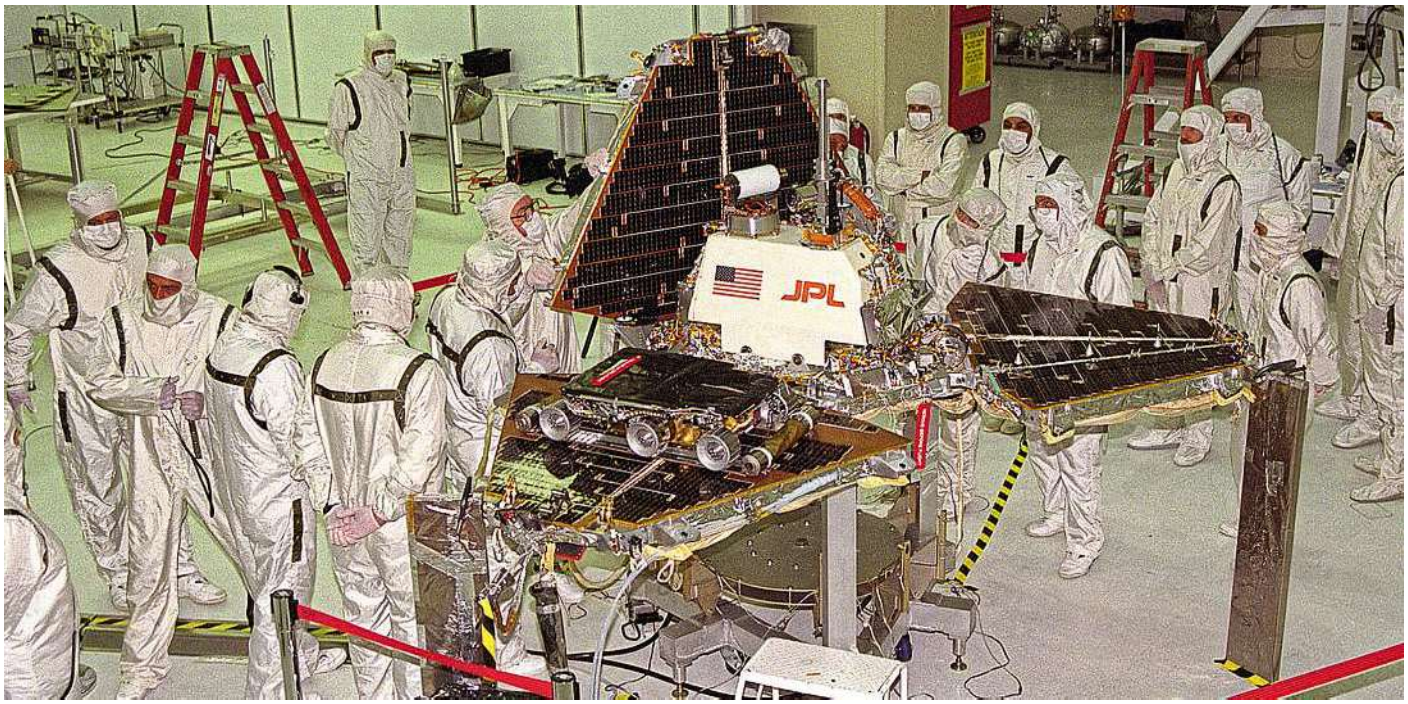

Bild: NASA

Neustart: *Ein Prioritätsinversions-Problem trat auch bei der Pathfinder-Marsmission der NASA auf. Der Fehler konnte zum Glück durch Software-Experten behoben werden.*

Funktionale Sicherheit von RTOS-Applikationen gewährleisten

Ein Echtzeit-Betriebssystem mit Multi-Threading kann ein Embedded-System effizienter machen. Dabei müssen mehrere Aspekte beachtet werden, sonst droht Chaos beim Verarbeiten der einzelnen Tasks.

DR. JOHAN KRAFT *

Systementwickler setzen bei Embedded-Designs immer häufiger auf ein Echtzeit-Betriebssystem (Real-Time Operating System, RTOS). Angesichts der zunehmenden Verarbeitungsleistung der Mikrocontroller lässt sich ein RTOS mit immer weniger Aufwand verarbeiten, während sich gleichzeitig mehrere Vorteile ergeben.

