Testen von eingebetteten und sicherheitskritischen Systemen

Referent: Oliver Degenhardt

Ausarbeitung zum Seminarvortrag
vom 3. März 2005

Betreuer: Dr. rer. nat, Dipl.-Phys. Klaus Wüst
Dr. rer. nat, Dipl.-Inf. Peter Kneisel

Introduction

In this composition I deliver insight in the topic of testing software, especially the testing of embedded systems. The composition includes the design and executing of modul-, integration- and systemtests. In addition I present the analyse of scheduling of realtime systems, especially the method TDMA (*Time Division Multiple Access*) and round robin.

Einleitung

In dieser Arbeit gebe ich einen ersten Einblick in die Thematik des Testens von Software, speziell von eingebetteten Systemen.

Sie beinhaltet unter anderem die Planung und Durchführung von Modul-, Integrations- und Systemtests. Des Weiteren wird das Thema „Schedulinganalyse für Echtzeitsysteme“ angesprochen und speziell auf die Analyseverfahren TDMA (*Time Division Multiple Access*) und Round Robin eingegangen.

Begonnen wird mit der Begriffserläuterung von eingebetteten Systemen in Kapitel 1. Danach folgt eine kurze „Motivation“ in Kapitel 2.

Danken möchte ich an dieser Stelle meinen beiden Betreuern Dr. rer. nat, Dipl.-Phys. Klaus Wüst und Dr. rer. nat, Dipl.-Inf. Peter Kneisel, die diese Arbeit ermöglicht haben, indem sie das Seminar angeboten haben.

Ein weiterer Dank geht an Dr. Stephan Grünfelder, der mir zwei nicht veröffentlichte Artikel zu diesem Thema hat zukommen lassen.

1 Grundlagen

Dieses Kapitel dient dazu, eine kurze Einführung in die Thematik von eingebetteten Systemen und Echtzeit- Systemen zu geben. Es werden dazu die Begriffe und die Eigenschaften dieser Systeme beschrieben. Des Weiteren werden die Unterschiede von sicherheitskritischen und nicht- sicherheitskritischen Systemen erläutert.

1. 1 Definition eingebetteter Systeme

Es ist ein unglücklicher Umstand, dass der Begriff „eingebettete Systeme“ nicht sehr eindeutig verwendet und benutzt wird. Er beschreibt eine ganze Bandbreite von verschiedenen Produkten und Anwendungen, die sich zum Teil in ihrer Struktur und ihrem Aufbau sehr unterscheiden. Man findet heute eingebettete Systeme in einer Vielzahl von Produkten der Automobilindustrie, der Verkehrstechnik, der Produktions- und Fertigungstechnik sowie der Telekommunikationsindustrie.

Der wesentliche Unterschied eingebetteter Systeme zu klassischen Rechensystemen besteht darin, dass sie ein Teil eines großen Gesamtsystems sind. Es handelt sich daher um eingebettete Computersysteme, die „weitestgehend unsichtbar“ ihre Arbeit im Hintergrund ausführen. Folgende Abbildung zeigt einen allgemeinen Aufbau solcher Systeme:

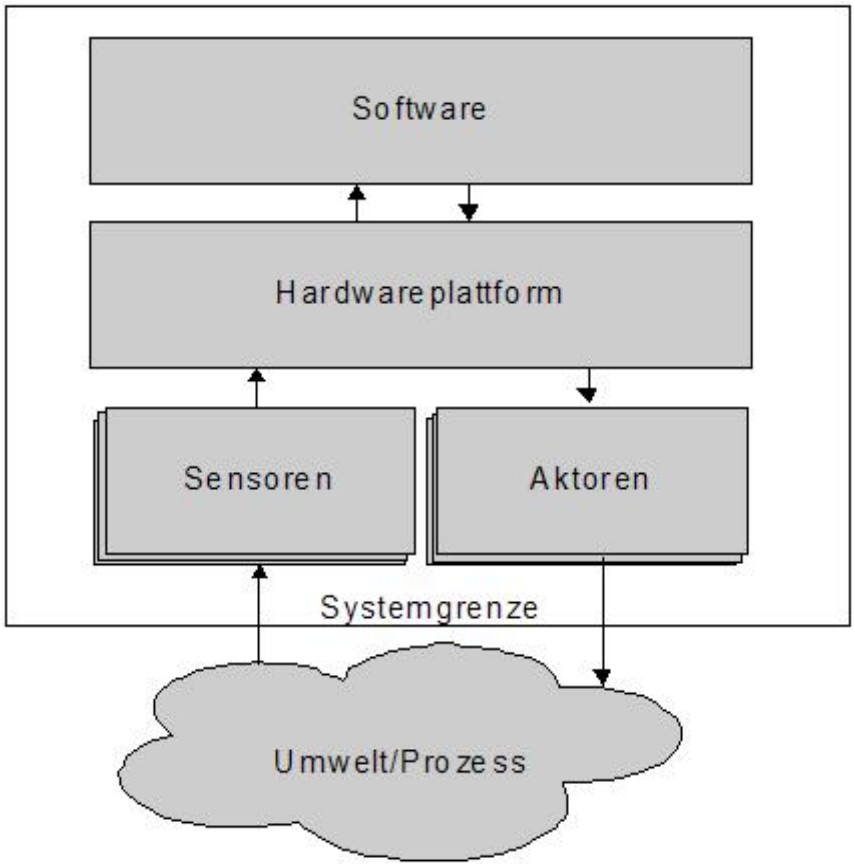

Abbildung 1.1.1: Aufbau eines eingebetteten Systems

Die große Vielfalt an Mikroprozessoren, derzeit in der Mehrzahl 4-, 8- und 16- Bit Prozessoren, die bei den einzelnen Anwendungsgebieten anzutreffen ist, ist ein weiteres Merkmal solcher Systeme.

Im Allgemeinen kann man eingebettete Systeme als Systeme beschreiben, die folgende Eigenschaften besitzen:

1. Reaktivität: Das System reagiert ständig auf Eingaben (z. B. von Sensoren) und verarbeitet all diese Eingaben (z.B. den ununterbrochenen Druck auf eine Taste) ohne dabei jemals zu terminieren.

2. Unendliche Abläufe/ ständige Betriebsbereitschaft: Damit die Reaktivität gewährleistet wird, befindet sich das System in ständiger Betriebsbereitschaft. Zum Beispiel muss die Steuerung eines Fahrstuhls ständig bereit sein, um auf Anforderungen (z.B. auf das Drücken der Etagetaste) zu reagieren.

3. Bestandteil eines anderen Systems: Das eingebettete System ist nur ein Teil eines ganzen Systems. Zum Beispiel ist das Anti- Blockiersystem (ABS) Bestandteil eines Fahrzeugs.

4. Stückzahl: Die Stückzahl, die von den Systemen hergestellt wird, ist meist sehr hoch (z.B. Handys).

5. Reliability (Zuverlässigkeit): Da die Ausbringungsmenge sehr hoch ist, muss das System auch eine hohe Zuverlässigkeit besitzen. Eine Rückführung des Produkts (das ein eingebettetes System beinhaltet) wäre sehr teuer. Dies gilt besonders für sicherheitskritische Systeme.

6. Maßgeschneiderte Hardware: Da eingebettete Systeme meist nur eine ganz spezielle Aufgabe zu erfüllen haben, ist auch die Hardware eines solchen Systems seiner Aufgabe ganz speziell angepasst. Z.B. hat ein Handy keinen Anschluss für einen externen Monitor.

Immer mehr mechanische und elektrische Komponenten werden durch Elektronik ersetzt. Dies führt zu einer Verringerung des Gewichts und des Volumens, sowie zu größerer Flexibilität durch die Software.
Eingebettete Systeme vereinigen somit die große Flexibilität der Software mit der Leistungsfähigkeit der Hardware.

Ein anderer Aspekt ist, dass die Komplexität der Systeme steigt. Ein Rasierapparat hat weniger als 100 KB Steuersoftware, während ein Navigationssystem mehr als 30 MB Software umfasst. Echtzeitanforderungen, Vernetzung von Steuergeräten und sicherheitskritische Aspekte erhöhen die Komplexität ebenfalls.

Verglichen mit herkömmlichen Computern sind eingebettete Systeme beschränkt und nur im seltensten Fall ausbaufähig[1]. Sie werden aber unter anderem wegen ihres niedrigen Stromverbrauchs, ihrer geringen Größe, ihrer Zuverlässigkeit und ihres niedrigen Preises eingesetzt.

[1] Bsp. für eine Ausnahme: VME (Virtual Machine Environment) mit PPCs

1. 2 Sicherheitskritische Systeme

Sicherheitskritische Systeme sind eingebettete Systeme, die für die Steuerung von Systemen verantwortlich sind, welche bei einer Fehlfunktion das Leben von Menschen gefährden. Für solche Systeme muss der Testaufwand so hoch sein, dass ein Fehlverhalten bzw. eine Fehlersituation praktisch ausgeschlossen ist.
Beispiele für sicherheitskritische Systeme sind:

- ABS- Steuerung bei PKWs
- Steuersysteme in Flugzeugen

1. 3 Nicht- sicherheitskritische Systeme

Nicht- sicherheitskritische Systeme sind solche Systeme, bei denen davon ausgegangen wird, dass bei einem Fehlverhalten kein Menschenleben bedroht ist. Man könnte jetzt anfangen zu diskutieren, dass auch eine Fehlfunktion in einem Toaster zu einem Hausbrand führen könnte und dadurch Leben gefährden könnte. Da diese Diskussion aber nicht Bestandteil dieser Arbeit ist, wird darauf nicht näher eingegangen. Der Testaufwand muss aus diesem Grund auch nicht ganz so hoch sein wie bei sicherheitskritischen Systemen.
Beispiele für nicht- sicherheitskritische Systeme:

- Videorekorder
- Telefone

1. 4 Definition Echtzeit- Systeme

Im Zuge der technischen Entwicklung nimmt die Verbreitung von Echtzeit- Systemen in den unterschiedlichsten Anwendungsbereichen rapide zu. Echtzeit- Systeme finden heutzutage ihren Einsatz in allen Bereichen wie z.B. in der Produktion, Luft- und Raumfahrttechnik, Medizintechnik und Militärtechnik.

