

Simulator PIC16f84

Dokumentation

Rechnerarchitektur

Hochschule Offenburg

bei Lehmann, Stefan, Dipl.-Ing. (FH)

Sommersemester 2021

Anton Kesy, Michael Antropov

Abstract

Die Dokumentation eines PIC16F84 Simulators in Java. Erklärungen zur Nutzung, Funktion und Umsetzung.

Inhaltsverzeichnis

[Einleitung Simulator 2](#_Toc71801400)

[Funktion der Programmoberfläche 3](#_Toc71801401)

[LST-Programmvorschau 4](#_Toc71801402)

[Wichtige Werte 4](#_Toc71801403)

[SFR Bits 4](#_Toc71801404)

[Stack 4](#_Toc71801405)

[Fileregister in 2 Banken 5](#_Toc71801406)

[Port A & Port B 5](#_Toc71801407)

[Timing 6](#_Toc71801408)

[Schrittbedienelemente 6](#_Toc71801409)

[Auswahl der LST 6](#_Toc71801410)

[Gliederung des Projektes 7](#_Toc71801411)

[Funktionsablauf 8](#_Toc71801412)

[BTFSx 9](#_Toc71801413)

[CALL 10](#_Toc71801414)

[MOVF 11](#_Toc71801415)

[RRF 12](#_Toc71801416)

[SUBWF 13](#_Toc71801417)

[DECFSZ 14](#_Toc71801418)

[XORLW 15](#_Toc71801419)

[Beschreibung einzelner Komponenten und Funktionen 16](#_Toc71801420)

[Interrupts 16](#_Toc71801421)

[Status-, Option und INTCON-Flags 17](#_Toc71801422)

[FileReader 18](#_Toc71801423)

[Instruktion Decoder 19](#_Toc71801424)

[Arithmetische Logische Einheit (ALU) 20](#_Toc71801425)

[Fazit 21](#_Toc71801426)

[Anhang 22](#_Toc71801427)

[Befehlsliste 22](#_Toc71801428)

[SFR Bits 23](#_Toc71801429)

[Status Register 23](#_Toc71801430)

[Option Register 24](#_Toc71801431)

[INTCON Register 25](#_Toc71801432)

[Fileregister 26](#_Toc71801433)

[Klassen Diagramme 27](#_Toc71801434)

# Einleitung Simulator

Ein Simulator ist eine spezifische Nachbildung, welche möglichst realitätsnah realisiert ist. Dadurch können grundlegende und spezielle Nutzung des Simulierten nachgebildet werden ohne das Simulierte zu benötigen. Auch kann kontrolliert der Simulator abgeändert werden, um spezifische Situationen abzubilden. Der Vorteil ist, dass die Simulation unter vollständiger Kontrolle und abgestimmten Verhältnissen laufen kann. Somit kann genaustens studiert werden, was wie funktioniert oder ähnliches. Auch für Testen und lernen der Nutzung kann ein Simulator sehr nützlich sein. Doch ist eine Simulation oft nicht mehr als eine Simulation und kann die Realität nicht perfekt abspiegeln. Die Komplexität der Entwicklung von Simulatoren steigt auch fast schon exponentiell, wenn das Simulierte komplexer ist.

In diesem Projekt wurde ein Simulator für den Microcontroller PIC16F84 in Java realisiert. Die Programmiersprache konnten wir nicht wählen, aber da wir beide die meiste Erfahrung mit Java zu dieser Zeit hatten, hätten wir auch Java zur Entwicklung gewählt. Die Funktionalität des PIC16F84 wurde nicht vollständig in diesem Simulator realisiert. Dennoch sind Grundfunktionalität vollständig simulierbar. Alle Instruktionen sind unterstützt und somit können einfache Programme ohne Abweichung abgespielt werden. Interrupts, PCLATH, Timer und Watchdog mit dem Prescaler sind auch simulierbar.

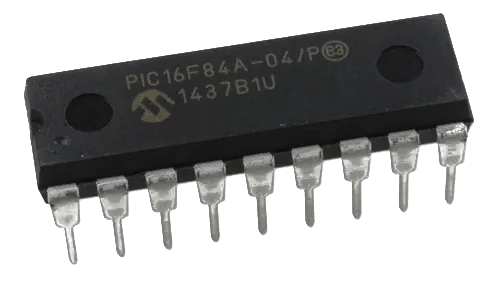
[[1]](#footnote-1)

Abbildung 1 PIC1684A

# Funktion der Programmoberfläche

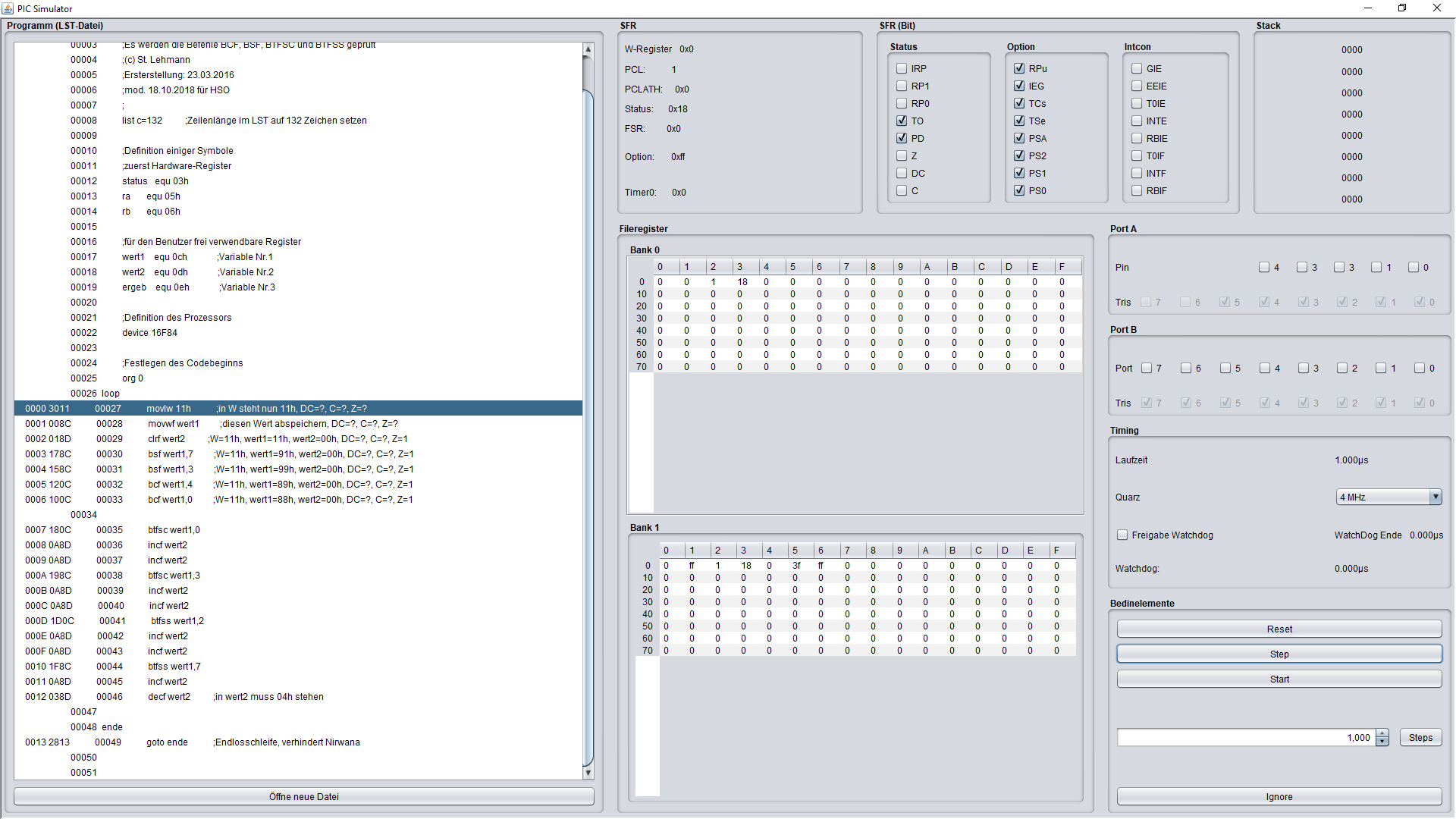
Die Benutzeroberfläche bietet eine detaillierte Einsicht in den genauen Ablauf den Microcontrollers. Es können die Befehle manuell abgearbeitet werden. Auch können die Register und Flags jederzeit manipuliert werden.

Abbildung 2 Komplettansicht Simulator

(4)

(3)

(2)

(1)

(5)

(6)

(7)

(8)

(9)

1. LST-Programm Vorschau
2. Wichtige Werte
3. SFR Bits
4. Stack
5. Fileregister in 2 Banken
6. Port A & Port B
7. Timing
8. Schrittbedienelemente
9. Auswahl der LST

Nach dem Öffnen des Programmes muss zunächst ein Programm im LST Format ausgewählt werden (9). Der File-Reader interpretiert das ausgewählte Programm und der Simulator erstellt einen neuen PIC mit dem Programm im Speicher geladen. Jetzt können die Einstellungen, wie zum Beispiel die Quarzfrequenz (7), angepasst werden. Der PIC steht noch auf Reset und wartet diesen und das zugehörige NOP durchzuführen. Danach kann mit Einzel-, N-Schritten oder automatisch der Simulator gesteuert werden (8).

## LST-Programmvorschau

In diesem Fenster wird die geladene LST-Datei als Text angezeigt. Die markierte Zeile zeigt auf den nächsten Befehl an, welche der PIC ausführen würde bei einem Schritt. Durch Doppelklick auf eine Zeile kann ein Breakpoint erstellt oder entfernt werden. Der Simulator stoppt bei automatischem Durchlauf oder N-Schritten vor diesem Breakpoint.

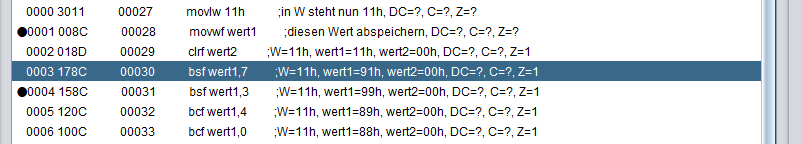


Abbildung 3 Schwarze Punkte stellen Breakpoints dar, blau markiert ist der nächste Schritt

## Wichtige Werte

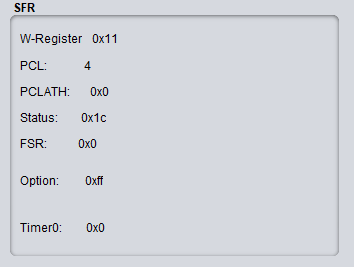
Hier sind alle relevanten und oft genutzten Werte dargestellt, wie der Programmzähler, der PCLATH und das W-Register.

Abbildung 4 Wichtige Werte

## SFR Bits

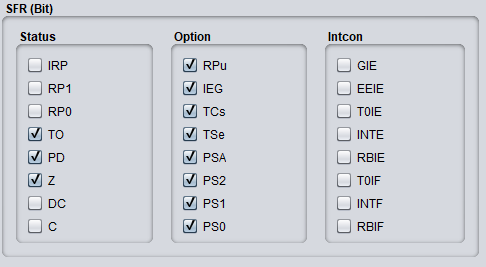
In diesem Fenster werden die SFR Bits angezeigt und durch betätigen der Checkboxen können diese auch direkt manipuliert werden. Die Bedeutung der einzelnen Bits ist im Anhang unter SFR Bits anhand des Datenblattes erklärt.

Abbildung 5 SFR Bits

## Stack

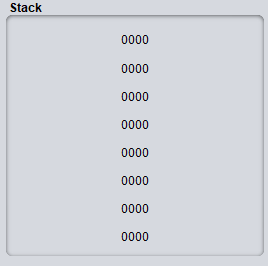
Der Stack ist durch eine einfache List von acht freien Adressen zum ablegen abgebildet. Beim Legen in den Stack ändern sie die Werte dementsprechend.

