Ausgewählte Kapitel eingebetteter Systeme

Java Real-Time Specification

Tobias Distler

28.06.2006

Einleitung

Die Möglichkeit Java bei der Programmierung ihrer Produkte einzusetzen ist ein seit längerer Zeit gehegter Wunsch in der Echtzeitsystem-Branche. Dies zeigt sich besonders dadurch, dass die zugehörige *Java Specification Request (JSR)*, in der die Erweiterung der *Java Language Specification* und der *Java Virtual Machine Specification* für Real-Time-Applikationen gefordert wird, die erste JSR überhaupt war.

Bevor jedoch eine einheitliche Spezifikation erarbeitet werden konnte, galt es, parallele Entwicklungen zu vermeiden: Aus Unzufriedenheit über die Art wie Sun Microsystems versucht hatte, die Entstehung neuer Java-Erweiterungen zu kontrollieren, taten sich im November 1998 u.a. Microsoft, Siemens und Hewlett Packard zur *Java Real Time Working Group* zusammen, die das Ziel verfolgte, ein eigenes, unabhängiges Echtzeit-Java hervor zu bringen. Nicht nur bei Sun führte dies zu der Befürchtung, die Akzeptanz von Java könnte sich durch die Existenz mehrerer, gegenseitig nicht kompatibler Varianten verringern, auch mögliche Kunden, wie z.B. das US-Verteidigungsministerium, versuchten mit Nachdruck eine gemeinschaftliche Entwicklung zu erreichen.[1] Im Januar 1999 einigte man sich schließlich auf die Schaffung der *Real-Time for Java Expert Group (RTJEG)*, die unter der Leitung von Greg Bollella, dessen Firma IBM von Sun mit der Führung der Initiative beauftragt worden war, im Jahr 2000 die *Java Real-Time Specification* [1] veröffentlichte.

Diese Ausarbeitung beschäftigt sich mit den Konzepten, die Einzug in die Spezifikation gefunden haben, und zeigt, wie sie umgesetzt wurden. Dabei werden die Neuerungen mit dem Begriff *Real-Time Specification Java (RTSJ)* bezeichnet, während *Java* die originale Programmiersprache meint, die noch keinen Echtzeitanforderungen genügt.

1. Grundlegende Eigenschaften

Zu Beginn ihrer Arbeit definierte die RTJEG eine Leitlinie, der die Echtzeit-Spezifikation von Java folgen sollte. Dadurch wollte man eine enge Bindung zu Java erreichen und sicherstellen, dass in RTSJ bestimmte Vorteile erhalten bleiben.

Keine Umgebungsbeschränkungen: Nachdem in den Jahren vor der Entwicklung der Java Real-Time Specification eine Reihe von Firmen eigene Lösungen für echtzeitfähiges Java, die meist nur in bestimmten Umgebungen (v.a. Java 2 Micro Edition) funktionierten, herausbrachten, wurde bestimmt, dass RTSJ-Implementierungen überall lauffähig sein müssen.

Rückwärtskompatibilität: Jede RTSJ-Umsetzung muss in der Lage sein, „normale“ Java-Programme, also solche ohne Echtzeit-Anforderungen, korrekt auszuführen.

„Write Once, Run Anywhere“ (WORA): Dieses bereits aus Java bekannte und dort grundlegende Prinzip sagt aus, dass der Bytecode eines Programms portabel sein muss und somit ein Neu-Kompilieren bei der Verwendung auf einem fremden System entfallen kann. Da diese Forderung Einbußen in für Echtzeit-Aufgaben wichtigen Bereichen nach sich ziehen kann, darf allerdings in speziellen Fällen, etwa zugunsten der Vorhersagbarkeit, von WORA abgewichen werden.

Aktuelle und zukünftige Verwendbarkeit: In erster Linie ist RTSJ auf aktuelle Echtzeit-Anforderungen ausgelegt. Darüber hinaus soll ein Hinzufügen neuer Features möglich sein.

Vorhersagbare Ausführung: Jeder Entwickler einer RTSJ-Implementierung ist dazu angehalten, die Vorhersagbarkeit der Programmausführung als oberstes zu erreichendes Ziel ins Auge zu fassen und sich bei auftretenden Tradeoff-Entscheidungen stets gegen Optionen zu entscheiden, die diese gefährden.

[1] JavaWorld, 15. Januar 1999: http://www.javaworld.com/javaworld/jw-02-1999/jw-02-idgns-split.html

Keine syntaktischen Erweiterungen: RTSJ muss vollständig mit den in Java gültigen Schlüsselwörtern auskommen und darf keinerlei neue Syntaxvariationen einführen.