Die wesentliche Anforderung dieser Systeme lautet knapp formuliert:

> „Die Gültigkeit einer Operation eines Echtzeit- Systems hängt ab von
>
> - dem logischen Ergebnis einer Berechnung und
> - von der physikalischen Zeit, die dafür benötigt wird“

Eine genauere Definition von Realzeit-Systemen lautet nach DIN 44300 [1985]:

> *„Realzeit-Systeme beziehungsweise Echtzeit- Systeme sind Computersysteme, die im Realzeitbetrieb arbeiten.*
> *Realzeitbetrieb wird definiert als der Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspannen verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen.“*

In einem Echtzeit- System ist also neben der logischen Richtigkeit der Antwort (funktionale Korrektheit) auch die Einhaltung garantierter Reaktionszeiten (temporale Korrektheit) von essentieller Bedeutung. Das zeitliche Verhalten bei Echtzeit- Systemen
ist also Teil der Spezifikation des Systemverhaltens und somit Gegenstand der Überprüfung.

Bei den zeitlichen Anforderungen der Echtzeit- Systeme kann man folgende Zeitbedingungen unterscheiden:

- **Harte Echtzeit:** Harte Zeitbedingungen stehen präzise fest und sind durch das System unbedingt einzuhalten. Abweichungen von den Zeitvorgaben sind nicht zulässig und stellen einen schwerwiegenden Fehler dar. In sicherheitsrelevanten Systemen kann das Verletzen der Zeitvorgaben zur Gefährdung von Menschenleben führen. Beispiele hierfür sind Navigationssysteme in der Luft- und Raumfahrt und Airbag-Zündungen.

- **Weiche Echtzeit:** Weiche Zeitbedingungen hingegen lassen sich oft nicht genau festlegen bzw. sie werden genau definiert, aber eine Überschreitung dieser Vorgabe führt zwar zu Störungen, das System bleibt aber funktionsfähig und das Betriebsziel kann mit Verspätung oder höheren Kosten noch erreicht werden. Beispiele hierfür sind Toaster, Kühlschrank, Telefon und MP3- Player.

Eingabedaten müssen bei einem Echtzeitsystem pünktlich eingelesen werden und die zu erzeugenden Ausgabedaten müssen innerhalb der vorgegebenen Zeitbedingung bereitgestellt werden. Primäres Betriebsziel ist also, ständig dazu bereit zu sein, rechtzeitig mit den gewünschten Ausgaben auf Eingabedaten zu reagieren (Reaktivität). Dies muss unabhängig von der Systemauslastung geschehen. Zu einem Echtzeitsystem gehört auch, dass es die zeitliche Gültigkeit von Informationen überwacht und die Verwendung nicht-zeitgemäßer Daten verhindert.

Um dies garantieren zu können, muss das zeitliche Systemverhalten vorhersehbar sein. Dies muss unabhängig davon gelten, ob die zu verarbeitenden Daten zufällig auftreten oder zu vorherbestimmten Zeitpunkten. Diese Forderung ist zur Zeit in der Praxis nur sehr selten zu erfüllen. Aufgrund der Komplexität der Systeme ist das zeitliche Verhalten heute verfügbarer Rechensysteme höchstens in Ausnahmefällen vollständig vorhersehbar. Solche Ausnahmen sind z.B. Flugzeuge, Nuklearwaffen.

2 Warum Software testen

Die Geschichte zeigt uns, dass Fehler zu schlimmen Katastrophen führen können. Ein Beispiel dafür, was zwar nicht direkt mit einem Softwarefehler zu tun hat, ist das Abstürzen des gesamten Systems von einem amerikanischen Zerstörer. In diesem Fall hat eine Motte, die auf einer Speicherplatine des Schiffes landete, zu einem Kurzschluss geführt. Seit diesem Absturz bezeichnet man Fehler auch als „Bugs“.

Natürlich ist es nicht nur eine Frage des Risikos, sondern auch eine Frage des Geldes. Fehler können zum Beispiel zu teuren Rückholaktionen für Firmen werden, falls ein Fehler zu spät erkannt wird. Es ist somit auch eine Frage für das Management, wie viel Ressourcen sie für Fehlersuche genehmigen, um den maximale Projektgewinn zu erhalten. Die folgende Abbildung veranschaulicht dieses Problem:

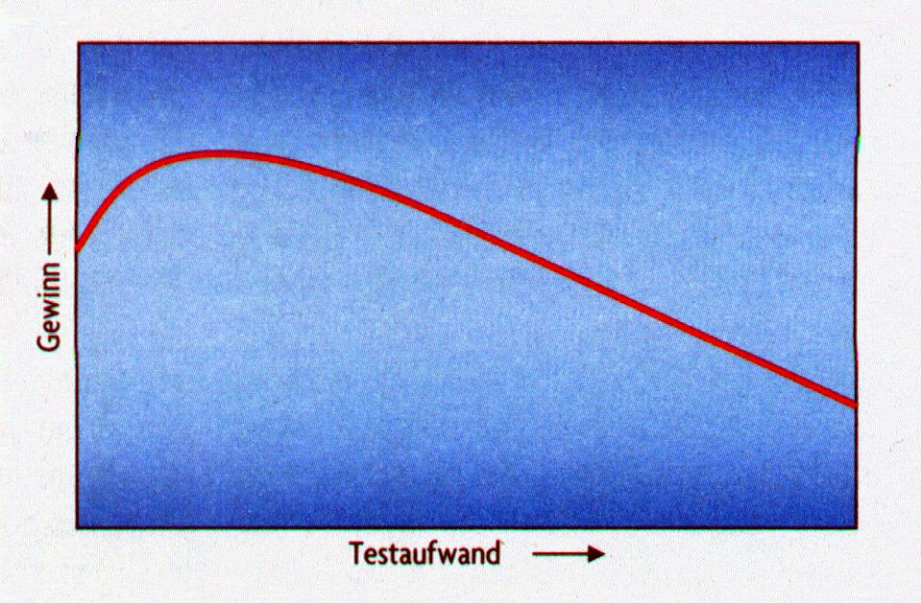

Abbildung 2.1: Testaufwandsplanung des Managements

Im Gegensatz zu dem Management ist es Aufgabe des Testers, mit den bereitgestellten Ressourcen so viele Fehler wie möglich zu finden. Dabei muss man berücksichtigen, dass Tests nur geeignet sind um Programmier- und Designfehler zu finden. Der beste Test hilft nicht gegen Kommunikationsfehler in der Spezifikation.

Aus dieser Erkenntnis wird klar, dass nicht alle Fehler auf den einfachen Programmierfehler geschoben werden können. Die nachfolgende Auflistung zeigt einige Gründe für das Entstehen von Fehlern:

- **Fehlerhafte oder keine Kommunikation bei der Spezifikation:** Bei der Spezifikation wird nicht genau festgelegt, was die Software leisten soll und was nicht. Des Weiteren kann es vorkommen, dass der Entwickler und der Auftraggeber die Spezifikation unbewusst unterschiedlich verstehen und interpretieren. Solche Probleme führen zu einem falschen Design der Software.

- **Änderung in der Anforderung:** Während der Entwicklung ändern sich die Anforderungen der Software ständig, so dass es kein Wunder ist, dass man schnell den Überblick verliert. Oft ist dann nicht mehr klar, in welcher Version welche

Anforderung realisiert wurde und welche Tests wegen einer Änderung modifiziert oder durchlaufen werden müssen.

- **Komplexität der Software:** Bei Projekten werden Module von Zulieferern oder anderen Projekten in Applikationen verwendet. Diese Module verwenden komplizierte Algorithmen, funktionieren auf einer Vielzahl von Plattformen usw. Bei solch einem großen Umfang kann es vorkommen, dass ein Programmierer nicht alles versteht und jedes Detail kennt. Diese Situation kann zu Fehlern führen.

- **Wachsender Zeitdruck:** Die Zeitpläne für Entwickler sind in der Regel sehr knapp bemessen. Da kommt es nicht selten vor, dass zum Ende eines Projektes der Abgabetermin auf den Entwickler einen enormen Druck ausübt und dass dadurch schnell etwas fertig gestellt werden muss. In dieser hektischen Arbeitsweise kommt es oft vor, dass etwas übersehen oder vergessen wird und somit zu Fehlern führt.

- **Psychologische Gründe:** Es ist ein Zeichen von Stärke, zuzugeben, etwas nicht verstanden zu haben oder etwas nicht zu können. Es ist jedoch nicht einfach, dies vor seinem Chef zuzugeben. Diese Situation führt dazu, dass manche Entwickler es nicht zugeben und es dazu zu Fehlern kommt. Des Weiteren überschätzen viele Chefs ihre Angestellten und beauftragen sie mit zu schweren Aufgaben.

- **Schlecht dokumentierter Code:** Wieder verwendete Quellcodes wurden von Programmierern entwickelt, die keine Freunde von guter Dokumentation waren bzw. ist für diese Programmierer die Dokumentation leicht verständlich, aber für andere Entwickler unverständlich.

- **Programmierfehler:** Entwickler sind auch nur Menschen und können sowohl beim Design als auch bei der Umsetzung des Designs Fehler machen.

3 Testen von eingebetteten und Echtzeit- Systemen

Das Planen von Testreihen ist im Allgemeinen eine Frage der Projektplanung. Es sollte schon während der Entwicklung des Software- Designs parallel mit dem Test- Design begonnen werden. Dies würde die Testphase verkürzen, untestbare Anforderungen identifizieren und ggf. die Notwendigkeit von spezieller Test- Hardware erkennen.