Da ein RTOS das Multi-Threading unterstützt, kann das System modularer aufgebaut und aus mehreren kleineren Programmen zusammengesetzt werden, die jeweils für eine bestimmte Funktion (USB, Display, TCP/IP, Bluetooth usw.) zuständig sind. Der Code ist dadurch einfacher zu verstehen und zu pflegen und auch die Effizienz verbessert sich erheblich – eine korrekte Umsetzung vorausgesetzt. Für das Design, das Debugging und das Testen der Software bringt das Multithreading jedoch eine Reihe neuer Herausforderungen mit sich, die es erforderlich machen, anders an die Softwareentwicklung heranzugehen. Bekannt sind diese potenziellen Probleme beispielsweise unter den Schlagworten Prioritätsinversion, Speicherinterferenz, Wettlaufsituationen (Race Conditions), Deadlocks, Live Locks und Verhungern (Starvation).

Das grundsätzliche Problem besteht darin, dass der statische Quellcode kein umfassendes Bild des dynamischen Verhaltens zur Laufzeit liefert, denn letzteres wird bei Multithreaded-Systemen von Wechselwirkungen zwischen den Tasks und von Timing-Variationen beeinflusst.

Bestimmte, aus dem Quellcode nicht ersichtliche Eigenschaften treten deshalb nur während der Ausführung zutage. Erfassen lassen sich solche Probleme mit ausgefeilten Trace-Tools, die sowohl die Arbeitsweise der Tasks selbst als auch ihre Interaktionen auf einer Zeitachse verdeutlichen und damit aufzeigen, wo sich Tasks gegenseitig blockieren und wo sie Ressourcen mit Beschlag belegen.

Auch wenn die Tasks augenscheinlich nicht voneinander abhängig sind, müssen sie doch möglicherweise auf globale Ressourcen, wie etwa Datenstrukturen oder Hardware-Schnittstellen, zugreifen, sodass es auf die Reihenfolge der globalen Ereignisse ankommt. Wenn die Tasks unterschiedlich

Dr. Johan Kraft
... ist Gründer und CEO des Tool-Entwicklers Percepio.

Bild: Percepio

```
void InitDesign(void)
{
    /* User event channel, used for logging CounterA */
    chA = xTraceRegisterString("Counter A");

    /* User event channel, used for logging CounterB */
    chB = xTraceRegisterString("Counter B");

    xTaskCreate(TaskA_Worker, "TaskA_Worker", 1000, NULL, 1, NULL);

    xTaskCreate(TaskB_Periodic, "TaskB_Periodic", 1000, NULL, 1, NULL);
}

void TaskA_Worker(void *argument)
{
  for(;;)
  {
      DoSomethingA(); /* Incrementing and logging Counter A */
  }
}

void TaskB_Periodic(void *argument)
{
  for(;;)
  {
      if (CheckSomething())
      {
          DoSomethingB(); /* Incrementing and logging Counter B */
      }
      HAL_Delay(10);
  }
}
```

Bild 1: *Einfaches Codebeispiel mit zwei RTOS-Tasks.*

auf derartige Ressourcen zugreifen, kann es zu Wettlaufsituationen und sporadischen Fehlern kommen, die sich im Quellcode nur sehr schwierig erkennen lassen. Mit einem Trace-Tool dagegen ist es möglich, Dinge, wie etwa Race Conditions, Deadlocks und Prioritätsprobleme zu identifizieren.

Portierung von Single-Loop-Design auf RTOS

Illustriert werden soll dies an einem recht einfachen Beispiel mit zwei RTOS-Tasks (Bild 1). Der Code wurde vielleicht zunächst als Single-Loop-Design konzipiert und erst später auf ein RTOS portiert, ohne dass dabei aber die Vorteile eines RTOS-Kernels vollständig erfasst wurden. Was man aus dem Code sofort entnehmen kann, ist die Tatsache, dass beide Tasks mit derselben Scheduling-Priorität laufen. Sind also beide gleichzeitig aktiv, werden sie mit der Auflösung des Tick-Interrupts des Betriebssystems abwechselnd ausgeführt. Man erkennt ferner die Verzögerungsfunktion in der Task „TaskB_Periodic“, die offensichtlich nur alle 10 ms laufen soll.

