Programmentwurf

Systemnahe Programmierung

Task-Verwaltung

6500913
4448199

TINF14B
4. Semester
2016

Inhalt

1 Einleitung

Prozessor ist das "Gehirn" des Computers. Der Prozessor ist eine Schaltung, die den Programmcode verarbeitet und die Grundfunktionen für die Informationsverarbeitung des Computers definiert.

Die Funktionsweise der verschiedenen Anwendungen basiert auf dem Prinzip des Durchführens einer bestimmter Reihenfolge von Befehlen und Daten, die in den Prozessor-Registern gespeichert werden. Diese Sequenz wird konsequent durchgeführt und dies wird ein Prozess genannt.

Die Reihenfolge der Befehlen geändert wird, wenn der Prozessor eine Sprunganweisung erhält, dann kann die Adresse des nächsten Befehls unterschiedlich sein. Ein weiteres Beispiel von der Veränderung des Prozesses ist der Fall, wenn ein Stoppbefehl oder eine vorübergehende Unterbrechung bzw. Interrupt empfängt wird.

Um zu bestimmen, welcher Prozess gestartet oder gestoppt werden soll, ist ein Scheduler verantwortlich. Das Scheduling der Aufgaben ist eine der wichtigsten Konzepte im Multitasking.

Es gibt 2 Strategien fürs Scheduling: präemptiven und non-präemptive.

Wenn das **kooperative (non-präemptive) Multitasking** verwendet wird, wird der nächste Prozess nur dann ausgeführt, wenn der aktuelle Prozess die CPU-Zeit an einen neuen Prozess explizit übergibt.

Bei der Verwendung vom **präemptiven Multitasking** trifft das Betriebssystem selbst die Entscheidung darüber, welchen Prozess zu unterbrochen oder zu starten. Dabei werden so genannte Zeitscheiben verwendet werden, die von den Prioritäten abhängig sind.

Das preemptive Multitasking erfordert die Verarbeitung von dem Timer-Interrupt. In dieser Art vom Multitasking kann die Ausführung eines Prozesses ausgeschaltet werden, um den anderen Prozess auszuführen. Dies kann genau zwischen zwei Anweisungen vom Prozess vorkommen. Die Verteilung der CPU-Zeit ist die Aufgabe des Schedulers.

In dieser Arbeit ist eine detaillierte Beschreibung eines Multitasking-Programms zu finden. Das Programm führt mehrere Prozesse und verwendet einen Timer-Interrupt als Scheduler. Die nachfolgenden Kapiteln beschreiben im Detail die Speicherzuweisung für Prozesse, die Timer-Einstellungen und die Anwendung vom Scheduler zur Steuerung der Prozesse. Eine visuelle Darstellung in Form von Flussdiagrammen, die die Arbeitsabläufe anschaulich beschreiben, ist im Anhang zu finden.

2 Aufgabenstellung

Das Ziel dieser Arbeit ist es, ein "Task-Verwaltung" Programm mit folgenden Funktionen zu schreiben:

- Es müssen drei Prozesse implementiert werden: "Konsolen-Prozess", "Prozess A" und "Prozess B".
- Prozesssteuerung sollte mit dem Scheduler erfolgen.
- Die Fähigkeit die Prozesse zu starten und zu stoppen.
- Jeder Prozess muss seine Priorität besitzen.
- Prozesssteuerung wird mit Hilfe des preemptive Multitasking durchgeführt .

Prozess A gibt die eingegebene Ziffer (n) gefolgt von n-Mal der Zeichenfolge ’ABCDE’ über die Serielle Schnittstelle 0 aus.

Prozess B gibt die Zeichen ’+’ und ’-’ abwechselnd jede Sekunde über die Serielle Schnittstelle 0 aus.

Konsolen-Prozess liest ständig Befehle (Zeichen) von der Seriellen Schnittstelle 0 ein und startet bzw. beendet andere Prozesse.

3 Kontrollausgabe

Zur Überprüfung der Funktionstüchtigkeit werden die Ausgaben auf vorgegebene Befehlssequenzen (im vorgegebenen Zeitraster) erzeugt und abgespeichert.

Sequenz 1

Beim Einbinden der **“os1.inc”** Datei und Starten des Programms wird die folgende Sequenz auf die Serielle Schnittstelle 0 gesendet:

’x’, ’1’, ’b’, ’x’, ’0’, ’c’, ’2’, ’b’, ’3’, ’c’, ’b’, ’4’, ’c’

Als Ergebnis soll die folgende Ausgabe entstehen:

1ABCDE+-2ABCDEABCDE+-3ABCDEABCDEABCDE+-+-+-+-+4ABCDEABCDEABCDEABCDE-+-+-

Und die folgende Ausgabe entsteht:

1ABCDE+-+-2ABCDEABCDE+-+3ABCDEABCDEABCDE-+-+-+-+-4ABCDEABCDEABCDEABCDE+-+-+

Sequenz 2

Beim Einbinden der **“os2.inc”** Datei wird die folgende Sequenz auf die Serielle Schnittstelle 0 gesendet:

’x’, ’9’, ’b’, ’x’, ’8’, ’b’, ’b’, ’1’, ’c’, ’c’, ’2’

Nach dem Einbinden und Starten des Programms entsteht die folgende Ausgabe:

9ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE+-+-+-+-8ABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDE+-+-+-+-+-+-++--1ABCDE++--+-2ABCDEABCDE

4 Antworten auf Fragen

a) Wie behandeln sie den doppelten Aufruf eines Prozesses solange dieser aktiv ist?
(z. B. bei der Befehlsfolge: ’b’, ’b’)

Beim Starten vom Prozess B, wird die folgende Prüfung ausgeführt:

- Wenn kein Prozess B läuft, starten einen neuen Prozess B.
- Wenn ein Prozess B läuft, starten den zweiten Prozess B.
- Wenn zwei Prozesse B laufen - nichts zu tun.

Somit hat das Programm die Möglichkeit, gleichzeitig zwei identische Prozesse B zu starten. Für den Prozess A ist es nicht verfügbar, denn der Prozess A wird nach der Durchführung seiner Aufgabe automatisch beendet.

b) Wie reagiert das Programm auf die Benutzung der Seriellen Schnittstelle 0 durch mehrere Prozesse?
(Z.B.: Können sich die Prozesse blockieren? Können "gesendete" Zeichen "verschwinden"?)

Das Steuern des Zugriffs auf Schreiben über die Serielle Schnittstelle basiert auf dem Prinzip des Semaphores (oder eher Mutex) - mit Hilfe der Variable "serialIsBusy", die den Prozesse meldet, ob die Serielle Schnittstelle frei oder besetzt ist. Nach dem Beginn der Übertragung wird die Variable gleich 1 gesetzt, nach dem Ende der Übertragung wird die ausgenullt.

Demgemäß, wenn eine Unterbrechung (Interrupt) vom Scheduler auftritt, und die Steuerung an einen anderen Prozess übergeben wird, der auch etwas auf die Serielle Schnittstelle sendet, verliert der erste Prozess den gesendeten Symbol nicht.

Weitere Informationen dazu ist im Kapitel "Interprozess Kommunikation" zu erhalten.

c) Wie wurden die Anforderung "Prioritäten" gelöst?

Für die Prioritäten wurde das 4-Byte-Array "priorities" festgelegt, mit 1 Byte für jeden Prozess.

Die Prioritäten bestimmen die CPU-Zeit, die für die Durchführung eines Prozesses zugewiesen wird, d.h. wie oft der Interrupt vom Timer 1 geruft wird.

Nach dem Interrupt vom Timer prüft die Funktion "scheduler", welcher Prozess gestartet werden sollte. Dementsprechend wird in TH1 den entsprechenden Prioritäten-Wert aus dem Array "priorities" geschrieben.

Je größere Wert wird in TH1 und TL1 geschrieben wird, desto öfter der Interrupt auftritt, d.h. je höhe die Priorität ist, desto kleine der Wert im Array "priorities" sein soll.

Das Programm arbeitet mit zwei 16-Bit-Timer. Der Interrupt vom Timer 1 kann die Timer 0 und folglich die Leistung der Funktion "waitOneSecond" beeinflussen, d.h. die Länge einer Sekunde somit sich ändern kann. Daher soll der Timer 1 mindestens 4 Mal schneller als der Timer 0 überlaufen.

Deswegen wird das Array "priorities" während der Initialisierung des Programms mit den folgenden Werten initialisiert: 0xC0, 0xC0 und 0xE0 für den Prozess A, B und Control beziehungsweise.

5 Programmeinstellungen

5.1 Prozessdaten

Für die Prozessdaten wird das 80-Byte-Array **processData** definiert, mit 20 Bytes für jeden Prozess (A, B, B2, Control). Das Array enthält alle Register (13 Bytes). Auch 2-Byte-ProgramCounter kann zum Array geschrieben werden (15 Byte gesamt) und 5 Byte bleiben für alle Fälle. In der aktuellen Version des Programms werden diese 5 Byte durch keinen Prozess verwendet, die sind für die zukünftige Verwendung aus Sicherheit reserviert.

Zum Speichern des aktuellen StackPointer werden die Arrays **stacks** und **stackStartAddr** verwendet, je 4 Byte groß, 1 Byte für jeden Prozess.

Im Array "stacks" werden die aktuellen Stack-Zeiger für jeden der vier Prozesse gehalten.
Im Array "stackStartAddr" werden die anfänglichen Stack-Zeiger für vier Prozesse gespeichert.
Zunächst sind die beiden Arrays gleich, aber sobald die Daten zum Stack geschrieben werden, gibt es den Unterschied zwischen Arrays.

Wenn der Prozess A beginnt, wird "stacks" in den SP aufgezeichnet (bei der Adresse processData).
Wenn der Prozess B1 beginnt, wird "stacks+1" in den SP aufgezeichnet (bei der Adresse processData+20).
Wenn der Prozess B2 beginnt, wird "stacks+2" in den SP aufgezeichnet (bei der Adresse processData+40).
Wenn der Prozess Control beginnt, wird "stacks+3" in den SP aufgezeichnet (bei der Adresse processData+60).

Beim Start eines neuen Prozesses, wird der SP auf den ursprünglichen Wert mit Hilfe von "stackStartAddr" eingestellt.

processIPTable (IP = InstructionPointer) ist der Program-Counter, der die Adresse des nächsten Befehls hält. Jeder Adresse des Programms sind in "processIPTable" 2 Byte zugewiesen, denn der PC ist 2 Byte groß. Der PC ist nicht vom Software verfügbar, er kann nicht durch die herkömmlichen Verfahren geschrieben und gelesen werden, das lässt sich nur mit Hilfe des Stack durch POP-Befehle ausführen. Nach dem Lesen des PCs schreibt das Programm dies zum "processIPTable" für einen bestimmten Prozess.

activeFlags zeigt an, welcher der Prozesse aktiv ist. Für das Array werden 4 Byte zugeordnet, obwohl es in der Theorie möglich wäre, nur 4 Bits zuzuweisen. Die Klarheit solcher Struktur ist viel höher als die Verständlichkeit der kodierten Variante.

isProcessA, **isProcessB**, **isProcessControl** sind die genannten Zeiger auf das niedrigstwertige (LSB), mittelwertige (CSB) und höchstwertige (MSB) Bit des Array "activeFlags".

processAddress ist die Startadresse des Prozesses, der ausgeführt (zu den laufenden Prozessen hinzugefügt) oder gelöscht werden soll.

statusFlag beschreibt einen der Flags, die das Status des Prozesses zeigen (newProcessFlag, deleteProcessFlag, noProcessFlag). Mit Hilfe dieser Flags entscheidet der Scheduler, welche Aktion zu erledigen - starten oder beenden des Prozesses.

newProcessFlag, deleteProcessFlag, noProcessFlag sind Konstanten und werden dazu verwendet, um die Verständlichkeit des Codes zu erhöhen. Die zeigen, welche Aktion der Scheduler erledigen soll: Start oder Löschung des Prozesses.

currentProcess ist der Index des aktuellen Prozesses, dem die Führung in der Funktion "scheduler" übertragen werden soll.

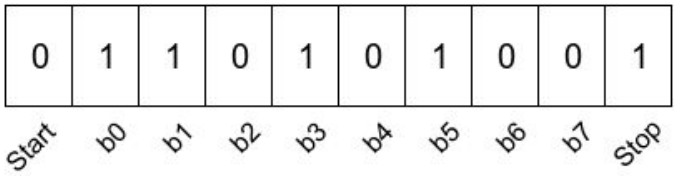
priorities sind die Prioritäten für den Timer 1. Eine ausführlichere Beschreibung ist im Kapitel 4, "Antworten auf Fragen" zu finden.

Das Bild unten zeigt die Speicherzuweisung entsprechend der obigen Beschreibung:

5.2 Berechnung der Baudrate

Für die Serielle Schnittstelle 0 wurden der Modus 1 (8-Bit-UART, variable Baudrate) und die Baudrate 14.400 entsprechend der Aufgabestellung gewählt .

Der ASCII-Wert des Zeichens "+" ist 43, der in einen 8-Bit-Binärwert "00101011" zerfällt. Die Daten werden von rechts nach links (LSB) übertragen. 10 Bits werden gesendet oder empfangen: ein Startbit (0), 8 Datenbits und ein Stopbit (1).



