maske == 0

Verschiedene Befehle

Tabelle 1-20: Verschiedene Befehle		
Mnemonic	**Funktion**	**Operation**
NOP	Keine Operation wird durchgeführt	PC+1 → PC
BGND	Eintreten in den aktiven Background-Modus	-
STOP	Stoppt den Takt auf dem Mikrocontroller	(SP-2) → SP; RTN_H:RTN_L → (SP):SP+1 SP-2 → SP; Y → (SP):(SP+1) SP-2 → SP; X → (SP):(SP+1) SP-2 → SP; B:A → (SP):(SP+1) SP-1 → SP; CCR → SP

Befehle für die Interruptverarbeitung

Tabelle 1-21: Befehle für die Interruptverarbeitung		
Mnemonic	**Funktion**	**Operation**
`RTI`	Rückkehr aus einer Interrupt-Service-Routine	(SP) → CCR; SP+1 → SP (SP):(SP+1) → B:A; SP+2 → SP (SP):(SP+1) → X; SP+2 → SP (SP):(SP+1) → Y; SP+2 → SP (SP):(SP+1) → PC; SP+2 → SP
`SWI`	Software-Interrupt	(SP-2) → SP; RTN_H:RTN_L → (SP):SP+1 SP-2 → SP; Y → (SP):(SP+1) SP-2 → SP; X → (SP):(SP+1) SP-2 → SP; B:A → (SP):(SP+1) SP-1 → SP; CCR → SP
`TRAP`	Nicht implementierter OpCode	(SP-2) → SP; RTN_H:RTN_L → (SP):SP+1 SP-2 → SP; Y → (SP):(SP+1) SP-2 → SP; X → (SP):(SP+1) SP-2 → SP; B:A → (SP):(SP+1) SP-1 → SP; CCR → SP
`WAI`	Warten auf einen Interrupt durch Übergang in den WAIT-Modus	(SP-2) → SP; RTN_H:RTN_L → (SP):SP+1 SP-2 → SP; Y → (SP):(SP+1) SP-2 → SP; X → (SP):(SP+1) SP-2 → SP; B:A → (SP):(SP+1) SP-1 → SP; CCR → SP

2.5 Interruptverarbeitung

Interrupts (Unterbrechungen) und Exceptions (Ausnahmen) sind Ereignisse, die die Reihenfolge der Befehlsabarbeitung ändern. Bei der CPU12 werden die Begriffe Interrupts und Exceptions synonym verwendet mit der Ausnahme, dass Exceptions auch die drei Resetquellen mit beinhalten. Interrupts können von unterschiedlichen Quellen generiert werden, wie z. B. Timern, SCI, SPI, XIRQ usw. Sobald ein Interrupt auftritt und sie auch freigegeben sind, wird der sich gerade in Bearbeitung befindliche Befehl beendet (mit Ausnahme einiger Fuzzy-Befehle) und eine spezielle Softwareroutine angesprochen, die Interrupt-Service-Routine (ISR) genannt wird. Die Adresse der ISR holt sich die CPU aus der Interrupttabelle, die am oberen Ende des Speichers liegt. Während der Ausführung der ISR, werden automatisch weitere Interrupts gesperrt. Am Ende der ISR wird der RTI-Befehl ausgeführt, der u. a. die Interrupts wieder freigibt, so dass weitere Interruptanforderungen beantwortet werden können.

Abb. 1-5 zeigt die Unterbrechung eines Hauptprogramms durch zwei Interrupts. Die Interrupts bleiben hier während der Interruptbearbeitung gesperrt. Wenn die erste Interruptanforderung auftritt, sind alle Interrupts global durch das I-Bit gesperrt (I-Bit ist Null). Deshalb wird die Anforderung erst nach der Freigabe der Interrupts durch den CLI-Befehl bearbeitet. Das Programm fährt nach dem CLI-Befehl mit der ISR1-Routine fort und kehrt nach der vollständigen Abarbeitung der Routine wieder ins Hauptprogramm zurück. Die zweite Interruptanforderung wird sofort nach Beendigung des sich in

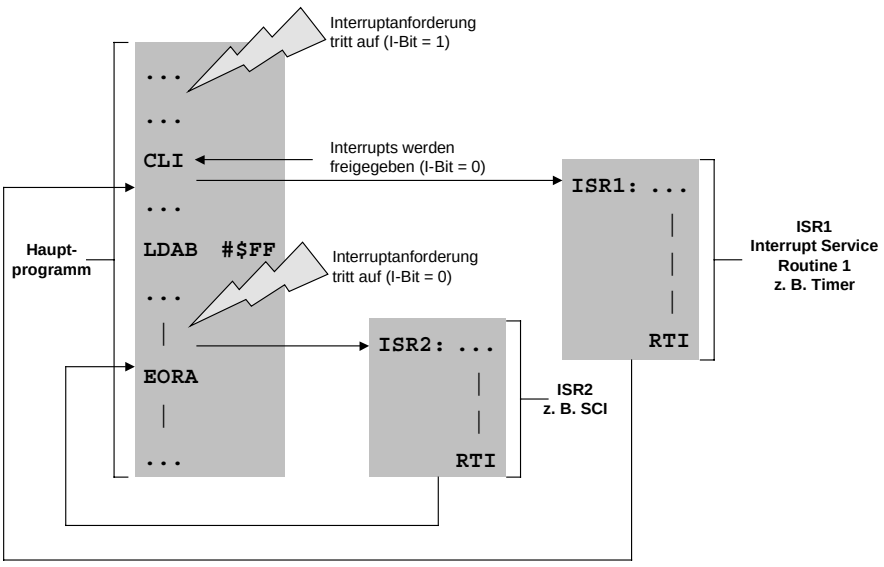

Abbildung 1-5: Unterbrechung des Programmflusses durch zwei Interrupts

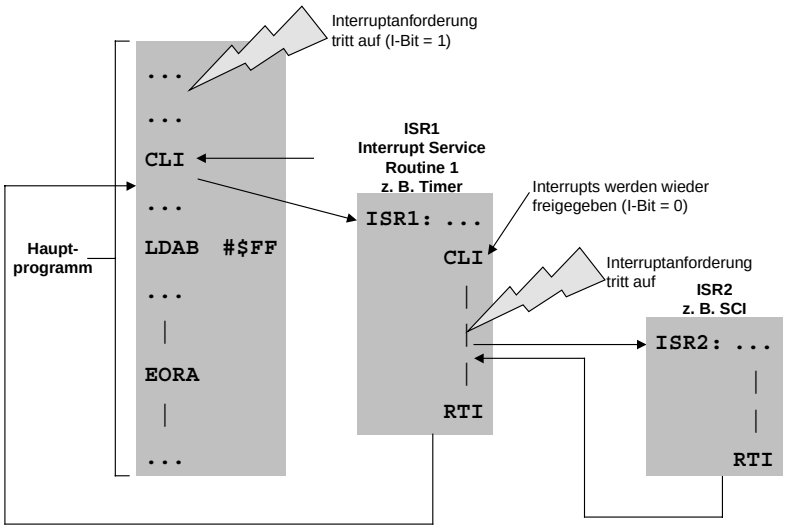

Abbildung 1-6: Verschachtelung von Interrupt-Service-Routinen

Bearbeitung befindlichen Befehls beantwortet und das Programm springt in die ISR2-Routine.

Abb. 1-6 zeigt die Verarbeitung mit verschachtelten Interrupts. Hier können durch Löschen des I-Bits in der ISR-Routine weitere Interrupts bearbeitet werden. Dauern ISR-Routinen sehr lange (was eigentlich beim Design der Software vermieden werden sollte), kann es erforderlich sein, diese ISR-Routine unterbrechbar zu machen. Verwendet man verschachtelte Interrupts muss man jedoch aufpassen, dass nicht zu viele Interrupts auftreten. Dies könnte zu einem Überlauf des Stapelspeichers führen, da für jede ISR der CPU-Kontext auf den Stapelspeicher abgelegt wird.

2.5.1 Exception-Prozess

Tritt eine Exception-Anforderung auf, wird abhängig von der Priorität der Exception die Adresse der entsprechenden ISR-Routine aus der Vektortabelle geholt. Die Priorität gibt bei gleichzeitigem Auftreten von mehreren Exceptions an, welcher Vorrang bei der Bearbeitung gegeben wird. Nach dem Holen des Vektors (Vector Fetch) werden entsprechend der Quelle der Exception drei unterschiedliche Pfade beschritten. Dabei gibt es jeweils einen eigenen Pfad für folgende Exception-Quellen:

- Reset
- Maskierbare Interruptanforderungen (I-Bit, X-Bit)
- SWI und TRAP

Resetverabeitung

Tritt ein Reset auf, werden im ersten Zyklus die S-, X- und I-Bits gesetzt (STOP-Instruktion gesperrt, XIRQ-Interrupt und alle maskierbaren Interrupts sind gesperrt) und alle CPU-Register auf Null gesetzt. Im nächsten Zyklus wird der Resetvektor aus der Resettabelle eingelesen. In den nächsten drei Zyklen wird die dreistufige Befehlspipeline durch Wortzugriffe auf den Speicher gefüllt. Der erste Befehl wird von der Adresse geholt, auf die der Resetvektor zeigt. Wenn die Pipeline gefüllt ist, beginnt die CPU den ersten Befehl in der Pipeline abzuarbeiten.