Freiheiten in der Implementierung: Da die Anforderungen für Echtzeit-Systeme in der Praxis mitunter weit auseinander gehen, sind an einigen Stellen der Spezifikation Spielräume für die Implementierung gelassen, die dazu dienen, das Produkt auf die jeweiligen Bedürfnisse eines Kunden anpassen zu können. Beispiele dafür sind die Entscheidung über die Verwendung von Algorithmen nach den Gesichtspunkten des Speicher- und Zeitverbrauchs, sowie die Möglichkeit, ein System durch zusätzliche Schedulingvarianten erweitern zu können.

Die sieben Forderungen an die Spezifikation geben eine klare Richtung vor: RTSJ soll in erster Linie eine Java-Erweiterung darstellen, bei der bestehende Regeln und Regelungen weiterhin ihre Gültigkeit besitzen. Ausnahmen können allein mit einer besseren Erfüllbarkeit von Echtzeit-Anforderungen begründet werden. Dadurch soll garantiert werden, dass sich RTSJ nahtlos in die Java-Welt einfügt.

2. Die Eckpfeiler

Die Real Time Java-Spezifikation besteht im Wesentlichen aus Neuerungen in den sieben Bereichen Scheduling, Memory Management, Synchronisation, asynchrones Eventhandling, asynchroner Kontrolltransfer, asynchrone Thread-Beendigung und Zugriff auf physikalischen Speicher. Die ersten drei Gebiete wurden von der RTJEG selbst als an Echtzeitanforderungen anpassungsbedürftig identifiziert, während die anderen Forderungen und Wünsche aus der Industrie abdecken, die aufgenommen wurden, um RTSJ flexibler zu gestalten.

2.1 Scheduling

Um der Vielzahl der in der Praxis verwendeten Scheduling- und Dispatching-Modellen Rechnung zu tragen, werden in der Spezifikation keine näheren Aussagen über die Eigenschaften möglicher Scheduling-Mechanismen getroffen. Einzig die Charakteristika des Basis-Schedulers, repräsentiert durch die *Scheduler*-Unterklasse *PriorityScheduler* werden vorgeschrieben: Er muss prioritätenbasiert sein (mindestens 28 unterschiedliche Prioritäten) und präemptiv arbeiten. Es ist zu beachten, dass eine strikte Trennung zwischen den Prioritäten vorgeschrieben ist, welche nicht durch weiter unten liegende Systemschichten, die z.B. eine Gruppe benachbarter Prioritäten zusammenlegen, aufgelöst werden darf. Von der Spezifikation folgenden Implementierungen wird erwartet, dass deren Scheduler neben den 28 Realtime-Prioritäten auch eine nicht definierte Anzahl sogenannter „native priorities“ zur Verfügung stellt. Diese werden dazu verwendet, die 10 in Java bereits vorhandenen Prioritäten in RTSJ zu übernehmen, wobei die genau Zuordnung dem Entwickler überlassen wird.

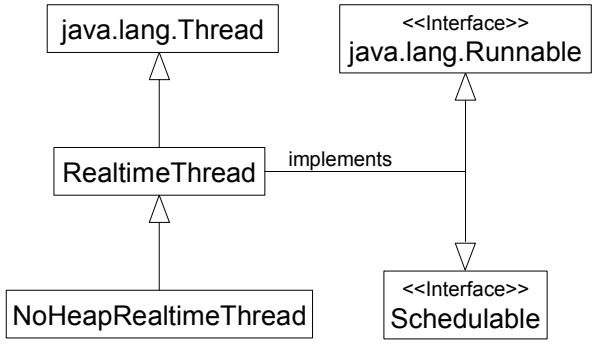

Abbildung 1: Übersicht Thread-Vererbung

Auf der Ebene der Threads ist für Echtzeitsysteme entscheidend, dass deren Ausführung vorhersagbar wird. Dazu wird in RTSJ das Konzept des „schedulebaren Objekts“ eingeführt, das durch die Implementierung des Interfaces *Schedulable* umgesetzt wird. Neben den drei vorgeschriebenen (*RealtimeThread*, *NoHeapRealtimeThread* und *AsyncEventHandler*) können weitere schedulebare Objekte hinzugefügt werden, deren Planung und Einlastung dann ebenfalls vom Scheduler übernommen werden.

RealtimeThread (RTT): Ein RealtimeThread ist für die „gewöhnlichen“ Aufgaben in RTSJ zuständig. Von seiner Oberklasse, einem einfachen Java-Thread (siehe Abbildung 1), unterscheidet er sich durch eine höhere (Echtzeit-) Priorität.

NoHeapRealtimeThread (NHRT): Diese Klasse stellt eine Spezialisierung eines RealtimeThreads dar, die v.a. dazu gedacht ist, zeitkritische Tasks auszuführen. Um eine kurze Antwortzeit zu erreichen, besitzt ein NHRT immer eine höhere Priorität als die Garbage Collection, kann diese also gegebenenfalls unterbrechen. Eine dabei drohende Systeminstabilität wird dadurch verhindert, dass es einem NoHeapRealtimeThread, wie es der Name andeutet, verboten wird, Objekte auf dem Heap zu referenzieren bzw. sogar (anderweitig erzeugte) bestehende Referenzen zu verändern. Abschnitt 2.2 („Speicherverwaltung“) beschreibt mit dem Immortal und dem Scope Memory zwei Speicherarten, die NHRTs im Gegensatz dazu ohne Einschränkung zur Verfügung stehen.