Da die 14400 bps übertragen werden, ist die Zeit zur Übertragung jedes dieser Bits:
$1 / baud\ rate = 1 / 14400 = 69$ Mikrosekunden pro Bit oder **690 Mikrosekunden pro Zeichen**.
Also, wir senden 14400 Bits pro Sekunde oder **~1440 Zeichen pro Sekunde**.

Die Baudrate-Berechnungsformel lautet wie folgt:

$$\frac{2^{SMOD} \times oscillator\ frequency}{64 \times baud\ rate\ generator\ overflow\ rate}$$

SMOD-Bit wird im Programm gelöscht → $2^{SMOD} = 2^0 = 1$
Die Oszillatorfrequenz wird in den Keil-Einstellungen entsprechend der Aufgabe eingestellt
$24\ MHz = 24\ 000\ 000$

"Baud rate generator overflow rate" wird durch die folgende Formel berechnet: $2^{10} - S0REL$
Im Gegenzug, $S0REL = S0RELH.1-0,\ S0RELL.7-0$

Also, um die Baudrate gleich 14.400 zu definieren, ist $x = S0RELH.1-0,\ S0RELL.7-0$ zu finden.

$$\frac{24.000.000}{64 \times (1.024 - x)} = 14.400 \quad \rightarrow \quad x = 998\ d \quad \rightarrow \quad 3E6\ h$$

S0RELH = 0x03
S0RELL = 0xE6

5.3 Initialisation

Die Funktion "init" wird als erstes nach dem Start des Programm ausgeführt. Ihre Aufgabe ist es, alle nötige Einstellugen der Timer, Register und Variablen zu erledigen.

Hier werden die Serielle Schnittstelle und Baud Rate konfiguriert:

```
; Set Serial0 to Mode 1 (8-bit UART, variable baud rate)
CLR SM0
SETB SM1

; set baud rate to 14400
MOV S0RELL, #0xE6
MOV S0RELH, #0x03
```

Danach werden beide Timer zum 16-Bit-Mode umgestellt und der Timer 0 wird mit den Startwerten initialisiert:

```
; set timer1 and timer0 to Mode 1 - 16-bit
MOV    A, #00010001b
MOV TMOD, A
; start timer1
SETB TR1
; init timer 0
MOV TL0, #0xB0
MOV TH0, #0x3C
; start timer 0
SETB TR0
```

In der Funktion werden auch die Werte von Hilfsvariablen und Prioritäten initialisiert.

```
MOV statusFlag, #noProcessFlag
MOV repCount, #0x00
MOV secondLength, #40d
MOV isTimer0Ovf, #0x00
MOV serialIsBusy, #0
MOV currentProcess, #3
; init priorities for each process
MOV priorities + 0, #0xC0 ;processA
MOV priorities + 1, #0xC0 ;processB
MOV priorities + 2, #0xC0 ;processB2
MOV priorities + 3, #0xE0 ;processControl
```

Als nächstes werden die Startadresse für alle Prozesse initialisiert. Alle Prozesse werden als inaktiv markiert und für jeden Prozess wird eigenen Stack festgelegt.

```
1. MOV DPTR, #processB
2. MOV processIPTable + 2, DPL
3. MOV processIPTable + 3, DPH
4. MOV isProcessB, #0x00
5. ...
6. ADD A, #20
7. MOV stacks + 1, A
8. MOV stackStartAddr + 1, A
```

Schließlich wird der Zeiger auf der Adresse vom Prozess Control gesetzt und dem Prozess wird der Flag "newProcess" zugewiesen. Diese Daten werden in der Funktion "scheduler" erhalten und der Prozess Control wird gestartet.

```
1. MOV DPTR, #processControl
2. MOV processAddress + 0, DPL
3. MOV processAddress + 1, DPH
4. MOV statusFlag, #newProcessFlag
```

6 Timer

Das Programm verwendet zwei 16-Bit-Timer: für 1-Sekunde-Verzögerung und für den Überlauf-Interrupt.

6.1 Timer Interrupt (Scheduler)

Der Timer 1 führt den Überlauf-Interrupt und die Hauptfunktionalität des Schedulers in der Funktion "scheduler". Die Aufgabe des Schedulers besteht darin, Prozesse zu verwalten - starten, wechseln oder beeden. Gemäß der Dokumentation wird der Timer-1-Überlauf bei der Adresse 0x1B gelegt. Daher wird an dieser Adresse ein absoluter Segment für den Timer-1-Überlauf definiert.

Zuallererst liest das Programm den Programmzähler (PC) von dem Stack, in dem der PC nach dem Aufruf des Interrupt-Handler gespeichert wird. Das Lesen vom PC vom Software ist nicht möglich, der kann nicht durch herkömmliche Verfahren geschrieben und gelesen werden, es lässt sich nur mit Hilfe des Stack durch POP-Befehle ausführen. Nach dem Lesen des PC schreibt das Programm dies zum "processIPTable" für einen bestimmten Prozess. Für jede Adresse sind in "processIPTable" 2 Byte zugewiesen (da PC 2 Byte groß ist).

```
1. POP DPH
2. POP DPL
3. MOV A, #processIPTable
4. ADD A, currentProcess
5. ADD A, currentProcess
6. MOV R0, A
7. MOV @R0, DPL
8. INC R0
9. MOV @R0, DPH
```

Dann werden die Werte aller Register des Mikrocontrollers in den aktuellen Stack aufgezeichnet.

```
1. pushRegisters:
2.         PUSH PSW
3.         ...
4.         PUSH DPL
```

Ins Array "stacks" beim Index "currentProcess" wird der Wert von StackPointer (SP) geschrieben.

```
1. MOV A, #stacks
2. ADD A, currentProcess
3. MOV R0, A
4. MOV @R0, SP
```

Danach wird die Variable "currentProcess" zyklisch erhöht. Zulässige Werte sind 0, 1, 2 oder 3. Wenn "currentProcess" den Wert 4 erreicht, wird der auf Null zurückgesetzt.

```
processTableLoop:
        MOV A, currentProcess
        INC A
        CJNE A, #4, notFirstProcess
                MOV A, #0
        notFirstProcess:
                MOV currentProcess, A
```

Der Wert von "statusFlag" wird dann geprüft. Wenn der Wert "newProcessFlag" (= 1) ist, wird die Funktion "startNewProcess" - Start des Prozesses - geruft. Wenn der Wert "stopProcessFlag" (= 2) ist, wird die Funktion "stopProcess" - Löschung des Prozesses - gestartet.

```
MOV R0, statusFlag
CJNE R0, #newProcessFlag, notNew
        JMP startNewProcess
notNew:
        CJNE R0, #stopProcessFlag, notNewNotStop
                JMP stopProcess
notNewNotStop:
```

Nach der Verarbeitung des Werts von "statusFlag", wird der zurückgesetzt, und das Programm geht weiter zum normalen Zyklus von der Ubertragungssteuerung zwischen der Prozesse. Dazu dient ein Array "activeFlags", dessen ein oder mehrere Bytes auf 1 gesetzt werden, wenn der entsprechende Prozess läuft, und auf 0, wenn der entsprechende Prozess nicht aktiv ist.

Das Array ist 4 Byte groß, was die Möglichkeit darstellt, vier Prozesse zu führen, ein Prozess A, zwei Prozesse B und ein Prozess Control. Nur der Prozess wird gestartet, für den das Byte in "activeFlags" gleich 1 ist und der ist am nächsten zu "currentProcess". Wenn "activeFlags" nur Nullen enthält, dann läuft das Programm im Zyklus.

```
MOV A, #activeFlags
ADD A, currentProcess
MOV R1, A
CJNE @R1, #0x01, processTableLoop
```

Wenn ein aktiver Prozess gefunden ist, wird in TH1 seine Priorität geschrieben - die Zeit bis zur nächsten Überprüfung.

```
MOV TL1, #0x00
CJNE R1, #isProcessA, notProcessA
        CLR TR1
        MOV TH1, priorities
        SETB TR1
...
```

Dann wird der SP mit Hilfe des Array "stacks" beim Index "currentProcess" eingestellt und alle Register werden gelesen.

```
1. loadStackPointer:
2.        MOV A, currentProcess
3.        ADD A, #stacks
4.        MOV R0, A
5.        MOV SP, @R0
6. popRegisters:
7.        POP DPL
8.        ...
9.        POP PSW
```

Und der letzte Teil liest die Funktion den gespeicherten PC für den unterbrochenen Prozess aus "processIPTable" und legt den PC zum Stack. RETI-Befehl stellt automatisch einen Übergang an die Adresse vom Stack, so dass die Rückkehr nicht an die Interrupt-Handler-Adresse geführt wird, sondern auf die gespeicherte Adresse von "processIPTable".

```
1. MOV A, #processIPTable
2. ADD A, currentProcess
3. ADD A, currentProcess
4. MOV R0, A
5. MOV DPL, @R0
6. PUSH DPL
7. INC R0
8. MOV DPH, @R0
9. PUSH DPH
10. RETI
```

6.2 Eine-Sekunde-Verzögerung

Zum Zählen von 1 Sekunde werden Timer 0 und die zyklische Strategie verwendet: das Programm wartet, bis das Überlaufbit TF0 (TCON.5) durch den Timer-0-Interrupt eingestellt wird.

Der Timer basiert auf der Geschwindigkeit von Quarzoszillator. Die Frequenz beträgt 24 MHz, wie in der Aufgabe definiert, d.h. die Geschwindigkeit von Quarzoszillator beträgt 24.000.000.

Der standarte 8051-Timer inkrementiert jede 12 Quarzzyklen.
Daher inkrementiert der Timer 0 $24.000.000 / 12 = 2.000.000$ Mal pro Sekunde.

Da der Timer 16-Bit ist, zählt er vor dem Zurücksetzen von 0 bis 65.535.
Das bedeutet, dass der Timer $2.000.000 / 65.536 = 30,5$ Mal pro Sekunde überläuft.

Somit muss der Zyklus von Setzen und Nullung von TF0 30,5 Mal wiederholt werden, und das wird genau 1 Sekunde dauern.

Aber es ist unmöglich den Zyklus 30,5 Mal zu wiederholen, und die Wiederholung von 30 oder 31 Mal wird zu einem falschen Ergebnis führen. Das bedeutet, die einfache Auszählung der Überläufe von 0 bis 65.536 passt in dem Fall nicht. Also, muss die zusätzliche Berechnung erledigt werden.

Das Ziel ist, den Timer-0-Überlauf bei so einer Frequenz zu initialisieren, die bis zum genau 1-Sekunden-Intervall summiert.

Ein 16-Bit-Timer-Zyklus taktet $65.536 / 2.000.000 = 0.033$ Sekunden.

Mit anderen Worten, der Timer 0 beginnt mit 0 zu zählen, zählt bis zum Überlauf auf 65.535 und das dauert 0,033 Sekunden. Das Problem besteht darin, dass eine Sekunde nicht durch 0,033 Sekunden ganzzahlig dividiert werden kann, deswegen gibt es die Ungenauigkeit.

Wenn anstelle von Überlauf alle 0,033 Sekunden, der Timer 0 alle 0,025 Sekunden sich überlaufen lässt, braucht man genau 40 Zyklen machen um genau eine Sekunde zu bekommen. In Bezug von Timer-Zyklen 0,025 Sekunden ist $2.000.000 \times 0,025 = 50.000$ lang. Das bedeutet, dass nachdem der Timer 50.000 Mal inkrementiert wird, ist 1/40 der Sekunde verstrichen.

Also, das Ziel ist, dass der Timer 0 alle 0,025 Sekunden anstatt alle 0,033 Sekunden überläuft und der volle Zyklus 40 Mal statt 30 oder 31 Mal wiederholt wird.

Dazu muss der Zählerstand bei $65.536 - 50.000 = 15.536$ initialisiert werden.

$15536_{10} = 3CB0_{16}$

Also: **TL0 = 0xB0**
TH0 = 0x3C

Zum Speichern der Zahl 40 (die Anzahl der Timer-0-Überlauf-Schleifen) wird die 1-Byte-Variable "secondLength" verwendet.

Alle Prozesse B verwenden den Timer 0 für die Rechnung von einer Sekunde, deswegen müssen die Prozesse keinen Überlauf-Flag überprüfen, sondern eine globale Variable. Der Timer setzt die Variable beim Überlauf, und jedes der Prozesse dekrementiert die während der Prüfung des Timers.

Wenn die Variable 0 erreicht, das bedeutet, dass der Timer neu gestartet werden muss. Es sollte für alle ausgeführten Prozessen gemeldet werden, dass der Überlauf entsteht. Jeder Prozess prüft, ob der Wert von der Variable ungleich 0 ist. Ist der Wert nicht 0, reduziert der Prozess den Wert. Wenn die Variable gleich 1 ist, tritt die nächste Erwartung auf den Überlauf.

