

Real-Time Architektur Design - Jonglieren mit der Zeit

Newsletter No 21

WILLERT.



Die Main - Loop, in Kombination mit Interrupt-Service - Routinen, hält sich hartnäckig als zentrale Laufzeitsteuerung des Architektur-Design von Embedded Software.

Der „2008 Embedded Market Study“ zufolge, ist die Beherrschung der steigenden Code-Komplexität zurzeit jedoch eine der Hauptanforderungen im Software Engineering. In derselben Studie geben 81% aller Entwickler, die kein RTOS einsetzen an, dass sie keine Notwendigkeit für den Einsatz eines RTOS sehen.

Ist diesen Software Entwicklern bewusst, dass eines der Hauptinstrumente, um Komplexität zu beherrschen, das Architektur-Design ist? Ich befürchte nicht, und aus diesem Grund greifen wir das Thema in dieser Newsletter auf.

Heute stellt sich oftmals das Architektur-Design als mächtigster Mechanismus im Software Engineering dar, um Software Qualitätsattribute wie Wiederverwendbarkeit, Verstehbarkeit, Wartbarkeit, Robustheit, Testbarkeit... zu verbessern. Warum ist das so?

Um diese Frage zu beantworten möchte ich erst einmal das Thema Komplexität beleuchten. Was ist eigentlich Software-Komplexität genau?

Latent steigende Komplexität

Wenn wir die heutige Software hinsichtlich der zu realisierenden Funktionalität betrachten, dann stoßen wir auf mehrere parallele Gesichtspunkte, die zu beachten sind. Ich möchte sie im Folgenden als Ebenen bezeichnen. Grundsätzlich hat jede Software die folgenden 4 Ebenen.

1. Logisches Verhalten
2. Zeitliches Verhalten
3. Datenfluss
4. Priorität zwischen Nebenläufigkeiten

Diese vier Ebenen finden wir in jeder Software und Änderungen in einer Ebene können sich über andere Ebenen in jede Ecke der Software auswirken. Nehmen wir als Beispiel die Main-Loop. Ändern wir das logische Verhalten einer Code-Sequenz innerhalb der Main-Loop, dann ändert sich auch das Zeitverhalten aller Code-Sequenzen. Nun kann es vorkommen, dass obwohl es nur eine kleine Zeitverschiebung gibt, diese gerade ausreichend ist, um die minimale Reaktionszeit einer anderen Sequenz zu überschreiten.

Komplex ist ein System u.a., wenn es verschiedene Ebenen hat und sich Änderungen in einem Teilbereich des Systems über die verschiedenen Ebenen in ganz anderen Bereichen auswirken kann.

Betrachten wir reale Systeme, dann gibt es noch weitere Ebenen, die bei Änderungen,

Erweiterungen oder Wiederverwendung berücksichtigt werden müssen.

So kann es sein, dass die Software verschiedene Betriebszustände kennt. Zum Beispiel neben dem eigentlichen Betriebsmodus einen Service- und Diagnosemodus und noch einen weiteren Modus, um Software Updates einzuspielen.

Dazu kann es sein, dass es mehrere Varianten des Gerätes gibt, zum Beispiel eine preiswerte und eine teure Variante, die dann noch zusätzlichen Komfort bietet. Die Software soll aber möglichst mit einer Version beide Varianten unterstützen.

Hinzu könnte kommen, dass unsere Software zu alten Versionen der Geräte kompatibel sein muss. Inzwischen sind einzelne Hardware-Komponenten in neuen Geräten verändert worden, in den alten Geräten arbeitet aber noch die vorherige Hardware. Die Software soll auf beiden Gerätegenerationen funktionieren.

Wenn wir weiter analysieren würden, täten sich noch eine ganze Reihe weiterer Ebenen auf.

Betrachten wir nun eine einzelne Sourcecode-Zeile in unserer Applikation, dann muss, bezogen auf eine Änderung, immer sichergestellt sein, dass alle Ebenen entsprechend berücksichtigt wurden. Die bedingte Compilation auf Basis der Compiler-Direktiven `#ifdef`, `#ifndef` und `#else` ist zum Beispiel ein oft eingesetztes Hilfsmittel, um einzelne Ebenen im Sourcecode darstellen zu können.

In der Praxis sind heute 7-12 Ebenen dieser Art zu berücksichtigen. Das macht den Hauptteil der Komplexität aus. Die Standardlösung, um Komplexität zu begegnen, ist ein lang bewährter Trick: „Teile und herrsche“. Wird ein komplexes System in einzelne weniger komplexe Teile aufgebrochen, dann lassen sich die Einzelteile sehr viel einfacher managen. Die Aufteilung von Software in einzelne Komponenten wird auch Software Architekturdesign genannt.

Der heute immer noch meisteingesetzte Mechanismus dieses zu tun, ist die Aufteilung in Module und Unterprogramme. Module liefern eine logische Struktur; die darin enthaltenen Unterprogramme werden jedoch häufig wieder zentral in einer `main()` Routine abgearbeitet. Hinsichtlich der Ebenen wirkt sich diese Art von Architekturdesign nicht hilfreich aus.

Wenn sich über die verschiedenen Ebenen Auswirkungen von einem Modul bzw. Unterprogramm zu anderen Unterprogrammen ergeben, funktioniert der Trick „Teile und herrsche“ nicht. Das haben Entwickler erkannt

und im Rahmen der OOP die sogenannte Entkopplung (Encapsulation) eingeführt.

Dieser Begriff wurde dann hauptsächlich in der Programmierung von sogenannter datenzentrierter Software (Datenbanken, Buchungssysteme ...) eingeführt. Hier ging es hauptsächlich darum, Datenkonsistenz zu gewährleisten und einzelne Teile des Systems auf der Datenflussebene untereinander zu entkoppeln.