AsyncEventHandler (AEH): für Erläuterungen siehe Abschnitt 2.4 („Asynchrones Eventhandling“)

2.2 Speicherverwaltung

Der Zwang in diesem Bereich für RTSJ Neuerungen einführen zu müssen entsteht aus der Tatsache, dass die klassische Garbage Collection (GC) aus Java wegen zu geringer Vorhersagbarkeit nicht den Echtzeitanforderungen entspricht. Da jedoch nicht grundsätzlich auf die Vorzüge einer automatischen Speicherverwaltung verzichtet werden soll und Algorithmen vorhanden sind, die diese Aufgabe echtzeitkonform bewältigen können, gibt die Spezifikation, ähnlich wie beim Scheduler, eine grobe Richtung vor, die aufzeigt, wie die Auswirkung der GC auf Echtzeit-Threads vorhersehbar wird. Darüber hinaus existiert auch ein Speicherbereich, der von der GC vollkommen unabhängig ist.
Um die neu hinzu kommenden Anforderungen an die Garbage Collection zu bewältigen wird, der Speicher in vier Bereiche aufgeteilt, die alle durch Unterklassen von *MemoryArea* repräsentiert werden:

Heap Memory: Der Heap Memory ist der gewöhnliche Speicher mit Garbage Collection, der durch die Unterklasse *HeapMemory*, ein Singleton, zugänglich ist.

Immortal Memory: Mit der Unterklasse *ImmortalMemory* wird auf von allen Threads gemeinsam verwendete Bereiche zugegriffen, in dem Objekte abgelegt werden können, die während der gesamten Laufzeit der Anwendung verfügbar sein müssen. Er wird von der Garbage Collection nicht überwacht und ist somit auch von NoHeapRealtimeThreads aus verwendbar.

Scoped Memory: Ein Scope ist ein gesonderter Bereich, der explizit oder implizit durch den Start eines Threads, dem er zugewiesen wurde, betreten werden kann. In ihm lassen sich Daten ablegen, deren Lebenszeiten gleich/ähnlich lang sind. Solange Referenzen von außen existieren, diese können z.B. durch den Aufruf der *enter*-Methode von *MemoryArea* erstellt werden, bleiben die Objekte im Inneren erhalten. Sobald sich der Referenzen-Zähler auf 0 reduziert, wird bei allen Objekten im Scope *finalize()* aufgerufen. Er kann erst wieder benutzt werden, wenn dieser Vorgang vollständig abgeschlossen ist.

Die Referenzierung von Objekten in einem Scope ist reglementiert, so darf es z.B. keine Verweise über Scope-Grenzen hinweg geben (es sei denn, die beteiligten Scopes sind in einander verschachtelt), des weiteren sind Referenzen aus dem Heap oder dem Immortal Memory verboten. Die virtuelle Maschine kümmert sich um die Einhaltung dieser Regeln und teilt eventuelle Verstöße durch das Werfen einer Exception mit.
Realisiert wird diese Speicherart mit Hilfe der abstrakten Klasse *ScopedMemory*, die eine Verbindung zu einem bestimmten Speicherabschnitt und dessen aktuellen Zustand darstellt.
Physical Memory: Der Physical Memory bietet direkten Zugriff auf den Speicher. Nähere Erläuterungen finden sich in Abschnitt 2.7 („Physikalischer Speicherzugriff“).

```
import javax.realtime.*;

public class MemoryExample extends RealtimeThread {
    public String immortalString;

    public void run() {
        ImmortalMemory im = ImmortalMemory.instance();
        im.enter(
            new Runnable() {
                public void run() {
                    immortalString = new String
                                     ("Who want's to live forever?");
                }
            }
        ); //enter-Ende
    }

    public static void main(String[] args) {
        MemoryExample me = new MemoryExample();
        me.start();
        System.out.println(me.immortalString);
    }
}
```

Codebeispiel 1: Verwendung von ImmortalMemory

In Codebeispiel 1 ist am Beispiel eines Strings (*immortalString*) gezeigt, wie Objekte im *ImmortalMemory* erschaffen bzw. abgelegt werden können. Zugriff auf diesen nicht teilbaren Speicherbereich erhält man über die *instance()*-Methode.
Eine zentrale Bedeutung beim Umgang mit *MemoryArea*s kommt der Funktion *enter()* zu, der ein Objekt übergeben werden kann, welches das Interface *Runnable* implementiert. Nach dem Aufruf von *enter()* wird die *run()*-Methode des *Runnable*-Objekts gestartet und auf dem zugeordneten Speicherbereich, hier dem *ImmortalMemory*, ausgeführt. Beim Beenden von *enter()* entscheidet die Art der verwendeten *MemoryArea* darüber, was im Weiteren mit den erstellten Objekten passiert. Im Beispiel oben bleiben sie bis zum Ende der Anwendung erhalten, hätte man sich für den Heap als Speicher entschieden, würden sie der Garbage Collection überlassen werden.