Mit dem Testen selbst kann erst begonnen werden, wenn die ersten Module der Software fertig gestellt sind. Es besteht daher eine umgekehrte Reihenfolge zwischen dem Test- Design und den Tests selbst. Dies wird aus dem Software- Entwicklungsmodel in der folgenden Abbildung deutlich:

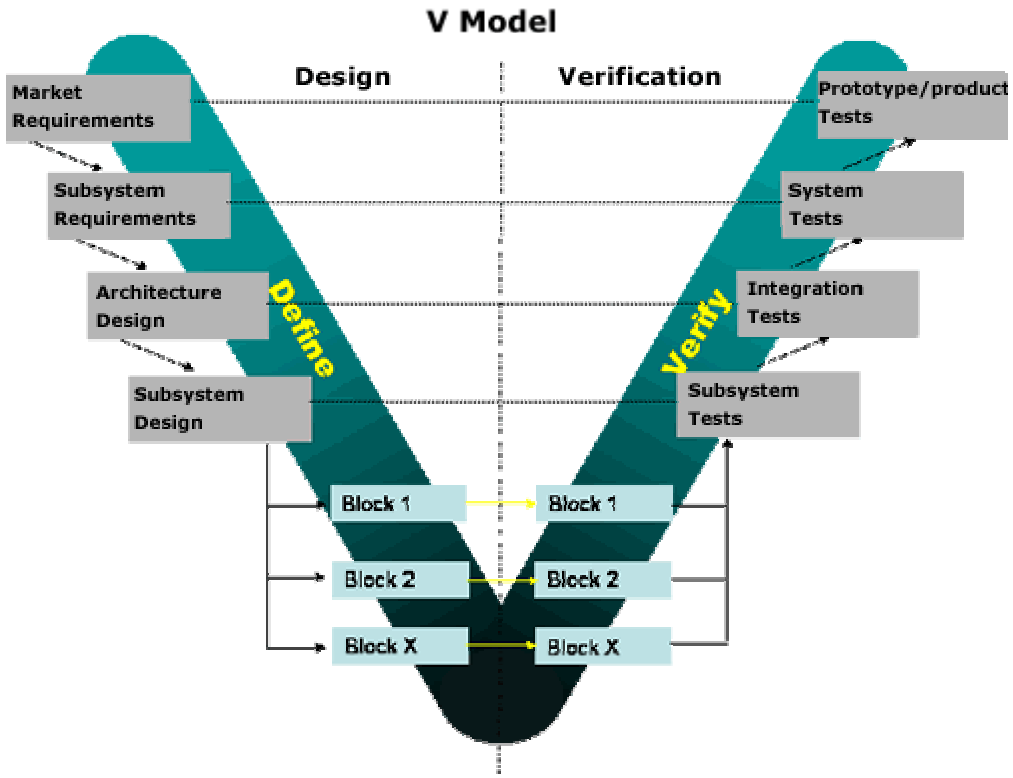

Abbildung 3.1: Das Software- Entwicklungsmodell (V-Modell)

Die nachfolgenden Unterkapitel sind in der Reihenfolge aufgebaut, in der die Software getestet wird. Begonnen wird daher mit dem Modultest, der ihm folgende Test ist der Integrationstest und der abschließende Test der Systemtest.

Die Tests selbst sollten nicht von den Entwicklern durchgeführt werden, die die Software selbst entwickelt haben. Testen selbst ist eine destruktive Tätigkeit, in der versucht wird, die Software zu brechen und Fehler zu finden. Dem Tester muss die Fehlerfindung Spaß machen, damit er sich anstrengt, Fehler zu finden. Welcher Entwickler hat Spaß dabei, seine eigenen Fehler zu finden?

3.1 Modultest

Der Modultest ist der erste Test, der in der Testreihe durchgeführt wird und ist somit der Test auf der untersten Ebene. Bei diesem Test werden die einzelnen Module vor dem

Zusammenfügen auf ihre Richtigkeit geprüft. Es gibt zwei gängige Methoden, einen Modultest durchzuführen.

Bei der ersten Methode, dem so genannten Black- Box- Test, werden sämtliche Testfälle anhand der Spezifikation des Moduls gebildet, d.h. der Tester hat keinerlei Einsicht in den Quellcode bzw. in die interne Realisierung des zu testenden Moduls. Er kommuniziert während des Tests ausschließlich über die extern sichtbare Schnittstelle mit dem Modul.

Bei der zweiten Methode wird der Quellcode im Gegensatz zum Black- Box- Test genau analysiert. Aus diesem Grund werden diese Testfälle White- Box- Test genannt.

3. 1. 1 Black- Box- Test

Bei dem Black- Box- Test beginnt man am besten mit der Identifikation bzw. Findung von Äquivalenzklassen. In einer Äquivalenzklasse finden sich alle Eingangsgrößen oder Resultate eines Moduls, die die Erwartung erfüllen, dass bei einem Fehler entweder alle Werte oder kein Wert betroffen sind.
Bei der Funktion, die den Absolutbetrag einer ganzen Zahl berechnet, existieren z. B. drei Äquivalenzklassen:

- Positive Zahlen
- Negative Zahlen
- Zahl Null

Wenn man alle Äquivalenzklassen definiert hat, wählt man aus jeder Klasse einen oder mehrere Vertreter. Anhand dieser wenigen Vertreter muss man versuchen, möglichst viele Fehler zu finden, bzw. abzuschätzen, ob ein Modul richtig funktioniert. Dies ist anhand von wenigen Werten nicht sehr einfach, weswegen die Werte wohl überlegt sein müssen.
Es empfiehlt sich, eine so genannte Grenzwertanalyse durchzuführen um von jeder Äquivalenzklasse die Grenzwerte zu finden. Hierbei darf nicht vergessen werden, auch die Grenzwerte für den Ausgang zu berücksichtigen. Dies geschieht, indem man die entsprechenden Eingangsgrößen wählt.

Die gewünschten Eingangswerte werden mit den dazugehörigen Aktionen und den erwarteten Ergebnissen in dem Testdesign festgelegt. Während der Testdurchführung werden die Ergebnisse mit den erwarteten Ergebnissen verglichen.

Die Testausführung kann manuell mit Papier, Stift und Hilfe eines Debuggers oder automatisch durchgeführt werden. Bei der automatischen Durchführung muss ein Programm geschrieben werden, welches das Modul initialisiert, aufruft und die Ergebnisse mit den Erwartungswerten vergleicht. Eine solches Programm bzw. eine solche Software wird „Treiber“ genannt.
Die Größe des Treibers hängt von dem Modul ab. Werden zum Beispiel andere Module von dem zu testenden Modul aufgerufen, so müssen diese simuliert werden, damit das Modul unabhängig bleibt. Das Simulieren dieser Module erfolgt durch so genannte „Stubs“ (Programmstümpfe), die übergebene Parameter entgegen nehmen und gegebenenfalls welche an das Modul zurückliefen.

Durch diese Programmstümpfe bekommt der Tester die Möglichkeit, die Parameter, die das zu testende Modul übergibt, zusätzlich zu prüfen und eine Fehlersituation durch falsche Rückgabewerte in dem aufgerufenen Modul zu simulieren.

Die Anzahl der Werte, die man zum Testen des Moduls verwendet, ist abhängig davon, wie sorgfältig das Modul getestet werden soll und wie viel Zeit und Geld das Management für das Testen einplant. Bei sicherheitskritischen Systemen könnte gefordert werden, dass man mindestens einen Wert von jedem Grenzwert zum Testen benutzt.

3. 1. 2 White- Box- Test

White- Box- Test sind zusätzliche Tests, die erstellt werden, falls nach den Black- Box- Tests die gewünschte Testabdeckung nicht erreicht wurde. Diese Testabdeckung wird mit Hilfe von Modultest-Werkzeugen gemessen. Diese Werkzeuge unterstützen den Tester beim Erstellen der Treiber und Stubs, indem sie den Code dafür generieren. Der Tester hat nur noch die Aufgabe, Eingangswerte und Erwartungswerte in einer Tabelle zu definieren.

Zur Messung der Testabdeckung fügen diese Programme Codes in die zu testende Software ein. Anhand dieses Codes wird es ihnen möglich, den Quellcode zu instrumentieren und dem Tester die Programmteile zu zeigen, die noch nicht durch den Test erreicht worden sind.

Der Tester hat somit die Möglichkeit, die Testfälle auszubessern bzw. neue Testfälle zu erstellen um die gewünschte Testabdeckung zu erreichen. Wie bereits erwähnt, tragen diese Testfälle den Namen „White-Box- Test“, da sie durch eine genaue Analyse des Quellcodes und nicht nur durch die Definition der Schnittstellen des Moduls entstehen.

3. 2 Integrationstest

Der wohl unbeliebteste Test der Testreihe ist der Integrationstest. Hierbei geht es darum, die bereits getesteten Module zusammenzufügen und festzustellen, ob es bei der Vereinigung Probleme gibt. Falls diese Tests erfolgreich verlaufen sind, kann man beruhigt mit den Systemtests beginnen.

3. 1. 1 Bottum Up Unit Tests (BUUT)

Die wohl sicherste Methode, einen Integrationstest durchzuführen, ist, gar keinen durchzuführen, sondern die Modultests so zu organisieren, dass man sie auch auf Integration testet. Wenn man sich für eine solche Teststrategie entscheidet, spricht man vom Bottum Up Unit Test.

Dadurch, dass während des Modultests das Modul, welches das zu testende Modul aufruft bzw. die Module die von dem Modul aufgerufen werden, durch Treiber und Stubs simuliert werden, ist der Testaufwand sehr hoch. Diesen Aufwand kann man nur reduzieren, indem man nur den Treiber simuliert und anstelle der Stubs die realen Module bzw. Systemschnittstellen verwendet.

Entscheidet man sich für diese Integrationsart, muss man darauf achten, dass man eine genaue Reihenfolge bei der Integration befolgt. Abbildung 3.1.1.1 zeigt ein Beispiel von Grünfelder, welches die einfachste Ausgangslage darstellt.