In Bild 2 ist eine visuelle Trace-Darstellung des Laufzeitverhaltens dieses Codes in Percepio Tracealyzer zu sehen. Links befindet sich ein Scheduling-Trace mit vertikaler, nach unten fortschreitender Zeitachse. Das CPU-Auslastungsdiagramm rechts oben gibt dagegen Auskunft darüber, wie sich die Prozessorzeit auf die Tasks verteilt. Bei diesem horizontal ausgerichteten Diagramm verläuft die Zeitachse von links nach rechts.

Wie man sieht, läuft TaskB_Periodic während 50 % der Zeit. Dies ist verwunderlich, denn eigentlich sollte diese Tasks ja nur alle 10 ms laufen, weshalb die Verzögerung in den Code eingebaut worden war. Die Erklärung hierfür ist, dass die verwendete Verzö-

Bild: Percepio

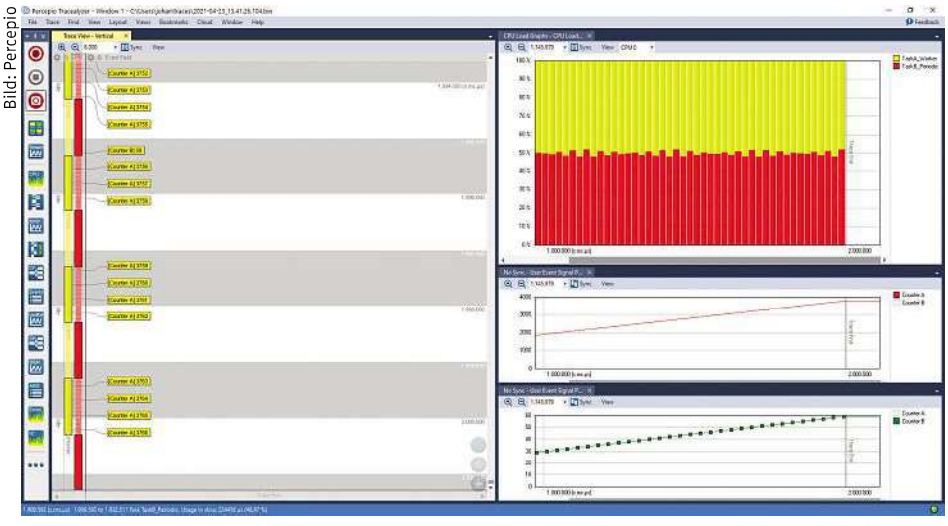

Bild 2: *Einfache Trace-Darstellung zweier RTOS-Tasks.*

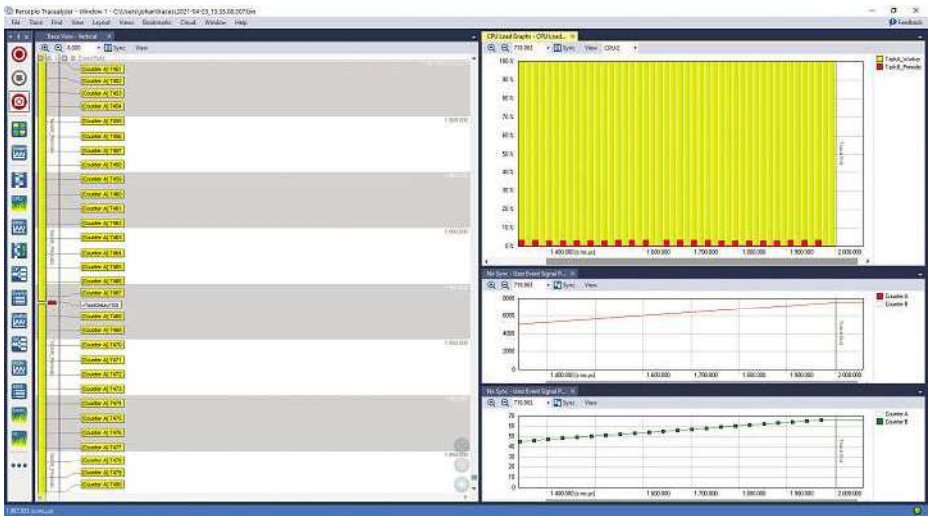

Bild: Percepio

Bild 3: *Indem TaskB nicht mehr in eine Warteschleife geschickt, sondern angehalten wird, kann TaskA nahezu 100 Prozent der Prozessorzeit nutzen und in der gleichen Zeit die doppelte Arbeit bewältigen.*