Verarbeitung der maskierbaren Interrupts

Bei den maskierbaren Interrupts berechnet die CPU nach dem Holen des Interruptvektors im ersten Zyklus die Rücksprungadresse und legt diese auf dem Stapelspeicher ab. Dabei zeigt die Rücksprungadresse auf die nächste Instruktion nach dem Sprungbefehl, bei der der Interrupt aufgetreten ist. Danach wird in drei Zyklen die Befehlspipeline neu gefüllt und die Register Y, X, D (B:A) und CCR auf den Stack gelegt. Das Setzen des I-Bits bei den maskierbaren Interrupts verhindert, dass ein neuer Interrupt ausgeführt werden kann. Tritt ein XIRQ-Interrupt auf, wird zusätzlich zum I-Bit auch das X-Bit gesetzt, um weitere XIRQ-Interrupts zu verbieten. Der Doppelakkumulator D wird aus Kompatibilitätsgründen zum 68HC11 in der Reihenfolge B und dann A auf den Stapelspeicher gelegt. Danach beginnt die CPU die ISR-Routine auszuführen.

SWI- und TRAP-Verarbeitung

Die CPU12 stellt zwei besondere Exceptions zur Verfügung. Der SWI-Interrupt ist ein Assemblerbefehl, mit dem ein Interrupt mit Hilfe der Software erzwungen werden kann. Eine TRAP-Exception tritt auf, wenn versucht wird, eine der 202 nicht verwendeten OpCodes auszuführen. Der einzige Unterschied in der Abarbeitung dieser beiden Exceptions zur Verarbeitung von maskierbaren Interrupts besteht darin, dass die Rücksprungadresse auf den Befehl nach dem SWI-Befehl oder nach dem fehlerhaften TRAP-OpCode zeigt.

2.5.2 Die Exception-Vektortabelle

Jeder Eintrag in der Vektortabelle zeigt auf einen Platz im Speicher, wo die entsprechende Interrupt-Service-Routine (ISR) passend zur Quelle des Interrupts zu finden ist. Die Einträge besitzen Wortgröße, sind also 16 Bit breit und können damit auf jede beliebige Stelle im 64-KByte-Adressraum des Mikrocontrollers zeigen. Der Interruptvektor kann nicht auf den erweiterten Speicher (Expanded Memory) zeigen, d. h. die ISR-Routinen dürfen sich nicht im erweiterten Speicher (Expanded Memory) befinden. Natürlich kann dann in der ISR ein Programm im erweiterten Speicher aufgerufen werden. Die Interrupt-Vektortabelle für den MC9S12DP256 ist in Tabelle 1-2 aufgelistet.

Die ersten sieben Einträge sind bei jedem HCS12-Derivat identisch. Die ersten drei Vektoren sind Reset-Vektoren. Sie können nicht maskiert werden, d. h. man kann sie nicht sperren. Ihre Priorität entspricht der Reihenfolge in der Vektortabelle, d. h. der Systemreset hat die höchste Priorität. Der vierte Eintrag ist der TRAP-Interrupt, der zuschlägt, wenn ein

nicht existierender OpCode eingelesen wurde. Auch die TRAP-Exception kann, wie die drei Resets, nicht maskiert werden. Die nächsten beiden Vektoren sind den externen Interrupts IRQ und XIRQ zugeordnet. Danach sind 120 Vektoren für gerätespezifische Interrupts vorgesehen, die von Derivat zu Derivat variieren können.

2.5.3 Priorität der Interrupts

Die Priorität eines Interrupts legt fest, welcher Interrupt vom System behandelt wird, wenn zwei oder mehrere Interruptanforderungen zur gleichen Zeit anstehen. Die Interruptpriorität ist zum einen durch die Reihenfolge in der Vektortabelle festgelegt. Je höher die Adresse des Vektors, um so höher ist die Priorität. Es besteht jedoch für den Anwender die Möglichkeit, einen mit dem I-Bit maskierbaren Interrupt über alle anderen zu stellen. Dies wird mit dem Register HPRIO (Highes Priority I Interrupt) gemacht, indem das niederwertigere Byte des Interruptvektors in das Register geschrieben wird. Das Register kann nur beschrieben werden, wenn das I-Bit gesetzt ist, also alle I-maskierbaren Interrupts gesperrt sind. Schreibt man einen Wert in das Register, das einem nicht existierenden oder einem nicht-maskierbaren Interrupt oder Reset zugeordnet ist, wird die Standardeinstellung verwendet, die durch die Reihenfolge in der Vektortabelle vorgegeben ist. Bit 0 des HPRIO-Registers ist immer Null, da nur gerade Adressen für die ISR-Routinen verwendet werden dürfen.

Vektoradresse	Quelle	CCR-Maske	Lokale Freigabemöglichkeit	HPRIO Default
$FFFE:$FFFF	Reset	Keine	Keine	–
$FFFC:$FFFD	Reset auf Grund eines Taktratenüberwachungsfehlers	Keine	COPCTL (CME, FCME)	–
$FFFA:$FFFB	Reset auf Grund des COP-Watchdogs	Keine	COP rate select	–
$FFF8:$FFF9	TRAP-Interrupt auf Grund einer nicht implementierten Instruktion	Keine	Keine	–
$FFF6:$FFF7	Software-Interrupt SWI	Keine	Keine	–
$FFF4:$FFF5	Externer Pin-Interrupt XIRQ	X-Bit	Keine	–
$FFF2:$FFF3	Externer Pin-Interrupt IRQ	I-Bit	INTCR (IRQEN)	$F2
$FFF0:$FFF1	Echtzeitinterrupt	I-Bit	CRGINT (RTIE)	$F0
$FFEE:$FFEF	Timer-Kanal 0	I-Bit	TMSK1 (C0I)	$EE
$FFEC:$FFED	Timer-Kanal 1	I-Bit	TMSK1 (C1I)	$EC
$FFEA:$FFEB	Timer-Kanal 2	I-Bit	TMSK1 (C2I)	$EA
$FFE8:$FFE9	Timer-Kanal 3	I-Bit	TMSK1 (C3I)	$E8
$FFE6:$FFE7	Timer-Kanal 4	I-Bit	TMSK1 (C4I)	$E6
$FFE4:$FFE5	Timer-Kanal 5	I-Bit	TMSK1 (C5I)	$E4
$FFE2:$FFE3	Timer-Kanal 6	I-Bit	TMSK1 (C6I)	$E2
$FFE0:$FFE1	Timer-Kanal 7	I-Bit	TMSK1 (C7I)	$E0
$FFDE:$FFDF	Timer-Überlauf	I-Bit	TMSK2 (TOI)	$DE
$FFDC:$FFDD	Pulsakkumulator A Überlauf	I-Bit	PACTL (PAOVI)	$DC
$FFDA:$FFDB	Pulsakkumulator Eingangsflanke	I-Bit	PACTL (PAI)	$DA
$FFD8:$FFD9	Serial Peripheral Interface SPI0	I-Bit	SP0CR1 (SPIE, SPTIE)	$D8
$FFD6:$FFD7	Serial Communication Interface SCI0	I-Bit	SC0CR2 (TIE-TCIE, RIE, ILIE)	$D6
$FFD4:$FFD5	Serial Communication Interface SCI1	I-Bit	SC1CR2 (TIE, TCIE, RIE, ILIE)	$D4

