|  |
| --- |
|  |
| Dokumentation des PIC16C84 Simulators |
| Studiengang Informationstechnik an der  Dualen Hochschule Baden-Württemberg Karlsruhe |
|  |
| **Benedikt Bock, Mario Waxenegger** |
| **19.05.2014** |

|  |
| --- |
|  |

# Realisierung

## Wahl der Programmiersprache und des Grundkonzepts

Bei der Wahl der Programmiersprache für dieses Projekt fiel unsere Wahl auf C++. Diese Sprache haben wir bereits theoretisch im 2. Semester erlernt. Das Projekt bot nun die Möglichkeit auch praktische Erfahrungen mit dieser Sprache zu sammeln.

Durch die Vorlesung Software Engineering I im 3. Semester haben wir bereits Einblicke in das Themenfeld Entwurfsmuster bekommen. Dies führte zu der Idee den Simulator gemäß dem Model-View-Controller-Entwurfsmuster (MVC-Entwurfsmuster) zu strukturieren. Dies hat zur Folge, dass die GUI als eigenständiges Modul aufgefasst wird. Dieses Entwurfsmuster wurde mit Sicherheit, vor allem bei der Trennung von Model und Controller, nicht in Gänze umgesetzt, da dies einen immensen Aufwand nach sich gezogen hätte und nicht in Relation zum eigentlich Projektziel stand.

## Programmstruktur

Das Programm besteht im Wesentlichen aus drei Klassen:

* Mainwindow Klasse
* Steuwerwerk Klasse
* Prozessor Klasse

Die Mainwindow Klasse verwaltet hierbei die GUI und enthält alle nötigen Methoden um die GUI aktuell zu halten. Außerdem enthält die Klasse eine Referenz auf ein Steuerwerkobjekt. Die Steuerwerkklasse übernimmt zum einen die Aufgaben des Controllers aus dem MVC-Entwurfsmuster und zum anderen grob die Aufgaben des Steuerwerks des PIC. Die Mainwindow Klasse ruft Methoden des Steuerwerks auf. Wenn die Steuerwerk Klasse Methoden ausführt, die Inforamtionen ändert, welche für die GUI wichtig sind, dann wird die Mainwindow Klasse benachrichtigt. Diese führt dann entsprechende Aktualisierungen aus.  
Die Prozessor Klasse stellt die Funktionalität der ALU des PIC nach. Im Wesentlichen werden hier die eigentlichen Befehle ausgeführt. Das Steuerwerk enthält ein Prozessorobjekt.

Es gibt noch ein paar zusätzliche Klassen, die im Folgenden aufgeführt werden. Die Prozessor Klasse beinhaltet ein Objekt der Speicher Klasse. Diese Klasse stellt den Speicher (incl. EEPROM) des PIC dar.   
Zusätzlich gibt es eine Codeline Klasse. Ein Objekt dieser Klasse repräsentiert eine Codezeile des PIC-Programms. Dazu werden in dem Objekt die Zeile selbst, die dezimale Befehlsrepräsentation, sowie ein boolescher Wert gespeichert. Der boolesche Wert gibt an ob auf diese Zeile ein Breakpoint gesetzt wurde. Organisiert werden die Objekte in einem Vektor, der im Steuerwerkobjekt hinterlegt ist. Erzeugt wird der Vektor durch eine weitere Klasse namens Parser. Diese Klasse stellt nur eine statische Methode zur Verfügung. Diese Methode liest die angegebene LST-Datei zeilenweise ein und erzeugt die Codelineobjekte.

Abschließend gibt es noch zwei weitere Klassen. Diese stellen allerdings keine eigenständige Funktionalität dar. Viel mehr beschreiben die beiden Klassen Funktionen des Simulators die in einem separaten Thread laufen müssen um die GUI flüssig zu halten. Die eine Klasse heißt Goklasse. Diese implementiert den „Go“ Button (siehe Kapitel goButton). Die andere Klasse ist die Sleep Klasse. Diese implementiert den Befehl „sleep“ (siehe Kapitel sleep).

Im Folgenden werden die wesentlichen Abläufe des Simulators dargestellt. Dabei ist zu beachten, dass hier nicht weiter auf die GUI eingegangen wird. In den Programmablaufplänen sind die Benachrichtigungen an die GUI nicht berücksichtigt.

### goButton

Wird der go Button gedrückt, so wird ein Objekt der GoKlasse erstellt und die Methode *run()* in einem separaten Thread ausgeführt. Abbildung 1 zeigt den PAP der *run()* Methode. In Listing 1 ist ab Zeile 18 Code zu sehen, der eine Verzögerung in die Schleife einbaut. Hintergrund ist, dass während *executeStep()* die GUI mehrmals Benachrichtigt wird. Die GUI braucht hier ein bisschen Zeit um die Benachrichtigungen abzuarbeiten. Zudem wird im Listing *isRunning* nicht direkt auf false gesetzt. Dies geschieht u.a. mit dem Aufruf *emit slotGoClicked()*.