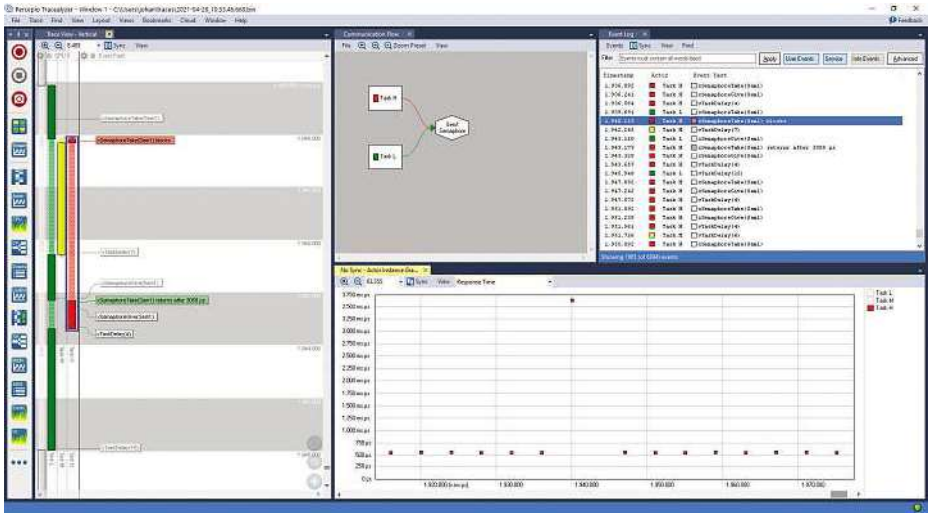

Bild: Percepio

Bild 4: *Tracealyzer-Darstellung der Prioritätsinversion.*

gerungsfunktion HAL_Delay nicht Bestandteil des RTOS-Kernels ist und daher die aktuell laufende Task nicht wirklich anhält, sondern lediglich für die angegebene Zeitspanne in einer Warteschleife aufhält. Mit dieser Implementierung werden somit also nahezu 50 % der Prozessorzeit vergeudet.

50 Prozent Prozessorzeit verschwendet

Um Abhilfe zu schaffen, wird TaskB so umgeschrieben, dass die vom RTOS zur Verfügung gestellte Verzögerungsfunktion vTaskDelay genutzt wird. Außerdem wird die Priorität von TaskB angehoben, sodass sie die Ausführung von TaskA unterbrechen kann, sobald sie aktiviert wird. So lässt sich sicherstellen, dass die Ausführung von TaskB unmittelbar nach dem Aufruf von vTaskDelay beginnen kann (Bild 3). Die Folge ist, dass TaskB nun exakt alle 10 ms ausgeführt wird und so lange läuft wie nötig. Während der Wartezeit verbraucht sie dagegen keine Prozessorzeit mehr und so kann TaskA nahezu die gesamte Zeit aktiv sein. Die Leistungsfähigkeit dieser Applikation hat sich also durch schlichtes Ändern von nur zwei Codezeilen im Prinzip verdoppelt. Diese Modifikation lässt sich ganz einfach durchführen, wenn man über die Abläufe zur Laufzeit Bescheid weiß. Sonst wäre dieses Problem unbemerkt geblieben – besonders dann, wenn die fraglichen Codezeilen tief in einer umfangreichen Codebasis verteilt gewesen wären.