Wir haben es bei technischen Systemen jedoch überwiegend mit laufzeitorientierter Software zu tun. Da helfen die allgemein bekannten Mittel für Encapsulation nicht viel weiter. Das heißt aber nicht, dass nicht auch in diesem Bereich Encapsulation möglich ist.

Halten wir noch einmal fest: Komplexität entsteht durch die folgenden Entwicklungen:

1. Zunahme der Komplexität des eigentlichen logischen Verhalten
2. Vergrößerung der Anzahl an Nebenläufigkeiten in unseren Systemen (Neben der eigentlichen Aufgabe müssen parallel Display, Bussysteme, Sicherheitsabfragen u.s.w. bedient werden.)
3. Steigende Anzahl an unterschiedlichen Gesichtspunkten (auch Ebenen genannt), denen unsere Software genügen muss.

Auf die Punkte 1 und 2 können wir Entwickler in der Regel nicht einwirken. An dem Punkt 3 haben wir jedoch Möglichkeiten über das Software - Architekturdesign Einfluss auf die Komplexität zu nehmen.

Die einfachste Möglichkeit ist natürlich immer die Vermeidung von Ebenen. Hier 2 Beispiele:

1. Zeit-Ebene:
Ist unsere Software auf Basis einer Main-Loop als Laufzeitarchitektur aufgebaut, dann kann die Zeit-Ebene vernachlässigt werden, wenn die CPU so schnell ist, dass die Reaktionszeiten immer abgesichert sind. Wenn es möglich ist, das Management davon zu überzeugen etwas mehr Geld in die Hardware-Architektur zu investieren, kann die Zeitanforderung vernachlässigt werden.
2. Gerätevarianten:
Heute ist es häufig möglich die Hardware - Basis verschiedener Geräte gleich zu gestalten. Die teure und preiswerte Gerätevariante ist hinsichtlich der

Hardware gleich und damit entfallen Unterschiede in der Software.

Derartige Möglichkeiten sollten bei neuen Geräten immer diskutiert werden. Aber in diesem Artikel möchten wir auch aufzeigen, wie bei gleicher Anzahl an Ebenen über das Software - Architekturdesign eine bessere Entkopplung der einzelnen Ebenen möglich ist.

Encapsulation in laufzeitorientierten Systemen

Wir können Software-Komplexität reduzieren, indem wir uns die Rahmenbedingungen der Architektur so gestalten, dass eine der Ebenen vernachlässigbar wird. Betrachten wir in diesem Zusammenhang noch einmal die Rahmenbedingungen von Embedded Realtime Systemen in Bezug auf die Zeit-Ebene.

Im Idealfall sollte unser System zu jedem Augenblick auf Ereignisse der Außenwelt ohne zeitliche Verzögerung reagieren. Diesen idealen Zustand können wir theoretisch natürlich nicht erreichen, da unsere CPU mit einem zeitlichen Raster reagiert (das theoretisch minimale Zeitraster wäre die Clock-Frequency). Was wir aber erreichen können ist, dass das Zeitraster so klein ist, dass die Reaktionszeiten unserer Applikation hinreichend erfüllt werden.

Ein Beispiel: Die Bildwiederholffrequenz eines Fernsehers beträgt 50 Bilder pro Sekunde. Diese Geschwindigkeit ist ausreichend, um unser Auge zu täuschen und Filme als fließende Bewegung erscheinen zu lassen, anstatt eine Aneinanderreihung von Einzelbildern. Also reicht es, wenn auch unser Mikrocontroller die Ansteuerung des Monitors auf Basis dieser Frequenz durchführt.

Aus dieser Frequenz lassen sich minimale Reaktionszeiten für die horizontale und vertikale Ablenkung errechnen. In diesen Zeiten müssen verschiedene Verarbeitungsschritte durchgeführt werden. Ein sehr einfaches Architektur- Design bestände aus einer Main-Loop, die zyklisch mit der erforderlichen Mindestzykluszeit ausgeführt wird und innerhalb eines Zyklus alle notwendigen Signalverarbeitungen durchführt.

Dieses einfache Design setzt voraus, dass alle notwendigen Signale zeitlich kontinuierlich (zu jedem Augenblick) an der CPU-Peripherie anliegen. Diese werden zyklisch abgefragt, bewertet und bearbeitet.

Es gibt nur eine grundlegende Voraussetzung für dieses Laufzeitdesign: die Verarbeitungsgeschwin-

digkeit der CPU muss ausreichend schnell sein, um die Zykluszeit der Main-Loop zu gewährleisten. In diesem Fall müsste für das Laufzeitarchitektur - Design einmalig die Grenzfrequenz des Systems herausgefunden werden. Nach jeder Änderung des Systems müsste lediglich die Zykluszeit der Main-Loop gemessen werden und solange die Grenzfrequenz des Systems nicht unterschritten wird, kann die Zeit-Ebene vernachlässigt werden.

Wird die Grenzfrequenz überschritten, dann würde einfach (aus SW Engineering-Sicht) eine performantere CPU eingesetzt werden.

Aber genau in diesem Punkt stoßen wir in der Realität irgendwann an die Grenzen unseres Systems. Eine schnellere CPU kann bedeuten, dass die Mikrocontroller-Familie gewechselt und damit verbunden ein Hardware-Redesign durchgeführt werden muss.

Der Aufwand für diesen Schritt ist erheblich. Da kann es einfacher sein in der Software Architektur eine Zeitanalyse durchzuführen, um festzustellen, dass einzelne Arbeitsschritte der Software innerhalb der Main-Loop nicht mit der idealen Zykluszeit durchgeführt werden müssen, sondern eine viel geringere Zykluszeit ausreichend wäre. Hier haben wir Potential für Zeitoptimierung und auf Basis der Änderung der Software lässt sich der aufwändige Schritt eines Hardware-Redesigns verhindern.

