Embedded Software Engineering Kongress

ESE Kongress 2015

Vortragsskript:

MyOS - Kochbuch für ein Mini-Betriebssystem
C-Implementierung eines eigenen Kernels auf dem Cortex-Mx

Remo Markgraf, MicroConsult GmbH

In kleineren Anwendungen setzen Sie normalerweise kein Betriebssystem ein, weil Sie den Overhead scheuen? Geht ja auch noch ganz gut ohne, bis dann die Änderungen kommen…

Hier finden Sie eine einfache Anleitung, wie Sie auch für solche Anwendungen ein ganz auf Ihre Bedürfnisse zugeschnittenes Mini-Betriebssystem realisieren können. Das Rad muss dazu nicht neu erfunden werden; der lauffähige Beispielcode steht zum Download zur freien Verfügung. Es ist gar nicht so schwer, sich für seine kleineren Projekte ein "Betriebssystem" zu schaffen.

While-Loop ade!

Die Antwort auf die Frage „Warum ein Mini-Betriebssystem?“ lässt sich am besten durch die Umkehrfrage geben „Was hält mich davon ab?“.

- Unnötige Komplexität, zusätzliche Abhängigkeiten
- Zusätzliche Ressourcen (Speicher, Laufzeit)
- Auswahl eines geeigneten Betriebssystems incl Einarbeitung

Was aber, wenn Sie diese Faktoren selbst im Griff haben, keine zusätzliche Abhängigkeit, keine Auswahl oder Einarbeitung, nur so viel Ressourcen, wie Sie für unbedingt nötig erachten? Dann wäre das schon bequem. Ergänzungen lassen sich einfach hinzufügen, ohne das Timing jedes Mal neu zu berechnen oder zumindest zu überschlagen. Und zuverlässiger würde es ja auch werden, wenn die Tasks nur ihre eigenen Zugriffsrechte hätten. Also los, so schwer ist das ja gar nicht - und die Cortex-Mx-Details dabei besser kennen zu lernen ist ein zusätzlicher Vorteil.

MyOS Kochrezept

Abb. 1: MyOS Kochbuch

Die Realisierung eines eigenen Mini-Betriebssystems ist leider etwas komplexer als das Kuchenbacken, doch mit der hier vorgestellten „Fertigbackmischung“ MyOS auch für weniger Geübte umsetzbar. In MyOS werden die architekturellen Vorkehrungen des Cortex-Mx ausgeschöpft. Eine Portierung auf eine andere Zielarchitektur ist daher nicht vorgesehen, da dies deutlich zu Lasten einer einfachen Implementierung und der Effizienz gehen würde. Das hier vorgestellt MyOS ist dem Cortex-Mx quasi „auf den Leib“ geschnitten.

Main und Process Stack des Cortex-Mx

Die Cortex-Mx-Architektur verfügt über zahlreiche Elemente, die zur Realisierung eines Betriebssystems sehr hilfreich sind. So verfügt sowohl die Architektur ARMv6-M (Cortex-M0, M0+, M1) als auch die ARMv7-M (Cortex-M3, M4, M7) über zwei umschaltbare Stack Pointer, den Main Stack Pointer (MSP) und den Process Stack Pointer (PSP).

Während der Ausführung von Interrupt Service Routinen (Handler Mode) wird immer der MSP verwendet. Nach dem Booten ist ebenfalls immer der MSP aktiv. Während des normalen Programmablaufs (Thread Mode) wird als Default ebenfalls der MSP verwendet.