Vektoradresse	Quelle	CCR-Maske	Lokale Freigabemöglichkeit	HPRIO Default
$FFD2:$FFD3	Analog-Digital-Wandler ATD0	I-Bit	ATD0CTL2 (ASCIE)	$D2
$FFD0:$FFD1	Analog-Digital-Wandler ATD1	I-Bit	ATD1CTL2 (ASCIE)	$D0
$FFCE:$FFCF	Port J	I-Bit	PTJIF (PTJIE)	$CE
$FFCC:$FFCD	Port H	I-Bit	PTHIF(PTHIE)	
$FFCA:$FFCB	Modulus-Abwärtszähler Unterlauf	I-Bit	MCCTL(MCZI)	$CA
$FFC8:$FFC9	Pulsakkumulator B Überlauf	I-Bit	PBCTL(PBOVI)	$C8
$FFC6:$FFC7	Takt- and Resetgenerator CRG eingerastet	I-Bit	PLLCR(LOCKIE)	$C6
$FFC4:$FFC5	Self-Clock Mode Enable SCME	I-Bit	PLLCR (SCMIE)	$C4
$FFC2:$FFC3	Byte Level Data Link Controller BDLC (J1850)	I-Bit	DLCBCR1(IE)	$C2
$FFC0:$FFC1	IIC-Bus	I-Bit	IBCR (IBIE)	$C0
$FFBE:$FFBF	Serial Peripheral Interface SPI1	I-Bit	SP1CR1 (SPIE. SPTIE)	$BE
$FFBC:$FFBD	Serial Peripheral Interface SPI2	I-Bit	SP2CR1 (SPIE, SPTIE)	$BC
$FFBA:$FFBB	EEPROM	I-Bit	EECTL(CCIE, CBEIE)	$BA
$FFB8:$FFB9	Flash	I-Bit	FCTL(CCIE, CBEIE)	$B8
$FFB6:$FFB7	msCAN 0 Aufwecken	I-Bit	C0RIER (WUPIE)	$B6
$FFB4:$FFB5	msCAN 0 Fehlerinterrupt	I-Bit	C0RIER (RWRNIE, TWRNIE, RERRIE,TERRE, BOFFIE,OVRIE)	$B4
$FFB2:$FFB3	msCAN 0 Empfangsinterrupt	I-Bit	C0RIER (RXFIE)	$B2
$FFB0:$FFB1	msCAN 0 Sendeinterrupt	I-Bit	C0TIER(TXEIE[2:0])	$B0
$FFAE:$FFAF	msCAN 1 Aufwecken	I-Bit	C1RIER (WUPIE)	$AE

Vektoradresse	Quelle	CCR-Maske	Lokale Freigabemöglichkeit	HPRIO Default
$FFAC:$FFAD	msCAN 1 Fehlerinterrupt	I-Bit	C1RIER (RWRNIE,TWRNIE, RERRIE,TERRE, BOFFIE,OVRIE)	$AC
$FFAA:$FFAB	msCAN 1 Empfangsinterrupt	I-Bit	C1RIER (RXFIE)	$AA
$FFA8:$FFA9	msCAN 1 Sendeinterrupt	I-Bit	C1TIER(TXEIE[2:0])	$A8
$FFA6:$FFA7	msCAN 2 Aufwecken	I-Bit	C2RIER (WUPIE)	$A6
$FFA4:$FFA5	msCAN 2 Fehlerinterrupt	I-Bit	TWRNIE, RERRIE,TERRE, BOFFIE,OVRIE)	$A4
$FFA2:$FFA3	msCAN 2 Empfangsinterrupt	I-Bit	C2RIER (RXFIE)	$A2
$FFA0:$FFA1	MSCAN 2 Sendeinterrupt	I-Bit	C2TIER(TXEIE[2:0])	$A0
$FF9E:$FF9F	MSCAN 3 Aufwecken	I-Bit	C3RIER (WUPIE)	$9E
$FF9C:$FF9D	MSCAN 3 Fehlerinterrupt	I-Bit	C3RIER (RWRNIE,TWRNIE, RERRIE,TERRE, BOFFIE,OVRIE)	$9C
$FF9A:$FF9B	MSCAN 3 Empfangsinterrupt	I-Bit	C3RIER (RXFIE)	$9A
$FF98:$FF99	MSCAN 3 Sendeinterrupt	I-Bit	C3TIER(TXEIE[2:0])	$98
$FF96:$FF97	MSCAN 4 Aufwecken	I-Bit	C4RIER (WUPIE)	$96
$FF94:$FF95	MSCAN 4 Fehlerinterrupt	I-Bit	C4RIER (RWRNIE,TWRNIE, RERRIE,TERRE, BOFFIE,OVRIE)	$94
$FF92:$FF93	MSCAN 4 Empfangsinterrupt	I-Bit	C4RIER (RXFIE)	$92
$FF90:$FF91	MSCAN 4 Sendeinterrupt	I-Bit	C4TIER(TXEIE[2:0])	$90
$FF8E:$FF8F	Port P Interrupt	I-Bit	PTPIF(PTPIE)	$8E
$FF8C:$FF8D	PWM Not-Shutdown	I-Bit	PWMSDN(PWMIE)	$8C
$FF8A:$FF8B	INT8A reserviert für spätere Verwendung	I-Bit	-	$8A
$FF88:$FF89	INT88 reserviert für spätere Verwendung	I-Bit	-	$88

Vektoradresse	Quelle	CCR-Maske	Lokale Freigabemöglichkeit	HPRIO Default
$FF86:$FF87	INT86 reserviert für spätere Verwendung	I-Bit	-	$86
$FF84:$FF85	INT84 reserviert für spätere Verwendung	I-Bit	-	$84
$FF82:$FF83	INT82 reserviert für spätere Verwendung	I-Bit	-	$82
$FF80:$FF81	INT80 reserviert für spätere Verwendung	I-Bit	-	$80

Tabelle 1-22: Tabelle der Interruptvektoren beim MC9S12DP256

2.5.4 Interruptprogrammierung in Assembler

Im folgenden soll ein Beispiel gezeigt werden, wie eine Interruptroutine in Assembler realisiert werden kann. Um dies an einem konkreten Beispiel zu zeigen, wird der Assembler des CodeWarriors von Metrowerks verwendet [4]. Natürlich kann das Beispiel auf jeden anderen CPU12-Assembler angepasst werden. Eine CPU12-Version des CodeWarriors befindet sich auf der beigefügten CD-ROM.

Anlegen der Interrupttabelle

Als erstes wird die Vektortabelle angelegt. Dabei ist zu beachten, dass alle Vektoren initialisiert werden, unabhängig davon, ob sie wirklich verwendet werden oder nicht. Nicht verwendete Interrupts sollten auf eine Dummy-Routine zeigen, die entweder eine Fehlerbehandlung durchführt oder zumindest das I-Bit durch einen RTI-Befehl löscht und damit weitere Interrupts nicht verhindert. Die Interrupttabelle kann beim CodeWarrior auf zwei verschiedene Arten initialisiert werden. Die eine Möglichkeit besteht darin, sie im Linker-Command-File anzulegen (diese Datei heißt beim CodeWarrior PRM-File). Sie kann aber auch in der Quelldatei initialisiert werden. Hier sollen beide Möglichkeiten dargestellt werden.

In der PRM-Datei können einzelne Vektoren angelegt werden und der Programmierer entscheidet, welche Interrupts er verwenden möchte. Das widerspricht jedoch guter Programmierung, wie oben schon ausgeführt wurde. Davon wird abgeraten. Für kleine Testprogramme kann man diese Vorgehensweise jedoch tolerieren. Diejenigen Interrupts, die in der PRM-Datei angelegt werden, müssen jedoch auch in der Quelldatei im Programm verwendet werden. Ansonsten gibt der Linker eine Fehlermeldung aus.

```
/**************************************
*  CPU12 Interrupt-Programm-Beispiel    *
*  Linker-Command-File (PRM-Datei)      *
**************************************/
NAMES   END
SECTIONS
    MY_RAM = READ_WRITE 0x1000 TO 0x2FFF;
    MY_ROM = READ_ONLY  0x3000 TO 0x3EFF;
PLACEMENT
    DEFAULT_ROM      INTO  MY_ROM;
    DEFAULT_RAM      INTO  MY_RAM;
END

STACKSIZE 0x1000

VECTOR 0 main
INIT main
VECTOR ADDRESS 0xFFF0 RTI_isr
ENTRIES
   *
END
```

Listing 1-1: Linker-Parameter-File aus dem Beispiel 1

Listing 1-1 zeigt eine Beispiel-PRM-Datei für die Implementation von einem Interrupt an Hand des Real-Time-Interrupts (RTI). Die Funktion `main` ist mit `INIT` als Einsprungmarke für das Programm nach dem Power-On-Reset definiert. Der Interruptvektor für den RTI-Interrupt kann in der Linker-Parameter-Datei oder im Quellprogramm definiert werden. Wie in dem Listing 1-1 zu sehen ist, wird im Falle der Definition des Vektors in der PRM-Datei die Vektoradresse dem Namen der ISR-Routine zugeordnet:

```
VECTOR ADDRESS 0xFFF0 RTI_isr
```

Es kann auch an Stelle der Vektoradresse die Nummer in der Vektortabelle angegeben werden, wobei die Zählweise beim Systemreset mit Null beginnt. Der RTI-Interrupt kann also auch folgendermaßen geschrieben werden:

```
VECTOR 7 RTI_isr
```

