Ein Simulationsmodell für den PIC16C84-Microcontroller

Claas Rink

11. Februar 1999

Studienarbeit am Fachbereich Informatik, Arbeitsbereich TECH, der Universität Hamburg

Zusammenfassung

Die vorliegende Studienarbeit beschreibt die Konzeption und Implementation einer möglichst hardwarenahen Simulation eines Microcontrollers eingebettet in die Simulationsumgebung HADES. Als zu simulierender Baustein wurde der PIC16C84 von Microchip ausgewählt.

Das Ergebnis ist ein in Java verfaßtes HADES-Modell mit grafischer Oberfläche und vielfältigen Konfigurationsmöglichkeiten, das genau die zugrunde liegende Hardware abbildet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Ziel

Das Thema dieser Studienarbeit lautet „Ein Simulationsmodell für den PIC16C84-Microcontroller“ und gliedert sich in drei Phasen:

- In der ersten Phase findet die Einarbeitung in die zugrunde liegenden Themen Simulation, Mikroprozessorarchitektur, HADES und Microcontroller statt. Außerdem muß sowohl eine geeignete Architektur als auch ein spezieller Baustein als Hardware-Vorbild für die Simulation ausgewählt werden.

- Die zweite Phase stellt die Implementation der Simulationssoftware dar. Da die Simulation im Rahmen von HADES stattfindet, muß als Programmiersprache Java gewählt werden. Eine Betrachtung der unterschiedlichen zur Implementation geeigneten Sprachen mit anschließender Auswahl der geeignetesten kann also hier entfallen. Die Vorauswahl der Programmiersprache stellt aber kein Hindernis dar, da Java ein leistungsfähiges Hilfsmittel für übersichtliche, strukturierte und objektorientierte Programmierunung ist.

- Die dritte Phase dient zur Zusammenstellung und Dokumentation der Ergebnisse der ersten beiden Phasen. Während dieser Phase entsteht der hier vorliegende Text.

1.2 Simulation

1.2.1 Warum Simulation eines Microcontrollers?

Obwohl gerade Massenware wie einfache Microcontroller günstig in Herstellung und Implementation sind, ist eine Simulation dieser Bausteine trotzdem sinnvoll und notwendig. Microcontroller sind oftmals nur einmal programmierbar oder werden vom Hersteller mit ROM ausgestattet, so daß nur wirklich fehlerfreie Steuersoftware für diese Bausteine verwendet werden sollte. Zur Entwicklung von Anwendungen bietet sich also eine Simulationssoftware an, die den virtuellen Chip beliebig oft wiederbeschreiben kann. Zusätzlich kann eine solche Simulation, wenn sie möglichst nah am Vorbild der Hardware implementiert ist, interne Vorgänge des Chips offenlegen und auf diesem Wege Problemfälle vermeiden helfen.

Für universitäre Zwecke ist eine Simulation besonders geeignet, da sie hier einfach und kostengünstig (gerade in Hinblick auf leere Kassen für Forschung und Lehre) reale Vorgänge verstehen und nachbilden hilft.

1.2.2 Abstraktion

Leider hat eine Simulation auch Nachteile, weshalb sich ein realer Vorgang auch nach intensivsten Simulationsabläufen anders verhalten kann als dessen simuliertes Modell. Eine Simulation stellt immer eine Abstraktion, also eine Vereinfachung der zugrundeliegenden realen Vorgänge dar. Mit möglichst wenig Beschreibungs-, Implementations- oder Betriebsmittelaufwand wird versucht, ein komplexes Verhalten möglichst vollständig abzubilden. Es gibt je nach Anspruch an die Wirklichkeitstreue verschiedene Möglichkeiten der Implementation. Dabei müssen allerdings auch immer Kompromisse zwischen Implementations-, Ausführungsaufwand und Realitätsnähe eingegangen werden.

Aus diesem Grund sind in der Praxis viele Simulationen nicht in der Lage, verschiedene Abstraktionsebenen nebeneinander gleichzeitig zu simulieren und so Zusammenhänge zwischen diesen Abstraktionsebenen zu verdeutlichen.

1.3 HADES

1.3.1 Was ist HADES?

Die Abkürzung HADES steht für „Hamburger Design System“ und ist eine in Java programmierte Simulationsumgebung, die am Fachbereich Informatik der Universität Hamburg entstanden ist. Das Ziel bei der HADES-Entwicklung war es, ein einerseits möglichst einfaches und andererseits leicht erweiterbares und damit potentiell mächtiges Simulationssystem hauptsächlich für Lehrzwecke zu erstellen. Und schon die zur Entstehungszeit dieses Textes verfügbare Version 0.60a bietet eine stabile, leicht zu erlernende und zu bedienende Software, die auch professionellen Ansprüchen genügt.

Die Erweiterbarkeit von HADES ist dessen eigentliche Stärke, denn neben normalen wie bei ähnlichen Simulationen verfügbaren Modellen für einfache elektronische Bauteile ist auch die Verwendung von vollkommen anderen Datentypen wie z.B. Bildern und als deren „Bauteile“ dann visuelle Filter möglich.

Die Implementation neuer Bauteile oder Datentypen gestaltet sich dabei denkbar einfach, da diese wie HADES selbst in der Programmiersprache Java verfaßt und über eine übersichtliche Schnittstelle mit HADES verbunden werden (diese Schnittstelle wird in Abschnitt 3.5.2, Seite 47, beschrieben).

Für HADES existieren inzwischen viele Modelle für weniger komplizierte elektronische Bauteile, die sich zu beliebigen Schaltungen zusammenfassen und abstrahieren lassen. Für wirklich komplexe Schaltungen lohnt sich allerdings der Entwurf eines eigenen Modells, da hier alle Freiheiten normaler Java-Programme inklusiver eigener Benutzeroberflächen zur Konfiguration und Bedienung zur Verfügung stehen.

Aufgrund seiner einfachen Erweiterbarkeit ist es HADES möglich, mehrere Abstraktionsebenen gleichzeitig zu simulieren. Die in HADES eingebundenen Modelle unterliegen dabei keinen Einschränkungen, so daß sogar Hardware und Software gleichzeitig simuliert werden können.

Eine einfache Anleitung zu HADES ist in [Hendrich 97] zu finden.

1.3.2 Ein Microcontroller-Modell für HADES

Die bisher existierenden Modelle für HADES sind hauptsächlich einfache logische Bauteile, 74er-Logik und einzelne komplexere Komponenten der Register-Transfer-Ebene. Die in dieser Arbeit im Vordergrund stehende Simulation eines Microcontrollers ist aus der Not heraus entstanden, HADES mit komplexeren Modellen zu versehen, die als programmierbare Einheiten eigenständige Steueraufgaben übernehmen können. Mit Sicherheit werden in Zukunft weitere programmierbare Modelle für HADES folgen. Hierbei bieten sich natürlich insbesondere Modelle von im Rahmen der Lehre entstehenden Prozessoren wie denen des Mikrorechnerprojekts (URL: `http://tech-www.informatik.uni-hamburg.de/-Lehre/mikrorechner/projekt.html`) an.

1.4 Auswahl des zu simulierenden Bausteins

1.4.1 Anforderungen

Bei der Auswahl des zu simulierenden Bausteins gibt es verschiedene Anforderungen:

- Der Baustein darf nicht zu komplex aufgebaut sein, damit die Simulation mit einem vertretbaren Aufwand entwickelt werden kann.
- Ausreichende Dokumentation – zumindest der Hauptblockebene – ist zwingend. Diese Dokumentation muß einfach zu erhalten sein.
- Damit simulierte Designs auch in die Realität umgesetzt werden können, muß der simulierte Microcontroller kostengünstig in Anschaffung und Betrieb sein.
- Einmal programmierbare Bausteine sind für Lehrzwecke ungeeignet, also muß der Programmspeicher des Bausteins vielfach beschreibbar sein.
- Die Handhabung des Bausteins darf nicht zu kompliziert sein: Die zur Inbetriebnahme notwendige externe Logik sollte minimal sein, auch sollte sich der Programmspeicher möglichst einfach füllen lassen.
- Außerdem muß der Baustein genügend Leistung und Speicherplatz bieten.

Manche dieser Anforderungen stehen in direktem Gegensatz zueinander, deshalb muß bei der Auswahl ein ausgewogener Kompromiß gefunden werden.

1.4.2 Ergebnis des Auswahlprozesses: Microchip PIC16C84

Unter den großen Anbietern von Microcontrollern befinden sich drei, deren Bausteine in die nähere Auswahl gekommen sind: Intel, Motorola und Microchip. Alle drei decken das volle Leistungsspektrum möglicher Microcontroller ab. Zuerst muß also eine Auswahl getroffen werden, welche Leistungsfähigkeit der ausgewählte Baustein besitzen soll. Hier bieten sich die 8-Bit-Bausteine an, deren Befehlssatz einigermaßen übersichtlich aufgebaut ist. Diese Bausteine sind weit verbreitet, gut dokumentiert und einfach und billig zu beziehen.

Von Intel gibt es als Vertreter der 8-Bit-Microcontroller die weitverbreitete Familie 8051. Eine Übersicht der Architektur-Basisdaten dieser Baureihe:

- 8-Bit-Daten- und bis zu 16-Bit-Adreßbus, über den auch externer Speicher angesteuert werden kann.
- Jeweils getrennte Adreßräume für Programmspeicher, Datenspeicher und Spezialregister.
- Bidirektionale IO-Ports ohne Datenrichtungsregister
- Teile des Datenspeichers als Register zugreifbar
- Sehr komplizierter Aufbau und Befehlssatz, Instruktionen umfassen minimal ein und maximal drei Bytes.

Als Konkurrenz zur 8051-Reihe von Intel gibt es von Motorola die Baureihe 6805. Diese Bausteine mit 8 Bit Datenbreite haben folgende Eckdaten:

- Gemeinsamer Adreßraum für Programmspeicher, Datenspeicher und Spezialregister
- Breite des Datenbusses liegt bei 8 Bit, der Adreßbus ist je nach Modell 11-13 Bit breit, also maximal 8 KB Adreßraum möglich.
- Die Befehle umfassen zwischen einem und drei Bytes, der Befehlssatz ist mittelmäßig kompliziert.

Alternative Microcontroller von Microchip sind die 8-Bit-Baureihen PIC16C5x, PIC16Cxx und PIC16Cxxx.

- Harvard-Architektur, also getrennter Programm- und Datenspeicher. Dadurch höhere Geschwindigkeit dank parallel möglichem Zugriff auf beide Speicher.
- Klar strukturierter und übersichtlicher Befehlssatz dank breitem Programmspeicher (12, 14 oder sogar 16 Bit, je nach Baureihe)
- Interner Aufbau ist RISC-ähnlich (jeder Befehl besteht aus genau einem Wort), also verhältnismäßig einfach strukturiert und simulierbar.
- Detaillierte Dokumentation aller Chips ist per WWW verfügbar.

Insbesondere die klare Struktur des Befehlssatzes und der relativ übersichtliche Aufbau ließen die Entscheidung auf die Bausteine von Microchip fallen. Als Kompromiß zwischen Einfachheit und Leistungsfähigkeit wurde hier die mittlere Baureihe (PIC16Cxx) gewählt. Diese bietet eine weite Auswahl an Bausteinen mit unterschiedlicher Peripherieausstattung. Die kleine Baureihe (PIC16C5x) ist schlechter ausgestattet und besitzt einige Einschränkungen, die die mittlere Baureihe nicht besitzt. Ein Mitglied der großen Baureihe (PIC16Cxxx) ist für den Rahmen einer Studienarbeit zu komplex aufgebaut.

Viele der Bausteine der mittleren Baureihe scheiden für die Simulation aus, da sie einen nur einmal beschreibbaren Programmspeicher besitzen. Auch die Modelle mit EPROM-Programmspeicher sind für die gewünschte Aufgabe zu unhandlich, da die Handhabung für eine Neuprogrammierung verhältnismäßig umständlich ist. Übrig bleiben die Modelle mit EEPROM-Speicher, hierbei wiederum die, die sich auch in-circuit programmieren lassen. Von diesen ist der geeignetste der PIC16C84, der die Basisperipherie mitbringt (IO-Ports, Timer, Watchdog-Timer, Interrupts), aber sonst keine weiteren aufwendig zu modellierenden Spezialitäten wie z.B. serielle Schnittstellen oder A/D-Wandler enthält.

Zur Auswahl des zu simulierenden Bausteins wurde als Gegenüberstellung von Motorola- und Intel-Microcontrollern [Esders 87] zu Rate gezogen. Die Beschreibung der mittleren Microchip-Baureihe ist in [Microchip 97-1] zu finden.

1.5 Das Ergebnis: Die Simulation Pic16C84

Das Ergebnis dieser Studienarbeit ist ein Satz von Java-Klassen, die ein HADES-Modell zur Simulation des PIC16C84 zur Verfügung stellen. Dabei werden alle Eigenschaften des PIC16C84 vollständig simuliert (bis auf das Verhalten auf elektrischer Ebene), so daß die Simulation alle auf dem PIC16C84 lauffähigen Maschinencode-Programme ausführen kann. Es stehen also alle Komponenten des PIC16C84 wie I/O-Ports, Interrupts, Timer, Watchdog-Timer, Daten-EEPROM und die komplette Kernfunktionalität zur Verfügung.

Die Anbindung an HADES erlaubt eine Benutzung der Simulation als ganz normales Bauteil-Modell, so daß mehrere Instanzen der PIC16C84-Simulation gleichzeitig und auch in Subdesigns verwendet werden können.

Das Modell besitzt eine grafische Oberfläche, die die Inspektion und Modifikation der Interna des Chips ermöglicht. Dabei wurde besonderen Wert auf die Darstellung aller für die Fehlersuche im Modell und auch im darauf simulierten Maschinencode-Programm wichtigen Parameter gelegt. Es ist möglich, Breakpoints abhängig von sowohl angesprungenen Programmspeicheradressen als auch bestimmten Registerinhalten zu setzen. Außerdem wird ein Signaldiagramm zur Verfügung gestellt, mit dem die wichtigsten internen Signale dargestellt werden und das durch einfache Programmiereingriffe auch an persönliche Wünsche angepaßt werden kann.

Das folgende Kapitel beschreibt genau den Aufbau des PIC16C84, damit dann im Kapitel Kapitel 3 eingehend die Softwarearchitektur der Simulation beschrieben werden kann. In diesem Kapitel ist dann auch die Bedienungsanleitung der Oberfläche enthalten.

2 Beschreibung PIC16C84

Der PIC16C84-Microcontroller (im folgenden PIC genannt) gehört zur Familie der mittleren Baureihe der Microchip-Microcontroller. Im folgenden wird die Architektur dieser Baureihe genauer beschrieben und es wird auf die Spezialitäten des PIC eingegangen. Diese Beschreibung basiert auf [Microchip 97-2], der Referenzdokumentation dieses Bausteins.

Zuerst wird ein Überblick über den grundsätzlichen Aufbau gegeben. Danach folgen genauere Angaben zur Speicher-/Register-Aufteilung, die Maschinensprache wird beschrieben, alle wichtigen „Peripherie"- Elemente und die Pin-Belegung werden vorgestellt.

2.1 Überblick Hardware-Architektur

Die mittlere Baureihe der Microchip-Microcontroller stellt einen Kompromiß aus Leistung und Komplexität (also Kosten) dar. Alle internen Operationen werden 8-bittig durchgeführt und es gibt keine komplexeren mathematischen Funktionen als Integer-Additionen und -Subtraktionen. Der Befehlssatz umfaßt nur die nötigsten Befehle und Adressierungsarten, die Programmierung in Maschinencode erfordert demnach einigen Einarbeitungsaufwand – der Befehlssatz ist zwar dank des kleinen Befehlssatzes schnell zu erlernen, wirklich effiziente Programme sind aber kompliziert zu erstellen.

Die Befehlsworte des Maschinencodes sind immer 14 Bit breit ausgelegt und bieten damit im Gegensatz zu anderen 8-bittigen Microcontrollern gleiche Ausführungszeiten für (fast, s.u.) alle Befehle und leichte Dekodierbarkeit der Befehle. Außerdem ist dank des orthogonalen Befehlssatzes auch eine direkte Programmierung im Maschinencode möglich. Hier macht sich die RISC-ähnliche Konzeption des PIC bemerkbar.

Der PIC-Baustein enthält neben den für eine CPU normalen Elementen wie ALU, Registerbank, Befehlsdekodierwerk und Pipeline auch die für Microcontroller typischen Bauteile wie Programmspeicher, Multifunktions-IO-Ports, verschiedene Timer und einen nicht flüchtigen Datenspeicher (EEPROM). Alle Spezialfunktionen werden dabei über spezielle Register, die ganz normal in die Registerbank eingeblendet sind, gesteuert. Es gibt also keine speziellen IO-Befehle, der PIC-Baustein benutzt Memory-Mapped-IO.

Die CPU benutzt für Programmspeicher und Registerbank jeweils einen eigenen Bus, es liegt also eine Harvard-Architektur vor. Dank des aus diesem Grund unabhängig voneinander möglichen Zugriffs auf Befehle und Daten kann dieser Zugriff parallel erfolgen. So kann von einem einfachen Pipelining gesprochen werden: Die erste Stufe der Pipeline holt den Befehl aus dem Programmspeicher, die zweite Stufe dekodiert den Befehl und führt ihn aus.

Da die Befehlsbreite auf 14 Bit beschränkt ist und möglichst viele Register innerhalb einer Registerbank zu finden sein sollen, sind die Maschinenbefehle Ein-Adreß-Befehle. Für arithmetische Befehle mit zwei Operanden ist also ein Arbeitsregister (in der PIC-Terminologie wird dieses das Work-Register genannt, im folgenden wird das hier übernommen) notwendig, das den zweiten Operanden beisteuert und eventuell das Ergebnis aufnimmt. Auch für den Transport eines Wertes von einem Register in ein anderes ist der Umweg über das Work-Register zwingend. Für alle Register-Befehle kann ausgewählt werden, ob das Ergebnis der Operation im Work-Register oder im angegeben Register gespeichert wird.

Eine Übersicht der Architektur ist in Abbildung 1 auf Seite 14 zu finden.