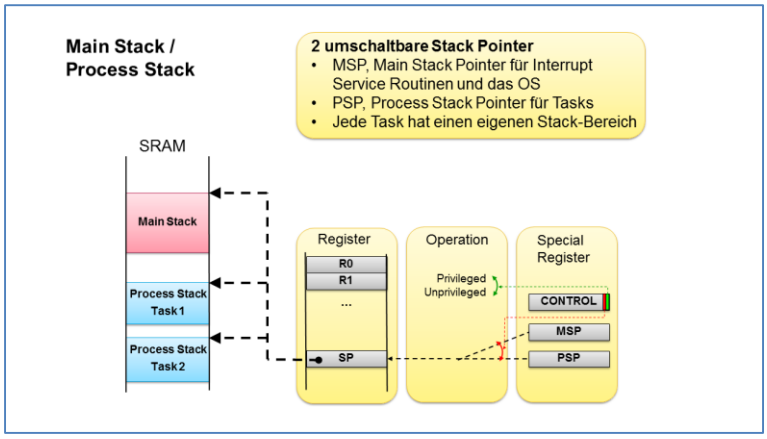

Abb. 2: Main und Process Stack

Die Verwendung des PSP muss durch Setzen eines Bits im CONTROL Register des Prozessors aktiviert werden. Daher wird in Anwendungen ohne Betriebssystem meistens nur der MSP verwendet. In Anwendungen mit Betriebssystem, auch dem hier vorgestellten kleinen MyOS, wird der MSP ausschließlich im Handler Mode und für das Betriebssystem selbst verwendet. Die Anwendungen (Tasks) verwenden den PSP. Um die Zuverlässigkeit des Systems zu erhöhen, ist es üblich, jeder Task einen eigenen Process Stack zuzuweisen und beim Umschalten zwischen den Tasks den PSP jeweils auf den taskspezifischen Process Stack zu setzen.

Privileged und unprivileged Operation

Durch Setzen eines weiteren Bits im CONTROL Register kann der Prozessor in den nicht-privilegierten Operationsmodus versetzt werden. In diesem Operationsmodus sind Zugriffe auf interne Register nur eingeschränkt möglich, und speziell im Zusammenspiel mit der Memory Protection Unit (MPU) kann der Zugriff auf Speicherbereiche gezielt freigegeben und gesperrt werden. Dies erhöht weiter die Zuverlässigkeit des Systems, da einzelne Tasks somit weder das Betriebssystem noch andere Tasks unerwünscht beeinflussen können. Mehr dazu später im Abschnitt MPU.

In Anwendungen mit Betriebssystem, auch dem hier vorgestellten kleinen MyOS, werden daher Tasks im nicht-privilegierten Operationsmodus betrieben. Sollte eine Task Operationen ausführen müssen, die den privilegierten Operationsmodus verlangen, so können diese gezielt über Betriebssystemaufrufe bereitgestellt werden (siehe SVC weiter unten).

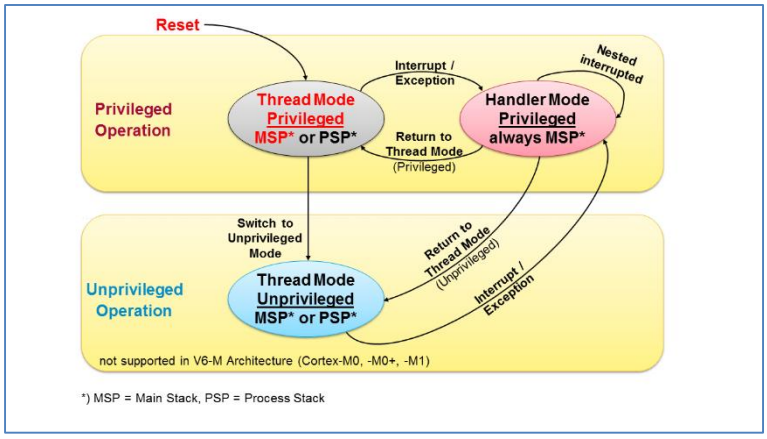

Abb 3: Privileged und Unprivileged Operation

Exception Modell des Cortex-Mx

Die Cortex-Mx-Architektur beinhaltet einen Nested Vectored Interrupt Controller (NVIC). Er verfügt über bis zu 240 Interrupt-Inputs (32 beim Cortex-M0, M0+, M1) zur Peripherie und 10 interne Exceptions (6 beim Cortex-M0, M0+, M1). Beim Auslösen eines Interrupts oder einer Exception (Event) wird aus der ISR-Vektortabelle die Adresse der entsprechenden Behandlungsroutine (Handler) gelesen und ausgeführt. Jeder Interrupt und jede Exception hat eine festgelegte oder definierbare Priorität. Eine Interrupt Service Routine kann durch einen höherprioren Event unterbrochen werden (Preemption) und wird nach Behandlung des höherprioren Events weiter ausgeführt. Ein anliegender Event mit gleicher oder niedrigerer Priorität wird im Anschluss an den aktuellen Event ausgeführt. Bei der Unterbrechung einer aktuellen Task oder eines niederprioren Events rettet der Controller bestimmte Registerinhalte (Kontext = R0-R4, R12, LR, PC, PSR, die mit Sternchen gekennzeichneten Register in Abbildung 5) auf den aktuellen Stack (Stacking) und stellt dessen Inhalt bei der Rückkehr aus der Service Routine wieder her (Unstacking). Diese Architektur erlaubt die Verwendung normaler C-Funktionen als ISR-Handler, da im Gegensatz zu vielen anderen Controllern kein spezieller Maschinenbefehl zur Rückkehr aus einer ISR benötigt wird.