Eine konsequente Systemanalyse aus zeitlicher Sicht führt immer zu verschiedenen Aufgaben mit unterschiedlichen Zykluszeiten (z.B. unterschiedliche Periodizitäten zwischen der Zeilen- und Bild- Wiederholffrequenz) plus ereignisorientierten Aufgaben ohne Zykluszeit (z.B. das Umschalten des Fernsehkanals oder die Änderung der Lautstärke, welche keine Zykluszeiten haben). Wird also das Architektur-Design so geändert, dass alle Aufgaben immer nur dann durchgeführt werden, wenn sie, bezogen auf ihre eigene Grenzfrequenz bzw. Reaktionszeit, notwendig sind, dann würden wir aus Laufzeitsicht eine ideale Auslastung der CPU erreichen.

Wenn wir diesen Gedanken aufgreifen ermöglicht sich eine ganz andere Art von Architektur-Design. Wir stellen für jede einzelne Aufgabe der CPU die individuelle Grenzfrequenz bzw. Reaktionszeit in den Vordergrund. Daraus resultiert ein ereignisorientiertes System. Ereignisse können auf der Daten-Ebene betrachtet werden, unsere Zeiten sind nun auf der äußeren Betrachtungsebene in die Daten-Ebene transformiert. Was uns fehlt ist ein Mechanismus, der diese Transformation wieder auf die

Ausführungszeit der CPU zurück transformiert, ohne dass wir uns darum kümmern müssen.

Was würden wir für ein derartiges Architektur - Design benötigen?

1. Änderungen der Außenwelt müssten erkannt werden und ereignisorientiert auf unser System einwirken.
2. Zyklische Anforderungen an unser System müssten ebenfalls in Ereignisse konvertiert werden. Es würde ein übergeordneter Timer benötigt, der auf Basis der jeweiligen Zykluszeit Events generiert.
3. Wir würden ein Laufzeitsystem benötigen, dass zum Einen die auftretenden Ereignisse und zum Anderen die Laufzeit der CPU verwaltet.
4. Nun gibt es noch etwas ganz Wesentliches. Da sich unser Interface zur Außenwelt nicht mehr in Form von Signalen (also zeitlich kontinuierlich), sondern in Form von Ereignissen (Zeitlich diskret) darstellt, muss sich das System von einem Zyklus zum nächsten selbständig merken in welchen Zuständen sich unsere Außenwelt befindet.

Der letzte Punkt ist für viele der gedanklich am schwierigsten zu erfassende. Hier noch einmal die beiden grundsätzlich unterschiedlichen Arten des Designs:

1. Unsere Außenwelt ist in Form von Signalen, also aus zeitlicher Sicht, kontinuierlich angebunden. Dann spielt die Rasterung unserer Ausführungszeit, solange sie nur schnell genug ist, keine Rolle. Wir können die Sensorsignale zu jedem Zeitpunkt abfragen. Ein derart aufgebautes System würde in jeder Zykluszeit jedes Sensorsignal abfragen und bewerten und die äußeren Zustände immer wieder neu erkennen.
2. Die Außenwelt ist ereignisorientiert, also zeitlich diskret angebunden (z.B. über einen Bus). Eine Änderung würde einmalig zu einem bestimmten Zeitpunkt einen Impuls an das System geben. Das System muss auf diesen Impuls reagieren. Der äußere Zustand kann nicht noch einmal abgefragt werden, da nur der Zustandswechsel (eine Änderung in der Außenwelt) auf unser System einwirkt, nicht der Zustand an sich.

Für unser ereignisorientiertes Architektur-Design bedeutet das, dass sich das System die äußeren

Zustände selbständig merken muss. Intern wird also ein Abbild der Zustände entsprechend der aktuellen Situation der Außenwelt gebildet.

Noch ein Beispiel: Unser System kennt zwei Betriebsmodi. Den eigentlichen Betriebsmodus, in dem das Gerät normal bedient werden kann und einen Servicemodus, in dem Grundeinstellungen vorgenommen werden können. Eine Taste des Systems hat entsprechend der beiden Betriebsmodi unterschiedliche Wirkung. Denken Sie an einen Monitor. Dort gibt es den Modus, um Bildbreite, Bildhöhe, Synchronisation u.s.w. einzustellen. Meistens werden dazu die normalen Bedienknöpfe mit einer weiteren Funktion belegt. Um die Einstellungen durchführen zu können muss in einen bestimmten Betriebsmodus gewechselt werden.

Wird nun die Taste betätigt und daraus ein Event im System erzeugt, dann muss das System implizit wissen, ob es sich im normalen Bedienungs- oder Servicemodus befindet.

Hier noch einmal der Vergleich der beiden Architekturansätze:

1. Zeitlich kontinuierliches System (Analoger Charakter), in dem es nur klassische Schalter gibt.
2. Zeitlich diskretes System, in dem es nur Taster gibt.

Bei früheren analogen Systemen hätte es für die Wahl des Betriebsmodus einen klassischen Schalter mit zwei Schalterstellungen gegeben. Dieser Schalter konnte in zeitlich kontinuierlichen Systemen zu jedem Zeitpunkt abgefragt werden. Heute werden in der Regel jedoch zeitlich diskrete Taster eingesetzt. Den Zustand des Systems muss sich das System implizit merken.

In einem zeitgetriebenen Design würden in einer Zeitschleife alle externen Sensoren abgefragt und ausgewertet werden. Nur die Auswertung aller Sensorsignale innerhalb eines Zyklus würde ein gültiges Ergebnis liefern.