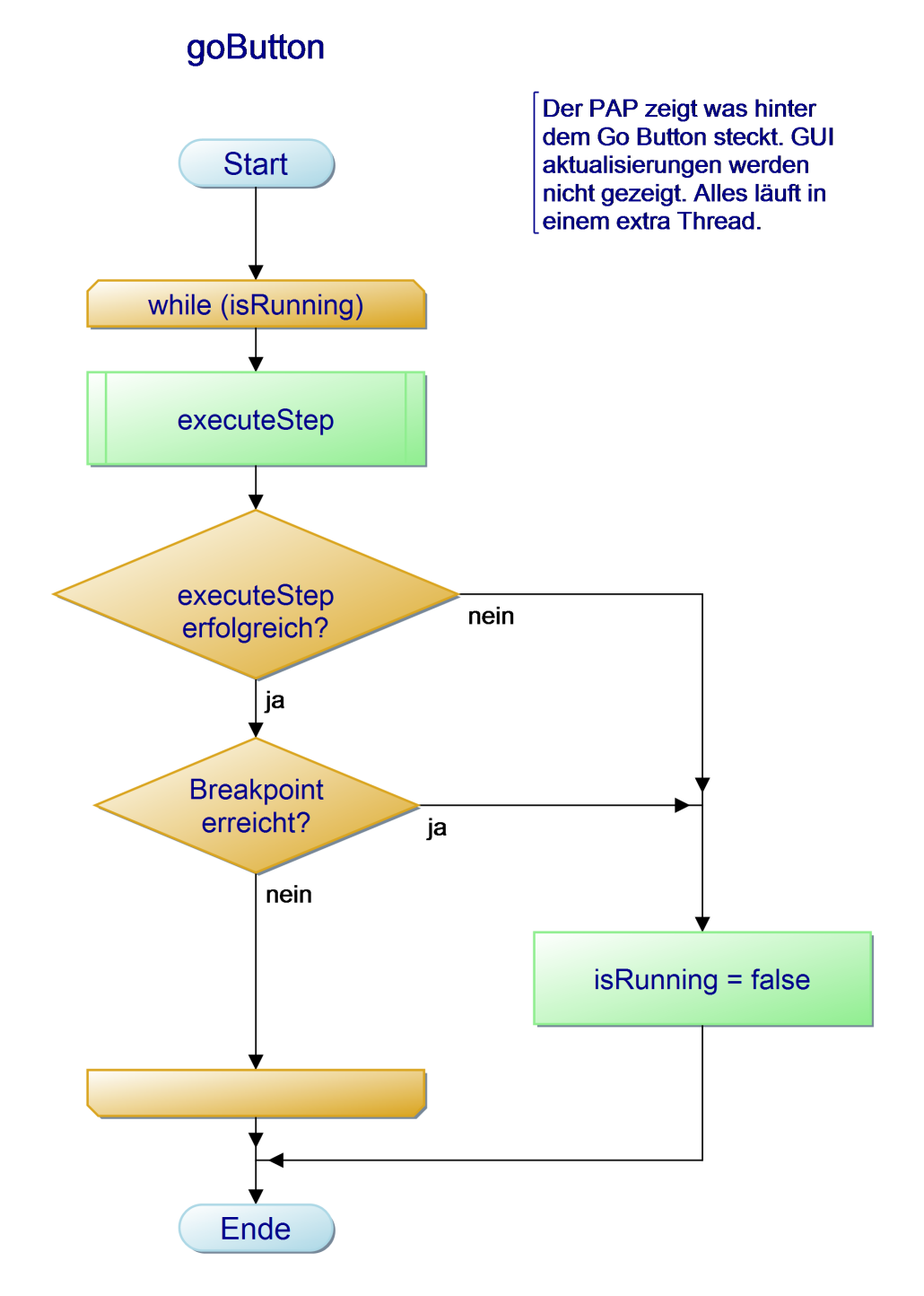


Abbildung 1: PAP GoKlasse::run()

|  |  |
| --- | --- |
|  | void GoKlasse**::**run**()** |
|  | **{** |
|  | **while(**sW**->**isRunning**)** |
|  | **{** |
|  | **if(!**sW**->**executeStep**())** |
|  | **{** |
|  | emit slotGoClicked**();** |
|  | **return;** |
|  | **}** |
|  |  |
|  | //Stop falls breakpoint erreicht |
|  | **if(**sW**->**pc**->**breakpoint**)** |
|  | **{** |
|  | emit slotGoClicked**();** |
|  | **return;** |
|  | **}** |
|  |  |
|  | //Pause |
|  | int v**=**50**;** // falls ds Signal nicht schnell genug verarbeitet wird -> mindestverzögerung 50ms |
|  | emit getVerzoegerung**(&**v**);** |
|  | QThread**::**msleep**(**50**);** // warte darauf, dass signal abgearbeitet wurde |
|  | QThread**::**msleep**(**v**-**50**);** |
|  | **}** |
|  | **}** |

Listing : GoKlasse::run()

### executeStep

Diese Funktion führt einen Befehl des Programms aus. Abbildung 2 zeigt den PAP der Methode. Der PAP ist selbsterklärend. Die Funktionsaufrufe (z.B. Interrupthandler) werden im Folgenden aufgeführt. Im Anschluss an den PAP ist der Programmcode zu finden. Funktionsaufrufe mit dem vorangestellten Schlüsselwort „emit“ stellen Benachrichtigungen an die GUI dar.  
Aufgerufen wird diese Methode zum einen durch das Mainwindow wenn der „Execute Step“ Button gedrückt wird und zum anderen zyklisch durch die Goklasse. Siehe hierzu Kapitel goklasse