Für ein Betriebssystem sind die drei Exceptions
- System Tick Timer (SysTick),
- Pendable Request for System Service (PendSV) und
- System Service Call (SVC)

von besonderer Bedeutung.

Der SysTick sorgt für das grundlegende Timing des Systems und wird auf eine bestimmte wiederkehrende Interruptfolge von z.B. 5ms eingestellt. Dem SysTick weist man eine relativ hohe Priorität zu. So kann er nur von höherprioren Interrupts unterbrochen oder kurzzeitig verzögert werden. Im SysTick-Handler zählt man einen TickCounter hoch und triggert den Scheduler, der für die Taskumschaltung sorgt.

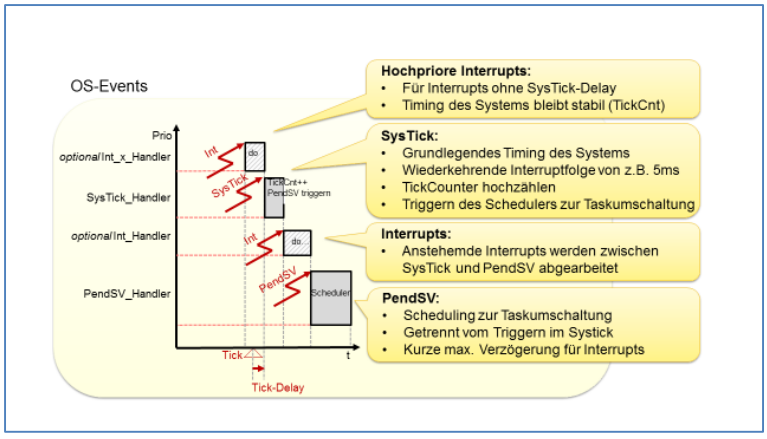

Abb. 4: Zusammenspiel des SysTick und PendSV

Das Triggern des Schedulers erfolgt über Auslösen einer PendSV-Exception. Im PendSV-Handler wird das Umschalten zwischen den Tasks erledigt (siehe Scheduler).Der PendSV erhält eine sehr niedrige Priorität. Durch die Trennung in das Auslösen des Schedulers im hochprioren SysTick und in das eigentliche Task Switching im niederprioren PendSV können anstehende Interrupts dazwischen ausgeführt werden. In zeitkritischen Anwendungen kann selbst die kurze Verzögerung durch den SysTick-Handler zu lange sein. Hier bietet sich an, den zeitkritischen Interrupts eine Priorität zu geben, die höher ist als der SysTick. So lassen sich sehr kurze Interrupt-Reaktionszeiten erzielen.

Der SVC ist ein „Software-Interrupt“ und wird durch einen eigenen Maschinenbefehl ausgelöst. Seine Priorität sollte höher als die des PendSV und niedriger als die der Interrupts sein. Dadurch werden Betriebssystemaufrufe nicht durch den Scheduler unterbrochen, also in der jeweiligen Task beendet, und dennoch Interrupts durch die Peripherie nicht verzögert. Im SVC-Handler werden, wie der Name „Supervisor Call“ schon nahelegt, Betriebssystemaufrufe realisiert. Da der SVC sofort ausgeführt wird, kann der Aufrufer dem SVC-Handler Parameter für den Betriebssystemaufruf mitgeben. Allerdings müssen diese immer vom Stack gelesen werden, da sich ein höherpriorer Interrupt „dazwischenmogeln“ könnte. Im SVCHandler (Handler Mode) arbeitet der Controller immer privileged und kann daher Zugriffe ausführen, zu denen er in einer unprivilegierten Task (Thread Mode) keine Berechtigung hätte. Die Zugriffsrechte werden maßgeblich durch die MPU gesteuert.