In einem ereignisgetriebenen System würde nur auf eintreffende Ereignisse reagiert werden. Welche Aktivität ein Ereignis verursacht hängt von den internen Zuständen des Systems ab.

Was ist der Unterschied beider Design-Ansätze bezogen auf die Komplexität?

Wir haben gesehen, dass die Anzahl der Ebenen, die bei der Entwicklung eines Softwaresystems zu berücksichtigen sind, negativ auf die Komplexität

einwirken. Elimination von Ebenen verbessert demzufolge die Beherrschbarkeit der Komplexität.

Schauen wir uns noch einmal die zwei Basis - Laufzeitarchitektur - Designmöglichkeiten hinsichtlich der Ebenen *Zeit* und *Daten* an.

1. Zeitgetriebener Ansatz z.B. in Form einer Main-Loop
Die Kommunikation der Nebenläufigkeiten untereinander geschieht über globale Variablen und zeitkontinuierlich angebundene Peripherie. Daraus resultiert, dass alle Daten zu jedem Augenblick gültig sind. Es gibt keine Verkopplung zwischen Zeitverhalten und Daten-Ebene, die zu berücksichtigen ist.
Dieser Zustand ändert sich, sobald Unterbrechbarkeit (Preemptives Verhalten) notwendig wird. In diesem Augenblick müssen Datenzugriffe gegenseitig geschützt werden und die Zeit- und Daten-Ebene werden miteinander verknüpft. Dieses ist also ein hinreichend guter Architektur-Ansatz, solange kein preemptives Verhalten notwendig ist.
2. Ereignisgetriebener Ansatz
Die Kommunikation der Nebenläufigkeiten geschieht asynchron. Änderungen der Daten (Ereignisse) treiben die Nebenläufigkeiten. Die Zeit-Ebene ordnet sich der Daten-Ebene unter.
Dieser Mechanismus funktioniert auch bei preemptivem Verhalten und sollte dementsprechend grundsätzlich gewählt werden, wenn preemptives Verhalten notwendig ist oder wenn die Peripherie ereignisorientiert an das System angebunden ist. Z.B. über einen CAN Bus.

Asynchrone Laufzeitarchitekturdesigns ermöglichen die Vernachlässigung der Zeit-Ebene über einen großen Zeitraum hinweg. Sie genügen den beiden folgenden Eigenschaften:

- Transformation der Zeitanforderungen auf die Datenfluss-Ebene. In der Betrachtung des Softwaresystems ist die Zeit-Ebene vernachlässigbar.
- In der Realität gilt die obige Eigenschaft natürlich nur bis zu einer Grenzfrequenz.

Da ereignisgetriebenen Laufzeitarchitekturen jedoch aus zeitlicher Sicht zu optimalen Designs führen ist die Grenzfrequenz sehr viel höher, als bei einem synchronen Laufzeitarchitekturdesign.

Exemplarisch wurde aufgezeigt, wie mit Hilfe eines geeigneten Architekturdesign entweder die Daten oder die Zeit-Ebene vernachlässigbar gestaltet werden kann.

Natürlich gibt es viele weitere Möglichkeiten im Architekturdesign die Beherrschbarkeit von komplexen Softwaresystemen zu verbessern, indem Ebenen untereinander entkoppelt werden.

Zusätzlich können auf Basis einer grafischen Notation (beispielsweise der UML) Ebenen unabhängig voneinander visualisiert werden, um dadurch die Verstehbarkeit zu erhöhen.

Die Basis für den beschriebenen Ansatz ist ein sogenanntes Laufzeitsystem. Das kann selber programmiert oder in Form eines Echtzeit Betriebssystems (RTOS Real Time Operation System) eingekauft werden. Heute gibt es eine große Auswahl an Betriebssystemen, angefangen bei Open Source-Systemen, über Systeme mit Preisen von wenigen tausend Euro, bis hin zu umfangreichen Systemen, die dann entsprechend mehr kosten. Für erste Erfolge reicht ein simples RTOS und bei der heutigen Kostenstruktur lohnt sich oftmals nicht einmal der Gedanke daran es selber programmieren zu wollen. Umgerechnet in die Personalkosten eines Software-Ingenieurs sprechen wir von 10 Manntagen. Der Aufwand, selbst für einen minimalen Scheduling-Algorithmus, liegt weit darüber.

Aber welches RTOS ist das geeignete? Wie Sie diese Entscheidung treffen ist in dem folgenden Artikel beschrieben.

Eine gute Orientierungs- und Entscheidungshilfe erhalten Sie außerdem in RTOS- und UML-Schulungen. Termine finden Sie im Anschluss des Artikels oder unter

<http://www.willert.de/events>

Kriterien zur Auswahl eines RTOS

Am Anfang einer jeden Design-Entscheidung steht in der Regel eine Systemanalyse. Bei der Fragestellung, ob ein RTOS, und wenn ja welches, eingesetzt werden soll, verhält es sich nicht anders. Die Grundlage für die Auswahl eines Betriebssystems basiert auf der Wahl des Architekturdesigns und den Anforderungen der Applikation in dieser Hinsicht.

Wie bereits im Leitartikel dieser Newsletter geschrieben ist das Laufzeitarchitekturdesign eine effiziente Möglichkeit der latent steigenden Komplexität zu begegnen. Um sich für ein geeignetes Architektur-Design zu entscheiden sind folgende Punkte zu beachten:

1. Welche Nebenläufigkeiten müssen in dem System realisiert werden
2. Wie sind die Reaktionszeit bzw. Periodizität jeder einzelnen Nebenläufigkeit
3. Was sind die maximalen Laufzeiten der einzelnen Nebenläufigkeiten
4. Welcher Art ist der überwiegende Charakter der Sensorik und Aktorik

Punkt 1 und 2 sind in der Regel auch schon bei Projektbeginn hinreichend genau analysierbar. Die Laufzeit einzelner Nebenläufigkeiten hingegen lässt sich oft erst zum Zeitpunkt der Implementation genau angeben. Aber das, ist wie wir später sehen werden, nicht ganz so wichtig. Vorerst reichen grobe Schätzungen.