Abbildung 6 Stack

## Fileregister in 2 Banken

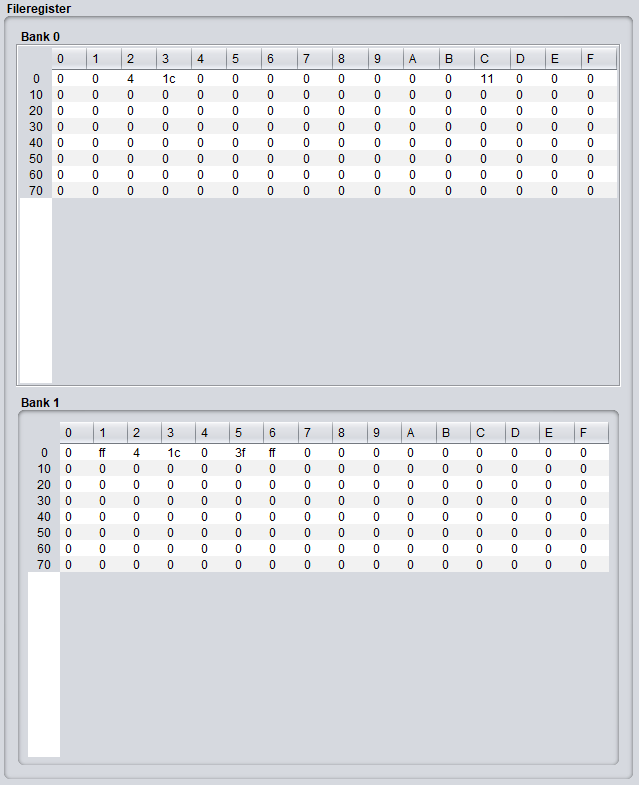
Der PIC16F84 hat nur zwei RAM Bänke zur Verfügung. Diese sind durch zwei Tabellen, welche jeweils 128 Adressen halten, abgebildet. Durch Doppelklick auf die Werte, können diese einfach und direkt manipuliert werden. Die Aufteilung des Speichers kann aus dem Datenblatt im Anhang entnommen werden.

Abbildung 7 FileRegister

## Port A & Port B

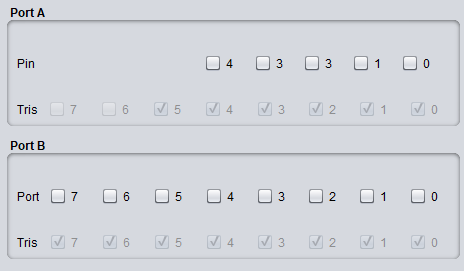
Hier sind der Port A & Port B durch Checkboxen abgebildet. Darunter ist direkt das zugehörige   
Tris-Register, welches entscheidet, ob der Port-Pin ein Eingang oder Ausgang ist. Checken der Box gilt als Hochflanke.

Abbildung 8 Port A & Port B

## Timing

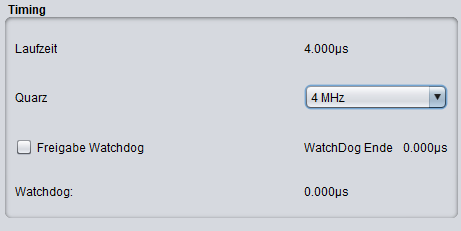
In dieser Gruppierung ist alles Wichtige, was mit der Laufzeit in Verbindung gebracht werden kann, abgebildet. Es kann, die Laufzeit des Microcontrollers in Microsekunden abgelesen werden, die Quarzfrequenz manuell einstellen, den WatchDog freigeben und auch dessen Laufzeit und Endzeit ablesen.

Abbildung 9 Timing

## Schrittbedienelemente

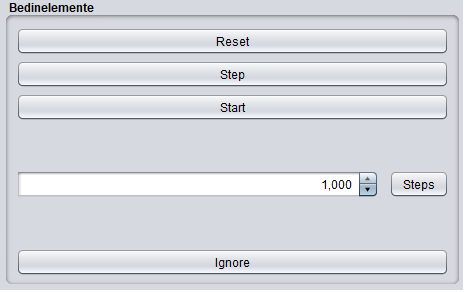
Hier wird der gesamte Simulator mit Schritten gesteuert. Die Reset-Taste setzt den PIC auf den Startwert zurück. Mit Step kann ein einzelner Schritt betätigt werden. Start/Stopp kontrolliert den automatischen Ablauf von Schritten. Mit N-Steps können die eingetragenen Schritte direkt betätigt werden. Ignore ignoriert den nächsten Schritt.

Abbildung 10 Schrittbedienung

## Auswahl der LST

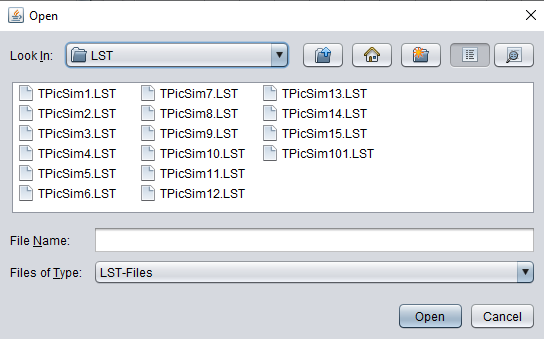
Um den PIC zu initialisieren muss er mit einem gültigen Programm geladen werden. Dies erfolgt durch LST-Dateien. Durch betätigen der Auswahl öffnet sich der File-Chooser und es kann die LST-Datei, welche eingespielt werden soll, ausgewählt werden.

Abbildung 11 FileChooser

# Gliederung des Projektes

Abbildung 12 Komplette Quellcodehierarchie

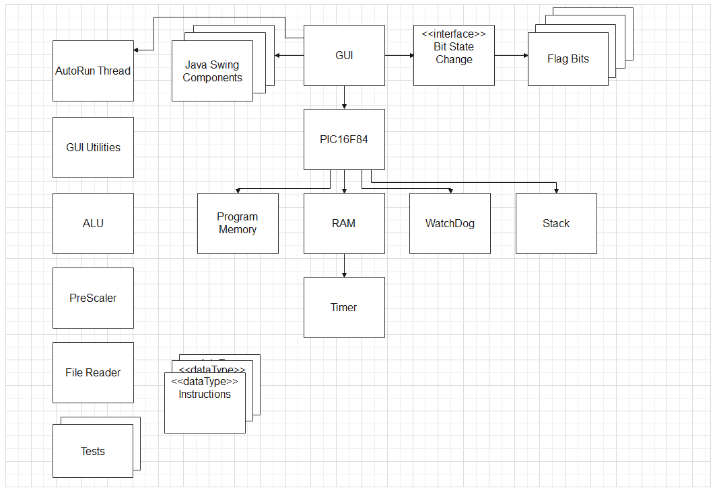
Die äußerste Trennung ist zwischen der GUI, File-Reader, Tests und dem PIC-Simulator, welche aber dennoch miteinander verbunden sind. In der GUI-Gruppierung sind alle GUI relevanten Objekte, wie Java Swing Komponenten oder der Thread zur automatischen Abarbeitung der CPU, vorhanden. Die PIC Gruppierung ist unter PIC und seine Elemente, wie die ALU, Stack oder RAM eingeteilt und Objekte welche zur Gruppierung Instruktionen gehören, wie zum Beispiel eine Datenklasse für die Instruktionen und die Arbeitsklasse für den Decoder.

Abbildung 13 Vereinfachtes UML des Projekts

Hier eine vereinfachte Darstellung der Programmstruktur in einem UML Diagramm. Die Benutzeroberfläche (GUI) steuert den gesamten Simulator. Auch besitzt die Benutzeroberfläche eigene Java Swing Komponenten, einen Thread, welchen den Simulator automatisch ablaufen lässt und die Implementierungen eines Interfaces Bit State Change, welche die Status Checkboxen steuern. Durch Benutzereingabe kann der PIC über das GUI gesteuert werden. Es können Zyklen abgearbeitet werden, die Daten ausgelesen und manipuliert werden. Der PIC besitzt mehrere Bausteine. Diese sind: Der Programmspeicher, in welchen die vom FileReader interpretierten LST-Programmdatei als Liste von Befehlen abgespeichert wird. Der Random Access Memory, welcher die Daten als Integer in einem zweidimensionalen Array von der Länge 128 hält und nebenbei noch eine Hilfsklasse, welchen den Timer repräsentiert. Den WatchDog und den Stack, von der Länge acht. Außerhalb dieser Verknüpfung gibt es noch einige Arbeitsklassen, wie die Arithmetische Logische Einheit (ALU), PreScaler für den Timer oder WatchDog, den FileReader zur Interpretation der LST-Dateien und eine Hilfsklasse für die Benutzeroberfläche.

# Funktionsablauf

Abbildung 14 Vereinfachtes PIC Flussdiagramm

Der FileReader bekommt eine LST-Datei welche er Interpretiert und in eine Array-Liste von LSTLine und InstructionLine umwandelt. Der PIC16F84 wird mit der InstructionLine-Liste initialisiert und er lädt die Befehle in seinen Programmspeicher. Bei Initialisierung erstell der PIC auch seinen RAM, welcher mit den Standardwerten geladen wird, einen neuen Watchdog und führt danach direkt ein Reset aus. Der Microcontroller ist nun mit der NoOperation-Instruktion geladen. Wenn von der Benutzeroberfläche der nächste Schritt gerufen wird, dann wird die momentan geladene Instruktion abgearbeitet. Dies geschieht im InstructionHandler, welcher mithilfe eines Switchs den aktuellen Befehl auswertet und die auszuführenden Funktionen aufruft. Aktuell ist im PIC ein NOP geladen, welches bei der Ausführung keine Operation betätigt. Am Ende wird die Laufzeit noch um die benötigte Zeit addiert. Diese wird aus den Zyklen, welche der Befehlt benötigt und der Quarzfrequenz des Microcontrollers berechnet. Auch werden der Timer und der Watchdog, wenn eingeschalten, behandelt. Zuletzt wird die nächste Instruktion vorbereitet. Dafür wird zuerst nach Interrupt überprüft, danach der nächste Befehl geladen und der Programmzähler inkrementiert. Danach kann von der Benutzeroberfläche wieder ein Schritt getätigt werden.

Folgend sind noch die Befehle BTFSC/BTFSS, CALL, MOVF, RRF, SUBWF, DECFSZ, XORLW und ihre spezielle Abarbeitung, genauer als Beispiele erklärt.

BTFSx

BTFSC (Bit Test, Skip if Clear) und BTFSS (Bit Test f, Skip if Set) sind beide zusammen implementiert. Sie Testen das Bit an der Stelle f im W-Register ob es bei BTFSC gleich Eins oder bei BTFSS gleich 0 ist. Wenn dies wahr ist, dann wird der nächste Befehlt übersprungen und somit der Programmzähler um insgesamt Zwei erhöht.

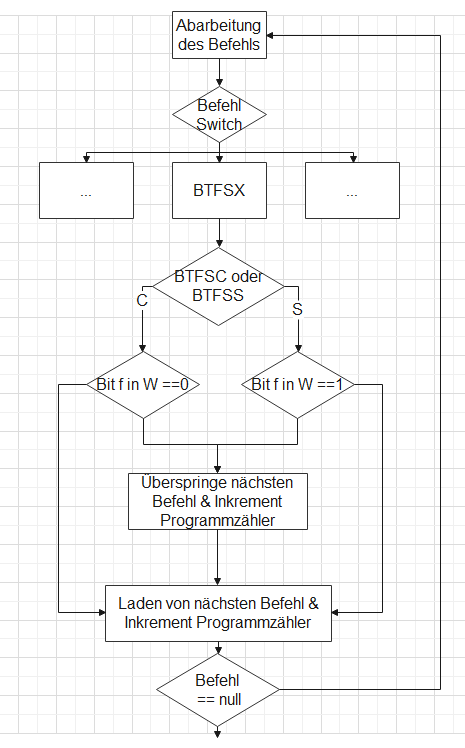
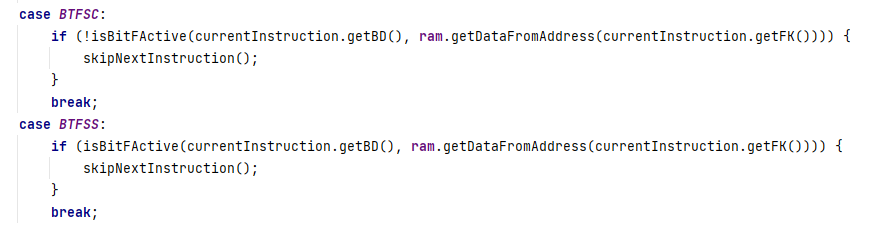
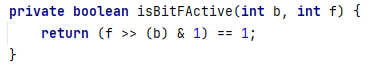


Abbildung 15 Codeausschnitt Befehlsswitch

Abbildung 16 Codeausschnitt isBitFActive

Abbildung 17 Flussdiagramm BTFSX

CALL

Mit dem Call (Call Subroutine) kann auf eine bestimmte Stelle im Programmspeicher gesprungen werden. Mit der Verknüpfung des PCLATH kann damit noch weiter als die eigentlich nur 256 Stellen gesprungen werden. Bei jedem Call wird der aktuelle Programmzähler, für den Return, in den Stack geschrieben. Danach wird der Programmzähler auf die elf Bit große literale Konstante und den PCLATH gestellt.