Der Vollzyklus der Verzögerung von 1 Sekunde wie folgt aussieht:

- Timer 0 wird nach Berechnungen konfiguriert (THTL = 3CB0).
- Die Erwartung auf den Überlauf läuft in der Funktion "waitOneSecond".
- Überlauf-Bit und der Timer werden in der Funktion "timer0Overflow" zurückgesetzt.
- R5 Register wird dekrementiert, und der Zyklus wird wiederholt, bis R5 0 erreicht.

```
timer0Overflow:
        MOV TL0, #0xB0
        MOV TH0, #0x3C
        MOV isTimer0Ovf, #0x07
        CLR TF0
RETI
```

```
waitOneSecond:
        MOV R5, secondLength
        timerOverflowLoop:
                ; default isTimer0Ovf = 000
                CJNE A, #0, nextTimer0Ovf
                        ; start timerOverflowLoop again
                nextTimer0Ovf:
                        ; set isTimer0Ovf = xx1
                        CJNE A, currentProcess, notTimer0OvfProcessB
                                ; current process is 01 (first ProcessB)
                                ; set isTimer0Ovf = xx0
                                JMP endTimer0OvfProcess
                notTimer0OvfProcessB:
                        ; set isTimer0Ovf = x1x
                        CJNE A, currentProcess, notTimer0OvfProcessB2
                                ; current process is 02 (second ProcessB)
                                ; set isTimer0Ovf = x0x
                                JMP endTimer0OvfProcess
                notTimer0OvfProcessB2:
                        JMP timerOverflowLoop
        endTimer0OvfProcess:
                DJNZ R5, timerOverflowLoop
```

Nach den Beobachtungen liegt die Länge von der Verzögerung im Bereich von 0,99621100 bis 1,00302600 Sekunden.

7 Prozesse

Die Prozesssteuerung wird mit zwei Funktionen ausgeführt: "startNewProcess" und "stopProcess". Aufruf einer der Funktionen wird nach dem Interrupt des Timer 1 in der Funktion "Scheduler" durchgeführt , basierend auf dem Wert der Variable "statusFlag".

```
1. MOV R0, statusFlag
2. CJNE R0, #newProcessFlag, notNew
3.        JMP startNewProcess
4. notNew:
5. CJNE R0, #stopProcessFlag, notNewNotDelete
6.        JMP stopProcess
7. notNewNotDelete:
```

Die Variable "statusFlag" wird bei Empfang eines bestimmten Symbols an die Serielle Schnittstelle im "ProcessControl" und in der Funktion "removeProcessA" gesetzt. Die gültigen Werte sind: “newProcessFlag”, “deleteProcessFlag”, “noProcessFlag”.

```
1. removeProcessA:
2.        MOV DPTR, #processA
3.        MOV processAddress + 0, DPL
4.        MOV processAddress + 1, DPH
5.        MOV statusFlag, #stopProcessFlag
```

7.1 Start des Prozesses

Um einen Prozesses zu starten, werden der aktuelle Zeiger und die Prozess-Adresse vergleicht. Wenn sie übereinstimmen, wird diese Adresse in den Program-Counter (PC) geschrieben und alle Register werden zum Stack gespeichert. Mit Hilfe der genannten Zeiger (isProcessA, isProcessB, isProcessB2, isProcessControl) wird das entsprechende Byte des Array "activeFlags" gleich 1 gesetzt. Der Index des nötigen Prozesses (0, 1, 2 oder 3) wird in die Variable "currentProcess" aufgezeichnet.

Auf der Grundlage dieser Daten wird der notwendige Prozess in der Funktion "scheduler" gestartet.

```
1.  processAtoStart:
2.          MOV DPTR, #processA
3.          MOV A,DPH
4.          CJNE A, processAddressH, processBtoStart
5.                  MOV A, DPL
6.                  CJNE A, processAddressL, processBtoStart
7.                          MOV processIPTable, DPL
8.                          MOV processIPTable+1, DPH
9.                          MOV SP, stackStartAddr
10.
11.                         PUSH PSW
12.                         ...
13.                         PUSH DPL
14.
15.                         MOV stacks, SP
16.                         MOV isProcessA, #0x01
17.                         MOV currentProcess, #0x00
18.                         JMP endStart
```

Es gibt die Möglichkeit, mehrere Prozesse B auszuführen. Es wird die Prüfung durchgeführt, wie viele Prozesse B bereits ausgeführt werden und es wird entschieden, welchen von den Prozessen B zu starten.

```
1.  MOV A, isProcessB
2.  ; check if Process B1 is running
3.  CJNE A, #0x01, processBNotRunning
4.          ; check if Process B2 is running
5.          checkIfProcessB2Runnig:
6.                  MOV A, isProcessB2
7.                  CJNE A, #0x00, endStart
8.                          ; process B2 is free to use - start it
9.                          JMP endStart
10. processBNotRunning:
11.         ; process B1 not runnig, start it
12.         JMP endStart
```

7.2 Löschung des Prozesses

Die Löschung des Prozesses wird in der gleichen Weise durchgeführt - mit Hilfe von der genannten Zeiger (isProcessA, isProcessB, isProcessB2, isProcessControl) wird das entsprechende Byte des Array "activeFlags" gleich 0 gesetzt. Diese Information bekommt die Funktion "scheduler" und der Prozess wird nach einem Interrupt nicht wieder gestartet.

```
1. processAtoStop:
2.         MOV DPTR, #processA
3.         MOV A, DPH
4.         CJNE A, processAddressH, processBtoStop
5.                 MOV A, DPL
6.                 CJNE A, processAddressL, processBtoStop
7.                         MOV isProcessA, #0x00
8.                         JMP endStop
```

7.3 Process Control

Der Prozess Control ist ein Steuerungs-Prozess, der den erforderlichen Prozess startet oder beendet.

Zunächst wartet der Prozess mit Hilfe einer endlosen Schleife auf Befehle auf UART0. Sobald ein Befehl empfangen wird, wird er in den Register 7 gespeichert und die Funktion "readSerial" wird geruft. Die Funktion **"readSerial"** überprüft, welches Zeichen auf UART0 gesendet wurde. Gemäß der Aufgabe werden die Symbole "b", "c" und "1-9" verarbeitet werden, die anderen werden ignoriert.

Zeichen	**Aktion**
1-9	Start des Prozess A
b	Start des Prozess B
c	Stop des Prozess B
Sonst	Keine Aktion

Zunächst wird das **Symbol "b"** überprüft. Wenn dieses Symbol empfangen wird, lässt sich der Zeiger auf die Anfangsadresse des Prozess B festlegen und es wird der "newProcessFlag" gesetzt. Dieser Flag wird beim nächsten Timer-Interrupt in der Funktion "scheduler" gelesen und die Funktion "startNewProcess" wird gestartet.

```
1. checkLetterB:
2. CJNE R7, #'b', checkLetterC
3.         MOV DPTR, #processB
4.         MOV processAddress + 1, DPH
5.         MOV processAddress + 0, DPL
6.         MOV statusFlag, #newProcessFlag
```

Das gleiche geschieht im Fall des **Symbols "c"**. Aber in diesem Fall wird der "deleteProcessFlag" gesetzt und die Funktion "stopProcess" wird geruft.

```
checkLetterC:
CJNE R7, #'c', checkNumbers
        MOV DPTR, #processB
        MOV processAddress + 1, DPH
        MOV processAddress + 0, DPL
        MOV statusFlag, #stopProcessFlag
```

Als nächstes werden die **Zeichen "1-9"** geprüft. Dazu wird die mathematische Operation SUBB (Subtraktion mit Übertrag) mit Hilfe von CY-Register verwendet. Wenn das eingegebene Zeichen größer als 1 oder kleiner als 9 ist, wird das Prozess A gestartet.

```
checkNumbers:
        MOV A, R7
        CLR C
        SUBB A, #'1'
        JC noValidProcess
        SUBB A, #0x09
        JNC noValidProcess
              MOV repCount, R7
              MOV DPTR, #processA
              MOV processAddress + 1, DPH
              MOV processAddress + 0, DPL
              MOV statusFlag, #newProcessFlag
```

Wenn das eingegebene Zeichen nicht eines der oben genannten ist, springt das Programm zurück zur endlosen Schleife und wartet auf eine neue Eingabe.

```
noValidProcess:
        MOV R7, #0x0
        CLR RI0
        JMP processControl
```

7.4 Process A

Die Aufgabe des Prozess A ist es, die Anzahl der Wiederholungen der Zeichenfolge und die Folge "ABCDE" auf UART zu schreiben. Zum Beispiel: 2ABCDEABCDE. Nach der Beendigung des Schreibens muss der Prozess ohne Eingreif des Benutzers beendet werden.

Für den Prozess A wird ein verschiebbarer Segment der Klasse CODE erstellt.

Dann läuft die Funktion "**printRepCountToUART**", die die Anzahl der Wiederholungen auf UART mithilfe der Variable "repCount" schreibt. Die Anzahl der Wiederholungen wird zur Variable "repCount" in der Funktion "**readSerial**" in Prozess Control geschrieben.

```
1. readSerial:
2.         ...
3.         MOV R7, S0BUF
4.         ...
5.         MOV repCount, R7
6.         ...
```

```
1. printRepCountToUART:
2.         CALL waitForSerialAndBlock
3.         MOV A, repCount
4.         MOV S0BUF, A
5.         CALL waitForLetterIsSent
```

Danach wird die Anzahl der Wiederholungen "repCount" zum R7 aufgezeichnet. Der Wert wird vom Format 'char' zum Integer durch das Subtrahieren von chr(0) umwandelt.

```
1. MOV A, repCount
2. SUBB A, #'0'
3. MOV R7, A
```

Als nächstes wird die Zeichenfolge "ABCDE" durch die Funktion "**printLettersToUART**" geschrieben. Diese Funktion wird wiederholt, bis "repCount" 0 erreicht.

```
1. printLettersToUART:
2.         ...
3.         CALL printABCDEToUART
4.         DJNZ R7, printLettersToUART
5.         ...
```

Die Funktion "**printABCDEToUART**" schreibt jedes Symbol der Folge mithilfe einer Schleife, in der der nächste Buchstabe durch das Inkrement erhalten wird, bis der letzte Buchstabe erreicht wird.

```
printABCDEToUART:
        MOV R1, #'A'
        loopAtoE:
                CALL waitForSeriaAndBlock
                MOV S0BUF, R1
                CALL waitForLetterIsSent
                INC R1
                CJNE R1, #'F', loopAtoE
```

Um auf UART ein Zeichen zu schreiben, lässt sich die Hilfsfunktion "**waitForLetterIsSent**" verwendet. In der Schleife wird das erste Bit von Register SCON (TI) geprüft, das für den Beginn und das Ende der Übertragung verantwortlich ist. Nachdem die Übertragung abgeschlossen ist, werden das Bit TI und die Hilfsvariable "serialIsBusy" zurückgesetzt.

```
waitForLetterIsSent:
        MOV A, S0CON
        JNB ACC.1, waitForLetterIsSent
        MOV serialIsBusy, #0
        ANL A, #11111101b
        MOV S0CON, A
```

Damit das gesendete Symbol nicht verloren wird (wenn die Steuerung an einen anderen Prozess übergeben wird), lässt sich die Hilfsfunktion "**waitForSeriaAndBlock**" verwenden. Der Prozess wartet, bis die Variable "serialIsBusy" gleich 0 ist, und setzt die auf 1, bis zum Ende der Übertragung.

```
waitForSeriaAndBlock:
        checkSerialIsBusy:
                NOP
                MOV A, serialIsBusy
        JB ACC.0, checkSerialIsBusy
        MOV serialIsBusy, #1
```

Sobald alle nötigen Zeichen geschrieben wurden, startet die Funktion "**removeProcessA**". Die setzt den Zeiger auf die Adresse des aktuellen Prozesses und schreibt ins Array "statusFlag" den "stopProcessFlag". Während des nächsten Timer-Interrupt wird die Funktion "scheduler" diesen Flag sehen und die Funktion "stopProcess" wird angeruft.

```
removeProcessA:
        MOV DPTR, #processA
        MOV processAddress + 0, DPL
        MOV processAddress + 1, DPH
        MOV statusFlag, #stopProcessFlag
```

7.5 Process B

Die Aufgabe des Prozess B ist es, die Zeichen + und - abwechselnd im 1-Sekunden Takt auf UART zu schreiben. Der Prozess wird solang ausgeführt, bis der Benutzer den beendet.

Für den Prozess B wird auch ein verschiebbarer Segment der Klasse CODE erstellt.

Dann wird die Variable "secondLength" in den Register 5 aufgezeichnet und die Hauptschleife "**processB**" wird aufgeruft.

```
1. processB:
2.         CALL printPlusToUART
3.         CALL waitOneSecond
4.         CALL printMinusToUART
5.         CALL waitOneSecond
6.         JMP processB
```

Die Schleife schreibt abwechselnd die Symbole Plus und Minus mithilfe der Funktionen "**printPlusToUART**" und "**printMinusToUART**".

```
1. printPlusToUART:
2.         MOV S0BUF, #'+'
3.         CALL waitForLetterIsSent
```

```
1. printMinusToUART:
2.         MOV S0BUF, #'-'
3.         CALL waitForLetterIsSent
```

Das Schreiben eines Symbols ähnelt Prozess A: mithilfe der Funktionen "**waitForLetterIsSent**" und "**waitForSeriaAndBlock**".

Für die Verzögerung einer Sekunde ist die Funktion "waitOneSecond" verantwortlich. Eine detaillierte Beschreibung der Funktion ist im Kapitel 6 "Eine Sekunde Verzögerung" zu finden.

