Hochschule Esslingen
University of Applied Sciences

Flandernstraße 101
73732 Esslingen
Deutschland
Telefon +49(0)711.397-49
Fax +49(0)711. 397-40 92

Studienarbeit

Entwicklung eines Echtzeitbetriebssystems nach OSEK Spezifikation

HSE Free OSEK

Bearbeitet von:

Franjo Rupcic
Eulenbühlweg 6
70437 Stuttgart
Tel.: 0711/551664

Studiengang:
Technische Informatik

Matrikelnummer: 726892

Mario Penava
Nürnbergerstraße 176
70374 Stuttgart
Tel.: 0711/5284287

Studiengang:
Technische Informatik

Matrikelnummer: 725645

Betreuer: Prof. Dr. rer. nat. (Purdue Univ.) Jörg Friedrich

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

In Anlehnung an die Vorlesung Prozessdatenverarbeitung, die sich mit der Funktion von Echtzeitbetriebssystemen beschäftigt, kam das Thema „Entwicklung eines Echtzeitbetriebssystems“ für die Studienarbeit zustande.
In der Industrie, vor allem im Bereich der KFZ Entwicklung, werden in der Regel so genannte OSEK konforme Betriebssysteme verwendet. Inzwischen ist ein großer Markt entstanden und immer mehr Firmen bieten kommerzielle Betriebssysteme an, die dem OSEK Standard entsprechen. Da es bisher keine Open Source Betriebssysteme gibt die OSEK konform und gut dokumentiert sind, ist die Idee entstanden ein frei erhältliches Echtzeitbetriebssystem zu entwickeln welches dem OSEK Standard gerecht wird und ohne kommerziellen Hintergrund von jedem genutzt und weiterentwickelt werden kann.
Da die Entwicklung eines solchen Betriebssystems sehr aufwendig ist, konnte das Betriebssystem noch nicht vollständig fertig entwickelt werden und bedarf somit noch weiteren Studien bzw. Diplomarbeit. Dennoch sind die wichtigsten Bestandteile des Kernels implementiert und für 2 Derivate funktionsfähig vorhanden.

1.1 Was ist ein Echtzeitbetriebssystem

Heutzutage verwendet jeder PC Anwender ein Betriebssystem (Windows XP, Mac OS, Linux, etc.). Ein Betriebssystem ist die Software, die die Verwendung eines Computers und seiner einzelnen Hardwarekomponenten ermöglicht. Es verwaltet Betriebsmittel wie Speicher, Ein- und Ausgabegeräte und steuert die Ausführung von Programmen. Betriebssysteme bestehen in der Regel aus einem Kern (Kernel), welcher die Hardware des Computers verwaltet, sowie grundlegenden Systemprogrammen, die dem Start des Betriebssystems und dessen Konfiguration dienen.
Es existiert eine ganze Reihe von unterschiedlichen Betriebssystemen welche zu unterschiedlichen Verwendungszwecken dienen. Beispielsweise in einem Handy oder einem Satellitenreceiver ist auch ein Betriebssystem vorhanden, welches andere Anforderungen und Funktionen besitzt als der heimische PC. Anwendungen am PC sind meistens nicht zeitkritisch während bei einem Handy sich die Anforderung auf die „Echtzeit“ des Telefonierens fixiert. Es würde sich niemand damit zufrieden geben wenn beim telefonieren die Sätze des Gesprächspartners 30 Sekunden später eintreffen, während das Empfangen einer email oder das Laden des email-Programms eine Verzögerung tolerieren würde.

Ein Echtzeitbetriebssystem oder auch RTOS (real-time operating system) ist ein Betriebssystem mit zusätzlichen Echtzeit-Funktionen für die Einhaltung von Zeitbedingungen und die Vorhersagbarkeit des Prozessverhaltens. Ein RTOS ist heutzutage in sehr vielen elektronischen Geräten vorzufinden wie z.B. Handys, Satellitenreceivern, Routern, Navigationssystemen, MP3 Player, Autoradios und neben vielen weiteren Produkten auch in Automobilen. In den letzten Jahren ging der Trend der Automobilhersteller immer mehr in die Richtung mehr elektronische Geräte in ihre Produkte zu integrieren, welche dem Fahrzeugführer ermöglichen sicherer, komfortabler, ökonomischer und ökologischer zu fahren. Diese so genannten „elektronischen Helfer“ sind mit Begriffen verbunden wie dem Antiblockiersystem (ABS), elektronisches Stabilitätsprogramm (ESP), Antriebs-Schlupf-Regelung (ASR) und vielen weiteren Anwendungen. „Elektronische Helfer“ werden in einzelnen eingebetteten Systemen hergestellt die sich Steuergeräte nennen welche von einer Vielzahl von Sensoren Informationen erhalten, diese dann mit Algorithmen auswerten und ggf. eine Reaktion darauf auslösen. Diese Steuergeräte besitzen ebenfalls wie das Handy oder der PC ein Betriebssystem welches unterschiedliche Prozesse, die gleichzeitig im Steuergeräte laufen, verwaltet. Z.B. eine Steuergerät was die elektronischen Fensterheber in einem Kraftfahrzeug verwaltet, würde die einzelnen Signale der Schalter auswerten (Fenster öffnen, Fenster schließen) und dementsprechend die Funktion ausführen. Da es in einem System mindestens 4 Schalter und 4 Fenster zu Verwalten gilt, muss das gleichzeitige drücken von mehreren Schalter möglich sein damit sich alle Fenster parallel ansteuern lassen. Da aber ein Prozessor nur eine Aufgabe nach der anderen abarbeiten kann, bietet sich ein Echtzeitbetriebssystem an. Die einzelnen Aufgaben werden dadurch zwar auch sequenziell ausgeführt, allerdings so schnell das es in der Anwendung so erscheint als würden die Prozesse parallel laufen (s.h. Abb. 1).

