



INSTITUTE FOR REAL-TIME COMPUTER SYSTEMS
TECHNISCHE UNIVERSITÄT MÜNCHEN
PROFESSOR SAMARJIT CHAKRABORTY



**Das ist ein etwas längerer Titel
dieses leeren Diplomarbeitssgerüsts**

Nadja Peters

Master's Thesis

**Das ist ein etwas längerer Titel
dieses leeren Diplomarbeitungsgerüsts**

Master's Thesis

Supervised by the Institute for Real-Time Computer Systems
Technische Universität München
Prof. Dr. sc. Samarjit Chakraborty

Executed bei Siemens

Advisor: Dipl.–Ing. Philipp Kindt

Author: Nadja Peters
Arcisstraße 21
80333 München

Submitted in May 2013

Acknowledgements

Vielen Dank . . .

München, im Monat Jahr

Inhaltsverzeichnis

Abbildungsverzeichnis	vii
Tabellenverzeichnis	ix
Abkürzungsverzeichnis	xi
1 Einleitung	1
1.1 Realzeitbetriebssysteme	1
1.2 Welche Eigenschaften werden verglichen?	1
1.3 Relevante Betriebssysteme	1
1.3.1 Linux	1
1.3.2 FreeRTOS	1
2 Benchmarking	3
2.1 Performanz	3
2.1.1 Latenzzeiten von Interrupts	4
2.2 RT-Features	6
2.2.1 Task Switching	6
2.3 Speicherzugriffe	6
2.3.1 FreeRTOS	7
2.3.2 Verifizierung	8
2.3.3 Multiprozessorunterstützung	9
3 Testszenarien	11
4 Auswertung der Ergebnisse	13
5 Zusammenfassung und Ausblick	15
6 Notizen	17
Literaturverzeichnis	19

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

RCS Realzeit-Computer-Systeme

RTOS Realtime Operation System (Realzeitbetriebssystem)

Zusammenfassung

Die Kurzfassung . . .

1 Einleitung

1.1 Realzeitbetriebssysteme

- Was sind RTOS und was macht sie aus?
- Warum verwendet man sie anstatt herkömmlicher Betriebssysteme oder dedizierter Hardware?
- Verschiedene Arten von RTOS und ihre Vor-/Nachteile (Wofür eignen sich bestimmte Systeme besonders gut, grober Überblick über vorhandenes)
- Beschreiben der Zielhardware/Randbedingungen
- Auskristallisieren, warum in der Arbeit gerade Linux RT Patch und FreeRTOS verwendet werden (evtl noch andere, z.B. MicroCOS, Xenomai)

1.2 Welche Eigenschaften werden verglichen?

1.3 Relevante Betriebssysteme

1.3.1 Linux

1.3.2 FreeRTOS

- Tasks können die gleiche Priorität haben
- Tick rate bestimmt Zeitauflösung → je öfter es aufgerufen wird, desto mehr Zeit wird für das Betriebssystem aufgewendet (Task switches werden dann ausgeführt)
- Normalerweise sollten ISR so kurz wie möglich sein. Deswegen wird ein hoch priorisierter Task aus der ISR aufgerufen, der eine Priorität größer oder gleich dem System-Interrupt hat und wird dadurch nicht durch das System unterbrochen

1 *Einleitung*

2 Benchmarking

Vergleichskriterien für Betriebssysteme

- Performanz
- Sicherheit
- Ressourcen-/Speicherverbrauch
- Speicherverwaltung

2.1 Performanz

1. Latenzzeiten/Jitter

- a) Interrupt durch Taster → Aufblinken von LED (die Zeit, die das Aufblinken benötigt, kann gemessen und abgezogen werden, so dass nur die Zeit vom Drücken des Tasters bis zum Ausführen der ISR bleibt)
- b) Andere Interruptquellen? (→ z.B CAN, Ethernet, SPI, ...)
- c) Verschiedene Taktzeiten von FreeRTOS

2. Durchsatz an Daten

- a) Ethernet
- b) CAN
- c) SPI

3. Bootzeit

- a) Was hat Einwirkungen auf die Bootzeit?
- b) Indikatoren → Wann ist das System hochgefahren?
- c) Bestimmtes Programm wird gestartet → z.B. Aufleuchten von LED
- d) Bestimmte Programme können den Bootvorgang aufzeichnen (Bootchart)

4. Scheduling/Context-Switching

- a) Hardware-Timer → genauer als Software-Timer

2 Benchmarking

- b) In FreeRTOS: Axi-Timer, der im PL instanziiert wird
- c) In Linux: `clock_gettime()` mit `CLOCK_MONOTONIC` (braucht keinen Treiber für Axi-Timer und Zugriffszeit auf Hardware fällt weg)
- d) Beispiel: Start Task1 and Timer → Sleep Task1 → WakeUp Task2 → Timer stop
- e) SPI