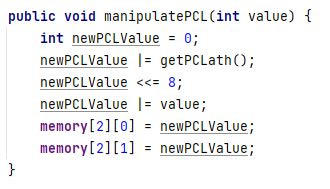
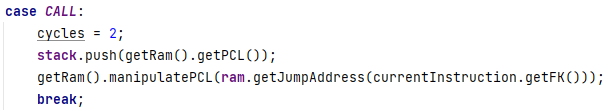


Abbildung 18 Codeausschnitt manipulatePCL

Abbildung 19 Codeausschnitt Befehlsswitch

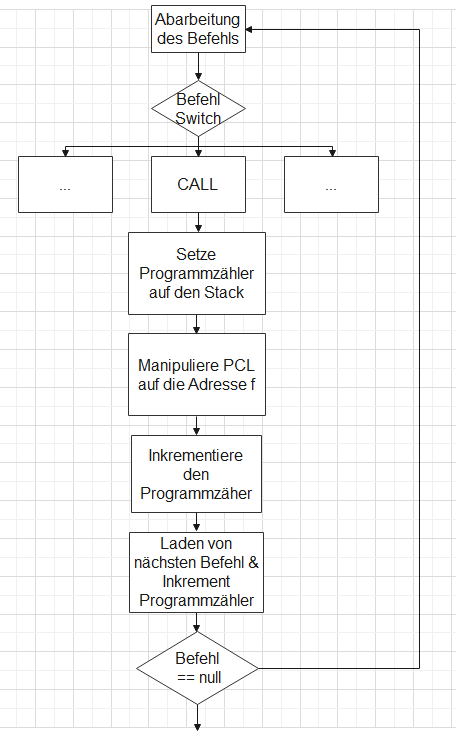
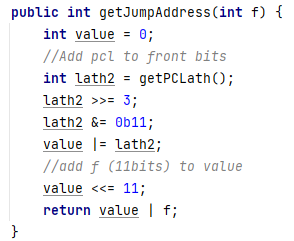


Abbildung 20 Codeausschnitt getJumpAdress

Abbildung 21 Flussdiagramm Call

MOVF

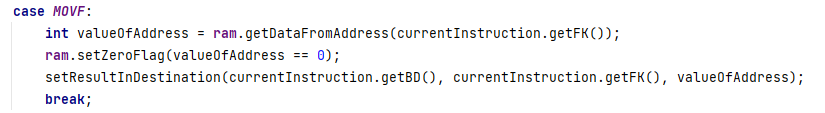
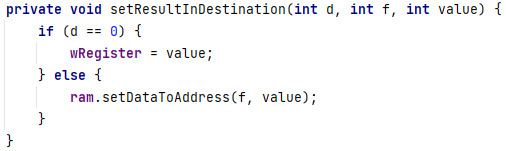
MOVF (Move f) speichert die Daten aus der Adresse f dem Destination-Bit abhängig in das W-Register oder wieder in die Adresse f. Das Zero-Flag wird gesetzt, wenn der Wert Null ist.

Abbildung 22 Codeausschnitt Befehlsswitch



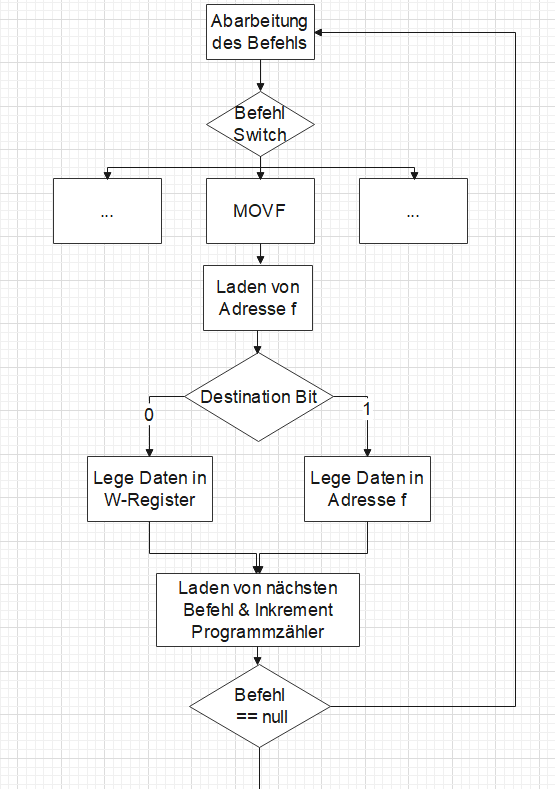


Abbildung 23 Codeausschnitt Destination Bit Auswerter

Abbildung 24 Flussdiagramm MOVF

RRF

RRF (Rotate Right f through Carry) rotiert die Daten auf der Adresse f im RAM um einen Bit nach rechts durch den Carry durch. Wenn das Bit d (Destination Bit) auf 0 steht, wird das Ergebnis im W-Register platziert, wenn nicht, dann wieder auf der Adresse f.

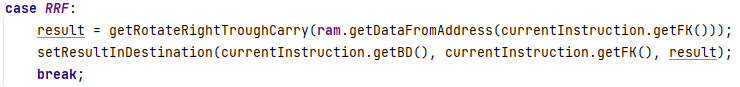


Abbildung 25 Codeausschnitt Befehlsswitch

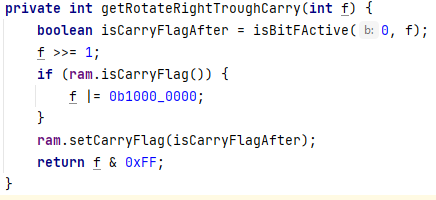
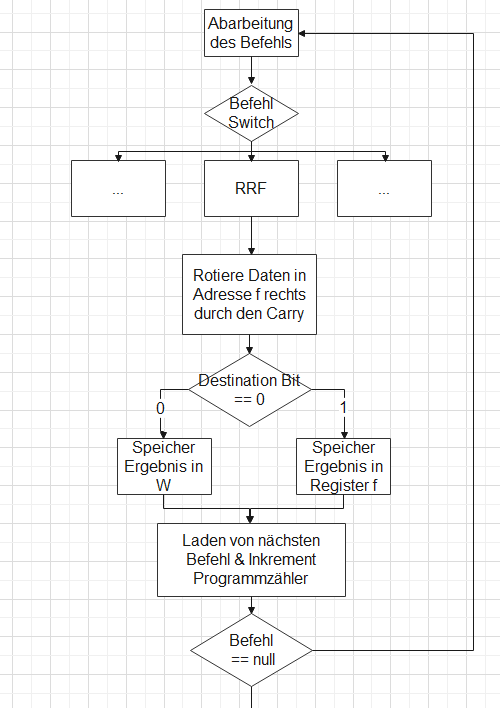


Abbildung 26 Flussdiagramm RRF

Abbildung 27 Codeausschnitt Rotiere Rechts durch Carry

SUBWF

SUBWF (Subtract W from f) subtrahiert die Daten im W-Register mit den Werten auf der Adresse f. Die Subtraktion wird mit Hilfe des addieren des Zweierkompliments verwirklicht. Die Arithmetische Logische Einheit überprüft auch direkt bei der Addition, ob die relevanten Flags, hier Carry, Digit-Carry und Zero gesetzt werden müssen und setzt diese direkt, wenn nötig. Wenn das Bit d (Destination Bit) auf 0 steht, wird das Ergebnis im W-Register platziert, wenn nicht, dann wieder auf der Adresse f.

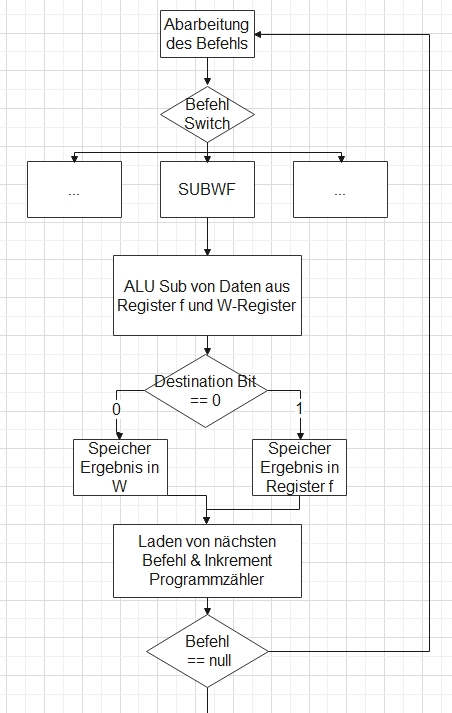
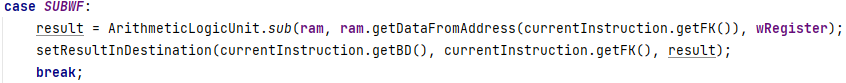


Abbildung 28 Codeausschnitt Befehlsswitch

Abbildung 29 Flussdiagramm SUBWF

DECFSZ

DECFSZ (Decrement f, Skip if 0) dekrementiert die Daten aus Adresse f. Wenn das Bit d (Destination Bit) auf 0 steht, wird das Ergebnis im W-Register platziert, wenn nicht, dann wieder auf der Adresse f. Wenn das Ergebnis Null ist, dann wird der nächste Befehlt übersprungen.

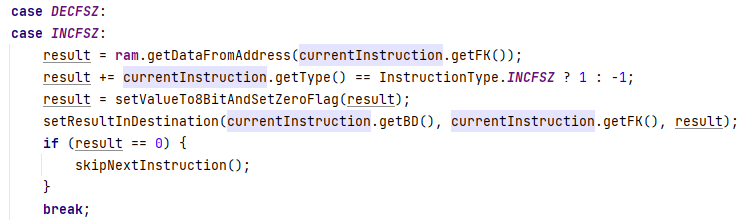


Abbildung 30 Codeausschnitt Befehlsswitch

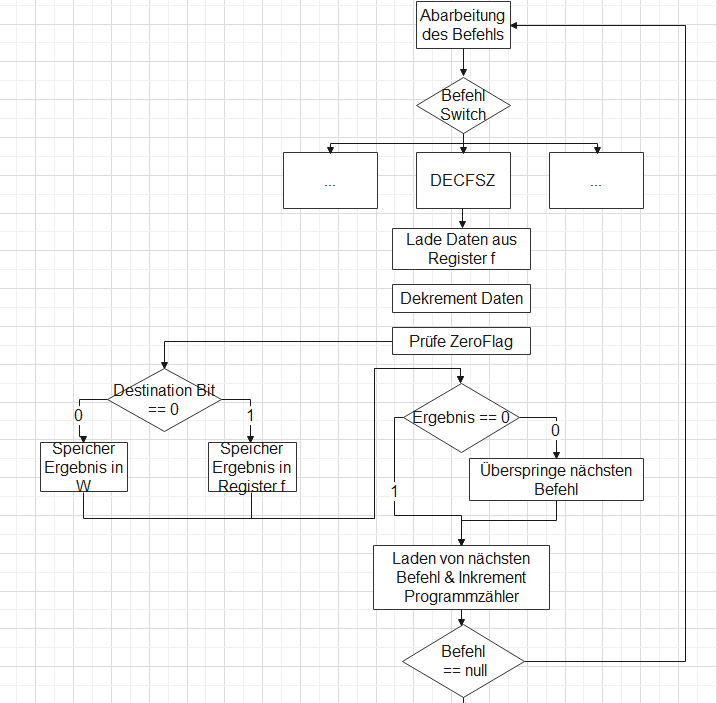


Abbildung 31 Flussdiagramm DECFSZ

XORLW

XORLW (Exclusive OR W with f) macht ein exklusives Oder mit den Werten aus dem W-Register und den Werten aus der Adresse f. Der ALU prüft bei dem XOR nach Flags und setzt, wenn nötig, das Zero-Flag. Wenn das Bit d (Destination Bit) auf 0 steht, wird das Ergebnis im W-Register platziert, wenn nicht, dann wieder auf der Adresse f.