7.8 Interprozess Kommunikation

Ein Prozess schreibt die Zeichen auf UART mithilfe des SCON-Registers. Während der Übertragung des Zeichen wird das Bit TI (SCON.1) geprüft. Wenn das Bit gleich 1 ist, ist die Übertragung beendet, wenn nicht, bleibt das Programm solange in der Schleife, bis das Bit gesetzt ist. Dies bedeutet, dass die Übertragung des Zeichen abgeschlossen ist.

```
1. waitForLetterIsSent:
2.         MOV A, S0CON
3. JNB ACC.1, waitForLetterIsSent
```

Sobald der Prozess die Übertragung beginnt, wird die Hilfsvariable "serialIsBusy" auf 1 gesetzt. Dies bedeutet, dass die Serielle Schnittstelle belegt ist. Wenn die Serielle Schnittstelle von einem anderen Prozess verwendet wird, wartet der aktuelle Prozess, bis die Schnittstelle frei ist, durch die Prüfung der Variable "serialIsBusy".

```
1. checkSerialIsBusy:
2.         NOP
3.         MOV A, serialIsBusy
4. JB ACC.0, checkSerialIsBusy
5. MOV serialIsBusy, #1
```

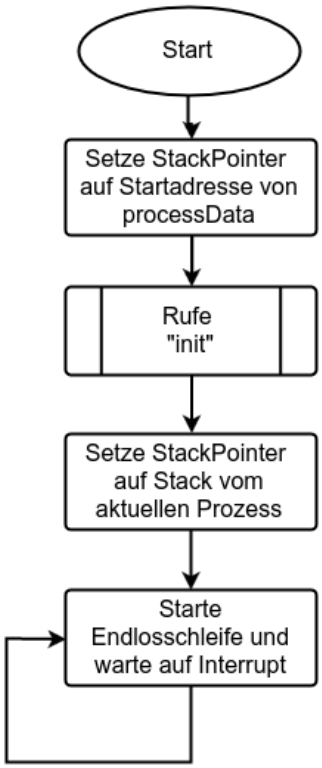
Nachdem die Übertragung abgeschlossen ist, setzt das Programm "serialIsBusy" gleich 0 (meldet dass die Serielle Schnittstelle verfügbar ist) und ermöglicht eine neue Übertragung (setzt das Bit TI = 0).

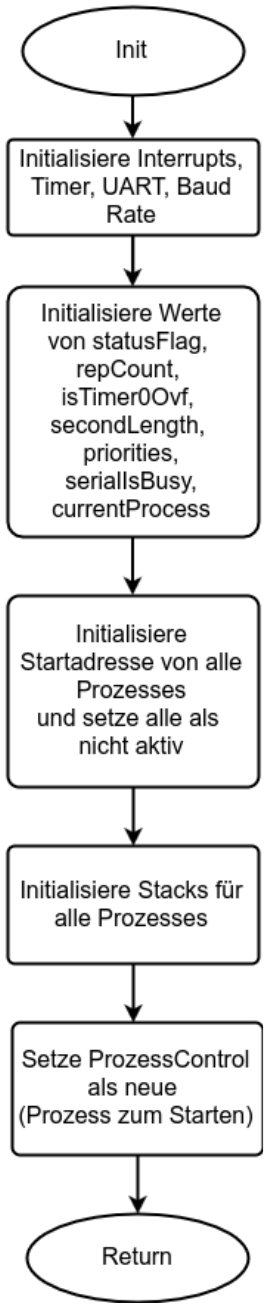
```
1. MOV serialIsBusy, #0
2. ANL A, #11111101b
3. MOV S0CON, A
```

Also, wenn ein Interrupt von dem Timer 1 auftritt, und die Steuerung wird an einen anderen Prozess übergeben werden, der auch ein eigenes Zeichen auf die Serielle Schnittstelle sendet, wird das Zeichen des ersten Prozesses nicht verloren.

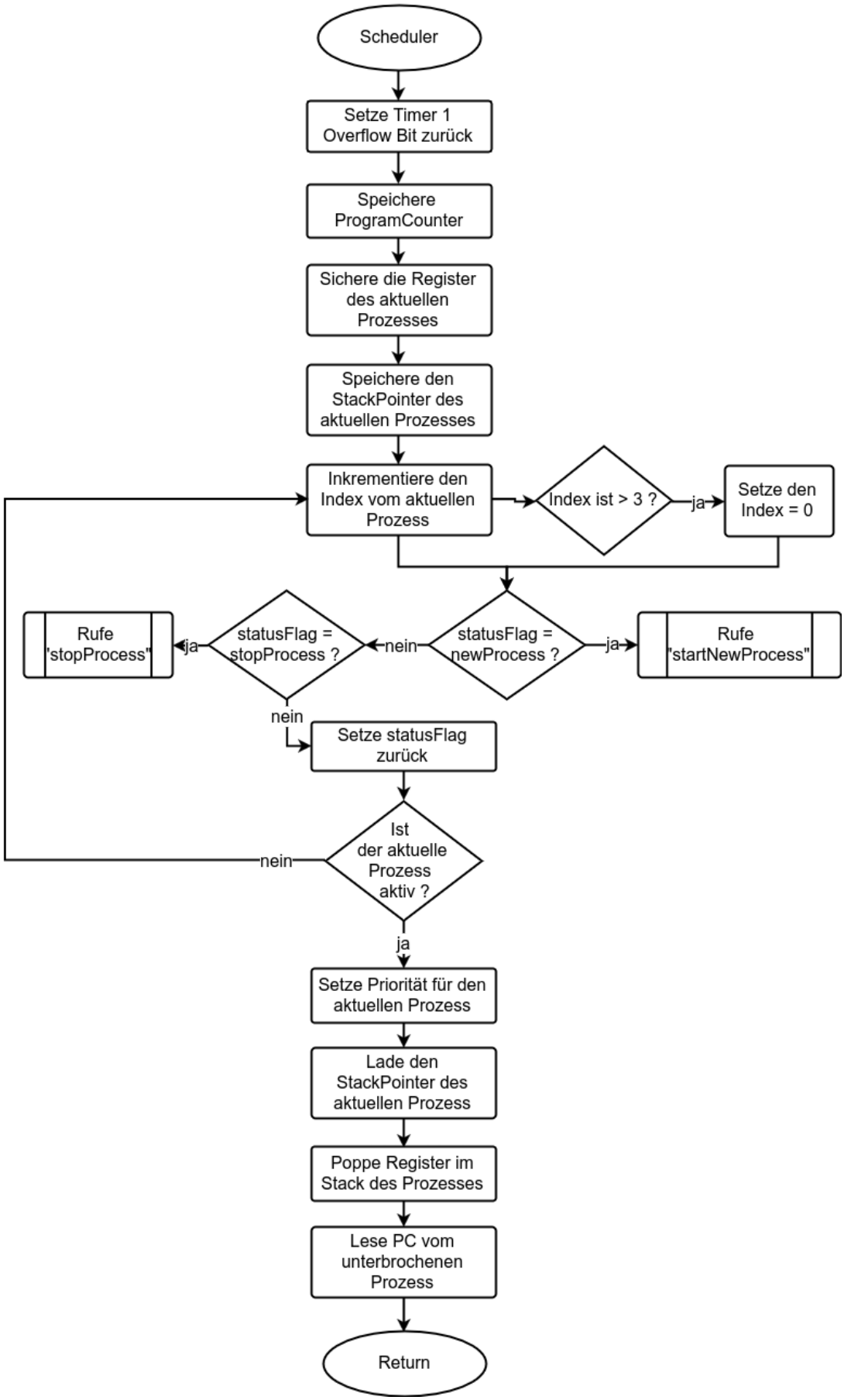
8 Programmfluss (Diagramme)