2.1.1 Latenzzeiten von Interrupts

Für die Latenzzeit von Interrupts soll ein Interrupt von einer externen Quelle ausgelöst werden und dann wird gemessen, wann die Interruptserviceroutine betreten wird. Konkret wird periodisch ein GPIO-Interrupt durch einen Signalgenerator in Hardware ausgelöst. Die GPIO wird über das EMIO-Interface angebunden. In der dazugehörigen ISR wird eine LED angeschaltet. Es wird die Zeit zwischen dem Setzen des Interruptsignals und aufblinken der LED gemessen. Von dieser Zeit muss abgezogen werden, wie lange das Anschalten der LED und die Zeitmessung an sich dauert. Grundsätzlich wird jede Messung 1024 Mal durchgeführt. Für die Zusatzmessungen sind die Durchschnittszeiten interessant. Für die Hauptmessung ist zusätzlich der Worst-Case-Fall zu beachten.

FreeRTOS

Bei FreeRTOS werden die Interrupts unabhängig vom Betriebssystem verwaltet. Der EMIO-GPIO-Interrupt hat nach dem System-Timer die höchste Priorität.

Axi-Timer: 50 MHz

Ergebnis:

Anz. Messungen	19100
Durchschnittswert	754,8ns
Standardabweichung	8,5977ns (19.31 ns)
Minimalwert	740ns
Maximalwert	780ns
Durchschnittswert LED-Anschalten	700-740ns (130 ns)
Durchschnittswert Timer-Overhead	520ns (9ns)
LED-Overhead bei ISR-Messung	220ns (121 ns)
Durchschnitt - Overhead	633,8ns
Durchschnitt - Overhead (Taktzyklen)	422 Zyklen

mit Task:

Anz. Messungen	19100
Durchschnittswert	754,8ns
Standardabweichung	8,5977ns (19.31 ns)
Minimalwert	719ns
Maximalwert	840ns
Durchschnittswert LED-Anschalten	700-740ns (130 ns)
Durchschnittswert Timer-Overhead	520ns (9ns)
LED-Overhead bei ISR-Messung	220ns (121 ns)
Durchschnitt - Overhead	633,8ns
Durchschnitt - Overhead (Taktzyklen)	422 Zyklen

LinuxRT

Messung mit wmb(): Ergebnis:

Anz. Messungen	19160
Durchschnittswert	13,569us
Standardabweichung	1,4412us
Minimalwert	8us
Maximalwert	28,8us
Durchschnittswert LED-Anschalten	
Durchschnittswert Timer-Overhead	
LED-Overhead bei ISR-Messung	
Durchschnitt - Overhead	s
Durchschnitt - Overhead (Taktzyklen)	

Messung ohne wmb(): Ergebnis:

Anz. Messungen	19130
Durchschnittswert	13,406us
Standardabweichung	1,3419us
Minimalwert	8,6us
Maximalwert	32us
Durchschnittswert LED-Anschalten	
Durchschnittswert Timer-Overhead	
LED-Overhead bei ISR-Messung	
Durchschnitt - Overhead	s
Durchschnitt - Overhead (Taktzyklen)	

Overhead Timer-Messung: 6*Zyklen*

Overhead Registerzuweisung: $19 - 6 = 13\text{Zyklen} = 19\text{ns}$

2.2 RT-Features

1. Welche Unterschiede/Gemeinsamkeiten gibt es zwischen FreeRTOS und Linux?
2. Prioritäten
3. Semaphore
 - a) Task1 setzt Timer und nimmt Semaphor → Schlafen → Task2 lässt Semaphor wieder los → Timer stop
 - b) Task 1 setzt Timer und nimmt Semaphor → Task1 lässt Semaphor wieder los und Timer wird beendet
 - c) Messen der Zeit, die benötigt wird, wenn ein Semaphor belegt ist und ein Task versucht, darauf zuzugreifen?
4. Message Passing (s. Semaphore)
5. Queues (s. Semaphore)
6. Flags (s. Semaphore)
7. Posix-Features in Linux

2.2.1 Task Switching

FreeRTOS

Es werden zwei Tasks *Task1* und *Task2* erzeugt. Diese Tasks haben einen Workload, der darin besteht, in einer For-Schleife eine Variable hochzuzählen. Nach jedem Inkrementieren der Variable wird ein Context-Switch erzwungen (mit `taskYIELD()`). Wenn die Variable eine bestimmte Höhe erreicht hat, wird das Experiment beendet. Es wird dabei die Zeit gemessen, die zwischen dem Betreten des ersten Tasks und dem Verlassen des letzten Tasks vergeht. Ein Task-Switch trifft also zwei Mal so häufig auf, wie die Schleife durchgelaufen wird.

Von der gemessenen Zeit muss noch der eigentliche Workload abgezogen werden. Dafür werden vor dem Starten der Tasks zwei For-Schleifen durchlaufen mit der gleichen Anzahl an Durchgängen wie in den Tasks.