Für Steuergeräte die Betriebssysteme verwenden gelten bestimmte Anforderungen wie z.B. ein hohes Maß an Sicherheit und Qualität. Da die Steuergeräte von verschiedenen Herstellern entwickelt werden und in der Regel in einem Automobil unterschiedliche Typen

vorhanden sind, wurden die Anforderungen und Schnittstellen der Echtzeitbetriebssysteme in einer Spezifikation zusammengefasst Namens OSEK.

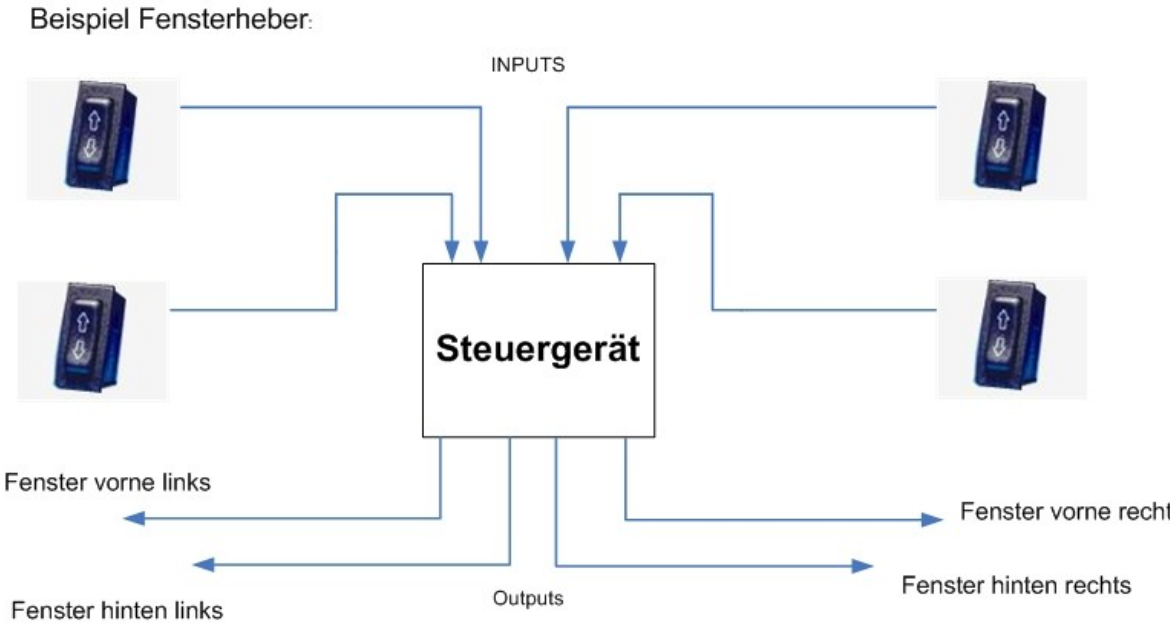


Abbildung 1: Beispiel für ein Steuergerät zur Fensteransteuerung

1.2 Was ist OSEK

OSEK steht für „**O**ffene **S**ysteme und deren Schnittstellen für die **E**lektronik im **K**raftfahrzeug“.
Es handelt sich dabei um einen Standard welcher von internationalen Automobilherstellern und deren Zulieferern erstellt wurde. Das Ziel dieser Spezifikation ist die Schaffung eines Industriestandards für eine offene Fahrzeugarchitektur mit verteilten Steuergeräten. Dazu werden ein Echtzeitbetriebssystem sowie Softwareschnittstellen und Funktionen für Kommunikation- und Netzwerkmanagement Hersteller übergreifend spezifiziert. Betriebssysteme die OSEK konform sind, bieten keine Funktionen um Systemkomponenten dynamisch zur Laufzeit zu erzeugen. Das gesamte System ist statisch konfiguriert und enthält alle benötigten Betriebssystemmittel. Dieses Vorgehen erhöht die Zuverlässigkeit sowie die Reaktionsfähigkeit des Systems. Da Betriebssysteme die nach dem OSEK Standard arbeiten, hauptsächlich in eingebetteten Systemen Verwendung finden, ist eine statische Konfiguration des Systems von Vorteil, da diese während der Laufzeit nicht verändert werden müssen. Des weiteren wird die Portabilität von verschiedenen Anwendungen und die Plattformunabhängigkeit gewährleistet.