Memory Protection Unit

Der Cortex-M0+, M3, M4, M7 verfügt über eine optionale MPU. Über die MPU lässt sich ein System wie folgt robuster und sicherer machen:

- Verhindern des Überschreibens von Stack- oder Speicherbereichen anderer Tasks oder des Betriebssystems
- Einschränkung des Zugriffs auf bestimmte Peripherie und bestimmte Speicherbereiche
- Definition von Speicherbereichen als nicht ausführbar und damit Verhinderung von Code Injections

Die MPU verfügt über acht Regionen, deren Größe und Startadresse unabhängig festgelegt werden können. Für jede Region wird für den privilegierten und für den nicht-privilegierten Operationsmodus die Zugriffsberechtigung - nur lesend, lesend und schreibend, kein Zugriff und nicht ausführbar – festgelegt. Jede Region lässt sich wiederum in 8 aneinandergrenzende und gleichgroße Subregionen unterteilen, die über ein Bitfeld separat ein- und ausgeschaltet werden können.

Die MPU wird über sechzehn 32-Bit-Register (2 pro Region) gesteuert, die über spezielle Multiregister-Maschinenbefehle in nur zwei Lese- und zwei Schreibbefehle komplett umprogrammiert werden können. Damit kann die MPU im Scheduler bei jedem Task Switching effizient dynamisch umprogrammiert werden. Hierfür wird im Task Control Block jeder Task ein Speicherbereich von 16 32-Bit-Werten angelegt, in welchem die jeweiligen MPU-Settings gespeichert und beim Task Switching in die MPU-Register kopiert werden. Die Speicherbereiche der Peripherie, z.B. der Ports, sind i.d.R. jeweils gleich groß und aufeinanderfolgend, z.B. 256 Byte je Port. Damit kann über die Subregions den einzelnen Tasks über nur eine Region gezielt Zugriff auf einzelne Ports ermöglicht werden.

Zusammenspiel C und Assembler

Es gibt wenige Anwendungsfälle, in denen man von C auf Assembler zurückgreifen sollte. Im MyOS-Kontext gibt es drei Bereiche, in denen dies sinnvoll ist:

- Sichern und Wiederherstellen des Kontexts im Scheduler, da hier am Stack. Auswahl der nächsten Task kann allerdings auch gut in C implementiert werden.
- Absicherung eines funktionierenden Stacks im Hardfault-Handler. Das eventuelle Dumpen der Fehlersituation kann dann mit dem abgesicherten Stack in C realisiert werden.
- Dispatchen der unterschiedlichen Betriebssystemaufrufe und Stackbehandlung der Parameter im SVC-Handler. Die eigentlichen Betriebssystemroutinen können dann wiederum in C geschrieben werden.

Keine Angst - das Rad muss nicht immer neu erfunden werden. Sie können MyOS ohne Assembler-Expertenwissen einsetzen und nach Belieben erweitern, da die oben genannten Anwendungsfälle bereits in der Downloadversion vorhanden und ggf. mit wenig Aufwand anpassbar sind. Die eigentlichen individuellen Ergänzungen lassen sich so in C realisieren.

Die “Procedure Call Standard for the ARM Architecture“ (AAPCS) regelt das reibungslose Zusammenspiel der Controller-Hardware mit SW-Routinen (Assembler, C, C++).

In den Registern R0-R3 werden die ersten vier Funktionsparameter beim Aufruf an eine Funktion übergeben. Für 64-Bit-Werte (longlong oder double) werden Registerpaare verwendet. Die Funktion kann über Register 0 (oder Registerpaar R0-1) ein Ergebnis an den Aufrufer zurückliefern. Werden in einer Funktion R4-R11 verwendet, so müssen dessen Werte vor Verwendung gesichert (auf dem Stack mit push) und vor Verlassen der Funktion wiederhergestellt werden (pop).

Dies macht der Compiler automatisch für Sie; in Assembler müssen Sie selbst dafür sorgen. AAPCS legt außerdem fest, dass der Stack immer 8 Byte aligned sein muss. Daher sollten Sie nur eine gerade Anzahl von Registern auf den Stack pushen.