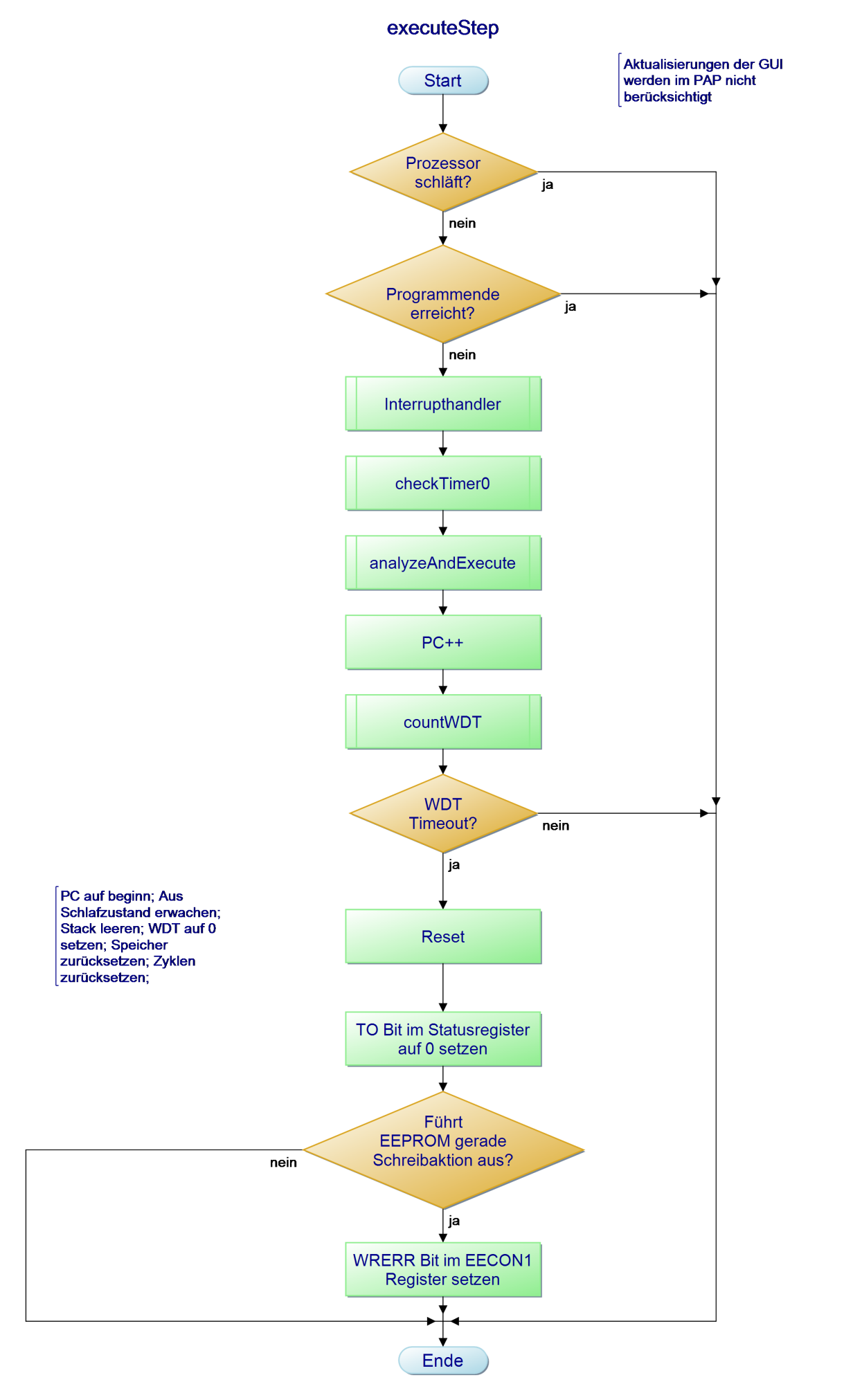


Abbildung : PAP Steuerwerk::executeStep()

|  |  |
| --- | --- |
|  | /\* |
|  | \* Gibt false zurück, wenn PC nicht mehr auf Codezeile zeigt, |
|  | \* d.h. wenn das Programm am Ende angelangt ist. |
|  | \* |
|  | \* Zeigt der PC auf einen gültigen Befehl, wird true zurückgegeben. |
|  | \*/ |
|  | bool Steuerwerk**::**executeStep**(**void**)** |
|  | **{** |
|  | **if(**isSleeping**())** |
|  | **return** **false;** |
|  | **if(**programmEndeErreicht**())** |
|  | **return** **false;** |
|  |  |
|  | testForInterrupt**();** |
|  | **if(**pc**->**breakpoint**)** |
|  | emit setLineColorRed**(**getCurrentLineNumber**()-**1**);** |
|  | **else** |
|  | emit setLineColorWhite**(**getCurrentLineNumber**()-**1**);** |
|  |  |
|  | **if(**pc **!=** maschinencode**.**end**())** |
|  | **{** |
|  | //Ursprünglichen Cycle Wert für Watchdog speichern |
|  | alu**->**vorherigeCycles **=** alu**->**cycles**;** |
|  | checkTimer0**();** |
|  | analyzeAndExecute**(**pc**->**command**);** |
|  | pc**++;** |
|  | alu**->**speicher**.**writePC**(**pc **-** maschinencode**.**begin**());** |
|  | countWDT**();** |
|  | **}** |
|  | **else** |
|  | **return** **false;** |
|  | emit refreshStorageGUI**();** |
|  |  |
|  | **if(**isWDTTimeOut**())** |
|  | **{**//WDT reset, Power on Reset |
|  | emit reset**();** |
|  | alu**->**speicher**.**writeOnBank**(**0**,**3**,**0x0008**);** //TO Bit im Status register clearen |
|  | **if(**int currentEECON1**=**alu**->**speicher**.**readOnBank**(**1**,**8**)&**0x0002**)**// Führt eeprom gerade ein schreibaktion durch? |
|  | alu**->**speicher**.**writeOnBank**(**1**,**8**,**currentEECON1 **|=** 0x0008**);** //-> WRERR bit (EECON1) setzen |
|  | emit refreshStorageGUI**();** |
|  | **return** **false;** |
|  | **}** |
|  |  |
|  | **if(!**programmEndeErreicht**())** |
|  | **{** |
|  |  |
|  | emit setLineColorGreen**(**getCurrentLineNumber**()-**1**);** |
|  | emit gotoLineNumber**(**getCurrentLineNumber**()-**1**);** |
|  | **}** |
|  | **else** |
|  | **{** |
|  | //GoKlasse muss beendet werden |
|  | **return** **false;** |
|  | **}** |
|  |  |
|  | **return** **true;** |
|  | **}** |

Listing : Steuerwerk::executeStep()

### Interrupthandler

Die Funktion Steuerwerk::testForInterrupt() entspricht dem Interrupthandler. Diese Funktion wird in Abbildung 3 als PAP dargestellt und Listing 3 zeigt den entsprechenden Auszug aus dem Programm. Zunächst wird geprüft ob Interrupts global aktiviert sind. Ist dies der Fall, so werden die einzelnen Interrupts des PIC (External RB0 Pin, TMR0 Overflow, PORTB change und EEPROM write complete) geprüft. Ist einer der Interrupts aktiviert und das entsprechende Flagbit gesetzt, so wird die Funktion callInterrupt aufgerufen. Näheres dazu, siehe weiter hinten im Kapitel.