8.1 Hauptprogramm

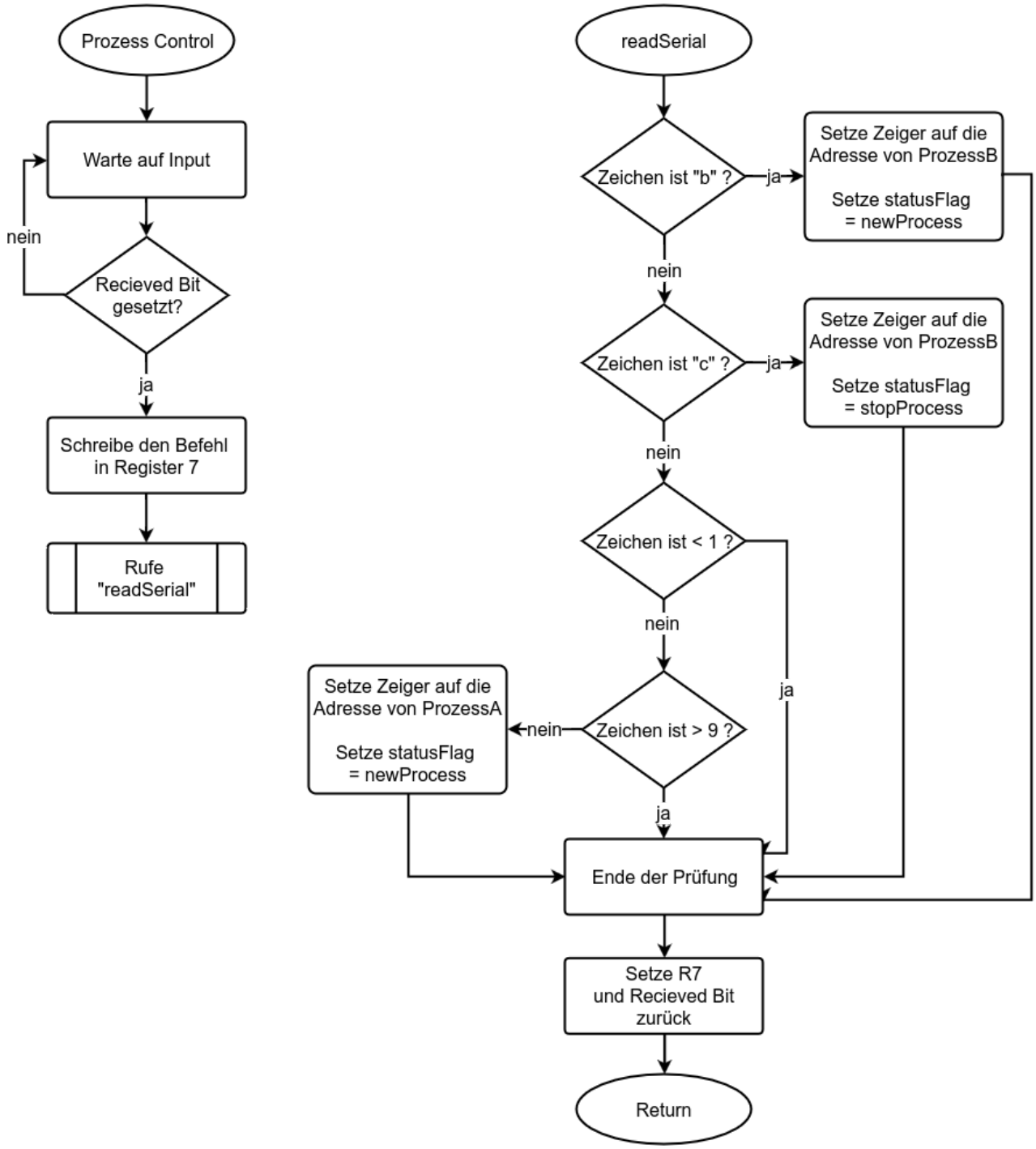




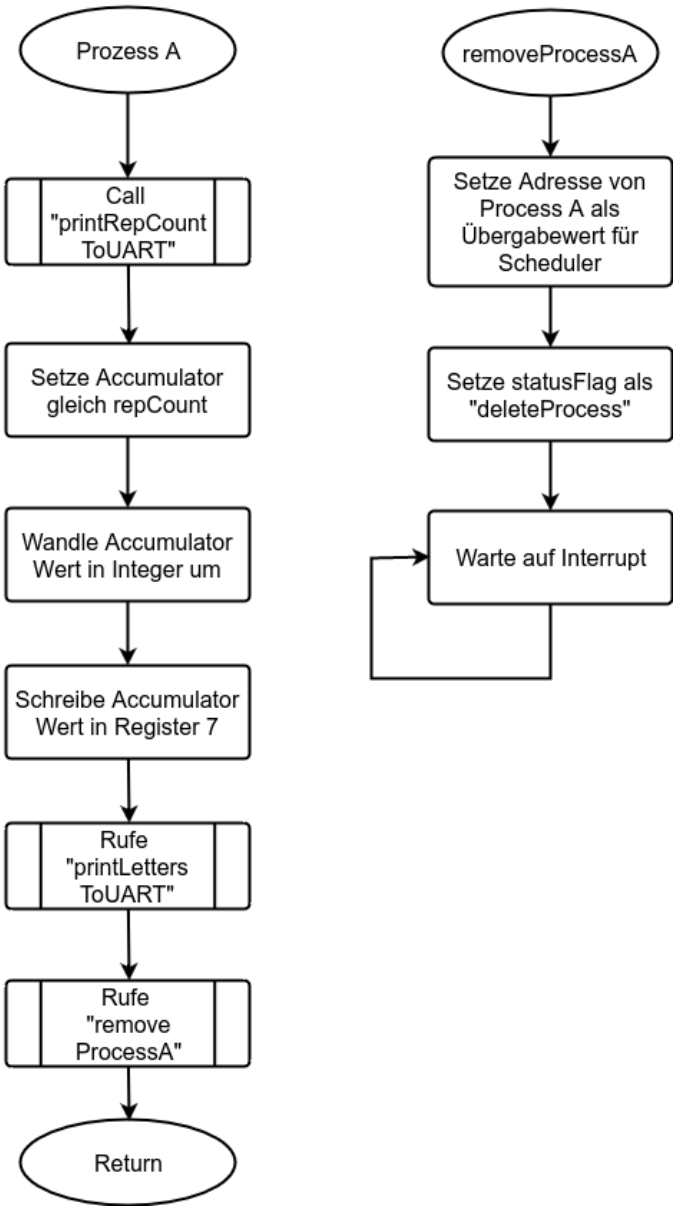
8.2 Scheduler

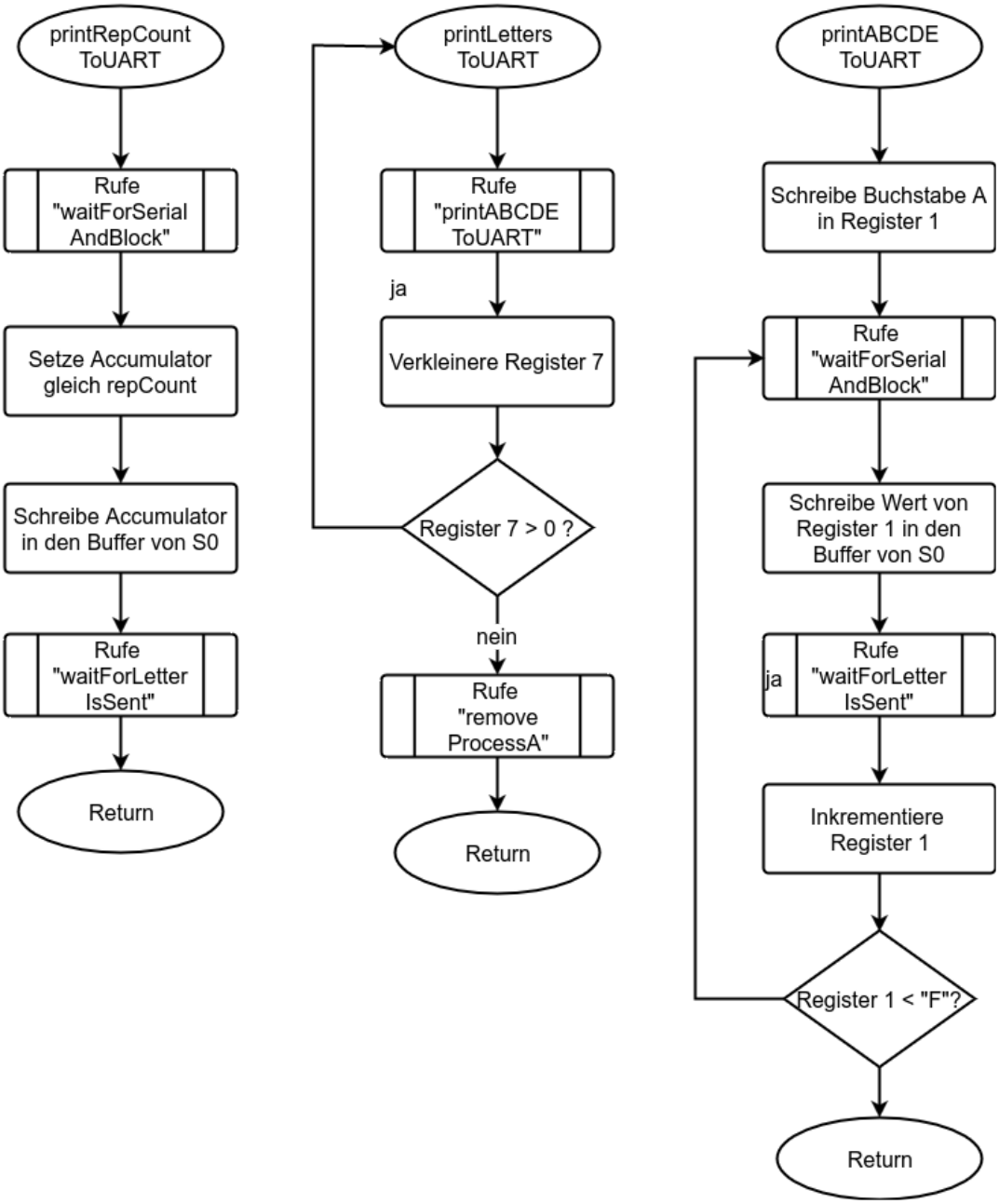


8.3 Prozess Control

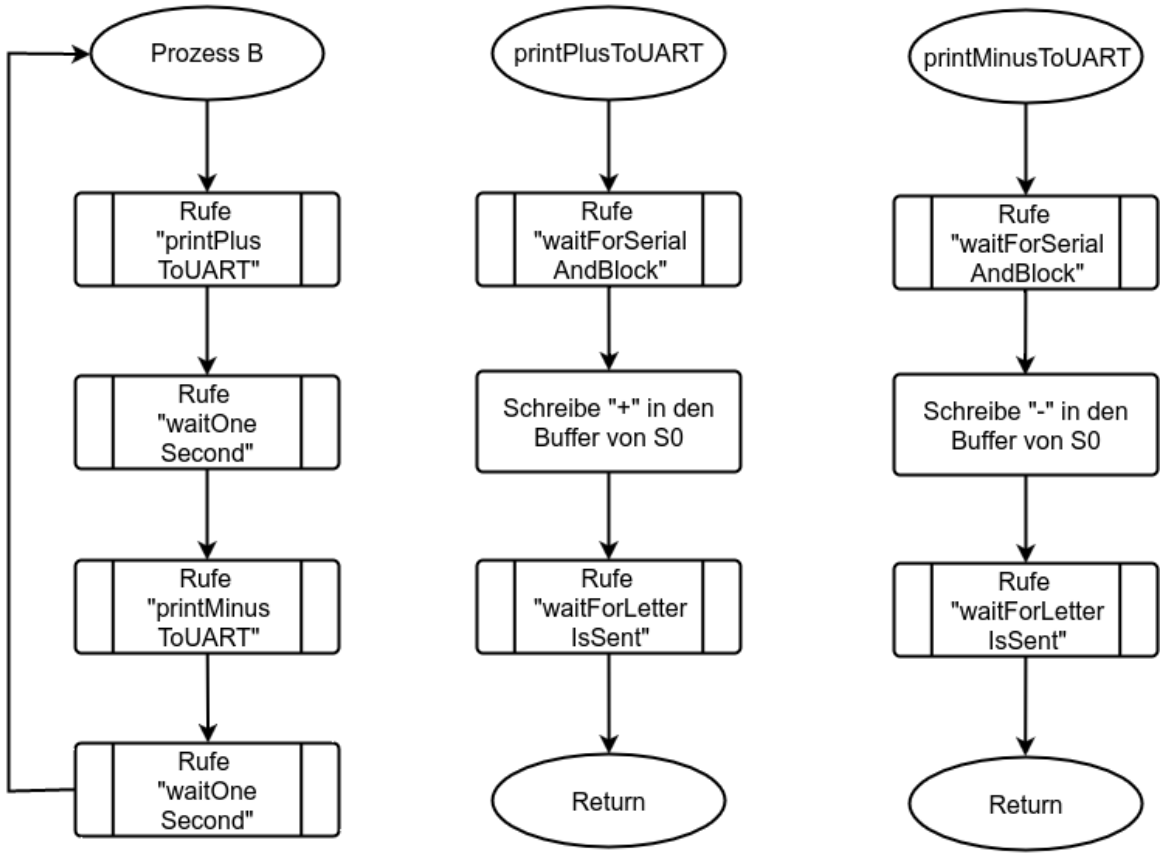


8.4 Prozess A

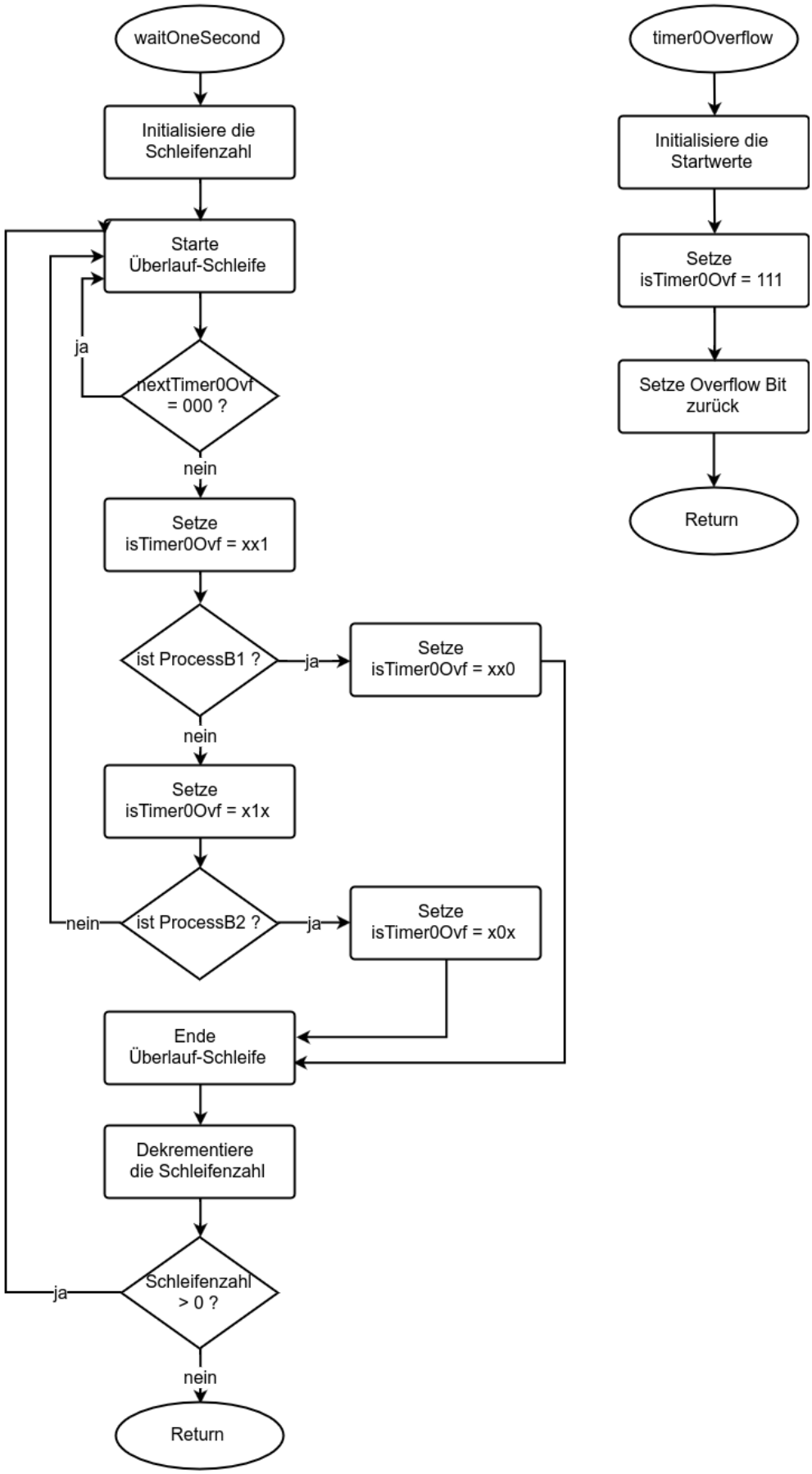

Prozess A
Call "printRepCount ToUART"
Setze Accumulator gleich repCount
Wandle Accumulator Wert in Integer um
Schreibe Accumulator Wert in Register 7
Rufe "printLetters ToUART"
Rufe "remove ProcessA"
Return
removeProcessA
Setze Adresse von Process A als Übergabewert für Scheduler
Setze statusFlag als "deleteProcess"
Warte auf Interrupt


printRepCount ToUART
Rufe "waitForSerial AndBlock"
Setze Accumulator gleich repCount
Schreibe Accumulator in den Buffer von S0
Rufe "waitForLetter IsSent"
Return
printLetters ToUART
Rufe "printABCDE ToUART"
ja
Verkleinere Register 7
Register 7 > 0 ?
nein
Rufe "remove ProcessA"
Return
printABCDE ToUART
Schreibe Buchstabe A in Register 1
Rufe "waitForSerial AndBlock"
Schreibe Wert von Register 1 in den Buffer von S0
ja
Rufe "waitForLetter IsSent"
Inkrementiere Register 1
Register 1 < "F"?
Return