2.3 Synchronisation

Bei Systemen in denen verschiedene Threads gemeinsame Ressourcen verwenden und/oder untereinander kommunizieren müssen, ist es erforderlich, sich Gedanken über Synchronisations-Mechanismen zu machen.
In RTSJ kommt dabei das in Echtzeitsystemen sehr verbreitete Priority-Inheritance-Protokoll zum Einsatz, mit dem die Gefahr einer Prioritätenumkehr gebannt werden kann. Darüber

hinaus ist mit dem Priority-Ceiling-Protokoll eine weitere Synchronisationsvariante für Systeme, die dies unterstützen, spezifiziert.
Bereits in Java war mit dem Schlüsselwort „synchronized“ eine Möglichkeit vorhanden, gegenseitigen Ausschluss mit Hilfe von Monitoren zu erreichen. Da die Autoren der RTSJ-Spezifikation dieses Konzept als sehr gelungen ansahen, wurde es auf *RealtimeThread*s und *NoHeapRealtimeThread*s erweitert. Um den Echtzeitanforderungen gerecht zu werden, muss jede RTSJ-konforme Implementierung eine obere Schranke für die Zeit, die benötigt wird einen freien kritischen Abschnitt zu betreten, garantieren.

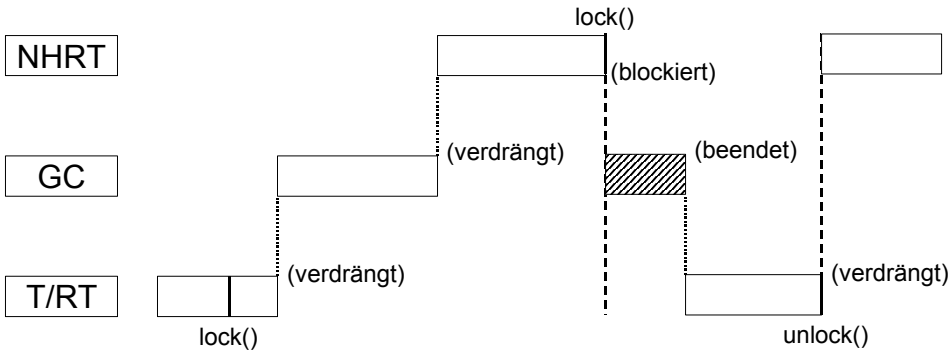

Abbildung 2: Garbage Collection verzögert NoHeapRealtimeThread

Besondere Aufmerksamkeit bei der Betrachtung von Synchronisationsaufgaben wurde auch auf folgende Situation gerichtet (siehe Abbildung 2, [2]): Ein laufender Java-*Thread* (oder *RealtimeThread*), der ein bestimmtes Betriebsmittel b belegt hat, wird durch die Garbage Collection verdrängt, die wiederum (vor ihrer Beendigung!) von einem neu eintreffenden *NoHeapRealtimeThread* verdrängt wird. Der *NHRT* versucht nun, das Betriebsmittel b zu belegen, blockiert dabei jedoch, weil aufgrund des Priority-Inheritance-Protokolls die Ausführung des Java-*Thread*s fortgesetzt werden würde. Da jedoch ein gewöhnlicher *Thread* niemals die GC dominieren darf, besonders nicht wenn diese noch nicht vollständig beendet ist, wird die GC fortgesetzt. Es entsteht somit eine Situation, in der ein *NHRT* auf unbestimmte Zeit (siehe Schraffierung) von der GC aufgehalten wird, ein Sachverhalt, der die Klasse *NoHeapRealtimeThread* ad absurdum führen würde.
Es muss also eine Lösung gefunden werden mit der *NHRT*s und *Thread*s sicher und nichtblockierend auf gemeinsame Ressourcen zugreifen bzw. mit einander kommunizieren können. Dies wird in RTSJ durch die drei Warteschlangen *WaitFreeWriteQueue*, *WaitFreeReadQueue* und *WaitFreeDequeue* gewährleistet.

WaitFreeWriteQueue (WFWQ): Diese Warteschlange bietet einem *NHRT* nicht-blockierendes Schreiben in einen Puffer, der auf der anderen Seite von einem Java-*Thread* ausgelesen werden kann. Die *write()*-Operation blockiert aus zwei Gründen nicht: Erstens wird sie asynchron verwendet und zweitens beendet sie sich bei vollem Puffer sofort (gekennzeichnet durch den Rückgabewert ‚false’). Letzteres hat zur Folge, dass bei mehreren *NHRT*-Schreibern die Abstimmung vom Programmierer übernommen werden muss. Im Gegensatz dazu ist *read()* synchronisiert und blockiert bei leerem Puffer.