1.2.1 Implementierungssprache OIL

Ein weiteres Merkmal des OSEK Standards ist die Implementierungssprache OIL (OSEK Implementation Language). In einer OIL-Konfigurationsdatei werden alle Objekte (Tasks, Events, Alarms, Messages etc.) mit ihren Eigenschaften definiert. Über eine Spezielle Konfigurationssoftware wird aus der OIL-File der OSEK Code erstellt (s.h. Abb. 2).

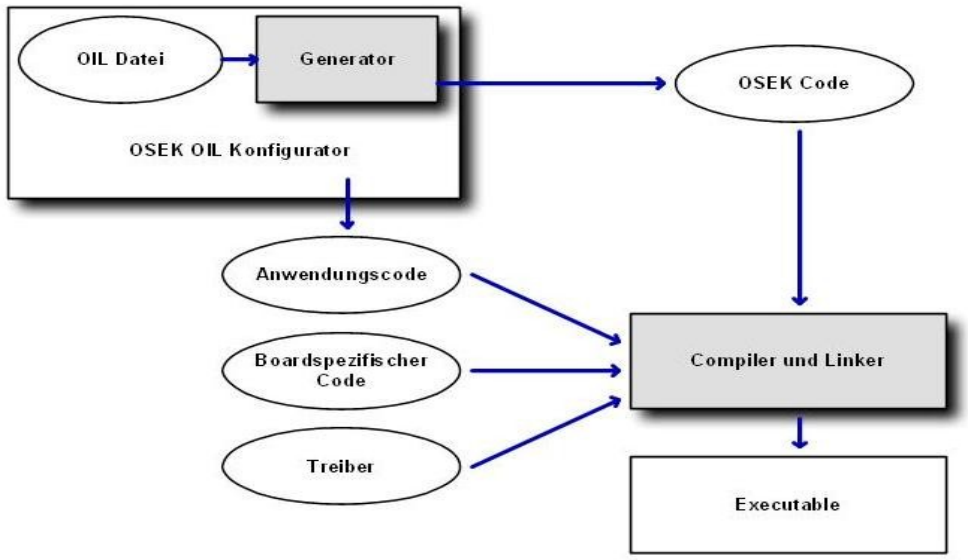


Abbildung 2: Erzeugung eines OSEK Projekts

1.2.2 Konformitätsklassen

Bei OSEK konformen Betriebssystemen wird darauf geachtet die Applikation so klein wie möglich zu halten, um die zur Verfügung stehenden Ressourcen (Rechenleistung, Speicher) optimal zu nutzen. Für manche Anwendungen sind bestimmte Funktionen die ein solches Betriebssystem anbietet nicht relevant. Aus diesem Grund gibt es die Konformitätsklassen mit denen das Betriebssystem angepasst werden kann. OSEK bietet gleich vier verschiedenen Konformitätsklassen an.

- BCC1 Basic Conformance Class 1
- BCC2 Basic Conformance Class 2
- ECC1 Extended Conformance Class 1
- ECC2 Extended Conformance Class 2

Abb. 3 zeigt die Unterschiede der einzelnen Konformitätsklassen.

