Berufsgenossenschaftliches
Institut für Arbeitsschutz – BGIA

BGIA-Report 7/2006

Selbsttests für Mikroprozessoren mit Sicherheitsaufgaben

oder:

„Quo vadis Fehler?“

HVBG
Hauptverband der
gewerblichen
Berufsgenossenschaften

Verfasser: Mario Mai, Günter Reuß
Berufsgenossenschaftliches Institut für Arbeitsschutz – BGIA
Sankt Augustin

Herausgeber: Hauptverband der gewerblichen Berufsgenossenschaften (HVBG)
Berufsgenossenschaftliches Institut für Arbeitsschutz – BGIA
Alte Heerstr. 111, D-53754 Sankt Augustin
Telefon: +49 / 02241 / 231 – 01
Telefax: +49 / 02241 / 231 – 1333
Internet: www.hvbg.de
– November 2006 –

ISBN: 3-88383-713-X

ISSN: 0173-0387

Selbsttests für Mikroprozessoren mit Sicherheitsaufgaben
oder:
„Quo vadis Fehler“?

Kurzfassung

Mit zunehmendem Einsatz von Mikroprozessoren in sicherheitstechnischen Produkten, wie Steuerungen und Sensoren, entstanden auch besondere Anforderungen an deren Sicherheit. Die Reaktion der Steuerung im Fehlerfall muss deterministisch sein und gefährliche Aktionen müssen verhindert werden können. Dieser Report beschreibt für den Teil der zusätzlichen Maßnahmen, die beim Bau sicherer Steuerungen getroffen werden müssen, Rechnertests, um diese Systeme für eine sicherheitstechnische Anwendung zu ertüchtigen. Bei den hier vorgestellten Maßnahmen handelt es sich um Softwaremodule in der Programmiersprache Assembler. Damit kann in Anlehnung an die in den Normen geforderten Fehleraufdeckungen, zusammen mit der jeweils verwendeten Systemarchitektur (Struktur), die notwendige Sicherheit erreicht werden. Die Beispiele sind Momentaufnahmen möglicher Lösungen, die aber keinen Anspruch auf Vollständigkeit erheben können.

Self-tests for microprocessors incorporating safety functions
or:
"Quo vadis, fault?"

Abstract

The increasingly widespread use of microprocessors in safety-related products such as controls and sensors has led to particular requirements being placed upon their safety. The response of the controller in the event of a fault must be deterministic, and hazardous operations must be preventable. For such supplementary measures which are to be taken during the design of safe controls, this report describes processor tests by which the systems concerned are to be made suitably robust for safety-related applications. The measures presented here are software modules written in the Assembler programming language. Based upon the arrangements for error detection required by the standards, these measures enable the required level of safety to be attained in conjunction with the system architecture (structure) employed. The measures described represent snapshots of possible solutions, and should be regarded as examples only.

Tests automatiques pour les microprocesseurs ayant des tâches de sécurité
Ou :
« Défaut quo vadis » ?

Résumé

L'utilisation croissante de microprocesseurs pour les produits dédiés à la sécurité, comme les commandes et les capteurs, a entraîné des exigences particulières concernant leur sécurité. La réaction de la commande en cas de défaut doit être assurée et les opérations dangereuses doivent pouvoir être empêchées. En ce qui concerne les mesures supplémentaires devant être prises pour la construction de commandes sûres, ce rapport décrit les tests automatisés permettant une utilisation de ces systèmes dans des applications de sécurité. Les mesures présentées ici traitent de modules logiciels dans le langage de programmation Assembleur. La sécurité nécessaire peut ainsi être atteinte selon les découvertes de défauts exigées dans les normes, avec l'architecture de système utilisée dans chaque cas. Les exemples sont des épreuves instantanées de résolutions possibles qui ne peuvent cependant pas prétendre être exhaustifs.

Autoverificación de microprocesadores dotados de funciones de seguridad o:

¿„Quo vadis error“?

Resumen

Con el creciente empleo de microprocesadores en aplicaciones relevantes para la seguridad, como mandos y sensores, también surgen exigencias específicas en razón a su seguridad. La reacción del mando en caso de fallo deberá ser determinística y acciones peligrosas deberán poderse evitar. El presente Report expone pruebas por ordenador para aquellas medidas adicionales, que deberán emprenderse a la hora de fabricar mandos seguros, a fin de habilitar semejantes sistemas para aplicaciones en razón de la seguridad. Las medidas presentadas se refieren a módulos de software en el lenguaje de programación Assembler. Con ello, y en conjunto con la respectiva arquitectura de sistema (estructura) utilizada, se puede alcanzar la seguridad requerida en conformidad con la detección de errores exigida por la normativa. Los ejemplos presentados son instantáneas de posibles soluciones, que no pretenden brindar respuestas a todas las interrogantes del caso.

Inhaltsverzeichnis

1 Einleitung 9

2 Arten von Selbsttests 11

2.1 Rechnertests 13

2.2 Peripherietests 13

3 Tests interner Blöcke und Einheiten des Prozessors 15

3.1 Grundlegende Tests 15

3.1.1 Programmzählertest (PC_TEST.ASM) 15

3.1.2 Akkutest (ACC_TEST.ASM) 17

3.1.3 Stackbefehlstest PUSH-, POP- und RET-Befehl (PPR_TEST.ASM) 18

3.2 Weitergehende Befehlstests 20

3.2.1 Jump if not Zero (JNZ_TEST.ASM) 20

3.2.2 Arithmetikbefehle (ARI_TEST.ASM) 23

3.2.3 Logikbefehle (ANL_TEST.ASM, ORL_TEST.ASM, XRL_TEST.ASM und CRS_TEST.ASM) 24

3.2.4 Logikbefehle (BIT_TEST.ASM) 25

3.2.5 Transferbefehle (TRANTEST.ASM) 28

4 Speichertests 31

4.1 Programmspeichertest (ROM_TEST.ASM) 31

4.2 Datenspeichertest (XRAMTEST.ASM) 34

5 Special-Function-Register-Test (SFR_TEST.ASM) 39

6 Porttests (IO_TEST.ASM) 41

7 Hauptprogramm 45

8 Schlussbemerkung 47

9 Literaturverzeichnis 49

1 Einleitung

Aus der modernen Technik ist der Mikroprozessor nicht mehr wegzudenken und für den Einsatz in komplexen Steuerungen [1] und modernen Schutzeinrichtungen, wie für den Personenschutz an Maschinen und Anlagen, ist seine Verwendung selbstverständlich geworden. Nur durch die Verwendung solcher Komponenten lassen sich Flexibilität, Vielseitigkeit, Zuverlässigkeit, Anpassbarkeit an den Produktionsprozess und nicht zuletzt auch akzeptable Kosten erreichen.

Um einen Mikroprozessor, der ein sehr komplexes Bauteil ist und dessen Verhalten im Fehlerfall nicht definiert werden kann, für den sicherheitstechnischen Einsatz zu ertüchtigen, müssen Maßnahmen getroffen werden, mit deren Hilfe Ausfälle im Betrieb aufgedeckt und entsprechende sicherheitsgerichtete Reaktionen eingeleitet werden können. In der Praxis bedeutet dies, dass neben der Funktion eine permanente Selbsttestung des Prozessors erfolgt. Bei dieser Testung werden bestimmte Operationen ausgeführt und die Ergebnisse mit einer Erwartungshaltung verglichen. Eine festgestellte Abweichung von der Erwartungshaltung muss zu einer definierten, in der Regel sicherheitsgerichteten, Reaktion führen.

Um die verschiedenen Verarbeitungseinheiten und Peripherieelemente eines Prozessorsystems zu testen, sind für diese Elemente jeweils spezielle Tests erforderlich, wie beispielsweise ein Programmspeichertest, ein Datenspeichertest, Befehlstests usw. Nach mängelfreiem Durchlauf aller Tests wird davon ausgegangen, dass das System hinreichend fehlerfrei ist.

Die in diesem Report vorgestellten Tests wurden auf einem Mikrocontroller des Typs 80C537 entwickelt; sie sind modular programmiert und kommentiert. Da von den besonderen Hardwareeigenschaften dieses Controllers bei der Entwicklung im Wesentlichen kein Gebrauch gemacht wurde, sollten die Tests ohne große Schwierigkeiten auf andere Derivate dieser weit verbreiteten Prozessorfamilie übertragbar sein. Eine eventuelle Portierung auf andere Prozessorarchitekturen sollte ebenfalls keine Probleme bereiten, wobei natürlich auf die ggf. größeren Hardwareunterschiede und damit

notwendige Anpassungen der Software zu achten ist. Die grundsätzlichen Überlegungen, die hinter dem Design dieser Rechnertests stehen, sollen wertvolle Anregungen oder eine Hilfestellung geben. Hinweise zur Wirksamkeit und Häufigkeit der Selbsttests bietet die Norm IEC 61508/VDE 0803 [2].

Die Software wurde gemäß dem Stand von Wissenschaft und Technik sorgfältig erstellt. Sie wird dem Nutzer unentgeltlich zur Verfügung gestellt.

Die Benutzung der Software erfolgt auf eigene Gefahr. Eine Haftung – gleich aus welchem Rechtsgrund – ist, soweit gesetzlich zulässig, ausgeschlossen. Insbesondere für Sach- und Rechtsmängel der Software sowie der damit zusammenhängenden Dokumentationen und Informationen wird – vor allem im Hinblick auf deren Richtigkeit, Fehlerfreiheit, Freiheit von Schutz- und Urheberrechten Dritter, Aktualität, Vollständigkeit und/oder Verwendbarkeit – außer bei Vorsatz oder Arglist nicht gehaftet.

Das Berufsgenossenschaftliche Institut für Arbeitsschutz – BGIA ist bemüht, seine Homepage virenfrei zu halten, gleichwohl kann keine Virenfreiheit der zur Verfügung gestellten Software und Informationen zugesichert werden. Dem Nutzer wird daher empfohlen, vor dem Herunterladen von Software, Dokumentationen oder Informationen selbst für angemessene Sicherheitsvorkehrungen und Virenscanner zu sorgen.

Die Software wurde durch Mitarbeiter(innen) im BGIA ohne Kontakt zu Kunden und deren Applikationen entwickelt, um auf der Basis von Anforderungen der gültigen Normen einsetzbare Beispiele für Nachfragen zu haben.