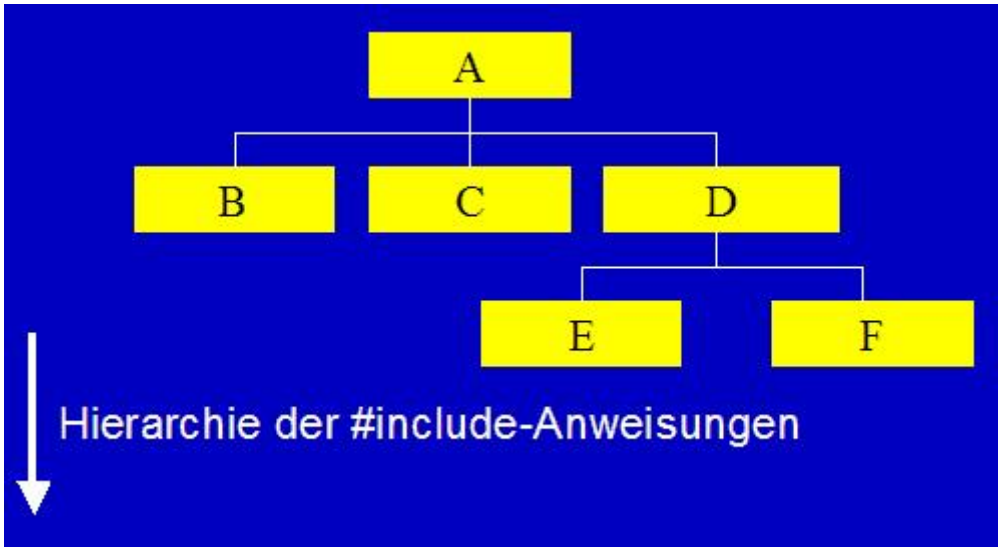

Abbildung 3.1 1.1: Abhängigkeiten gemäß der #include- Anweisungen

Wie man in diesem Bild erkennen kann, verwendet A die Funktionen oder Typen von B, C und D. Somit wäre A, laut der obigen Vereinbarung, ein Treiber für B, C und D.

Beginnt man nun mit den Tests, so ist es am sinnvollsten, mit den Dateien zu beginnen, die keine anderen inkludieren. Hierzu muss man sich, wie schon beim Modultest beschrieben, eine Testumgebung schreiben, die das Modul aufruft und die Ergebnisse vergleicht. Wenn die Hardware bei diesem Modultest betroffen ist, so spricht man von Hardware/ Software-Integrationstests.

Sind nun alle diese Module bzw. Dateien, die keine anderen inkludieren, getestet, so kann man mit Dateien fortfahren, die nur Dateien inkludieren, die bis zu diesem Zeitpunkt schon getestet wurden. Nach diesem System geht man weiter vor, bis alle Module/ Dateien getestet wurden. Das Bild in Abbildung 3.1.1.2 zeigt einen Zwischenschritt in diesem Prozess nach Grünfelder.

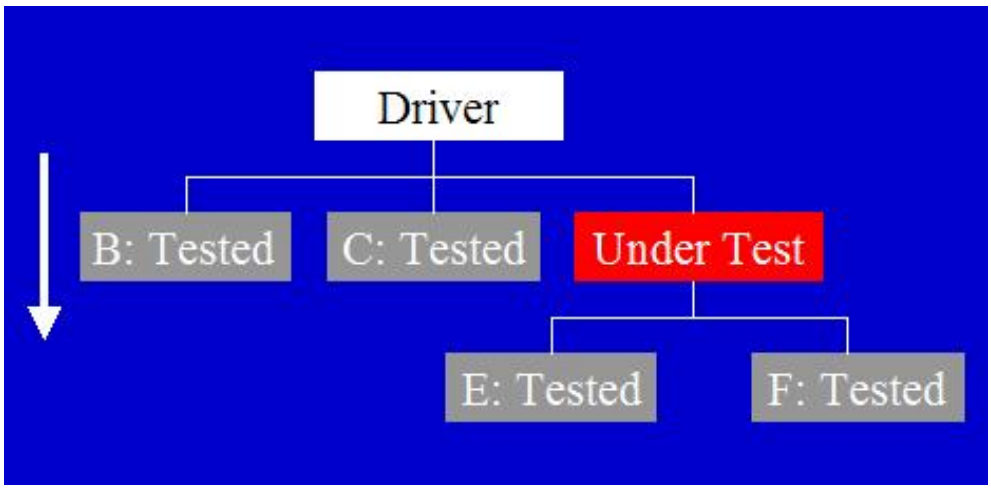

Abbildung 3.1.1.2: Test von Modul D

Diese Methode des Integrationstests ist die sicherste Methode und spart auch Arbeit. Allerdings verlangsamt sie den Testprozess während der Software- Entwicklung im Team erheblich, da man evt. auf das Fertigstellen eines Moduls warten muss. Dieses Problem kann man ein wenig einschränken, indem man eine Variante wählt, in der man die Integrationsreihenfolge ein wenig flexibler gestalten kann. Eine mögliche Variante wäre, erst alle Module isoliert zu testen und dann die Stubs Stück für Stück zu ersetzen.

3. 1. 2 Strukturierte Integrationstests (SIT)

Die Methode des strukturierten Integrationstest wurde von der Methode Strukturiertes Testen, die der Amerikaner Thomas McCabe 1982 vorstellte, abgeleitet. In seiner Idee richtet man sich nach den Kontrollflussdiagrammen, siehe Abbildung 3.1.2.1, wobei die Knoten in den Graphen Anweisungen und die Kanten Ausführungsreihenfolgen sind.

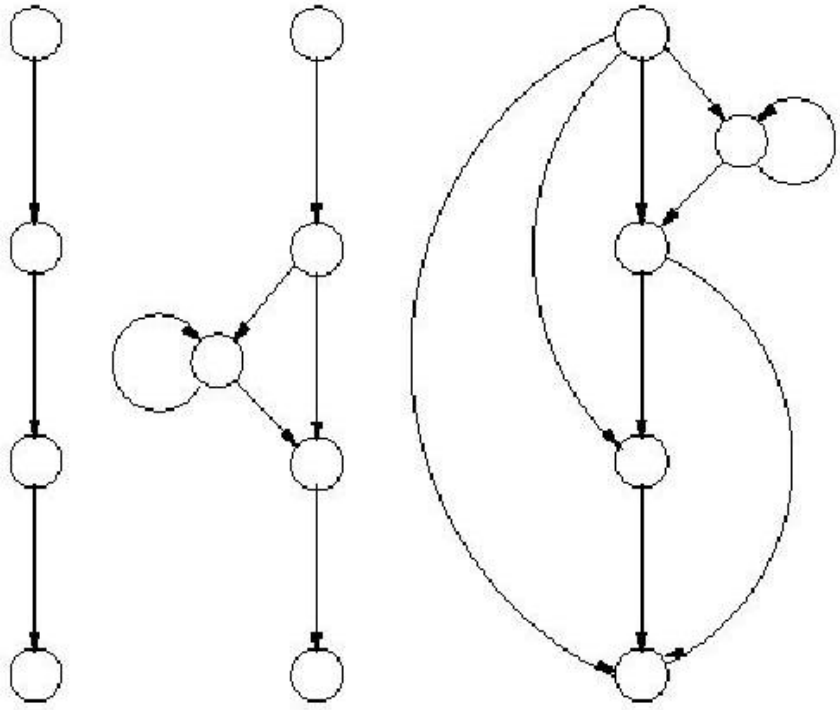

Abbildung 3.1.2.1: Kontrollflussdiagramme

Bei Anwendung dieser Methode auf den Integrationstest muss man nur die Schnittstellen des frisch integrierten Moduls beachten. Es wird davon ausgegangen, dass die Schnittstellen zu anderen Modulen wie Funktionsaufrufe sind und dass keine globalen Variablen existieren. Diese Aufrufe werden durch farblich markierte Knoten repräsentiert.

Wenn man sich jetzt überlegt, dass es für die Integration nicht nötig ist, Programmschleifen zu durchlaufen, die keinen markierten Knoten besitzen, kann man den Graphen auf die für den Integrationstest relevanten Knoten reduzieren, indem man derartige Programmteile entfernt.

Für diese Reduktion des Graphen existieren folgende Regeln:

1. Markierte Knoten dürfen nicht entfernt werden.
2. Nicht- markierte Knoten, die keine Verzweigungen enthalten, werden entfernt.
3. Kanten, die zum Beginn einer Schleife führen, die nur unmarkierte Knoten enthält, werden entfernt.
4. Kanten, die zwei Knoten so verbindet, dass kein Alternativpfad für diese Verbindung mit markierten Knoten existiert, werden entfernt.

Ein Beispiel für das Anwenden dieser Regeln sieht man in den Abbildungen 3.1.2.2-3.