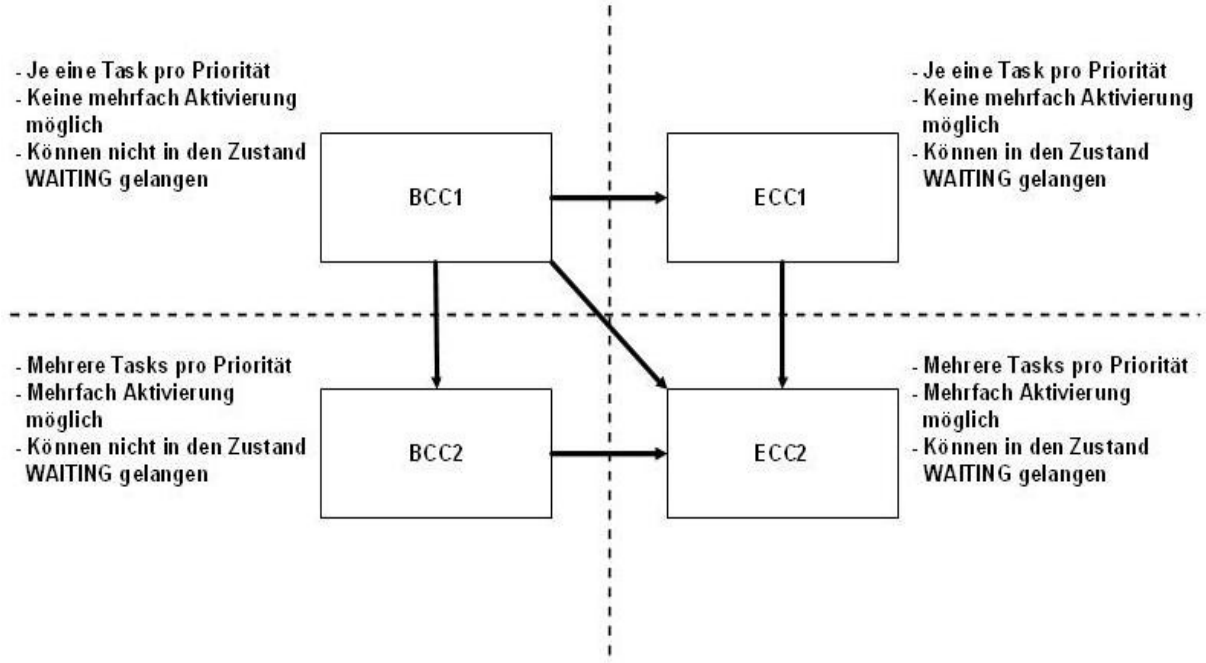


Abbildung 3: Konformitätsklassen

1.2.3 Task-Verwaltung

Der eigentliche Anwendungscode wird in einzelne Tasks geschrieben. Diese Tasks werden nun quasi parallel zueinander abgearbeitet. Das hat wiederum zur Folge, dass die Tasks um verfügbare Rechenzeit konkurrieren. Die eigentliche Aufgabe des Betriebssystems besteht nun darin, die Anwendung zu organisieren und sie zu strukturieren. Jede einzelne Task besitzt immer einen bestimmten Zustand in dem sie sich befindet. Je nach dem was derzeit im RTOS passiert wechselt der Zustand der jeweiligen Task.

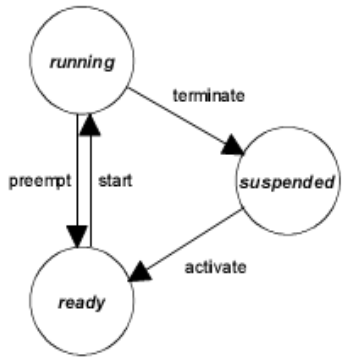


Abbildung 4: Basic-Task

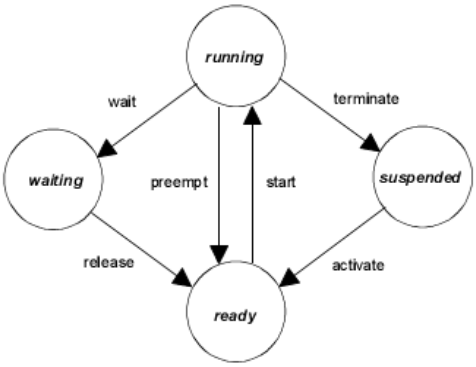


Abbildung 5: Extended-Task

In einem OSEK konformen Betriebssystem wird zwischen zwei verschiedenen Taskmodellen unterschieden. Einmal die Basic Task und zum anderen die Extended Task.

Basic-Task

Die Basic Tasks können nur drei Zustände einnehmen:

Suspended (inaktiv)
Ready (bereit)
Running (laufend)

Das hat zur Folge, dass eine Task zwar unterbrochen werden kann, jedoch nicht auf ein Ereignis (Event) warten kann. Also sind in diesem Modell das Betriebssystemmittel der Events nicht anwendbar.

Extended Task

Die Extended Tasks können vier Zustände einnehmen:

Suspended (inaktiv)
Ready (bereit)
Running (laufend)
Waiting (wartend)

Hier ist auch der Zustand Waiting möglich. In diesem Modell können alle Betriebssystemmittel verwendet werden.

1.3 Kommerzielle OSEK Distributionen

Es sind etliche OSEK konforme Betriebssysteme auf dem Markt erhältlich.
Wie z.B.:

- osCAN von der Firma Vector-Informatik
- RTA-OSEK von der Firma ETAS
- proOSEK von der Firma 3SOFT

Kurze Beschreibung von osCAN und RTA-OSEK

osCAN wurde von der Firma Vector-Informatik entwickelt. Es handelt sich dabei um ein prioritätsbasierendes, preemptives Echtzeitbetriebssystem für Kfz- Steuersysteme welches die OSEK Spezifikation vollständig erfüllt. osCAN unterstützt alle vier Konformitätsklassen, demzufolge werden Basic-, so wie auch Extended Task unterstützt. Der Scheduler kann als nicht preemptiver Scheduler, als preemptiver Scheduler oder auch im Mixed-Mode (preemptiver und nicht preemptiver) arbeiten.
D.h. es wird zwischen den einzelnen Tasks unterschieden ob diese unterbrechbar oder nicht unterbrechbar sind. Aufgrund der langjährigen Erfahrung und engen Zusammenarbeit mit den verschiedenen Automobilherstellen, zählt osCAN somit zu den verbreitetsten Distributionen. Neben dem Betriebssystem werden eine große Anzahl Treiber und Portierungen, Protokolle sowie Entwicklungsumgebungen angeboten, welche dem Kunden eine gute Komplettlösung zum Thema Softwareentwicklung in Kfz-Systemen bietet.

RTA-OSEK wurde von der ETAS Gruppe entwickelt. Es ist ein preemptives Echtzeitbetriebssystem welches die Standarte der OSEK Spezifikation sowie auch der AUTOSAR-OS Spezifikation implementiert. Neben RTA-OSEK werden weitere Softwarepakte mit angeboten, welche bei der Entwicklung von Software für Steuergeräte behilflich sein sollen.