2 Arten von Selbsttests

Die hier realisierten Tests lassen sich grundsätzlich in zwei Gruppen einteilen: Zum einen in die so genannten Rechnertests, welche die korrekte Funktion der CPU (Central Processing Unit), der chipinternen Peripherieelemente sowie des internen und/oder externen RAM (Random Access Memory) testen, zum anderen in Tests, die die Anbindung des Prozessors zur Außenwelt (z. B. über die Ports) auf Fehlerfreiheit testen (Peripherietests). Während Rechnertests relativ unabhängig von der jeweiligen praktischen Anwendung sind, müssen z. B. für Porttests extern angeschlossene zu steuernde Einheiten Berücksichtigung finden. Die jeweilige Hardware wird fester Bestandteil dieser Peripherietests und deshalb können deren Eigenschaften, wie z. B. Filtereffekte, unmittelbaren Einfluss auf die Realisierung eines solchen Porttests haben [3].

Abbildung 1 (siehe Seite 12) zeigt die Prinzipdarstellung einer in der Sicherheitstechnik verbreiteten Rechnerstruktur. Das Rechnersystem besteht aus dem Prozessor, dem Herzstück des Systems, dem ROM (Read only Memory), in dem das Programm für den Prozessor abgespeichert ist, und dem RAM, in dem Daten während des Betriebs abgelegt werden. Optional ist auch ein nicht flüchtiger wiederbeschreibbarer Speicherbaustein möglich; dies ist abhängig von der Anwendung und in diesem Beispielsystem nicht berücksichtigt. Das Beispielsystem ist über geeignete Schnittstellen interaktiv mit einer zu steuernden Hardware verbunden.

Eine wesentliche Komponente im dargestellten Rechnersystem stellt der Watchdog (WD) dar, der die Funktion des Prozessors überwacht. Für diese Struktur ist es notwendig, dass der WD über einen unabhängigen Abschaltweg verfügt, mit dessen Hilfe die zu steuernde Hardware unabhängig vom Zustand des Prozessors in den sicheren Zustand geschaltet werden kann. Hierbei wird vorausgesetzt, dass das System einen sicheren Zustand hat, der durch Abschalten der Energie erreicht werden kann. Der WD hat einen eigenen, vom Prozessor unabhängigen Takt, da die Unabhängigkeit des WD für die Abschaltung bei gemeinsamer Taktversorgung nicht gewährleistet ist.

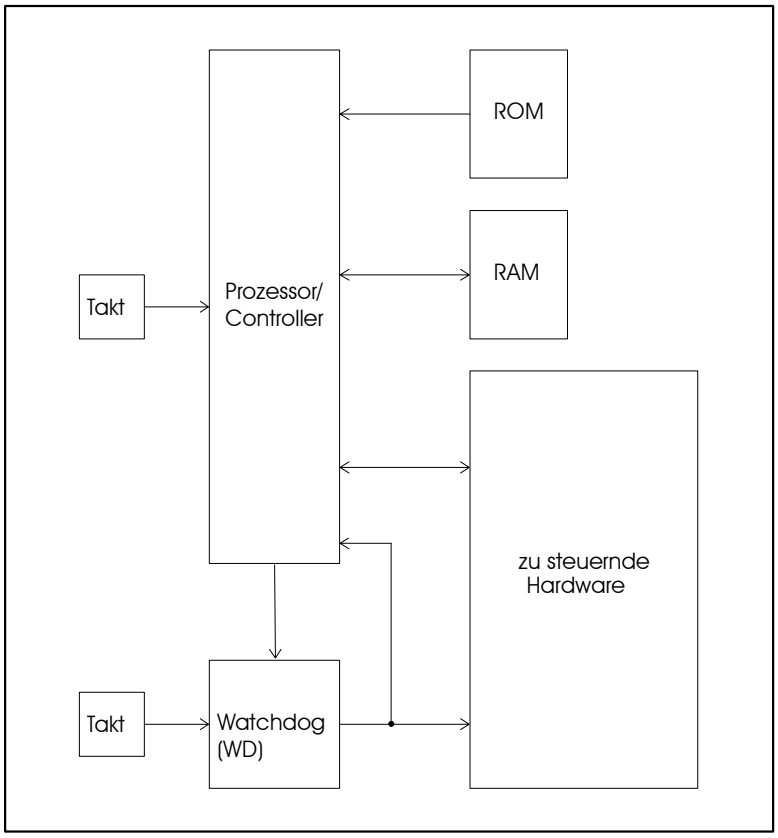


Abbildung 1:
Allgemeine Blockstruktur
eines Rechnersystems

Ein anderes, hier nicht sichtbares aber wichtiges Merkmal des WD ist, dass dieser durch ein Zeitfenster vor Mehrfachtriggerungen innerhalb des Zeitfensters geschützt wird. Durch die Wahl des Zeitfensters kann eine engere Kopplung an das Programm des Prozessors erreicht werden. Bei einer Mehrfachtriggerung innerhalb des Zeitfensters könnte auf eine fehlerhafte Abarbeitung des Programms geschlossen werden und wie beim Ausbleiben der Triggerimpulse der sichere Zustand eingeleitet werden.

Die Struktur des Rechnersystems in Abbildung 1 kann im Hinblick auf die Lokalität der Speicher variieren. Für die Gestaltung der Selbsttests hätte es keinen wesentlichen Einfluss, ob die Speicher (RAM, ROM) zusammen auf dem Chip integriert sind.

Grundsätzlich gilt, dass die Steuerung bei einem Selbsttest mit negativem Ergebnis den sicheren Zustand einer Maschine oder Anlage einleiten oder aufrechterhalten muss.

2.1 Rechnertests

Im Folgenden finden sich Beispiele zu folgenden Rechnertests:

- ❒ Grundlegende Tests
  - Akkutest, Test elementarer Sprungbefehle
  - Stackbefehlstest
  - Programmzählertest (Sprung auf Programminseln)
- ❒ Weitergehende Befehlstests
  - Test der Transferbefehle
  - Test der logischen Operationen
  - Test der arithmetischen Operationen
- ❒ Speichertests
  - Programmspeichertest (ROM-Test)
  - Datenspeichertest (RAM-Test)
- ❒ Special-Function-Register-Test
  - Test der Register zur Steuerung interner Peripheriekomponenten (Timer, Zähler usw.)

2.2 Peripherietests

Zu den Peripherietests gehören z. B. der

- ❒ Test der korrekten Funktion der I/O-Ports
- ❒ Test der externen Peripherie

3 Tests interner Blöcke und Einheiten des Prozessors

In einigen der folgenden Abschnitte sind zur Verdeutlichung der wichtigsten Zusammenhänge Ausschnitte aus den Assemblerquelltexten dargestellt. Zur Erklärung von Variablen, Konstanten und Funktionen sei auf den vollständigen Quelltext hingewiesen; hierzu existiert eine Deklarationsdatei (DEC.ASM), in der die Konstanten und Speicherplätze der Variablen definiert werden.