Auch wenn dies nur ein ganz einfaches Beispiel war, gilt doch für alle RTOS-basierten Applikationen der Grundsatz, dass das Timing der Software so stabil und deterministisch wie möglich sein muss. Dies wiederum verlangt nach minimalen Timing-Variationen. Sind mehrere Tasks mit wechselndem Timing vorhanden, die sich gegenseitig beeinflussen, kann es eine Unmenge möglicher Verarbeitungsmuster geben. Das Resultat ist ein chaotisches Laufzeitverhalten, das sich nur schwierig mit hinreichender Verlässlichkeit testen und debuggen lässt.

Ein namhaftes Beispiel aus den 1980er Jahren ist das computergesteuerte Strahlentherapiegerät Therac-25, das mit zahlreichen Softwareproblemen (z. B. Race Conditions) behaftet war, wodurch es in mindestens sechs Fällen zu einer massiven Überdosierung der Strahlung kam. Dieses Beispiel mag extrem sein, aber es zeigt, dass es auch in einem noch so gut entwickelten System gelegentlich zu nicht geprüften Verarbeitungsmustern kommen kann, die mit statischer Codeanalyse nahezu unmöglich zu reproduzieren sind.

Semaphoren können Probleme hervorrufen

Betrachten wir nun einen weiteren dynamischen Laufzeitfehler, der per Codeinspektion keinesfalls detektierbar wäre. Es ist ein grundlegendes Wesensmerkmal von Echtzeit-Betriebssystemen, dass Tasks mit höherer Priorität vor solchen ausgeführt werden, deren Priorität niedriger ist. Niederpriore Tasks werden angehalten, wenn eine höherpriore Task anklopft und die CPU anfordert. Paradoxerweise kann jedoch genau das Umgekehrte passieren und die höherpriore Task muss auf diejenige mit niedrigerer Priorität warten. Dieses Problem ist unter der Bezeichnung Prioritätsinversion bekannt. Wie kommt es dazu?

Gelegentlich nutzt man Semaphoren, um den Zugang zu gemeinsam genutzten Ressourcen zu koordinieren. Sobald eine Task in einen kritischen Abschnitt zur Nutzung einer globalen Ressource eintritt, versucht sie den zugehörigen Semaphor zu belegen. Ist der Semaphor bereits belegt, weil bereits eine andere Task den kritischen Abschnitt ausführt, hält der RTOS-Kernel die Task an, bis die andere Task den Semaphor freigibt. Dieses relativ gängige Schema verursacht ein potenzielles Problem, sobald eine niederpriore Task, die den Semaphor belegt, von einer anderen Task mittlerer Priorität angehalten werden kann. In diesem Fall nämlich werden alle Tasks, die auf den Semaphor warten, angehalten – auch solche mit höherer Priorität. So kommt es zur besagten Prioritätsinversion, bei der die höherpriore Task auf die Task geringerer Priorität warten muss (Bild 4). Ein höchst prominentes Beispiel für einen Fall, in dem die Prioritätsinversion zu Pro-

Bild: Percepio

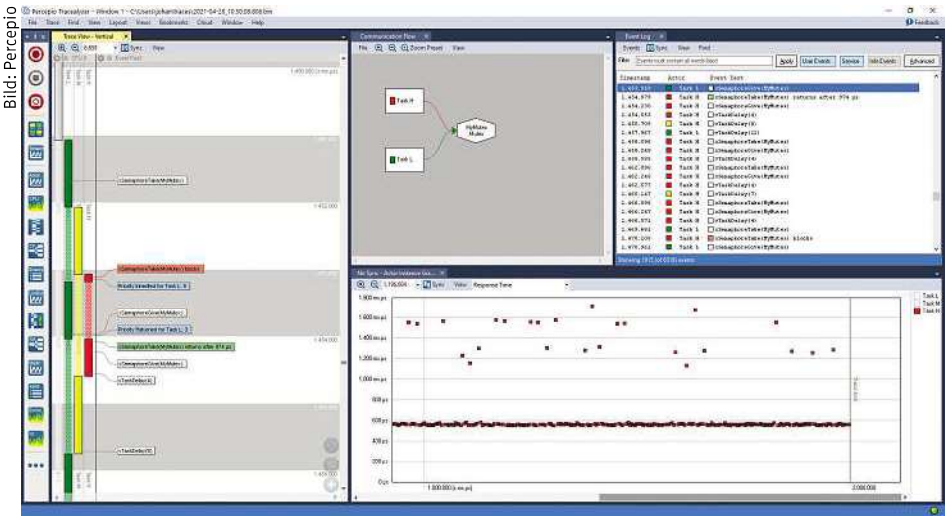

Bild 5: *Vermeidung der Prioritätsinversion durch Prioritätsvererbung (kenntlich an den blauen Event-Labels)*