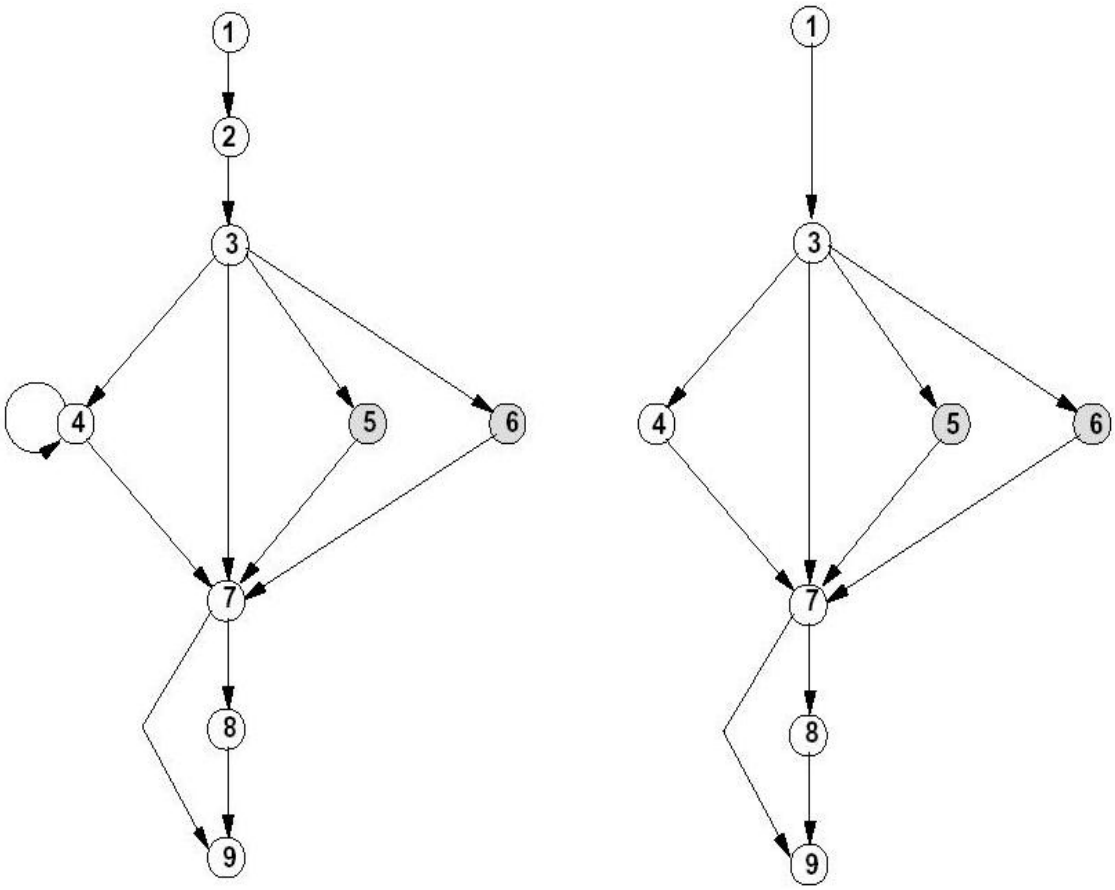

Abbildung 3.1.2.2: Vollständiger Graph und Graph nach zwei Reduktionsschritten

In dieser Abbildung sieht man links den vollständigen Graphen und recht den Graphen nach zwei Reduktionsschritten. Es wurde mit Hilfe von Regel 2 der Knoten 2 entfernt und mit Regel 3 die Miniaturschleife bei Knoten 4 entfernt. Die nächste Abbildung zeigt einen weiteren Reduktionsschritt, bei dem Knoten 4 mit Regel 2 entfernt wurde und die Kante von Knoten 7 nach Knoten 9 mit der Regel 4 entfernt wurde. Zusätzlich zeigt es den vollständig reduzierten Graphen, bei dem die Knoten 8, 9, 1 mit Regel 2 entfernt wurden.

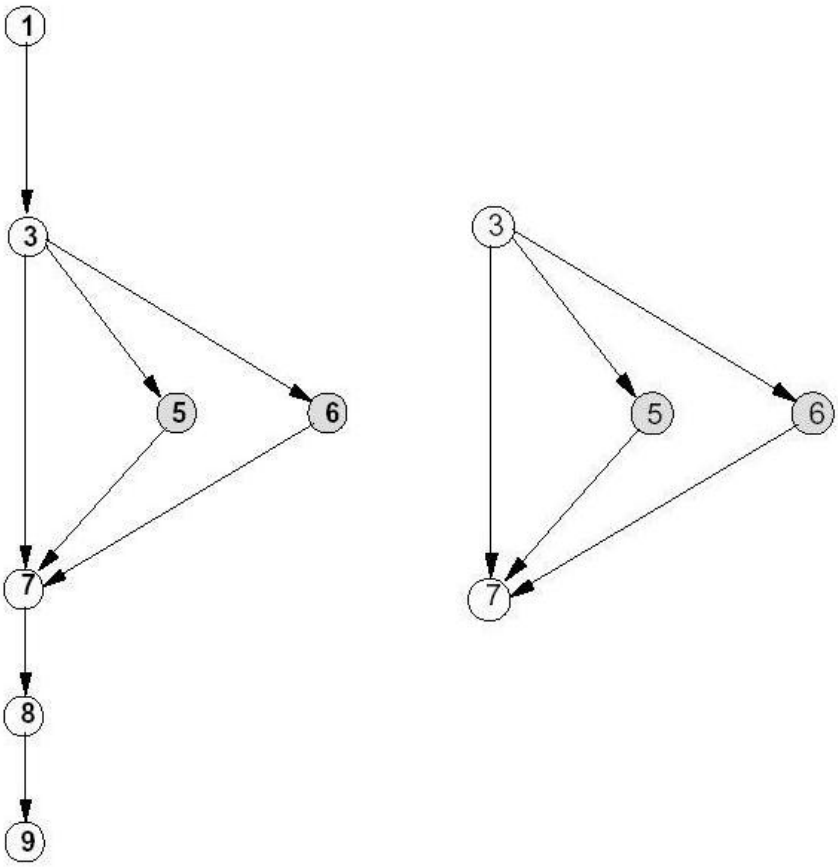

Abbildung 3.1.2.3: Weiterer Induktionsschritt und vollständig reduzierter Graph

Nachdem man die Reduktion des Graphen abgeschlossen hat, kann man mit dem Integrationstest nach der Methode von McCabe beginnen. Hierzu wird versucht, die minimale Anzahl voneinander unabhängiger Programmpfade, die noch nicht durch eine Linearkombination getestet wurden, zu durchlaufen.

Ein Weg, den Grünfelder vorschlägt um zu einem Satz von unabhängigen Programmpfaden zu gelangen, ist zunächst, einen beliebigen Pfad auszuwählen und ihn als erstes Element dieses Satzes zu definieren. Danach würde für jedes weitere Mitglied gelten, den Exekutionspfad aus diesem Satz an einer einzigen Verzweigung zu ändern.

Der linke Graph in der obigen Abbildung 3.1.2.1 würde den einfachsten Fall anzeigen, in dem nur ein Pfad möglich wäre. Dagegen wäre ein möglicher Kontrollfluss im mittleren Graphen eine Abarbeitung von oben nach unten. Es würde, wenn nur eine Verzweigung geändert werden soll, die Möglichkeit bleiben, bei der Abzweigung den linken Weg einzuschlagen und dabei die Schleife nicht zu berücksichtigen. Als letzte Variante würde der Exekutionspfad bleiben, in dem auch die Schleife durchlaufen würde. Schleifen werden dabei, wenn überhaupt, höchstens einmal durchlaufen.

Durch Kombination dieser möglichen Pfade würde man alle Möglichkeiten für Programmflüsse darstellen. Diese Anzahl der möglichen Programmpfade wird „zyklomatische Komplexität“ genannt und ist ein Maß für den Testaufwand. Berechnet wird sie aus der Anzahl der Kanten minus der Anzahl der Knoten plus zwei.

Zu beachten bleibt noch, dass, falls globale Variablen als Schnittstellen zu den Modulen gehören, die Instruktionen, die auf diese Variablen zugreifen, im Graphen zu markieren sind. In den Testfällen werden die Eingangsgrößen, der Funktionsaufruf bzw. die Aktion an den Systemgrenzen und die erwarteten Reaktionen/Ausgangsgrößen beschrieben. Dabei ist zu beachten, dass die Testfälle an den Außengrenzen des integrierten Softwarestandes ansetzen. Dies bedeutet auch, dass Aufrufe von Funktionen des integrierten Moduls durch andere Module in dem Verfahren erfasst werden müssen. Würde man dies nicht beachten, so würde nach Grünfelder, der Integrationstest das Zielsystem nur als Dienstleister für das neu integrierte Modul sehen und die eigentliche Funktion des Zielsystems außer Acht lassen. Es würden daher keine Fehler an diesen Aufruf-Schnittstellen erfasst werden!

3. 1. 3 Testabdeckung der Aufrufe von Unterprogrammen (TAU)

Die einfachste Teststrategie von diesen drei Methoden ist das Testen nach der Strategie von „Testabdeckung der Aufrufe von Unterprogrammen“. Diese Strategie arbeitet mit dem Messen der Testabdeckung von funktionalen Systemtests. Genauer gesagt wird der Anteil der ausgeführten Aufrufe von Unterprogrammen in der Software gemessen. Diese Testabdeckung nennt man „Call Pair Coverage“.

Es werden also nach dem vollständigen Test der Module alle Module auf einmal zusammen gefügt und mit den Systemtests begonnen, die die Einhaltung der Anforderungen des Gesamtsystems überprüfen.

Wenn nun z.B. die Software nur aus einem Hauptprogramm und einem Unterprogramm besteht, welches an zwei Stellen im Hauptprogramm aufgerufen wird, dann ist die Call Pair Coverage der Test 50%, falls das Unterprogramm nur einmal aufgerufen wird.
Diese Tests werden nun so lange durchlaufen bis die Testabdeckung bei 100% ist. Falls das geschehen ist, kann man davon ausgehen, dass die Software in Bezug auf den Integrationstest ordnungsgemäß funktioniert.

Wenn alle Systemtests durchlaufen sind und die Testabdeckung noch nicht die 100% erreicht hat, ist dies ein Zeichen dafür, dass das Design mehr Funktionalität beabsichtigt hat, welches durch den Test nicht abgedeckt wird. In diesem Fall ist die Liste der Testfälle um weitere Tests zu erweitern, falls kein Test versehentlich übersehen wurde. Laut Grünfelder werden diese ungetesteten Schnittstellen gelegentlich bei Hardware/Software- Schnittstellen erkannt, wenn etwa regelmäßig die korrekte Funktion der Hardware geprüft wird oder an der Schnittstelle der Anwendersoftware zum Betriebssystem, wenn das Vorhandensein von Systemressourcen geprüft wird.