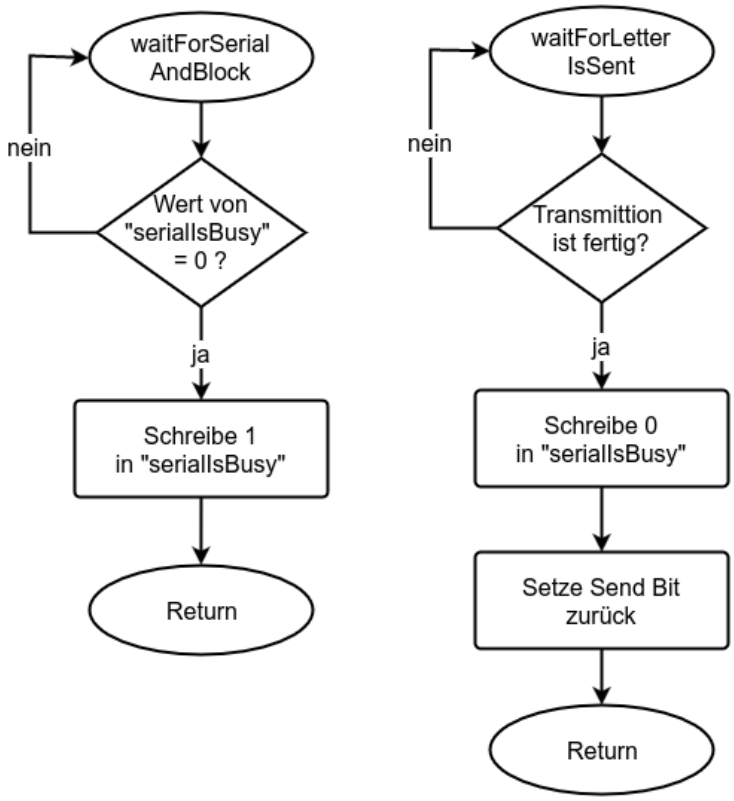
8.5 Prozess B


Prozess B
Rufe "printPlus ToUART"
Rufe "waitOne Second"
Rufe "printMinus ToUART"
Rufe "waitOne Second"
printPlusToUART
Rufe "waitForSerial AndBlock"
Schreibe "+" in den Buffer von S0
Rufe "waitForLetter IsSent"
Return
printMinusToUART
Rufe "waitForSerial AndBlock"
Schreibe "-" in den Buffer von S0
Rufe "waitForLetter IsSent"
Return

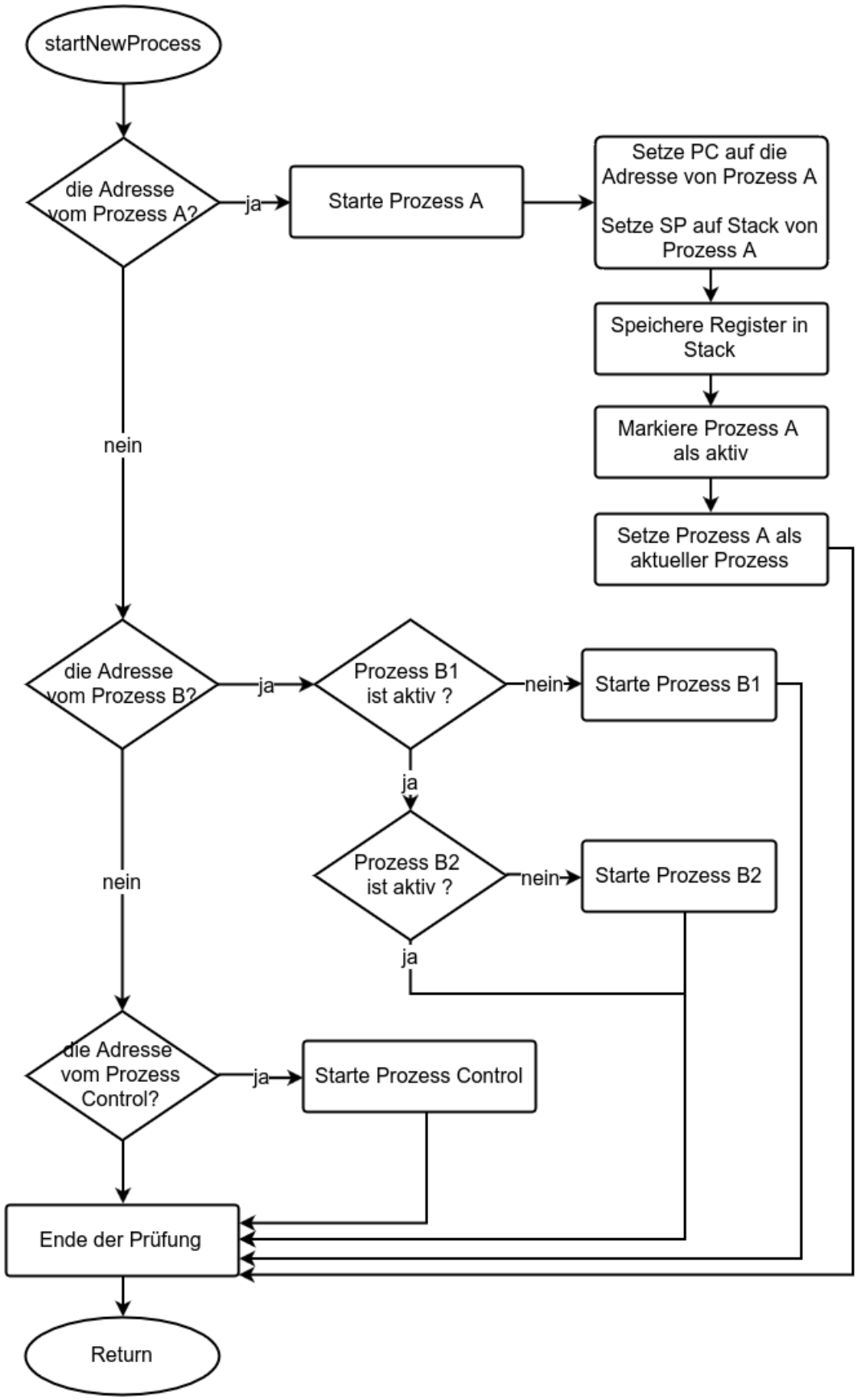
8.6 Eine-Sekunde-Verzögerung



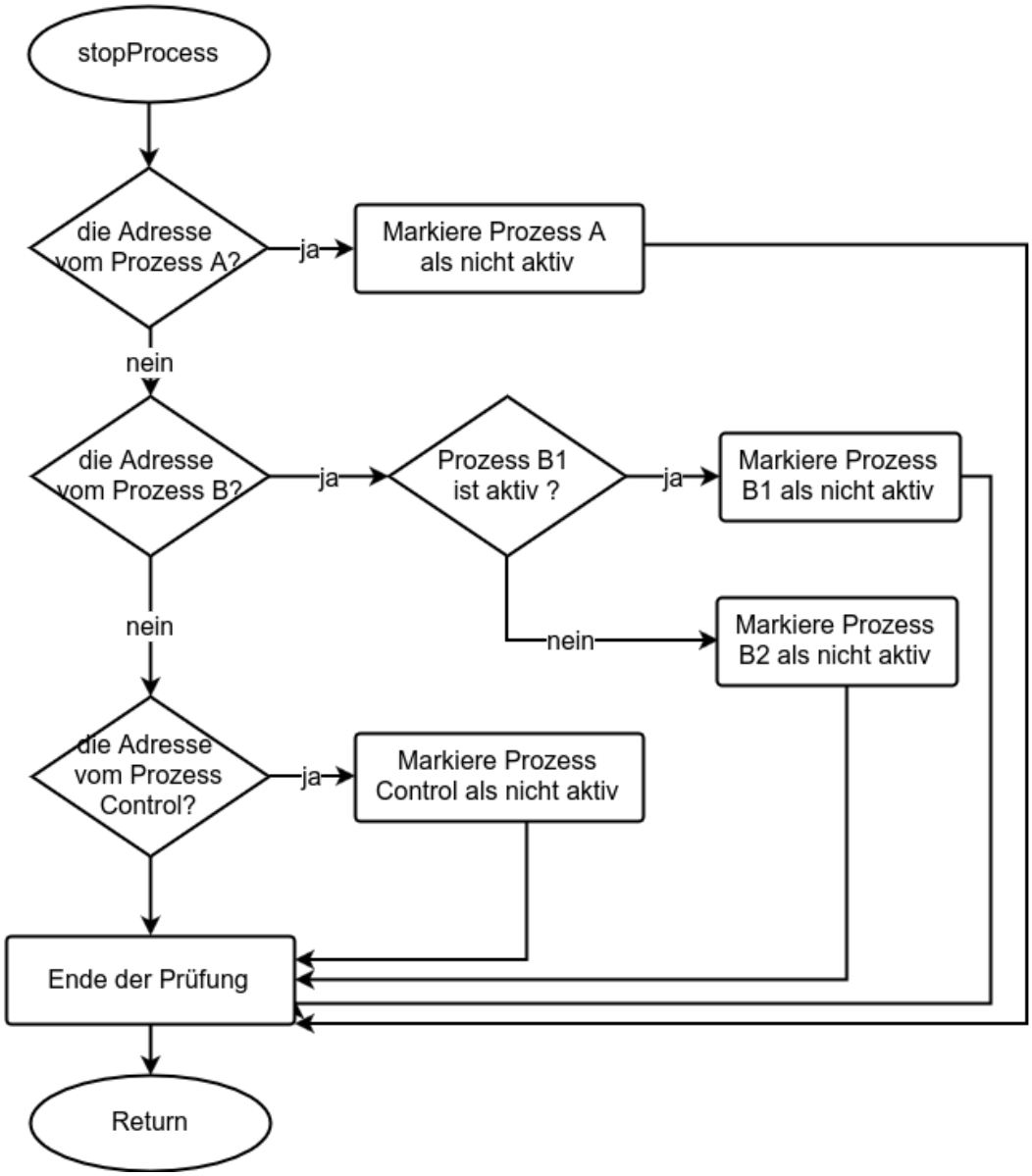
8.7 Hilfsfunktionen fürs Schreiben auf UART