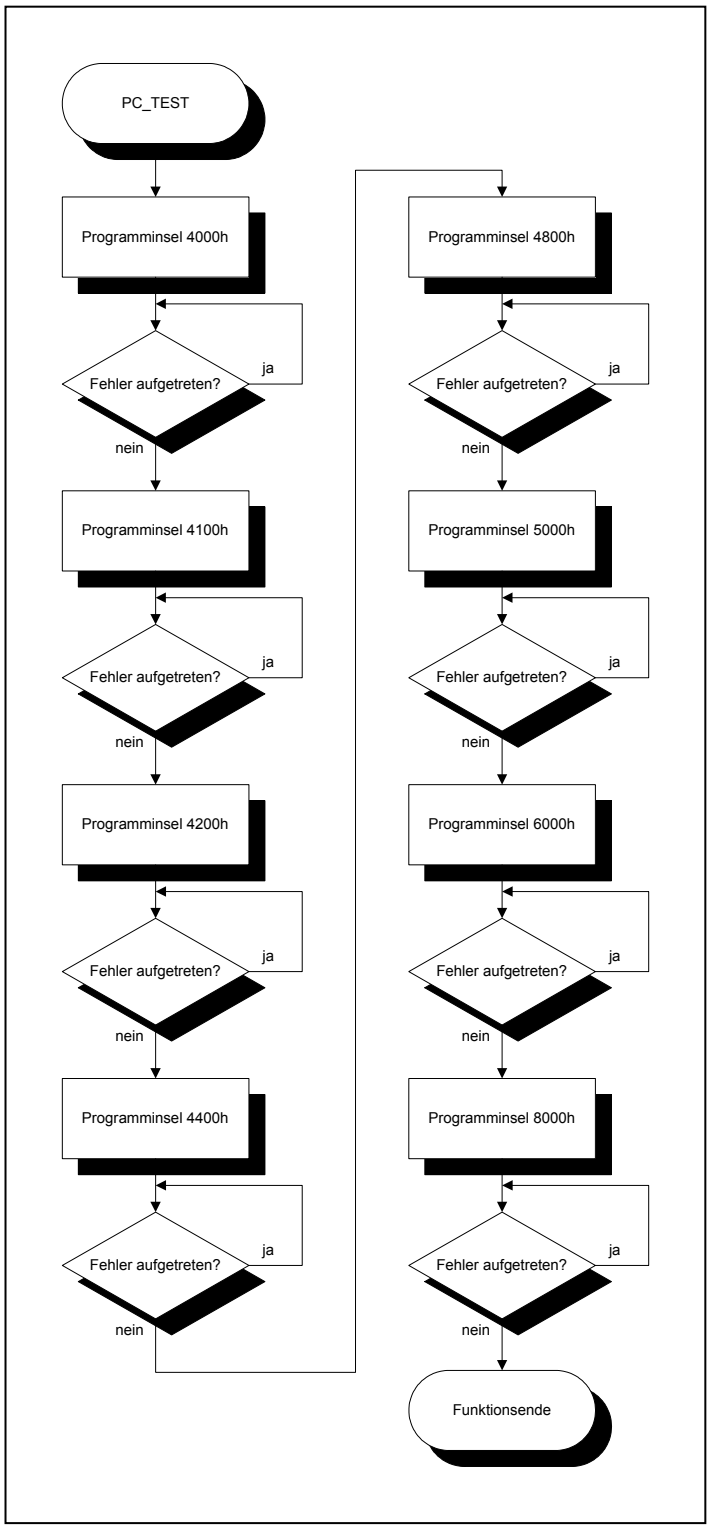
3.1 Grundlegende Tests

3.1.1 Programmzählertest (PC_TEST.ASM)

Der Programmzähler ist die Komponente einer CPU, die bestimmt, welche Stelle des abzuarbeitenden Programms gerade ausgeführt wird. Bei einer Fehlfunktion dieses Zeigers, wie beispielsweise einem Stuck-at-Fehler bestimmter Bits oder einem Übersprechen von Bits, ist der ordnungsgemäße Ablauf des Programms nicht mehr gewährleistet. Befehle werden übersprungen und die Abarbeitung erfolgt an einer völlig anderen Stelle als dies vom Programmierer vorgesehen war. Damit ist die ordnungsgemäße Funktion der CPU nicht mehr gewährleistet. Um solche Fehler im Programmzähler feststellen zu können und den Rechner in einen sicheren Zustand zu überführen, ist es notwendig, den Programmzähler zu testen. In diesem Beispiel werden Programminseln, die im Programmspeicher in unterschiedlichen Adressbereichen angeordnet sind, angesprungen, um dort eine Rechenoperation mit Vergleich durchzuführen und bei einem festgestellten Fehler in einer Endlosschleife zu verweilen. Generell gilt, dass dieses Verweilen in einer Endlosschleife dazu führt, dass der externe WD nicht mehr retriggert werden kann und das System somit über diesen zweiten Abschaltweg in den sicheren Zustand gebracht wird.

Der grobe Ablauf des realisierten Programmzählertests ist aus Abbildung 2 (siehe Seite 16) ersichtlich.

Abbildung 2:
Ablauf Programmzählertest



3.1.2 Akkutest (ACC_TEST.ASM)

Der Akkumulator als das zentrale Register einer CPU ist bei nahezu allen Operationen beteiligt. Er dient dabei vielfach sowohl als Quellregister als auch als Zielregister für einen Großteil der ausgeführten Befehle. Daher hat die korrekte Funktion dieses Registers eine große Bedeutung für die einwandfreie Funktion der gesamten CPU. Die Funktion des Akkumulators wird mithilfe des Testprogramms überprüft, indem das Testprogramm eine „1“ durch den Akkumulator „schiebt“ und nach Ende des Schiebens festgestellt wird, ob das Schieben genau so viele Schiebeoperationen benötigt hat, wie beim Entwurf des Tests vorgesehen waren. Im Falle eines Fehlers wird ein Fehlerspeicher mit einer entsprechenden Nummer geladen und das Programm verweilt in einer Endlosschleife.

```
; Modul: ACC_TEST.ASM

; Version 1.00          BIA - Referat 5.2                   Datum: 15.09.2000

; Funktion:
; Test des Akkumulators mit einer wandernden 1. Der Hintergrund kann nicht
; getestet werden, da man den Akku als Rechenregister für eine
; Hintergrundanalyse benötigt, sodass er letztendlich immer seinen Inhalt
; verändert. Die häufige Benutzung des Akkumulators sorgt jedoch dafür, dass
; Fehler auch anderweitig erkannt werden.
; Sollte das Carry nach weniger als 7 Schritten gesetzt sein, ist der
; Schleifenzähler noch nicht auf 0, sodass ein Akkufehler wahrscheinlich ist.
; Man könnte die Routine auch so ändern, dass der Initialwert von R1 07 ist.
; Damit entfiele der zusätzliche DEC R1 Befehl. Hier wird jedoch bewusst das
; Carry nach 7 Durchläufen der Schleife mitgetestet, sodass man zum einen JC
; testet, zum anderen eine dauernde 00 im Akku aufdeckt.

; Eingabeparameter: keine
; Ausgabeparameter: keine
; Veränderte Register: A, R1, PSW
; Veränderte Speicherbereiche: keine

; Fehleraussprung: keiner

; Übergeordnete Module: CPU_TEST.ASM
; Untergeordnete Module: keine
; aufgerufen von: MAIN
; ruft auf: -
```

```
; Stacktiefe: 2 Bytes
; Verwendete Registerbank: aktuelle Bank
; ************************************************************************
ACC_TEST:       CLR     C
                MOV     R1,#08                  ; R1 sei Schleifenzähler
                MOV     A,#01                   ; Testbyte: 0000$0001
ACC_1:          RLC     A                       ; wandernde 1 durch Carry-Flag
                JC      ACC_2                   ; Wenn 1 im Carry angekommen Springe zu ACC_2
                DJNZ    R1, ACC_1               ; Dekrementiere Schleifenzähler
ACC_2:          DEC     R1                      ; Dekrementiere Schleifenzähler nochmals
                CJNE    R1, #00h, ACC_ERR       ; hier geschieht der eigentliche Akkutest:
                SJMP    ACC_ENDE                ; Ende der Routine
ACC_ERR:        MOV     FEHLER,#FEHLER_ACC      ; Fehlerspeicher laden
                SJMP    ACC_ERR                 ; Endlosschleife
ACC_ENDE:       RET
```

3.1.3 Stackbefehlstest PUSH-, POP- und RET-Befehl (PPR_TEST.ASM)

Die PUSH- und POP-Befehle dienen dazu, Daten auf dem Stack kurzzeitig abzulegen. Oftmals werden diese in Funktionen verwendet, um Registerinhalte nach Funktionseinsprung zu „retten“ und den alten Registerinhalt vor Funktionsaussprung wiederherzustellen. Bei diesem Test wird überprüft, ob der Stackpointer korrekt inkrementiert und dekrementiert wird. Des Weiteren wird ausgewertet, ob der auf dem Stack gespeicherte Wert wieder korrekt vom Stack abgeholt wird.

Der RET-Befehl dient dazu, Unterprogramme wieder zu verlassen, wobei die Rücksprungadresse vorher automatisch von der CPU auf dem Stack abgelegt wurde.
Um den RET-Befehl zu testen, wird eine definierte Rücksprungadresse auf dem Stack abgelegt und anschließend der RET-Befehl ausgeführt. Daraufhin muss die CPU zur vorher bestimmten Adresse springen und dort die Programmausführung fortführen.

```
; Modul: PPR_TEST.ASM

; Version 1.00          BIA - Referat 5.2                   Datum: 15.09.2000

; Funktion:
; Getestet wird die Funktion des PUSH- und POP-Befehls sowie des RET-Befehls.

; Eingabeparameter: keine
; Ausgabeparameter: keine
; Veränderte Register: A, R0, R1, PSW
; Veränderte Speicherbereiche: keine

; Fehleraussprung: nach Funktion ERROR

; Übergeordnete Module: CPU_TEST.ASM
; Untergeordnete Module: keine
; aufgerufen von: MAIN
; ruft auf: -

; Stacktiefe: 4 Bytes
; Verwendete Registerbank: aktuelle Bank
; ************************************************************************
PPR_TEST:
PPR_PUSHPOP:    MOV     R0,SP                   ; Stackpointerposition speichern
                MOV     R1,SP
                INC     R1                      ; Adresse des zu pushenden Bytes herstellen
                MOV     A,#0AAh
                PUSH    ACC
                MOV     A,R1                    ; Erwartungshaltung und Vergleich
                CJNE    A,SP,PPR_err1

                CLR     A
                MOV     A,@R1
                CJNE    A,#0AAH,PPR_err1        ; Erwartungshaltung und Vergleich

                CLR     A
                POP     ACC
                CJNE    A,#0AAH,PPR_err1        ; Erwartungshaltung und Vergleich

                MOV     A,R0                    ; Erwartungshaltung und Vergleich
                CJNE    A,SP,PPR_err1


PPR_RET:        MOV     DPTR,#PPR_RET1          ; Rücksprungadresse in 16-bit-Register
                                                  speichern
                PUSH    DPL                     ; Rücksprungadresse auf Stack
                PUSH    DPH
                RET                             ; Rücksprung (zu PPR_RET1)
```

```
                NOP
                NOP
                NOP
                NOP
                NOP
                SJMP    PPR_err1
PPR_RET1:       SJMP    PPR_ENDE
                NOP
                NOP
                NOP
                NOP
                NOP
PPR_err1:       MOV     FEHLER,#FEHLER_PPR      ; Fehlerspeicher laden
                LJMP    ERROR
PPR_ENDE:       RET
```

3.2 Weitergehende Befehlstests

Um die einwandfreie Funktion der einzelnen Maschinenbefehle zu testen, wurde im Rahmen der Befehlstests jeder Befehl in allen möglichen Adressierungsarten verwendet und das Ergebnis mit einer Erwartungshaltung verglichen.