WaitFreeReadQueue (WFRQ): Eine *WaitFreeReadQueue* funktioniert analog zu einer *WFWQ*, nur mit umgekehrten Vorzeichen.

WaitFreeDequeue: Werden nichtblockierendes Lesen und Schreiben in beide Richtungen benötigt, kann die Klasse *WaitFreeDequeue* verwendet werden, die eine *WFWQ* mit einer *WFRQ* kombiniert.

2.4 Asynchrones Eventhandling

Die Behandlung von asynchronen Ereignissen, üblicherweise Vorgänge in der realen Welt, auf die das System reagieren soll, funktioniert nach dem Event (hier vertreten durch die Klasse *AsyncEvent*) – Event-Handler (*AsyncEventHandler*) – Prinzip. Tritt ein Ereignis auf, wird dies beim AsyncEvent-Objekt durch den Aufruf von *fire()* repräsentiert. Daraufhin wird dafür gesorgt, dass die *handleAsyncEvent()*-Methode des zugehörigen AsyncEventHandler-Objekts gescheduled wird, die dann die Ereignisbehandlung übernimmt. Dies ist möglich, da die Klasse AsyncEventHandler das Interfaces *Runnable* implementiert und somit einem Thread ähnelt.

AsyncEvent: Mit Hilfe dieser Klasse werden in RTSJ mögliche asynchrone Ereignisse (z.B. das Eintreffen eines Hardware-Interrupts) realisiert. Um auf sie reagieren zu können, teilt man einem AsyncEvent-Objekt ein oder mehrere AsyncEventHandler zu, die beim Aufruf von fire() alle gescheduled werden. In Fällen, in denen die Behandlung zum Zeitpunkt des Eintreffens des nächsten Ereignisses des selben Typs noch nicht abgeschlossen ist, sorgt ein Zähler dafür, dass kein Ereignis verloren geht.

AsyncEventHandler: Ein AsyncEventHandler lässt sich mit einem die CPU und den Speicher nur gering belastenden Thread vergleichen. Beim Feuern eines AsyncEvents wird die *run()*-Methode aktiviert und der Handler zur Ausführung an einen Kontext gebunden, wobei nicht vorgeschrieben ist, dass jeder AsyncEventHandler einen separaten Kontext verwenden muss. Diese mögliche Zusammenfassung der Handler resultiert aus dem Wunsch, RTSJ auf Echtzeitsysteme mit vielen (1000 oder sogar 10000) verschiedenen Ereignissen auszulegen. Sieht ein Programmierer es als erforderlich an, einen AsyncEventHandler genau einem bestimmten Kontext zuzuordnen, steht ihm die Unterklasse *BoundAsyncEventHandler* zur Verfügung.

2.5 Asynchroner Kontrolltransfer

Asynchrone Kontrolltransfers (AKTs) werden i.A. immer dann vollzogen, wenn besondere Vorkommnisse es erfordern, dass ein anderer als der aktuell laufende Thread zur Ausführung kommt. In der Praxis kommen sie außerdem bei der Verwendung von iterativen Algorithmen zum Tragen, bei denen nicht exakt vorhersagbar ist, nach welchem Durchlauf das Ergebnis eine geforderte Genauigkeit haben wird, weil sie ein gute Möglichkeit darstellen, nach Ablauf der maximal vorgesehenen Zeit einfach von der Berechnung auf die Ausgabe umzuschalten.
Die Umsetzung von asynchronen Kontrolltransfers in RTSJ lässt sich durch folgende Merkmale bzw. Forderungen charakterisieren:

- Um Code zu schützen, der ohne Rücksichtnahme auf AKT geschrieben wurde, muss ein Bereich explizit angeben, dass er darauf ausgelegt ist. Die Kennzeichnung erfolgt durch den Anhang „throws AsynchronouslyInterruptedException“ an die Methodensignatur. Des weiteren fordert die Spezifikation eine Möglichkeit, AKT in begrenzten Bereichen, z.B. zur Konsistenzsicherung abzuschalten.
- AKT kann entweder direkt, aus einem anderen Thread heraus, oder indirekt, durch einen asynchronen Eventhandler, an- bzw. abgeschalten werden.
- Durch AKT ist es möglich, einen Thread sicher abzubrechen.
- Eine Implementierung muss bei geschachtelten AKTs auf eine korrekte Behandlung der Ereignisse achten.
- Es sollten Konstrukte für oft auftretende Fälle wie Timer-Behandlung oder Thread-Beendigung vorhanden sein.

- AKT darf bei Programmen, die es nicht benötigen, keinen Overhead erzeugen.
- Sollte der Programmcode bereits vor seiner Deadline komplett ausgeführt worden sein, müssen der Abbruch der Timeout-Berechnung und die Ressourcen-Rückgabe automatisch erfolgen.