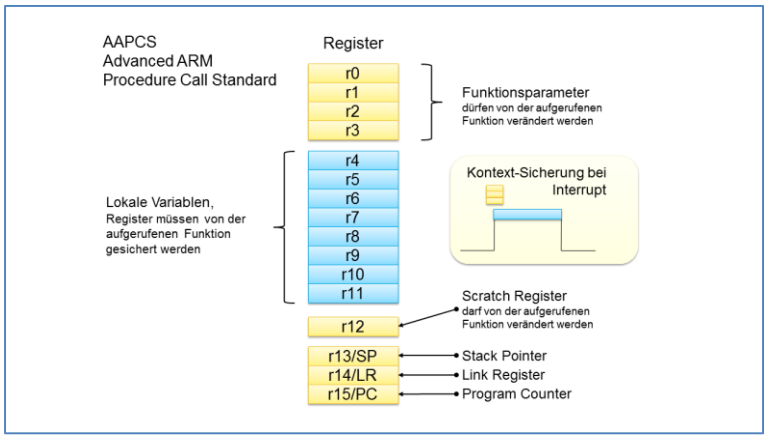

Abb. 5: Register-Verwendung gemäß AAPCS

Die Syntax für die Implementierung von Inline- oder Embedded-Assembler-Routinen in C-Modulen ist stark abhängig von der verwendeten Toolchain. Es empfiehlt sich daher, für Routinen in Assembler lieber eigene Files mit der Endung s zu verwenden. Als Interface erstellen Sie eine h Datei, welche die Funktionsprototypen in C-Syntax enthält. Für den aufrufenden Programmteil besteht somit kein Unterschied, ob eine Routine in C/C++ oder Assembler realisiert ist. Zum Aufruf von C-Routinen aus Assembler heraus müssen Sie den Funktionsnamen als externe Referenz deklarieren, z.B. in Keil µVision und in IAR mit IMPORT oder in GNU mit .extern und sich natürlich an die AAPCS halten. In den Download-Dateien finden Sie für die drei oben erwähnten Anwendungsfälle passende Codebeispiele, die Sie meistens ohne Änderungen verwenden können.

Scheduler

Das Kernstück jedes Betriebssystems ist der Scheduler. Er übernimmt die Zuteilung der Prozessor-Ressourcen auf die verschiedenen Tasks. Damit der Scheduler möglichst unabhängig von dem Verfahren der Zuteilung ist, wird er in drei Teile zerlegt.

- 1. Sichern des Kontexts der aktuellen Task (in Assembler)
- 2. Auswahl der nächsten Task (in C oder Assembler)
- 3. Herstellen des neuen Kontextes und Umschalten auf die neue Task (in Assembler), ggf. inklusive der dynamischen Umschaltung der MPU-Register

Wie bereits im Exception-Modell erwähnt, wird der Scheduler als Interrupt Service Routine der PendSV-Exception realisiert und vom SystemTick getriggert.

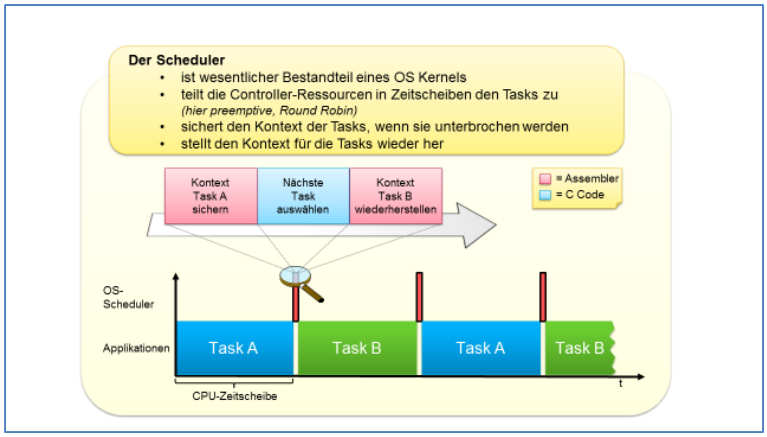

Abb. 6: MyOS Scheduler-Aufteilung

Round Robin

Ein einfaches Verfahren zur Auswahl der nächsten Task ist das Round Robin. Dabei wird den Tasks der Reihe nach eine bestimmte Zeitscheibe zugeordnet. Das einfachste Verfahren ergibt sich, wenn man die Zeitscheiben gleich lang lässt und auch keine Auswahl-Prioritäten verwendet. Wird der Auswahl-Algorithmus des Schedulers in C implementiert, lässt sich auf einfache Weise eine Priorisierung hinzufügen.