Offen bleibt hier noch die Frage, wie man feststellt, wann die Testabdeckung 100% beträgt.
Die billigste Möglichkeit ist das Anlegen einer Cross- Reference- Tabelle, in der alle Aufrufe von Unterprogrammen notiert werden. Diese Tabelle kann manuell oder werkzeugunterstützt angelegt werden. An dieser Stelle muss man nur noch einen Testfall identifizieren, der den ersten Aufruf in der Tabelle aufruft. Ein Breakpoint an dieser Stelle gibt beim Durchführen des Testfalls die Sicherheit, dass die gewünschte Stelle durchlaufen wurde. Wenn man Echtzeit- Systeme testet, oder bei nicht vorhanden sein eines Debuggers, muss darauf verzichtet werden. An dieser Stelle muss man sich dann auf die Analyse des Quellcodes verlassen. Der letzte Schritt des Testlaufes ist das Verifizieren des Ergebnisses.

Eine einfachere Variante der Erhebung der Testabdeckung ist die Verwendung eines Instrumentierungswerkzeugs, welches im Lieferumfang von Werkzeugen von Modultest enthalten ist. Bei Verwendung eines solchen Werkzeuges muss man vor jedem Unterprogrammaufruf ein Aufruf einer Log-Funktion einfügen. Werden jetzt alle Aufrufe dieser Log-Funktionen gezählt, berichtet die Bibliotheksfunktion des Instrumentierungswerkzeuges eine Testabdeckung von 100%.

3. 3 Systemtest

Die letzte Stufe in der Testreihe sind die Systemtests. Nach diesen Tests wird das Produkt an den Kunden übergeben. Diese Tatsache, dass das Produkt danach an den Kunden übergeben wird, macht diese Tests für Software jeder Branche und Sicherheitsstufe enorm wichtig. Diese Tests testen das Gesamtprodukt und sollten auf dem Zielsystem oder einem System laufen, welches das Zielsystem ausreichend genug simuliert.

Man spricht von funktionalen Systemtests oder auch Funktionstests ist, wenn die Tests die Umsetzung der Anforderungen prüfen, die in der Spezifikation festgelegt wurden. Damit man die Übersicht bei diesen Test nicht verliert und dadurch die Frage entsteht, welcher Test welche Anforderungen testet bzw. getestet hat, bedient man sich der Hilfe von Traceability-Tabellen. Abbildung 3.3.1 zeigt eine dieser Tabellen.

Req#	Req Short Text	Tested in
R/SRD/INIT/10	Self Test	ST-IOPS-FT01
R/SRD/INIT/20	Self Test Fail Silence	ST-IOPS-FT01
R/SRD/PP/05	Ready Message	ST-IOPS-FT01
R/SRD/PP/20	PP-bus Performance	ST-IOPS-FT02
R/SRD/PP/30	Reply Message Format	ST-IOPS-FT02
R/SRD/PP/40	SetPosDebTime Cmd	ST-IOPS-FT03
R/SRD/PP/50	SetNegDebTime Cmd	ST-IOPS-FT03
R/SRD/PP/60	SetRepIntv Cmd	ST-IOPS-FT02
R/SRD/PP/70	Start Command	ST-IOPS-FT02
R/SRD/PP/80	Report Message Format	ST-IOPS-FT02
R/SRD/SI/10	Switch Position Sampling Rate	ST-IOPS-FT03
R/SRD/SI/30	Debouncing Algorithm	ST-IOPS-FT04, ST-IOPS-FT03
R/SRD/RA/10	Missing Parameters	ST-IOPS-FT05
R/SRD/RA/20	PP-bus Error	ST-IOPS-FT05
R/SRD/RA/30	Watchdog Kicking	ST-IOPS-FT05
R/SRD/SB/10	Maximum CPU Load	ST-IOPS-FT06
R/SRD/SB/20	CPU Idle Time	ST-IOPS-FT07
R/SRD/SB/30	Memory Footprint	ST-IOPS-FT08

Abbildung 3.3.1: Traceability-Tabelle von Funktionstests

Beim Erstellen dieser Tabelle muss man darauf achten, dass man die richtige Kombination von Anzahl der Tests und Menge von Anforderungstests in einem Test findet. Falls man zu viele Anforderungen auf einmal testet, erzeugt man eine geringe Anzahl riesiger Tests, die

schwer wartbar sind. In dem anderen Fall erzeugt man viele einfache Tests. Die richtige Mischung zu finden, ist Aufgabe des Testdesigns.

Bei den Tests muss eine Menge von Möglichkeiten bedacht werden. Falls eine graphische Benutzerschnittstelle (GUI) vorhanden ist, muss diese auch zu genüge getestet werden. Die folgende Aufzählung skizziert einige Punkte, die nach Grünfelder bedacht werden müssen:

- Ob Tastatur- und Maus- Navigation durch die Fenster richtig funktioniert
- Ob Drag-and-Drop an den Stellen, an denen es erwartet wird, erlaubt und verboten ist
- Ob GUI-Standarts eingehalten werden (Größe, Position, Initialzustand von Fenstern)
- Ob die Eingabe unsinniger Werte – etwa minus 300 Grad Celsius – unterbunden wird
- Ob Fensterinhalte erneuert werden, wenn intensiv gerechnet wird
- Ob ein Abbruch rechenintensiver Funktionen in vertretbarer Zeit möglich ist
- Ob der am Bildschirm dargestellte Inhalt in sich konsistent ist oder etwa ein Fenster einen aktuelleren Datenstand anzeigt als ein anderes gleichzeitig sichtbares

Eine Automatisierung dieser GUI- Tests ist durch Capture/Replay-Tools möglich. Solche Tests speichern die Interaktion mit dem System in einer Makrosprache, die eine exakte Wiederholung zu einem späteren Zeitpunkt ermöglicht. Während dieser Wiederholung findet ein Vergleich der Bildschirmausgabe mit gespeicherten Bildschirminhalten statt.

Falls die Software eine Datenbank verwendet, so sollte diese einzeln getestet werden. Dabei sollte man jeden möglichen Datenbankzugriff testen und während dessen gültige und ungültige Daten einspeisen bzw. abzufragen. Die Datensätze müssen nun noch in der Datenbank auf Korrektheit kontrolliert werden.

Ein weiterer Teil des Systemtest sollte der Performance- Test bzw. Belastungstest sein. Er testet das Verhalten des Systems unter verschiedener erwarteter Last, so wie Überlast. Neben der Richtigkeit der Reaktion ist hier vor allem die Reaktionszeit von Bedeutung.

Ein spezieller Test ist hierbei der Stresstest, der darauf abzielt, Fehler, die aufgrund geringer Systemressourcen oder wegen eines Kampfes um Ressourcen auftauchen, zu finden. Gründe dafür können zu geringer Hauptspeicher oder zu wenig Plattenspeicher sein. In diesem Fall könnten Fehler auftreten, die bei ausreichend Ressourcen, nicht auftreten würden. Weitere Gründe für solche Fehler könnten durch einen Kampf um Exklusivrechte beim Datenbankzugriff oder durch hohe Buslast, hohe Netzauslastung und die Blockade von Schnittstellen durch parallel bediente Programme sein.

Wenn man an solche Fehlerquellen denkt, sollte man auch daran denken, die Ressourcen darauf zu prüfen, ob sie unter normaler Belastung für die geplante Lebensdauer ausreichend vorhanden sind. Solche Tests tragen den Namen „Ressourcen-Tests“ und könnten zum Beispiel die CPU-Last oder den Speicherverbrauch testen.

Der letzte Test in dieser Reihe von Systemtests bezieht sich auf die Kontrolle des Benutzerhandbuches. Es wäre schließlich keine Verbesserung des Rufs der Firma, wenn in dem Handbuch Behauptungen stünden, die nicht zutreffen würden. Für diesen Zweck kann man z. B. Beispiele aus dem Handbuch Schritt für Schritt durchgehen und schauen, ob sie mit der aktuellen Softwareversion übereinstimmen. Zusätzlich muss es noch auf inhaltliche Fehler geprüft werden.

Wenn alle Tests erfolgreich verlaufen sind, so müsste der Auslieferung nichts mehr im Wege stehen. Man sollte sich trotzdem im Klaren darüber sein, dass Software nie zu 100% getestet ist. Wenn man sie auf 100% testen wollte, so müsste man Jahre evt. Jahrzehnte nur mit Testen verbringen.

3. 4 Host- oder Target- Testing

Beim Testen von Software, die speziell für eingebettete Systeme geschrieben wurde, bleibt die Frage offen, wo man die Softwaretests am besten durchführt. Es stehen zwei Möglichkeiten zur Auswahl, zwischen denen man sich entscheiden kann. Wenn man sich entscheidet, die Software auf dem Zielsystem zu testen, spricht man vom „Target- Testing". Der andere Fall wäre das „Host- Testing", welches die Software auf dem Entwicklungssystem testet. Welche Entscheidung man letztendlich trifft, hängt von mehreren Faktoren ab.

Natürlich sprechen viele Faktoren dafür, dass man die Software auf dem Zielsystem testet:

- Datentypen auf dem Zielsystem haben eine andere Wertigkeit als auf dem Entwicklersystem. Der Typ int hat die Bitbreite des Zielsystem und das Vorzeichen des Typs char ist so definiert wie im Zielsystem.

- Betriebsystemroutinen müssen nicht durch Programmstümpfe ersetzt werden

- Durch evtl. gemischte C/Assembler- Programmierung sind die Tests nur am Zielsystem durchführbar. Eine andere Möglichkeit dazu wäre ein Test am Simulator des Prozessors, welcher aber auch Fehler haben könnte, da es auch nur Software ist.

- Der Test findet nur so Compiler- Fehler oder Fehler der Standartbibliothek des Compilers.

- Interpretationsfreiheiten des Compiler- Herstellers ist nur im Zielsystem möglich, wie z.B. Rechts-Shift eines int-Wertes.

Es existieren aber auch gute Gründe, die gegen ein Testen auf dem Zielsystem sprechen:

- Das Zielsystem bietet nur selten die Möglichkeit, das Testresultat auf einem Bildschirm wiederzugeben.