2.6 Asynchrone Thread-Beendigung

Soll ein Echtzeitsystem auf eine große Menge verschiedener Zustände ausgelegt sein, gibt es i.A. zwei Herangehensweisen: Entweder es wird dafür gesorgt, dass ein paar wenige Threads existieren, die aufgrund hoher Komplexität viele Zustände bedienen können, oder man verwendet mehrere, dafür einfacher strukturierte, Ablaufstränge. Die Umsetzung der zweiten, vom Design her deutlich eleganteren, Variante wird durch die Existenz eines Mechanismus zur asynchronen Beendigung von Threads erleichtert. Da die in Java vorhandenen Methoden *stop()* und *destroy()* diese Aufgabe allerdings nicht zuverlässig bewältigen können, werden in RTSJ zu diesem Zweck Konzepte des Asynchronen Eventhandlings und des Asynchronen Kontrolltransfers (AKT) kombiniert: Mit Hilfe von Event-Handlern wird auf Ereignisse gewartet, die einen Zustandswechsel bedingen, der dann über AKT abgewickelt wird.
Darüber hinaus existiert auch die Möglichkeit einen Thread unter Zuhilfenahme einer *AsynchronouslyInterruptedException*, die auch bei AKTs zum Einsatz kommt wird, sicher zu beenden: Ruft das System bei einem Thread die *interrupt()*-Methode auf, werden zuerst alle seine nichtunterbrechbaren Abschnitte komplett ausgeführt, bevor es schließlich zu seinem endgültigen Abbruch kommt.
Der Vorteil der RTSJ-Varianten gegenüber Java besteht darin, dass weder Objekte im Speicher zurückgelassen werden (vgl. *stop()*), noch die Gefahr des Auftretens von Deadlocks (vgl. *destroy()*) besteht.

2.7 Physikalischer Speicherzugriff

Da es in der Praxis bei vielen Echtzeitanwendungen von Vorteil ist, direkten Zugriff auf den Speicher zu besitzen, stellt RTSJ Klassen bereit, mit denen u.a. auch die Erstellung von Objekten auf physikalischer Ebene ermöglicht wird. Die große Diversifizierung ist dazu gedacht, unterschiedliche Eigenschaften von Speicherbereichen, wie etwa eine kurze Zugriffsdauer bei statischem RAM, ausnutzen zu können.

RawMemoryAccess: Mit einer Instanz der Klasse *RawMemoryAccess* erhält man direkten Zugriff auf den Speicher, der durch einen Offset zur Basis adressiert wird und mit *get*()- und *set*()-Methoden in den Granularitäten Byte, Word, Long oder Arrays dieser Typen ausgelesen bzw. beschrieben werden kann. Aus Sicherheitsgründen ist es untersagt, Referenzen auf ein Java-Objekt in diesem Speicherbereich abzulegen, da so die Möglichkeit bestünde, die Typ-Überprüfung zu überlisten. *RawMemoryAccess* gestattet Entwicklern somit in Java geschriebene Treiber, Memory-mapped I/O, Flash-Speicher, Batterie-gestützten RAM, etc. in ihren Echtzeitsystemen zu verwenden.

ImmortalPhysicalMemory: Der Einsatz von *ImmortalPhysicalMemory* kann analog zur Verwendung von *ImmortalMemory* (vgl. Abschnitt 2.2 „Speicherverwaltung“) erfolgen, da der einzige Unterschied im Speicherort der Objekte besteht.

VTPhysicalMemory, LTPhysicalMemory: Der Beginn der Klassennamen „VT“ und „LT“ sagt aus, dass die Allokation von neuen Objekten in diesen Speicherbereichen in „variable time“ bzw. „linear time“ erfolgt. Ein weiterer Unterschied besteht darin, dass bei *LTPhysicalMemory* die Speichergröße einen festen Wert besitzt, während sie bei *VTPhysicalMemory* von einem Initial- zu einem Maximalwert wachsen kann.

3. Weitere Klassen

Neben den großen Neuerung werden in der Java Real-Time Spezifikation weitere Klassen beschrieben, die Programmierer unterstützen sollen. Im Folgenden werden Beispiele aus drei Bereichen vorgestellt.