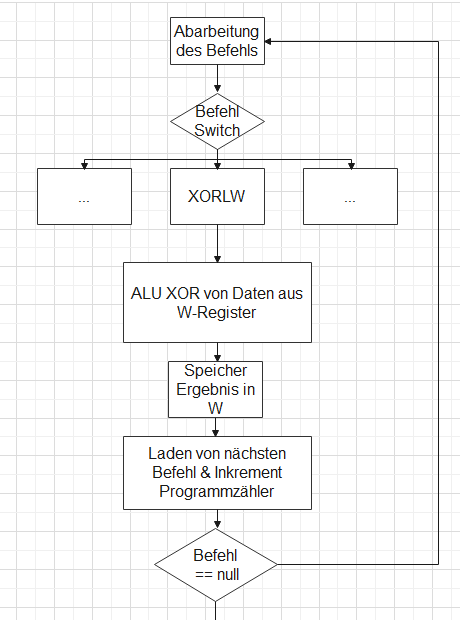
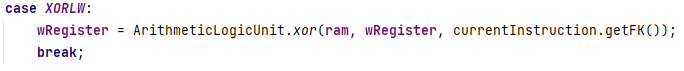


Abbildung 32 Codeausschnitt Befehlsswitch

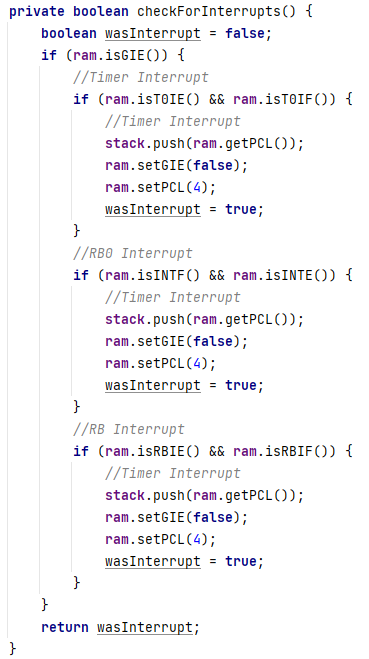
Abbildung 33 Flussdiagramm XORLW

# Beschreibung einzelner Komponenten und Funktionen

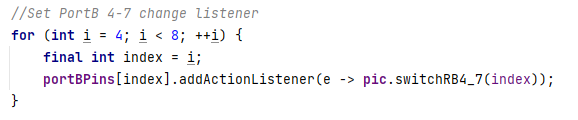
## Interrupts

Es gibt vier wichtige Interrupts bei dem Microcontroller: Den TMR0-Interrupt, welcher aktiviert wird, wenn der Timer überläuft. Den RB-Port-Change-Interrupt, welche sich aktiviert, wenn an den Pins 4-7 von Port B eine Änderung eintritt. Den RB0/INT-Interrupt, welcher durch flanken an Port B Pin 0 ausgelöst wird. Und den EE-Write-Complete-Interrupt, welcher gesetzt wird, wenn das EEPROM fertig mit schreiben ist. Letzteren Interrupt wurde mit dem EEPROM nicht in diesem Simulator realisiert. Alle anderen Interrupts werden bei jedem Zyklus überprüft, ob Interrupts generell aktiviert sind, durch das Global-Interrupt-Enable-Bit angezeit, und dann die einzelnen Flags der Interrupts.

Abbildung 34 Codeausschnitt der Interrupt Überprüfung



Der TMR0-Interrupt wird bei jedem Timer-Wert-Wechsel überprüft. Die Port-B-Interrupts werden durch einen ActionListener in der Benutzeroberfläche gerufen. Beim aktivieren der Pins wird überprüft ob ein Interrupt aktiviert werden muss oder nicht.

Hier als Beispiel der RB-Port-Change-Interrupts

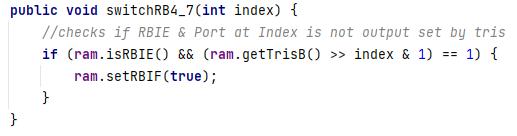
Abbildung 35 Codeausschnitt setzen des ActionListener

Abbildung 36 Codeausschnitt Überprüfung auf RB4-7 Interrupt

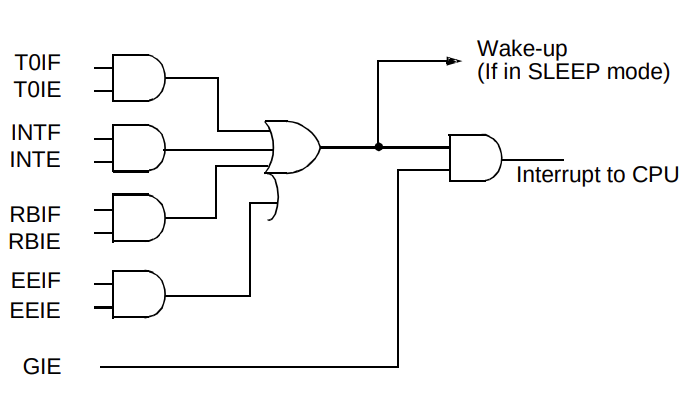
[[2]](#footnote-2)

Abbildung 37 Interrupt Logic

## Status-, Option und INTCON-Flags

Die einzelnen Register haben ihren eigenen Platz im Speicher. Das Statusregister ist auf 03h, das Optionregister auf 81h, also der zweiten Bank. Das INTCON steht auf 0Bh. Jedes Bit der einzelnen Adressen hat eine eigene Bedeutung, welche im Anhang auf dem Datenblatt ersichtlich ist. Das Bit 2 des Statusregisters zum Beispiel zeigt an, ob ein Zero-Flag gesetzt ist. Durch setzen der Flag erhöht sich der Wert des Statusregisters um 2², also um 4 und beim resetten verringert der Wert sich. Somit steht in den Registern eigentlich die Summe bzw. der Stellvertreterwert der Gruppierung der Flags.

In dem Simulator wurden diese auch genau so realisiert. Es können die Adressen direkt beschrieben und wie Werte genutzt werden, aber durch Hilfsmethoden können auch einfach nur die Flags verändert werden.

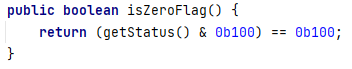
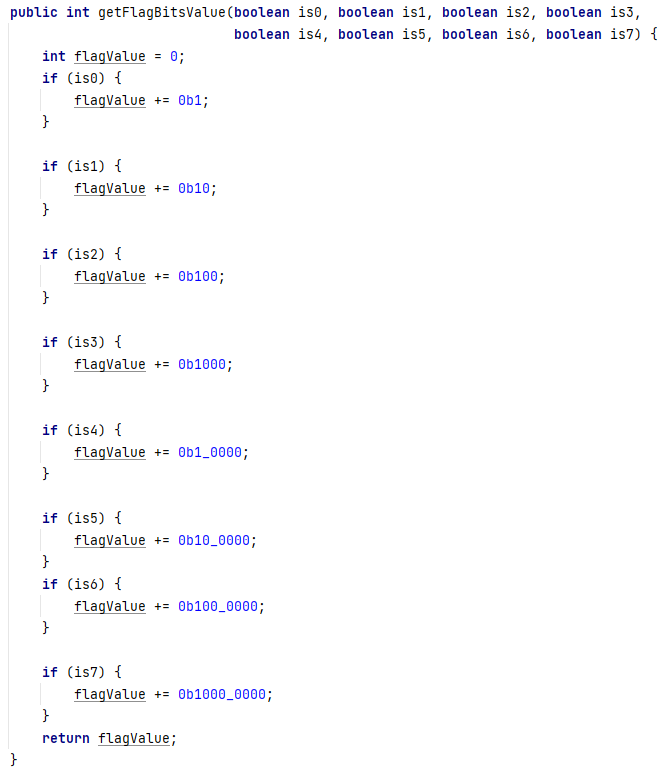
In der Benutzeroberfläche implementiert jeder CheckBox-Listener das BitChange-Interface welches den genauen Bit festlegt, welcher diese CheckBox repräsentiert.

Abbildung 38 Codeausschnitt setzte Flag Register Value

Abbildung 39 Codeausschnitt isZeroFlag

## FileReader

Der FileReader wird mit einer LST-Datei erstellt, welche er Zeile für Zeile interpretiert. Die LST-Dateien werden mit einem Scanner auf ISO\_8859\_1 geladen und per Zeile ausgelesen. Hier der Code für die Interpretation der übergebenden Zeile.

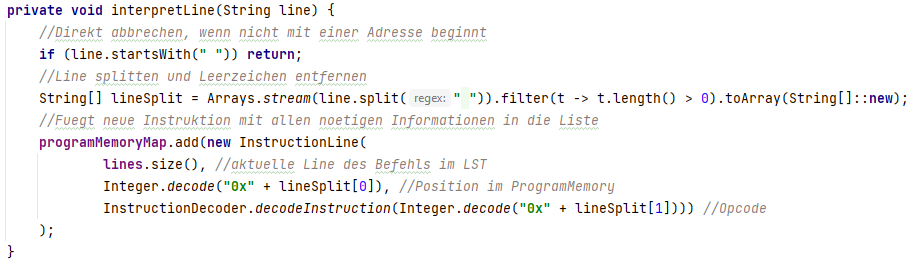


Abbildung 40 Codeausschnitt Interpreter der Zeilen

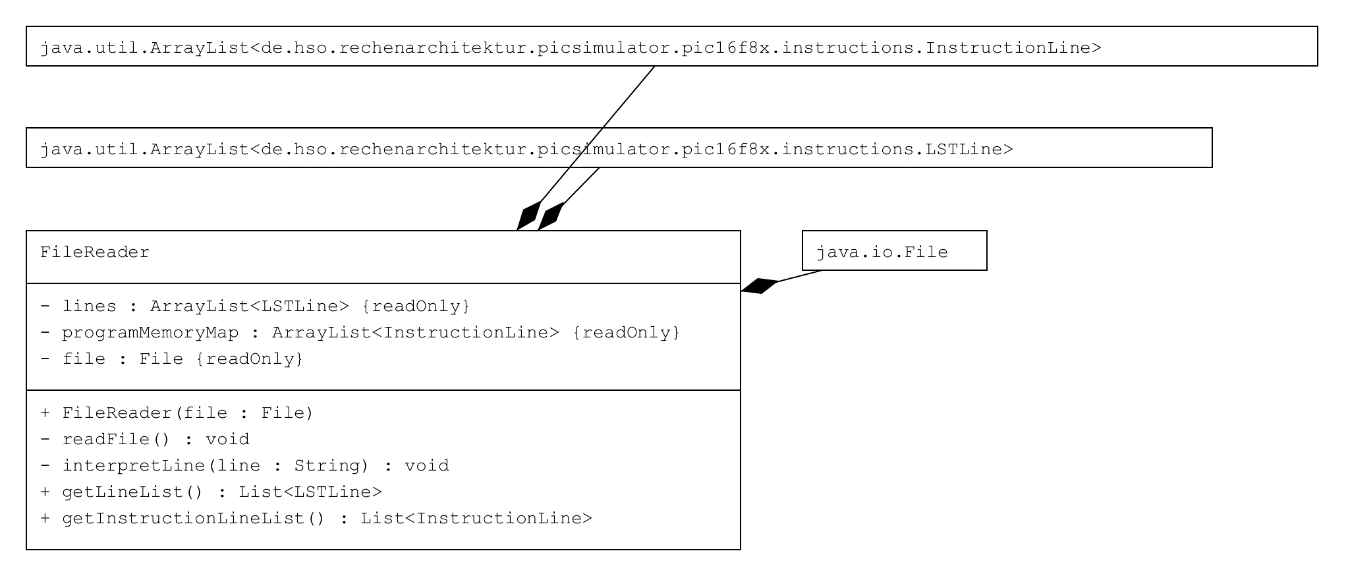


Abbildung 41 UML FileReader

## Instruktion Decoder

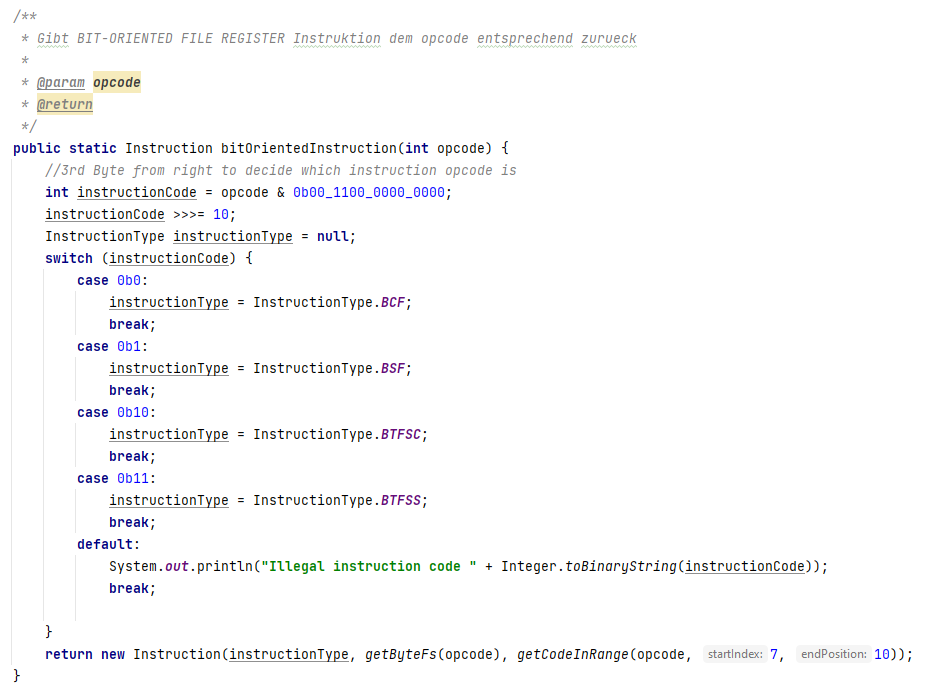
Der Instruktion Decoder dekodiert die Hex-Werte des Programmcodes, den   
14-Bit Opcode (operation code), in Befehle mit Hilfe von Bitmasken. Die zwei höchstwertigen Bits im Opcode entscheiden, welche Gruppierung die Operation besitzt und welche weiteren Bits ausgelesen und eingeteilt werden müssen.