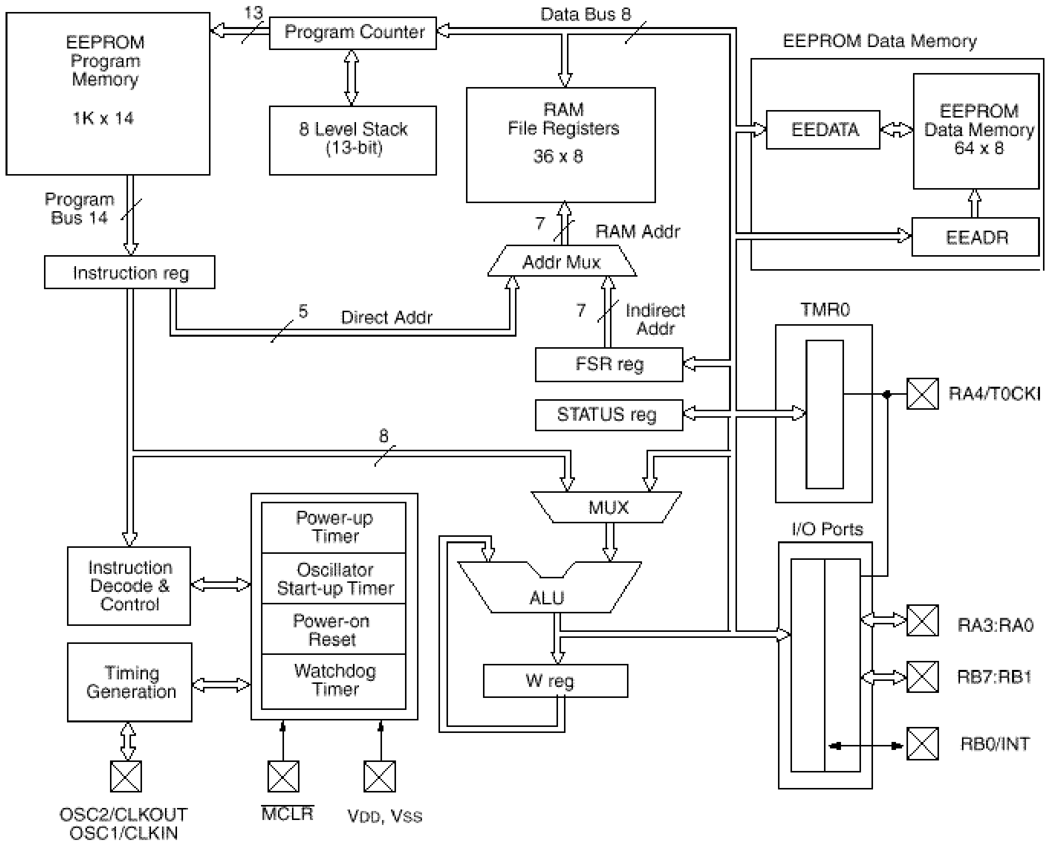

Abbildung 1: Hardware-Architektur des PIC16C84

2.2 „Peripherie"-Elemente

2.2.1 IO-Ports

Der PIC hat zwei IO-Register, eines mit 5 Bit Breite (Port A) und eines mit 8 Bit Breite (Port B). Diese Register sind normal in die Registerbank eingeblendet und können wie alle anderen gelesen und beschrieben werden. Für jedes Bit dieser PORT-Register existiert ein IO-Pin, dessen Zustand durch Beschreiben des entsprechenden Bits im PORT-Register beeinflußt werden kann. Auch kann der Zustand des IO-Pins durch Auslesen des entsprechenden Bits im dazugehörigen Register abgefragt werden. Für beide Ports existiert aus diesem Grund je ein Datenrichtungsregister (TRIS-Register), in dem für jedes einzelne Bit festgelegt werden kann, ob es als Eingang oder als Ausgang arbeitet.

2.2.2 Interrupts

Es gibt vier mögliche Interruptquellen:

- Überlauf von Timer-0
- Das Ende eines Schreibvorgangs im EEPROM

- Entweder ein Schaltvorgang zur 0 oder zur 1 (selektierbar) an Port B, Bit 0
- Irgendein Schaltvorgang an Port B, Bits 4 bis 7

Zur Selektierung der verschiedenen möglichen Interruptquellen steht im INTCON-Register für jede Interruptquelle je ein Bit zur Aktivierung der Quelle und ein Bit zur Abfrage, ob diese Quelle ein Interrupt ausgelöst hat, zur Verfügung.

2.2.3 Timer-0

Der Timer-0 ist ein normaler 8-Bit-Zähler. Normalerweise wird der Timer-0 bei jedem vierten Taktsignal hochgezählt. Er arbeitet also mit derselben Geschwindigkeit wie der Befehlstakt. Statt dessen kann der Timer-0 auch durch ein externes Signal Zählimpulse bekommen. Hierfür ist dann Bit 4 von Port A zuständig.

Da der Timer wie alle Register nur 8 Bit breit ist, gibt es noch die Möglichkeit, dem Timer-0 einen sogenannten Prescaler vorzuschalten, der die Anzahl der Zählimpulse in konfigurierbaren Verhältnissen teilt. Wenn der Prescaler also auf ein Teilungsverhältnis von 1:64 eingestellt ist wird der Timer nur bei jedem vierundsechzigsten Zählimpuls erhöht. Der Prescaler steht dem Timer-0 allerdings nicht uneingeschränkt zur Verfügung. Watchdog-Timer und Timer-0 teilen sich den Prescaler. Dieser kann also immer nur einem der beiden Timer zugeordnet sein. Wenn der Prescaler dem Timer-0 zugeordnet ist, liegt sein maximales Teilungsverhältnis bei 1:256. Der Timer arbeitet in diesem Fall wie ein 16-bittiger Timer, bei dem nur auf die obersten 8 Bit zugegriffen werden kann.

Ist der Timer-0 als Interruptquelle ausgewählt, dann wird beim Übergang von 255 auf 0 der Interrupt ausgelöst. Der Timer-0 kann aber auch ohne Interrupts benutzt werden, da der Wert des Timers jederzeit über ein in die Registerbank eingeblendetes Register ausgelesen und geändert werden kann.

Für die Konfiguration von Timer-0 ist das OPTION-Register zuständig. Hier kann die Zählimpulsquelle und das Teilungsverhältnis des Prescalers eingestellt werden.

Andere Bausteine der mittleren Baureihe haben eventuell noch weitere Timer, deshalb die Bezeichnung Timer-0. Der PIC besitzt (bis auf den folgend beschriebenen Watchdog-Timer) nur den Timer-0.

2.2.4 Watchdog-Timer

Zusätzlich zum normalen Timer-0 enthält der PIC noch einen 8 Bit breiten Watchdog-Timer (WDT). Dieser hat zwei verschiedene Aufgaben:

- Befindet sich der PIC gerade im SLEEP-Modus (wurde er also durch den Befehl SLEEP angehalten) und läuft der WDT über, dann wird der PIC wieder aufgeweckt und die Befehlsausführung nahtlos hinter dem SLEEP-Befehl fortgesetzt. Ein Bit im STATUS-Register gibt dabei an, daß der PIC durch den WDT aufgeweckt wurde.
- Wenn sich der PIC nicht im SLEEP-Modus befindet und ein Überlauf des WDT auftritt wird ein Reset ausgelöst: Bestimmte Spezial-Register werden auf Default-Werte zurück- und die Prorammausführung an Adresse 0 fortgesetzt. Auch in diesem Fall wird ein bestimmtes Bit im STATUS-Register gesetzt, so daß nach Auftreten eines WDT-Resets darauf reagiert werden kann.

Da der WDT wieder ein normales 8-Bit-Register darstellt (das allerdings nicht in die Registerbank eingeblendet ist) und die Zählimpulse intern ungefähr alle 0,07 ms ausgelöst werden (dieser Wert variiert je nach Versorgungsspannung und Temperatur), liefert der WDT ca. alle 18 ms einen Überlauf. Diese Zeitspanne ist für viele Aufgaben zu kurz, deshalb kann der Prescaler des Timer-0 auch dem WDT zugewiesen werden. In diesem Fall ermöglicht der Prescaler Teilungsverhältnisse bis zu 1:128.

Die Konfiguration des Prescalers erfolgt über das OPTION-Register. Zur Laufzeit eines Programms kann der WDT nicht an- oder abgeschaltet werden. Dieses ist nur durch Programmierung von Bit 2 in Speicherstelle 2007 (hex) des Programmspeichers möglich.

Der WDT kann, da er nicht in die Registerbank eingeblendet ist, nicht direkt beeinflußt werden. Ein Maschinenprogramm kann nur über den Befehl CLRWDT Einfluß auf den WDT nehmen und seinen Zähler (und den Prescaler, falls dieser dem WDT zugewiesen ist) auf Null zurücksetzen.

2.2.5 Daten-EEPROM

Im PIC enthalten ist ein nichtflüchtiger Datenspeicher mit 64 Byte Inhalt. Über die Register EEADR, EEDATA, EECON1 und EECON2 kann wahlfrei auf alle Bytes zugegriffen werden, allerdings sind Schreibzugriffe ziemlich langsam im Verhältnis zu normalen Registerzugriffen. Aus diesem Grund finden sie asynchron statt.

Wenn ein Wert ins EEPROM geschrieben werden soll, dann wird die Zieladresse und der Wert in die Register EEADR und EEDATA geschrieben. Durch verschiedene Bits, die in EECON1 und EECON2 gesetzt werden, wird der Schreibvorgang ausgelöst. Das Ende wird (falls aktiviert) durch einen Interrupt signalisiert, kann aber auch über Status-Bits im EECON1 abgefragt werden. Ein Schreibzyklus dauert ungefähr 10 ms.

Das PIC-Handbuch empfiehlt nach einem Schreibvorgang ein erneutes Lesen des Wertes, um dessen Richtigkeit zu überprüfen. Ein Lesevorgang wird wie ein Schreibvorgang vorbereitet, nur werden ein paar andere Bits in den EECON1-/EECON2-Registern gesetzt. Das Lesen passiert synchron, es muß also nicht auf das Ergebnis des Lesevorgangs gewartet werden.

2.3 Speicheraufteilung

Der Speicher des PIC-Bausteins teilt sich in Programmspeicher (ROM) und Registerbank (RAM). Beide sind physikalisch und logisch vollkommen voneinander getrennt.

Hier werden nun Programmspeicher, Datenspeicher und alle wichtigen Spezialregister im einzelnen beschrieben.

2.3.1 Programmspeicher

Der Programmspeicher der mittleren Familie besitzt einen Adreßraum von 8192 14-Bit-Worten. Hiervon sind allerdings nur 2K gleichzeitig „sichtbar", da der Programmspeicher in Bänken zu je 2K organisiert ist und diese per Segmentbits umgeschaltet werden. Reset- und Interrupt-Vektor sind nur in der ersten Bank enthalten, die im Bedarfsfall automatisch aktiviert wird. Der PIC enthält nur 1K Programmspeicher, alle höheren Adressen werden automatisch auf bestehende abgebildet. Deshalb muß bei der Benutzung (und bei der Simulation) des PIC keine weitere Rücksicht auf das Bankswitching genommen werden.

Der Programmspeicher der mittleren Baureihe kann je nach Modell verschieden ausgeführt sein. Möglich sind ROM (also wird direkt bei der Hestellung des Chips sein Programm festgelegt), PROM (genau einmal programmierbar), EPROM (mehrmals programmierbar, löschen allerdings nur durch UV-Licht) und EEPROM (mehrmals programmier- und direkt löschbar). Der PIC ist mit EEPROM ausgestattet und bietet als zusätzlichen Vorteil die In-Circuit-Programmierung, der Chip kann also, wenn die Schaltung entsprechend aufgebaut ist, zur Programmierung in der Schaltung belassen werden.

2.3.2 Datenspeicher

Der Datenspeicher (also die Registerbank) ist ziemlich komplex aufgebaut, da sowohl normale Datenregister, interne Steuerungsregister (wie der Programmzähler) und „Peripherie"-Register enthalten sind. Alle diese Register werden auf die gleiche Art und Weise angesprochen und benutzt. Als einzige Einschränkung gibt es einige Register, die Schreibzugriffe ignorieren, also nur gelesen werden können. Auch die Register sind in vier Bänke aufgeteilt, von denen jede 128 Register enthalten kann. Allerdings sind einige wichtige Register in allen Bänken eingeblendet, so daß sich die Anzahl der maximal möglichen Register auf 416 beschränkt.

Der PIC enthält 36 normale Datenregister, 5 interne Steuerungsregister und 11 „Peripherie"-Register, die über zwei Bänke verteilt sind. Die Übersicht über die Register und deren Adressen sind in Tabelle 1 zu finden.

Adr	Bank 0: Name	Bank 1: Name	Adr
00h	INDF	INDF	80h
01h	TMR0	OPTION	81h
02h	PCL	PCL	82h
03h	STATUS	STATUS	83h
04h	FSR	FSR	84h
05h	PORTA	TRISA	85h
06h	PORTB	TRISB	86h
07h	frei	frei	87h
08h	EEDATA	EECON1	88h
09h	EEADR	EECON2	89h
0Ah	PCLATH	PCLATH	8Ah
0Bh	INTCON	INTCON	8Bh
0Ch ... 2Fh	36 normale Datenregister	Spiegelung von Bank 0	8Ch ... AFh
30h ... 7Fh	frei	frei	B0h ... FFh

Tabelle 1: Übersicht der Register

Die Spezialregister werden nun nachfolgend vorgestellt. Die Reihenfolge orientiert sich dabei nicht an den Adressen der Register, sondern an deren Wichtigkeit und den didaktischen Abhängigkeiten.

2.3.3 STATUS-Register

Das Statusregister enthält Bits zur Repräsentation des ALU-Status (Carry, Digit-Carry, Zero), Informationen über Aufweckmechanismen (Power-down und Time-out) und Bits zur Selektierung der Registerbank (RP0 und RP1 für direkte und IRP für indirekte Adressierung). Da beim PIC die Register allerdings nur auf zwei Registerbänke beschränkt sind, sind RP1 und IRP nicht weiter von Bedeutung, nur RP0 wird für die Selektierung der zwei verschiedenen Bänke benutzt.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
IRP	RP1	RP0	/TO	/PD	Z	DC	C
Bank-auswahl indirekt	Bank-auswahl direkt	Bank-auswahl direkt	Time-out	Power-down	Zero	Digit carry	Carry

Tabelle 2: STATUS-Register

2.3.4 PCL- und PCLATH-Register

Da für die 8K Programmspeicher insgesamt 13 Bit für die Adressierung benötigt werden, ist der Programmzähler auf zwei Register aufgeteilt. PCL enthält die unteren 8 Bit des Programmzählers und PCLATH die oberen fünf, von denen je nach Befehlsklasse allerdings auch weniger verwendet werden können. PCLATH bildet leider nicht direkt die oberen fünf Bit des PC ab, nur wenn ein neuer Wert in den PC geschrieben wird, wird PCLATH als Quelle für die oberen fünf (oder zwei bei Verwendung der Sprungbefehle) genommen.

2.3.5 FSR- und INDF-Register

Diese beiden Register ermöglichen indirekte Adressierung. Wird das FSR-Register mit einer Registeradresse gefüllt, kann über Zugriffe auf INDF der Inhalt des Registers, dessen Adresse im FSR steht, beeinflußt werden. Hierbei ist der direkte Zugriff auf 256 Register, also die ersten beiden Registerbänke möglich. Sollen die dritte und vierte Registerbank selektiert werden (was allerdings nur bei anderen Modellen der mittleren Baureihe Sinn macht), dann muß das IRP-Bit des STATUS-Registers gesetzt sein. Wird das FSR-Register mit der Adresse des INDF-Registers selbst gefüllt, dann werden nachfolgende Zugriffe auf INDF wie Zugriffe auf nicht implementierte Register behandelt: Schreibzugriffe werden ignoriert und Lesezugriffe liefern immer die 0 als Ergebnis.

2.3.6 INTCON-Register

Für alle internen und externen Interruptquellen kann hier angegeben werden, ob sie einen Interrupt auslösen können und welche Quelle den Interrupt ausgelöst hat. Die verschiedenen externen Quellen sind Bit 0 von Port B, eine Wertänderung der Bits 4 bis 7 und ein Überlauf des Timer-0 (wenn dieser seinen Zählimpuls von Port A/ Bit 4 bekommt). Interne Interruptquellen sind entweder Timer 0 (wenn dieser seinen Zählimpuls intern bekommt) oder das Ende eines EEPROM-Schreibvorgangs.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
Global Interrupt enable	Eeprom Interrupt enable	TMR0 Interrupt enable	RB0 Interrupt enable	RB Port change I. enable	TMR0 flag	RB0 flag	RB Port change flag

Tabelle 3: INTCON-Register

2.3.7 OPTION-Register

Im OPTION-Register wird der Prescaler vom Timer-0 oder vom Watchdog-Timer konfiguriert und die Flanke für den Interrupt auf Bit 0 von Port B und den externen Timer-0-Taktgeber (Bit 4 von Port A) gewählt.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
/RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
PORTB pull-up enable	Interrupt Flanken-auswahl	TMR0-Quellen-auswahl	TMR0-Flanken-auswahl	Prescale Zuwei-sung	Prescale Verhält-nis Bit 2	Prescale Verhält-nis Bit 1	Prescale Verhält-nis Bit 0

Tabelle 4: OPTION-Register

2.3.8 TMR0-Register

Das TMR0-Register enthält den Wert von Timer-0. Das Register ist ein ganz normales Schreib-/Leseregister, so daß Stand des Timers kann also jederzeit abgefragt werden kann. Wird allerdings ein Wert in das TMR0-Register geschrieben, wird der Timer drei Befehlstakte, also zwölf externe Takte, angehalten. Erst danach arbeitet er normal.

2.3.9 TRISA- und TRISB-Register

Diese beiden Register sind die Datenrichtungsregister für die Ports A und B. Hier wird für jedes einzelne Bit der Ports eingestellt, ob sie als Input (1) oder Output (0) funktionieren.

2.3.10 PORTA- und PORTB-Register

Jedes Bit der Port-Register repräsentiert einen binären IO-Pin des Chips, abhängig von TRISA/TRISB sind die Pins Eingänge oder Ausgänge.

2.3.11 EEADR-Register

Hier wird die Adresse hinterlegt, die für den nächsten Zugriff (sowohl lesend als auch schreibend) auf das EEPROM benutzt werden soll.

2.3.12 EEDATA-Register

Vor einem Schreibzugriff auf das EEPROM muß im EEDATA-Register der zu schreibende Wert angegeben werden. Nach einem Lesezugriff ist hier der gelesene Wert zu finden.

2.3.13 EECON1- und EECON2-Register

EECON1 kontrontrolliert die Zugriffe auf das EEPROM. Durch das Setzen bestimmter Bits werden Lese- und Schreibzugriffe initiiert. Das EECON2-Register muß vor einem Zugriff auf das EEPROM mit ziemlich magischen Werten in einer festgelegten Reihenfolge beschrieben werden. Hiermit werden also die Zugriffe vorbereitet.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	-	-	EEIF	WRERR	WREN	WR	RD
			Eeprom Interrupt flag	Write error	Write enable	Start write	Start read

Tabelle 5: EECON1-Register

2.4 Maschinensprache

2.4.1 Befehlsklassen

Der Befehlssatz der mittleren Familie besitzt vier verschiedene Befehlsklassen mit jeweils einheiltlichem Format. Die obersten beiden Bits entscheiden über die Befehlsklasse. Die restlichen 12 Bit werden im folgenden einzeln aufgeschlüsselt.

- Byte-orientierte Registerbefehle (00): Diese Befehle besitzen einen 4-bittigen Opcode. Ein Bit, das Destination-Bit, gibt an, ob das Ergebnis der Operation im Work-Register (d=0) oder im in den Befehl kodierten Register (d=1) abgelegt werden soll und sieben Bit dienen zur Auswahl des Registers. In diese Befehlsklasse fallen Transportbefehle, registerorientierte arithmetische oder logische Befehle und die Spezialbefehle.