Der Linker berechnet sich dann die richtige Vektoradresse selbst. Das zugehörige Quellprogramm ist in Listing 1-2 dargestellt. Der Programmeinsprung nach dem Reset ist `main`. Die RTI-ISR-Routine wurde hier mit `RTI_isr` bezeichnet und endet mit einem `RTI`.

```
;*****************************************
;*  CPU12 Interrupt-Programm-Beispiel    *
;*  Quelldatei Beispiel 1                *
;*****************************************
        XDEF main,RTI_isr,Dummy_ISR
        XDEF …     ; Weitere Definitionen folgen hier

DataSection:    SECTION
; Hier können die Datendefinitionen stehen

CodeSection:    SECTION
; Hier folgen die Anwenderprogramme

main:   ...             ; Hier kann der Anwender-Code
        ...             ; des Hauptprogramms stehen
        BRA  main

RTI_isr:...             ; Hier kann der Anwender-Code
        ...             ; der Interrupt-Service-Routine
        RTI             ; stehen

Dummy_ISR:              ; Hier kann der Anwender-Code stehen
        ...             ; der Interrupt Service Routine
        ...             ; stehen
        RTI

RTI_ISR:...             ; Hier kann der Anwender-Code
        ...             ; der Interrupt Service Routine
        ...             ; stehen
        RTI

...                     ; Weitere ISR-Routinen können hier folgen
```

Listing 1-2: Quellprogramm aus dem Beispiel 1

Die Definition der Vektortabelle im Quellprogramm ist der Angabe in der PRM-Datei vorzuziehen, da das Programm dann einfacher mit einem anderen Assembler compiliert werden kann. Nicht alle Assembler für die CPU12 unterstützen die Initialisierung im Linker-Command-File. Damit wird das Programm besser portierbar. Die Vektortabelle wird über DC.W-Anweisungen in der Sektion „`VectorTable`“ als Wortkonstante (2 Bytes) definiert, da die Vektoren bei der CPU12 16 Bit breit sind. Beim Einfügen der Tabelle in den Quellcode ist darauf zu achten, dass kein Eintrag ausgelassen wird, da sich sonst die Adressen verschieben.

Alle Einträge müssen mit der Anweisung `XDEF` öffentlich zugänglich gemacht werden, wenn sich die ISR-Routinen in der gleichen Datei befinden, wie die Vektortabelle. Schreibt man die ISR-Routinen in eine separate, eigene Datei, müssen die Einträge mit der Anweisung `XREF` importiert werden.

In der Linker-Parameter-Datei (PRM File) wird ein Speicherbereich für die Vektortabelle `VectorTable` reserviert und im Abschnitt `SECTIONS` zugeteilt:

```
Vector = READ_ONLY 0xFF80 TO 0xFFFF;
```

Im Abschnitt `PLACEMENT` wird dann dieser Speicherbereich dem Bereich `VectorTable` zugewiesen:

```
VectorTable     INTO Vector;
```

Ein Ausschnitt aus der Quelldatei befindet sich in Listing 1-3. Das gesamte Listing befindet sich auf der CD-ROM als Beispiel 2.

Beim Schreiben von Interrupt-Service-Routinen ist zu beachten, dass die CPU12 automatisch alle Register auf den Stapelspeicher rettet, d. h. in der ISR-Routine müssen sie nicht mehr auf den Stapelspeicher gelegt werden. Jede ISR-Routine muss mit einem RTI-Befehl beendet werden. Er sorgt dafür, dass das I-Bit wieder gelöscht wird und die Register vom Stapelspeicher wieder zurückgelesen werden. Die Interruptroutine (ISR) muss sich im nicht-gebankten Speicher befinden.

Interrupt-Service-Routinen (ISR)

Interrupt-Service-Routinen gleichen normalen Funktionen oder Unterprogrammen, mit dem Unterschied, dass sie an Stelle eines RTS-Befehls (Return from Subroutine) mit einem RTI (Return from Interrupt) verlassen werden. Der RTI-Befehl bewirkt, dass die Registers einschließlich des CCR-Register vom Stapelspeicher restauriert werden. ISR-Routinen sollten möglichst kurz gehalten werden, da üblicherweise während deren Ausübung keine weiteren Interrupts bearbeitet werden können. Eine zu lange ISR kann zur Folge haben, dass mehrere Interruptanforderungen derselben Interruptquelle (z. B. Timer) aufeinander folgen, ohne bearbeitet werden zu können. Sie gehen dann verloren, da eine Interruptanforderung derselben Quelle nur einmal anhängig (Pending) bleiben kann. In einer ISR-Routine sollte man lediglich die Daten des Peripheriemoduls (empfangene Daten, Zählerwerte usw.) auslesen, entsprechende Flags für das Hauptprogramm setzen und das Interruptstatusflag löschen. Weitere Aktionen, insbesondere mathematische Berechnungen verlagert man in die Hauptschleife des Programms.

Bei jeder Interruptanforderung von einem Peripheriemodul muss das entsprechende Interruptstatusflag zurückgesetzt werden, damit weitere Interruptanforderungen desselben

```
;***********************************************
;*  CPU12 Interrupt-Programm-Beispiel        *
;* Beispiel 2                                *
;***********************************************

            XDEF   main
            XDEF   Dummy_ISR
            XDEF   PWM_Shutdown_ISR,Counter_ISR
initStk:   EQU     $3FFF+1

DataSection:     SECTION
;
; Hier können die Datendefinitionen stehen
;

CodeSection: SECTION
;
; Hier folgen die Anwenderprogramme
;

main:   LDS     #initStk        ; Stackpointer laden
        CLI                   ; Interrupts freigeben
        ; Hier kann der Anwender-Code
        ; des Hauptprogramms stehen
        BRA     main

Dummy_ISR: ; Hier kann der Anwender-Code
        ; der Interrupt Service Routine
        ; stehen
        RTI

RTI_ISR:   ; Hier kann der Anwender-Code
        ; der Interrupt Service Routine
        ; stehen
        RTI
;
; Definition der Vektortabelle
;

VectorTable: SECTION

   DC.W Dummy_ISR       ; reserviert
   ...  ; Hier folgen weitere Interruptvektoren
   DC.W Clock_Mon_ISR   ; Clock Monitor
   DC.W main         ; Programmeingang, Hauptprogramm
```

Listing 1-3: Ausschnitt aus dem Quellprogramm von Beispiel 2

Moduls wieder zugelassen sind. Das Interruptstatusflag befindet sich in einem der Kontrollregister des Peripheriemoduls. Es zeigt an, dass ein Interrupt für dieses Modul anhängig ist und bearbeitet werden möchte. Ist das lokale Interruptfreigabebit der Peripheriefunktion gesetzt, d. h. Interrupts für diese Quelle sind freigegeben, dann löst das lokale Interruptstatusflag einen Interrupt aus. Ansonsten zeigt es lediglich an, dass ein Ereignis aufgetreten ist, welches einen Interrupt auslösen könnte. Wird das

Interruptstatusflag in der ISR nicht zurückgesetzt, kann es ständig Interrupts auslösen oder weitere Interrupts sperren. Was bei einem nicht zurückgesetzten Interruptstatusflag passiert ist implementationsabhängig. Die CPU12 führt dann ständig die gleiche ISR-Routine aus und befindet sich quasi in einer Endlosschleife.

Die Art und Weise, wie das Interruptstatusflag zurückgesetzt wird, kann bei den einzelnen Peripheriefunktionen unterschiedlich sein. So gibt es die Möglichkeit, dass durch einen Lesezugriff auf ein Statusregister, sich das Interruptstatusflag automatisch löscht. Dies ist z. B. bei der SCI der Fall. Wird ein Datum empfangen, wird dies durch das gesetzte Interruptstatusflag RDRF (Receive Data Register Full Flag) im SCISR1-Register (SCI Status Register 1) angezeigt. Durch Lesen des Kontrollregisters SCISR1 und anschließendes Auslesen des Datenregisters SCIDRL wird das RDRF-Flag wieder zurückgesetzt. Eine andere Möglichkeit besteht darin, in das Statusflag eine eins zu schreiben, damit es rückgesetzt wird. Das ist beispielsweise der Fall beim Timer-Überlauf-Interrupt. Tritt ein Überlauf auf, wird das Interruptstatusflag TOF (Timer Overflow Flag) im TFLG2-Register (Main Timer Interrupt Flag Register 2) gesetzt. Das Rücksetzen des Flags erreicht man, indem eine 1 auf die Stelle des Flags (hier Bit 7) geschrieben wird. Die dritte Möglichkeit ist, das Interruptflag direkt zurückzusetzen, d. h. eine Null an die Stelle des Interruptflags im Kontrollregister zu schreiben. Dies ist beispielsweise der Fall beim Empfang eines Datagramms der msCAN-Schnittstelle. Wurde eine vollständige Botschaft empfangen, wird das RXF-Flag (Receive Buffer Full) im CANRFLG-Register (msCAN Reveiver Flag Register) gesetzt. Das Flag wird gelöscht, indem eine Null an seine Stelle im Register (Bit 0) geschrieben wird.