- Es kostet viel Zeit, einen Emulator für das Zielsystem zu programmieren.

- Das Zielsystem bietet nur einen Debugger der halb so mächtig ist wie der auf dem Entwicklersystem.

- Der virtuelle Hauptspeicher ist im Entwicklersystem unbegrenzt, was den Einsatz von beliebig großer Testsoftware ermöglicht.

- Meist existiert vom Zielsystem nur ein Prototyp, d.h. alle Entwickler müssen sich die Zeit am Prototyp teilen.

4 Schedulinganalyseverfahren für Echtzeit- Systeme

In diesem Kapitel wird eine kurze Einführung geben, was man unter Schedulinganalysen versteht und weshalb sie gemacht werden. Des Weiteren werden zwei unterschiedliche Verfahren vorgestellt, wie man eine solche Analyse durchführen kann. Es werden dabei nur die grundlegenden Ideen der einzelnen Verfahren vorgestellt, da eine komplette Erläuterung den Rahmen dieser Arbeit sprengen würde.

An Echtzeitsysteme wird die Forderung gestellt, sowohl die Korrektheit als auch das Einhalten einer zeitlichen Schranke zu garantieren. Die Ermittlung dieser zeitlichen Schranke, welche nicht überschritten wird (Deadline), wird durch Scheduling- Verfahren ermöglicht. Diese Verfahren erlauben sogar im Vorhinein eine Analyse der Systeme.

Der Begriff Scheduling bedeutet im Deutschen etwa so viel wie „Ablaufsteuerung“ oder „Ablaufplanung“. Dies bedeutet eine zeitliche wie auch räumliche Zuordnung zu Instanzen, welche eine solche durchführen können. Ein Schedule ist ein konkreter Ablaufplan für ein gegebenes System. In diesem Zusammenhang bezeichnet Scheduling allerdings die Zuordnung von Tasks (Prozessen) zu einem Prozessor.

4.1 TDMA Scheduling Verfahren

Das TDMA (*Time Division Multiple Access*)- Verfahren ist ein sehr verbreitetes rundenbasiertes Scheduling- Verfahren. Jeder Task hat dabei pro Runde ein Zeitfenster (Slot), in welchem er den Prozessor nutzen kann. Diese Zeitfenster behalten in jeder Runde die Reihenfolge und ihre Größe bei. Falls ein Prozess (Task) in einer Runde nicht fertig gestellt wird, so wird die Bearbeitung in der nächsten Runde fortgesetzt.

Dieses Vorgehen kann man sich mit einem so genannten Gantt- Diagramm, welches zu graphischen Darstellung verwendet werden, verdeutlichen. Abbildung 4.1.1 zeigt einen solchen Graphen.

In der vertikalen Anordnung findet man die laufenden Prozesse, während die horizontale die Zeitspanne darstellt. Wenn nun ein Prozess den Prozessor in einer bestimmten Zeit nutzt, so wird dies durch Ausfüllen des Bereiches gekennzeichnet.

Das Verfahren wird an dieser Stelle mit Hilfe eines kleinen Beispieles verdeutlicht. Für dieses Beispiel werden eine Reihe von Abkürzunge und Variablen benötigt, welche allgemein für die Beschreibung von Scheduling- Verfahren genutzt werden.

Task_i	I=1,...,n
core(i)	Ausführungszeit von Task_i
slot(i)	Zeitbereich von Task_i pro Runde
avgDist(i)	Durchschnittlicher Zeitabstand zwischen zwei Aktivierungen von Task_i
turn	Rundenzeit, $\sum_j$ slot(j)
n_turn(i)	Von Task_i benötigte Anzahl von Runden zur Ausführung. n_turn(i)=[core(i)/slot(i)]
resp(i)	Antwortzeit von Task_i; Differenz zwischen Aktivierung und Beendigung der Ausführung

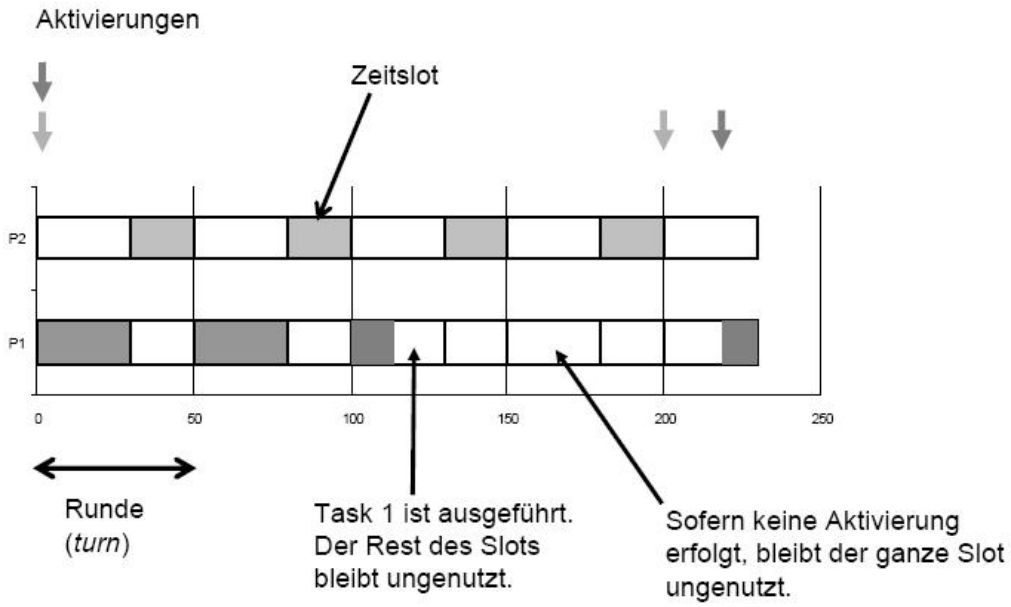

Abbildung 4.1.1: Gnatt- Diagramm des TDMA Verfahrens

Wie aus der Abbildung ersichtlich wird, besteht der Schedule aus zwei Prozessen. Der erste Prozess P1 hat dabei eine Slotgröße von 30 Zeiteinheiten, während die Slotgröße von P2 nur 20 Zeiteinheiten beträgt. Ihre Aktivierungen erfolgen durchschnittlich alle 220 bzw. 200 Zeiteinheiten.

Die Rundenzeit wird berechnet, indem man die Summe der einzelnen Zeitslots nimmt. In unserem Beispiel wäre die Rundenzeit *turn=30+20=50* Zeiteinheiten. Die Ausführungszeit der einzelnen Prozesse, also die core execution time, kann man aus der Graphig ablesen und wie in Abbildung 4.1.2 berechnen.

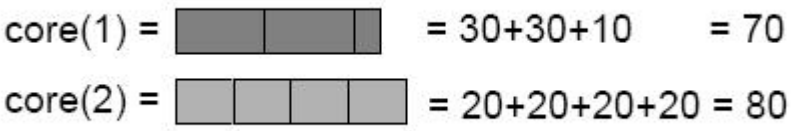

Abbildung 4.1.2: core execution time

Die Antwortzeit erhält man, indem man den Zeitabstand zwischen der Aktivierung und dem Zeitpunkt bestimmt, an dem der Task vollständig beendet ist. Die Abbildung 4.1.3 verdeutlicht dies noch einmal.

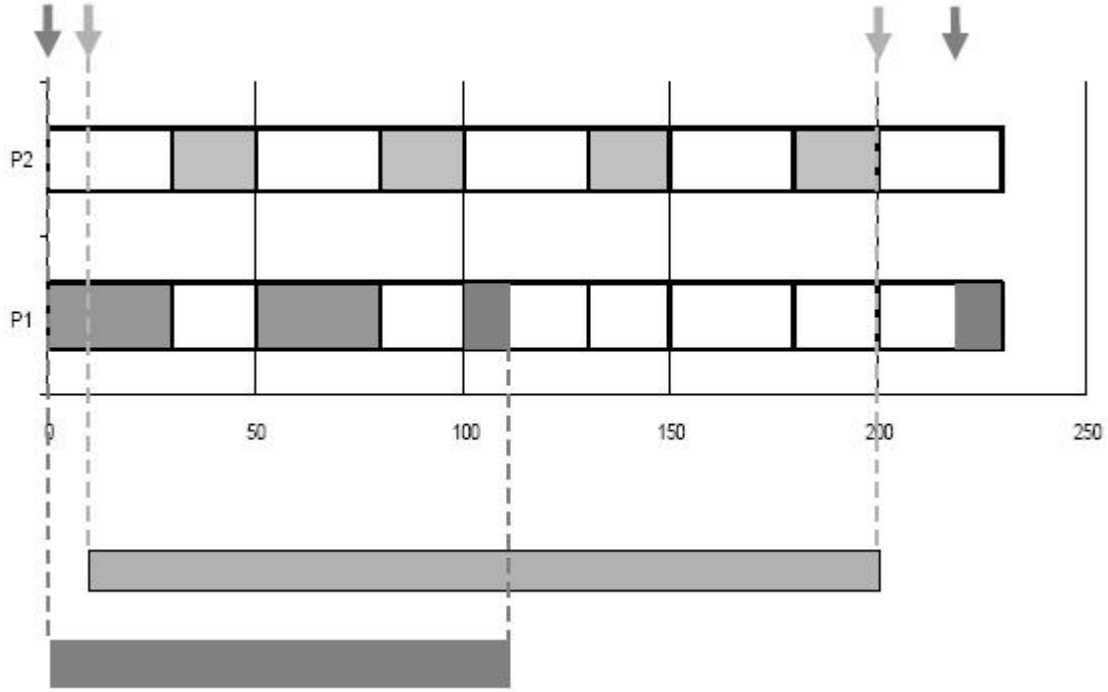

Abbildung 4.1.3: Antwortzeit der Prozesse

Für unser Beispiel wäre resp(1) gleich 110 Zeiteinheiten und resp(2) gleich 190 Zeiteinheiten lang. Diese Antwortzeit spielt für die Analyse eine wichtige Rolle, da die minimalen bzw. maximalen Antwortzeiten für den Best Case bzw. Worst Case stehen.