3.1 Parameter-Objekte

Die Vorgehensweise bei der Übergabe von Parametern an Klassen-Konstruktoren unterscheidet sich in RTSJ deutlich gegenüber der in Java: Anstatt Werte oder Referenzen direkt zu übergeben werden sie in Parameter-Objekten gekapselt, so dass sich die Anzahl der zu übergebenden Referenzen verringert.
Besondere Anwendung findet dieses Prinzip beim Interface *Schedulable*, mit dem sämtliche Charakteristika eines Threads verwaltet werden können. Es ist zu beachten, dass sich durch die „by reference"-Übergabe der Parameter nachträgliche Änderungen an einem Parameter-Objekt auf das Verhalten eines Threads auswirken, was v.a. dann zu Komplikationen führen kann, wenn mehrere Threads an das selbe Parameter-Objekt gebunden wurden.
Um den unterschiedlichen Bedeutungen von Parametern Rechnung zu tragen, wurden für *Schedulable* u.a. folgende zwei Klassen definiert, die beide unterschiedliche Eigenschaften kennzeichnen:

SchedulingParameters: *SchedulingParameters* ist eine abstrakte Klasse, die die beiden vordefinierten Unterklassen *PriorityParameters* und *ImportanceParameters* beinhaltet und Informationen für den Scheduler bereit stellt. Während sich mit einem *PriorityParameters*-Objekt einem Thread eine Priorität zuweisen lässt, ist *ImportanceParameters* dafür gedacht, die Ausführungsreihenfolge bei Prioritätengleichheit zu regeln. Letzteres stellt eine Möglichkeit dar, auf einer etwas feingranulareren Ebene Einfluss zunehmen, die der Basis-Scheduler anbietet kann, jedoch nicht muss.

ReleaseParameters: Mit Hilfe dieser Klasse lassen sich die Ausführungskosten sowie die Deadline von Threads festlegen. Um jeweils auf eine Nichteinhaltung der Werte reagieren zu können, besteht die Möglichkeit, dem Konstruktor zwei Referenzen auf AsyncEventHandler zu übergeben.
Die Unterklasse *PeriodicParameters* dient der Realisierung periodischer Tasks, wogegen *AperiodicParameters* für schedulebare Objekte gedacht ist, die jederzeit laufbereit werden können. Als weitere Spezialisierung der aperiodischen Threads existiert darüber hinaus eine Parameter-Klasse (*SporadicParameters*) für sporadische Aufgaben.

Codebeispiel 2 demonstriert den Einsatz von Parameter-Objekten am Beispiel von *Priority*- und *PeriodicParameter* bei der Verwendung eines *RealtimeThread*s. Zuerst wird die Priorität, in diesem Fall die niedrigste, die der Scheduler anbietet, bestimmt, mit der der Thread später laufen soll. Anschließend wird seine Periode auf 20 Millisekunden (und 0 Nanosekunden) festgelegt und in einem *PeriodicParameter*-Objekt gekapselt. Bei der Erstellung des Threads werden schließlich seinem Konstuktor alle für ihn bestimmten Parameter-Objekte übergeben und der RealtimeThread gestartet. Während seiner Ausführung gibt der Thread die Zeile „`Und nochmal...`" auf dem Bildschirm aus und wartet anschließend in der Methode *waitForNextPeriod()* das Ende seiner Periode ab.

```
import javax.realtime.*;

public class ParameterObjectsExample {
    public static void main(String[] args) {
        int prio = PriorityScheduler.instance().getMinPriority();
        PriorityParameters prioPara = new PriorityParameters(prio);

        RelativeTime period = new RelativeTime(20, 0);

        PeriodicParameters perioPara =
                        new PeriodicParameters(null, period, null,
                                               null, null, null);

        RealtimeThread rt = new RealtimeThread(prioPara, perioPara) {
            public void run() {
                 while (waitForNextPeriod()) {
                     System.out.println("Und nochmal...");
                 }
            }
        };

        rt.start();
    }
}
```

Codebeispiel 2: Verwendung von Parameter-Objekten

3.2 Zeitdarstellungen

Für die korrekte Funktionsweise eines Echtzeitsystems ist eine präzise Zeitmessung bzw. die Möglichkeit absolute und relative Zeiten genau ausdrücken zu können unerlässlich. In RSTJ stehen dafür die Klassen *AbsoluteTime*, *RelativeTime* und *RationalTime* zur Verfügung, die alle in der abstrakten Oberklasse *HighResolutionTime* zusammengefasst sind und, soweit das zugrundeliegende System dies unterstützt, eine Genauigkeit im Nanosekundenbereich bieten.

AbsoluteTime: Repräsentation eines Zeitpunkts durch die vergangenen Milli- und Nanosekunden seit dem 1. Januar 1970, 0:00 Uhr.

RelativeTime: Darstellung einer Zeitspanne zwischen einem beliebigen Zeitpunkt und dem Jetzt. (Für Verwendungszweck siehe Codebeispiel 2)

RationalTime: Diese Zeitdarstellung bietet eine Hilfe beim Einsatz von Periodizitäten, da sie die Häufigkeit h eines Ereignisses in einem bestimmten Zeitintervall z beschreibt. In Verbindung mit einem Timer bedeutet dies z.B., dass der Timer h-mal während z feuert. Obwohl in der Spezifikation nicht ausdrücklich vorgeschrieben, wird empfohlen, die h Ereignisse uniform auf z zu verteilen, um Missverständnisse zu vermeiden.