3.2.1 Jump if not Zero (JNZ_TEST.ASM)

Bei dieser bedingten Verzweigung wird ein Sprung davon abhängig gemacht, ob der Akkumulator den Wert Null enthält oder ob der Akkuinhalt ungleich Null ist. Um den Test durchzuführen, wird ein Wert in den Akkumulator und ein gleicher Wert in den Hilfsakkumulator geschrieben. Die beiden Werte werden voneinander subtrahiert und das Rechenergebnis im Akku erwartet. Im Anschluss an diese Subtraktion wird mit dem JNZ-Befehl geprüft, ob das Ergebnis korrekt ist. Im Fehlerfall wird ein Fehlerspeicher beschrieben und das Programm verweilt in einer Endlosschleife. Die Überprüfung wird mit zwei verschiedenen Werten durchgeführt. Sollte bei einer Sprungausführung zwischen einen Befehl eingesprungen werden, sollen die NOP-Befehle den STACK bereinigen und ein „Fangen“ des Prozessors bei der nächsten Schleife ermöglichen.

```
; Modul: JNZ_TEST.ASM

; Version 1.00          BIA - Referat 5.2                   Datum: 15.09.2000

; Funktion:
; Test von JNZ für die komplementären Akkumulatorinhalte AAh
; und 55h bei 5 Rücksprungschritten.

; Eingabeparameter: keine
; Ausgabeparameter: keine
; Veränderte Register: A, B, PSW
; Veränderte Speicherbereiche: keine

; Fehleraussprung: keiner

; Übergeordnete Module: CPU_TEST.ASM
; Untergeordnete Module: keine
; aufgerufen von: MAIN
; ruft auf: -

; Stacktiefe: 2 Bytes
; Verwendete Registerbank: keine
; ***************************************************************************
JNZ_TEST:       CLR     C
                MOV     A,#0AAh                 ; Lade Akku mit 1. Testbyte
                MOV     B,#0AAh                 ; Lade Register B mit 1. Testbyte
                SUBB    A,B                     ; Vergleich von A und B durch Subtraktion
Error1:         NOP
                NOP
                NOP
                NOP
                NOP
                MOV     FEHLER,#FEHLER_JNZ      ; Fehlerspeicher laden
                JNZ     Error1                  ; Sollte ein Fehler bei den Registern oder der
                                                ; Subtraktion
                                                ; auftreten, so zwingt JNZ zur Endlosschleife,
                                                ; die relative
                                                ; Adresse ist hier immer 05.
                MOV     FEHLER,#0               ; Fehlerspeicher löschen
                JZ      Error2                  ; Falls Ergebnis OK, Sprung zum CJNE-Test
                MOV     FEHLER,#FEHLER_JNZ      ; Fehlerspeicher laden
                NOP
                NOP
                NOP
                NOP
                NOP
Error8:         SJMP    Error8                  ; Beide Sprünge haben versagt > Endlosschleife
```

```
Error2:         NOP
                NOP
                NOP
                NOP
                NOP
                MOV     FEHLER,#FEHLER_JNZ      ; Fehlerspeicher laden
                CJNE    A,#00h,Error2           ; Sollte JNZ ignoriert werden, fängt sich hier
                                                ; der Controller in einer Endlosschleife
                MOV     FEHLER,#0               ; Fehlerspeicher löschen

; Gleiche Vorgehensweise für den Wert 55h
                MOV     A,#055h                 ; Lade Akku 2. Testbyte
                MOV     B,#055h                 ; Lade Register B mit 2. Testbyte
                SUBB    A,B                     ; Vergleich von A und B durch Subtraktion
Error3:         NOP
                NOP
                NOP
                NOP
                NOP
                MOV     FEHLER,#FEHLER_JNZ      ; Fehlerspeicher laden
                JNZ     Error1                  ; Sollte ein Fehler bei den Registern oder der
                                                ; Subtraktion
                                                ; auftreten, so zwingt JNZ zur Endlosschleife,
                                                ; die relative Adresse ist hier immer 05.
                MOV     FEHLER,#0               ; Fehlerspeicher löschen
                JZ      Error4                  ; Falls Ergebnis OK, Sprung zum CJNE-Test
                MOV     FEHLER,#FEHLER_JNZ      ; Fehlerspeicher laden
                NOP
                NOP
                NOP
                NOP
                NOP

Error9:         SJMP    Error9                  ; Beide Sprünge haben versagt > Endlosschleife
Error4:         NOP
                NOP
                NOP
                NOP
                NOP
                MOV     FEHLER,#FEHLER_JNZ      ; Fehlerspeicher laden
                CJNE    A,#00h,Error4           ; Sollte JNZ ignoriert werden, fängt sich hier
                                                ; der Controller in einer Endlosschleife
                MOV     FEHLER,#0               ; Fehlerspeicher löschen
                RET
```

3.2.2 Arithmetikbefehle (ARI_TEST.ASM)

Um die verschiedenen Arithmetikbefehle der CPU testen zu können, werden ausgewählte Werte in die entsprechenden Register geschrieben. Die Arithmetikoperation wird ausgeführt, das Ergebnis mit der Erwartungshaltung verglichen und ggf. wird mit einer gespeicherten Fehlernummer in eine Fehlerroutine verzweigt. Die Tests berücksichtigen auch verschiedene Adressierungsarten der Arithmetikbefehle, wie direkte Adressierung, indirekte Adressierung oder eine Operation mit einem konkreten Wert. Exemplarisch ist hier nur einer der Tests aufgeführt.

```
; Modul: ARI_TEST.ASM

; Version 1.00          BIA - Referat 5.2                   Datum: 15.09.2000

; Funktion:
; Test von Arithmetikbefehlen: ADD, ADDC, SUBB, INC, DEC, MUL, DIV, DA

; Eingabeparameter: keine
; Ausgabeparameter: keine
; Veränderte Register: siehe Einzelfunktionen
; Veränderte Speicherbereiche: keine, außer speziell für diese Tests reservierte
; Speicherzellen

; Fehleraussprung: nach Funktion ERROR

; Übergeordnete Module: LOGITEST.ASM
; Untergeordnete Module: keine

; Verwendete Registerbank: aktuelle Bank
; ***************************************************************************



; ******************** ADD_TEST 1 ***************************************
; Funktion: Test von ADD A,R0

; aufgerufen von: ADD_TEST
; ruft auf: -

; Veränderte Register: A, R0, PSW
; Stacktiefe: 2 Bytes
; ***************************************************************************
ADD_TEST1:      MOV     A,#0AAh                 ; Testmuster 1 laden
                MOV     R0,#55h                 ; Testmuster 2 laden
                ADD     A,R0                    ; Befehlstest
```

```
                CJNE    A,#0FFh,ADD_err1        ; Vergleich mit Erwartungshaltung
                RET
ADD_err1:       LJMP    ERROR_ARI
```

Übrige Tests: siehe Quelltext ARI_TEST.ASM

3.2.3 Logikbefehle (ANL_TEST.ASM, ORL_TEST.ASM, XRL_TEST.ASM und CRS_TEST.ASM)

Der Test der Logikbefehle verläuft ähnlich wie der Test der Arithmetikbefehle. Auch hier werden ausgewählte Werte in die entsprechenden Register geschrieben, die Operationen werden ausgeführt und das Ergebnis wird mit der Erwartungshaltung verglichen. Es werden verschiedene Adressierungsarten berücksichtigt. Exemplarisch ist auch hier nur einer der Tests aufgeführt.

```
; Modul: ANL_TEST.ASM

; Version 1.00          BIA - Referat 5.2                   Datum: 15.09.2000

; Funktion:
; Die Unterprogramme testen den Verknüpfungsbefehl ANL, indem AAh
; (10101010B) und 56h (01010110B) bitweise logisch UND verknüpft
; werden und das berechnete Ergebnis mit dem erwarteten Ergebnis
; von 02h (00000010B) verglichen wird.

; Eingabeparameter: keine
; Ausgabeparameter: keine
; Veränderte Register: siehe Einzelfunktionen
; Veränderte Speicherbereiche: keine, außer speziell für diese Tests reservierte
; Speicherzellen

; Fehleraussprung: nach Funktion ERROR

; Übergeordnete Module: LOGITEST.ASM
; Untergeordnete Module: keine
; aufgerufen von: ANL_TEST
; ruft auf: -

; Stacktiefe: 2 Bytes
; Verwendete Registerbank: aktuelle Bank
; ***************************************************************************
```

```
; ******************** ANL_TEST 1 ***************************************
; Funktion: Test von ANL A,R0
; Veränderte Register: A, R0, PSW
; ***********************************************************************
ANL_TEST1:      MOV     A,#0AAh                 ; Testmuster 1 laden
                MOV     R0,#56h                 ; Testmuster 2 laden
                ANL     A,R0                    ; Befehlstest
                CJNE    A,#02h,ANL_err1         ; Vergleich mit Erwartungshaltung
                RET
ANL_err1:       LJMP    ERROR_ANL
```

Übrige Tests: siehe Quelltexte ANL_TEST.ASM, ORL_TEST.ASM, XRL_TEST.ASM, CRS_TEST.ASM

3.2.4 Logikbefehle (BIT_TEST.ASM)

Hierbei werden der bitadressierbare Speicherbereich sowie die Funktion der Bitbefehle getestet.

Der Speicherbereich wird überprüft, indem Bits gezielt verändert werden und danach der übrige bitadressierbare Speicherbereich auf Veränderungen untersucht wird.
Die bitorientierten Logikbefehle werden in einer separaten Funktion getestet, indem die verschiedenen Bitbefehle angewandt und das Ergebnis jeweils mit der Erwartungshaltung verglichen wird.

```
; Modul: BIT_TEST.ASM

; Version 1.00          BIA - Referat 5.2                   Datum: 15.09.2000

; Funktion:
; Die Unterprogramme testen die BIT-Adressierung im bitadressierbaren
; Speicherbereich von 20H bis 2FH (internes RAM). Daraus ergeben
; sich 128 Bitadressen. Aus Gründen des Zeitaufwandes werden
; acht gezielt ausgesuchte Adressen für die Tests ausgewählt
; (wandernde 1 vor 0-Hintergrund in der Bitadresse).
; Vorgehensweise:
; Zunächst wird der Speicherbereich gelöscht.
; Dann wird jeweils das ausgewählte Bit gesetzt und überprüft.
; Der restliche Speicher wird jeweils auf korrekten Inhalt überprüft.
```

```
Danach wird das jeweils gesetzte Bit wieder gelöscht.
; (Die Bitbefehle SETB bit und CLR bit werden bei diesen Operationen gleich-
; zeitig getestet).
; In einem weiteren Test werden die Befehle für logische Bitoperationen getestet.

; Eingabeparameter: keine
; Ausgabeparameter: keine
; Veränderte Register: siehe Einzelfunktionen
; Veränderte Speicherbereiche: siehe Einzelfunktionen

; Fehleraussprung: nach Funktion ERROR

; Übergeordnete Module: LOGITEST.ASM
; Untergeordnete Module: keine

; Verwendete Registerbank: aktuelle Bank
; ***************************************************************************



; ********************* BIT_TEST1 ***********************************************
; Funktion: Testbit Adresse 01h

; aufgerufen von: BIT_TEST_A
; ruft auf: BIT_SPEICHER_RUECKL

; Veränderte Register: A, PSW
; Veränderte Speicherbereiche: Testbit (Bit 01h)

; Stacktiefe: 2 Bytes + aufgerufene Funktionen
; ***************************************************************************
BIT_TEST1:      MOV     BYTE_ADR,#20h           ; Merker für Byte-Adressenausschluß vorbelegen
                SETB    01h                     ; setzt Testbit
                MOV     A,INTRAM20
                CJNE    A,#02h,Bit_err          ; Vergleich mit Erwartungshaltung
                LCALL   BIT_SPEICHER_RUECKL     ; übrigen bitadressierbaren Speicher prüfen
                CLR     01h                     ; löscht Testbit
                RET
```