Wird im Programm der Interrupt einer Peripheriefunktion durch Löschen des entsprechenden Interruptfreigabebits gesperrt, sollte darauf geachtet werden, dass vorher das I-Flag gesetzt ist. Das ist entweder in einer ISR-Routine automatisch der Fall oder, nachdem im Programm der Befehl SEI ausgeführt wurde. Bei den 68HC12-Derivaten kann es auf Grund eines Designfehlers zu unbeabsichtigten falschen Interrupts kommen, wenn gerade beim Rücksetzen des Interruptfreigabebits ein Interrupt auftritt und das I-Bit nicht gesetzt ist. Fälschlicherweise wird dann ein SWI-Interrupt ausgeführt. Ein Beispiel für das korrekte Sperren eines Interrupts ist in Listing 1-4 dargestellt.

Beim Setzen des I-Flags durch das Programm mit dem SEI-Kommando (oder ORCC #$10) werden sofort alle maskierbaren Interrupts gesperrt. Das Freigeben der Interrupts durch den CLI-Befehl (oder ANDCC #$EF) wird um einen Taktzyklus (E Clock) verzögert. Der Grund hierfür besteht in der Anwendung des WAI-Befehls. Er stellt eine

```
.
.
PSHC              ; CCR (Status des I-Bits) merken
SEI               ; Sperren aller maskierbaren Interrupts
BCLR TMSK1,#$01   ; Timer Kanal 0 Interrupts sperren
PULC              ; CCR restaurieren
.
.
```

Listing 1-4: Sperren der maskierbaren Interrupts (I-Bit = 1) vor dem Sperren von Peripherieinterrupts

```
.
.
CLI     ; Freigabe der maskierbaren Interrupts
WAI     ; WAIT-Modus, warten auf Interrupts
.
.
```

Listing 1-5: Eintritt in den WAIT-Zustand bei vorheriger Freigabe der Interrupts

Besonderheit dar, da er die CPU12 veranlasst in den WAIT-Zustand überzugehen, die CPU vom Takt abzuklemmen und auf einen Interrupt zu warten. Damit wird verhindert, dass zwischen dem CLI- und dem WAI-Befehl ein Interrupt bearbeitet wird und der Mikrocontroller nach der Interruptbearbeitung in den WAIT-Zustand geht (siehe Listing 1-5).

Bei anderen Mikrocontrollerarchitekturen müssen am Beginn der ISR-Routine weitere Interrupts gesperrt und am Ende vor dem RTI die Interrupts wieder freigegeben werden. Das ist bei der CPU12 nicht erforderlich!

2.5.5 Interruptprogrammierung in C

Natürlich können und sollen Interruptroutinen auch in C geschrieben werden. Da die C-Compiler für die CPU12 sehr effektiv sind, ist es heutzutage nicht mehr notwendig, ISR-Routinen in Assembler zu verfassen. Es kann natürlich im Einzelfall notwendig sein, ein

```
   /*************************************
*  CPU12 Interrupt-Programm-Beispiel *
*  Linker-Command-File (PRM-Datei)   *
*  Beispiel 3: MC9S12DP256         *
*************************************/

NAMES   END

SECTIONS
/* Definition des Speicherbereichs für die Vektortabelle */
   Vector = READ_ONLY 0xFF80 TO 0xFFFF;
END

PLACEMENT
/* Die Section 'VectorTable' wird an die entsprechende   */
/* Adresse gelegt                                        */
   VECTORS      INTO Vector;
END

INIT _Startup

ENTRIES
   VectorTable
END
```

Listing 1-6: Linker-Parameter-File für eine Vektortabelle in C (Beispiel 3)

Programmteilstück in Assembler zu verbessern und zu optimieren.

Anlegen der Interrupttabelle

Auch in C gibt es, wie in Assembler, die beiden Möglichkeiten, eine Interruptvektortabelle anzulegen. Die Art und Weise, die Vektortabelle im Linker-Parameter-File einzutragen ist in Abschnitt 1.5.4 beschrieben und kann auch hier angewendet werden. Eine weitere Möglichkeit besteht darin, die Vektortabelle in die Quelldatei zu schreiben. Listing 1-7 zeigt diese Möglichkeit am Beispiel des MC9S12DP256-Mikrocontrollers auf (Beispiel 3 auf der CD-ROM).

Damit der C-Compiler eine Funktion als Interrupt-Service Routine erkennt, muss ihm über die Pragma-Anweisung „`#pragma TRAP_PROC`“ mitgeteilt werden, dass es sich bei der

```
/*********************************************
*  CPU12 Interrupt-Programm-Beispiel           *
*  Beispiel 3                                  *
*********************************************/

/* Funktionsprototypen  */
void Dummy_ISR( void );         // Dummy ISR-Routine
void PWM_Shutdown_ISR( void );  // PWM Emergency Shutdown
void Port_P_ISR( void );        // Port P
...
...
void Clock_Mon_ISR( void );     // Clock Monitor

/* Extern definierte Funktionen */
extern void _Startup( void );   // Startprogramm

/* Interrupt-Service-Routinen   */
#pragma TRAP_PROC
void Dummy_ISR( void )
{}

#pragma TRAP_PROC
void PWM_Shutdown_ISR( void )
{}

...
...

/* Vektortabelle                */
#pragma CONST_SEG VECTORS

void (* const VectorTable[])() =
{  Dummy_ISR,      // reserviert
   ...
   ...
   Clock_Mon_ISR,  // Clock Monitor
   _Startup        // Programmeingang, Hauptprogramm
};
```

Listing 1-7: Interrupt-Vektortabelle in C

folgenden Funktion um eine Interruptroutine handelt. Die Pragma-Anweisung ist nur für die folgende Funktion gültig, deshalb muss sie vor jede ISR-Routine geschrieben werden. Eine andere Möglichkeit wäre, vor die Funktion das Schlüsselwort „`interrupt`“ zu setzen.

Die Vektortabelle „`VectorTable`“ wird durch die Pragma-Anweisung „`#pragma CONST_SEG VECTORS`“ in einen im Linker-Command-File festgelegten Speicherbereich gelegt. Das zu Listing 1-7 gehörige Linker-Command-File ist in Listing 1-6 aufgeführt. Dort wird das konstante Segment „VECTORS“ in den Adressbereich der Interruptvektoren von 0xFF80 bis 0xFFFF festgelegt.

2.5.6 Zeiten bei der Interruptverarbeitung

Zwischen dem Auftreten der Interruptanforderung und der Abarbeitung der ISR-Routine vergeht eine bestimmte Zeit, die Interruptverzögerungszeit (Interrupt Latency) genannt wird. Diese Zeit sollte möglichst kurz sein, damit schnell auf die Anforderungen reagiert werden kann. Embedded-Anwendungen sind oft durch viele Echtzeitanforderungen gekennzeichnet, die es gilt, schnell zu bearbeiten. Die maximale Interruptverzögerungszeit wird bei der CPU12 durch drei Faktoren bestimmt: Die Ausführungszeit des am längsten dauernden Befehls, die Zeit, die benötigt wird, die CPU-Register auf den Stack zu retten und die Zeit den Interruptvektor zu holen und den ersten Befehl der ISR-Routine auszuführen. Bei der folgenden Betrachtung wird davon ausgegangen, dass nur On-Chip-Speicher verwendet werden, der Mikrocontroller sich also im Single-Chip-Modus befindet.

Der längste nicht unterbrechbare Befehl ist EMACS (Extended Multiply and Accumulate, Signed), der 13 Taktzyklen benötigt. Es gibt zwar Fuzzy-Logik-Befehle, die sehr viel länger brauchen, wie z. B. WAV, REV und REVW. Diese Befehle werden jedoch bei einer Interruptanforderung unterbrochen. Das Speichern der Register auf den Stapelspeicher benötigt weitere 4 Taktzyklen und einen zusätzlichen Taktzyklus für die Bestimmung der Rücksprungadresse. Man sollte bedenken, dass deshalb in der ISR-Routine keine Push- und Pull-Befehle mehr erforderlich sind. Weitere 3 Taktzyklen sind für das Füllen der Instruktionspipeline notwendig. Das ergibt in Summe 13 + 4 + 1 + 3 = 21 Taktzyklen. Die Tabelle 1-23 gibt verschiedene maximale Interruptverzögerungszeiten für unterschiedliche Taktfrequenzen an.