Abbildung 42 UML InstructionDecoder

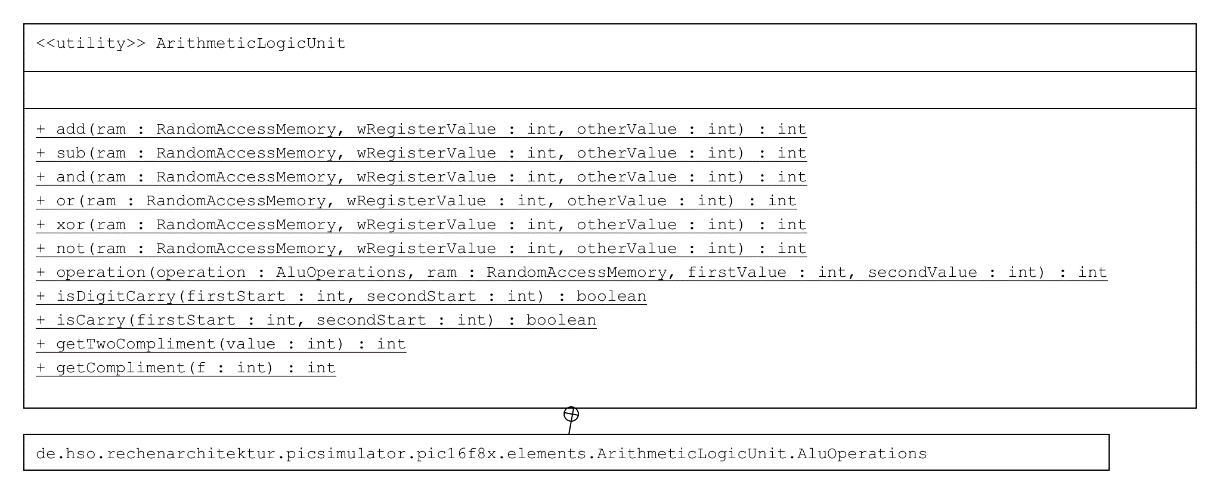
Abbildung 43 Codeausschnitt Einteilung der genauen Befehle

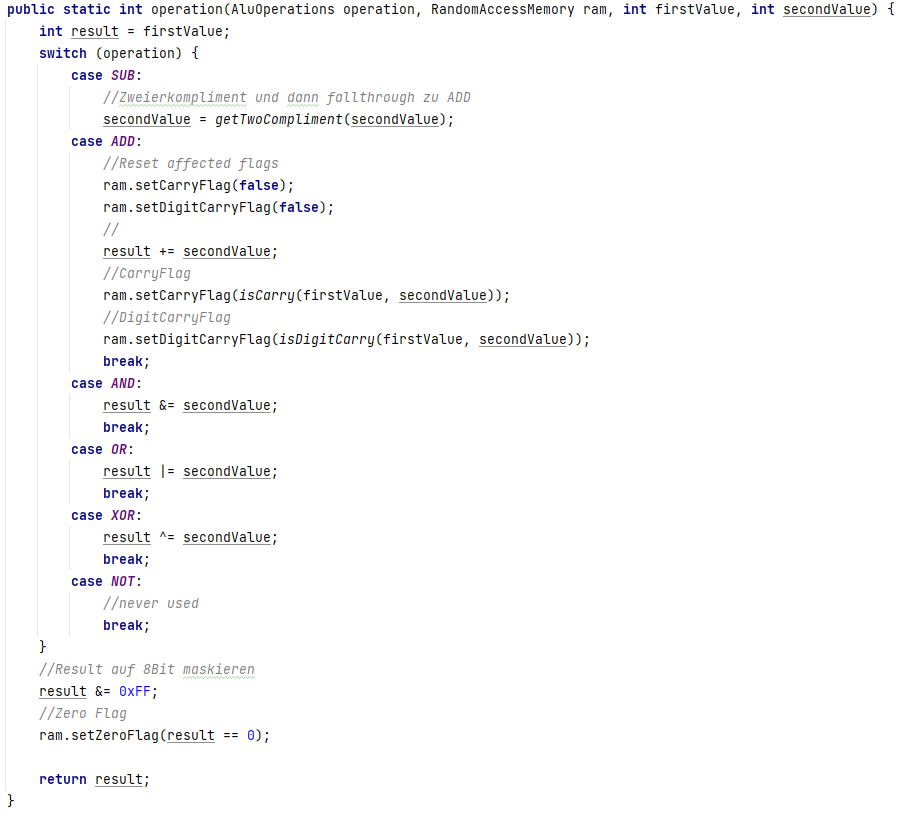
Abbildung 44 Codeausschnitt der Gruppierung der Instruktionen

Die List mit den allen Befehlen und den zugehörigen Opcode steht im Anhang. Zuerst wird der Opcode zu der Gruppierung der Operationen ausgewertet und weitergeleitet. Zum Beispiel nehmen wir hier den Opcode 0b0110\_1010\_1111. Dieser würde, da die höchstwertigen Bits 01 sind, als Bit orientierte File Register Operation eingeteilt werden.

Danach werden in den jeweiligen Gruppen mit einer spezifisichen Bitmaske die genauen Befehle ermittelt. In unserem Beispiel sind die Bits an der Stelle 7-9, von dem minderwertigsten und ab Null gezählt, 11, was bedeutdet, dass unser Befehl ein BTFSS ist. Danach müssen nurnoch die restlichen Bits korrekt gruppiert ausgelesen werden. Bei dem BTFSS stehen die Bit 0-6 für f, also die Adresse welche der Befehlt testen soll. Unser Beispiel ist also ein BTFSS, welches das Bit 0b101, also 5 in der Adresse 0b10\_1111 also 47 auf gesetzt überprüft und falls Wahr den nächsten Befehl durch einen NOP austauscht.

## Arithmetische Logische Einheit (ALU)

Die Arbeitsklasse ALU wird mit der Operation, dem RAM und den nötigen Werten aufgerufen. Durch eine Switch wird entschieden was wie bearbeitet werden soll. Bei der Subtraktion wird der zweite Wert in durch das Zweierkomplement ersetzt und wie eine Addition normal fortgeführt. Da es einen Fehler in der Konstruktion des Microcontrollers gibt, welche auch hier eingebaut wurde, werden keine extra Vorkehrungen bei der Subtraktion getroffen. Die Flags werden vom ALU direkt in den RAM gesetzt.



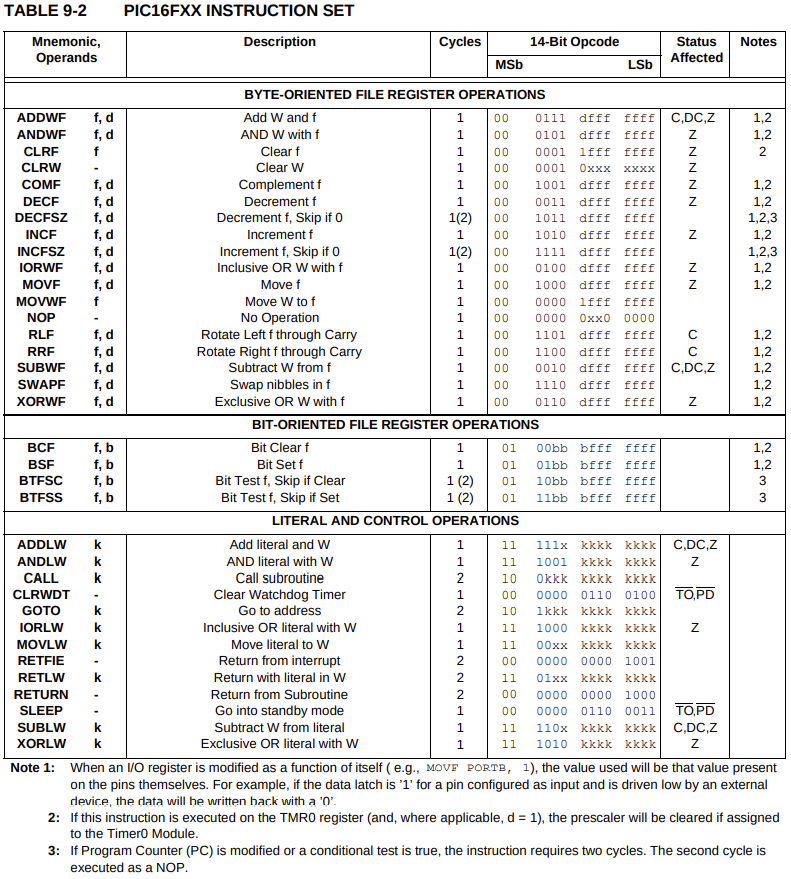
# Fazit

Zu Beginn des Projektes war uns nicht klar, wie wir dieses beginnen oder fertigstellen sollten. Es war alles sehr neu und zu komplex um direkt alles zu verstehen. Nachdem wir das Projekt in kleine Aufgaben eingeteilt hatten und einige von diesen dann auch schnell erledigen konnten, sah es machbarer aus. Das größte Problem war unser Unwissen, wie der PIC funktioniert. Somit konnte nicht wirklich exakt geplant werden. Was bekannt war wurde entwickelt und falls neue Erkenntnisse einkamen wurden oft alte Implementierungen verworfen oder mussten stark abgeändert werden. Es war ein ständiges lernen und anpassen in während der Laufzeit des Projektes. Java war keine Erschwernis für uns, nur hatten wir beide noch nie ein GUI in Java erstellt und mit Hilfe von Swing und dem „Intellij-WindowBuilder“ haben wir es zwar geschafft ein funktionsfähiges GUI zu bauen, aber haben wir dort die „best practice“ und gedachte Anwendung nicht eingehalten. Dies hat zu Beginn aber den Fortschritt sehr zurückgehalten. Zuerst wollten wir den Simulator 1:1 wie das Original realisieren. Dies stellte sich aber als ineffizient dar. Deswegen fingen wir frisch an. Dort wurden dann viele Funktionen des PICs abstrahiert um diese in Java um einiges schneller realisieren zu können. Das Fehlen einer genaueren Planung war ein Segen und Fluch zugleich. Eine Grobplanung welche festlegte, wie im Allgemeinen alles Funktionieren sollte, was für Klassen es gab und wie die GUI aussehen sollte hatten wir uns erstellt. Genauere Spezifikationen über Methoden und Attribute hatten wir nicht abgemacht. Dadurch konnten wir bei neuen Erkenntnissen oder Ideen flexibel alles anpassen. Die generelle Koordination war aber sehr erschwert, was bei einem kleinen Team nicht so schlimm wie bei Größeren ist. Nachdem die Grundfunktionen des Simulators implementiert waren, lief der restliche Entwicklungsprozess relativ schnell und leicht ab. Bei dem Verständnis des Timers und WatchDog hatten wir zu Beginn Schwierigkeiten, welche aber durch exaktes Recherchieren in dem Datenblattes des Herstellers, gelöst wurde. Vieles wurde später dennoch umgeschrieben oder gelöscht, da durch neue Ideen alte Funktionen obsolet wurden. Einiges konnte aber nicht angepasst werden, da es schon zu tief in der Implementation verankert war und der Aufwand alles anzupassen nicht Wert war. Hier zum Beispiel kommt der RAM, welcher als 128 langes 2d-Array verwirklicht wurde, um die Banken einfacher zu erkennen und zu simulieren. An vielen Stellen mussten aber die Zwei Dimensionen des Arrays hintereinander geknüpft werden um korrektes Verhalten zu erlangen.

Wenn wir das Projekt erneut realisieren würden, würden wir alles zu Beginn genauer Planen und festlegen. Da wir dann auch schon die korrekte Funktion des PICs kennen, wäre dies auch ohne Probleme möglich.