Übrige Bittests und Hilfsfunktionen: siehe Quelltext BIT_TEST.ASM

```
; ***************************************************************************
Bit_err:        MOV     FEHLER,#FEHLER_BIT      ; Fehlerspeicher laden
                LJMP    ERROR
; ***************************************************************************
```

```
; ********************* BIT_SPEICHER_RUECKL *********************************
; Funktion: Überprüft den bitadressierbaren Speicherbereich (20h-2Fh) mit
;           Ausnahme des gerade beschriebenen Bytes auf korrekten Inhalt.

; aufgerufen von: BIT_TEST1 - BIT_TEST_8
; ruft auf: -

; Veränderte Register: A, R0, PSW
; Veränderte Speicherbereiche: keine

; Stacktiefe: 2 Bytes
; ***************************************************************************
BIT_SPEICHER_RUECKL:
                MOV     R0,#20h                 ; Anfang des bitadressierbaren Bereichs
comp2:          MOV     A,R0
                CJNE    A,BYTE_ADR,comp3        ; Position des Datenbytes, welches =! 0 ist ?
                INC     R0                      ; falls ja, Position inkrementieren
comp3:          CJNE    R0,#30h,comp4           ; Ende des bitadressierbaren Bereichs?
                SJMP    comp5
comp4:          CJNE    @R0,#00h,BIT_err        ; aktuelles Byte==0 ?
                INC     R0                      ; Position inkrementieren
                SJMP    comp2
comp5:          RET


; ********************* BITBEFEHLE ************************************
; Funktion: Überprüft die bitorientierten Befehle auf korrekte Funktion

; aufgerufen von: BIT_TEST_D
; ruft auf: -

; Veränderte Register: PSW
; Veränderte Speicherbereiche: Bitadresse 01h

; Stacktiefe: 2 Bytes
; ***************************************************************************
BITBEFEHLE:     SETB    01h
                MOV     C,01h                   ; Transfer
                JNC     BIT_err                 ; Ergebnis OK?
                CPL     C
                ANL     C,01h
                JC      BIT_err                 ; Ergebnis OK?
                ORL     C,01h
                JNC     BIT_err                 ; Ergebnis OK?
                CLR     C
                JC      BIT_err                 ; Ergebnis OK?
```

```
                SETB    C
                CLR     01h
                MOV     01h,C                   ; Transfer
                JNB     01h,BIT_err             ; Ergebnis OK?
                ANL     C,/01h
                JC      BIT_err                 ; Ergebnis OK?
                ORL     C,/01h
                JC      BIT_err                 ; Ergebnis OK?
                CLR     01h
                JB      01h,BIT_err             ; Ergebnis OK?
                CPL     01h
                JNB     01h,BIT_err             ; Ergebnis OK?
                RET
```

3.2.5 Transferbefehle (TRANTEST.ASM)

Für die Überprüfung der Transferbefehle werden die einzelnen Befehle mit den verschiedenen Adressierungsarten ausgeführt und das Ergebnis wird mit der Erwartungshaltung verglichen.

```
; Modul: TRANTEST.ASM

; Version 1.00          BIA - Referat 5.2                   Datum: 15.09.2000

; Funktion:
; Test von Transferbefehlen: MOV, XCH, MOVX, MOVC

; Eingabeparameter: keine
; Ausgabeparameter: keine
; Veränderte Register: siehe Einzelfunktionen
; Veränderte Speicherbereiche: keine, außer speziell für diese Tests reservierte
; Speicherzellen

; Fehleraussprung: nach Funktion ERROR

; Übergeordnete Module: CPU_TEST.ASM
; Untergeordnete Module: keine

; Stacktiefe: siehe Einzelfunktionen
; Verwendete Registerbank: aktuelle Bank
```

```
; Hinweis:
; Vom Anwenderprogramm aus sollten die Sammelfunktionen aufgerufen werden.
; ************************************************************************



; ******************** MOV_TEST1 ****************************************
; Funktion: Test von MOV A,R0

; aufgerufen von: MOV_TEST
; ruft auf: -

; Veränderte Register: A, R0, PSW

; Stacktiefe: 2 Bytes
; ************************************************************************
MOV_TEST1:      MOV     A,#55h                  ; Akku initialisieren
                MOV     R0,#0AAh                ; Testmuster 1 laden
                MOV     A,R0                    ; Befehlstest
                CJNE    A,#0AAh,TRAN_err1       ; Vergleich mit Erwartungshaltung
                RET
```

Übrige Tests: siehe Quelltext TRANTEST.ASM

4 Speichertests

4.1 Programmspeichertest (ROM_TEST.ASM)

Der Programmspeicher eines Prozessors ist die Komponente, die bestimmt, welche Funktion das System zu erfüllen hat. Ein bloßer Austausch der im Programmspeicher enthaltenen Software kann aus einem Steuerungsprozessor für einen Videorecorder einen Steuerungsprozessor für einen Schaufelradbagger machen. Aus diesem Grunde ist es äußerst wichtig, dass die Software über die gesamte Lebensdauer eines Systems keine ungewünschte Veränderung erfährt. Dies könnte passieren, indem auch nur ein einziges Bit des Programmspeichers seinen Wert ändert. Durch diesen Vorgang kann sich die Bedeutung eines Befehls so verändern, dass der Prozessor dann an einer bestimmten Stelle des Programms eine völlig andere Anweisung ausführt. Dies führt im günstigsten Fall dazu, dass der Prozessor „abstürzt“, eine ungewollte Ausführung bestimmter vielleicht gefährlicher Funktionen ist aber nicht auszuschließen.

Um zu verhindern, dass eine derartige Veränderung einer Speicherzelle unbemerkt bleibt, wird der Inhalt des Programmspeichers vom Prozessor ebenfalls ständig überprüft. Dies kann erfolgen, indem über die gespeicherten Werte eine Signatur bzw. Prüfsumme (hier CRC 16) [4] gebildet wird, die dann mit einer bei der erstmaligen Programmierung des Programmspeichers erstellten und zusätzlich abgespeicherten Signatur verglichen wird. Änderungen der Signatur werden bemerkt und der Prozessor kann Maßnahmen ergreifen, indem er eine vorher definierte Fehlerreaktion ausführt, in einer Endlosschleife verbleibt und damit das System in den sicheren Zustand versetzt.

Bei dem hier realisierten Test (Abbildungen 3 und 4, siehe Seite 32 und 33) wird der Programmspeicher in m-Segmente aufgeteilt. Jedes dieser Segmente wird unterbrechungsfrei in die Signaturbildung eingebunden; nach Abarbeitung eines Segments steht der Prozessor für andere Aufgaben zur Verfügung. Hierbei erreicht man, dass der Test in handhabbare kleine Zeitschreiben aufgeteilt wird und der Prozessor die für die Steuerungsaufgabe notwendige Reaktionszeit einhalten kann.

Abbildung 3:
Segmentierung des ROM-Tests

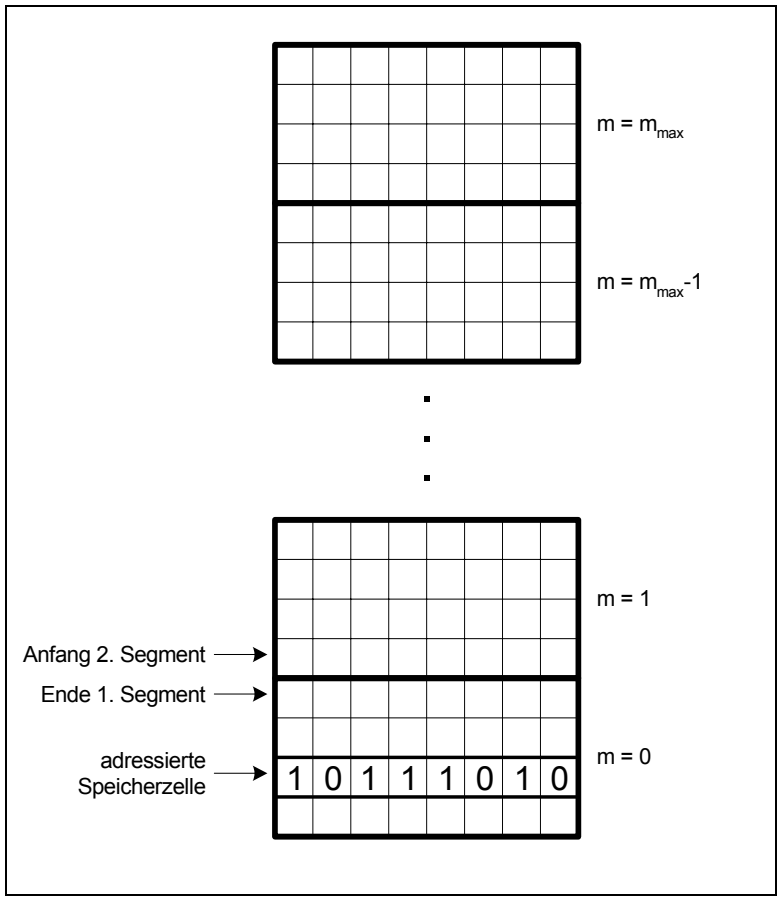
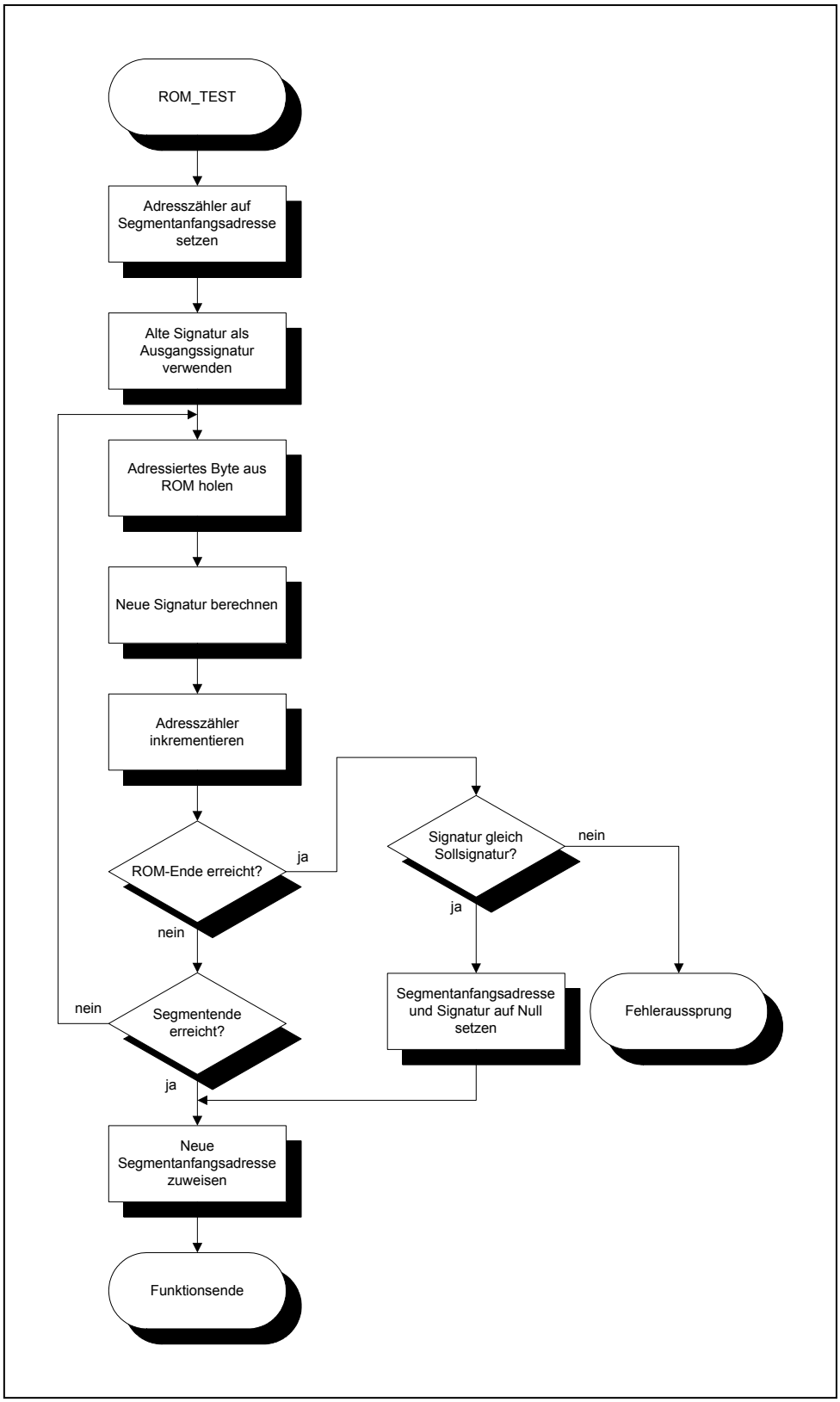