Neben den beiden vorgestellten Betriebssystemen gibt es noch sehr viele andere OSEK Distributionen die kostenpflichtig sind.

1.4 HSE Free OSEK

Das Thema dieser Studienarbeit ist es ein portables, prioritätsbasierendes und preemptives Echtzeitbetriebssystem zu entwickeln, welches die Standarte der OSEK Spezifikation implementiert. Das RTOS bekam den Namen „HSE Free OSEK“ aus dem Grund, da die Studienarbeit an der Hochschule Esslingen durchgeführt worden ist und es eine kostenlose Alternative zu den in Kapitel 1.3 genannten Betriebssystemen sein soll. Die Besonderheit an HSE Free OSEK ist nicht nur das es kostenlos verfügbar ist, sondern eben auch dass der Sourcecode des RTOS für jeden zugänglich ist und diese Dokumentation welche für das Verstehen , Verwenden und Weiterentwickeln behilflich sein soll. Es eignet sich besonders gut für Studenten die sich mit hardwarenaher Programmierung befassen und sich dafür interessieren wie ein Betriebssystem wirklich funktioniert. Die folgenden Kapitel werden versuchen den Sachverhalt genauer zu beschreiben jedoch ist eine gewisses Grundwissen über Betriebssystemelemente notwendig.

1.5 Hardware

1.5.1 Freescale HCS12

Das HSE Free OSEK Betriebssystem wurde auf einem Dragon12 Evaluation-Board entwickelt, welches mit einem Motorola Freescale MC9S12DP 256B Mikrocontroller bestückt ist.

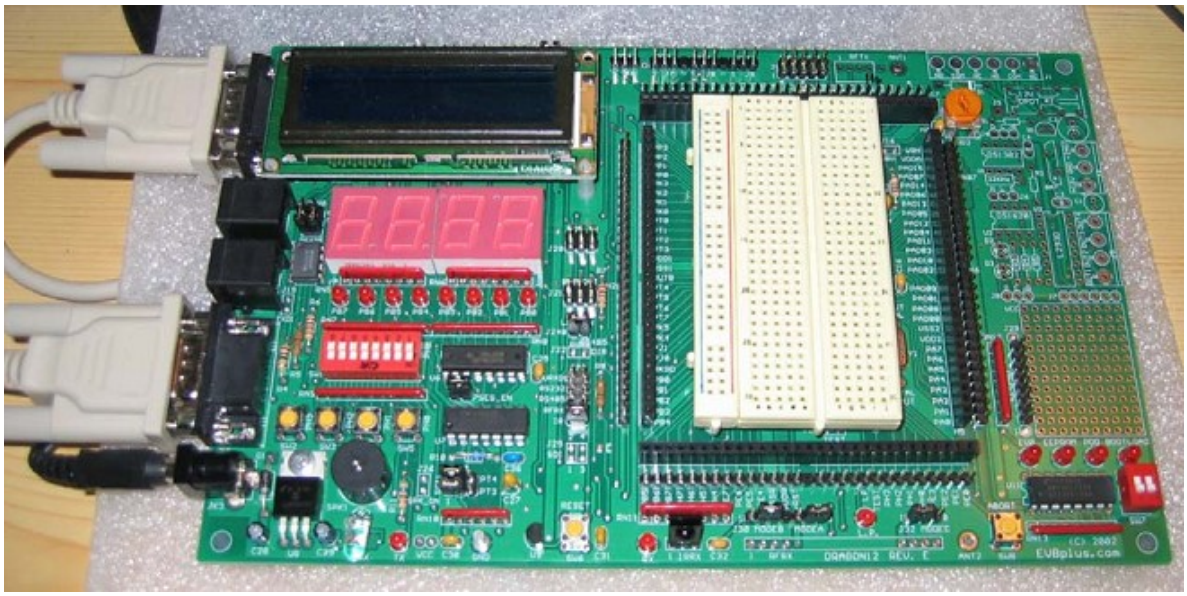


Abbildung 6: Dragon12 Evaluationboard

Motorola Freescale MC9S12DP 256B

- 16-Bit HCS12 CPU
- 256K Flash EEPROM
- 4K Byte EEPROM
- 12K Byte RAM
- zwei 16-Bit Index-Register X und Y
- einen 16-Bit Stack-Zeiger (Stack Pointer SP)
- einen 16-Bit Instruktionszeiger (Progam Counter PC)
- ein 8-Bit Statusregister (Condition Code Register CCR)

Dragon12 Evaluationsboard

- zwei serielle Schnittstellen
- ein 16*2 LCD Display
- acht LEDs (über Port B angeschlossen)
- vier Buttons (über Port H angeschlossen)
- eine sieben Segment Anzeige
- MSCAN Anschlüsse

HCS12 Serial Monitor

Der Mikrocontroller besitzt einen HCS12 Serial Monitor welcher 23 Debug Kommandos unterstützt, FLASH/EEPROM Programmierung erlaubt und das Debuggen über die RS-232 serielle Schnittstelle ermöglicht. Somit ist es auch möglich einen Reset auf der Ziel MCU durchzuführen, den Speicher und die CPU Register zu lesen und zu verändert.