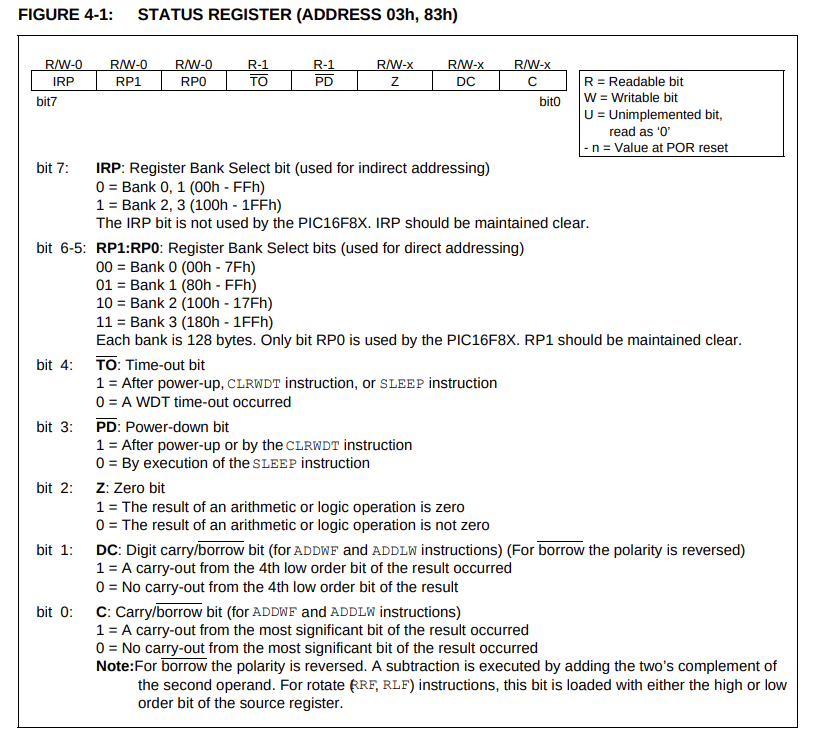
Anhang

## Befehlsliste

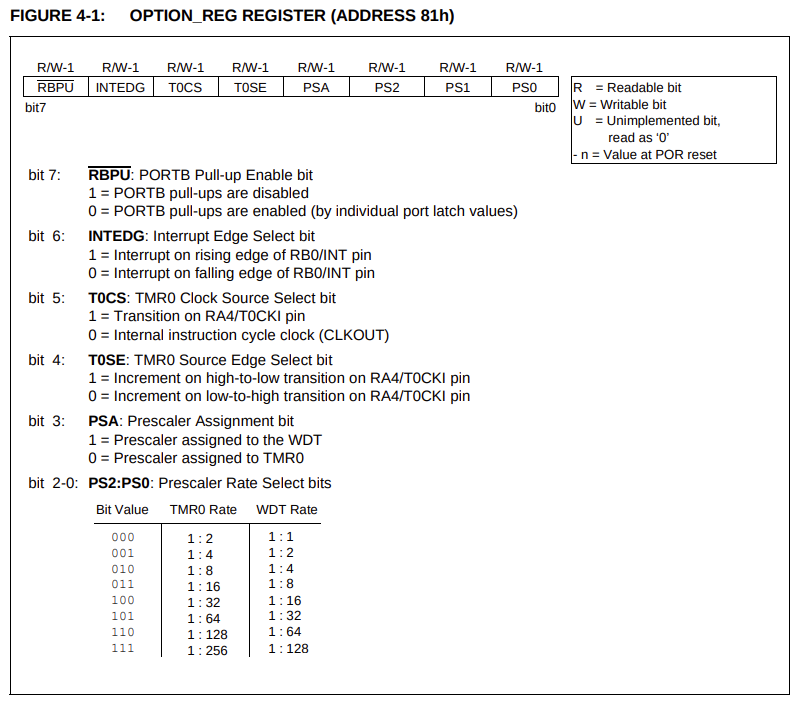
[[3]](#footnote-3)

## SFR Bits

### Status Register

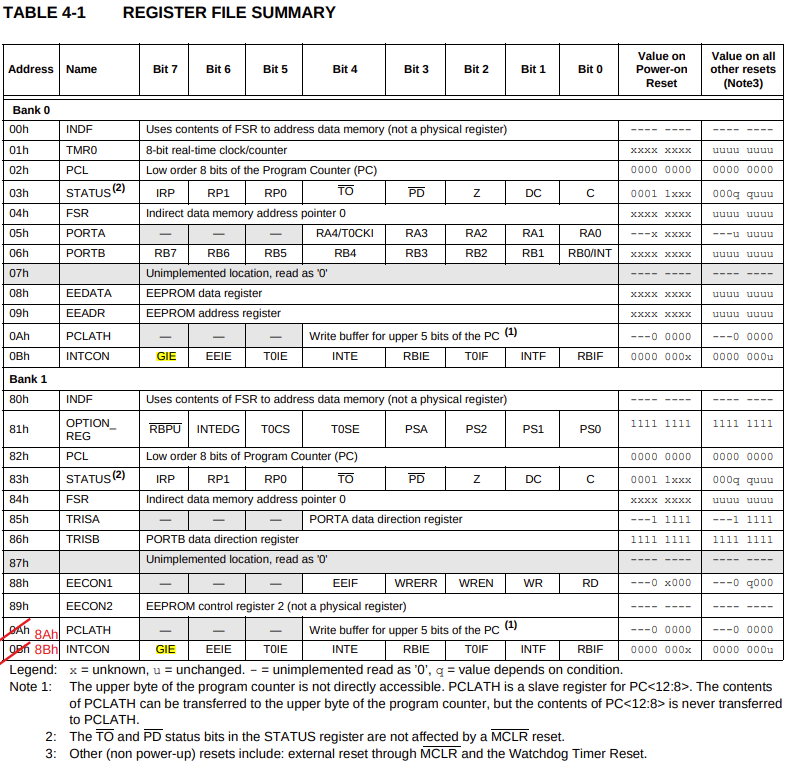
[[4]](#footnote-4)

### Option Register

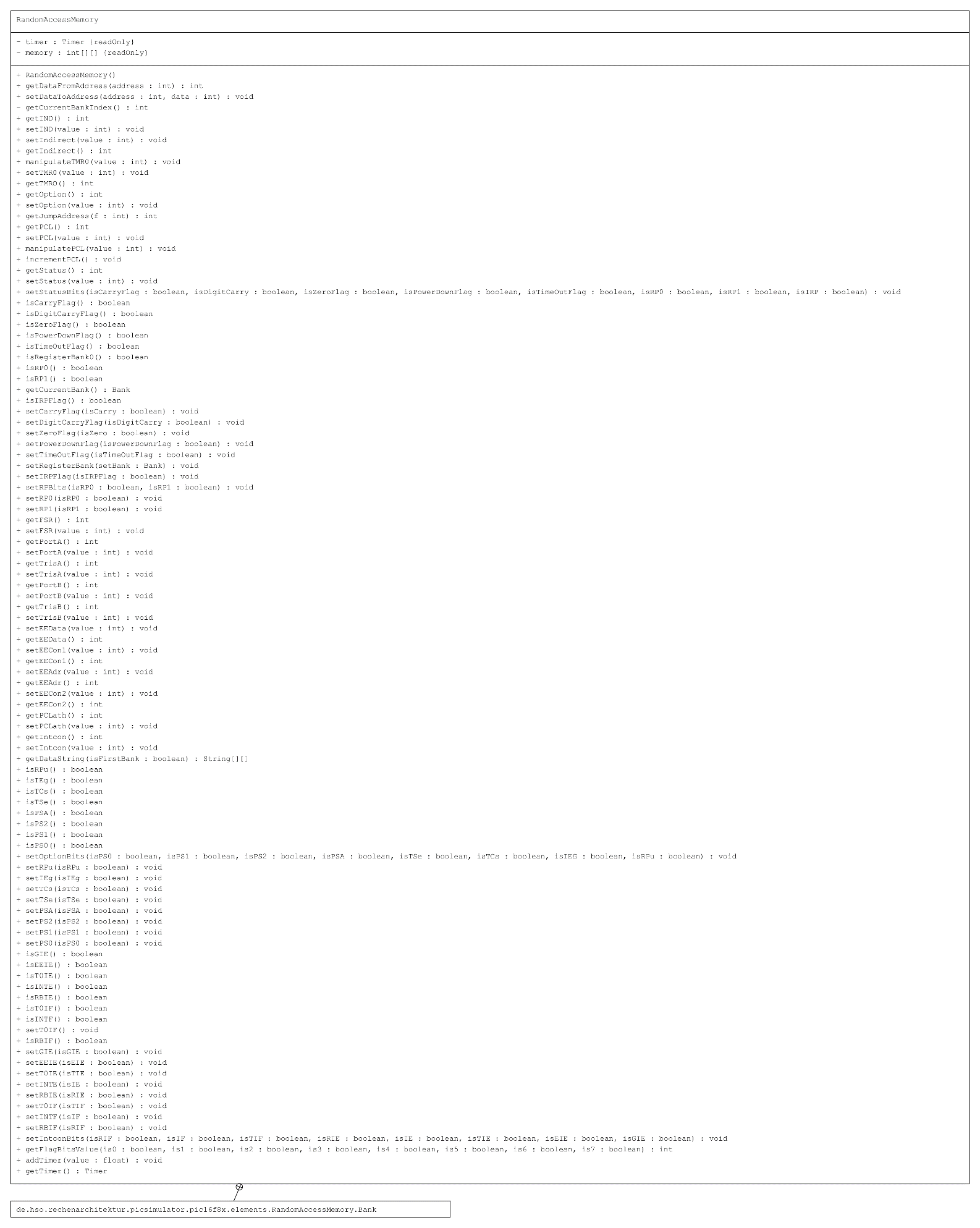
[[5]](#footnote-5)