Der Charakter der Aktorik und Sensorik wird zeitkontinuierlich oder ereignisgetrieben unterschieden.

Systemanalyse

1. Schritt: Vorentscheidung zeit- oder ereignisgetriebenes Design

An erster Stelle gilt es herauszufinden, ob der überwiegende Teil der Sensorik in seiner Natur tendenziell zeitkontinuierlichen oder ereignisorientierten Charakter besitzt. Das gibt uns den ersten Hinweis auf unsere innere Laufzeitarchitektur, denn sie sollte grundsätzlich analog zur Art der Anbindung der Peripherie sein. Hierzu eine wichtige Anmerkung: Seit Jahren gibt es einen ständigen Trend hin zu

ereignisorientierter Anbindung der Außenwelt. Im Zweifel würde ich mich also immer für eine ereignisgetriebene Architektur entscheiden. Folgend ein paar grobe Richtlinien.

- Passive Sensoren sind vom Charakter eher zeitkontinuierlich, aktive Sensoren eher ereignisgetrieben
- Über Bussysteme angebundene Sensorik ist vom Charakter her, von wenigen Ausnahmen abgesehen, ereignisgetrieben
- Schalter an einem digitalen I/O Port sind zeitkontinuierlich, Taster an einem Interrupt-Eingang ereignisgetrieben

Von der äußeren Anbindung der Peripherie sollte auch die innere Architektur abgeleitet werden, da es zu einem grundsätzlich harmonischeren Architekturdesign führt. Aber Achtung: Historisch gesehen wurden die meisten Designs zeitlich kontinuierlich realisiert. Aus diesem Grund ist oft auch die Peripherie zeitlich kontinuierlich angebunden, obwohl der inhärente Charakter ereignisgetrieben ist. Noch heute ist es häufig so, dass Sensoren aus historischen Gründen über einen Bus pollend abgefragt werden, obwohl der Sensor ein Signal schicken könnte. Hier gilt es die Systeme von Altlasten zu befreien.

2. Schritt: Analyse der Nebenläufigkeiten

Nun analysieren wir die Nebenläufigkeiten, die unser System erfüllen muss. Dazu bietet es sich an eine kleine Excel-Liste zu erstellen in der die Nebenläufigkeit, deren Zeitanforderung und die geschätzte Laufzeit eingetragen werden.

Wichtige Eigenschaften, die uns helfen sind:

Reaktionszeit bzw. Periodizität: Damit ist die Zeit gemeint, in der eine Nebenläufigkeit regieren muss. Das kann entweder ereignisorientiert sein oder in einem wiederkehrenden Zeitraster, dann wird diese Zeit auch Zykluszeit genannt.

Laufzeit: Das ist die Zeit, die die eigentliche Nebenläufigkeit für ihre eigene Abarbeitung benötigt.

Implementations-Ebene: Hier kennen wir grundsätzlich die Interrupt-Ebene und Programm-Ebene. In diesem Fall wird die Programm-Ebene als RTOS-Ebene bezeichnet.

Nebenläufigkeit	Geforderte Reaktionszeit bzw. Periodizität	Geschätzte Laufzeit der Abarbeitung	Implementations-Ebene	Scheduling Aufschlag
Motor Regelung	100 µs Zykl.	?	?	?
Display Ansteuerung	100 ms Ev.	1 ms	RTOS	10 µs
Tastatur Ansteuerung	100 ms Zykl.	100 µs	RTOS	10 µs
Drucküberwachung Pneumatik	10 ms Ev.	1 ms	RTOS	10 µs
Temperatur Überwachung Motorschutz	100 ms Ev.	100 µs	RTOS	10 µs
FFT Regelabweichung Selbstlernroutine	10 s Ev.	100 ms	RTOS	10 µs
Stromregelung für Pneumatikventile	1 ms Zykl.	?	RTOS	10 µs
...
CAN Sende- und Empfangsroutinen	10 µs Ev.	?	ISR	800 ns
PWM Motor Ansteuerung	100 µs Zykl.	?	ISR	800 ns
Abfrage Digital Input	1 ms Ev.	1 µs	ISR	800 ns
AD Konvertierung	1 µs Zykl.	1 µs	ISR	800 ns
...

Abbildung: Beispiel Systemanalyse der Laufzeitarchitektur

(Für eine erste Systemanalyse reicht eine grobe Abschätzung. Werte, die zu diesem Zeitpunkt auch nicht grob geschätzt werden können sind mit einem ? versehen. Im Verlauf des Projektes können die Zahlen dann mit dem Grad der Erfahrung nachgebessert werden. Routinen, die als HW-Treiber ohnehin auf der Interrupt-Ebene realisiert werden sollen, werden sofort als ISR gekennzeichnet.)

Scheduling Aufschlag: Jede Unterbrechung einer Nebenläufigkeit durch eine andere, beinhaltet einen Laufzeit-Overhead. Die Zeit, die eine Unterbrechung verursacht wird auch als Scheduling - oder Kontext Switch Time bezeichnet. Diese ist bei der Auswahl eines Scheduling-Patterns zu berücksichtigen.

Verschiedene RTOS-Systeme können hier wenige µs benötigen, aber auch einige ms.