2.3 Speicherzugriffe

1. Speicherplatzverbrauch des gesamten Systems
2. MPU-Unterstützung

3. In welchem Rahmen sind dynamische Speicherzugriffe möglich?
4. Ggf. Zeitverbrauch bei Speicherallokation/-fragmentierung
 - a) Allokation von z.B. 1000 Paketen und Messen der Zeit
 - b) Vergleich von Context Switch mit Speicher Allokation und ohne (?)
 - c) Vergleich von verschiedenen Methoden der Speicherallokation → Was ist der Worst Case, der passieren kann?

2.3.1 FreeRTOS

Es gibt immer eine Mindestfrakturgröße. Außerdem sind die Funktionen Thread-Save, d.h. können durch keinen anderen Task unterbrochen werden. Ausnahmen davon bildet je nach Implementierung Fall 3.

Heap_1.c

Blöcke werden allokiert, wenn genug Speicher da ist und nie wieder freigegeben.

Zugriffszeit	Konstant
Worst Case	Nicht mehr genügend Speicher vorhanden
Schlussfolgerung	Schnell, aber vorher überlegen, ob der Speicher für die Lebensdauer der Anwendung reicht.
Testfall	Einfaches Allozieren, da kein Rechenaufwand durch Freigaben notwendig.

Heap_2.c

Es gibt eine minimale Blockgröße. Es gibt eine Liste, in der die Blöcke nach Größe sortiert sind. Es wird immer der nächst größte Block alloziert → Iteration durch Liste. Kein Verschmelzen von Blocks bei Freigabe. Zu große Blocks werden aufgeteilt. Der neu entstandene Block wird wieder in die Liste einsortiert. Nur sinnvoll, wenn der allozierte Speicher immer in etwa die gleiche Größe hat.

Heap_3.c

Maskierte malloc und free Aufrufe des jeweiligen Compilers.

2 Benchmarking

Zugriffszeit	Am Anfang konstant, weil nur ein Block. Sobald die Liste mehrere Elemente besitzt, ist die Zugriffszeit linear abhängig von der Länge der Liste.
Worst Case	Nicht mehr genügend Speicher vorhanden oder es ist Speicher vorhanden, aber nicht mehr an einem Stück oder es gibt sehr viele kleine Segmente in der Liste und nur ein größeres ganz hinten
Schlussfolgerung	Durch Freigaben langsamer als in Fall eins. Nicht sinnvoll, wenn allozierte Blockgröße variiert.
Testfall	Allozieren von möglichst vielen minimal großen Blöcken und einem, der die doppelte Größe hat. Alle wieder freigeben → Lange Liste mit vielen Einträgen → Nochmal den größeren Block allozieren. Die Zeit für die Längste Freigabe kann auch gemessen werden.

Zugriffszeit	
Worst Case	
Schlussfolgerung	
Testfall	

Heap_4.c

Liste mit Blockzeigern und Blockgröße. Liste wird durchsucht, bis ein passendes Element gefunden wird. Bei Freigabe werden nebeneinander liegende Blöcke wieder zusammengeführt.

Zugriffszeit	Wie in Fall zwei, aber insgesamt schneller, da Blöcke bei der Freigabe wieder zusammengeführt werden und insgesamt tendenziell weniger Blöcke durchiteriert werden müssen.
Worst Case	Wie in Fall zwei
Schlussfolgerung	Flexibelste Alternative, Freigabe ist geringfügig langsamer als in Fall zwei, weil Blöcke noch zusammengeführt werden.
Testfall	Allozieren wie in Fall zwei. Freigabe von jedem zweiten Block, sodass Speicher segmentiert bleibt. Dann nochmal den hintersten Block allozieren.

2.3.2 Verifizierung

- Unter welchen Voraussetzungen ist eine Verifizierung möglich?
- Verifizierung bei einem ganz bestimmten Szenario

2.3.3 Multiprozessorunterstützung

3 Testscenarien

- Verschiedene Auslastung (IO, Tasks, ohne Belastung, Speicherzugriffe) mit unterschiedlichen Ausführungsperioden
- Wie wirkt sich RT-Patch aus im Vergleich zum normalen Linux?
- Ggf. vorhandene Messinstrumente von Linux nutzen (?)
- Wie kann man die Performanz v.a. von Linux verbessern? → Abspecken des Systems

4 Auswertung der Ergebnisse

4 *Auswertung der Ergebnisse*

5 Zusammenfassung und Ausblick

5 Zusammenfassung und Ausblick

6 Notizen

- Alternative Betriebssysteme: microCOS, microLinux, Xenomai
- Multiprozessoren
- Verbesserungen für RT-Linux?
- Allgemeine, anerkannte Benchmarking-Methoden (Algorithmen): RheaStone ist für Realzeit-Betriebssysteme geeignet (Whetstone und Dhrystone eher für verschiedene Hardwarearchitekturen)
- Messen von Bootzeiten mit bootchart
- Linux schneller machen durch Laden der Treiber zur Laufzeit?

Literaturverzeichnis