ACHTUNG!!!

Der Serial Monitor und das Betriebssystem HSE Free OSEK arbeiten mit dem selben Software Interrupt (SWI),aus diesem Grund kann es zu einem Konflikt kommen.
Das selbe Problem ist bereits bei einer Studienarbeit vorgekommen, welche sich mit der Portierung von µC/OS II auf das HCS12 Derivat beschäftigt hat. Die Lösung des Problems, wird in der Studienarbeit von Alexander Hess beschrieben und sieht folgendermaßen aus:

```
#define  __ContextSwitch() __asm  jsr ContextSwitch;

#define  __SingleCtxSw()   __asm   jsr SingleCtxSw;

#define __INTCTXSW()    __asm   jsr INTCTXSW;
```

1.5.2 NXP (Philips) LPC2148

Das HSE Free OSEK Betriebssystem wurde auf einem OLIMEX Development-Board entwickelt, welches mit einem NXP (Philips) LPC2148 Mikrocontroller bestückt ist.

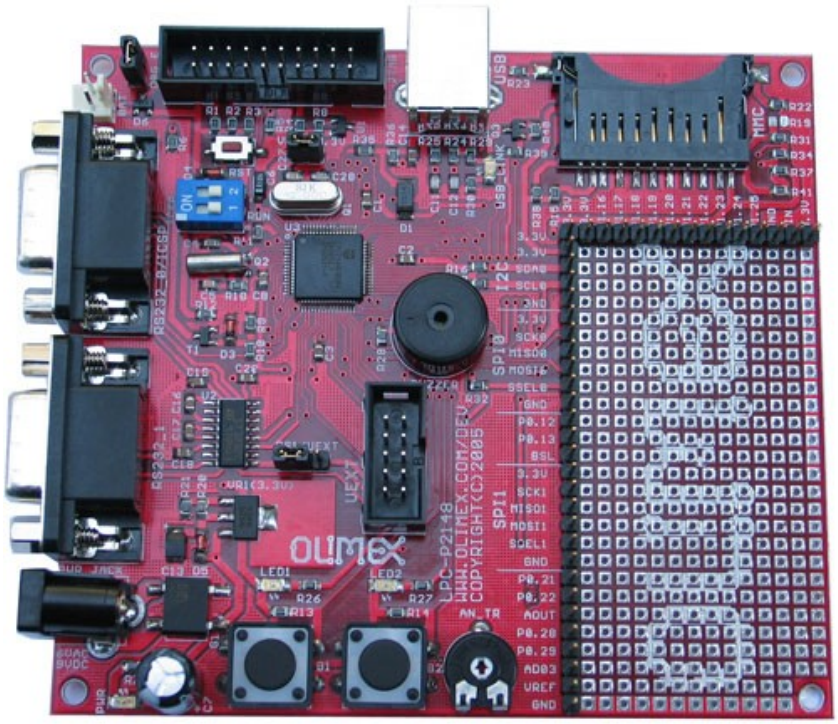


Abbildung 7: OLIMEX LPC2148 Development-Board

Technische Daten:

NXP (Philips) LPC2148

- 16/32 -Bit Mikrocontroller mit 512 kByte Flash-Speicher, 42kByte RAM
- Taktrate bis zu 60 MHz
- USB 2.0 Unterstützung, USB Schnittstelle, USB LED
- SPI, 2xUART
- SD/MMC Karten-Schnittstelle (über SPI ansteuerbar)
- 2 Knöpfen
- 2 LED's
- 1 Buzzer
- 12 Mhz Quartz
- Real Time Clock (RTC), 2 x 32bit TIMERS
- 6 x PWM

Der verwendete Mikrocontroller, sowie die Peripherie welches das Development-Board mit sich bringt, sind eigentlich nicht typisch in der Entwicklung von Kfz-Steuergeräten. Die OSEK Spezifikation wurde zwar für den Bereich Kfz-Entwicklung erstellt, jedoch ist ein OSEK Betriebssystem portabel und für die meisten Embedded-Anwendungen geeignet.

Keil ULINK Debugger

Um HSE Free OSEK auf dem LPC2148 OLIMEX Board zu testen und zu debuggen, wurde der Keil ULINK USB-JTAG Adapter verwendet. Dieser Adapter kann verwendet werden um alle ARM Derivate, welche über eine JTAG Schnittstelle verfügen, während der Laufzeit zu Debuggen und um den Flash-Speicher zu beschreiben.

Abbildung 8: KEIL ULINK USB-JTAG Adapter

1.6 Entwicklungsumgebungen

Da man das HSE Free OSEK auf zwei verschiedenen Prozessortypen entwickelt hat, wurde aus diesem Grund mit zwei verschiedenen Entwicklungsumgebungen gearbeitet. Für den Freescale Prozessor wurde die Entwicklungsumgebung von Metrowerks namens Codewarrior verwendet und für den LPC2148 (ARM) Prozessor die von der Firma Keil namens µVision.