- Bit-orientierte Registerbefehle (01): Zwei Bit beschreiben den Opcode, drei Bit das zu beeinflussende Bit und sieben Bit dienen wieder zur Auswahl des Registers. Mit diesen Befehlen läßt sich ein Bit setzen und löschen, außerdem kann ein Bit getestet und abhängig vom Inhalt die Ausführung des nächsten Befehls ausgelassen werden.

- Sprungbefehle (10): Ein Bit wählt aus, ob CALL (Rücksprungadresse wird auf den Stack geschrieben) oder GOTO (kein Rücksprung möglich) ausgeführt wird. 11 Bit geben die Zieladresse an (Die Bits 12 und 13, die für eine vollständige Adresse benötigt werden, werden dem PCLATH-Register entnommen).

- Absolutwertbefehle (11): Befehle dieser Gruppe bestehen aus vier Bit Opcode und acht Bit Konstante, die entweder direkt in das Work-Register geschrieben wird (RETLW und MOVLW) oder mit diesem arithmetisch/logisch verknüpft wird.

Die Übersicht über den kompletten Befehlssatz der mittleren Baureihe ist in Tabelle 6 auf Seite 21 zu finden. Der PIC unterstützt noch zwei weitere Befehle: TRIS und OPTION. Diese Befehle sind nur aus Gründen der Abwärtskompatibilität im Befehlssatz des PIC enthalten, sollten aber nicht verwendet werden.

Befehl	Operand	Beschreibung	Zyklen	Opcode
ADDWF	r,d	Addiert W und r	1	`00 0111 drrr rrrr`
ANDWF	r,d	AND W und r	1	`00 0101 drrr rrrr`
CLRF	r	Löscht r	1	`00 0001 1rrr rrrr`
CLRW		Löscht W	1	`00 0001 0xxx xxxx`
CLRWDT		Löscht Watchdog-Timer	1	`00 0000 0110 0100`
COMF	r,d	Komplement r	1	`00 1001 drrr rrrr`
DECF	r,d	Dekrement r	1	`00 0011 drrr rrrr`
DECFSZ	r,d	Dekrement r, Sprung bei 0	1 (2)	`00 1011 drrr rrrr`
INCF	r,d	Inkrement r	1	`00 1010 drrr rrrr`
INCFSZ	r,d	Inkrement r, Sprung bei 0	1 (2)	`00 1111 drrr rrrr`
IORWF	r,d	OR W und r	1	`00 0100 drrr rrrr`
MOVF	r,d	Transport r nach W oder r	1	`00 1000 drrr rrrr`
MOVWF	r	Transport W nach r	1	`00 0000 1rrr rrrr`
NOP		Tut nichts	1	`00 0000 0xx0 0000`
RETFIE		Rücksprung vom Interrupt	2	`00 0000 0000 1001`
RETURN		Rücksprung	2	`00 0000 0000 1000`
RLF	r,d	Rotiert links mit Carry	1	`00 1101 drrr rrrr`
RRF	r,d	Rotiert rechts mit Carry	1	`00 1100 drrr rrrr`
SLEEP		aktiviert Sleep-Modus	1	`00 0000 0110 0011`
SUBWF	r,d	Subtrahiert W von r	1	`00 0010 drrr rrrr`
SWAPF	r,d	Tauscht halbe Bytes von r	1	`00 1110 drrr rrrr`
XORWF	r,d	XOR W und r	1	`00 0110 drrr rrrr`
BCF	r,b	Löscht Bit b von r	1	`01 00bb bfff ffff`
BSF	r,b	Setzt Bit b in r	1	`01 01bb bfff ffff`
BTFSC	r,b	Testet b, Sprung bei gelöscht	1 (2)	`01 10bb bfff ffff`
BTFSS	r,b	Testet b, Sprung bei gesetzt	1 (2)	`01 11bb bfff ffff`
CALL	k	Ruft Unterroutine auf	2	`10 0kkk kkkk kkkk`
GOTO	k	Springt zu Adresse k	2	`10 1kkk kkkk kkkk`
ADDLW	k	Addiert Konstante und W	1	`11 111x kkkk kkkk`
ANDLW	k	AND Konst und W	1	`11 1001 kkkk kkkk`
IORLW	k	OR Konst und W	1	`11 1000 kkkk kkkk`
MOVLW	k	Transport Konst nach W	1	`11 00xx kkkk kkkk`
RETLW	k	Rücksprung mit Konst nach W	2	`11 01xx kkkk kkkk`
SUBLW	k	Subtrahiert Konst und W	1	`11 110x kkkk kkkk`
XORLW	k	XOR Konst und W	1	`11 1010 kkkk kkkk`

Tabelle 6: Befehlsübersicht

2.4.2 Einige einführende Programm-Beispiele

Als einfaches Beispiel soll hier die Multiplikation zweier in zwei verschiedenen Registern stehender Werte (op1, op2) dienen. Das Ergebnis (result) der Multiplikation wird in ein drittes Register geschrieben. Eventuelle Überläufe werden nicht beachtet, die Multiplikation wird durch eine Abfolge von einfachen Additionen erledigt.

```
  clrw                          ; Work-Register loeschen
  movf          op1, F          ; STATUS-Bits fuer op1 setzen
  btfsc         STATUS, Z       ; Test, ob op1 gleich null
  goto          endloop         ; Wenn ja, dann fertig
loop:
  addwf         op2, W          ; Zweiter Operand wird addiert
  decfsz        counter         ; Counter herunterzaehlen
  goto          loop            ; Sprung wenn Counter nicht gleich null
endloop:
  movwf         result          ; Ergebnis ins Result-Register
```

Ein weiteres Beispiel soll die Benutzung von indirekter Adressierung verdeutlichen. Hier sollen die Register 0x20 bis 0x2f gelöscht werden.

```
  bcf           STATUS, IRP     ; Die ersten beiden Registerbaenke
  movlw         0x20            ; Startadresse ins Work-Register
  movwf         FSR             ; FSR bekommt die Startadresse
next:
  clrf          INDF            ; Nun indirektes Loeschen
  incf          FSR, F          ; FSR weiterzaehlen
  btfss         FSR, 4          ; 16 Bytes erledigt?
  goto          next            ; Wenn nicht, dann Sprung
continue:                       ; Vorgang abgeschlossen
```

Im Anhang A.1 (Seite 51) sind weitere, komplexere Beispiele zur Benutzung des Befehlssatzes des PIC zu finden.

2.4.3 Programmierung der PIC-Eigenheiten

Die Benutzung der verschiedenen Komponenten des PIC ist in einigen Fällen etwas komplizierter als auf den ersten Blick ersichtlich. Die Behandlung dieser Fälle wird hier nun in einigen Programmbeispielen behandelt.

Soll der Aufruf einer Interrupt-Service-Routine vom eigentlich ausgeführten Programm nicht bemerkt werden, dann ist es notwendig, daß das Work- und das STATUS-Register nach Ausführung der Interrupt-Service-Routine wieder hergestellt werden. Da jedoch auch einfache MOVF-Befehle das STATUS-Register beeinflussen, muß zu diesem Zweck der Swap-Befehl benutzt werden. Die folgende Routine benötigt in jeder Bank das gleiche Register `work_temp` sowie in Bank 0 das Register `status_temp` zur Sicherung von Work- und STATUS-Register.

```
movwf  work_temp          ; Work-Register sichern
swapf  STATUS, W          ; STATUS- ins Work-Register swappen
bcf    STATUS, RP0        ; Bank 0 aktivieren
movwf  status_temp        ; und sichern

(... hier folgt die eigentliche Interrupt-Service-Routine)

swapf  status_temp, W     ; Gesicherten STATUS ins Work-Register
movwf  STATUS             ; und wieder ins STATUS-Register schreiben
swapf  work_temp, F       ; Gesichertes Work-Register swappen
swapf  work_temp, W       ; und nochmals ins Work-Register swappen
```

Wenn einer der Ports zur Kommunikation mit der Umwelt benutzt werden soll, muß dieser vorher initialisiert werden. Beispielhaft sollen nun die unteren vier Bits des Port B als Eingänge und die oberen vier Bits als Ausgänge benutzt werden. Die Ausgänge sollen von Beginn an (also ab dem Moment, ab dem diese Pins als Ausgänge benutzt werden) auf „1“ gesetzt sein.

```
clrf   STATUS             ; Bank 0 aktivieren
movlw  0xf0               ; Obere 4 Bits setzen
movwf  PORTB              ; und ins Port-Register schreiben
bsf    STATUS, RP0        ; Bank 1 aktivieren
movlw  0x0f               ; Datenrichtung festlegen
movwf  TRISB              ; und ins TRISB-Register schreiben
```

Die Benutzung des Timer0 und des WDT setzt einige Vorsicht voraus, damit nicht während der Initialisierung oder der Zuweisung des Prescalers ungewollt ein Überlauf oder – noch schlimmer – ein WDT-Reset ausgelöst wird. Soll die Zuordnung des Prescalers vom TMR0 zum WDT geändert werden, sind folgende Instruktionen notwendig:

```
bsf    STATUS, RP0        ; Bank 1
movlw  b'xx0x0xxx'        ; Alter Prescaler-Wert muss
movwf  OPTION_REG         ; angegeben werden
bcf    STATUS, RP0        ; Bank 0
clrf   TMR0               ; Timer0 und Prescaler zuruecksetzen
bsf    STATUS, RP1        ; Bank 1
movlw  b'xxxx1xxx'        ; Wie zweite Zeile, nur jetzt mit
movwf  OPTION_REG         ; Prescaler-Zuordnung zum WDT
clrwdt                    ; WDT und Prescaler zuruecksetzen
movlw  b'xxxx1xxx'        ; Nun wird der neue Prescaler-Wert
movwf  OPTION_REG         ; ins OPTION-Register geschrieben
bcf    STATUS, RP0        ; Bank 0
```

Soll die Benutzung des Prescalers vom WDT zum Timer0 wechseln, dann ist nicht ganz so viel Aufwand zu treiben:

```
clrwdt                  ; WDT und Prescaler zuruecksetzen
bsf    STATUS, RP0      ; Bank 1
movlw  b'xxxx0xxx'      ; TMR0 bekommt Prescaler, neuer
movwf  OPTION_REG       ; Prescaler-Wert kann angegeben werden
bcf    STATUS, RP0      ; Bank 0
```

2.5 Pipeline/Taktung

Die Befehlsbearbeitung ist in fünf Phasen unterteilt:

- Fetch: Der Befehl wird aus dem Programmspeicher geholt.
- Decode: Der Befehl wird dekodiert und die einzelnen Funktionseinheiten der CPU für den Befehl konfiguriert.
- Read: Für den Befehl benötigte Daten werden gelesen.
- Execute: Das Ergebnis des Befehls wird berechnet.
- Write: Das Ergebnis wird geschrieben.

Da Programmspeicher und Datenspeicher voneinander getrennt sind und über verschiedene Busse adressiert und ausgelesen werden, kann die Fetch-Phase für den nächsten Befehl parallel zu den anderen Phasen des aktuellen Befehls ausgeführt werden. Daraus ergibt sich eine zweistufige Pipeline.

Falls ein Sprung durchgeführt wird, muß der parallel schon gelesene nächste Befehl verworfen werden. Daraus folgt eine Befehlsausführungszeit von 4 Takten für normale Befehle und 8 Takten für durchgeführte Sprungbefehle (BTFSC, BTFSS, DECFSZ und INCFSZ benötigen also nur dann 8 Takte, wenn die Bedingung zutrifft).

Der Pin Osc1/Clkin ist der Takteingang für die einzelnen Phasen, am Pin Osc2/Clkout steht der Befehlstakt, also das Osc1-Signal viermal langsamer zur Verfügung. Da der PIC je nach Ausführung mit maximal 20 MHz Takt versorgt werden kann, sind 5 Millionen Befehle pro Sekunde möglich. Diese Rechenleistung genügt auch für komplexere Aufgaben bis in den Bereich der einfachen Echtzeit-Signalverbarbeitung.

Die Abfolge der verschiedenen Taktsignale und die daraus resultierende Befehlsbearbeitung ist in Abbildung 2 auf Seite 25 dargestellt.

2.6 Pin-Belegung

Die Bausteine der mittleren Baureihe sind in den verschiedensten Gehäuseformen mit unterschiedlichen Pin-Belegungen erhältlich (diese hängt natürlich von der auf dem Chip integrierten „Peripherie“ ab). Der PIC ist in den Gehäuseformen PDIP (Plastic Dual Inline Package) und SOIC (Small Outline IC) erhältlich. Es gibt Pins für die Versorgungsspannung, Reset, Takteingang, Taktausgang, Port A und Port B, also insgesamt 18 Pins.

Das Pin-Diagramm liegt in Abbildung 3, Seite 25, vor.

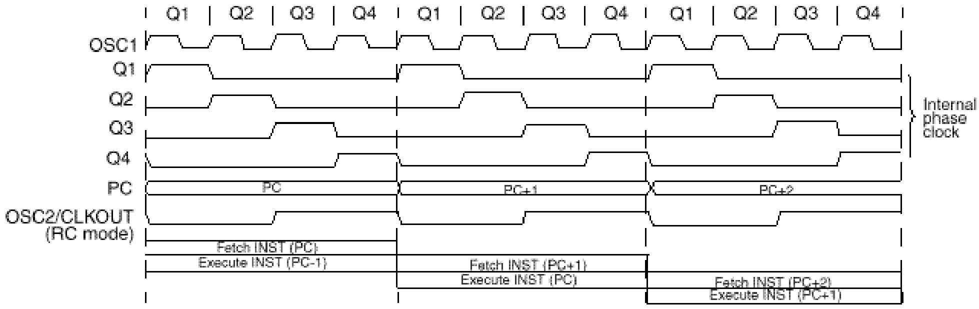

Abbildung 2: Pipeline und Taktung

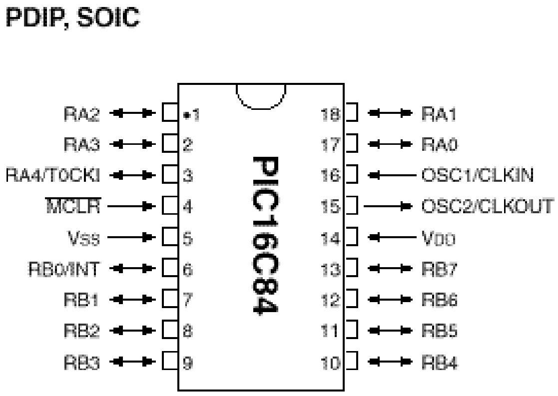

Abbildung 3: Pin-Belegung des PIC16C84

3 Software-Architektur

3.1 Einleitung

Die Simulation des PIC-Bausteins ist einerseits möglichst nah am Aufbau der Hardware orientiert (soweit anhand der Datenblätter auf ein Hardware-Design zurückgeschlossen werden kann) und andererseits möglichst übersichtlich programmiert. An einigen Stellen schließt sich die Befolgung beider Prinzipien leider aus, was dann einige Vereinfachungen der angenommenen Hardware und etwas kompliziertere Softwarestrukturen nach sich zieht.

Im folgenden wird zuerst ein grundlegender Überblick über die Software-Architektur gegeben. Danach werden die einzelnen Komponenten sowohl des simulierten PIC-Bausteins als auch der benutzten Visualisierungselemente beschrieben.

Die Simulation umfaßt nur den normalen Betrieb des PIC-Bausteins, nicht die Programmierphase des Programmspeichers. Der Programmspeicher kann durch externe Dateien (mit der Dateinamensendung „.hex“ oder „.rom“, siehe Abschnitt 3.3.16, Seite 35) initialisiert werden.

3.2 Dreiteilung Kern/Visualisierung/HADES-Anbindung

Die Simulation ist in drei Teile aufgeteilt:

- Der Kern enthält die eigentliche Simulation des Chips ohne IO-Funktionen. Hier ist das komplette Verhalten des PIC-Bausteins bis zu den Interrupt- und den Port-Registern abgebildet.
- Der nächste Teil ist die Visualisierung des Kerns. Hier werden alle wichtigen Elemente des Kerns angezeigt und können verändert werden.
- Der dritte Teil ist die HADES-Anbindung, hier werden die Register, die direkt auf die Pins Einfluß nehmen (und umgekehrt), mit den Werten aus der umgebenden Simulation gefüllt. Hier liegt also die Schnittstelle vor, die den Kern mit der HADES-Simulation verbindet.

Der Kern ist theoretisch sowohl alleine mit der HADES-Anbindung als auch alleine mit der Visualisierungskomponente lauffähig. Sollen nur interne Vorgänge des PIC-Bausteins betrachtet werden, reicht die Visualisierung zusammen mit dem Kern aus. Soll der Baustein innerhalb von HADES simuliert werden, würde zwar der Kern nur mit der HADES-Anbindung zusammen funktionieren, allerdings ist zur Bedienung des Kerns die Visualisierung zwingend notwendig (z.B. zum Füllen des Programmspeichers).

Einige Elemente des Kerns, die von externen Zeitsignalen abhängig sind (also der WDT und das EEPROM), sind nur im HADES-Umfeld lauffähig.

Die Abbildung 4 auf Seite 28 zeigt alle wichtigen an der Simulation beteiligten Klassen im Überblick.