Die reine Interruptverzögerungszeit gibt jedoch noch nicht die Echtzeitleistungsfähigkeit eines Systems an, da es nicht berücksichtigt, dass in einem realen System mehrere

Busfrequenz	Zykluszeit	Interruptverzögerungszeit
8 MHz	125 ns	2,625 µs
25 MHz	40 ns	0,840 µs
33 MHz	30,3 ns	0,636 µs

Tabelle 1-23: Interruptverzögerungszeiten für verschiedene Busfrequenzen

Interruptanforderungen auf ihre Bearbeitung warten. Die Angabe der Interruptreaktionszeit ist schwierig und vom Software-Design abhängig. Sie gibt an, nach welcher maximalen Zeit eine Reaktion auf einen Interrupt in einem realen System zu erwarten ist. Die CPU12 stellt zwei Hardware-Mechanismen zur Verfügung, um die Interruptreaktionszeit zu verbessern. Ist mehr als eine Interruptanforderung im System anhängig, werden die CPU-Register beim RTI der ersten ISR-Routine nicht restauriert, sondern lediglich der Stackpointer angepasst. Das spart zum einen die Zeit für das Rückspeichern der Register vom Stack und zum anderen das erneute Abspeichern der Register auf den Stack beim Abbarbeiten der zweiten ISR-Routine. Ein RTI-Befehl benötigt einen Taktzyklus, wenn keine weiteren Interrupts anhängig (Pending) sind. Er benötigt zwei Taktzyklen, wenn weitere Interrupts zu bearbeiten sind. Der zweite Taktzyklus wird für das Anpassen des Stackpointers aufgewendet. Eine andere Möglichkeit, die Interruptreaktionszeiten des Systems zu verbessern, besteht darin, die höchste Priorität für einen maskierbaren Interrupt mit Hilfe des HPRIO-Registers einzustellen. Dies kann man auch dynamisch durchführen. Wird beispielsweise in einem Netzwerk eine schnelle Kommunikation über die SPI oder SCI gefordert, kann die Priorität der Kommunikationsschnittstelle an oberster Stelle stehen. Damit nun das System nicht unnötig durch Nachrichten blockiert wird, die nicht für den empfangenen Knoten bestimmt sind, wird die Priorität in der ISR-Routine der Kommunikationsschnittstelle verringert, wenn das zu empfangende Datagramm nicht für den Empfänger bestimmt ist. Dazu wird das erste empfangene Byte als Adresse

ITCR – Interrupt Test Control Register, Adressoffset: $0015

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	0	0	0	WRTINT	ADR3	ADR2	ADR1	ADR0
Reset:	0	0	0	0	1	1	1	1

WRTINT – Write to the Interrupt Test Registers
Das Bit kann jederzeit gelesen werden, jedoch beschrieben nur, wenn sich die MCU im Special Mode befindet und das I-Bit im CCR-Register gesetzt ist.
1 = Trennt die Interrupteingänge von dem Prioritätendecoder und verwendet an Stelle dessen die Werte im ITEST-Register
0 = Verbietet das Schreiben auf die Testregister; Lesen der Testregister geben den Status des Interrupts wieder.
Jeder Interrupt, der zu dem Zeitpunkt anliegt, wenn WRTINT gesetzt wird, bleibt bis er überschrieben worden ist.

ADR3 bis ADR0 – Test Regiser Select Bits
Die Bits können jederzeit beschrieben und gelesen werden.
Diese Bits bestimmen die zweite Stelle (von rechts) der Interruptvektoradresse. Eine $07 in ADR geschrieben, wählt die Vektoren $FF70 bis $FF7F. Der einzelne Vektor aus diesem Bereich wird durch ITEST festgelegt.

ITEST – Interrupt Test Register
Adressoffset: $0016

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	INTE	INTC	INTA	INT8	INT6	INT4	INT2	INT0
Reset:	0	0	0	0	0	0	0	0

INTE, INTC, INTA, INT8, INTC, INT4, INT2, INT0
Befindet sich in ADR3-ADR0 der Wert $0F, kann nur Bit 2-0 im ITEST-Register zugegriffen werden, da nur die durch das I- und X-Bit maskierbaren Interrupts getestet werden können. Die Bits 7-3 werden dann als Null gelesen. Das Register kann man beschreiben, wenn WRTINT = 1 und I-Bit = 0 in CCR ist.

Abbildung 1-7: Beschreibung der Kontrollregister ITCR und ITEST

ausgewertet und die Priorität von SCI oder SPI im HPRIO-Register entsprechend gesetzt.

2.5.7 Testen von Interrupts

Das Testen von ISR-Routinen ist meistens aufwendig, da es oft schwierig ist, die Interruptursache zu emulieren. Bei seriellen Schnittstellen benötigt man einen weiteren Teilhaber an der Schnittstelle, der Daten senden und empfangen kann. Bei der SCI ist das noch relativ einfach, da oft schon ein einfaches Terminal(programm) genügt. Aufwendiger wird das jedoch bei der CAN-, IIC- oder SPI-Schnittstelle. Damit man nicht die Ursache eines Interrupts für den Test der Software emulieren muss, hat die HCS12D-Familie einen Testmodus eingebaut, mit dem man mit Software den Interrupt erzwingen kann. Dieser Testmodus steht nur in den Spezialmodi zur Verfügung, d. h. wenn beispielsweise über das BDM-Interface (Background-Debug-Modus) getestet wird.

Die HCS12D-Mikrocontroller stellen für den Test von Interrupts zwei Kontrollregister zur Verfügung. Im ITCR-Register (Interrupt Test Control Register) befindet sich ein Kontrollbit (WRTINT-Flag, Write to the Interrupt Test Register), mit dem das eigentliche Testregister gesperrt oder freigegeben werden kann. Zusätzlich wird ein Teil der Vektoradresse in diesem Register angegeben. Im eigentlichen Testregister ITEST (Interrupt Test Register) wird der zu testende Interruptvektor ausgewählt. Eine ausführlichere Beschreibung der beiden Interrupttestregister ist in Abb. 1-7 ersichtlich.

2.6 Resets

Die Resets sind besondere Ausnahmezustände (Exceptions), die eigene Vektoren haben und nicht maskiert werden können. Der Reset-Zustand stellt einen wichtigen Zustand für den Mikrocontroller dar. In diesem Zustand läuft die CPU nicht mehr und die Register, Peripheriegruppen und Pins werden in einen vordefinierten Zustand versetzt.

2.6.1 Resetquellen

Insgesamt gibt es vier möglichen Quellen für einen Reset:

1. Power-On-Reset
2. Externer Reset
3. Watchdog-Reset (COP, Computer operates properly)
4. Taktüberwachungsreset (Clock Monitor)

Der Power-On-Reset wird aktiviert, wenn die Versorgungsspannung angelegt wird. Durch den externen Reset, der über einen eigenen Pin verfügt, kann der Mikrocontroller auch von außerhalb in den Resetzustand versetzt werden oder der Mikrocontroller bringt durch einen internen Reset sich selbst und externe Geräte in den Resetzustand. Beide Resets, Power-On-Reset und externer Reset teilen sich den selben Resetvektor.

Viele Mikrocontroller haben zur Überwachung einen Watchdog eingebaut. Dabei handelt es sich um einen Zähler, der beim Überlauf einen Reset erzeugt, wenn er nicht vorher

zurückgesetzt wird. Meist ist die Zeitspanne vom Starten des Zähler bis zum Überlauf in weiten Bereichen flexibel einstellbar. Der COP-Reset und der Taktüberwachungsreset (Clock Monitor Reset) teilen sich ebenfalls einen Vektor.

Der Übergang des Mikrocontrollers in den Reset-Zustand ist vollständig asynchron und benötigt deshalb keinen Systemtakt. Zum Verlassen des Resetzustandes ist natürlich ein Takt notwendig.

2.6.2 Power-On-Reset

Der Power-On-Reset tritt beim Einschalten des Mikrocontrollers oder auch bei kurzzeitigen Spannungseinbrüchen auf. Eine auf dem Mikroontroller vorhandene Schaltung überwacht die interne Versorgungsspannung V_{DD} (Details sind in Kapitel 6.5 zu finden) und führt bei Erreichen eines bestimmten Spannungswertes eine Power-On-Reset-Sequenz aus. Der Verlauf dieser Sequenz hängt davon ab, ob der externe Reset-Pin auf den Logikpegel High oder Low gezogen wurde (siehe Abb. X.X).

Abbildung X.X: Power-On-Reset Sequenz in verschiedenen Fällen

a) externer /RESET-Pin wird mit Pullup auf H gezogen

b) externer /RESET-Pin bleibt eine Weile auf L

Der externe Reset-Pin ist Ein- und Ausgang zugleich. Deswegen sollte an den Reset-Pin ein externer Pull-Up-Widerstand angebracht werden, der den Reset immer auf einem logischen High-Pegel hält, ihn also deaktiviert. Tritt nun ein interner oder externer Reset auf, wird der Pin auf Low gezogen und aktiviert. Damit läßt sich der Reset-Pin als Master-Reset für das gesamte System verwenden (Abb. X.X a).

Wenn der Reset-Pin von einer externen Quelle aktiviert wird (Pin auf Low, Masse), dann geht der Mikrocontroller in den Reset-Zustand. Diesen verlässt er erst wieder, wenn der Reset-Pin wieder auf High gezogen wird. Üblicherweise kommt die externe Auslösung des Resets von einer Reset- oder Spannungsüberwachungs-Schaltung. Die interne POR-Schaltung kann lediglich die interne Versorgungsspannung V_{DD} überwachen, deswegen wird empfohlen, trotzdem noch eine externe Spannungsüberwachung zu verwenden.