Abbildung 4:
Ablauf des ROM-Tests

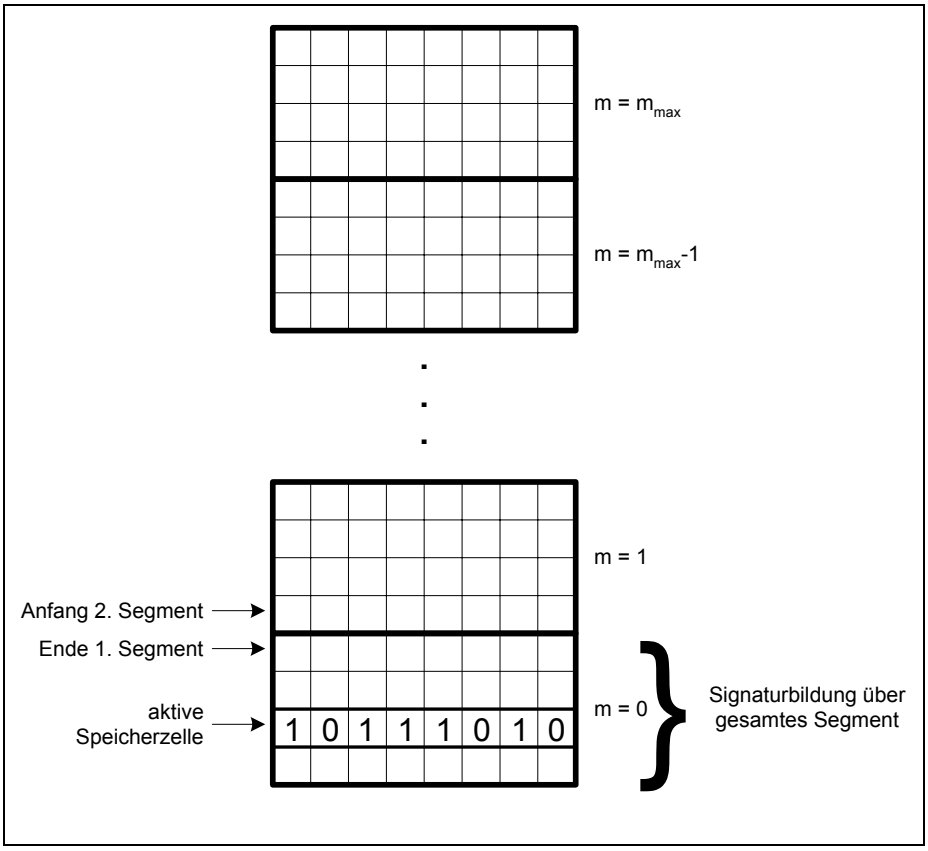


4.2 Datenspeichertest (XRAMTEST.ASM)

Für den Datenspeicher gelten prinzipiell die gleichen Aussagen wie für den Programmspeicher. Eine ungewollte Veränderung einzelner Datenbits kann ebenfalls zu ungewollten und ggf. auch gefährlichen Reaktionen des Prozessors führen. Deshalb muss auch dieser Datenspeicher auf einwandfreie Funktion getestet werden, sodass z. B. sowohl ein „Festklemmen“ von Bits auf einem bestimmten Wert als auch der Kurzschluss beliebiger Bits zu beliebigen anderen Bits aufgedeckt wird und eine entsprechende Fehlerreaktion eingeleitet werden kann.

Wie auch beim ROM-Test, wird eine Segmentierung (Abbildung 5) vorgenommen, sodass der Test in Zeitscheiben, die klein genug sind, um die für die Steuerungsaufgabe notwendige Reaktionszeit des Prozessors sicherzustellen, aufgeteilt werden kann.

Abbildung 5:
Segmentierung des RAM-Tests



Zu unterscheiden ist hier noch zwischen der Funktion XRAMTEST1, die eine einzelne Speicherzelle auf interne Fehler überprüft, und der Funktion XRAMTEST2, die Kurzschlüsse zwischen beliebigen Speicherzellen aufdeckt (Abbildungen 6 und 7, siehe Seite 36).