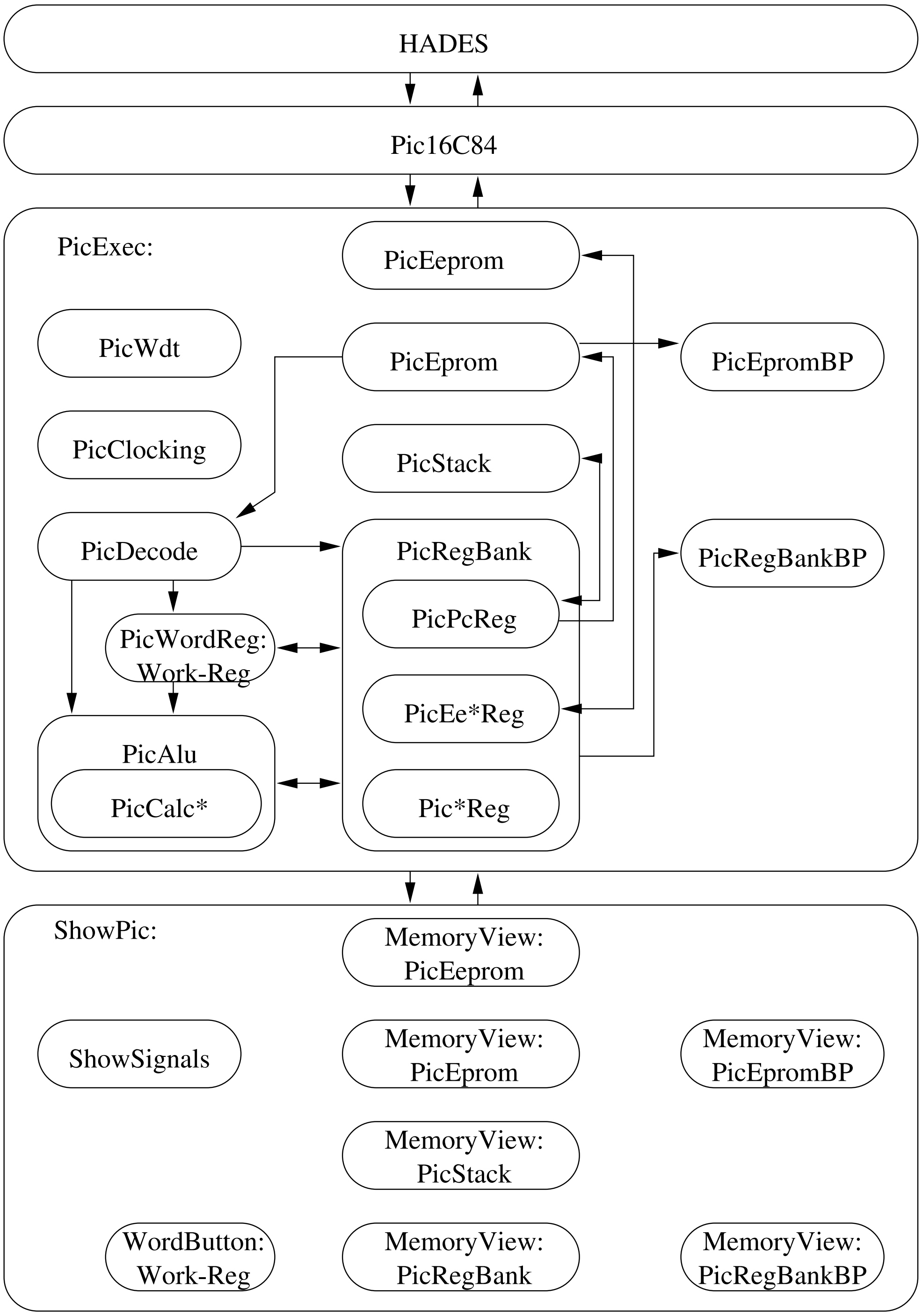

Abbildung 4: Die Software-Architektur

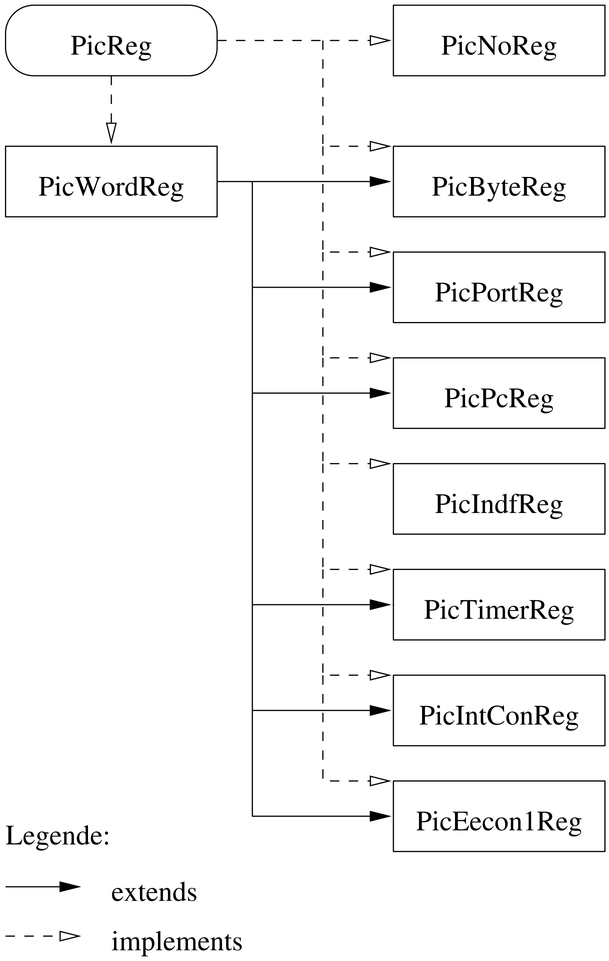

Abbildung 5: Die verschiedenen Registerklassen

3.3 Der Kern

3.3.1 Klassenüberblick

Als elemtarer Datentyp und somit als Grundlage dient die Klasse PicBitReg. Aufbauend auf diesem sind (fast) alle weiteren Register definiert, insbesondere das PicWordReg. Dieses wiederum dient als Grundlage für (fast) alle Spezialregister (PicByteReg, PicEecon1Reg, PicIntConReg, PicPcReg, PicPortReg, PicTimerReg). Alle Register haben die Schnittstelle PicReg gemeinsam. Eine Übersicht der Register ist in Abbildung 5 zu finden.

Die Zusammenfassung aller Register ist die Registerbank, PicRegBank (eigentlich ist in der Microchip-Terminologie ja immer vom mehreren Registerbänken die Rede, hier werden alle Register zu einer Registerbank zusammengefaßt). Diese bildet zusammen mit weiteren Komponenten wie dem Programmspeicher (PicEprom), der arithmetisch-logischen Einheit (ALU, PicAlu, basiert auf den verschiedenen PicCalc*-Klassen), der Taktung (PicClocking), dem Befehlsdekodierer (PicDecode), dem Stack (PicStack), dem EEPROM (PicEeprom) und dem Watchdog-Timer (PicWdt) die Ausführungseinheit (PicExec), also den Kern.

Nicht unmittelbar zum Kern selbst gehörig, aber zum Debuggen des Kerns unerläßlich

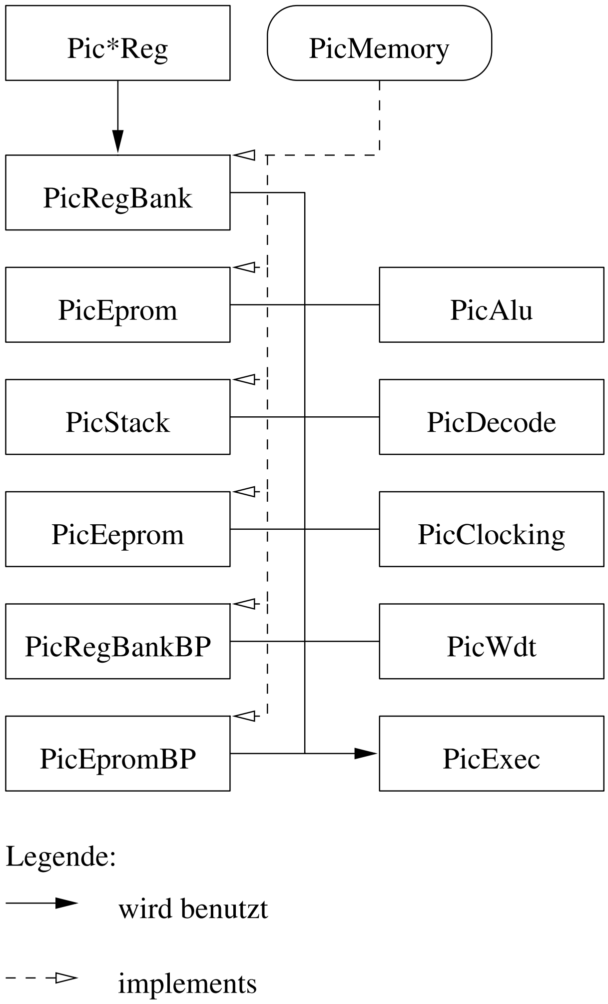

Abbildung 6: Alle Kern-Klassen

sind die Breakpoint-Speicher. Es gibt zwei, einen für die Register-Breakpoints (die Klasse nennt sich also PicRegBankBreakPoints) und einen für Programmspeicher-Breakpoints (PicEpromBreakPoints).

Die Klassen, die auf irgendeine Art und Weise Speicher darstellen (also PicRegBank, PicRegBankBreakPoints, PicEprom, PicEpromBreakPoints, PicEeprom und PicStack) basieren alle auf einem gemeinsamen Interface, dem PicMemory.

Die den Kern bildenden Klassen sind in Abbildung 6 so angeordnet, daß die interne Struktur erkennbar ist.

Allen am Kern beteiligten Klassen sind die beiden Methoden por() und reset() gemeinsam, mit denen die Objekte in den Zustand nach dem Einschalten bzw. in den Zustand nach einem Reset versetzt werden können.

Bei der Programmierung der verschiedenen Komponenten wurde versucht, die Implementation möglichst unabhängig sowohl von der Baureihe als auch vom verwendeten Chip zu halten. Es dürften also keine Probleme entstehen, wenn statt dem PIC16C84 ein anderer Baustein der mittleren Baureihe basierend auf den hier vorgestellten Klassen implementiert werden soll. Auch Simulationen der Bausteine der anderen Familien können basierend auf der hier vorgestellten Architektur programmiert werden, allerdings sind dann aufgrund wesentlich anderer Hardware-Voraussetzungen einige etwas größere Änderungen an vielen Klassen erforderlich.

Nun werden die Interfaces und Klassen des Kerns im einzelnen beschrieben.

3.3.2 PicReg

Als Interface für alle folgenden Register dient das PicReg. Da alle verschiedenartigen Register zur Registerbank zusammengefaßt werden, ist ein Interface mit einheitlichen Zugriffs- und Verwaltungsmethoden notwendig. Hier eine Übersicht der von diesem Interface zur Verfügung gestellten Methoden:

public int getSize(); Die Breite (also die Anzahl der Bits) des Registers kann mit getSize() festgestellt werden. Hier wird die Anzahl der Bits zurückgeliefert, auf die mit read() und write() zugegriffen wird.

public int getSizeAll(); Da es außer read() und write() noch die Methoden readAll() und writeAll() gibt, die eventuell einen Zugriff auf mehr Bits zulassen, gibt es zu diesem Zweck die Bitbreitenermittlung extra für die *All()-Methoden mit dem Namen getSizeAll(). Bei normalen Registern unterscheiden sich die Zugriffe zwischen normalen und *All()-Methoden nicht in der Bitbreite, nur bei wenigen Spezialregistern ist für diesen Zweck eine Spezialbehandlung implementiert.

public void por(); Jedes auf PicReg basierendes Register muß einen Power-on-reset zur Verfügung stellen. Hier wird also der Initialwert in das Register geschrieben.

public void reset(); Der normale Reset muß natürlich auch angeboten werden. Normale Register tun hier nichts.

public int read(); Mit read() kann der Inhalt des Registers ausgelesen werden.

public int readAll(); Einige Register bestehen aus einem allgemein zugreifbaren Bereich und einigen Spezial-Bits, die nur vom Steuerwerk, nicht aber von normalen Maschinenbefehlen ausgelesen werden können. readAll() ist die Methode, die sowohl die normalen mit read() auslesbaren als auch die besonderen Bits als Ergebnis liefert. Bei normalen Registern entspricht read() readAll(), bei speziellen Registern wie z. B. dem Programmcounter (PicPcReg) liefert das normale read() nur den Wert, der innerhalb eines Maschinencode-Programms ausgewertet werden kann (beim PC also 8 Bit), und readAll() im Gegensatz dazu den kompletten Inhalt des Registers (beim PC also 13 Bit).

public void write(int newValue); Das Register kann mit write() mit neuem Inhalt gefüllt werden.

public void writeAll(int newValue); Wie readAll() gibt es auch writeAll(), um wirklich alle Bits, also sowohl die normal zugreifbaren als auch die speziell vorhandenen, in das Register schreiben zu können.

public void setBit(int selectedBit, boolean newValue); Die Bits eines Registers können auch bitweise gesetzt werden, hier ist allerdings nur Zugriff auf die normal mit read() und write() beeinflußbaren Bits möglich.

public boolean getBit(int selectedBit); Entsprechend zum Setzen der Bits mit setBit() können diese auch mit getBit() wieder ausgelesen werden.

public void setBreakPoint(int value); Jedes einzelne Register kann mit einem Breakpoint versehen werden. Die Programmausführung oder sogar die komplette HADES-Simulation kann angehalten werden, sobald ein mit einem Breakpoint versehenes Register erreicht wird. Dabei können Breakpoints festgelegt werden, die nur dann aktiviert werden wenn ein ganz bestimmter Wert in ein Register geschrieben wird (dann muß beim Aufruf von setBreakPoint() dieser Wert übergeben werden). Es gibt auch die Möglichkeit, Breakpoints festzulegen, die auf jeden Schreibzugriff eines Registers reagieren (dann muß beim Aufruf von setBreakPoint() -2 übergeben werden). Falls ein Breakpoint gelöscht werden soll, bekommt setBreakPoint() die -1 übergeben.

public int getBreakPoint(); Der Breakpoint-Zustand eines Registers kann mit getBreakPoint() auch wieder ausgelesen werden. Diese Methode wird hauptsächlich zur Visualisierung benötigt.

3.3.3 PicBitReg

Ein Bit wird nicht etwa durch „false“ und „true“ dargestellt, sondern komplexer durch die neun verschiedenen Werte, die durch hades.models.StdLogic1164 definiert werden. Damit ist es möglich, alle Werte, die innerhalb von HADES auftreten können, auch in der PIC-Simulation zu berücksichtigen.

Ein PicBitReg nimmt genau einen StdLogic1164-Wert [Hendrich 97] auf und kann sowohl mit den logischen Werten „false“ und „true“ als auch mit den StdLogic1164-Konstanten beschrieben werden, je nachdem, ob das PicBitReg extern oder durch den Kern beschrieben wird.

3.3.4 PicWordReg

Ein Wort wird durch ein Array von PicBitReg (maximal 24 Bit) dargestellt und kann geschrieben und gelesen werden. Da das PicWordReg auf dem PicReg basiert, kann auf das PicWordReg sowohl wort- als auch bitweise zugegriffen werden. Das PicWordReg kennt keinen Unterschied zwischen den normalen und den *All()-Methoden.

3.3.5 PicByteReg

Ein PicByteReg ist ein vom PicWordReg abgeleitetes Register mit der fest eingestellten Registerbreite von 8 Bit. Dieser Registertyp wird für alle Register verwendet, die keine Spezialfunktionen erfüllen.

3.3.6 PicNoReg

Nicht implementierte Register lassen sich beim PIC-Baustein auslesen (als Ergebnis wird immer eine 0 geliefert) und beschreiben (Wert wird ignoriert). Das PicNoReg implementiert alle Methoden des PicReg entsprechend dieser Spezifikation. Alle Registeradressen, für die kein wirkliches Register zur Verfügung steht, sind mit dem PicNoReg realisiert. Da sich hinter dem PicNoReg kein wirkliches Register verbirgt, ist es nicht vom PicWordReg abgeleitet.

3.3.7 PicPortReg

Ein Port-Register kann auch von außen, also von der umgebenden HADES-Simulation beschrieben werden. Das funktioniert allerdings nur, wenn das zum Port gehörende Datenrichtungsregister (TRIS) einige Bits des Port-Registers als Eingang festlegt. Aus diesem Grund wird der Konstruktor des PicPortReg mit dem entsprechenden TRIS-Register aufgerufen und für die HADES-Anbindung steht die Methode setBitExtern() bereit, die nur die Bits setzt, die im TRIS-Register als Eingang ausgewiesen sind. Die übrige Funktionalität des PicPortReg ist vom PicWordReg abgeleitet.

3.3.8 PicPcReg

Der Programmzähler ist intern 13 Bit breit, im PCL-Register sind aber natürlich nur 8 Bit sichtbar. Das PicPcReg stellt einerseits das PCL-Register dar, kann aber über die speziellen *All()-Methoden auch den ganzen Programmzähler mit 13 Bit manipulieren. Dazu muß dem PicPcReg beim Konstruktoraufruf das PCLATH-Register mit übergeben werden. Das PicPcReg ist vom PicWordReg abgeleitet.

3.3.9 PicIndfReg

Das INDF-Register dient zur indirekten Adressierung und bietet Zugriff auf das Register, dessen Adresse im FSR-Register steht. Deshalb muß dem Konstruktor sowohl das FSR-Register als auch die gesamte Registerbank übergeben werden. Zusätzlich benötigt der Konstruktor noch das IRP-Bit des STATUS-Registers, damit alle 512 theoretischen Register adressierbar sind (in der Praxis spielt das keine Rolle, da beim PIC nur die ersten zwei Registerbänke genutzt werden und deshalb keine Registeradressen größer als 256 benötigt werden). Alle vom PicWordReg zur Verfügung gestellten Zugriffsmethoden werden beim PicIndfReg durch indirekte Varianten ersetzt. Dabei dient die interne Methode findSource() zur Ermittlung des zu beeinflussenden Registers.

Das PicIndfReg zeigt ein Spezialverhalten, wenn ins FSR-Register eine 0 geschrieben wird: Folgende Zugriffe auf das PicIndfReg verhalten sich so wie Zugriffe auf ein PicNoReg, ignorieren also Schreibaufträge und liefern bei Leseaufträgen eine 0.

Da das PicIndfReg selbst keinen Wert speichert, ist es auch nicht direkt vom PicWordReg abgeleitet, sondern implementiert selbst die Schnittstelle zum PicReg.

3.3.10 PicTimerReg

Eigentlich teilen sich der WDT und der Timer-0 den Prescaler, das läßt sich allerdings nur unpraktisch implementieren. Aus diesem Grund hat das PicTimerReg seinen eigenen Prescaler, der allerdings entsprechend den Bits im OPTION-Register normal funktioniert und bei Aktivierung des WDT-Prescalers abgeschaltet wird. Analog hat auch der WDT seinen eigenen Prescaler, der aber immer nur dann aktiviert wird, wenn der Timer-0-Prescaler nicht eingeschaltet ist, doch dazu später mehr.

Das TMR0-Register hat eine Besonderheit: Wenn ein Wert hinein geschrieben wird, dann wird der Prescaler zurückgesetzt und der Timer drei Takte angehalten, erst danach zählt der Timer seine Zählimpulse. Dieses Verhalten ist durch ein Lock-Flag nachgebildet, das erst nach drei Taktimpulsen den Timer freigibt.

Dann gibt es zwei Methoden, die den Timer hochzählen: internInc() wird in der dritten Befehlsausführungsphase aufgerufen und externInc() immer, wenn die richtige Flanke am Pin 4 von Port A erscheint. Je nach T0CS im OPTION-Register wird dann der Timer bzw. der Prescaler erhöht.

Das PicTimerReg ist, wie die meisten Register, vom PicWordReg abgeleitet und implementiert auf diesem Wege das PicReg-Interface.

3.3.11 PicIntConReg

Diese Klasse bietet Methoden, die entsprechend den Interruptquellen die richtigen Bits setzt und mit couldBeIntr() und isIntr() allgemein die Information zur Verfügung stellt, ob eine Interrupt-Ursache oder sogar ein Interrupt vorliegt und in die Interrupt-Routine verzweigt werden muß. Diese Methoden dienen eigentlich als Makros, ansonsten hat das PicIntConReg keine weitere Funktionalität, die über ein normales PicWordReg hinausgeht, von dem es abgeleitet wird.

3.3.12 PicEecon1Reg

In dieser Klasse wird aufbauend auf dem normalen PicWordReg bei jeder Schreiboperation darauf geachtet, ob sich die untersten beiden Bits geändert haben. Wenn Bit 0 auf 1 gesetzt wurde, liegt ein Lesezugriff auf das Daten-EEPROM vor, deshalb wird in diesem Fall eeprom.-readEeprom() (siehe Abschnitt 3.3.18, Seite 36) aufgerufen. Wenn Bit 1 auf 1 gesetzt wurde, wird ein Schreibzugriff initiiert, indem eeprom.writeEeprom() aufgerufen wird. Vor Benutzung des PicEecon1Reg muß diesem Register das Daten-EEPROM mit setEeprom() zur Verfügung gestellt werden.