Die eigentliche Analyse besteht jetzt darin, Berechnungen durchzuführen, die eine Aussage über die Schedulbarkeit des Systems machen. Diese Berechnungen werden „schedulability conditions“ genannt. Falls diese Berechnungen ergeben, dass das System schedulbar ist, wird mit der Analyse begonnen. Diese Analyse überprüft jeden einzelnen Prozess und bestimmt die minimale und maximale Antwortzeit.

4.2 Round Robin Verfahren

Das Round Robin Verfahren ist dem TDMA- Verfahren sehr ähnlich und beruht auf dem gleichen Prinzip. Der wesentliche Unterschied ist, dass die Prozesse, falls sie frühzeitig fertig sind, die benötigten Ressourcen freigeben. Dies hat den großen Vorteil, dass der Prozessor zum Beispiel immer zu 100% ausgelastet ist. Veranschaulicht wird das in Abbildung 4.2.1. Aus diesem Grund wird das Round Robin Verfahren eingesetzt, wo sich mehrere Tasks eine wichtige Ressource teilen müssen. Ein Beispiel dafür sind Betriebssysteme, welche die CPU, solange noch Daten anstehen, immer zu 100% auslasten.

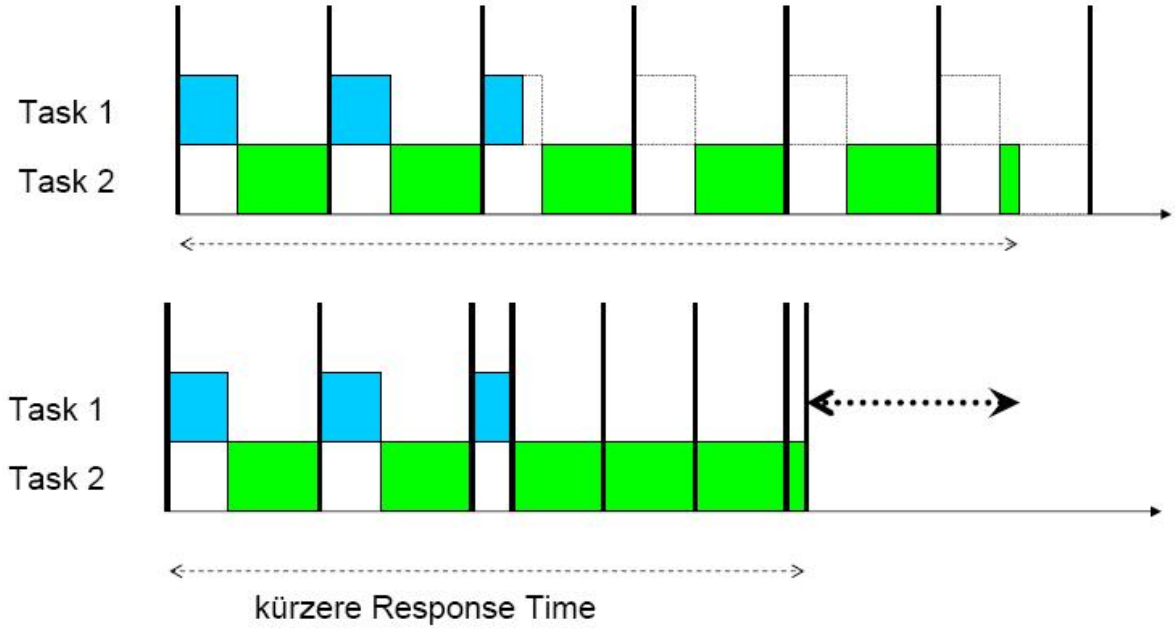

Abbildung 4.2.1: Unterschiedliche Prozessorausnutzung

Genauer gesagt verwenden Betriebssysteme eine Round Robin Variante, welche auch Prioritäten von Prozessen beachten. Gründe für höhere Prioritäten sind zum Beispiel Kernel-Funktionen oder Interrupts.

Die Prozesse, die eine niedrigere Priorität haben, werden erst ausgeführt, wenn diejenigen mit der höheren Priorität beendet sind, siehe Abbildung 4.2.1.
Eine andere Variante wäre, dass Applikationen mit einer niedrigeren Priorität kürzere Zeitslots erhalten, d.h. dass der Prozess mit der Priorität 0 (höchste Priorität) die meiste Zeit zur Verfügung hat. Der Prozess mit der Priorität 1 hätte einen etwas kleineren Zeitslot zur Verfügung, usw.

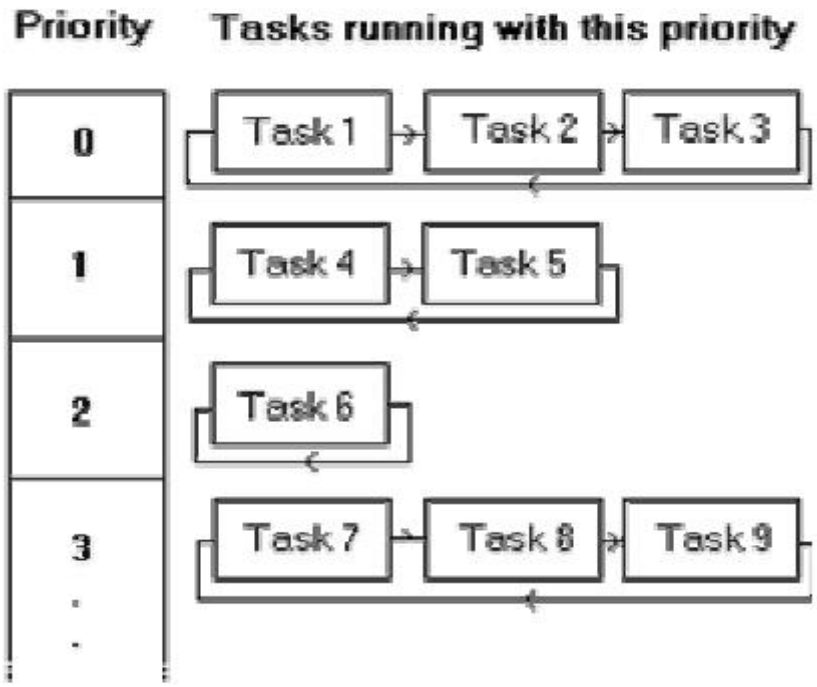

Abbildung 4.2.1: Prioritäten der Prozesse

Die Nachteile von Round Robin Verfahren sind, dass die Ausführung der Tasks von einander abhängig sind. Es muss sich also eine Zentrale, der „scheduler“, um die Verteilung der Slots kümmern und diese regeln. Zusätzlich wird die spätere Worst- Case- Analyse durch die Abhängigkeit komplizierter.

Man muss also bei der Berechnung von Antwortzeit nicht nur die Prozesse beachten, sondern auch, welche Priorität sie haben. Dazu wird zu jeder Prioritätsstufe ein Zeitslot reserviert, in dem alle Applikationen dieser Priorität ablaufen können. Innerhalb dieser Prioritätsstufe werden dann alle Tasks durch Round Robin geschedult. Falls eine Applikation mehrere Threads hat, kann diese Applikation innerhalb ihres Zeitslots die Threads auch schedulen.

Allerdings braucht man das System auf Schedulbarkeit nicht mehr zu prüfen, wenn die CPU-Auslastung unter 100% bleibt. Dies ergibt sich daraus, dass das System zu dem Zeitpunkt kommt, an dem keine Arbeit mehr zu erledigen ist. Falls das nicht der Fall wäre, hätte die CPU eine Auslastung von mehr als 100%. Die notwendige Bedingung, dass ein System schedulbar sein kann, wenn die Tasks zusammen nicht mehr als 100% der CPU beanspruchen, ist also auch ein hinreichendes Kriterium für die Schedulability eines Systems bei Round Robin.

5 Fazit

Das Testen von Software, in diesem Fall für eingebettete Systeme, ist ein sehr interessantes Thema, welches genug Material für eine ganze Vorlesung bereitstellt. Leider war es mir nicht möglich, die nötige Literatur zu finden, die speziell auf das Testen von eingebetteten Systemen eingeht, da viele Hersteller sehr „knauserisch“ bei der Veröffentlichung dieses Materials sind.

Des Weiteren brauch man auch die benötigte Entwicklungsumgebung, wenn man sich mit dem Testen von eingebetteten Systemen beschäftigen möchte, da man sich zumindest auch mal mit dem Testen auf dem Target- System vertraut machen sollte.

Testen von Software ist eine anspruchsvolle Aufgabe, die man nicht von heute auf morgen lernt. Es gehört schon eine Menge an Erfahrung dazu, bis man richtig testen kann.
Ich habe im Laufe dieser Arbeit nur einen kurzen Einblick in die Thematik bekommen und versuchte diesen Einblick weiter zu vermitteln bzw. einen Anreiz zu geben, sich damit zu beschäftigen.

Literaturverzeichnis

[1] Dr. Grünfelder, St.: Den Fehlern auf der Spur Teil 1. Elektronik Ausg. 22, 2004

[2] Dr. Grünfelder, St.: Den Fehlern auf der Spur Teil 2. Elektronik Ausg. 23, 2004

[3] Dr. Grünfelder, St.: Den Fehlern auf der Spur Teil 3. Nicht veröffentlicht

[4] Dr. Grünfelder, St.: Den Fehlern auf der Spur Teil 4. Nicht veröffentlicht

[5] Kigeland, J.: Scheduling-Analysen für Echtzeitsysteme. www.ida.ing.tu-bs-de/academics/seminars/archiv/downloads/ss2003/RoundRobin.pdf

[6] Otto, Th.: Scheduling-Analysen für Echtzeitsysteme. www.ida.ing.tu-bs-de/academics/seminars/archiv/downloads/ss2003/TDMA.pdf