Aus dem Beispiel der Laufzeitarchitektur-Analyse (Abbildung) werden nun folgende Systemanforderungen abgeleitet:

1. Gibt es Laufzeiten, die länger sind als die kürzesten Reaktionszeiten. Jede

Überschneidung erfordert Unterbrechbarkeit (Preemption) eines Systems.

In obiger Tabelle erkennen wir, dass die Laufzeit der FFT um den Faktor 1000 mal größer ist, als die Reaktionszeiten z.B. der Motor-Regelung. Wäre das die einzige Zeitüberschneidung, dann könnte man sich überlegen die Motor Regelung als ISR (Interrupt-Service-Routine) auf der Interrupt-Ebene zu implementieren und auf einen Scheduling-Pattern auf der Programm-Ebene zu verzichten.

Aber wir sehen, es gibt noch eine weitere Überschneidung: Die Drucküberwachung und die Displayansteuerung mit den geschätzten Laufzeiten im Bereich von 1 ms überschneiden sich mit der Reaktionszeit der Stromregelung der Pneumatik-Ventile. Das gleiche gilt für die Drucküberwachung Pneumatik.

Erfahrungsgemäß wird es im Verlauf der Implementation bzw. im weiteren Projektverlauf tendenziell zu weiteren Überschneidungen kommen, die in einer Grobanalyse nicht erkannt werden. Ich persönlich würde bereits ab zwei potentiellen Überschneidungen den Einsatz eines preemptiven Scheduling-Pattern in Betracht ziehen.

2. Wird auf Basis der obigen ersten Analyse der Einsatz eines RTOS in Betracht gezogen, dann werden nun die auf der Hardware-Architektur möglichen Schedulingzeiten der verfügbaren Betriebssysteme herangezogen, um die notwendigen Reaktionszeiten abzusichern.

Für ein aktuelles System auf Basis einer ARM 7 Architektur mit dem RTOS embOS z.B., läge die Kontext Switch Time bei ca. 8 μ s. Bei gleichem RTOS auf einem Atmel AVR läge sie bei ca. 50 μ s. Nun gibt es grundsätzlich verschiedene mögliche Ansätze. An dieser Stelle exemplarisch zwei davon:

- A. Aus Software Engineering Gesichtspunkten soll ein RTOS eingesetzt werden. Das hätte dann Auswirkungen auf die Wahl der CPU. In diesem Fall würde der AVR grenzwertig in der Prozessorleistung sein.
- B. Aus HW-Kostengründen steht als CPU bereits der AVR fest, dann würde die Reaktionszeit der Motor-Regelung mit 100 μ s gegenüber der Schedulingzeit von 50 μ s grenzwertig sein. In diesem Fall müsste diese Funktion als ISR implementiert werden. Warum? Nehmen wir einmal die Schedulingzeit von

50 μ s, dann ergibt sich bei einer zyklischen Reaktionszeit von 100 μ s bereits 50 % Belastung der Rechenleistung der CPU, wenn diese Routine auf RTOS-Ebene implementiert werden soll. Eine Faustformel sagt, dass die minimalen Reaktionszeiten Faktor 10 länger sein sollten, als die Schedulingzeit.

Auch die Routinen „Stromregelung für Pneumatik-Ventile“ müssen als ISR implementiert werden. Um ein Gitter in der Reaktionszeit zu vermeiden, muss diese Routine die Drucküberwachung, die Displayansteuerung und die FFT unterbrechen können.

Im Fall B. würden mehr Routinen auf der Interrupt-Ebene implementiert werden müssen, als auf der eigentlichen Programm-Ebene.

3. Wir erkennen, dass der überwiegende Teil der Nebenläufigkeiten ereignisgetriebenen Charakter hat. Die Sensoranbindung hält sich die Waage. In diesem Fall würde ich zu einer ereignisgetriebenen Implementation des Laufzeitarchitekturdesigns tendieren.

Grundsätzlich lassen sich die beiden Welten zeit- und ereignisgetrieben ineinander transformieren. Aber das bedeutet Aufwand. Dieser Aufwand kann leicht vermieden werden, wenn der grundsätzliche Charakter der Laufzeit günstig gewählt wird.

Resümee

Die Zeitanalyse der Nebenläufigkeiten liefert die grundlegenden Informationen für die HW- und SW-Architektur-Entscheidungen. Hier noch einmal die wichtigsten Basisinformationen für eine Architekturentscheidung:

- Überschneidungen von Reaktionszeit und Laufzeit erfordern preemptives Verhalten und sprechen, wenn sie häufiger vorkommen, für den Einsatz eines RTOS
- Reaktionszeiten der Nebenläufigkeiten und Schedulingzeiten des Laufzeitsystems geben Hinweise auf die Implementations-Ebene der Nebenläufigkeiten. Grundsätzlich sollte die Interrupt-Ebene wenigen zeitkritischen Routinen vorbehalten bleiben.
- Die Multiplikation der Reaktionszeiten mit der Summe aus Laufzeit und Schedulingzeit gibt eine grobe Aussage über die notwendige Rechenperformance der HW (In der Praxis können grobe Richtwerte

häufig aus den Erfahrungen vorheriger Projekte abgeleitet werden).

- Die grundsätzlichen Charaktere der Peripherie und der Nebenläufigkeiten geben den Ausschlag für ein grundsätzliches zeit- oder ereignisgetriebenes Laufzeitarchitekturdesign.
- Bei mehr als fünf Nebenläufigkeiten sollte heute der Einsatz eines RTOS immer in Erwägung gezogen werden.

Weitere Informationen zu Scheduling Pattern:

<http://de.wikipedia.org/wiki/Prozess-Scheduler>

Welches RTOS ist das Richtige?