3.3.13 PicMemory

Das zweite Interface, das ähnlich strukturierte Klassen des Kerns zusammenfaßt, ist das PicMemory-Interface. Es dient dem Zweck, alle Klassen, die wie ein Speicher aufgebaut sind, auf eine einheitliche Art und Weise für die Außenwelt (also in diesem Fall für die Visualisierung) darzustellen. Die Zugriffe auf Speicher über die PicMemory-Methoden finden flach, also nicht durch irgendwelche Bank-Switching-Register beeinflußt statt.

public int getWordWidth(); Da bei den verschiedenen Pic-Speichern unterschiedliche Wortbreiten vorliegen, kann diese bei jedem Speicher abgefragt werden.

public int getMemorySize(); Die Größe eines Pic-Speichers muß nicht einer Zweier-Potenz entsprechen, sie kann beliebig sein. Mit getMemorySize() kann sie abgefragt werden.

public int readMemory(int address); readMemory() liefert den Inhalt einer Speicheradresse.

public void writeMemory(int address, int word); Als Gegenstück zum readMemory() kann mit writeMemory() eine bestimmte Speicherstelle mit einem neuen Wert beschrieben werden. Dabei sind auch auf eigentlich schreibgeschützte Speicherstellen Schreibzugriffe möglich.

Für Zugriffe auf Bit-Ebene stehen im Gegensatz zum PicReg-Interface keine Methoden zur Verfügung, da diese für die Visualisierung der Pic-Speicher nicht benötigt werden. Bit-orientierte Methoden sind aber genauso wie die oben beschriebenen read- und writeMemory()-Methoden denkbar und würden bei Bit-orientierter Visualisierung zwingend notwendig sein.

3.3.14 PicRegBank

In der Klasse PicRegBank sind alle Register zu den Registerbänken zusammengefaßt. Auf diese Register kann einheitlich sowohl in Bänken als auch im PicMemory-Interface-Stil zugegriffen werden. Die einzelnen Methoden teilen sich in allgemeine Methoden, die PicMemory-Methoden und Methoden zur Beeinflussung der Register unter Rücksicht auf die Bankorganisation. Diese letztgenannten Methoden berücksichtigen also immer die Bits im STATUS-Register, die für die Auswahl der richtigen Registerbank verantwortlich sind, und dienen somit direkt der Ausführung der PIC-Maschinenbefehle.

3.3.15 PicRegBankBreakPoints

Für jedes Register kann ein Breakpoint gesetzt werden. Auf diese Breakpoints kann geordnet über die Klasse PicRegBankBreakPoints zugegriffen werden. Dabei kommt wieder das PicMemory-Interface zum Einsatz. Für jedes Register kann entweder ein Wert von 0 bis 255 als Breakpoint angegeben werden. In diesem Fall wird die Programmausführung unterbrochen, sobald in das entsprechende Register der angegebene Wert geschrieben wird (bzw. ein Bit verändert wird, so daß der angegebene Wert erreicht wird). Außerdem können noch die Werte -1 (kein Breakpoint gesetzt) und -2 (Breakpoint bei jedem Schreibzugriff) angegeben werden.

Daraus folgt, daß mehr als 8 Bit notwendig sind, um den Breakpoint-Status eines Registers darzustellen. Die PicRegBankBreakPoints sind also auf 12 Bit Breite eingestellt, damit eine Visualisierung auf Nibble-Ebene (ein Nibble besteht aus vier Bits und läßt sich einfach durch eine Hexadezimal-Ziffer darstellen) möglich ist.

3.3.16 PicEprom

PicEprom ist der Programmspeicher des PIC. Er ist 14 Bit breit und 1024 Speicherstellen lang. Außer den PicMemory-Methoden zum normalen Beschreiben und Auslesen des PicEproms existieren noch Methoden für File-Operationen.

Die File-Operationen erlauben das Abspeichern und erneute Laden des Programmspeichers. Dabei kommen zwei verschiedene Dateiformate zum Einsatz, einerseits das von Intel eingeführte und von Microchip auf 16 Bit erweiterte „Intel-HEX-Format" (siehe hierzu hades.utils.IntelHexFile) und andererseits ein proprietäres Format („Pic-ROM-Format"), dessen Beschreibung gleich folgt. Das Intel-HEX-Format wird nur lesend unterstützt und dient dem Einlesen von Files, die vom Microchip-Assembler erzeugt werden. Das Pic-ROM-Format dient dem Einlesen und Abspeichern von ROM-Inhalten. Dank seines einfachen Aufbaus können Dateien dieses Formats aber auch einfach unter Zuhilfenahme eines ASCII-Editors verändert werden. Die Intel-HEX-Dateien sind an der Dateinamenserweiterung „.hex“ zu erkennen, die Pic-ROM-Dateien tragen die Dateinamensendung „.rom“. Soll mit der load()-Methode eine Datei eingelesen werden, so wird anhand der Dateinamenserweiterung entschieden, welches Dateiformat angenommen wird.

Pic-ROM-Dateien enthalten nacheinander entweder Adressen (diese sind durch einen abschließenden Doppelpunkt zu erkennen) oder Datenwerte, getrennt durch Leerzeichen oder Zeilenvorschübe. Dabei sind sowohl Adressen als auch Daten hexdezimale Werte. Das Einlesen einer Pic-ROM-Datei geht folgendermaßen vor sich: Zuerst wird der Adreßzähler auf 0 gesetzt. Folgend wird in einer Schleife bis zum Dateiende der nächste Wert eingelesen. Wenn es sich bei diesem Wert um eine Adresse handelt, wird der Adreßzähler auf diesen Wert gesetzt. Handelt es sich nicht um eine Adresse, dann wird der Wert an die durch den Adreßzähler angegebene Adresse geschrieben und der Adreßzähler um eins erhöht.

3.3.17 PicEpromBreakPoints

Wie zur PicRegBank existiert auch zum PicEprom eine entsprechende Klasse zum Zugriff auf die Breakpoints des Programmspeichers. Die Breite des Breakpoint-Speichers ist in diesem Fall nur auf vier Bit beschränkt, da für jede Programmspeicheradresse nur festgelegt werden kann, ob bei Erreichen dieser Adresse ein Breakpoints ausgelöst werden soll. Jeder Wert, der sich von Null unterscheidet, stellt dabei einen Breakpoint dar.

Die Bearbeitung der Programmspeicher-Breakpoints findet im PicPcReg statt, da hier die beste Kontrolle über die gerade bearbeiteten Speicheradressen ausgeübt werden kann. Dazu wird bei jedem Schreibzugriff auf das PicPcReg (also auch bei normaler Erhöhung des PC zum Zugriff auf den nächsten auszuführenden Befehl) überprüft, ob PicEpromBreakPoints für die im PC enthaltene Adresse einen Breakpoint vorsieht.

3.3.18 PicEeprom

Ähnlich dem PicEprom existiert auch eine Implementation des Daten-EERPOM-Speichers mit dem Namen PicEeprom. Grundsätzlich werden hier alle Methoden inklusive der Datei-Operationen zur Verfügung gestellt, die auch im PicEprom realisiert sind. Zusätzlich gibt es beim PicEeprom noch die Möglichkeit Lese- und Schreiboperationen auszuführen, die statt direkter Parameterübergabe die Werte aus EEADR und EEDATA übernehmen. Außerdem wird noch eine Method wakeup() angeboten, die vom Wakeup-Mechanismus des Pic16C84 (siehe Abschnitt 3.5.8 auf Seite 48) benutzt wird, um die wirkliche Schreiboperation durchzuführen. Der Konstrukter erwartet deshalb die Übergabe des EEADR-, EEDATA- und EECON1-Registers. Das EECON2-Register, das im Vorbild zur Vorbereitung von Zugriffen auf das EEPROM dient, wird hier nicht weiter beachtet.

3.3.19 PicStack

Der PIC enthält einen Hardware-Stack mit acht Speicherstellen für Rücksprungadressen. Sind bereits alle acht Speicherstellen des Stack belegt und soll ein weiterer Wert auf den Stack geschrieben werden, wird ohne Rückmeldung das erste Element des Stacks überschrieben.

Die Implementation des PicMemory-Interfaces enthält eine Besonderheit: Die Größe des Stacks wird hier mit neun Elementen angegeben. Das neunte Element ist dabei der Stackpointer. Aus diesem Grund ist es in der Visualisierung möglich, den Stackpointer ganz normal als

Speicherstelle zu behandeln und auf diesem Wege beeinflussen zu können. Doch dazu mehr im Abschnitt 3.4.5 auf Seite 42.

Ansonsten enthält der Stack die für eine Stack-Implementation üblichen Methoden push() und pop(), die sich bei einem Unter-/Überlauf wie im ersten Absatz beschrieben verhalten.

Für den Stack gibt es keine entsprechende Breakpoint-Klasse, da sich diese Funktionalität ziemlich einfach durch entsprechende Breakpoints im Programmspeicher nachbilden läßt. Trotzdem sind auch im PicStack entsprechende Grundlagen implementiert, um eine Breakpoint-Verwaltung nachzurüsten. Zu diesem Zweck müßte nur eine Klasse PicStackBreakPoints geschrieben werden, die dann den Zugriff auf die Breakpoint-Werte der Stack-Register konform zum PicMemory-Interface erlauben und sich nur gering von der Klasse PicRegBankBreakPoints unterscheiden würde.

3.3.20 PicDecode

Bevor mit der Verarbeitung der einzelnen Befehle begonnen werden kann, müssen die Befehle dekodiert werden. Dieses passiert bei der Implementation des PIC an zentraler Stelle im PicDecode. Hier wird jeder Befehl in verschiedene Einzelteile zerlegt. Als Ergebnis werden dann einige Bit-Gruppen angeboten, die in ähnlicher Form als Steuersignale in der Hardware vorliegen müßten. Dabei ist der relativ übersichtliche und wohlgeordnete Aufbau der PIC-Maschinen-Befehle eine große Erleichterung, so daß die Dekodierung der Befehle wirklich nur ein Aufteilen der Befehls-Bits in die einzelnen Bit-Gruppen darstellt. Ein Teil der Dekodierarbeit ist allerdings in andere Bauteile verlegt, so werden z.B. Teile der ALU-Befehle auch in der PicAlu-Klasse dekodiert. Diese Implementation bietet sich allerdings auch wegen der klaren Trennung der verschiedenen Befehlsgruppen an.

Die einzelnen, von PicDecode zur Verfügung gestellten Signale folgen nun in einer Übersicht:

Bits 12-13: Die Befehlsklasse, wie schon im zweiten Kapitel beschrieben. Diese beiden Bits entscheiden, wie die restlichen Bits kodiert sind. Im PicDecode wird allerdings nicht darauf geachtet, hier werden (wie in der Hardware wahrscheinlich auch) alle verschiedenen Dekodierungen „gleichzeitig“ vorgenommen.

Bits 0-6: Falls ein Befehl der Klasse der Byte-orientierten Registerbefehle angehört, dann ist hier die Registernummer enthalten.

Bit 7: Dieses Bit gibt bei Byte-orientierten Registerbefehlen an, ob das Ergebnis des Befehls in das angegebene Register oder in das Work-Register (zurück-) geschrieben werden soll.

Bits 7-9: Bei den Bit-orientierten Befehlen wird mit diesen drei Bits die Bitnummer des zu beeinflussenden Bits angegeben.

Bits 0-7: Die unteren acht Bit stellen bei Absolutwertbefehlen den Absolutwert dar.

Bits 0-10: Bei Sprungbefehlen mit Adreßangaben sind die unteren elf Bit die Zieladresse des Sprungs.

Bits 8-11: Falls ein Byte-orientierter Registerbefehl oder ein Absolutwertbefehl vorliegt, stellen die Bits 8-11 den Befehlsanteil zur Steuerung der ALU dar.

Bits 10-11: Die Bit-orientierten Befehle finden ihren ALU-Befehlsanteil in den Bits 10-11.

Bit 11: Falls ein Sprungbefehl vorliegt, gibt Bit 11 an, ob die Rücksprungadresse auf dem Stack zwischengespeichert werden soll, und unterscheidet damit GOTO und CALL.

Jedes einzelne der gerade aufgeführten Signale stellt PicDecode zur Verfügung damit andere Baugruppen des Kern diese auswerten können. Dabei stellt die zur Verfügung gestellte Registernummer einen Spezialfall dar, denn hier werden schon die Bits RP0 und RP1 des STATUS-Registers mit berücksichtigt.

3.3.21 PicAlu

Die Implementation der arithmetischen, logischen und Bit-orientierten PIC-Maschinen-Befehle ist im Quelltext PicAlu.java zu finden. Hier sind mehrere Klassen enthalten: Einerseits die Klasse PicAlu selber und andererseits für jede Rechenoperation eine PicCalc*-Klasse. Da viele der arithmetischen und logischen Befehle im PIC mehrfach vorhanden sind (für Operationen einerseits mit dem Work-Register und einem normalen Register und andererseits mit einer Konstante und dem Work-Register als Operanden), werden diese Klassen mehrfach benutzt. Dafür werden in der PicAlu-Klasse zwei Tabellen angelegt, die den verschiedenen ALU-Opcodes (die Bits 8-12 eines Befehls) der Befehle die verschiedenen PicCalc*-Klassen zuordnen. Für die Bit-orientierten Befehle lohnte sich ein solches Vorgehen nicht, da es hiervon nur vier verschiedene Befehle gibt und diese nicht mehrfach verwendet werden.

Da einige der ALU-/Bit-Befehle einen bedingten Sprung über den als nächstes auszuführenden Befehl hinweg ausführen, gibt es in jeder PicCalc- und in der PicAlu-Klasse die Möglichkeit, eine Variable zu setzen, die ankündigt, daß der nächste Befehl ausgelassen werden soll.

3.3.22 PicClocking

Die PicClocking-Klasse liefert die verschiedenen Taktsignale, die intern und extern vom PIC angeboten werden. Von diesen Signalen gibt es insgesamt sechs verschiedene: Der PIC hat einen Takteingang OSC1, von dem der Haupttakt kommt. Aus diesem Taktsignal werden alle weiteren Takte generiert, so für jede der vier Befehlsausführungsphasen ein Signal und der externe Taktausgang OSC2, der den Befehlstakt, also ein Viertel von OSC1, darstellt.

Im PicClocking sind alle Taktsignale als PicBitReg realisiert. Deshalb können alle Takte einfach abgefragt und visualisiert werden (dazu später mehr). Die Methode setOsc1() bietet die Möglichkeit, OSC1 zu beeinflussen. Abhängig davon werden dann die anderen Takte gesetzt.

3.3.23 PicWdt

Der Watchdog-Timer (WDT) ist in der Klasse PicWdt realisiert. Damit der WDT aktiviert und deaktiviert werden kann, existiert die Methode setEnabled(). Wenn der WDT aktiviert ist, dann muß die Umgebung (also im Normalfall die HADES-Simulation) dafür sorgen, daß die Methode inc() ungefähr alle 70 μs aufgerufen wird.

Die Methode inc() enthält einen eigenen Prescaler (dieser wird im realen Vorbild mit dem Timer-0 geteilt, ein solches Verhalten ist hier aus Vereinfachungsgründen nicht realisiert), der nur dann eingeschaltet ist, wenn der Prescaler von Timer-0 deaktiviert ist. Unter Beachtung aller für den WDT wichtigen Bits im OPTION-Register wird der WDT also so lange hochgezählt, bis ein Überlauf eintritt. Wenn sich der PIC gerade im SLEEP-Modus befindet, wird er wieder aufgeweckt, sonst wird der komplette Kern zurückgesetzt. In diesem Fall werden also die reset()-Methoden aller Komponenten ausgeführt.

3.3.24 PicBreakPoint

Damit die verschiedenen Komponenten, denen es erlaubt ist einen Breakpoint auszulösen, diesen an zentraler Stelle melden können, existert die Klasse PicBreakPoint. Wenn ein Breakpoint auftritt, dann wird dieser mit report() gemeldet. Danach kann die HADES-Anbindung mit getIsBreakPoint() abfragen, ob ein Breakpoint aufgetreten ist, und die HADES-Simulationsausführung anhalten.

3.3.25 PicExec

Die Zusammenfassung aller zuvor beschriebenen Bauteil-Klassen ist PicExec. Diese Klasse stellt also den Kern dar. Hier werden die verschiedenen Befehlsbearbeitungsphasen nacheinander durchgeführt, so wie von der Pipeline definiert. Für diesen Zweck gibt es verschiedene Methoden, deren Zweck nun im einzelnen aufgezählt wird.

public void fetchNextInstruction(int nextInstr); In der vierten Befehlsausführungsphase wird schon der nächste Befehl aus dem PicEprom geholt. Zu diesem Zweck existiert die Methode fetchNextIntruction(). Nicht unter allen Umständen wird allerdings der nächste Befehl aus dem PicEprom geholt. Falls der vorige Befehl das skip-Bit gesetzt hat (das passiert bei ALU-Befehlen, die angeben können, ob der nächste Befehl übersprungen werden soll), wird der „NOP“-Befehl als nächste Instruktion gesetzt. Falls fetchNextInstruction() einen anderen Wert als -1 übergeben bekommt, wird dieser Wert als Befehl angenommen. Und sonst wird der als nächstes auszuführende Befehl aus dem PicEprom geholt. Der Mechanismus, einen Befehl direkt an fetchNextInstruction zu übergeben, wird nur zum Aufruf von Interrupts benötigt.

public void incPc(); Der Programmzähler wird im Normalfall immer um eins erhöht, nachdem ein Befehl bearbeitet wurde. Falls allerdings ein Sprung vorliegt, wird der PC statt dessen mit der Zieladresse des Sprungs geladen.

public void firstCycle(); In der ersten Phase der Befehlsausführung (eigentlich ist diese Phase schon die zweite, wenn das Lesen des Befehls aus dem PicEprom auch schon als Phase gilt, allerdings wird dieses in der PIC-Terminologie anders gehandhabt) wird der PC erhöht und der Befehl dekodiert. Also wird zuerst incPC() und danach PicDecode.decode() aufgerufen.

public void secondCycle(); In der zweiten Befehlsausführungsphase wird der Operand des Befehls gelesen. Je nach Befehlsklasse wird also entweder ein Registerinhalt oder ein Teil des Befehls selbst dem Operanden zugewiesen. Außerdem wird das skip-Bit zurückgesetzt.

public void thirdCycle(); Die dritte Phase ruft je nach Befehlsklasse PicAlu auf. Außerdem wird der Timer-0 inkrementiert (Aufruf von PicTimerReg.internInc()).

public void fourthCycle(); Während der vierten Phase werden bei Byte-orientierten Registerbefehlen die Spezialbefehle (CLRWDT, RETURN, RETFIE, SLEEP) dekodiert und ausgeführt. Liegt kein Spezialbefehl vor, wird – wie auch bei Bit-orientierten Befehlen und Absolutwertbefehlen – das Ergebnis der PicAlu-Operation entweder ins Work- oder ins Ziel-Register geschrieben. Sprungbefehle merken ihr Ziel für die nächste incPc()-Ausführung vor. Zuletzt werden am Ende von fourthCycle() noch die Interrupts bearbeitet: Liegt ein Interrupt vor (PicIntConReg.isIntr() wird abgefragt), dann wird (nach Verminderung des PC um eins, damit kein Befehl verloren geht) der Befehl „CALL 4" ausgeführt (es wird also fetchNextInstruction(2004h) aufgerufen). Wenn kein Interrupt vorliegt wird einfach der nächste Befehl geholt (fetchNextInstruction(-1)). Dann werden noch die verschiedenen Interrupt-Bits der Registerbank ausgewertet und das PicIntConReg entsprechend beeinflußt.

public void advance(); Hiermit wird PicExec signalisiert, daß nun der nächste Takt anliegt und abhängig vom SLEEP-Mode PicClocking weitergezählt werden soll.

public void clock(); Die Methode clock() wertet den Zustand von PicClocking aus und ruft, wenn es etwas zu tun gibt, die entsprechende *Cycle()-Methode auf.