blemen führte, war ein Raumfahrzeug der NASA. Die NASA-Ingenieure hatten bei der Pathfinder-Mission zum Mars im Jahr 1997 ein Prioritätsinversions-Problem übersehen. Es gab damals eine lange laufende Task, die in seltenen Fällen eine höherpriore Task an der Ausführung hindern konnte, woraufhin ein Watchdog-Timer ablief.

Prioritätsinversion: Mars-Mission in Gefahr

Dies führte während der Mission zu einem Reset des Gesamtsystems. Den NASA-Ingenieuren gelang es jedoch binnen weniger Tage, das Problem zu lösen. Sie konnten die Abläufe an einer exakten Kopie des Systems auf der Erde reproduzieren, während sie die Verarbeitung des Systems per Tracing verfolgten, um das Problem schließlich zu lokalisieren. Abhilfe gegen Prioritätsinversionen kann ein RTOS-Feature schaffen, das als Prioritätsvererbung (Priority Inheritance) bezeichnet wird. Hierbei wird die Scheduling-Priorität der Task, die gerade die Ressource belegt, auf das Niveau der wartenden Task angehoben, um Unterbrechungen durch Tasks mit dazwischen liegender Priorität auszuschließen.

Diese Funktionalität ist oft bei Mutex-Objekten gegeben. Diese haben Ähnlichkeit mit Semaphoren, sind aber speziell für gegenseitigen Ausschluss vorgesehen. Wenn eine hochpriore Task einen Mutex zu belegen versucht, wird sie so lange blockiert, wie der Mutex von der bisherigen Task belegt wird. Während dieser Zeit aber „erbt“ die niederpriore Task die Priorität der wartenden, höherprioren Task. Eine Task mittlerer Priorität kann daher die gerade laufende Task nicht unterbrechen, bis diese den kritischen Abschnitt absolviert und den Mutex freigegeben hat (Bild 5). Die Prioritätsänderungen sind in der Trace-Ansicht in Bild 6 klar zu erkennen. Ohne Prioritätsvererbung wäre das mittlere Teilstück der niederprioren Task nicht ausgeführt und die Ressource somit nicht freigegeben worden, bis die Task mittlerer Priorität beendet gewesen wäre.

Zur Vermeidung dieses Problems halten viele Echtzeit-Betriebssysteme Mutex-Objekte bereit, bei denen die Prioritätsvererbung grundsätzlich aktiviert ist. Im Fall der Pathfinder-Mission war die Prioritätsvererbung beim Einrichten des Mutex dagegen nur eine Option, die von den NASA-Ingenieuren deaktiviert wurde, um die Mutex-Operationen ein wenig schneller zu machen – keine gute Idee, wie sich herausstellte.

Prioritäten vererben für geordneten Task-Ablauf

Wie nicht zuletzt dieses Beispiel zeigt, kann es sinnvoll sein, zum Schutz kritischer Codeabschnitte nicht auf Semaphoren, sondern auf Mutexe mit Prioritätsvererbung zu setzen, um Prioritätsinversions-Probleme zu vermeiden. Letztere können allerdings dennoch auftreten, wenn andere gemeinsam genutzte Ressourcen (z. B. Message Queues) die Ausführung blockieren. Trace-Tools wie Tracealyzer liefern eine detaillierte Analyse eines arbeitenden Systems. Hierdurch wird es für Entwickler ersichtlich, wie die Tasks laufen und miteinander interagieren und exotische Laufzeitprobleme wie etwa Prioritätsinversionen lassen sich leichter identifizieren, um die Zuverlässigkeit des Systems zu verbessern. // **ME**

Percepio