Prioritäten

Die Priorität einer Task lässt sich auf verschiedene Art und Weise realisieren. Sehr einfach wird die Umsetzung, wenn man die Zeitscheiben immer gleich lang lässt und durch die Priorität nur die Zuteilungshäufigkeit steuert.

Ein einfaches Verfahren, das in dem downloadbaren Quellcode auch so umgesetzt ist, besteht darin, eine Schleife zu verwenden, deren Zählweite bei jedem Durchlauf inkrementiert wird. Also beim ersten Durchlauf von 0 bis 1, dann von 0 bis 2, usw. Die Schleifenzählvariable wird als static deklariert, behält somit zwischen den Scheduler-Aufrufen ihren Wert und setzt die Arbeit an der letzten Stelle fort. Wenn die höchste vergebene Priorität erreicht ist, wird der Schleifenzähler zurückgesetzt.

In der Schleife werden der Reihe nach die Tasks mit der entsprechenden Priorität ausgewählt. Existieren mehrere Tasks mit der gleichen Priorität, werden sie in der Reihenfolge abgearbeitet, in der sie erzeugt wurden. Natürlich könnte man das Konzept um eine Subpriorität erweitern. Dies ginge jedoch zu Lasten einer höheren Umschaltzeit zwischen den Tasks und ist für das Anwendungsziel daher eher übertrieben.

Beispiel: Es gibt …
2 Tasks A und B mit Prio 1, eine Task C mit Prio 2 und eine Task D mit Prio 3, wobei eine kleine Prioritätszahl eine hohe Priorität der Task bedeutet.

Als Auswahlreihenfolge ergibt sich damit:
- Runde 1: A B
- Runde 2: A B C
- Runde 3: A B C D
- Runde 1: …

In diesem Beispiel gibt es also in Summe der Runden 2+3+4 = 9 Taskaufrufe.

Die Verteilung der CPU-Zeit ergibt sich daher zu:
- A: 3x entsprechend 3/9 ~= 33% der Zeit
- B: 3x entsprechend 3/9 ~= 33% der Zeit
- C: 2x entsprechend 2/9 ~= 22% der Zeit
- D: 1x entsprechend 1/9 ~= 11% der Zeit

Alternativ lässt sich die Priorität auch dadurch steuern, dass eine Task mehrere Zeitscheiben erhält, ohne dass dazu der Scheduler aufgerufen werden muss. Dazu wird im SysTick jeweils nur eine taskspezifische Variable heruntergezählt und der Scheduler erst bei Erreichen des Wertes 0 aufgerufen. Hier entfällt der mehrfache Overhead des Schedulers; dafür würde das Hochsetzen der Priorität einer Task in einer Interruptroutine nicht beim nächsten Tick Wirkung zeigen können, sondern erst, wenn die n Ticks der aktuellen Task vergangen sind.

Interrupts

Folgende unterschiedliche und sich ergänzende Verfahren zur Behandlung von Interrupts werden im Kontext von MyOS hier kurz umrissen:

- Registrierung von taskspezifischen Interrupt-Handlern
- Interrupt-Handler im privilegierten Modus
- Interrupt-Handler im nicht-privilegierten Modus

Um ein architekturelles Layering und die Unabhängigkeit des Betriebssystems von den Applikationen zu ermöglichen, ist es notwendig, dass Tasks ihre Interrupt-Handler beim Betriebssystem registrieren. Dazu wird einfach der Funktionspointer auf den entsprechenden Handler als Parameter eines SVC-Aufrufs mitgegeben. Die über den SVC aufgerufene Betriebssystemroutine schreibt den Funktionspointer in eine Liste. Beim Auftreten des entsprechenden Interrupts werden die Routinen in der Liste der Reihe nach ausgeführt. Wer die Reihenfolge bestimmen möchte, kann die einfache Liste in eine Linked-List abändern und die Registrierungsroutine um eine Sortierung erweitern. In den meisten Fällen wird wohl eher nur eine einzige Interrupt-Routine pro Interrupt registriert werden, so dass sich dieser Zusatzaufwand erübrigt.