3.4 Visualisierung

3.4.1 Überblick

Die Beschreibung der Visualisierung gliedert sich eigentlich in zwei Teile. Einerseits müssen alle Klassen beschrieben werden und andererseits ist auch die Bedienung der Visualisierung ein wichtiges Thema. Beides läßt sich allerdings auch gut kombinieren, da die Vorgehensweisen innerhalb der Klassen immer für bestimmte Bedienungskonzepte verantwortlich sind. Deshalb wird die Beschreibung der Klassen direkt mit der Bedienungsanleitung verknüpft.

In diesem Kapitel wird auch auf einige Klassen eingegangen, die zwar im Rahmen dieses Projektes entstanden sind, momentan aber nicht weiter genutzt werden, sondern nur während der Entwicklungsphase zu Debugzwecken verwendet wurden oder durch fortgeschrittenere Elemente ersetzt wurden. Diese Klassen haben ihren Zweck erfüllt und können eventuell bei ähnlichen Projekten wieder verwendet werden, deshalb die Beschreibung hier.

3.4.2 WordButton

Während der Entwicklung des Kerns war es notwendig, ein Eingabeelement, mit dem ein PicReg beeinflußt werden kann, zur Verfügung stehen zu haben. Insbesondere interne Variablen, die nicht über ein PicMemory-Interface zur Verfügung stehen (Beispiele hierfür sind das Work-Register oder das nächste Sprungziel) lassen sich mit einem WordButton einfach anzeigen und verändern. Der Name „WordButton“ kommt daher, daß ursprünglich ein PicWordReg vorausgesetzt wurde. Seit Einführung des PicReg-Interfaces kann aber jedes Pic*Reg benutzt werden.

Der WordButton ist von einem Panel abgeleitet und enthalt einen Label mit dem Namen des WordButton und ein TextField zur Anzeige und Änderung des Wertes. Dementsprechend bekommt der Konstruktor einen Namen und ein Register übergeben. Die einzige weitere Methode des WordButton nennt sich update() und zeichnet den WordButton neu, mit aktualisiertem Inhalt.

Für die Reaktion auf einen neu eingegebenen Text im TextField existiert die Klasse WordTextListener. Hier wird der eingegebene Text in eine Zahl gewandelt und im Register abgespeichert. Für den Zugriff auf das Register werden dabei die writeAll()- und readAll()-Methoden verwendet, damit auch spezielle Register wie z.B. Stack-Register oder der Programmzähler in ihrer vollen Breite verwendet werden können.

Die aktuelle Version der Visualisierung benutzt den WordButton nur noch zur Anzeige des Work-Registers und des Programmzählers. Alle anderen für den normalen Simulationsbetrieb wichtigen Register sind auch in die Registerbank eingeblendet und werden auf diesem Wege angezeigt.

3.4.3 BitButton

Nur während der Entwicklungszeit wurde auch noch ein zum WordButton analoger Button zur Beeinflussung von einzelnen Bits verwendet. Dieser BitButton bekommt als Eingabeparameter einen Namen und ein PicBitReg und zeigt den Namen als Button und dahinter entweder „set“ oder „clr“ an. Der Aufbau des BitButton entspricht weitgehend dem des WordButton, es gibt also einen BitButtonListener, der den Wert des PicBitReg alterniert.

3.4.4 MultiBitButton

Ein weiteres nur während der Entwicklungszeit verwendetes Eingabeelement ist der MultiBitButton, der mehrere BitButtons so zusammenfaßt, daß ein PicWordReg (hier wird wirklich ein PicWordReg statt eines PicReg verwendet, damit auf die einzelnen PicBitReg eines PicWordReg zugegriffen werden kann) angezeigt und dessen Bits einzeln verändert werden können. Der MultiBitButton wurde insbesondere zur Visualisierung der komplexen Steuerungsregister wie STATUS-Register, OPTION-Register oder INTCON-Register eingeführt. Aufgrund des großen Platzbedarfs und der relativ langsamen Bildschirmdarstellung wurden diese Bedienelemente dann aber durch die Darstellung im MemoryView verdrängt.

Der Konstruktor des MultiBitButton bekommt einen Namen, die Namen der einzelnen Bits, getrennt durch Leerzeichen in einem String, und das entsprechende PicWordReg übergeben. Die Namen und die PicBitReg des PicWordReg werden dann auf einzelne BitButton verteilt. Genauso arbeitet die update()-Methode, diese verteilt den update()-Aufruf einfach auf die update()-Aufrufe der beteiligten BitButtons.

3.4.5 MemoryView

Zur Anzeige einer unter Berücksichtigung des PicMemory-Interfaces gestalteten Klasse kann ein MemoryView eingesetzt werden. Hier wird der Inhalt des Speichers als scrollbare Liste angezeigt. Zur Änderung einzelner Einträge steht ein TextField zur Verfügung.

Die Parameter des Konstruktoraufrufs sind der Name des MemoryView, der PicMemory, der angezeigt werden soll, die Anzahl der Zeilen, die dieser Speicher enthält, die Anzahl der sichtbaren Zeilen und die Anzahl der Worte pro Zeile. Außerdem kann noch festgelegt werden, ob es sich bei diesem Speicher um einen Stack handelt, denn in dem Fall wird das letzte Element des Speichers als Stackpointer markiert.

Aufgrund der Anzahl der sichtbaren Zeilen, der Worte pro Zeile und der Breite der Worte wird eine Liste aufgebaut, die zeilenweise den Inhalt des PicMemory darstellt. Dabei wird am Beginn jeder Zeile die Adresse der folgenden Werte angezeigt. Für den Aufbau dieser Liste ist die Methode buildList() verantwortlich, die auch bei jedem update() die Werte der Liste aktualisiert. Wird mit der Maus auf eine Zeile der Liste geklickt, dann erscheint der Inhalt dieser Zeile im Textfield und kann geändert werden. Sobald die Änderung mit der RETURN-Taste beendet wurde, wird der Inhalt des Textfields geparst und in den PicMemory übernommen. Das Parsing der Eingabe wird dabei ähnlich wie das in Abschnitt 3.3.16, Seite 35, beschriebene Einlesen von „.rom“-Dateien gehandhabt: Werte mit folgendem Doppelpunkt sind Adressen für nachfolgende Werte ohne Doppelpunkt, die dann aufsteigend an die entsprechenden Adressen geschrieben werden.

Als weitere Bedienelemente stehen für die Benutzung eines MemoryView noch die Buttons „Up“ und „Down“ zur Verfügung, mit die Anzeige der Liste im PicMemory bewegt werden kann. Wird auf einen der beiden Buttons geklickt, dann ruft der entsprechende Listener eine Methode auf, die den Offset der Speicherdarstellung entsprechend so modifiziert, daß der Inhalt des MemoryView nun zur Hälfte neu dargestellt werden kann. Danach wird diese Änderung mit einem update()-Aufruf neu dargestellt.

Die MemoryView-Klasse wird zur Anzeige aller mit dem PicMemory-Interface ausgestatteten Klassen verwendet. Diese sind PicEprom, PicEpromBreakPoints, PicRegBank, PicRegBankBreakPoints, PicEeprom und PicStack.

3.4.6 ShowSignals

Die Visualisierung des PIC enthält eine eigene Komponente, die ein einfaches Signaldiagramm der wichtigsten internen Register und Steuersignale darstellt. Dazu kann die Historie sowohl von PicBitReg als auch von PicReg mitgeschnitten und angezeigt werden.

Die Basisklasse SignalElement bietet die Funktionalität, für ein Signal dessen Vergangenheit zu verwalten. Ein Signalelement bekommt zu diesem Zweck beim Konstruktoraufruf die Anzahl der zu speichernden vergangenen Werte und einen Namen übergeben. Mit dem Aufruf von advance() kann dann ein neuer Wert als jüngste Vergangenheit übergeben werden. Es handelt sich bei der Implementation des Vergangenheitsspeichers um ein Array von Interegerwerten, die als Ringpuffer benutzt werden. Eine besondere Rolle spielt der Wert -10: Dieser Wert kann mit dem Aufruf von advanceBreak() gespeichert werden und signalisiert eine Unterbrechung der Zeitkontinuität.

Um Werte der Vergangenheit aus dem SignalElement abzufragen, existiert die Methode getHistory(). Diese Methode bekommt als Parameter einen Wert übergeben, der angibt, wie weit aus der Vergangenheit der Wert sein soll. Aus diesem Grund müssen negative Werte angegeben werden, die Null ist die Gegenwart. Mit getHistoryCapacity() kann die Größe des Vergangenheitsspeichers abgefragt werden und getHistorySize() liefert die Anzahl der gerade aktuell im Vergangenheitsspeicher vorliegenden Werte.

Dann gibt es noch die grundsätzliche Unterscheidung, ob ein Signal simpel ist oder nicht. Simple Signale werden speziell dargestellt, sie sind eigentlich nur Boole'sche Werte (genaugenommen StdLogic1164-Werte). SignalElement bietet die Methode isSimpleSignal() zur Abfrage, ob ein simples Signal vorliegt, zum Überschreiben durch erbende Klassen an.

Vom SignalElement werden zwei verschiedene Klassen abgeleitet, einerseits das BitSignalElement (dieses bekommt im Konstruktor zusätzlich ein PicBitReg übergeben) und andererseits das WordSignalElement (hier wird ein PicWordReg im Konstruktoraufruf mit übergeben). Beide Klassen gehorchen dem Interface Advanceable, das nur die Methode advance() für die Übernahme des aktuellen Werts aus dem Register und die Methode advanceBreak() für die Einfügung einer Pause bietet und aus diesem Grund für die einheitliche Bearbeitung beider verschiedener Signalklassen wichtig ist. Das WordSignalElement stellt eine einfache Implementation des Advanceable-Interfaces auf Basis der SignalElement-Klasse dar. Das BitSignalElement ist ein wenig komplizierter, da hier auf die verschiedenen StdLogic1164-Werte, die ein PicBitReg enthalten kann, eingegangen werden muß. Diese Werte werden in spezielle Werte kleiner 0 umgesetzt, die dann bei der Darstellung in speziellen Farben dargestellt werden und Ausnahmefälle anzeigen können.

Die Klasse ShowSignals, die sich dann um die Darstellung der zuvor mitgeschnittenen Signale kümmert, ist von der Canvas-Klasse abgeleitet. Spezielle, also nicht durch diese Canvas-Ableitung vorgegebene Methoden sind add() (damit wird entweder ein PicBitReg oder ein PicWordReg zur Signalanzeige vorbereitet), advance() (Verteilung auf die einzelnen, zu den Signalen gehörenden advance()-Methoden) und advanceBreak() (wie bekannt).

Die paint()-Methode stellt dann die Signale wie von gängigen Signaldiagrammen bekannt dar.

3.4.7 ShowPic

Die ShowPic-Klasse ist die Zusammenfassung aller Visualisierungselemente. Diese werden hier angelegt, auf die verschiedenen Frames verteilt und arrangiert. Hier wird sozusagen die

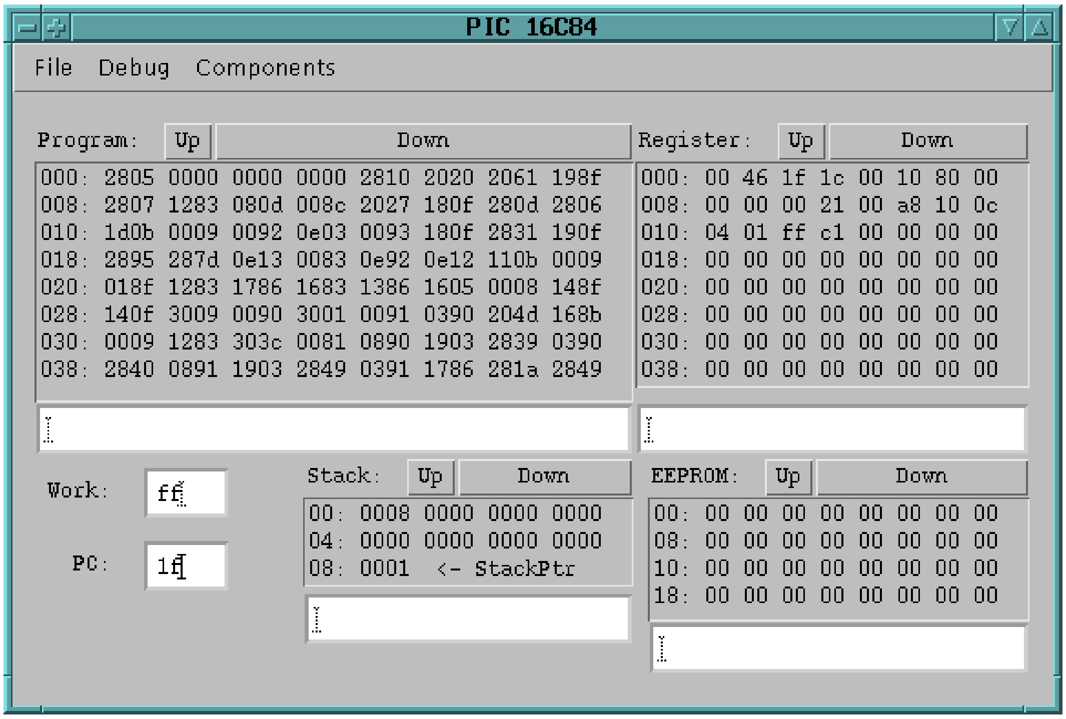

Abbildung 7: Das Hauptfenster der Visualisierung

Funktionalität der Einzelkomponenten zu einem stimmigen Ganzen verdrahtet.

Es werden drei verschiedene Fenster angelegt: Ein Hauptfenster (siehe Abbildung 7), in dem die für den normalen Betrieb wichtigen Daten und Menüs angezeigt werden. Hier erscheint ein je MemoryView für den PicEprom, die PicRegBank, den PicStack und den PicEeprom. Außerdem wird hier noch das Work-Register und der Programmzähler (siehe PicPcReg, Abschnitt 3.3.8, Seite 33) über einen WordButton angezeigt. Dazu kommt das Fenster für die Breakpoint-Verwaltung (siehe Abbildung 8, Seite 45). In diesem Fenster werden zwei MemoryViews angezeigt, einer für die Programmspeicher-Breakpoints und einer für die Register-Breakpoints. Im letzten Fenster erscheinen die Signale in Form eines ShowSignals-Elements (siehe Abbildung 9 auf Seite 46).

Das Menu des Hauptfensters:

File – New: Dieser Menupunkt dient dazu, den PIC vollkommen zurückzusetzen. Zu diesem Zweck wird PicExec.por() aufgerufen. Hiermit werden also alle beteiligten Komponenten in den Einschaltzustand zurückversetzt und der Programmspeicher gelöscht.

File – Open...: Wenn der Programmspeicher mit einem Programm gefüllt werden soll, dann sorgt dieser Menupunkt dafür, daß zuerst ein File-Requester aufgerufen und danach die PicEprom.load()-Methode ausgeführt wird.

File – SaveAs...: Zum Abspeichern des Programmspeichers dient der SaveAs-Menupunkt. Hier wird wieder ein File-Requester geöffnet und danach die PicEprom.save()-Methode aufgerufen.

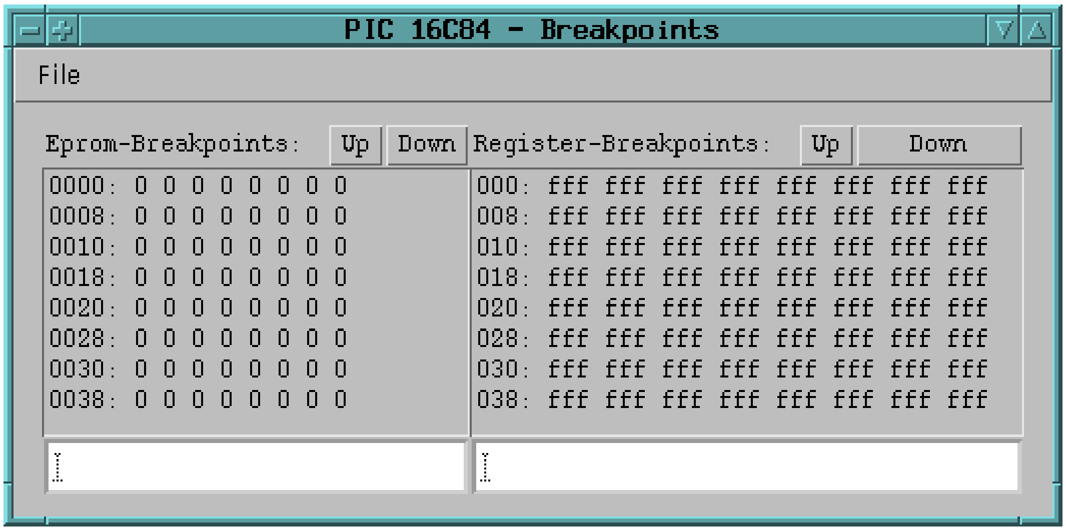

Abbildung 8: Das Fenster zur Darstellung der Breakpoints

File – Close: Das Hauptfenster kann mit dem Close-Menupunkt wieder geschlossen werden. Das bedeutet nicht, daß dadurch die Simulation beeinflußt wird, diese läuft normal weiter (allerdings wesentlich schneller). Weitere ShowPic-Fenster werden von diesem Close nicht beeinflußt, sie haben ihren eigenen Menupunkt für diesen Zweck.

Debug – Breakpoints...: Wenn die Breakpoints angezeigt oder beeinflußt werden sollen, dann sorgt dieser Menupunkt für die Anzeige des Breakpoint-Fensters.

Debug – Signals...: Dieser Menupunkt öffnet das Signal-Fenster, in dem ein ShowSignals-Objekt die Vergangenheit der wichtigsten Signale darstellt. Die Signale werden allerdings nur genau dann protokolliert, wenn das Signal-Fenster geöffnet ist. Leider sorgt ein geöffnetes Signalfenster für eine langsamere Simulationsgeschwindigkeit.