Reaktionszeiten

Sie haben sich dafür entschieden ein RTOS einzusetzen und jetzt gilt es das Richtige auszuwählen. *(An dieser Stelle wird vorausgesetzt, dass das RTOS die Basis-Scheduling-Mechanismen unterstützt. Das ist heute bei nahezu allen am Markt verfügbaren Betriebssystemen der Fall, sollte aber im Zweifel überprüft werden.)*

Aus unserer Systemanalyse der Nebenläufigkeiten kennen wir die geforderte Scheduling-Zeit. Hier trennen sich schon einmal grob die verschiedenen Systeme. Grundsätzlich könnten Real Time Operating Systeme, wie der Name bereits sagt, ihrem Echtzeitverhalten entsprechend zugeordnet werden.

1. Ressourceneffiziente Betriebssysteme mit wenig Diensten und sehr schnellem Zeitverhalten.
2. Betriebssysteme, die viele Dienste und Treiber zur Verfügung stellen, dafür aber auch entsprechende Ressourcen benötigen und entsprechend langsam reagieren.

An dieser Stelle grob exemplarisch die Reaktionszeiten verschiedener Kategorien von Betriebssystemen orientiert an einer ARM7 HW-Architektur:

Schedulingzeiten und Interrupt-Latenzzeiten liegen bei den folgenden kleinen effizienten Betriebssystemen im Bereich von wenigen μ s: ARTX (enthalten in der RT Lib der Firma Keil/ARM), embOS (Fa. Segger), FreeRTOS, (FreeRTOS.ORG), CMX RTX (Fa. CMX Systems), μ C OS (Fa. Micrium)

Schedulingzeiten und Interrupt-Latenzzeiten liegen bei den folgenden komplexeren Betriebssystemen im Bereich von einigen 100 μ s bis hin zu einigen ms: Windows CE (.NET), LINUX, VxWorks.

Dazwischen bietet der Markt eine große Auswahl an Betriebssystemen: EUROS, PXROS, QNX, Enea OSE ... um nur einige zu nennen.

Die Abstimmung der notwendigen Reaktionszeiten in Kombination mit dem RTOS und einer entsprechend performten HW-Architektur ist die Basis einer jeden Laufzeit-Architekturdesign- Entscheidung. Was hilft Ihnen der Einsatz eines RTOS, welches in Kombination mit der gewählten Hardware-Architektur nicht die notwendigen Schedulingzeiten erfüllt und sie dann die Hälfte Ihrer Nebenläufigkeiten auf der Interrupt-Ebene implementieren müssen, wo sie die meisten Dienste eines RTOS nicht in Anspruch nehmen können?

Abdeckung der benötigten Dienste und Funktionen

Heutige Applikationen müssen über Bussysteme kommunizieren, sollen PC-Kompatible auf Speicherkarten lesen und schreiben können, haben komplexe Displaysteuerung, sollen dynamisch Softwareupdates durchführen, Web Services zur Verfügung stellen...

Für viele dieser Anforderungen bieten die unterschiedlichen Betriebssystemhersteller bereits fertige Lösungen. Der Grad der Abdeckung der eigenen Anforderungen an Diensten und Treibern fließt selbstverständlich in den Entscheidungsprozess mit ein. Hier können die komplexeren Betriebssysteme wie Windows CE oder LINUX ihre Karten ausspielen.

Betriebsbewährtheit

Ein weiteres nicht unwichtiges Entscheidungskriterium ist die Betriebsbewährtheit des RTOS in Kombination mit Compiler, IDE und Prozessor - Architektur.

An dieser Stelle gibt es besser oder schlechter bewährte Kombinationen. Muss ein RTOS erst noch für eine spezifische Kombination angepasst werden, dann sollten Sie sich ernsthaft überlegen, ob Sie nicht auf die bereits vorhandene Kombination setzen und gegebenenfalls die Wahl des Compilers überdenken oder ein anderes RTOS in Betracht ziehen.

Sie möchten z.B. die ARM7 Architektur auf Basis des GNU Compilers einsetzen und haben ein RTOS in die engere Wahl gezogen, welches jedoch nur

den Keil/ARM Compiler unterstützt. (In dieser Kombination wird das RTOS schon von zahlreichen Kunden eingesetzt).

In diesem Fall würde ich persönlich immer die Compilerentscheidung der RTOS Auswahl hinten anstellen. Es kann natürlich sein, dass triftige Gründe für einen bestimmten Compiler sprechen. Aber 3.000,- € Ersparnis für den kostenlosen GNU Compiler wären für mich kein triftiger Grund, eher schon die Übernahme eines Projektes, dass auf Basis des GNU Compilers entwickelt wurde.

Weitere Entscheidungskriterien

Sourcecode

Ist das RTOS im Sourcecode verfügbar? Viele Entwickler denken, der Sourcecode wird benötigt, um eigene Änderungen durchführen zu können. Diesen Grund halte ich für nicht so wichtig. Für die Wartung und Pflege ist der RTOS - Hersteller zuständig. Viel interessanter ist der Sourcecode z.B. beim Debuggen. Es kann sehr unangenehm sein, wenn Sie beim Single Stepp durch das System immer wieder im Disassembler landen, wenn Sie in die RTOS - Routinen steppen.

System Level Debugging

Trotz einem guten Architektur-Design und einem betriebsbewährten RTOS wird Ihre Applikation nicht immer sofort fehlerfrei laufen. Hier hilft ein sogenannter System Level Debugger. In ihm können Sie die Ressourcen und Abläufe Ihrer Architektur dynamisch betrachten, um Fehler einfacher zu finden. Ein solcher Debugger lohnt sich auf jeden Fall und die ausgewählte Toolkette sollte einen System Level Debugger enthalten. Es hängt von der Wahl des RTOS und Debuggers ab. In manchen Fällen wird der Debugger mit dem RTOS geliefert, in anderen gibt es eine definierte Schnittstelle zum Debugger.