Da der Cortex-Mx beim Ausführen von Interrupts automatisch in den Handler-Mode wechselt, der privilegierten Zugriff erlaubt und den MSP nutzt, ist der oben beschriebene zweite Spiegelpunkt „Interrupt-Handler im privilegierten Modus“ mit der Registrierung bereits abgedeckt. Der Anwendungsfall des dritten Spiegelpunkts „Interrupt-Handler im nichtprivilegierten Modus“ ist interessant, da man die Zuverlässigkeit des Systems dadurch weiter steigern kann. Jede Routine erhält nur die Zugriffsrechte, die sie tatsächlich benötigt. Also warum eine Interrupt-Routine im privilegierten Modus ausführen, wenn sie das gar nicht benötigt? Im Cortex-Mx ist dieser Fall leider nur umständlich realisierbar. Dazu werden in Assembler die Stacks manipuliert und Inhalte umkopiert. Letztlich wird der Handler-Modus zur Ausführung des Interrupt-Handlers im nicht-privilegierten Modus temporär verlassen, um danach wiederum im Handler-Modus die Aufräumarbeiten auf den Stacks durchzuführen. Weitere Details würden diesen Rahmen hier leider sprengen und finden sich in z.B. weiterführender Literatur [1, Kapitel 23 Advanced Topics].

Wait States

Eine immer wiederkehrende Aufgabe ist das Warten einer Task auf das Verstreichen einer bestimmten Zeitspanne oder auf das Auftreten eines bestimmten Events. Letzteres ist nicht zu verwechseln mit der Reaktion auf einen Interrupt. In der Interrupt-Routine wird üblicherweise die sofortige, meist von der HW geforderte Reaktion verarbeitet, z.B. Lesen eines empfangenen Datenwertes. Der empfangene Wert kann in der Interrupt-Routine z.B. in einen Pufferbereich geschrieben werden.

Die eigentliche Verarbeitung kann jedoch zeitversetzt in der Task erfolgen. Dazu muss die Task auf diesen Event warten. Das Warten wird durch den Scheduler erledigt, indem eine wartende Task so lange nicht an die Reihe kommt, bis die Wartebedingung nicht mehr erfüllt ist oder eine optionale maximale Zeit verstrichen ist. Da hier nun der Fall auftreten kann, dass alle Tasks warten, braucht man eine Idle-Task. Man könnte auch die Zeit damit verbringen, ständig zu prüfen, ob eine der Bedingungen sich geändert hat, doch ist es sinnvoller, den Controller in der Idle-Task schlafen zu legen, dadurch den Stromverbrauch erheblich zu reduzieren und ihn durch auftretende Events wieder aufzuwecken.

Eine einfache Realisierung des Wartens auf das Verstreichen einer vorgegebenen OsWaitTicks() mit der Anzahl zu wartender Ticks als Parameter auf. In der entsprechenden Betriebssystemroutine wird zur Anzahl der bereits verstrichenen Ticks (TickCounter) die Anzahl der zu wartenden Ticks hinzugerechnet und im TCB der Task gespeichert.

Dann wird der Scheduler über den PendSV ausgelöst. Im Scheduler wird die Task solange nicht aktiviert, bis der aktuelle TickCounter größer ist als der im TCB gespeicherte Wert Damit kehrt die Verarbeitung erst aus der OSWaitTick-Routine zurück, wenn die Anzahl der zu wartenden Ticks verstrichen ist.

Zusammenfassung

Für Anwendungen, in denen Sie komplexe Peripheriezugriffe z.B. über einen TCP/IP-Stack auf Ethernet benötigen, kommen Sie deutlich an die Grenzen eines eigenen Ansatzes. Hier würden Sie das Rad neu erfinden und tun gut daran, auf ein bestehendes Betriebssystem zurückzugreifen. Für weniger komplexe Anwendungsfälle, in denen Sie bislang kein Betriebssystem einsetzen und solche, in denen Sie schon mit dem Gedanken daran gespielt haben, aber den Overhead scheuten, ebnet die „Fertigbackmischung MyOS Downloads“ den Weg. Probieren Sie es doch einfach aus!