Components – Watchdog-Timer: Der WDT kann per Menu aktiviert und deaktiviert werden. Ist er aktiv, dann wird je nach OPTION-Register und Verwendung des SLEEP-Befehls alle 18 ms bis alle 2,5 Simulations-Sekunden ein Reset ausgelöst. Daher sollte der WDT nur dann aktiviert sein, wenn er wirklich für den Programmablauf benötigt wird.

Zur Erinnerung noch ein Wort zu den Breakpoints: Initialisiert werden die Programmspeicher-Breakpoints mit 0 und die Register-Breakpoints mit fffh. Diese Werte sind gleichbedeutend mit „kein Breakpoint gesetzt“. Für die Programmspeicher-Breakpoints muß ein beliebiger Wert ungleich Null an die entsprechende Speicherstelle geschrieben werden. Die Register-Breakpoints verhalten sich ein wenig anders: Wird ein Wert größer oder gleich Null in eine Speicherstelle geschrieben, so wird ein Breakpoint ausgelöst, sobald dieser Wert in das entsprechende Register geschrieben wird. Falls der Wert ffeh in einer Breakpoint-Speicherstelle steht, wird bei jedem Schreibzugriff auf das entsprechende Register ein Breakpoint ausgelöst. Siehe hierzu auch Abschnitt 3.3.15, Seite 35, zu den Register-Breakpoints und Abschnitt 3.3.17, Seite 36, zu den Programmspeicher-Breakpoints.

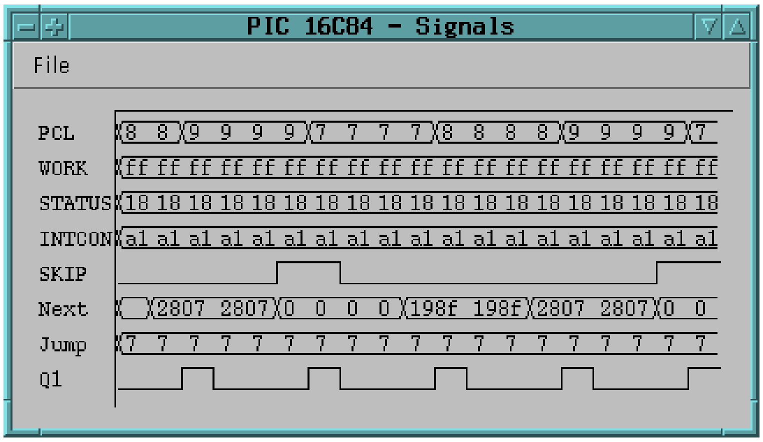

Abbildung 9: Das Fenster zur Darstellung der Signale

3.5 HADES-Anbindung

3.5.1 Überblick

In diesem Kapitel muß zuerst die Schnittstelle beschrieben werden, mit der externe Modelle an HADES angebunden werden. Darauf folgt die Betrachtung der Implementation der wichtigsten zur HADES-Schnittstelle gehörigen Methoden für den PIC, also der Klasse Pic16C84.

3.5.2 HADES-Schnittstelle zu externen Modellen

Alle externen Modelle müssen von hades.simulator.SimObject abgeleitet werden und einige dessen Methoden überschreiben. Zusätzlich gibt es noch das Interface hades.simulator.-Wakeable, das einem Modell die zusätzliche Fähigkeit verleiht, sich ohne ein Simulator-Event aufwecken zu lassen. Nun folgen in einer Übersicht alle für ein externes Modell wichtigen SimObject-Methoden, die überschrieben oder speziell behandelt werden müssen.

Konstruktor: Hier muß als erste Methode super(), also der Konstruktor vom SimObject aufgerufen werden. Im Konstruktor werden alle Ein- und Ausgänge, also die Verbindungen zum HADES, definiert. Dazu existiert im SimObject ein Array mit Elementen der Klasse Port und dem Namen ports. Dieses Array bekommt alle Ports, die das Modell benötigt, zugewiesen.

getSymbolResourceName(): Diese Methode liefert den Namen des Symbols. Im Normalfall ist das der Klassenname mit der Erweiterung „.sym“.

toString(): Der vollständige Name des Objekts wird von toString() geliefert.

write(): Soll ein HADES-Design gespeichert werden, hat jedes Modell die Möglichkeit, eigene spezielle Konfigurationsdaten in die zu schreibende Datei zu integrieren. Diese Konfigurationsdaten können dann bei erneuter Instanziierung von initialize() ausgewertet werden.

initialize(): Falls das Objekt eine spezielle Initialisierung benötigt, kann diese hier ausgeführt werden. Hier können z.B. weitere Dateien für die Konfiguration des Objektes eingelesen werden. initialize() bekommt genau die Daten übergeben, die von write() erzeugt werden.

elaborate(): Für den Start der Simulation wird elaborate() aufgerufen. Hier wird das Objekt also für den Simulationsablauf selbst vorbereitet. Hier wäre der richtige Ort, initiale Wakeup-Events zu erzeugen (siehe Erklärung des Wakeable-Interfaces weiter unten).

evaluate(): Wenn der Simulator ein Event an einem der Ports des Modells feststellt, dann wird evaluate aufgerufen. evaluate() muß dann alle Eingänge abfragen und auf mögliche Veränderungen reagieren, also entsprechend interne Vorgänge anstoßen und eventuell neue Events an den Ausgangsports zu erzeugen. Zur Erzeugung von neuen Events existiert die Methode getSimulator().scheduleEvent(), die für einen bestimmten Port dessen nächsten Wert für einen bestimmten Zeitpunkt festlegen kann.

configure(): Falls es sich bei dem implementierten Modell um eine komplexere Anwendung handelt, die vom Benutzer konfiguriert werden kann, dann sollte das Modell die Methode configure() implementieren und bei dessen Aufruf dafür sorgen, daß eine Benutzerschnittstelle zur Konfiguration des Modells sichtbar wird.

Das hades.simulator.Wakeable-Interface definiert nur eine einzige zu implementierende Methode. Diese trägt den Namen wakeup() und wird aufgerufen, sobald ein Objekt aufgeweckt werden soll. Falls also innerhalb eines zu simulierenden Bauteils interne zeitabhängige Vorgänge ablaufen, die nicht über die Eingänge gesteuert werden, muß diese Methode verwendet werden. Ein Beispiel für ein Bauteil, das mit dem Wakeable-Interface ausgestattet werden muß, ist ein Takterzeuger, der zum Generieren eines Taktsignals in definierten Zeitabständen aufgeweckt wird.

3.5.3 Pic16C84()

Der Konstruktor der HADES-Schnittstelle zum PIC initialisiert das basierende SimObject, PicExec und ShowPic und legt die Ports an. Die Ports entsprechen (bis auf die Stromversorgung) genau den Pins des PIC. Dementsprechend werden sie mit den Namen der Pins versehen.

3.5.4 getSymbolResourceName()

Diese Methode liefert den Namen des Symboldatei, also „Pic16C84.sym“. Eine genaue Beschreibung dieser Datei ist im Anhang A.2 auf Seite 57 zu finden.

3.5.5 write()

Wenn ein HADES-Design gespeichert wird, schreibt das Pic16C84-Objekt den zuletzt benutzen Namen zur Speicherung des Programmspeichers in die Design-Datei. Dieser Dateiname wird in initialize() dazu benutzt, PicEprom dazu zu veranlassen, den Programmspeicherinhalt zu rekonstruieren.

3.5.6 initialize()

Zur Initialisierung eines PIC wird versucht, dessen zuletzt gespeicherten Programmspeicherinhalt einzulesen. Zu diesem Zweck wird aus dem dem Aufruf übergebenen String der darin enthaltene Dateiname entnommen und damit PicEprom veranlaßt, die entsprechende Datei einzulesen.

3.5.7 elaborate()

Der PIC enthält einen WDT, der alle 0,07 ms einen Impuls benötigt. In der elaborate()-Methode wird der initiale WDT-Aufruf eingeleitet, der dann zum Aufruf von wakeup() führt.

3.5.8 wakeup()

Wenn HADES ein Wakeup-Event liefert, dann bedeutet das, daß der WDT inkrementiert und daß eventuell ein Daten-EEPROM-Schreibvorgang ausgeführt werden muß. Deshalb wird durch einen Aufruf von pic.wdt.inc() der WDT also erhöht und durch einen Aufruf von pic.eeprom.wakeup() bei Bedarf der Inhalt von EEDATA ins EEPROM geschrieben. Danach wird dann noch das nächste Wakeup-Event erzeugt, das dann wieder den WDT erhöhen soll.

Der EEPROM-Schreibvorgang ist in der Simulation wesentlich schneller abgeschlossen als in der Realität: [Microchip 97-2] gibt an, daß es ungefähr 10 ms dauert, bis ein Wert ins EEPROM übernommen wurde. Die Simulation beendet den EEPROM-Schreibvorgang schon nach 0,07 ms, damit die Simulation durch das Schreiben ins EEPROM nicht zu langsam wird. Durch einen Counter in PicEeprom.wakeup(), der erst nach dem 140. Aufruf der Methode den Schreibvorgang durchführt, ließe sich das reale Verhalten nachbilden.

Andere Bausteine der mittleren Baureihe besitzen eventuell weitere Timer, die unabhängig von der Außenwelt agieren. Diese müßten an diesem Ort analog zum WDT berücksichtigt werden.

3.5.9 evaluate()

Die eigentliche Schnittstelle zwischen HADES und PicExec ist in evaluate() implementiert. Hier werden alle Eingabe-Pins abgefragt und alle Ausgabe-Pins gesetzt. Abhängig vom Takt- und Reset-Signal wird PicExec.clock() ausgeführt. Außerdem werden noch die Interrupts vorbereitet und, wenn sichtbar, die ShowPic-Fenster auf den neuesten Stand gebracht.

3.5.10 configure()

Soll das Modell konfiguriert werden, wird ShowPic.frame.show() aufgerufen. Das schon im Konstruktor fertig initialisierte Hauptfenster der Visualisierung wird daraufhin angezeigt, der Benutzer hat nun also die volle Kontrolle über den PIC.

A Anhang

A.1 Beispielprogramme

Zur Verdeutlichung der Arbeitsweise des PIC folgen nun zwei einfache und ein komplexes Beispiel.

A.1.1 Erstes einfaches Beispiel: 4-Bit-Counter

In diesem Beispiel soll ein einfacher 4-Bit-Counter aufgebaut werden, der die Anzahl der Zählimpulse eines Eingangs an einem 4-bittigen Ausgang anzeigt. Zu diesem Zweck werden Pin 0 von Port B als Eingang und die Pins 4-7 als Ausgang verwendet. Jeder Impuls am Eingang löst einen Interrupt aus, der in der Interrupt-Service-Routine das Zählregister erhöht. Dieses Beispiel stellt nicht die optimale Implementationsmöglichkeit für diese Aufgabe dar, sondern ist als einerseits möglichst einfach zu verstehende und andererseits möglichst viele Subkomponenten (IO-Ports, Interrupts) des PIC benutzende Demonstration zu verstehen.

Ein HADES-Design, das dieses Beispiel unterstützt, muß einen Clock-Generator mit dem CLKIN, jeweils einen Input mit /MCLR und mit RB0/INT und z.B. ein Hex-Display mit RB4-RB7 verbinden. Abbildung 10 auf Seite 52 zeigt ein HADES-Design für den 4-Bit-Counter.

```
TITLE           "4-Bit-Zaehler: PIC16C84"
SUBTITLE        "Zaehlimpulse von RB0/INT"
;
;**********************************************************************
; Dieser Zaehler bekommt seine Zaehlimpulse von Port B/ Pin 0
; und gibt das Ergebnis an den oberen 4 Bits von Port B aus. Dieses
; Beispiel ist fuer den PIC16C84 implementiert, sollte aber auch mit
; anderen aehnlichen Bausteinen funktionieren.
; Fuer die Ausfuehrung dieses Beispiels muss der WDT deaktiviert sein.
;**********************************************************************
;
  Processor     16C84
  Radix         DEC
  EXPAND
;
  include       "p16c84.inc"
;
_ResetVector    set     0x00
_IntVector      set     0x04
;
Zaehler         set     0x20


  ORG     _ResetVector  ; Hier beginnt die Programmausfuehrung
  call    Init          ; Initialisierung

InfiniteLoop:
```

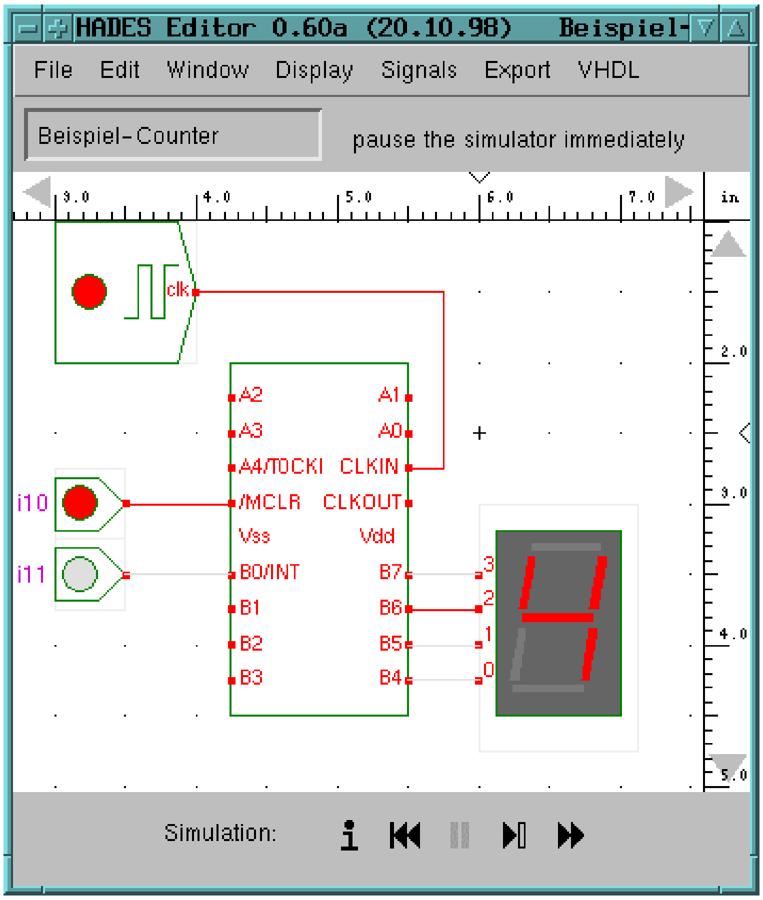

Abbildung 10: HADES-Design des 4-Bit-Counters

```
  goto     InfiniteLoop   ; Sinnlose Endlosschleife

  ORG      _IntVector     ; Interrupt-Adresse
  goto     IntRoutine     ; Auswertung des Interrupts

;*************************************************************************
; Die Initialisierung setzt die unteren 4 Bits von PortB als Eingang
; und die oberen vier als Ausgang. Ausserdem wird der Interrupt fuer
; RB0/INT erlaubt und der Zaehler zurueckgesetzt
;*************************************************************************

Init:
  bsf      STATUS, RP0    ; Bank 1 aktivieren
  movlw    0xE0           ; Steigende Flanke, kein Pullup fuer
  movwf    OPTION_REG     ; RB0/INT-PIN festlegen, TMR0 extern
  movlw    0x10           ; Nur RB0/INT erlauben
  movwf    INTCON         ; INTCON ist in beiden Baenken
  movlw    0x0F           ; Maske fuer Eingangswahl
```

```
  movwf   TRISB          ; ins TRISB
  bcf     STATUS, RP0    ; Wieder Bank 0
  clrf    Zaehler        ; Der Zaehler wird genullt
  clrf    PORTB          ; Die Ausgabe wird zurueckgesetzt
  retfie                 ; RETURN mit Interrupt aktivieren

;**********************************************************************
; Die Interrupt-Service-Routine ist in diesem Fall einfach, da sich
; nicht um die Umgebung gekuemmert werden muss, also keine Status-Bits
; und kein Work-Register gespeichert und wieder hergestellt werden muss.
; Hier wird einfach gezaehlt und das Ergebnis an Port B angelegt.
;**********************************************************************

IntRoutine:
  incf    Zaehler        ; Zaehler erhoehen
  swapf   Zaehler,W      ; Zaehler-Nibbles vertauscht ins W
  movwf   PORTB          ; Und in Port B schreiben
  bsf     STATUS, RP0    ; Bank 1 aktivieren
  movlw   0x10           ; Nur RB0/INT erlauben
  movwf   INTCON         ; Ab ins INTCON damit
  bcf     STATUS, RP0    ; Bank 0 aktivieren
  retfie                 ; Interrupt verlassen

  END
```

A.1.2 Zweites einfaches Beispiel: Watchdog-Timer-Demo

Das zweite Beispiel demonstriert die Benutzung des Watchdog-Timers indem jeder durch den WDT ausgelöste Reset gezählt und am Port B ausgegeben wird. Am Time-out-Bit des STATUS-Registers läßt sich feststellen, ob ein Reset durch den WDT oder den /MCLR-Eingang ausgelöst wurde. So kann bei Programmstart je nach TO-Bit ein Counter hochgezählt werden.