1.6.1 Metrowerks Codewarrior

Die Entwicklungsumgebung CodeWarrior ist eine Integrierte Entwicklungsumgebung (IDE) der Firma Metrowerks, eines Tochterunternehmens von Motorola. Mit CodeWarrior kann man Software für viele Anwendungsbereiche vom eingebetteten System bis hin zur Desktopanwendung erstellen.
Der CodeWarrior bietet die Möglichkeit mit ein und derselben Entwicklungsumgebung in 4 verschiedenen Programmiersprachen (Java, C, C++ und Pascal) Programme zu schreiben. Diese lassen sich dann für 11 unterschiedliche Plattformen (Windows 95/ NT, Mac OS, Mac OS X, BeOS, PlayStation, Palm Pilot u.a.) kompilieren. Weiterhin ist er auf drei verschiedenen Betriebssystemen (MacOS, BeOS und Windows 95/NT) ausführbar, die auch noch mit einer einheitlichen Benutzeroberfläche verfügbar ist.
Die große Auswahl an Plattformen erkauft man sich allerdings mit fehlender Spezialisierung: So besitzt der CodeWarrior keinen eigenen Programmteil zur Erstellung von Benutzeroberflächen (GUI-Builder), da die GUIs der Plattformen alle unterschiedlich sind. Der CodeWarrior ist auch ein beliebtes Entwicklungswerkzeug für Macintosh-Applikationen gewesen. Da CodeWarrior keine Intel-Binaries erzeugen kann, sind Entwickler künftig auf Alternativen wie Xcode gezwungen.

HSE Free OSEK wurde in der Codewarrior Version 5.5.1272 erstellt welche den Default-Compiler „ANSI-C/cC++ Compiler for HC12 V-5.0.24 Build 4047“ sowie den Default-Linker „SmartLinker V-5.0.22 Build 4047“ verwendet.

1.6.2 Keil µVision

Keil Software ist führender Anbieter für 8051, 251, C166 und ARM Mikrocontroller Development Tools. Alle Tools sind in µVision3 integriert. µVision3 ist eine IDE der zweiten Generation, die Project Manager, Source Code Editor und Programm Debugger in einer leistungsfähigen Umgebung kombiniert. Die µVision3 IDE integriert Compiler, Assembler, Real-Time OS, Project Verwaltung und Debugger in einem intelligenten Environment. µVision3 integriert außerdem Utilities und enthält Schnittstellen zu Third-party Development Tools. Zusätzlich bietet die Firma Keil zu fast jedem unterstützen Mikrocontroller sehr umfangreiche Codebeispiele zur Verwendung der Peripherie sowie eine weiter Vielzahl von Produkten welche das Entwickeln von Embedded Software vereinfachen.

HSE Free OSEK wurde in der µVision3 Version 3.3.4 erstellt welche den Default-Compiler „CARM V2.40c“ sowie den Default-Linker „CARM Linker V2.40c“ verwendet.

2 Einstieg mit HSE Free OSEK

2.1 Installation der Arbeitsumgebung unter Metrowerks Codewarrior

Um das HSE Free OSEK anwenden zu können, ist es zuerst notwendig den Codewarrior dem OSEK Projekt anzupassen. Auf der CD, welche der Dokumentation beigefügt ist, befindet sich ein Verzeichnis namens HSE Free OSEK V0.1. Dieses Verzeichnis ist für die nachfolgenden Schritte relevant. Um Konflikte zu vermeiden wird empfohlen mit der Codewarrior IDE Version 5.5.1272 zu arbeiten. Standardmäßig arbeitet die IDE mit dem ANSI-C/cC++ Compiler for HCS12 V-5.0.24 (Build 4047) und dem SmartLinker V-5.0.22.

Zuerst wird ein neues Projekt erzeugt. Dazu startet man den Codewarrior und wählt zuerst „***File → New***“. Als nächstes erscheint einem ein Fenster in dem man den Projektnamen und den Arbeitsordner wählt. Im hier dargestellten Beispiel nennt sich das Projekt „***HSE Free OSEK HCS12***“. Es ist noch die Option „***HC(S)12 New Project Wizard***“ zu wählen und das ganze zu bestätigen.

Abbildung 9: Projekt erzeugen1

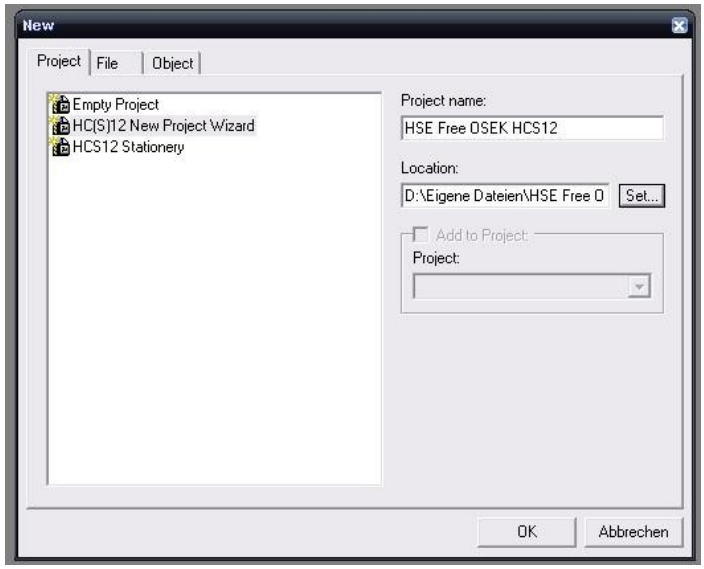


Abbildung 10: Projekt erzeugen 2

Als nächstes wird das verwendete Derivat ausgewählt. Das HSE Free OSEK ist zum momentanen Zeitpunkt nur für 2 Derivate einsetzbar. In diesem Fall ist das „***MC9S12DP256B***“ Derivat zu wählen.