Die Flags werden bei den entsprechenden Interrupts gesetzt.

Hier muss noch realisierung der Flags geschrieben werden. slotRBValueChanged() (<- RB0 und RB changed) checkTimer0 für TMR0 und EEPROM Write für EEPROM Interrupt



Abbildung : PAP Steuerwerk::testForInterrupt()

|  |  |
| --- | --- |
|  | void Steuerwerk**::**testForInterrupt**()** |
|  | **{** |
|  | int intcon **=** alu**->**speicher**.**readOnBank**(**0**,**0x0b**);** //Intconregister lesen |
|  | **if(**intcon **&** 0x0080**)** //ist das GIE Bit gesetzt? |
|  | **{** |
|  | **if((**intcon**&**0x0020**)&&(**intcon**&**0x0004**))** //sind T0IE unf T0IF gesetzt? |
|  | **{** |
|  | callInterrupt**();** |
|  | **return;** |
|  | **}** |
|  | **if((**intcon**&**0x0010**)&&(**intcon**&**0x0002**))** //sind INTE unf INTF gesetzt? |
|  | **{** |
|  | callInterrupt**();** |
|  | **return;** |
|  | **}** |
|  | **if((**intcon**&**0x0008**)&&(**intcon**&**0x0001**))** //sind RBIE unf RBIF gesetzt? |
|  | **{** |
|  | callInterrupt**();** |
|  | **return;** |
|  | **}** |
|  | **if((**intcon**&**0x0040**)&&(**alu**->**speicher**.**readOnBank**(**1**,**0x08**)&**0x0010**))** //sind EEIE unf EEIF (in EECON1 Register) gesetzt? |
|  | **{** |
|  | callInterrupt**();** |
|  | **return;** |
|  | **}** |
|  | **}** |
|  | **}** |

Listing : Steuerwerk::testForInterrupt()

**callInterrupt**

Wie bereits erwähnt ruft diese Funktion die eigentliche Interruptroutine des Programms auf. Dazu werden zunächst alle Interrupts global deaktiviert. Anschließend wird der PC auf den Stack gepusht. Abschließend wird Adresse 4 in den PC geladen. Listing 4 zeigt den entsprechenden Code. Der Sprung ist hierbei mit dem call Befehl des PIC umgesetzt. Besondere beachtung gilt Zeile 12. Hier wird der PC dekrementiert. Dies hängt damit zusammen, dass während der call Routine der PC erhöht wird. Man möchte normalerweise an den nachfolgenden Befehl zurück springen. Bei einem Interrupt ist das anders. Hier will man an den Befehl zurück springen, an dem der Interrupt eingetreten ist. Daher Zeile 12.

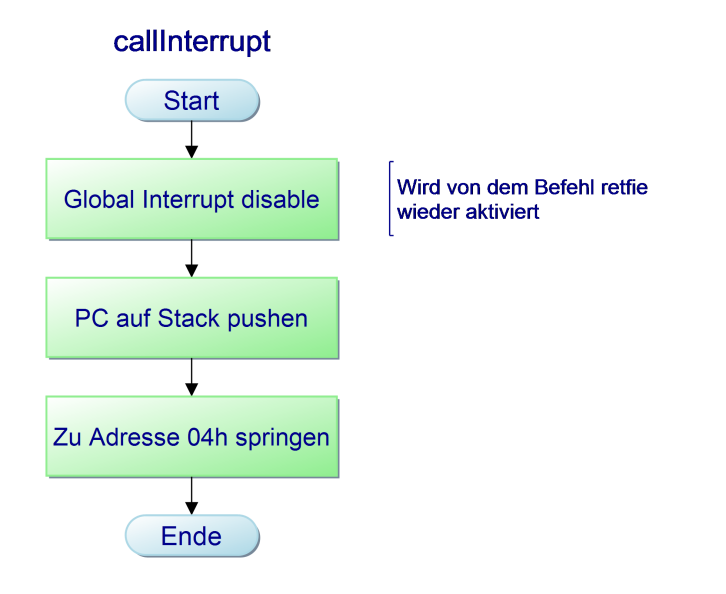


Abbildung : PAP Steuerwerk::callInterrupt()

|  |  |
| --- | --- |
|  | void Steuerwerk**::**callInterrupt**()** |
|  | **{** |
|  | //GIE deaktivieren |
|  | int newValue **=** alu**->**speicher**.**readOnBank**(**0**,**0xb**)** **&** 0x007f**;** |
|  | alu**->**speicher**.**writeOnBank**(**0**,**0xb**,**newValue**);** |
|  | //aktuelle Zeile deaktivieren |
|  | **if(**pc**->**breakpoint**)** |
|  | emit setLineColorRed**(**getCurrentLineNumber**()-**1**);** |
|  | **else** |
|  | emit setLineColorWhite**(**getCurrentLineNumber**()-**1**);** |
|  | //PC-- |
|  | pc**--;** |
|  | //Call 0x0004 |
|  | alu**->**call**(**0x0004**,this);** |
|  | //neuen PC in PCL schreiben |
|  | alu**->**speicher**.**writePC**(**pc **-** maschinencode**.**begin**());** |
|  | //Befehl 0x0004 aktivieren |
|  | emit setLineColorGreen**(**getCurrentLineNumber**()-**1**);** |
|  | emit gotoLineNumber**(**getCurrentLineNumber**()-**1**);** |
|  | **}** |

Listing : Steuwerk::callInterrupt()

**Realisierung der Flags**

Die Flags werden bei entsprechenden Ereignissen gesetzt. Für den Timer0 Overflow Interrupt wird das Flag von der checkTimer0 Methode geprüft und gesetzt (siehe Kapitel checkTimer0). Ebenso wird der EEPROM write complete Interrupt vom EEPROM write Thread ausgelöst (siehe Kapitel EEPROM).