Zur Simulation dieses Beispiels sollten zwei Hex-Displays mit dem Port B, ein Taktgenerator mit CLKIN und ein Input mit /MCLR verbunden werden (ein HADES-Design für diesen Aufbau ist in Abbildung 11 auf Seite 54 zu finden). Außerdem muß im Hauptfenster der PIC-Visualisierung im Menu „Components“ der Watchdog-Timer aktiviert werden.

```
TITLE           "WDT-Demo"
SUBTITLE        "Benutzung des Watchdog-Timer"
;
;**********************************************************************
; In diesem Beispiel wird die Anzahl der aufeinander folgenden
; WDT-Resets gezaehlt und am Port B ausgegeben. Dabei wird die
; Zeitspanne bis zum WDT-Ueberlauf moeglichst kurz gewaehlt.
;**********************************************************************
```

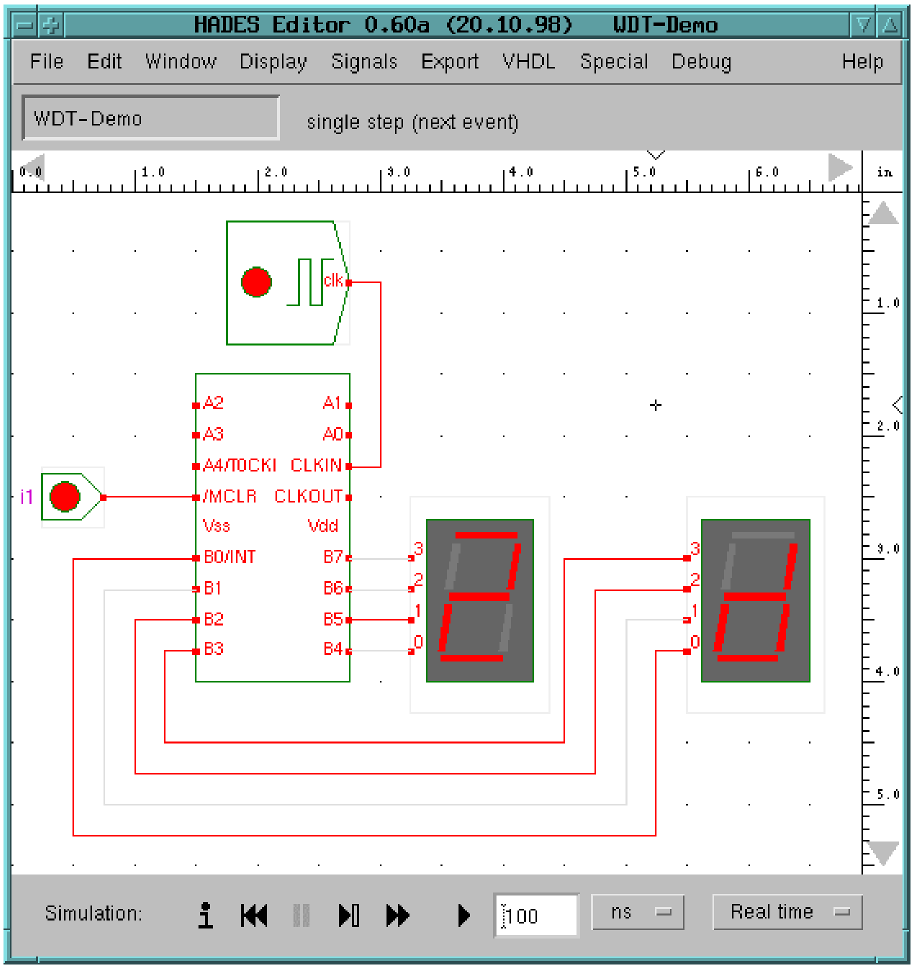

Abbildung 11: HADES-Design der Watchdog-Timer-Demonstration

```
;
  Processor        16C84
  Radix            DEC
  EXPAND
;
  include          "p16c84.inc"
;
_ResetVector       set 0x00

Zaehler            set 0x10


  ORG     _ResetVector         ; Hier beginnt die Programmausfuehrung
  call    Init                 ; Initialisierung

InfiniteLoop:
```

```
  goto    InfiniteLoop        ; Sinnlose Endlosschleife

;*********************************************************************
; Die Initialisierung prueft, ob ein WDT-Reset aufgetreten ist. Wenn ja,
; wird der Zaehler hochgezaehlt. Sonst wird der Zaehler zurueckgesetzt.
; Der Zaehlerstand wird an Port B ausgegeben.
;*********************************************************************

Init:
  movlw   0xff                ; 0xff ins Work-Register
  btfsc   STATUS, 4           ; Ist das Time-out-Bit gesetzt?
  movwf   Zaehler             ; Wenn nicht, dann 0xff in den Zaehler
  clrwdt                      ; Nun schnell WDT zuruecksetzen
  incf    Zaehler,F           ; Zaehler erhoehen, falls kein WDT-Reset
                              ; war, steht nun eine 0 im Zaehler
  bsf     STATUS, RP0         ; Bank 1 aktivieren
  movlw   b'11110111'         ; Prescaler an externen TMR0
  movwf   OPTION_REG          ; und ins OPTION-Register damit
  movlw   0x00                ; Maske fuer Ausgangswahl
  movwf   TRISB               ; ins TRISB
  bcf     STATUS, RP0         ; Wieder Bank 0
  movf    Zaehler,W           ; Zaehler ins Work-Register
  movwf   PORTB               ; und dann am Port B ausgeben
  return

  END
```

A.1.3 Ein komplexes Beispiel: Ein RS232-Repeater

Die Application Note 555 (AN555) von Microchip zeigt die Implementation einer RS232-Schnittstelle mit dem PIC16C71. Dieser Baustein ist dem PIC16C84 sehr ähnlich, deshalb soll diese Application Note als Grundlage für das zweite, komplexe Beispiel dienen.

Die AN555 beschreibt einerseits generelle RS232-Routinen, die einen Pic-Baustein der mittleren Baureihe ohne eingebaute serielle Schnittstelle zu serieller Halbduplex-Kommunikation gemäß dem RS232-Protokoll befähigen, und andererseits eine speziell auf den PIC16C71 ausgerichtete Anwendung, die dessen A/D-Wandler nutzt. Die generellen Routinen können ohne Änderung mit dem PIC16C84 benutzt werden. Da dieser Baustein jedoch keinen A/D-Wandler zur Verfügung stellt, wird hier im folgenden die Modifikation der auf den PIC16C71 ausgerichteten Anwendung beschrieben, damit diese mit dem PIC16C84 zusammenarbeitet und einen anderen Zweck erfüllt. Nun soll die Anwendung darin bestehen, einen RS232-Repeater zu implementieren, der PIC16C84 arbeitet dann also als Echo-Device und versendet jedes empfangene Byte sofort wieder.

Die erforderlichen Änderungen am zur AN555 gehörigen Quelltext rs232.asm beschränken sich auf wenige Punkte:

- Die Kommandos `Processor 16C71` und `include "p16c71.inc"` müssen auf den 16C84 umgestellt werden, damit dessen Adressen und Konstanten bekannt sind.

- Die Unterroutinen `GetADCon0` und `InitAToD` können ersatzlos gestrichen werden.
- Die Hauptroutine zwischen `Start:` und `goto WaitForNextSel` wird durch folgende Hauptroutine ersetzt:

```
Start:
  call    InitSerialPort
;
WaitForNextSel:
;
; Get byte
;
  call    GetChar         ; wait for a byte reception
  btfsc   _rcvOver        ; _rcvOver Gets Cleared when a Byte
                          ; Is Received (in RxReg)
  goto    $-1             ; USER can perform other jobs here,
                          ; can poll _rcvOver bit
;
; Now the byte has been received
;
  bcf     STATUS,RP0      ; make sure to select Page 0
  movf    RxReg,W         ; WREG = Received byte
  movwf   TxReg           ; received byte will be transmitted
;
; Echo byte
;
  call    PutChar
  btfsc   _txmtProgress
  goto    $-1             ; Loop Until Transmission Over,
                          ; User Can Perform Other Jobs
;
  goto    WaitForNextSel ; wait for next selection
```

Dieses Beispiel läßt sich einfach mit einem seriellen Terminal wie dem hades.models.-special.SerialTerminal in HADES ausprobieren. Dazu muß dessen Ausgang TX mit dem Pin 4 von Port A und dessen Eingang RX mit dem Pin 7 von Port B verbunden und einem Pullup verbunden werden. Außerdem muß /MCLR mit einem Input versehen und ein Clock-Generator mit CLKIN verbunden werden (siehe Abbildung 12, Seite 57).

Nach einigen Versuchen hat sich eine Bitrate von 300 Bit/s zusammen mit 8 Bit, keine Parity, einem Stopbit und keinem Hardware-Handshake zusammen mit einer PIC-Clockfrequenz von 500 KHz als simulierbar herausgestellt, allerdings läuft die Simulation dann wesentlich langsamer als die Realzeit (Linux-JDK 1.1.6 auf einem P133 simuliert ca. 1/600 Realzeit).

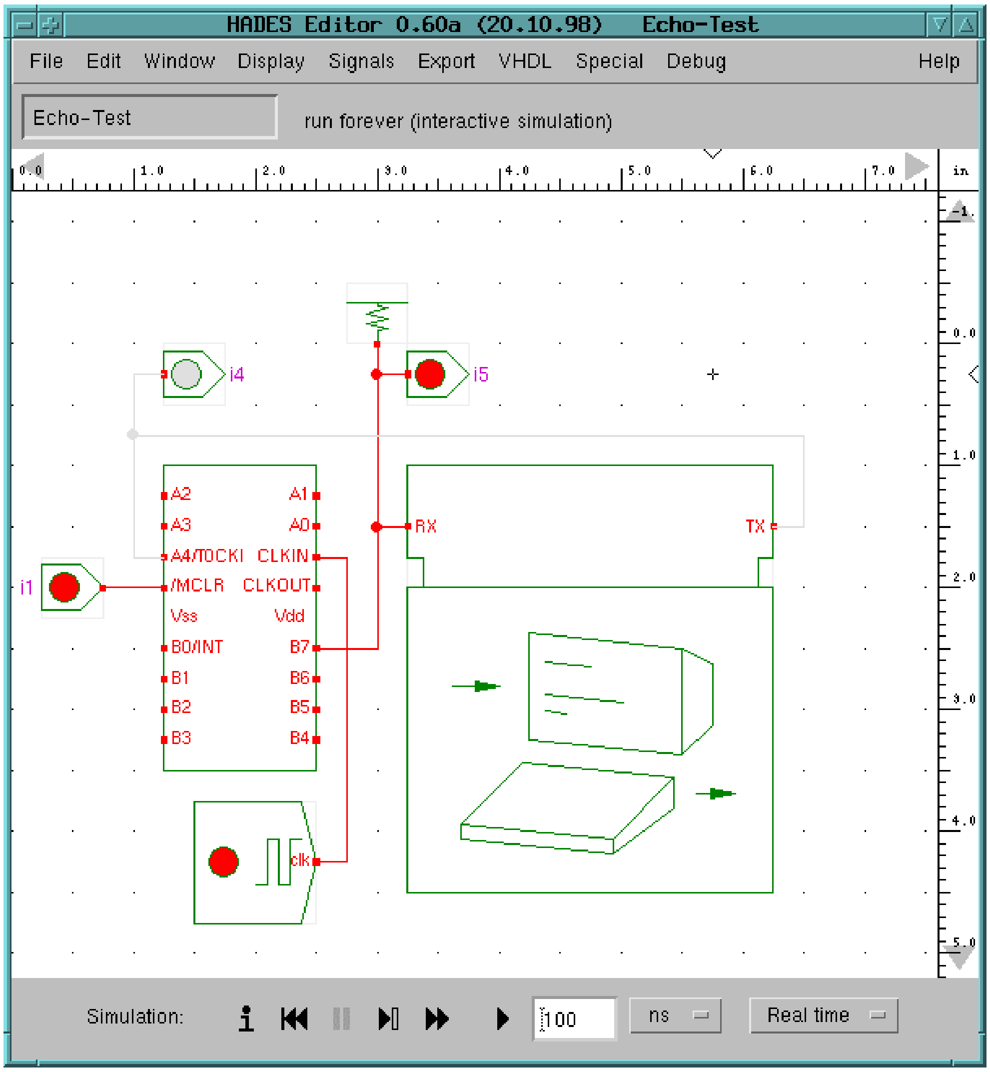

Abbildung 12: HADES-Design des RS232-Repeaters

Wenn höhere Bitraten verwendet werden sollen, dann muß die PIC-Clockfrequenz entsprechend hochgesetzt werden. Da die PIC-Simulation mit bis zu 20 MHz betrieben werden kann sind maximal 9600 Bit/s möglich.

A.2 Die Symboldatei

Soll der PIC16C84 innerhalb von HADES dargestellt werden, wird die Symboldatei mit dem Namen „Pic16C84.sym“ benutzt. Diese Datei definiert den PIC als rechteckigen Baustein mit einer Port-Anordnung und -Beschriftung entsprechend dem originalen Pin-Diagramm:

```
hades.symbols.BboxRectangle 0 0 3000 6000
hades.symbols.Rectangle 0 0 3000 6000
hades.symbols.ClassLabel 1900 450 2 PIC16C84
```

```
hades.symbols.InstanceLabel 2600 5850 2 i0

hades.symbols.PortSymbol 0 600 A2
hades.symbols.PortLabel 150 700 A2
hades.symbols.PortSymbol 0 1200 A3
hades.symbols.PortLabel 150 1300 A3
hades.symbols.PortSymbol 0 1800 A4
hades.symbols.PortLabel 150 1900 A4/T0CKI
hades.symbols.PortSymbol 0 2400 /MCLR
hades.symbols.PortLabel 150 2500 /MCLR
hades.symbols.PortLabel 150 3100 Vss
hades.symbols.PortSymbol 0 3600 B0
hades.symbols.PortLabel 150 3700 B0/INT
hades.symbols.PortSymbol 0 4200 B1
hades.symbols.PortLabel 150 4300 B1
hades.symbols.PortSymbol 0 4800 B2
hades.symbols.PortLabel 150 4900 B2
hades.symbols.PortSymbol 0 5400 B3
hades.symbols.PortLabel 150 5500 B3

hades.symbols.PortSymbol 3000 600 A1
hades.symbols.PortLabel 2500 700 A1
hades.symbols.PortSymbol 3000 1200 A0
hades.symbols.PortLabel 2500 1300 A0
hades.symbols.PortSymbol 3000 1800 CLKIN
hades.symbols.PortLabel 1850 1900 CLKIN
hades.symbols.PortSymbol 3000 2400 CLKOUT
hades.symbols.PortLabel 1550 2500 CLKOUT
hades.symbols.PortLabel 2200 3100 Vdd
hades.symbols.PortSymbol 3000 3600 B7
hades.symbols.PortLabel 2500 3700 B7
hades.symbols.PortSymbol 3000 4200 B6
hades.symbols.PortLabel 2500 4300 B6
hades.symbols.PortSymbol 3000 4800 B5
hades.symbols.PortLabel 2500 4900 B5
hades.symbols.PortSymbol 3000 5400 B4
hades.symbols.PortLabel 2500 5500 B4
```

Die Bildschirmdarstellung dieser Symboldatei ist in den verschiebenen vorher aufgeführten Screenshots der Beispiele zu erkennen.

A.3 Vorstellung des PIC16C84-Simulators von Microchip

Von Microchip ist auch ein Simulator für den PIC16C84 erhältlich. Dieser ist in der integrierten Entwicklungsumgebung MPLAB enthalten und bietet Debugging auf Quelltextebene ähnlich vergleichbarer Produkte für Hochsprachen-Programmentwicklung. Der Simulator kann alle im PIC enthaltenen Speicher anzeigen, der Benutzer kann alle Parameter modifizieren. Wenn ein Quelltext assembliert wird, können alle Labels des Programms in den verschiedenen Speichern inspiziert werden. Es lassen sich auch einzelne Assemblerbefehle eingeben, die sofort ausgeführt werden. Die Integration der Simulation in den normalen Entwicklungsprozeß kann als gut gelungen bezeichnet werden.

Soll allerdings ein Programm simuliert werden, daß auf externe Ereignisse reagiert, also Port A und Port B benutzt, ist ein kompliziertes Vorgehen erforderlich: Mit sogenannten Stimuli werden die Eingänge des PIC zu bestimmten Zeiten auf bestimmte Werte gesetzt. Die Timings der Stimuli müssen dabei aufwendig von Hand erstellt werden. Da mit der Microchip-Simulation keine weiteren Bauteile simuliert werden können, ist die Nachbildung einer komplizierteren elektronischen Schaltung mit dieser Simulation nicht möglich.

Als Fazit läßt sich feststellen, daß die Microchip-Simulation sehr gut zur Programmentwicklung geeignet ist, dabei dient die Simulation allerdings mehr Debug-Zwecken als dem Abbilden realer Vorgänge. Genau diese Lücke soll die hier vorliegende Pic16C84-Simulation im HADES-Umfeld füllen: Mit ihr lassen sich Programme zusammen mit der gewünschen Umgebung des PIC realisieren, hier wird also nicht nur der PIC selbst, sondern auch alle am zu simulierenden Projekt beteiligten weiteren Bauteile berücksichtigt. Da die Pic16C84-Simulation mit den von MPLAB bereitgestellten „.hex"-Dateien umgehen kann, steht dem gemeinsamen Einsatz beider Simulationen nichts im Wege.

A.4 Ein einfacher Programmer

Soll nicht nur die Simulation, sondern ein wirklicher PIC16C84-Baustein verwendet werden, gibt es verschiedene Möglichkeiten, den Programmspeicher zu füllen. Ein einfacher und leicht zu realisierender Ansatz ist der Programmer von Sam J. Engström. Eine genaue Beschreibung des Programmers ist in [Engström 98] zu finden. Dort ist auch eine MS-DOS-Software zu bekommen, mit der der Programmer angesteuert werden kann. Abbildung 13 auf Seite 60 zeigt den Aufbau.

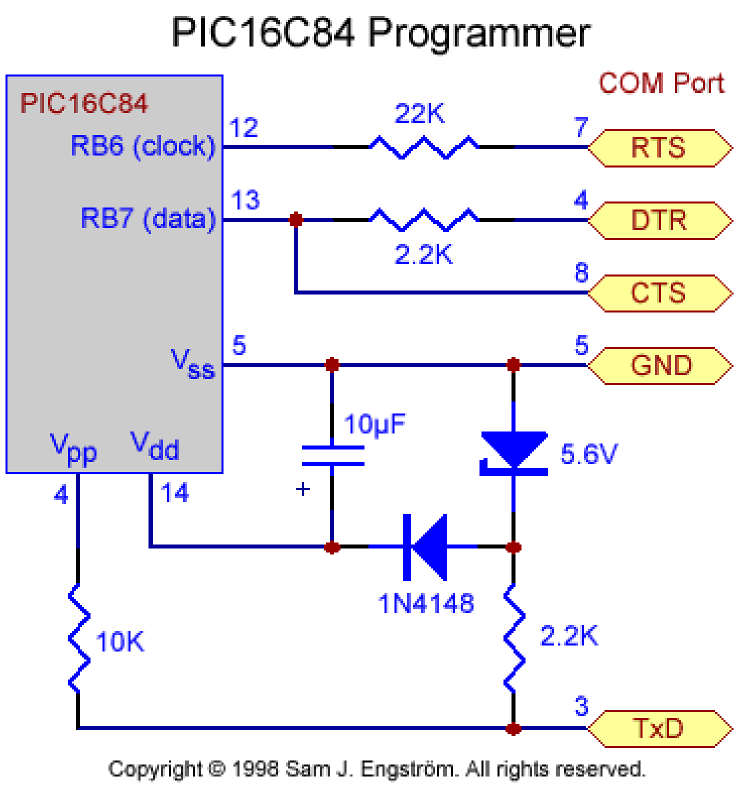

Abbildung 13: Der PIC-Programmer von Sam J. Engström

A.5 Literatur

Literatur

[Engström 98] Sam J. Engström, *PIC Programming - Getting Started*, 1998
`http://www.clinet.fi/~newmedia/pic/index.html`

[Esders 87] E. Esders, *Singlechip-Prozessoren: On-Chip-Peripherie und Kommunikationsmöglichkeiten moderner Mikrocontroller*, Franzis', München, 1987

[Hendrich 97] N. Hendrich, *HADES Tutorial 0.43*, Fachbereich Informatik, Universität Hamburg, 1997
`http://tech-www.informatik.uni-hamburg.de/applets/hades/`

[Microchip 97-1] *PICmicro Mid-Range MCU Family Reference Manual*, Microchip Technology Inc., Chandler, AZ., 1997
`http://www.microchip.com/0/Lit/PICmicro/Refernce/index.htm`

[Microchip 97-2] *PIC16C84: 8-bit CMOS EEPROM Microcontroller*, Microchip Technology Inc., Chandler, AZ., 1997
`http://www.microchip.com/0/Lit/PICmicro/16F8X/30445/index.htm`

[Microchip 97-3] Amar Palacherla, Scott Fink, *AN555: Software Implementation of Asynchronous Serial I/O*, Microchip Technology Inc., Chandler, AZ., 1997
`http://www.microchip.com/0/Appnote/Category/16CXX/index.htm`