Download von C- und Assembler-Source-Files

Über den Link www.microconsult.de/MyOS können Sie Sich eine Basisversion des hier besprochenen Systems herunterladen.

Mit einer E-Mail an MyOS@microconsult.de erhalten Sie gerne auch weiterführende Codebeispiele, die auch im neuen MicroConsult-Training zu diesem Thema Verwendung finden.

Weiterführung und Ausblick

In einem Vortrag und Artikel lässt sich dieses Thema über eine oberflächliche Behandlung hinaus leider nicht behandeln.

Weitere hochinteressante Themen, z.B. Wie stelle ich sicher, dass eine Task …
- nur bestimmte Betriebssystemaufrufe ausführen kann,
- gesteuert über Semaphore nur auf bestimmte Ressourcen zugreifen kann,
- nur Pointer auf Bereiche mit Zugriffserlaubnis verwenden kann,
- nur Interrupts im nicht privilegierten Zustand ausführen kann,
- dynamisch neue Tasks anlegen darf,
- einen Prozessmonitor zum Debuggen mitlaufen lassen kann,
- im Fault-Handler die richtigen Informationen gedumped werden
- Zugriffsverletzungen so abfangen kann, dass das System weiterläuft

… mussten hier leider unberührt bleiben.

Gerne geben wir auf diese Fragen Antworten, zusammen mit vielen Übungen im neuen Training zu diesem Thema, das ab Frühjahr 2016 bei MicroConsult buchbar ist.

Begriffe und Abkürzungen

- Fault Handler: Eine Exception Routine, die angesprungen wird, wenn im Controller ein entsprechender Fehlerzustand auftritt.
- Scheduler: Programmteil im Kern des Betriebssystems, der für das Umschalten zwischen den Tasks verantwortlich ist.
- Task Control Block: Speicherbereich im RAM je Task; enthält Platz für den PSP, Register, MPU-Settings, Priorität, Waitzyklen, usw.
- AAPCS: Procedure Call Standard for the ARM Architecture
- CMSIS: Cortex Microcontroller Software Interface Standard
- CPU: Central Processing Unit (Prozessorkern)
- ISR: Interrupt or Exception Service Routine
- MPU: Memory Protection Unit
- MSP: Main Stack Pointer
- NVIC: Nested Vectored Interrupt Controller
- OS: Operation System (Betriebssystem)
- PendSV: Pendable Request for System Service (Exception)
- PSP: Process Stack Pointer
- RTOS: Real-Time Operation System
- SVC: System Service Call (Exception)
- SysTick: System Tick Timer (Exception)

Literatur- und Quellenverzeichnis

[1] Definitive Guide to ARM® Cortex-M3 and Cortex-M4 Processors, Joseph Yiu, Newnes, Dez 2013
[2] The Definitive Guide to the ARM® Cortex-M0, Joseph Yiu, Newnes, Feb 2011
[3] The Designer's Guide to the Cortex-M Processor Family: A Tutorial Approach,
Trevor Martin, Newnes, Mai 2013
[4] Real-Time Operating Systems for Arm® Cortex-M Microcontrollers,
Jonathan Valvano, CreateSpace Independent Publishing Platform, Jan 2012
[5] Introduction to Arm® Cortex(TM)-M Microcontrollers
Jonathan Valvano, CreateSpace Independent Publishing Platform, Mai 2012
[6] Procedure Call Standard for the ARM Architecture
http://infocenter arm com/help/topic/com arm doc ihi0042e/IHI0042E_aapcs pdf
[7] Cortex Microcontroller Software Interface Standard
www.arm.com/products/processors/cortex-m/cortex-microcontrollersoftware-interface-standard.php
[8] MicroConsult Cortex-Mx Training
www.microconsult.de/um/cortex-mx
[9] MicroConsult Embedded-C Training
www.microconsult.de/um/emb-c

Autor

Remo Markgraf ist Senior Management Consultant bei der MicroConsult GmbH. Neben Begeisterung für Innovation und Leidenschaft für Embedded-Systeme verfügt er über langjährige Projekt- und internationale Führungserfahrung in Softwareentwicklung, Systems Engineering, Projekt-, Produkt-, Innovations- und Business Development Management sowie dem technischen Vertrieb.

Kontakt

Internet: www.microconsult.de
E-Mail: r.markgraf@microconsult.de