Abbildung 6:
Ablauf des RAM-Tests 1

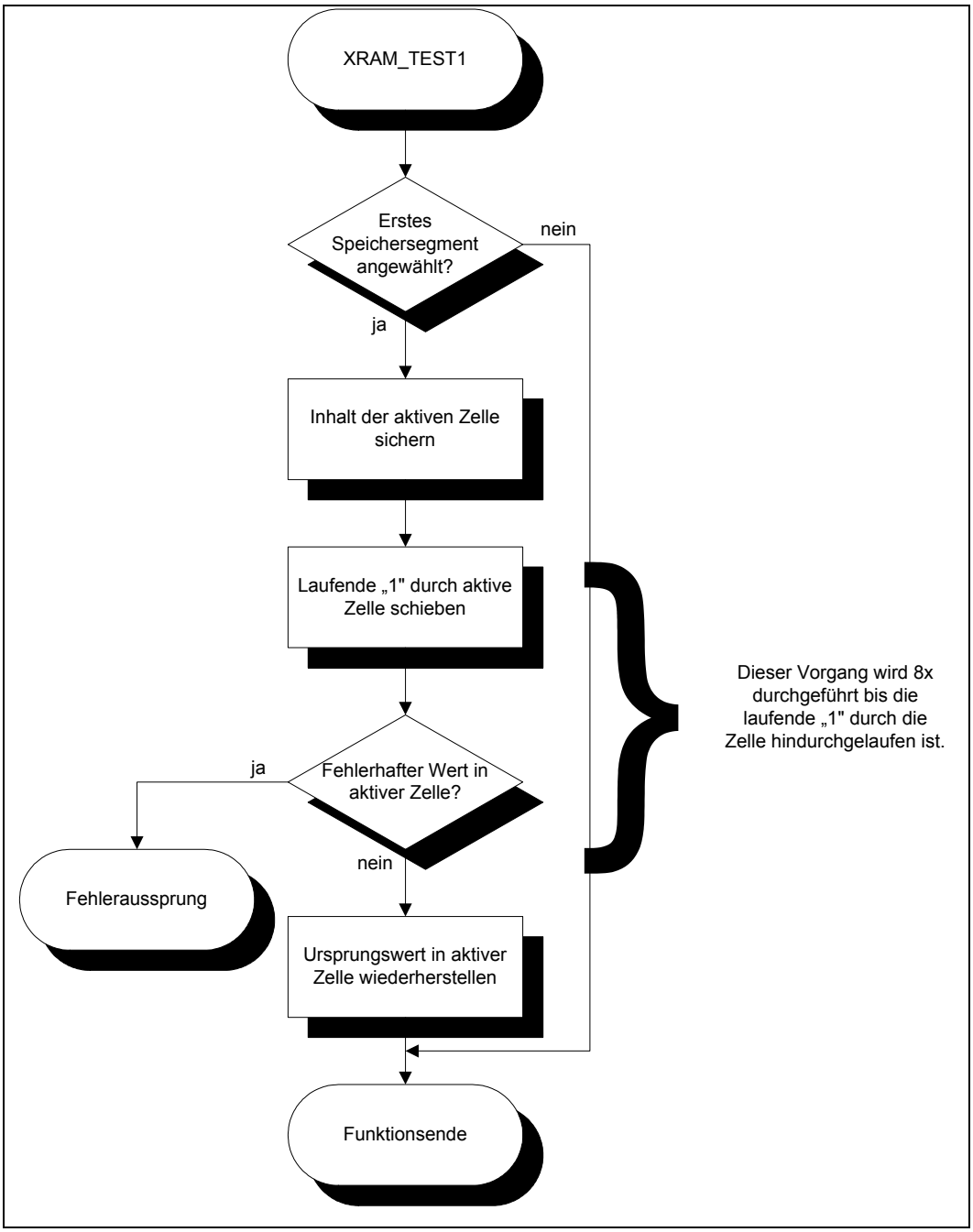
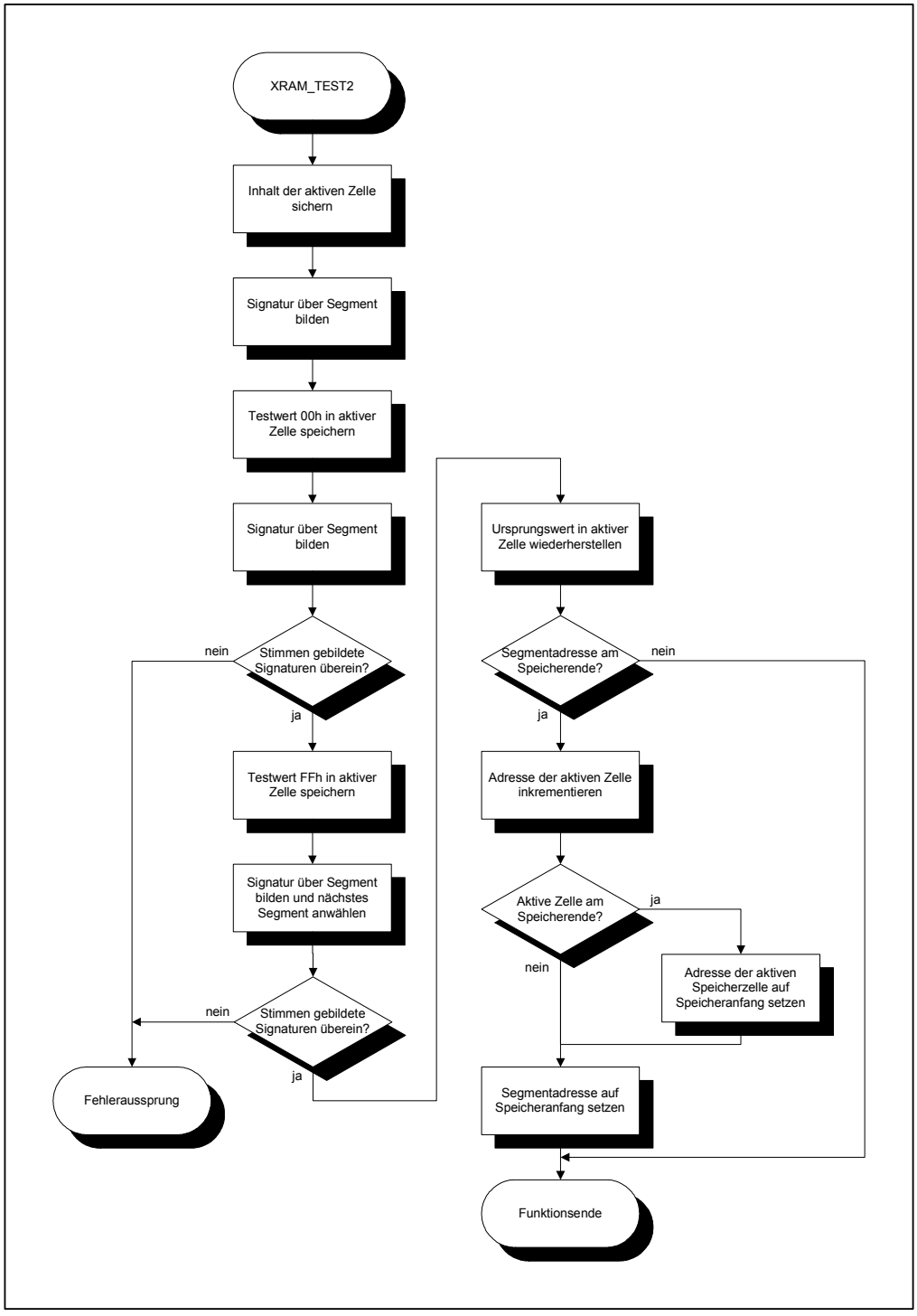


Abbildung 7:
Ablauf des RAM-Tests 2



Zur Aufdeckung von Kurzschlüssen zwischen beliebigen Speicherzellen wird jedes Byte des gesamten zu überprüfenden Speichers gegen den restlichen Speicher überprüft.

Dazu wird das zu überprüfende Byte mit verschiedenen Werten beschrieben und daraufhin der übrige Speicher auf Veränderungen überprüft. Diese Überprüfung erfolgt durch eine Signaturbildung, sodass bei Unterschieden im Wert der Signatur vor und nach Beschreiben des jeweils zu prüfenden Bytes von einem Kurzschluss zwischen Speicherzellen auszugehen ist. Um die für den RAM-Test notwendige Zeitscheibe klein zu halten, wird das RAM, wie oben bereits erwähnt, in m-Segmente aufgeteilt und nur über jeweils ein Segment eine Signatur gebildet. Hierbei ist sichergestellt, dass der Prozessor nicht für längere Zeit nur mit dem Test beschäftigt ist und in Ermangelung von Rechenzeit eine zeitgerechte Abarbeitung des Anwenderprogramms unmöglich wird.

Die Datei XRAMTEST.ASM enthält weitere Erläuterungen zum RAM-Test.

5 Special-Function-Register-Test (SFR_TEST.ASM)

Um die internen Special Function Register des Mikrocontrollers zu testen, werden diese, ebenso wie andere Speicherzellen auch, einzeln mit Werten beschrieben, nachdem der jeweilige Inhalt gesichert wurde. Weiter wird eine Signatur über die übrigen, gerade nicht zur Überprüfung anstehenden Zellen gebildet. Nach Beschreiben der gerade geprüften Zelle wird wiederum eine Signatur über die restlichen Speicherzellen gebildet, um so feststellen zu können, ob ein Übersprechen zu anderen Zellen stattgefunden hat.

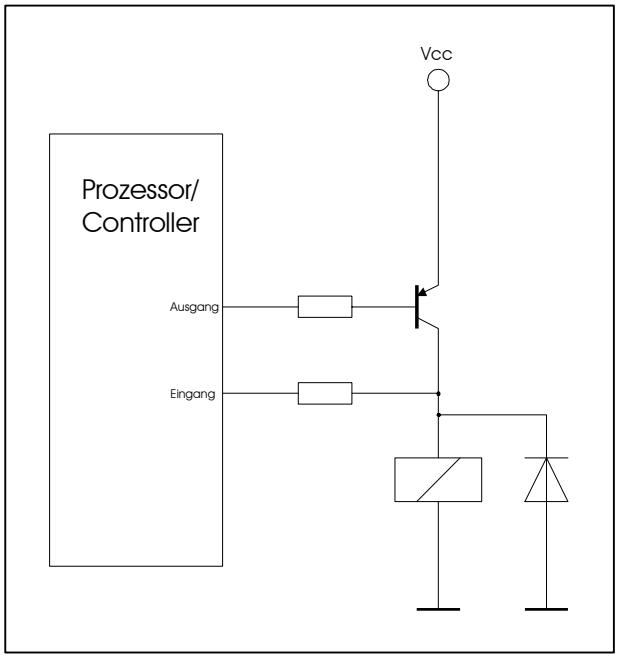
Grundsätzlich ist zu beachten, dass das Beschreiben einiger Special Function Register Funktionen bestimmter Peripheriekomponenten des Controllers auslöst (z. B. Beschreiben des Senderegisters der seriellen Schnittstelle). Als Konsequenz daraus sind einige dieser Register nicht ohne weiteres testbar, sodass ggf. auf andere Testmethoden zurückgegriffen werden muss.

Als Besonderheit des hier im Beispiel realisierten Tests ist zu nennen, dass die Inhalte der zu überprüfenden Special Function Register zuerst in einen Bereich des externen Datenspeichers kopiert werden, um dort die Bildung der Signatur einfacher durchzuführen, da die Special Function Register sich nicht indirekt adressieren lassen.

6 Porttests (IO_TEST.ASM)

In Abbildung 8 ist schematisch dargestellt, wie eine Portanschaltung ausgeführt sein kann, um dem Prozessor die Möglichkeit der Rücklesung des aktuellen Zustands der Ausgangsperipherie zu geben. Eine Überprüfung der Funktionsfähigkeit des Transistors kann jederzeit bei durchgesteuerter Kollektor-Emitter-Strecke erfolgen, indem das Ansteuersignal vom Prozessor für wenige hundert Mikrosekunden bis zu einigen Millisekunden (abhängig von der Filterwirkung der externen Beschaltung) weggenommen und das ordnungsgemäße Abschalten über den Porteingang zurückgelesen wird. So wird überprüft, ob der Prozessor das Relais noch abschalten könnte, wenn die Notwendigkeit dazu besteht. Die Dauer der Rücknahme des Ansteuersignals muss so gewählt werden, dass das Relais nicht abfallen kann und die Maschinensteuerung durch die Testmaßnahme beeinflusst würde.

Abbildung 8:
Portanschaltung mit Rücklesung



Um eine teilweise Unabhängigkeit des für die Relaisansteuerung verwendeten Portausgangs vom rücklesenden Porteingang zu erreichen, sollten verschiedene Ports verwendet werden (z. B. Port 1.5 für das Ausgangsbit und Port 3.5 als Eingangsbit).