Preis

Echtzeitbetriebssysteme müssen heute nicht mehr teuer sein. Es gibt viele Systeme, deren Preise im Bereich von wenigen tausend Euro pro Arbeitsplatz liegen. Für diesen Betrag lässt sich nicht einmal die Funktionalität eines simplen RTOS selber entwickeln.

Anders wird es, wenn zu den Entwicklungslizenzen noch sogenannte Vertriebslizenzen (auch Royalties genannt) hinzukommen. Einige der RTOS-Anbieter haben derartige Preismodelle. Dann kann der Einsatz eines RTOS schnell in den Bereich von mehreren zehntausend Euro ansteigen. Auch das

kann rentabel sein, wenn die gelieferte Lösung entsprechende Vorteile bietet. Diese Konzepte kommen aber zunehmend aus der Mode, weil sich häufig Alternativen anbieten.

Open Source Betriebssysteme, wie beispielsweise LINUX, sind oftmals nur auf den ersten Blick kostenlos. Diese Systeme müssen evtl. an eigene Hardware angebunden werden oder es muss ein sogenanntes Board Support Package gekauft werden. Am Ende entstehen auch dort Kosten, die nicht zu vernachlässigen sind.

Abschließende Betrachtung

Im Lauf der Zeit sind die Anforderungen an heutige Applikationen schleichend, aber permanent gewachsen. Nicht unbedingt nur neue Funktionalitäten sind die Ursache für steigende Komplexität, sondern deren Ineinandergreifen.

Entkopplung ist der wirksamste Mechanismus dem zu begegnen. Die Grundlage für ein entkoppeltes Software System liegt im Architekturdesign und hier an erster Stelle das Laufzeit - Architekturdesign. In der Praxis ist aus diesem Grund der Einsatz eines RTOS eine sehr gute Möglichkeit der steigenden Komplexität zu begegnen.

Wenn Sie noch nicht ganz sicher sind, ob und wie auch in Ihrem Fall ein RTOS helfen kann das Architektur Design zu verbessern, dann nehmen Sie sich 1 Tag Zeit und besuchen unseren

Workshop:

Software-Architektur-Design für Embedded Systeme

Termine: **22.09.2009 in Hannover**
23.09.2009 in Ulm

Preis: **199,- € zzgl.gesetzl.Mwst**
incl. Verpflegung

Infos unter: **willert.de/events**

Anmeldungen:**willert.de/anmeldung**

Ein Workshop, den jeder Embedded Software-entwickler besuchen sollte, der ein neues Projekt startet und von Anfang an die Grundlagen für eine langlebige Software-Architektur legen möchte.

Eine Gemeinschaftsaktion der Firmen:

Ein Werkzeug ist nur die eine Seite der Medaille. Die Methodik der Anwendung die andere.

Tools + Schulung = Erfolg

Um Ihnen die Entscheidung, in Ihr Software-Architektur-Design zu investieren, leichter zu machen und einen erfolgreichen Einstieg zu ermöglichen, gibt es bei Willert bis zum Ende des Jahres 2009 zu den Software-Architektur-Produkten und Schulungen besonders günstige Komplettpakete:

- **Schulung**
(RTOS und Software-Architektur)
2 oder 3 Tage
(Inhalte siehe graues Feld -->)

ab 499,- €

- **RTOS + Schulung**
Keil RL - ARM (RTOS)
2 Tage SW-Arch.- u. RTOS - Schulung

3.400,- €

- **Compiler + RTOS + Schulung**
Keil MDK-ARM (Compiler)
Keil RL-ARM (RTOS)
Keil Ulink2 + Eva-Board
3 Tage SW-Arch.- und RTOS-
Schulung incl. Cortex-Architektur

6.750,- €

Bestellung zu o.g. Preisen nur bei :

Willert Software Tools GmbH

rschaak@willert.de

Tel.: 05722 - 9678 60

Embedded Software-Architektur mit dem Keil RTX Real-Time Kernel

Schulung am 02.- 04.11.2009 in Leipzig

Einführung in Keil µVision (KEIL RealView MDK) und dem ARM Cortex-M3

1. Tag:

- ARM Cortex-M3 Core (Architektur, Befehlssatz, Interrupts)
- Unterschiede ARM7, ARM9 und Cortex-M3
- Keil RealView MDK Tool-Chain
- Keil µVision Simulation und Target-Debugging
- Praktische Beispiele mit der Familie STM32 (GPIO, ADC, CAN, UART, Interrupt-Handling)

RTOS KEIL RTX Real-Time Kernel

2. und 3. (halber) Tag:

- Latent „gewachsene“ Software - Grenzen und Probleme
- Embedded-Software-Architekturdesign
- Software - Schichten und Möglichkeiten der Entkopplung
- RTOS Grundlagen
- Scheduling und Kommunikation
- Keil RTX Real Time Kernel
 - Grundlagen
 - Konfiguration und Ressourcen
 - Tasks, Messages und Events
 - Echtzeitanforderungen
- Praktische Beispiele mit dem RTX Real-Time-Kernel

RL- ARM RealView® Real-Time Library

3. (halber) Tag:

- Einführung und Überblick der RL-ARM RTL (TcpNet, FlashFileSystem, CAN, USB)
- Praktisches Beispiel mit der RealView® Real-Time Library (Vorführung)

2.+ 3. Tag 499,-€ / 3 Tage 599,- €

Anmelden unter:

www.willert.de/anmeldung



UMLforum.de

Herausgeber:

Willert Software Tools GmbH

Hannoversche Straße 21

31675 Bückeburg

www.willert.de info@willert.de

Tel. 05722 - 9678 60 Fax 05722 9678 80