Für den RB0 und RB changed Interrupt befindet sich die auslösende Methode im Mainwindow. Die Methode heißt slotRBValueChanged und wird ausgeführt, wenn ein Wert von PORTB manuell geändert wird. Dies entspricht zwar streng genommen nicht dem MVC-Entwurfsmuster, da das Steuerwerk vom Mainwindow verändert wird, macht aber durchaus Sinn. Diese beiden Interrupts sind die einzigen beiden Interrupts, die über Pins von außen getriggert werden.

Hier muss noch realisierung der Flags geschrieben werden. slotRBValueChanged() (<- RB0 und RB changed) checkTimer0 für TMR0 und EEPROM Write für EEPROM Interrupt; Code und erklärung hier (MW)

### checkTimer0

Diese Methode implementiert die Funktionen des Timer0 (TMR0). Dabei kann der TMR0 durch einen externen Takt oder intern hochgezählt werden. Zudem kann jeweils noch ein Vorteiler dazu geschaltet werden. Für den externen Takt kann festgelegt werden, ob auf eine steigende oder fallende Flanke reagiert werden soll. Abbildung 5 zeigt den entsprechenden PAP. Listing 5 zeigt den entsprechenden Programmabschnitt. *externalClockCycles* entspricht dabei dem Takt aus dem PAP.

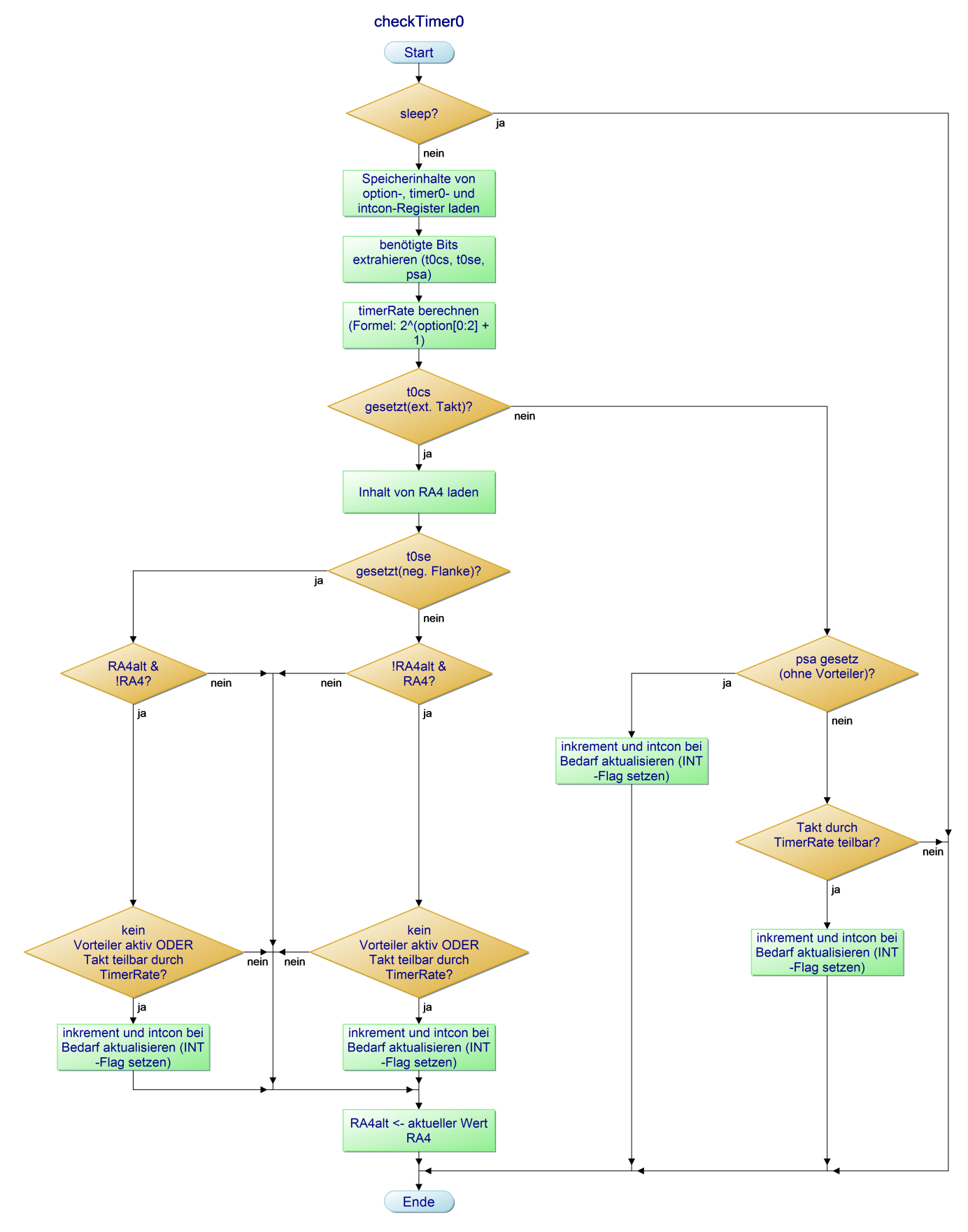


Abbildung : PAP Steuerwerk::checkTimer0()