8.8 Start des Prozesses



8.9 Löschung des Prozesses



9 Listing

9.1 Hauptprogramm

```
1.  $NOMOD51
2.  #include <Reg517a.inc>
3.
4.  EXTRN CODE (processA, processB, processControl)
5.  PUBLIC processAddress, statusFlag, stacks
6.  PUBLIC stopProcess, startNewProcess ; helper functions
7.  PUBLIC newProcessFlag, stopProcessFlag, noProcessFlag ; flags
8.  PUBLIC repCount, secondLength, serialIsBusy, isTimer0Ovf ; helper constants
9.  PUBLIC currentProcess
10.
11. ; define status flags values
12. noProcessFlag EQU 0
13. newProcessFlag EQU 1
14. stopProcessFlag EQU 2
15.
16. ; define a relocatable segment of the memory class DATA for the program
17. programData SEGMENT DATA
18.         ; switch to programData segment
19.         RSEG programData
20.
21.         ; Program Counter (Instruction Pointer)
22.         processIPTable: DS 8
23.
24.         ; -------X | -------X | -------X | -------X
25.         ; processA | processB | processB2 | processControl
26.         activeFlags: DS 4
27.         isProcessA EQU activeFlags
28.         isProcessB EQU activeFlags + 1
29.         isProcessB2 EQU activeFlags + 2
30.         isProcessControl EQU activeFlags + 3
31.
32.         ; to tell the scheduler which process has to be started or stopped
33.         processAddress: DS 2
34.         processAddressL EQU processAddress
35.         processAddressH EQU processAddress + 1
36.
37.         ; to tell the scheduler if the process, stored in processAddress,
38.         ; has to be started or stopped.
39.         ; Possible options: newProcessFlag, stopProcessFlag, noProcessFlag
40.         statusFlag: DS 1
41.
42.         ; pointer to the current process
43.         currentProcess: DS 1
44.
```

```
45.         ; to save process priorities in array
46.         priorities: DS 4
47.
48.         ; to save processes StackPointers in array
49.         stacks: DS 4
50.
51.         ; array of initial StackPointers
52.         stackStartAddr: DS 4
53.
54.         ; to store the number, which point how many times to repeat "ABCDE"
55.         repCount: DS 1
56.
57.         ; for the magic number - to make the second length ~1 sec
58.         secondLength: DS 1
59.
60.         ; to check if sending to UART is done
61.         serialIsBusy: DS 1
62.
63.         isTimer0Ovf: DS 1
64.
65.         ; 20 Byte for each process
66.         processData: DS 80
67.
68.
69. ; define the absolute segments for the timer interrupts
70. CSEG AT 1Bh
71. JMP scheduler
72.
73. CSEG AT 0Bh
74. JMP timer0Overflow
75.
76. ; define an absolute segment
77. ; program execution always starts on reset at location 0000
78. CSEG AT 0
79. ; jump to the start of the program
80. JMP start
81.
82. ; define a relocatable segment of the memory class CODE for the program
83. programCode SEGMENT CODE
84.         ; switch to the created relocatable segment
85.         RSEG programCode
86.
87. start:
88.         ; reset watchdog timer
89.         SETB WDT
90.         SETB SWDT
91.
92.         ; set SP to the new stack for the program
93.         MOV SP, #processData
94.         CALL init
95.
```

```
96.       ; initialize Stack Pointer
97.       MOV A, #stacks
98.       ADD A, currentProcess
99.       MOV R0, A
100.      mov SP, @R0
101.
102.      ; endless loop to make the scheduler work forever
103.      endlessStartLoop:
104.            NOP
105.            NOP
106.            ; reset watchdog timer
107.            SETB WDT
108.            SETB SWDT
109.      JMP endlessStartLoop
110.
111.  ; enables interrupts and UARTs, sets timer modes and
112.  ; initializes the needed data
113.  init:
114.      ; enable all interrupts, each interrupt source is individually enabled
115.      SETB EAL
116.      ; enable Timer1 overflow interrupt
117.      SETB ET1
118.      ; enable Timer0 overflow interrupt
119.      SETB ET0
120.
121.      configSerial0:
122.            ; SCON = SM0 | SM1 | SM2 | REN | TB8 | RB8 | TI | RI
123.            ;          0     1     0     1
124.            ; Set Serial0 to Mode 1 (8-bit UART, variable baud rate - mode 1)
125.            CLR SM0
126.            SETB SM1
127.
128.            ; enable receiving
129.            SETB REN0
130.
131.            ; enable programmable baudrate generator
132.            SETB BD
133.
134.            ; set SMOD = 0
135.            MOV A, PCON
136.            CLR ACC.7
137.            MOV PCON, A
138.
139.            ; set baud rate to 14400
140.            MOV S0RELL, #0xE6
141.            MOV S0RELH, #0x03
142.
143.      configTimer:
144.            ; set mode of timer1 and timer0 to Mode 1 - 16-bit (M1=0, M0=1)
145.            ; TMOD = Gate | C/T | M1 | M0 | Gate | C/T | M1 | M0
146.            MOV A, #00010001b
```

```
147.             MOV TMOD, A
148.
149.             ; start timer1
150.             ; TCON = TF1 | TR1 | TF0 | TR0 | IE1 | IT1 | IE0 | IT0
151.             SETB TR1
152.
153.             ; init timer 0
154.             MOV TL0, #0xB0
155.             MOV TH0, #0x3C
156.             ; start timer 0
157.             SETB TR0
158.
159.     ; initialize statusFlag to 0 - there are no processes yet
160.     MOV statusFlag, #noProcessFlag
161.
162.     ; init repeat times with 0
163.     MOV repCount, #0x00
164.
165.     ; init secondLength with magic loop number
166.     MOV secondLength, #40d
167.
168.     MOV isTimer0Ovf, #0x00
169.
170.     ; set that serial port is free
171.     MOV serialIsBusy, #0
172.
173.     ; init priorities for each process
174.     MOV priorities + 0, #0xC0 ;processA
175.     MOV priorities + 1, #0xC0 ;processB
176.     MOV priorities + 2, #0xC0 ;processB2
177.     MOV priorities + 3, #0xE0 ;processControl
178.
179.     ; set currentProcess to the process A row
180.     ; the value will be set to the next one (0) by the scheduler
181.     MOV currentProcess, #3
182.
183.     ; initialize start addresses of the processes and
184.     ; set all processes to inactive mode
185.
186.     ; Process A start address
187.     MOV DPTR, #processA
188.     MOV processIPTable , DPL
189.     MOV processIPTable + 1, DPH
190.     ; Process A is inactive
191.     MOV isProcessA, #0x00
192.
193.     ; Process B start address
194.     MOV DPTR, #processB
195.     MOV processIPTable + 2, DPL
196.     MOV processIPTable + 3, DPH
197.     ; Process B is inactive
```

```
198.      MOV isProcessB, #0x00
199.
200.      ; Process B2 start address
201.      MOV DPTR, #processB
202.      MOV processIPTable + 4, DPL
203.      MOV processIPTable + 5, DPH
204.      ; Process B2 is inactive
205.      MOV isProcessB2, #0x00
206.
207.      ; Process Control start address
208.      MOV DPTR, #processControl
209.      MOV processIPTable + 6, DPL
210.      MOV processIPTable + 7, DPH
211.      ; Process Control is inactive
212.      MOV isProcessControl, #0x00
213.
214.      ; init Stack Pointers for each process
215.      ; process A
216.      MOV A, #processData
217.      MOV stacks, A
218.      MOV stackStartAddr, A
219.      ; process B - processData+20
220.      ADD A, #20
221.      MOV stacks + 1, A
222.      MOV stackStartAddr + 1, A
223.      ; Process B2 - processData+40
224.      ADD A, #20
225.      MOV stacks + 2, A
226.      MOV stackStartAddr + 2, A
227.      ; Process Control - processData+60
228.      ADD A, #20
229.      MOV stacks + 3, A
230.      MOV stackStartAddr + 3, A
231.
232.      ; call ProcessControl
233.      ; set the statusFlag to newProcessFlag
234.      ; to let processControle be started by the scheduler
235.      MOV DPTR, #processControl
236.      MOV processAddress + 0, DPL
237.      MOV processAddress + 1, DPH
238.      MOV statusFlag, #newProcessFlag
239.  RET
```

9.2 Scheduler

```
1. ; timer interrupt that determines the current process
2. ; and starts or deletes processes
3. scheduler:
4.         SETB WDT
5.         SETB SWDT
6.
7.         ; clear Timer 1 overflow flag
8.         CLR TF1
9.
10.        ; save Program Counter (PC, 2 Bytes)
11.        POP DPH
12.        POP DPL
13.        MOV A, #processIPTable
14.        ADD A, currentProcess
15.        ADD A, currentProcess
16.        MOV R0, A
17.        MOV @R0, DPL
18.        INC R0
19.        MOV @R0, DPH
20.
21.        ; push registers to the stack
22.        PUSH PSW
23.        PUSH 0
24.        PUSH 1
25.        PUSH 2
26.        PUSH 3
27.        PUSH 4
28.        PUSH 5
29.        PUSH 6
30.        PUSH 7
31.        PUSH ACC
32.        PUSH B
33.        PUSH DPH
34.        PUSH DPL
35.
36.        ; save StackPointer
37.        MOV A, #stacks
38.        ; INC R0
39.        ADD A, currentProcess
40.        MOV R0, A
41.        MOV @R0, SP
42.
43.        ; iterate until an active process is found
44.        activeProcessLoop:
45.              ; reset watchdog timer
46.              SETB WDT
47.              SETB SWDT
```

```
48.             ; increment currentProcess, initial value = 3
49.             MOV A, currentProcess
50.             INC A
51.             CJNE A, #4, notFirstProcess
52.                     ; reset currentProcess if it already points to the last row
53.                     MOV A, #0
54.             notFirstProcess:
55.                     MOV currentProcess, A
56.
57.             ; check status flags
58.             MOV R0, statusFlag
59.             ; status flag = new
60.             CJNE R0, #newProcessFlag, notNew
61.                     JMP startNewProcess
62.             notNew:
63.             ; status flag = delete
64.             CJNE R0, #stopProcessFlag, notNewNotStop
65.                     JMP stopProcess
66.             notNewNotStop:
67.             ; reset statusFlag
68.             MOV statusFlag, #noProcessFlag
69.
70.             ; check active flag
71.             MOV A, #activeFlags
72.             ADD A, currentProcess
73.             MOV R1, A
74.             CJNE @R1, #0x01, activeProcessLoop
75.
76.     ; set timer according to priority
77.     MOV TL1, #0x00
78.     ; check if process A
79.     CJNE R1, #isProcessA, notProcessA
80.     CLR TR1
81.     MOV TH1, priorities
82.     SETB TR1
83.     notProcessA:
84.     ; check if process B
85.     CJNE R1, #isProcessB, notProcessB
86.             CLR TR1
87.             MOV TH1, priorities + 1
88.             SETB TR1
89.     notProcessB:
90.     ; check if process B2
91.     CJNE R1, #isProcessB, notProcessB2
92.             CLR TR1
93.             MOV TH1, priorities + 2
94.             SETB TR1
95.     notProcessB2:
96.     ; check if process Control
97.     CJNE R1, #isProcessControl, notProcessControl
98.             CLR TR1
```

```
99.             MOV TH1, priorities + 3
100.            SETB TR1
101.      notProcessControl:
102.
103.      ; restore the SP of the current process
104.      loadStackPointer:
105.            MOV A, currentProcess
106.            ADD A, #stacks
107.            MOV R0, A
108.            MOV SP, @R0
109.
110.      ; pop registers from the stack for the current process
111.      popRegisters:
112.            POP DPL
113.            POP DPH
114.            POP B
115.            POP ACC
116.            POP 7
117.            POP 6
118.            POP 5
119.            POP 4
120.            POP 3
121.            POP 2
122.            POP 1
123.            POP 0
124.            POP PSW
125.
126.      MOV A, #processIPTable
127.      ADD A, currentProcess
128.      ADD A, currentProcess
129.      MOV R0, A
130.      MOV DPL, @R0
131.      PUSH DPL
132.      INC R0
133.      MOV DPH, @R0
134.      PUSH DPH
135.   RETI
```

9.3 Eine-Sekunde-Interrupt

```
1.  timer0Overflow:
2.        MOV TL0, #0xB0
3.        MOV TH0, #0x3C ; #15536d
4.        MOV isTimer0Ovf, #0x07 ;111
5.
6.        ; reset overflow flag
7.        CLR TF0
8.  RETI
```

9.4 Start des Prozesses

```
1.  ; called from the scheduler if newProcessFlag flag is set
2.  startNewProcess:
3.          ; reset watchdog timer
4.          SETB WDT
5.          SETB SWDT
6.
7.          ; determine the process to start and set its active flag to 1
8.          processAtoStart:
9.                  ; compare the current process address with the address of processA
10.                 MOV DPTR, #processA
11.                 MOV A,DPH
12.                 CJNE A, processAddressH, processBtoStart
13.                         ; still chance that it's processA
14.                         MOV A, DPL
15.                         CJNE A, processAddressL, processBtoStart
16.                                 ; yes, it's process A to start, set PC
17.                                 MOV processIPTable, DPL
18.                                 MOV processIPTable+1, DPH
19.                                 ; move stack pointer to the begin of the stack
20.                                 MOV SP, stackStartAddr
21.                                 ; push startadress of the process on the stack
22.                                 PUSH PSW
23.                                 PUSH 0
24.                                 PUSH 1
25.                                 PUSH 2
26.                                 PUSH 3
27.                                 PUSH 4
28.                                 PUSH 5
29.                                 PUSH 6
30.                                 PUSH 7
31.                                 PUSH ACC
32.                                 PUSH B
33.                                 PUSH DPH
34.                                 PUSH DPL
35.                                 ; store the changed stackpointer
36.                                 ; and set the active flag of the process to 1
37.                                 MOV stacks, SP
38.                                 MOV isProcessA, #0x01
39.                                 MOV currentProcess, #0x00
40.                                 JMP endStart
41.
42.         processBtoStart:
43.                 ; compare the current process address with the address of processB
44.                 MOV DPTR, #processB
45.                 MOV A, DPH
46.                 CJNE A, processAddressH, processControlToStart
47.                         ; still chance that it's processB
```

```
48.                     MOV A, DPL
49.                     CJNE A, processAddressL, processControlToStart
50.                             ; yes, it's process B to start
51.                             ; check if processB is runnig already
52.                             MOV A, isProcessB
53.                             CJNE A, #0x01, processBNotRunning
54.                                     ; processB is runnig, start a second one
55.                                     ; check if processB2 is runnig
56.                                     checkIfProcessB2Runnig:
57.                                             MOV A, isProcessB2
58.                                             CJNE A, #0x00, endCheckProcessB
59.                                                     ; process B2 is free to use
60.                                                     MOV processIPTable+4, DPL
61.                                                     MOV processIPTable+5, DPH
62.                                                     MOV SP, stackStartAddr + 2
63.                                                     ; push startadress on the stack
64.                                                     PUSH PSW
65.                                                     PUSH 0
66.                                                     PUSH 1
67.                                                     PUSH 2
68.                                                     PUSH 3
69.                                                     PUSH 4
70.                                                     PUSH 5
71.                                                     PUSH 6
72.                                                     PUSH 7
73.                                                     PUSH ACC
74.                                                     PUSH B
75.                                                     PUSH DPH
76.                                                     PUSH DPL
77.                                                     ; store the changed stackpointer
78.                                                     ; and set the active flag
79.                                                     MOV stacks + 2, SP
80.                                                     MOV isProcessB2, #0x01
81.                                                     MOV currentProcess, #0x02
82.                                                     JMP endCheckProcessB
83.
84.                             processBNotRunning:
85.                                     ; processB not runnig, start it, set PC
86.                                     MOV processIPTable+2, DPL
87.                                     MOV processIPTable+3, DPH
88.                                     MOV SP, stackStartAddr + 1
89.                                     ; push startadress of the process on the stack
90.                                     PUSH PSW
91.                                     PUSH 0
92.                                     PUSH 1
93.                                     PUSH 2
94.                                     PUSH 3
95.                                     PUSH 4
96.                                     PUSH 5
97.                                     PUSH 6
98.                                     PUSH 7
```

```
99.                                 PUSH ACC
100.                                PUSH B
101.                                PUSH DPH
102.                                PUSH DPL
103.                                ; store the changed stackpointer
104.                                ; and set the active flag of the process to 1
105.                                MOV stacks + 1, SP
106.                                MOV isProcessB, #0x01
107.                                MOV currentProcess, #0x01
108.                                JMP endCheckProcessB
109.
110.                        endCheckProcessB:
111.                                JMP endStart
112.
113.    processControlToStart:
114.            MOV DPTR, #processControl
115.            MOV A,DPH
116.            CJNE A, processAddressH, endStart
117.                    MOV A,DPL
118.                    CJNE A, processAddressL, endStart
119.                            MOV processIPTable+6, DPL
120.                            MOV processIPTable+7, DPH
121.                            MOV SP, stackStartAddr + 3
122.                            ; push startadress of the process on the stack
123.                            PUSH PSW
124.                            PUSH 0
125.                            PUSH 1
126.                            PUSH 2
127.                            PUSH 3
128.                            PUSH 4
129.                            PUSH 5
130.                            PUSH 6
131.                            PUSH 7
132.                            PUSH ACC
133.                            PUSH B
134.                            PUSH DPH
135.                            PUSH DPL
136.                            ; store the changed stackpointer
137.                            ; and set the active flag of the process to 1
138.                            MOV stacks + 3, SP
139.                            MOV isProcessControl, #0x01
140.                            MOV currentProcess, #0x03
141.                            JMP endStart
142.    endStart:
143.            JMP notNewNotStop
```