3.3 Clock und Timer

Timer in RTSJ sind Spezialformen von AsyncEvents und existieren in den zwei Varianten *PeriodicTimer*, für periodische, und *OneShotTimer*, für einmalige Ereignisse. Beides sind Unterklassen von *Timer*, die mit *Clock*, einer abstrakten Klasse, die die System-Echtzeit-Uhr repräsentiert, arbeiten.

Clock: Durch den Einsatz einer abstrakten Klasse an dieser Stelle soll die Verwendung mehrerer Uhren, die z.B. mit unterschiedlichen Auflösungen arbeiten, ermöglicht werden. Dadurch verspricht man sich eine Reduzierung des Overheads der entsteht, wenn eine Task mit gröberer den selben Timer wie eine Task mit feinerer Zeitanforderung benutzen muss.

Timer: Jeder Timer bleibt nach seiner Erstellung solange inaktiv bis seine *start()*-Methode aufgerufen wird. Es besteht danach die Möglichkeit ihn wieder zu disablen, was nicht dazu führt, dass er zu zählen aufhört, sondern nur ein Feuern der Ereignisse verhindert. Dies kann wiederum mit *enable()* rückgängig gemacht werden, wobei beachtet werden muss, dass evtl. in der disabled-Phase aufgetretene Ereignisse nicht gequeued werden, also verloren sind.

OneShotTimer: Diese Timer-Art wird für einmalig auftretenden Ereignisse verwendet, die auch dann angezeigt werden, wenn der Timer aktuell disabled ist. Obwohl auf den ersten Blick inkonsequent, wurde hier der Bedeutung von One-Shot-Timern bei der Timeout-Behandlung Rechnung getragen, bei der es darauf ankommt, dass der Ablauf einer Frist auch wirklich gemeldet wird.

PeriodicTimer: PeriodicTimer finden bei regelmäßig in selben Abständen stattfindenden Ereignissen Anwendung. Dabei wird die Konstanz der Periodenlänge höher eingestuft als die Exaktheit des Beginns der Zählung, was dazu führt, dass der Einsatz eines relativen Startzeitpunkts 0 erlaubt wird, der sich geringfügig vom Zeitpunkt des start()-Aufrufs unterscheiden darf.

Fazit

Die in der Java Real-Time Specification enthaltenen Vorgaben stellen einen wichtigen Schritt in dem Bestreben dar, die Programmiersprache Java als Option bei der Programmierung im Echtzeitbereich zur Verfügung zu stellen. Gerade die Neuerungen in den Bereichen ‚Behandlung von sporadischen und periodischen Tasks', ‚Verhinderung von Prioritätsinversionen' und ‚sichere Thread-Beendigung' haben entscheidende Lücken in Java geschlossen. Im Besonderen gilt dies natürlich auch für die Einführung des NoHeapRealtimeThread-Konzepts, mit Hilfe dessen endlich die ominöse GarbageCollection beherrschbar wird. Es kann somit festgestellt werden, dass RTSJ es einem Programmierer, der die Vorzüge von Java zu schätzen weiß, ermöglicht, diese auch bei der Entwicklung von Systemen mit Echtzeit-Anforderungen nutzen zu können.
Betrachtet man jedoch aktuell die Welt der Echtzeitsysteme, so muss man feststellen, dass die Java Real-Time Specification bisher nicht den erwünschten durchschlagenden Erfolg brachte. Dies verwundert besonders, wenn man bedenkt, dass ihre Fertigstellung, von der Branche damals ungeduldig erwartet, mehr als fünf Jahre zurück liegt. Zieht man in Betracht, dass mit JTime von TimeSys erst 2003 die erste vollständig spezifikationstreue Implementierung auf den Markt gekommen ist, wird verständlich, warum einige Firmen (z.B Esmertec, aJile) sich dazu entschieden haben, ihre eigenen Lösungen im Java-Echtzeitbereich zu entwickeln, die dann eben nicht hundertprozentig kompatibel sind.[2] Auch die Tatsache, dass es bei Sun selbst bis 2005 gedauert hat bis mit Java Real-Time System 1.0 die erste eigene Plattform veröffentlicht wurde, lässt vermuten, dass ein Triumphzug bisher v.a. deshalb nicht angetreten werden konnte, weil man einfach zu langsam war.

[2] Király András: „Real Time Specification for Java (RTSJ)", 2001, Uni Zürich
www.ifi.unizh.ch/~riedl/lectures/rtsj.pdf

Quellen

[1] The Real-Time for Java Expert Group: "The Real-Time Specification for Java"
Addison Wesley Longman, 1st edition (January 15, 2000)

[2] Greg Bollella, James Gosling: "The Real-Time Specification for Java"
IEEE Computing June 2000 (Vol. 33, No. 6) pp 47-54

[3] Frank Christoph Köther: "Real Time Java: Ein Überblick", 2004, Uni Siegen
http://pi.informatik.uni-siegen.de/niere/lehre/SS04/SeminarFinal/10_koether/Ausarbeitung.pdf