|  |  |
| --- | --- |
|  | void Steuerwerk**::**checkTimer0**()** |
|  | **{** |
|  | // wenn PIC im sleep-Modus ist, soll Timer nicht behandelt werden |
|  | **if(**isSleeping**())** |
|  | **return;** |
|  |  |
|  | // benötigte Register laden |
|  | int option **=** alu**->**speicher**.**readOnBank**(**1**,** 0x01**);** |
|  | int timer0 **=** alu**->**speicher**.**readOnBank**(**0**,** 0x01**);** |
|  | int intcon **=** alu**->**speicher**.**readOnBank**(**0**,** 0x0B**);** |
|  |  |
|  | // Flags aus Registern auslesen |
|  | bool t0cs **=** CHECK\_BIT**(**option**,** 5**);** // TMR0 clock select |
|  | bool t0se **=** CHECK\_BIT**(**option**,** 4**);** // TMR0 select edge |
|  | bool psa **=** CHECK\_BIT**(**option**,** 3**);** // pre-scaler activated |
|  |  |
|  | int timerRate **=** **(**int**)** pow**(**2.0d**,** **(**double**)((**0x07 **&** option**)** **+** 1**));** |
|  |  |
|  | **if(**t0cs**)** // RA4 - externer Takt |
|  | **{** |
|  | int ra **=** alu**->**speicher**.**readOnBank**(**0**,** 0x05**);** |
|  | bool RA4 **=** CHECK\_BIT**(**ra**,** 4**);** |
|  |  |
|  | **if(**t0se**)** // negative Flanke |
|  | **{** |
|  | **if(**RA4alt **&&** **!**RA4**)** |
|  | **{** |
|  | // wenn kein Vorteiler aktiv ist oder der n-te Takt gegeben ist |
|  | **if(**psa **||** externalClockCycles **%** timerRate **==** 0**)** |
|  | **if(**incrementTimerAndCheckOverflow**(**timer0**))** // bei Überlauf BIT2 im INTCON-Register setzen |
|  | setTimer0InterruptFlag**(**intcon**);** |
|  |  |
|  | externalClockCycles**++;** |
|  | **}** |
|  | **}** |
|  | **else** // positive Flanke |
|  | **{** |
|  | **if(!**RA4alt **&&** RA4**)** |
|  | **{** |
|  | // wenn kein Vorteiler aktiv ist oder der n-te Takt gegeben ist |
|  | **if(**psa **||** externalClockCycles **%** timerRate **==** 0**)** |
|  | **if(**incrementTimerAndCheckOverflow**(**timer0**))** // bei Überlauf BIT2 im INTCON-Register setzen |
|  | setTimer0InterruptFlag**(**intcon**);** |
|  |  |
|  | externalClockCycles**++;** |
|  | **}** |
|  | **}** |
|  | ra **=** alu**->**speicher**.**readOnBank**(**0**,** 0x05**);** |
|  |  |
|  | RA4alt **=** CHECK\_BIT**(**ra**,** 4**);** |
|  | **}** |
|  | **else** // interner Takt |
|  | **{** |
|  | **if(**psa**)** // ohne Vorteiler --> Vorteiler ist dem Watchdog zugewiesen |
|  | **{** |
|  | **if(**incrementTimerAndCheckOverflow**(**timer0**))** // bei Überlauf BIT2 im INTCON-Register setzen |
|  | setTimer0InterruptFlag**(**intcon**);** |
|  | **}** |
|  | **else** // mit Vorteiler |
|  | **{** |
|  | **if(**alu**->**getCycles**()** **%** timerRate **==** 0**)** |
|  | **if(**incrementTimerAndCheckOverflow**(**timer0**))** // bei Überlauf BIT2 im INTCON-Register setzen |
|  | setTimer0InterruptFlag**(**intcon**);** |
|  | **}** |
|  | **}** |
|  | **}** |

Listing : Steuerwerk::checkTimer0()

### analyzeAndExecute

Die Methode interpretiert den aktuellen Befehlscode und führt die entsprechende Methode des Prozessors aus. Mehr Informationen zu den Befehlsmethoden finden sich in Kapitel Befehle. Zu beachten ist, dass Listing 6 nur einen Ausschnitt der Methode zeigt.