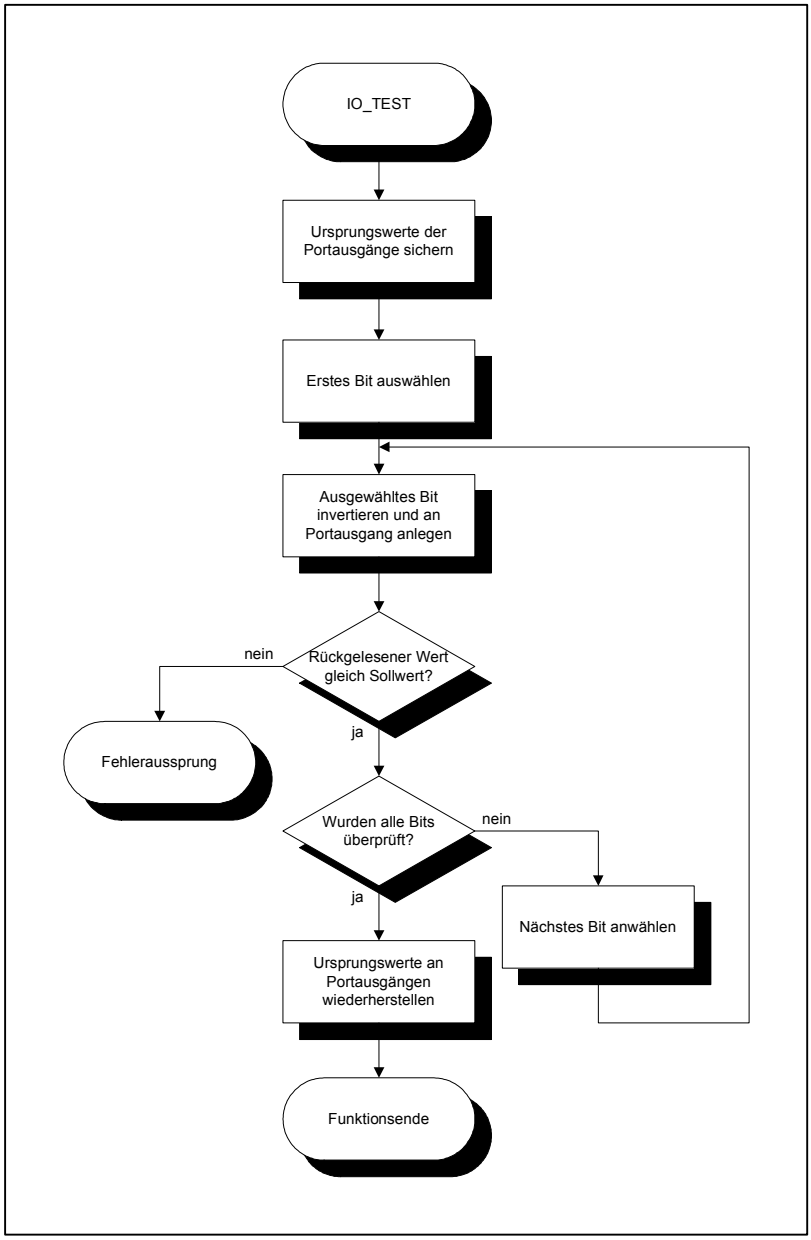
Neben dem hier beschriebenen Test ist es auch erforderlich, eine Fehleraufdeckung in Bezug auf den Aktor selbst, beispielsweise das in Abbildung 8 gezeigte Relais, durchzuführen. Hierbei wird üblicherweise der Schaltzustand eines Öffnerkontaktes des zwangsgeführten Ausgangsrelais über einen Eingangsport rückgelesen. Bei diesem Test ist es unumgänglich, dass eine Anforderung an das Schalten des Relais durch den Prozess erfolgt. Würde das Relais aufgrund interner Fehler nicht abfallen können, wird dies durch die Rückmeldung des zwangsgeführten Kontaktes bemerkt und der zweite Abschaltweg über den Watchdog muss den sicheren Zustand einleiten. Ein Wiedereinschalten der Steuerung wird dann bei bleibendem Fehler sicher verhindert.

Um diese Diagnosemöglichkeiten nutzen zu können, müssen dazu sowohl bei der Hardware- als auch auf der Software die Voraussetzungen geschaffen werden. Die Maßnahmen zur Testung der Ausgangsports auf der Hardwareseite sind in Abbildung 8 dargestellt. Softwareseitig ist eine Funktion zu erstellen, die überprüft, ob der Ausgang des Controllers noch in der Lage ist zu schalten.

Die hier realisierte Funktion (Abbildung 9, siehe Seite 43) fragt den Status der Ausgänge ab und sichert diesen. Nun wird jedes Bit des Ports der Reihe nach invertiert[1], das erwartete Bitmuster eingelesen und mit dem ausgegebenen Bitmuster verglichen. So werden eventuell vorhandene statische Zustände von Aus- und Eingängen der jeweiligen Ports aufgedeckt.

[1] Anmerkung: Bei diesem Verfahren ist dringend sicherzustellen, dass eine auch kurzzeitige „1“ nicht zu einer Ansteuerung der externen Peripherie führen darf.

Abbildung 9:
Ablauf des Porttests



7 Hauptprogramm

In Abbildung 10 ist dargestellt, wie der grundsätzliche Aufbau des Hauptprogramms gestaltet werden kann, um sowohl die Selbsttests als auch das Anwenderprogramm einzubinden.

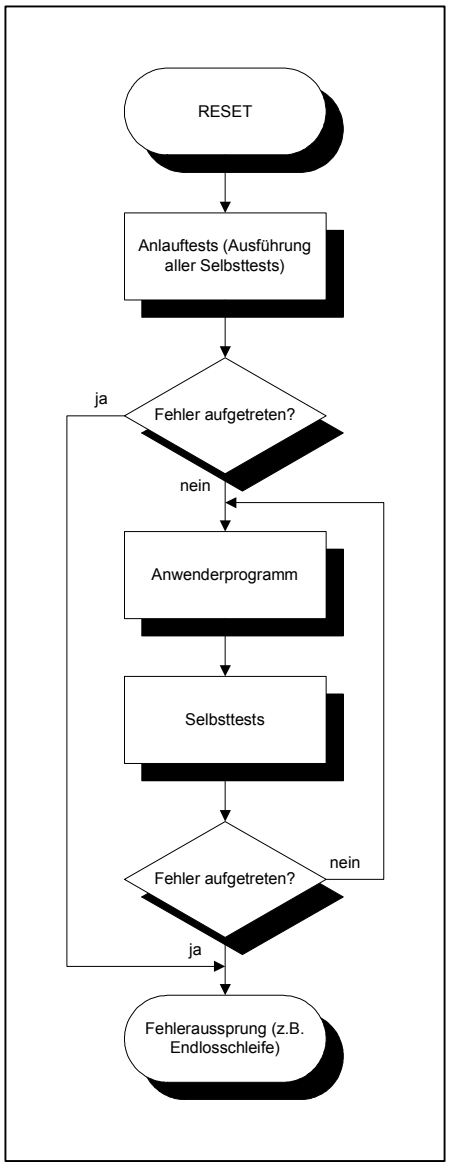


Abbildung 10:
Ablauf Hauptprogramm

Hierbei ist es wichtig, dass nach Einschalten des Rechnersystems und vor dem erstmaligen Abarbeiten des Anwenderprogramms alle Tests einmal komplett durchlaufen werden. Durch diesen Anlauftest wird sichergestellt, dass nur ein einwandfrei

arbeitendes Rechnersystem den Betrieb aufnimmt. Nach Ausführung aller Tests (CPU_TEST.ASM) kann die Bearbeitung der ständig ablaufenden Selbsttests in Zeitscheiben unterteilt erfolgen.

Das Hauptprogramm wird je nach Einsatzart des Mikrocontrollers völlig unterschiedlich programmiert sein. Hierbei ist sowohl eine ablaufgesteuerte als auch eine ereignisgesteuerte Programmierung denkbar; auf die Erklärung eines geeigneten Testmanagers muss deshalb an dieser Stelle verzichtet werden.

8 Schlussbemerkung

Dieser Report soll verdeutlichen, dass es nicht nur den einen „richtigen Weg“ geben kann. Die vorgestellten Beispiele sind als Anregung gedacht, um einen ersten Einstieg zu erleichtern. Für weitere Überlegungen hin zu einem sicheren System ist es unbedingt erforderlich zu erkennen, dass die Basis hierzu bereits in der sicherheitsgerichteten Struktur gelegt wird. Die in diesem Report vorgestellten Rechnertests sind als Ergänzung zu sehen, um auch Fehlerakkumulierungen zu verhindern. Die Reaktion auf festgestellte Fehler kann systembedingten Umständen angepasst werden. Sollten bei der Umsetzung Fragen zu klären sein, bietet das BGIA entsprechende Beratung an.

„Quo vadis Fehler?“, so lautet die im Untertitel gestellte Frage. Alle durch die Tests festgestellten Fehler laufen zum Watchdog, um dort indirekt durch die Endlosschleife im aktuellen Testprogramm den Watchdog eben nicht mehr mit Triggersignalen zu versorgen und somit den sicheren Zustand durch Auslösen des zweiten Abschaltweges herbeizuführen. Dies ist eine der möglichen Strukturen, die in einem Bündel weiterer Maßnamen geeignet ist, im Maschinenschutz die Auswirkung zufälliger Ausfälle zu beherrschen.

9 Literaturverzeichnis

[1] *Kleinbreuer, W.; Kreutzkampf, F.; Meffert, K.; Reinert, D.*: Kategorien für sicherheitsbezogene Steuerungen nach EN 954-1. BIA-Report 6/97. Hrsg.: Hauptverband der gewerblichen Berufsgenossenschaften (HVBG), Sankt Augustin 1997

[2] IEC 61508/VDE 0803: Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme. Beuth, Berlin 2002

[3] *Klug, J.; Schaefer, M.*: Fehlererkennende Maßnahmen in Mikroprozessoren. Erich Schmidt, Bielefeld 1997

[4] *Leisengang, D.*: Klassifikation und Einsatz von Signaturregistern zur Fehlererkennung in digitalen Schaltungen. Dissertation Technische Universität München, 1982

Weiterführende Literatur:

Halang, W. A.; Konakovsky, R.: Sicherheitsgerichtete Echtzeitsysteme. Oldenbourg, München 1999

Leisengang, D.: Signaturanalyse in der Datenverarbeitung. Elektronik 21 (1983), S. 67-72

Hölscher, H.; Rader, J.: Mikrocomputer in der Sicherheitstechnik. Verlag TÜV Rheinland, 1984

Schaefer, M.; Gnedina, A.; Bömer, T.; Büllesbach, K.-H.; Grigulewitsch, W.; Reuß, G.; Reinert, D.: Programmierregeln für die Erstellung von Software für Steuerungen mit Sicherheitsaufgaben. Verlag für neue Wissenschaft, Bremerhaven 1998

Siemens Microcomputer Components, SAB 80C517/80C537, 8-Bit CMOS Single-Chip Microcontroller. Siemens, München 1983