9.5 Löschung des Prozesses

```
1.  ; called from the scheduler if stopProcessFlag flag is set
2.  stopProcess:
3.          ; reset watchdog timer
4.          SETB WDT
5.          SETB SWDT
6.
7.          ; determine the process to delete and set its active flag to 0
8.          processAtoStop:
9.                  MOV DPTR, #processA
10.                 MOV A, DPH
11.                 CJNE A, processAddressH, processBtoStop
12.                         MOV A, DPL
13.                         CJNE A, processAddressL, processBtoStop
14.                                 MOV isProcessA, #0x00
15.                                 JMP endStop
16.
17.         processBtoStop:
18.                 MOV DPTR, #processB
19.                 MOV A,DPH
20.                 CJNE A, processAddressH, processControlToStop
21.                         MOV A, DPL
22.                         CJNE A, processAddressL, processControlToStop
23.                                 MOV A, isProcessB
24.                                 CJNE A, #0x01, processB2toStop
25.                                         ; stop Process B1
26.                                         MOV isProcessB, #0x00
27.                                         JMP endStop
28.                                 processB2toStop:
29.                                         ; stop Process B2
30.                                         MOV isProcessB2, #0x00
31.                                         JMP endStop
32.
33.         processControlToStop:
34.                 MOV DPTR, #processControl
35.                 MOV A, DPH
36.                 CJNE A, processAddressH, endStop
37.                         MOV A,DPL
38.                         CJNE A, processAddressL, endStop
39.                                 MOV isProcessControl, #0x00
40.                                 JMP endStop
41.
42.         endStop:
43.                 JMP notNewNotStop
```

9.6 Prozess A

```
1. $NOMOD51
2. #include <Reg517a.inc>
3.
4. NAME processA
5. PUBLIC processA
6. EXTRN DATA (processAddress, statusFlag, repCount, serialIsBusy)
7. EXTRN NUMBER (stopProcessFlag)
8.
9. ; define a relocatable segment of the memory class CODE for the processA
10. processASegment SEGMENT CODE
11.       ; switch to the created relocatable segment
12.       RSEG processASegment
13.
14. processA:
15.       ; print number of loops to UART
16.       CALL printRepCountToUART
17.
18.       ; convert the char '1' to integer 1 and write to R7
19.       MOV A, repCount
20.       SUBB A, #'0'
21.       MOV R7, A
22.
23.       CALL printLettersToUART
24.       CALL removeProcessA
25.
26.
27.  printRepCountToUART:
28.       ; loop until serial port is free
29.       CALL waitForSerialAndBlock
30.
31.       ; print the repeat time number to UART
32.       MOV A, repCount
33.       MOV S0BUF, A
34.
35.       ; loop until output of a character is finished
36.       CALL waitForLetterIsSent
37. RET
38.
39.
40. ; loop for printing of letters to UART
41. printLettersToUART:
42.       ; reset watchdog timer
43.       SETB WDT
44.       SETB SWDT
45.
46.       ; print the characters "abcde"
47.       CALL printABCDEToUART
```

```
48.
49.         ; decrease R7 by 1, print letters until R7 = 0
50.         DJNZ R7, printLettersToUART
51.
52.         ; set stop-flag for the processA
53.         CALL removeProcessA
54. RET
55.
56.
57. ; prints the characters 'abcde' to UART
58. printABCDEToUART:
59.         ; write to R1 ascii value of the character 'a' (#97d)
60.         MOV R1, #'A'
61.
62.         ; loop while R1 < 'F'
63.         loopAtoE:
64.                 CALL waitForSerialAndBlock
65.                 MOV S0BUF, R1
66.
67.                 ; loop until output of a character is finished
68.                 CALL waitForLetterIsSent
69.
70.                 ; increase R1 (letter) by 1
71.                 INC R1
72.
73.                 ; if R1 == #102d ('F') - stop, else - loop further
74.                 CJNE R1, #'F', loopAtoE
75. RET
76.
77.
78. ; wait for serial is free and block it after that
79. waitForSerialAndBlock:
80.         ; while serial is busy do nothing
81.         checkSerialIsBusy:
82.                 MOV A, serialIsBusy
83.         JB ACC.0, checkSerialIsBusy
84.         ; block serial
85.         MOV serialIsBusy, #1
86. RET
87.
88.
89. ; loop until output of a character is finished
90. waitForLetterIsSent:
91.         ; SCON = SM0 | SM1 | SM2 | REN | TB8 | RB8 | TI | RI
92.         ; we need the 1st bit (TI0) to find out if the data transmittion is finished
93.         MOV    A, S0CON
94.         ; if the TI0 is not set --> transmittion not finished --> wait (go to loop)
95.         JNB    ACC.1, waitForLetterIsSent
96.         ; free serial
97.         MOV serialIsBusy, #0
98.         ; reset TI0 (Serial port transmitter interrupt flag) for the next output
```

```
99.       ANL A, #11111101b
100.      MOV S0CON, A
101.  RET
102.
103.
104.  removeProcessA:
105.      ; stop processA
106.      MOV DPTR, #processA
107.      MOV processAddress + 0, DPL
108.      MOV processAddress + 1, DPH
109.      MOV statusFlag, #stopProcessFlag
110.
111.      ; loop until processor time of processA is over
112.      endlessLoop:
113.            NOP
114.            NOP
115.      JMP endlessLoop
116.  RET
117.
118.  END
```

9.7 Prozess B

```
1.  $NOMOD51
2.  #include <Reg517a.inc>
3.
4.  NAME processB
5.
6.  EXTRN DATA (secondLength, serialIsBusy, isTimer0Ovf, currentProcess)
7.  PUBLIC processB
8.
9.  ; define a relocable segment of the memory class CODE for the processB
10. processBSegment SEGMENT CODE
11.         ; switch to the created relocable segment
12.         RSEG processBSegment
13.
14.
15. processB:
16.         CALL printPlusToUART
17.         CALL waitOneSecond
18.         CALL printMinusToUART
19.         CALL waitOneSecond
20.         JMP processB
21.
22.
23. ; write the character '+' to UART
24. printPlusToUART:
25.         CALL waitForSeriaAndBlock
26.         MOV S0BUF, #'+' ;#43d
27.         CALL waitForLetterIsSent
28.         MOV A, currentProcess
29. RET
30.
31.
32. ; write the character '-' to UART
33. printMinusToUART:
34.         CALL waitForSeriaAndBlock
35.         MOV S0BUF, #'-' ;#45d
36.         CALL waitForLetterIsSent
37.         MOV A, currentProcess
38. RET
39.
40.
41. ; wait until serial is free and block it after that
42. waitForSeriaAndBlock:
43.         ; while serial is busy do nothing
44.         checkSerialIsBusy:
45.                 MOV A, serialIsBusy
46.         JB ACC.0, checkSerialIsBusy
47.         ; block serial
```

```
48.         MOV serialIsBusy, #1
49.     RET
50.
51.
52.     ; loop until output of a character is finished
53.     waitForLetterIsSent:
54.         ; SCON = SM0 | SM1 | SM2 | REN | TB8 | RB8 | TI | RI
55.         ; we need the 1st bit (TI0) to find out if the data transmittion is finished
56.         MOV    A, S0CON
57.         ; if the TI0 is not set --> transmittion not finished --> wait (go to loop)
58.         JNB    ACC.1, waitForLetterIsSent
59.         ; free serial
60.         MOV serialIsBusy, #0
61.         ; reset TI0 (Serial port transmitter interrupt flag) for the letters
62.         ANL A, #11111101b
63.         MOV S0CON, A
64.     RET
65.
66.
67.     ; wait for ~1 second - 40 cycles, 0,025 sec each
68.     waitOneSecond:
69.         MOV R5, secondLength ; #40d
70.
71.         ; wait for Timer0 overflow
72.         timerOverflowLoop:
73.             MOV A, isTimer0Ovf ; default isTimer0Ovf = 000
74.             CJNE A, #0, nextTimer0Ovf
75.                 JMP notTimer0OvfProcessB2 ; start loop again
76.             nextTimer0Ovf:
77.                 ANL A, #0x01 ; set isTimer0Ovf = xx1
78.                 CJNE A, currentProcess, notTimer0OvfProcessB
79.                     ; current process is 01 (first ProcessB)
80.                     MOV A, isTimer0Ovf
81.                     ANL A, #0xFE ; set isTimer0Ovf = xx0
82.                     MOV isTimer0Ovf, A
83.                     JMP endTimer0OvfProcess ; end overflow for ProcessB1
84.             notTimer0OvfProcessB:
85.                 MOV A, isTimer0Ovf
86.                 ANL A, #0x02 ; set isTimer0Ovf = x1x
87.                 ; if isTimer0Ovf = 000 - another process is working
88.                 JZ notTimer0OvfProcessB2
89.                 CJNE A, currentProcess, notTimer0OvfProcessB2
90.                     ; current process is 02 (second ProcessB)
91.                     MOV A, isTimer0Ovf
92.                     ANL A, #0xFD ; set isTimer0Ovf = x0x
93.                     MOV isTimer0Ovf, A
94.                     JMP endTimer0OvfProcess ; end overflow for ProcessB2
95.             notTimer0OvfProcessB2:
96.         JMP timerOverflowLoop
97.         endTimer0OvfProcess:
98.
```

```
99.       ; reset watchdog
100.      SETB WDT
101.      SETB SWDT
102.
103.      ; return to a new overflow loop if R5 is not yet = 0
104.      DJNZ R5, timerOverflowLoop
105.  RET
106.
107.  END
```

9.8 Prozess Control

```
1.  $NOMOD51
2.  #include <Reg517a.inc>
3.
4.  NAME processControl
5.
6.  EXTRN CODE (processA, processB)
7.  EXTRN DATA (processAddress, statusFlag, repCount)
8.  EXTRN NUMBER (newProcessFlag, stopProcessFlag)
9.  PUBLIC processControl
10.
11. ; define a relocable segment of the memory class CODE for the processBControl
12. processControlSegment SEGMENT CODE
13.         ; switch to the created relocable segment
14.         RSEG processControlSegment
15.
16. processControl:
17.         ; reset watchdog timer
18.         SETB WDT
19.         SETB SWDT
20.
21.         ; wait for input on UART
22.         waitForInput:
23.         JNB RI0, waitForInput
24.
25.         ; read input and call the serial handler
26.         MOV R7, S0BUF
27.         CALL readSerial
28.
29. ; start or delete processes according to a received charachter
30. readSerial:
31.         checkLetterB:
32.         CJNE R7, #'b', checkLetterC
33.                 ; start processB
34.                 MOV DPTR, #processB
35.                 MOV processAddress + 1, DPH
36.                 MOV processAddress + 0, DPL
37.                 MOV statusFlag, #newProcessFlag
38.                 JMP noValidProcess
39.
40.         checkLetterC:
41.         CJNE R7, #'c', checkNumbers
42.                 ; delete processB
43.                 MOV DPTR, #processB
44.                 MOV processAddress + 1, DPH
45.                 MOV processAddress + 0, DPL
46.                 MOV statusFlag, #stopProcessFlag
```

```
47.             JMP noValidProcess
48.
49.       checkNumbers:
50.             ; check input on R7 and set parameters accordingly
51.             MOV A, R7
52.             ; clear CY (carry)
53.             CLR C
54.             ; if less than '1' - jump to noValidProcess
55.             SUBB A, #'1'
56.             JC noValidProcess
57.             ; if greater than '9' - jump to noValidProcess
58.             SUBB A, #0x09
59.             ; jump if the carry flag is not set
60.             JNC noValidProcess
61.                   ; start processA
62.                   MOV repCount, R7
63.                   MOV DPTR, #processA
64.                   MOV processAddress + 1, DPH
65.                   MOV processAddress + 0, DPL
66.                   MOV statusFlag, #newProcessFlag
67.
68.       noValidProcess:
69.             ; reset R7
70.             MOV R7, #0x0
71.             ; reset receiver interrupt flag
72.             CLR RI0
73.             ; wait further
74.             JMP processControl
75. END
```