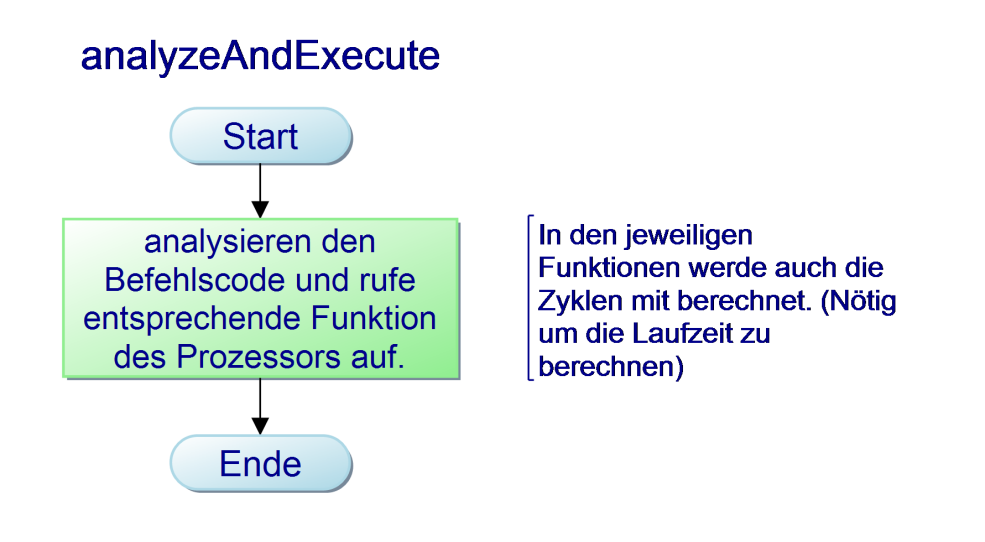


Abbildung 6: PAP Steuerwerk::analyzeAndExecute

|  |  |
| --- | --- |
|  | void Steuerwerk**::**analyzeAndExecute**(**int command**)** |
|  | **{** |
|  | command **=** command **&** 0x3FFF**;** |
|  |  |
|  | // BYTE-ORIENTED FILE REGISTER OPERATIONS |
|  | // ADDWF |
|  | // 00 0111 dfff ffff == 0x0700 |
|  | //& 11 1111 0000 0000 == 0x3f00 |
|  | **if((**command**&**0x3f00**)==**0x0700**)** |
|  | alu**->**addwf**(**command**);** |
|  |  |
|  | // ANDWF |
|  | // 00 0101 dfff ffff = 0x0500 |
|  | //& 11 1111 0000 0000 = 0x3f00 |
|  | **if((**command**&**0x3f00**)==**0x0500**)** |
|  | alu**->**andwf**(**command**);** |
|  |  |
|  | // CLRF |
|  | // 00 0001 1fff ffff == 0c0180 |
|  | //& 11 1111 1000 0000 == 0x3f80 |
|  | **if(** **(**command**&**0x3f80**)** **==** 0x0180**)** |
|  | alu**->**clrf**(**command**);** |
|  |  |
|  | //CLRW |
|  | // 00 0001 0xxx xxxx = 0x0100 |
|  | //& 11 1111 1000 0000 = 0x3F80 |
|  | **if((**command **&** 0x3f80**)==** 0x0100**)** |
|  | alu**->**clrw**();** |
|  |  |
|  | /\* |
|  | \* Hier folgen alle anderen Befehle |
|  | \* nach dem gleichen Schema. |
|  | \*/ |
|  |  |
|  | //RETLW |
|  | // 11 01xx kkkk kkkk = 0x3400 |
|  | // 11 1100 0000 0000 = 0x3C00 |
|  | **if((**command **&** 0x3c00**)==**0x3400**)** |
|  | alu**->**retlw**(**command**,this);** |
|  |  |
|  | //RETURN |
|  | // 00 0000 0000 1000 = 0x0008 |
|  | **if(**command **==** 0x0008**)** |
|  | alu**->**preturn**(this);** |
|  |  |
|  | // SLEEP |
|  | // 00 0000 0110 0011 = 0x0063 |
|  | **if(**command**==**0x0063**)** |
|  | alu**->**psleep**(this);** |
|  |  |
|  | // SUBLW |
|  | // 11 110x kkkk kkkk = 0x3c00 |
|  | **if((**command**&**0x3e00**)==**0x3c00**)** |
|  | alu**->**sublw**(**command**);** |
|  |  |
|  | // XORLW |
|  | // 11 1010 kkkk kkkk = 0x3A00 |
|  | //& 11 1111 0000 0000 = 0x3F00 |
|  | **if((**command**&**0x3f00**)==**0x3a00**)** |
|  | alu**->**xorlw**(**command**);** |
|  | **}** |

Listing : Steuerwerk::analyzeAndExecute()

### countWDT und WDTTimeout

Die Methode countWDT zählt den internen Watchdog Timer hoch. Der Watchdog wird in Zyklen gezählt. So kann in Abhängigkeit der eingestellten Frequenz die vergangene Zeit berechnet werden.

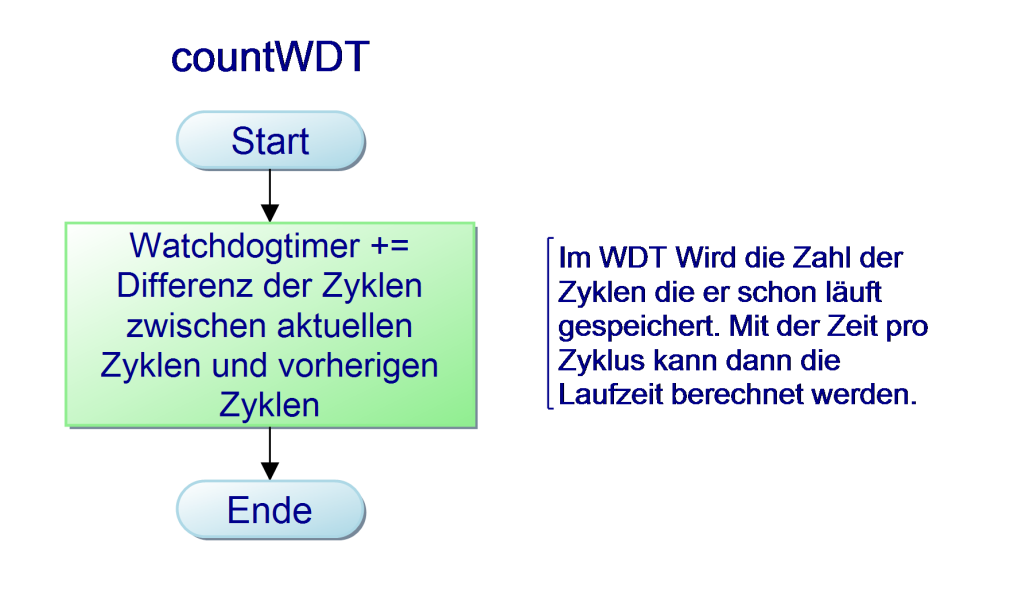


Abbildung : PAP Steuerwerk::countWDT()

|  |  |
| --- | --- |
|  | void Steuerwerk**::**countWDT**(**void**)** |
|  | **{** |
|  | wdt **+=** alu**->**cycles **-** alu**->**vorherigeCycles**;** |
|  | **}** |

Listing : Steuerwerk::countWDT

Hinter der Verzweigung *WDT Timeout?* in Abbildung 2 auf Seite 5 verbirgt sich eine weitere Methode. Abbildung 8 zeigt den PAP. Im Listing ist die Abfrage durch eine Switch-Case-Anweisung realisiert.