Bei einem Power-On-Reset gibt es erst mal eine eingebaute Verzögerungszeit, die 4096 ECLK-Taktzyklen beträgt. Erst dann beginnt die eigentliche Reset-Rückkehr-Sequenz. Diese wartet weitere 64 ECKL-Taktzyklen ab. Da die Rückkehr aus dem Reset immer mit diesen Zählzeiten verbunden ist, funktioniert sie nur, wenn ein Systemtakt vorhanden ist.

Der Reset-Pin wird während der gesamten Zeit von 4096 + 64 Taktzyklen über die interne Schaltung auf L getrieben (unabhängig davon, ob er von einer externen Quelle ebenfalls auf L gelegt wurde). Danach wird dieser interne Treiber abgeschaltet und nach weiteren 32 Taktzyklen wird überprüft, ob der Reset-Pin nun auf logisch H liegt (das kann er dann aber nur, wenn ihn keine externe Quelle mehr auf L treibt). Deswegen sollte die externe Beschaltung des Reset-Pins keine großen Kapazitäten beinhalten, damit die Spannung an dem Pin innerhalb von 32 ECLK-Taktzyklen von L auf H steigen kann.

Wenn der Reset-Pin nach den 32 ECLK-Zyklen immer noch als L festgestellt wird, dann nimmt man an, daß es sich bei dem Reset um ein Ereignis handelt, welches von einem externen Baustein ausgelöst wurde. Deswegen wird dann, wenn der Reset-Pin endlich

losgelassen wird (also auf H geht) der normale Reset-Vektor $FFFE:FFFF in den Programmzähler geladen und auf die Adresse gesprungen, auf welche dieser verweist.

Falls dagegen der Reset-Pin nach den besagten 32-ECLK Zyklen auf H steht, dann wird angenommen, daß es sich um einen Reset von einer internen Quelle gehandelt hat, also COP Time-Out oder Clock Monitor Fehler.

Deswegen wird zuerst die Clock Monitor Schaltung geprüft und wenn diese anzeigt, daß ein Fehler vorlag, dann wird der Clock-Monitor-Vektor $FFFC:FFFD aufgenommen und auf die entsprechende Adresse gesprungen.

Falls kein Clock Monitor Fehler vorlag, dann wird der COP-Watchdog geprüft. Falls dieser signalisiert, daß ein Timeout vorlag, dann wird der COP-Vektor $FFFA:FFFB verwendet.

Falls weder Clock Monitor noch COP Schaltung einen entsprechenden Fehler signalisiert hatten, kommt doch wieder der normale Reset-Vektor $FFFE:FFFF zur Anwendung.

2.6.3 Watchdog-Reset

Der COP-Watchdog ist ein interner Zähler, der beim Überlauf einen Reset erzeugt. Wird der Watchdog jedoch vor dem Überlauf mit Hilfe der Software zurückgesetzt, wird dieser Reset verhindert. Man kann den Watchdog dazu verwenden, den Ablauf der Software zu überprüfen. Dazu plaziert man die Befehle für das Rücksetzen des Watchdogs in verschiedene Teile der Software. Werden diese Bereiche nicht nach einer bestimmten Zeit von der Software ausgeführt, läuft der Watchdogzähler über und bring das System in den Grundzustand (Reset). Damit verhindert man, dass sich das Programm ungewollt in Endlosschleifen befindet und das System lähmt.

Im Watchdog-Kontrollregister COPCTL ermöglichen drei Bits die Auswahl von sieben verschiedenen Zeitintervallen für Dauer des Watchdog-Zählers bis zum Überlauf (Time-out). Bei aktiviertem COP-Watchdog muss das laufende Programm nacheinander die Werte $55 und $AA in das ARMCOP-Register schreiben, um den Watchdog zurückzusetzen. Und das natürlich bevor das vorgegebene Zeitintervall abgelaufen ist. Nach dem Rücksetzen des Watchdogs beginnt das Zeitintervall erneut. Der Mikrocontroller führt einen Reset durch, wenn das Zeitintervall abläuft, ohne das ein Wert eingetragen wurde, oder wenn eine andere Sequenz als $55 und $AA in das Register geschrieben wird.

Eine zusätzliche Sicherheit bietet eine Zeitfenster-Option, die aktiviert durch das Setzen des WCOP-Bits (Windoded COP) im COPCTL-Register (COP Control Register) aktiviert wird. In diesem Modus muss das Zurücksetzen des Watchdog-Zählers innerhalb eines bestimmten Zeitfensters erfolgen. Dieses Zeitfenster liegt im letzten Viertel des Zeitintervalls des Watchdog-Zählers. Erfolgt das Rücksetzen zu früh oder zu spät, wird ein Reset ausgelöst. Der COP-Zähler kann auch durch Löschen der Bits RTICTL[6:0] und COPCTL[2:0] zurückgesetzt werden.

2.6.4 Clock-Monitor-Reset

Wenn die Taktüberwachung aktiv ist, wird ein Reset ausgelöst, falls die Taktfrequenz

unter eine vorgegebene Grenze fällt. Die Clock-Monitor-Schaltung benutzt als Referenz die Zeitkonstante eines internen RC-Glied, damit ist man unabhängig vom eigentlichen Takt. Falls innerhalb der RC-Periode keine Taktflanken festgestellt werden, wird ein Reset ausgelöst. Als Eingangstakt dafür wird der REFCLK verwendet.

Die Funktion des Clock Monitors wird durch das CME-Bit (Clock Monitor Enable) im PLLCTL-Register (PLL Control Register) aktiviert. Da der Clock Monitor als Referenz die Zeitkonstante des RC-Glieds verwendet, wird zu dessen Arbeit kein weiteres Taktsignal benötigt. Falls der Takt also vollständig ausfällt, wird der Mikrocontroller trotzdem über die Clock-Monitor-Schaltung in den Reset-Zustand versetzt, aus welchem er jedoch ohne Takt nicht mehr herauskommt.

2.6.5 Auswirkungen des Reset

Wenn ein Reset auftritt, dann werden die Kontrollregister und die Pins des Mikrocontrollers auf einen definierten Startzustand gesetzt. Der Reset wirkt sich auf die Betriebsarten des Mikrocontrollers, auf die Takterzeugung und Watchdog, die Interrupts, die Pins, die CPU, den Speicher und auch auf die Peripherie-Module aus.

Betriebsarten

Die Betriebsarten und die Verteilung des Speichers des Mikrocontrollers werden durch den Zustand der BKGD-, MODA- und MODB-Pins während des Reset definiert und finden sich während der steigenden Flanke des Reserts im MODE-Register wieder. Die Betriebsarten können während des Betriebs des Mikrocontrollers mit gewissen Einschränkungen umgeschaltet werden.

Takterzeugung und Watchdog

Der Watchdog-Zähler ist nach einem Reset aktiviert und die Bits CR[2:0] sind auf das längst mögliche Zeitintervall eingestellt. Die Taktüberwachungsschaltung (Clock Monitor) ist aber deaktiviert. Das Real-Time-Interrupt-System zur automatischen Erzeugung von Echtzeit-Interrupts ist ausgeschaltet. Das entsprechende RTIF-Flag ist gelöscht, damit werden die Interrupts maskiert. Die Bits zur Einstellung der Wiederholrate der Real-Time-Interrupts sind gelöscht und müssen initialisiert werden.

Interrupts

Die Interruptprioritäten sind nach einem Reset auf die Reihenfolge in der Interruptvektortabelle festgelegt. Der IRQ-Interrupt nimmt damit die höchste Priorität ein (siehe Tabelle X.X). Alle maskierbaren Interrupts sind nach einem Reset gesperrt (I- und X-Bit sind gesetzt). Auch die entsprechenden Interrupt-Freigabebits in den einzelnen Modulen sind so gesetzt, dass die entsprechenden Interrupts gesperrt sind und einzeln freigegeben werden müssen.

Port-Pins

Wenn der Mikrocontroller im Single-Chip-Modus aus dem Reset kommt, sind alle Port-

Pins als Eingänge mit hohem Eingangswidwerstand (Tri-State) konfiguriert. Falls der Mikrocontroller in einem der erweiterten Modi (Expanded Modes) gestartet wird, werden die Pins der Ports A und B als Address- und Datenbus und Port E als Kontrollbus verwendet. Alle anderen Pins sind alls allgemeine Eingänge geschaltet.

CPU12

Nach dem Reset springt die CPU zu einer vorgegebenen Adresse, die im Reset-Vektor definiert ist und beginnt Instruktionen anszuführen. Der Inhalt des Stackpointers ist auf die Adresse $FF geladen und die Werte in den Akkumulatoren und X- und Y-Registern sind auf Null eingestellt. Die Interruptflags X und I und das S-Bit des CCR-Registers sind gesetzt, so dass Interrupts und der STOP-Befehl unterdrückt werden.