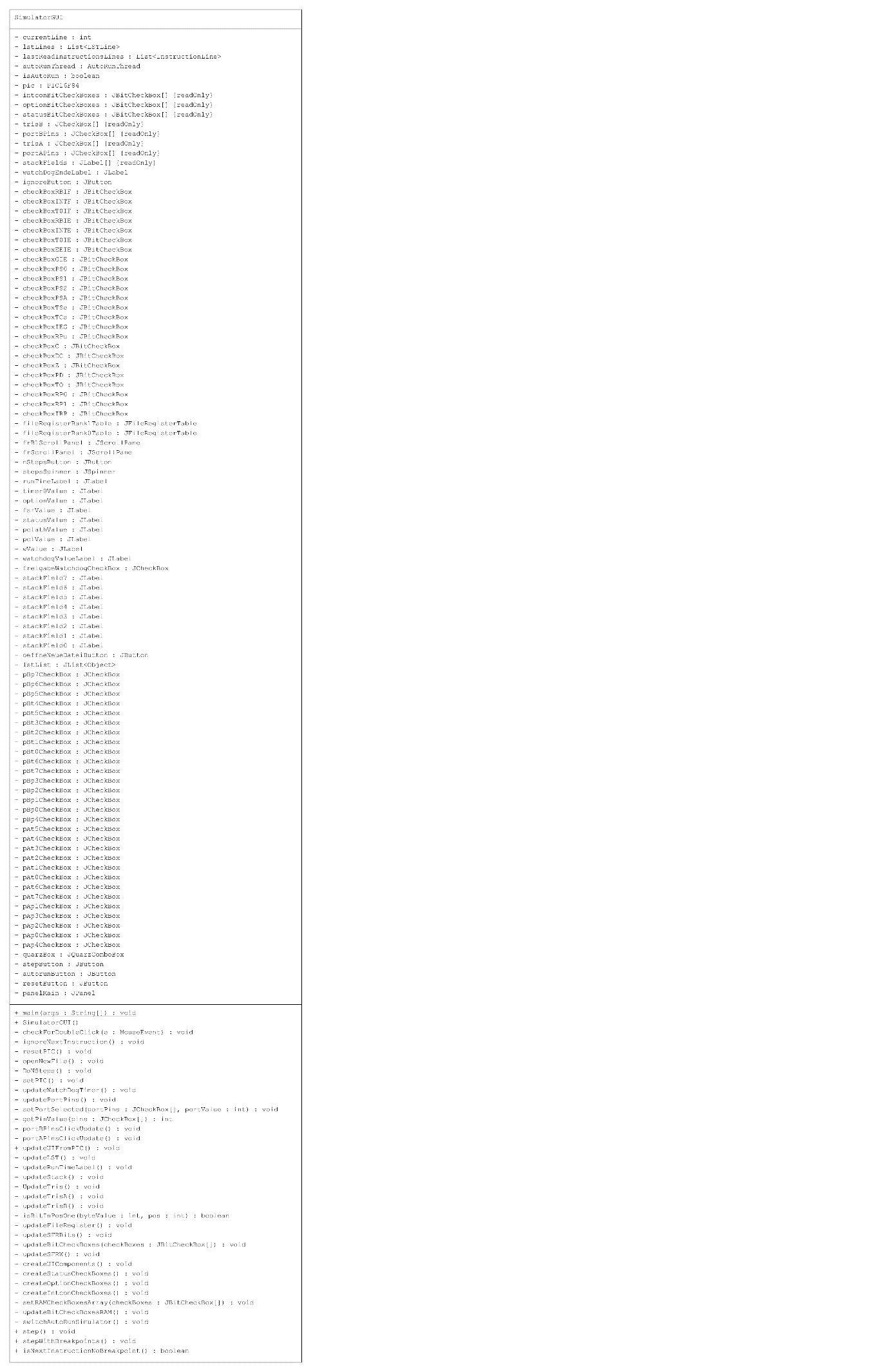
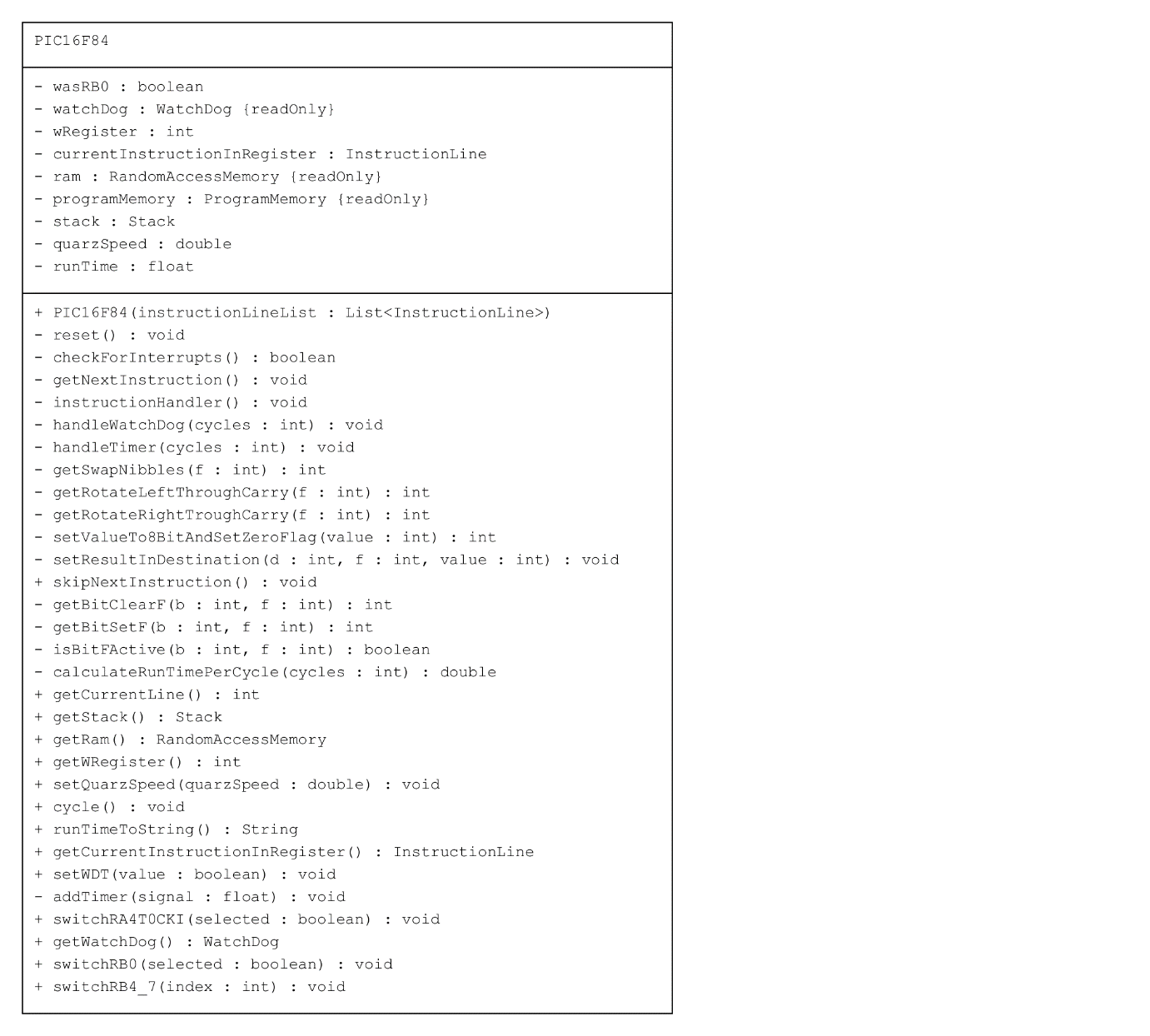
### [[6]](#footnote-6)INTCON Register

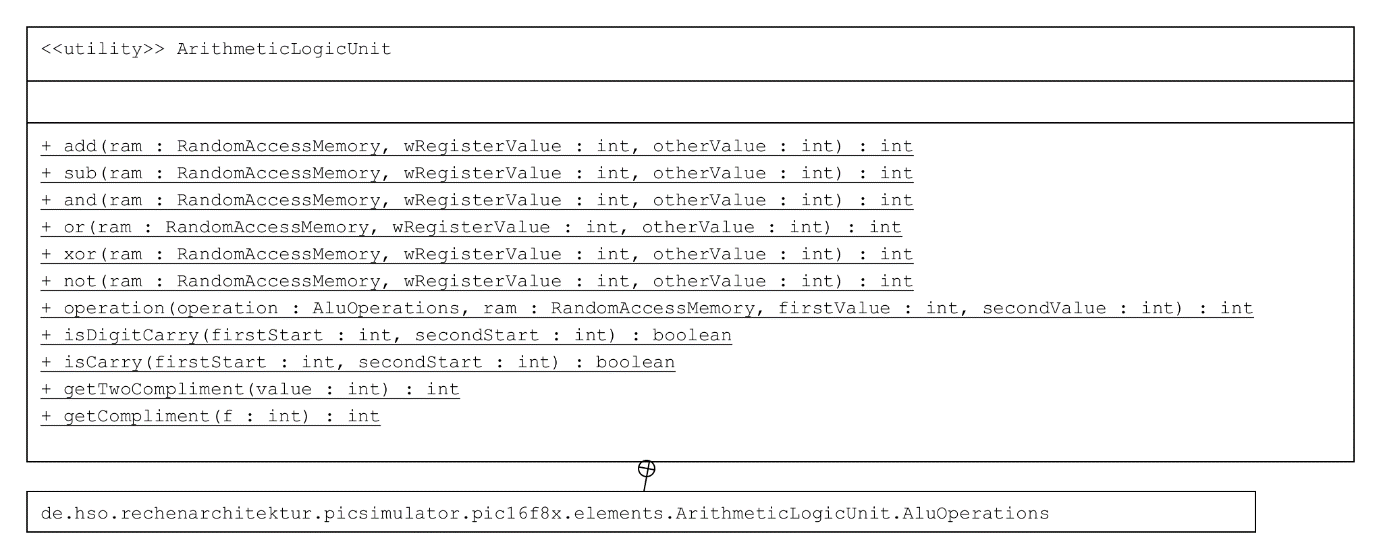
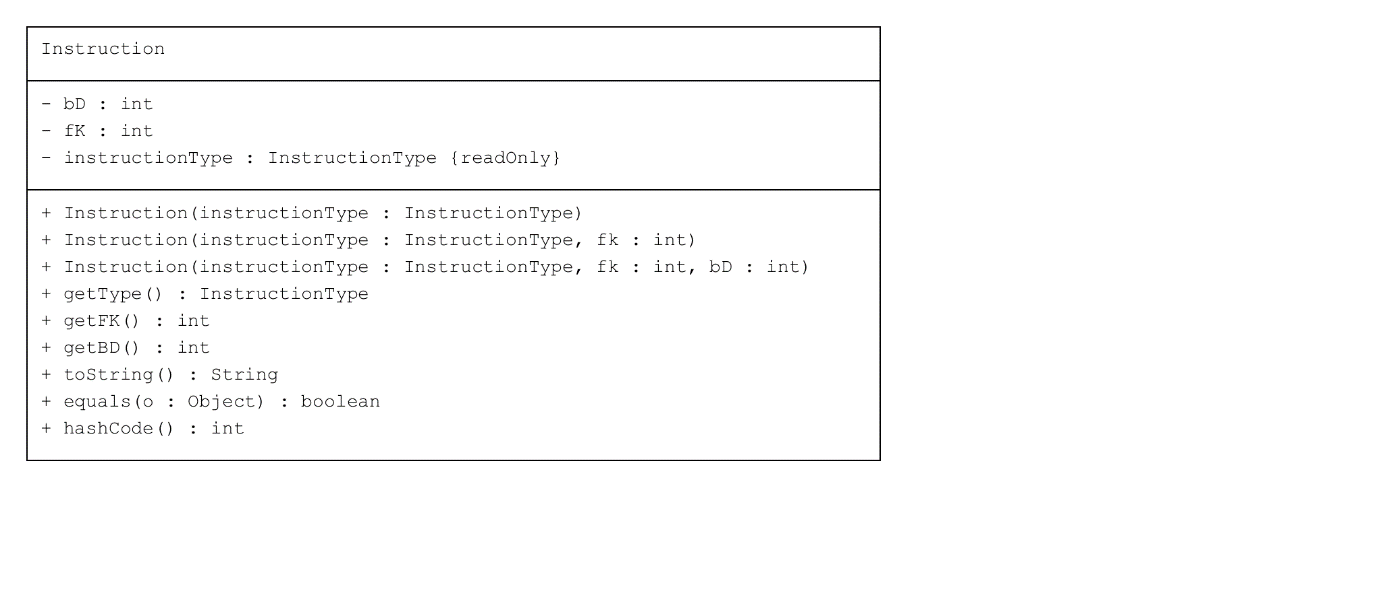
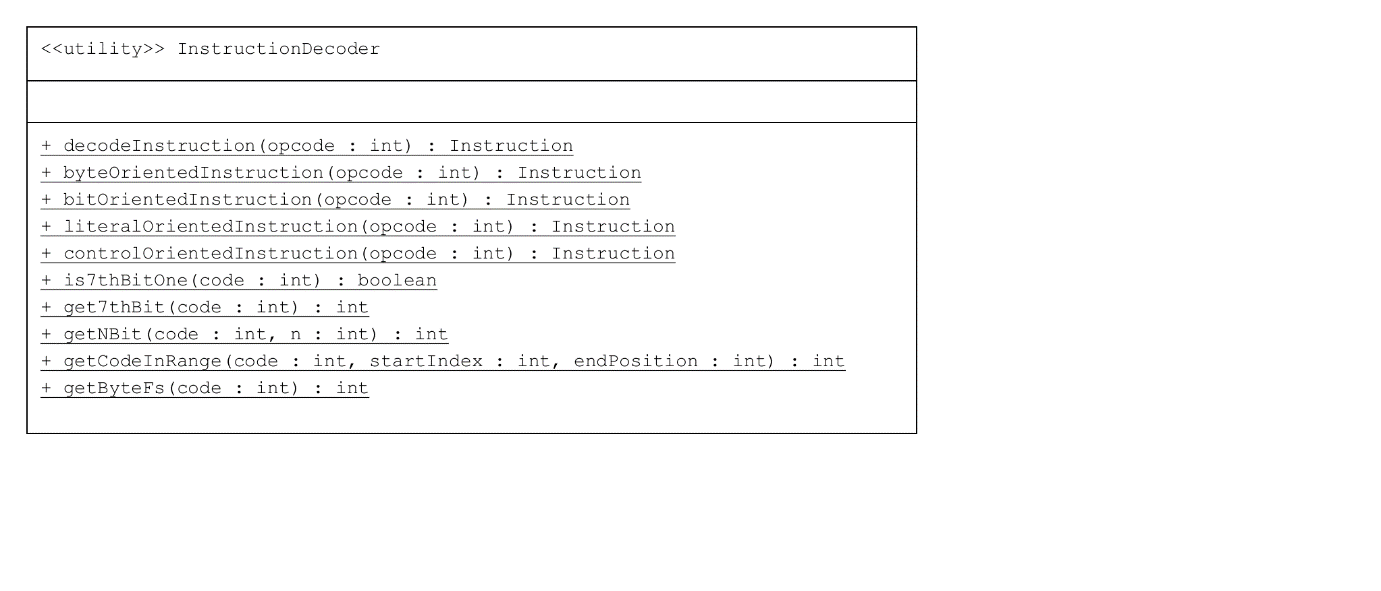
## Fileregister