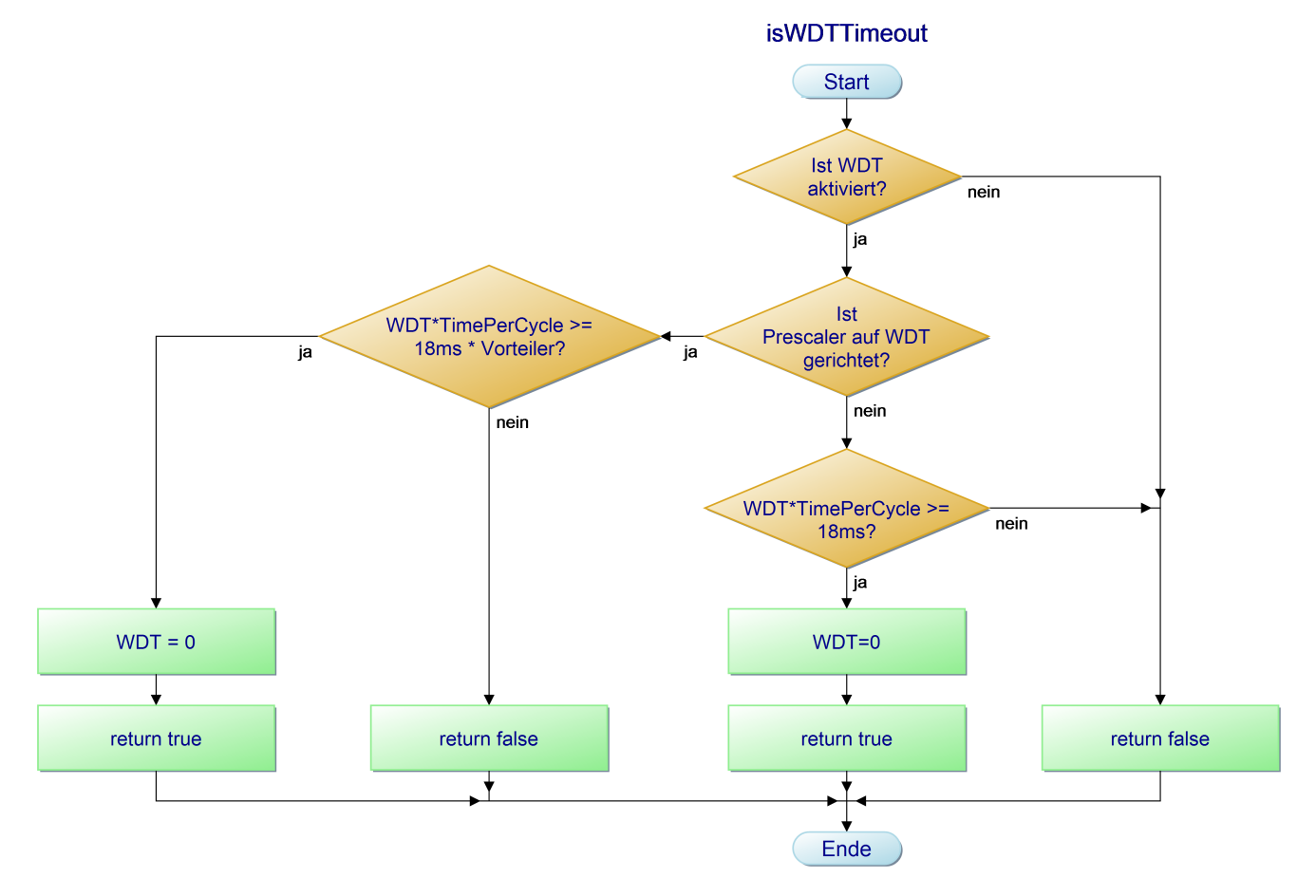


Abbildung : PAP Steuerwerk::isWDTTimeOut()

|  |  |
| --- | --- |
|  | bool Steuerwerk**::**isWDTTimeOut**()** |
|  | **{** |
|  | **if((**alu**->**speicher**.**address\_2007h **&** 0x0004**)==**0**)** //WDT ist deaktiviert |
|  | **return** **false;** |
|  |  |
|  | **if((**alu**->**speicher**.**readOnBank**(**1**,**1**)&**0x0008**)==**0**)** //Prescaler ist auf TMR0 gerichtet -> 18ms (1:1) |
|  | **{** |
|  | **if(**wdt**\***alu**->**timePerCycle**>=**18000**)** //timer >= 18 ms? |
|  | **{** |
|  | wdt**=**0**;** |
|  | **return** **true;** |
|  | **}** |
|  | **else** |
|  | **return** **false;** |
|  | **}** |
|  | **switch** **(**alu**->**speicher**.**readOnBank**(**1**,**1**)&**0x0007**)** **{** |
|  | **case** 0**:** //1:1 |
|  | **if(**wdt**\***alu**->**timePerCycle**>=**18000**)** //timer >= 18 ms? |
|  | **{** |
|  | wdt**=**0**;** |
|  | **return** **true;** |
|  | **}** |
|  | **else** |
|  | **return** **false;** |
|  | **break;** |
|  | **case** 1**:** //1:2 |
|  | **if(**wdt**\***alu**->**timePerCycle**>=**2**\***18000**)** |
|  | **{** |
|  | wdt**=**0**;** |
|  | **return** **true;** |
|  | **}** |
|  | **else** |
|  | **return** **false;** |
|  | **break;** |
|  |  |
|  | /\* |
|  | \* Alle anderen Fälle |
|  | \*/ |
|  |  |
|  | **case** 7**:** //1:128 |
|  | **if(**wdt**\***alu**->**timePerCycle**>=**128**\***18000**)** |
|  | **{** |
|  | wdt**=**0**;** |
|  | **return** **true;** |
|  | **}** |
|  | **else** |
|  | **return** **false;** |
|  | **break;** |
|  | **default:** |
|  | **break;** |
|  | **}** |
|  | **return** **false;** |
|  | **}** |

Listing : Steuerwerk::isWDTTimeOut()