Speicher

Nach dem Resets sind die Speicherbereiche entsprechend der Modi auf bestimmte Adressen festgelegt.

Abbildung X.X: Speicherverteilung des MC9S12DP256 im

Single Chip Modus nach dem Reset

Beim MC9S12DP256 befindet sich der interne Registerblock von $0000 bis $03FF. Die 12 KByte RAM-Speicher liegen von $1000 und $3FFF. Das EEPROM-Modul beginnt bei Adresse $0000 und endet bei $0FFF. Es überlappt mit dem internen Registerblock und sollte deshalb, wenn man es vollständig verwenden will, verschoben werden. Im Single-Chip-Modus liegt je ein 16 KByte großes Flash-Speichermodul zwischen $4000 und $7FFF sowie zwischen $C000 und $FFFF. Auf den Rest des Flash-Speichers kann in 16 KByte großen Blöcken zwischen $8000 und $BFFF zugegriffen werden (Abbildung X.X).

Peripherie

Alle Periepheriemodule, wie beispielsweise Timer, PWM, SCI, SPI, I²C, J1850, CAN und A/D-Wandler sind nach dem Reset ausgeschaltet und müssen für einen ordnungsgemäßen Betrieb entsprechend freigegeben und initialisiert werden.

2.7 Betriebsarten des Mikrocontrollers

Die 68HC12/HCS12-Mikrocontroller können in unterschiedlichen Betriebsarten (Modes) starten und betrieben werden. Je nach eingestellter Betriebsart stehen unterschiedliche Speicherkonfigurationen zur Verfügung. Die Betriebsarten werden nach dem Reset über die drei Konfigurationspins MODA, MODB und MODC festgelegt. Nach dem Starten des Mikrocontrollers können die Betriebsarten auch per Software umgestellt werden. Der Zustand der MODA-, MODB- und MODC-Pins spiegelt sich in dem MODE-Register wieder.

Die acht verfügbaren Betriebsarten kann man in zwei übergeordnete Betriebsarten

einteilen: Grund- (Normal Mode) und Testbetriebsart (Special Mode). In der Grundbetriebsart (Normal Mode) sind einige Bits in den Kontrollregister gegen ungewollte Änderungen geschützt. Die Testbetriebsarten erlauben größeren Zugriff auf diese geschützten Bits, die für Testzwecke vorhanden sind.

#	MODC	MODB	MODA	Beschreibung der Betriebsart
1	0	0	0	Special Single Chip Mode, BDM ist erlaubt und sofort nach dem RESET aktiv
2	0	0	1	Emulation Expanded Narrow Mode, [1]
3	0	1	0	Special Test Mode (Expanded Wide), [1]
4	0	1	1	Emualtion Expanded Wide Mode, [1]
5	1	0	0	Normal Single Chip Mode, [1]
6	1	0	1	Normal Expanded Narrow Mode, [1]
7	1	1	0	Peripheral Mode, BDM ist erlaubt, aber Busoperationen würden Buskonflikte erzeugen (sollte nicht verwendet werden)
8	1	1	1	Normal Expanded Wide Mode, [1]

[1] BDM ist erlaubt, muss aber durch ein BDM-Kommando aktiviert werden.

Tabelle X.X: Betriebsarten der 68HC12/HCS12-Mikrocontroller

Der BDM-Mode (Background Debug Mode) ist in allen Betriebsarten verfügbar. Er muss aber in allen Betriebsarten, mit Ausnahme des Special-Single-Chip-Modus durch ein BDM-Kommando aktiviert werden. TabelleX.X zeigt die verfügbaren Betriebsarten. Als Bezeichnung der Betriebsart wurde der englische Begriff beibehlten. Die 68HC12/HCS12-Mikrocontroller können als Einzelchiplösung arbeiten, d. h. es wird nur der interne Speicher verwendet. Diese Betriebart wird Single-Chip.-Modus genannt. Damit der Mikrocontroller externen Speicher ansprechen kann, muss die Betriebsart Expanded-Modus gewählt werden. Die Ports PA, PB und PE stehen dann als Daten- und Adressbus zur Verfügung. Der Expanded-Modus kann noch einmal in "narrow" und "wide" unterteilt werden, je nachdem man 8 Bit breite oder 16 Bit breite Speicher ansprechen will. Die Spezialbetriebsarten stehen für Testzwecke bei der Entwicklung oder für den Chiptest beim Hersteller Motorola zur Verfügung.

2.7.1 Die Grundbetriebsarten

Die Grundbetriebsarten werden verwendet, wenn der Test der Hard- und Software abgeschlossen ist. In diesen Betriebsarten steht zwar die BDM-Funktionalität zur Verfügung, sie muss aber über ein BDM-Kommando aktiviert werden. Es gibt drei

Grundbetriebsarten.

Normal Single-Chip Mode

In dem Normal-Single-Chip-Modus wird der Mikrocontroller als Einzelchip ohne externe Speicher verwendet. Alle Portpins der Ports PA, PB und PE stehen als Ein- und Ausgabepins zur Verfügung. In dieser Betriebsart kann das MODE-Register nur einmal beschrieben werden und ist nach einmaligem Beschreiben für jeden weiteren Schreibvorgang gesperrt. Dadurch kann man per Software einmalig am Anfang die Konfiguration festlegen. Diese kann man danach nicht mehr gewollt oder auch ungewollt verändern. Die Bustaktfrequenz kann über den Pin ECLK ausgegeben werden, um z. B. externe Bausteine mit einem konstanten Systemtakt zu versorgen.

Normal-Expanded-Modus

In dem Normal-Expanded-Wide-Modus werden die Ports PA und PB als 16 Bit breiter gemultiplexter Adress- und Datenbus konfiguriert. Das Port PE steht für Kontrollsignale zur Verfügung. Im Normal-Expanded-Narrow-Modus wird an Stelle von 16 Bit breiten Speicherbausteinen die preisgünstigeren 8 Bit breiten Speicher verwendet. Der Preisvorteil wird natürlich mit einem zusätzlichen Buszyklus erkauft, da der Mikrocontroller zweimal zugreifen muss, um 16-Bit zu lesen oder zu schreiben. Die Ports PA und PB sind als 16-Bit Adress- und Datenbus konfiguriert, wobei Port PA mit den Daten gemultiplext wird. Port PE steht wieder für die Kontrollsignale zur Verfügung.

Zu beiden Expanded-Modi gibt es eine Emulationsbetriebsart, die während der Entwicklung mit einem Logikanalysator verwendet werden kann. Alle für die Buskontrolle notwendigen Signale werden an den dafür vorgesehenen Pins ausgegeben.

2.7.2 Die Testbetriebsarten

Es stehen drei Testbetriebsarten zur Verfügung, die jeweils in den Grundbetriebsarten eine Entsprechung finden: Special-Single-Chip-Modus, Speical-Test-Modus und Peripheral-Modus. Man verwendet sie üblicherweise während der Entwicklungsphase oder zum Testen des Mikrocontrollers durch den Hersteller. Zum Unterschied zu den Grundbetriebsarten, ist bei den Testbetriebsarten die BDM-Schnittstelle nach dem Reset automatisch aktiviert. Der Mikrocontroller liest nicht den Reset-Vektor und startet damit auch nicht das Applikationsprogramm. An Stelle dessen übernimmt die BDM-Firmware, die aus einem speziellen vom Hersteller Motorola festgelegten Programm besteht, die Kontrolle über das Programm und wartet auf weitere BDM-Kommandos.

Im Peripheral-Modus wird die CPU12 deaktiviert und ein externer Master übernimmt die Kontrolle über die Peripheriebausteine. Diese Betriebsart wird normalerweise von einem Schaltkreistester zum Testen der Peripheriebausteine bzw. zum Testen der Schaltung verwendet. Man kann natürlich diese Betriebsart auch dazu verwenden, den Mikrocontroller als Multiperipheriebaustein zu verwenden. Eine andere CPU greift dann auf die Ressourcen des Mikrocontrollers zu.

2.8 Literaturverzeichnis und Web-Adressen

[1] STAR12 V1.5 Core, User Guide, Version 1.2, Document Number: S12CPU15UG/D, Motorola, www.motorola.com/semiconductors

[2] Gordon Doughman, Programming the Motorola M68HC12 Family, Annabooks 2000

[3] Thomas Beierlein, Olaf Hagenbruch, Taschenbuch Mikroprozessortechnik, Fachbuchverlag Leipzig/Hanser, 2. Auflage 2001

[4] Eine Demolizenz für den CodeWarrior kann über license_europe@metrowerks.com bezogen werden. Weitere Informationen zum CodeWarrior findet man bei www.metrowerks.com.