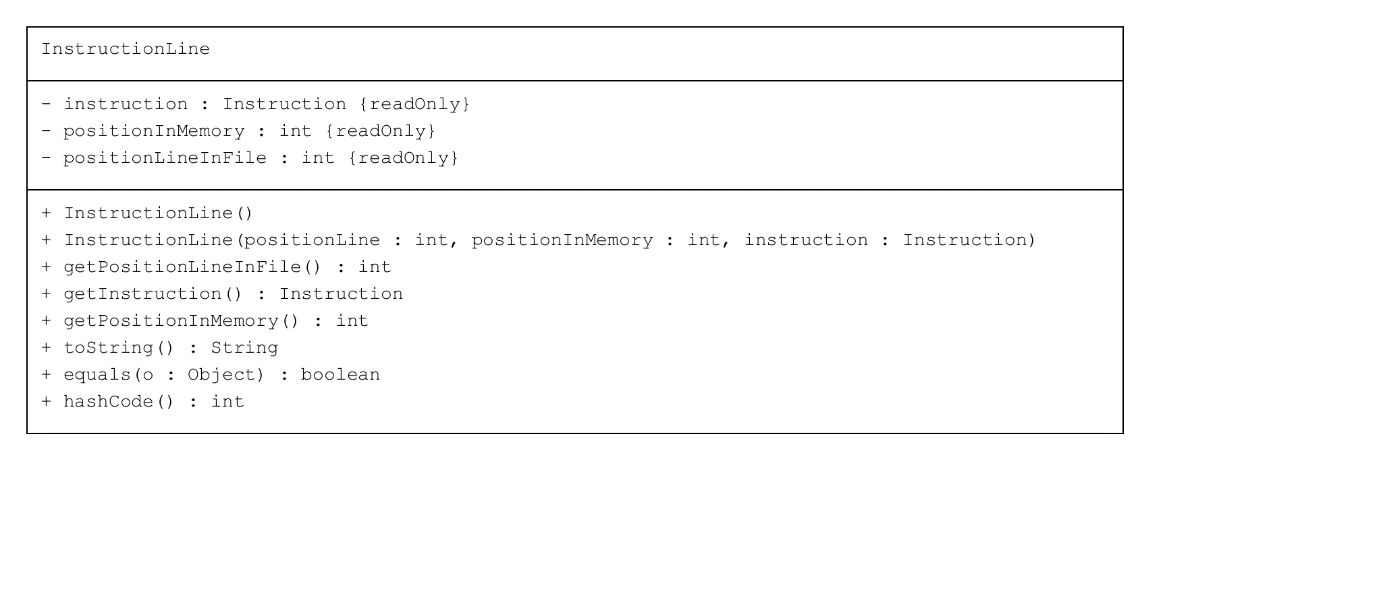
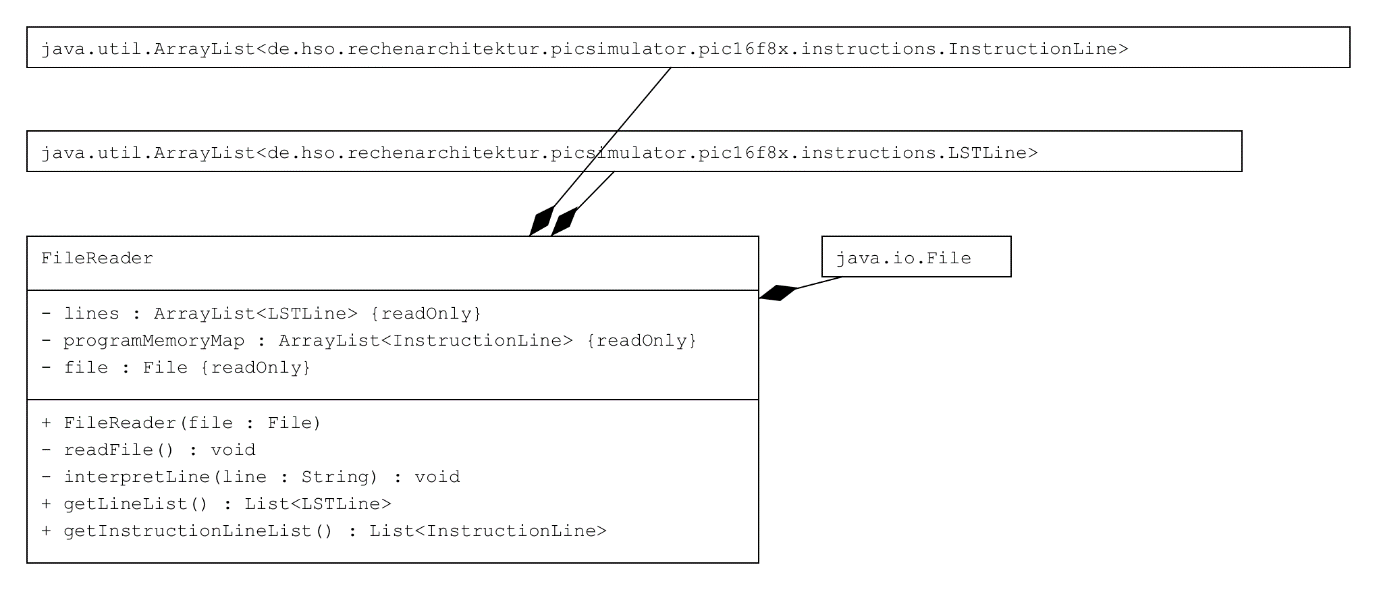
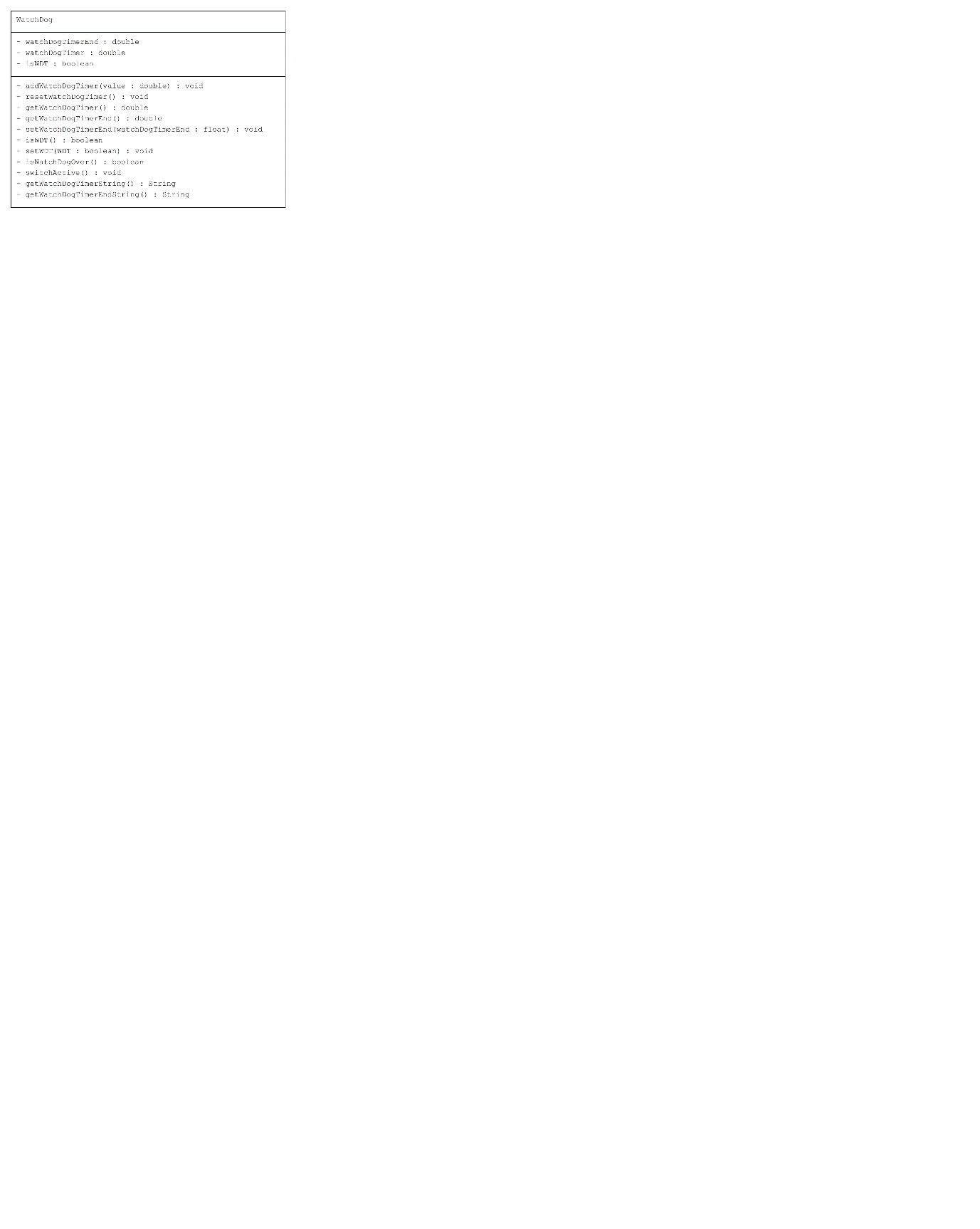
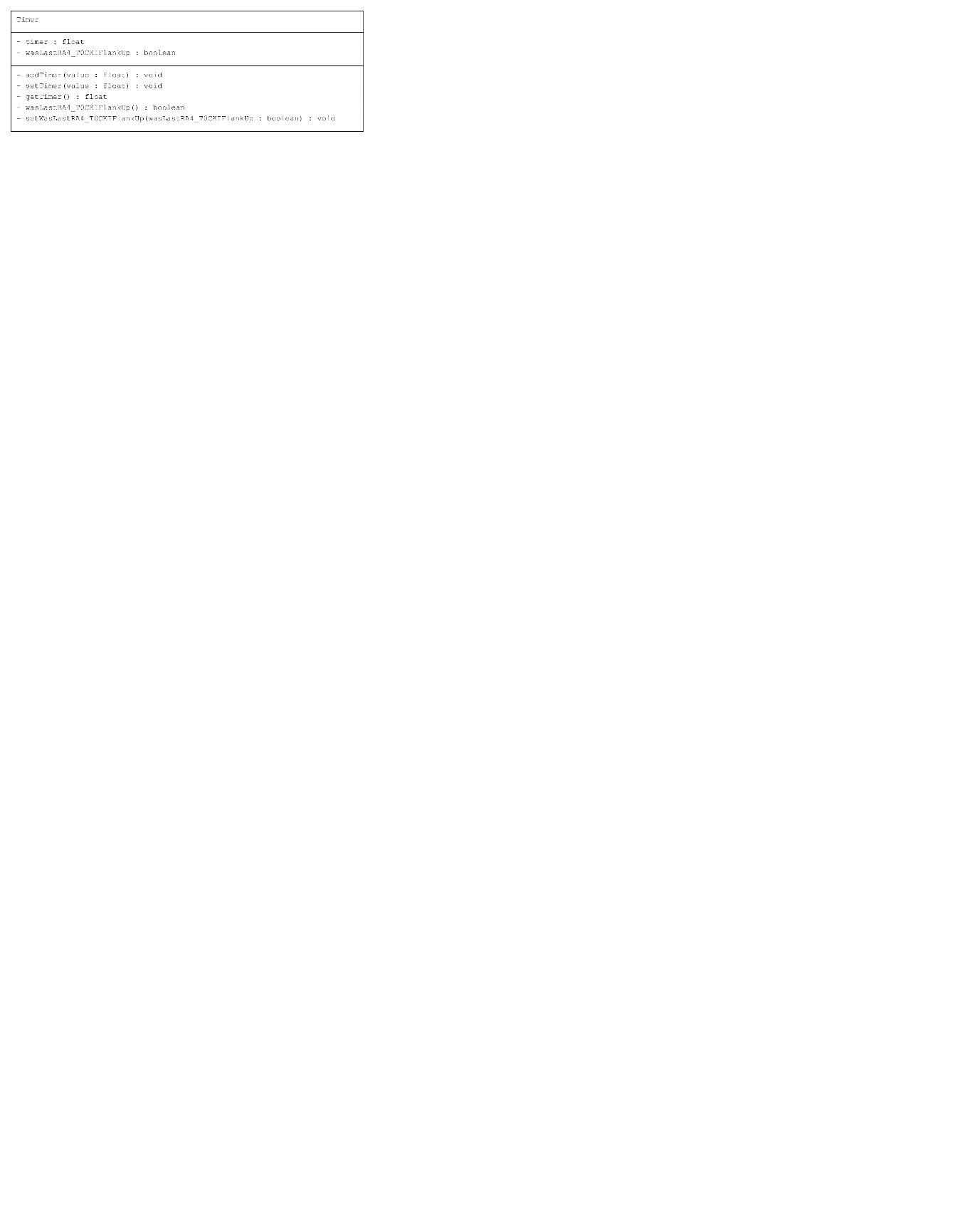
[[7]](#footnote-7)

## Klassen Diagramme









1. Bild von https://articulo.mercadolibre.com.ar/ 13.Mai.2021 [↑](#footnote-ref-1)
2. Interrupt Logic aus dem Datenblatt PIC16F8X von Microchip [↑](#footnote-ref-2)
3. Datenblatt PIC16F8X von Microchip [↑](#footnote-ref-3)
4. Datenblatt PIC16F8X von Microchip [↑](#footnote-ref-4)
5. Datenblatt PIC16F8X von Microchip [↑](#footnote-ref-5)
6. Datenblatt PIC16F8X von Microchip [↑](#footnote-ref-6)
7. Datenblatt PIC16F8X von Microchip [↑](#footnote-ref-7)