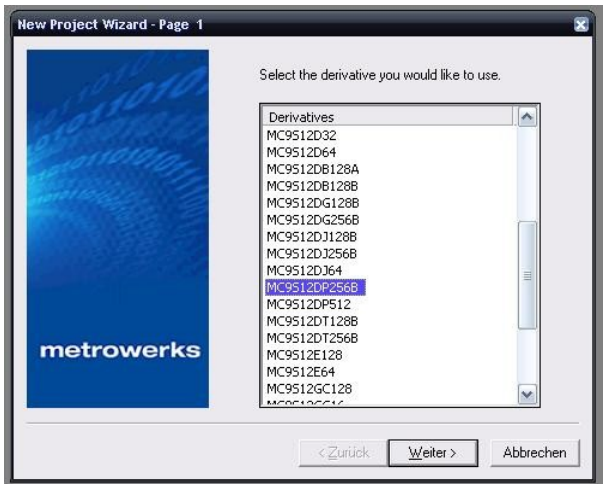


Abbildung 11: Projekt erzeugen 3

Im nächsten Fenster wählt man die Programmiersprache C. Die darauf folgenden Fenster sind mit ***NONE*** oder ***NO*** zu bestätigen, bis das Fenster erscheint in welchem man das MEMORY MODEL auswählen kann. Es ist wichtig das man an dieser stelle „***SMALL***“ wählt. Als nächsten folgt die Wahl mit welchem Debugger man arbeitet. In diesem Fall wurde mit dem „***Full Chip Simulato***r“ und dem „***Serial Monitor***“ gearbeitet.

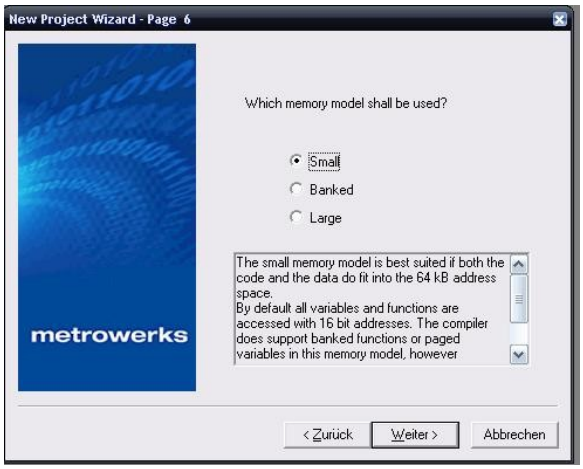


Abbildung 12: Projekt erzeugen 4

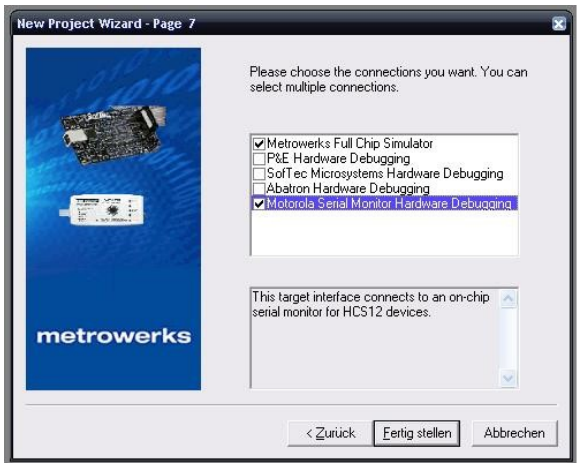


Abbildung 13: Projekt erzeugen 5

Nun hat man die Arbeitsumgebung eingestellt. Als nächstes folgt das Einfügen des HSE Free OSEK in das neu erstellte Projekt. Dazu ist es notwendig einige vom Projektwizard erstellte Ordner Im Projekt zu löschen.
Dazu gehören folgende Ordner: „Sources, Startup Code und Libraries“.

Die Dateien der 3 gelöschten Ordner sind im Betriebssystem ebenfalls in geordneter Form vorhanden. Dies geschieht aus dem Grund um mit höheren Versionen vom Codewarrior eine Kompatibilität zu gewährleisten, falls sich die Files vom Hersteller Metrowerks ändern sollten.

Der nächste Schritt wäre es folgende Ordner im Projekt hinzuzufügen:

OSEK Code, OSEK Hardware Specific, Application, Hardware, Interrupt Code.

Zum Schluss sollte das Projekt so aussehen wie in Abb. 14.

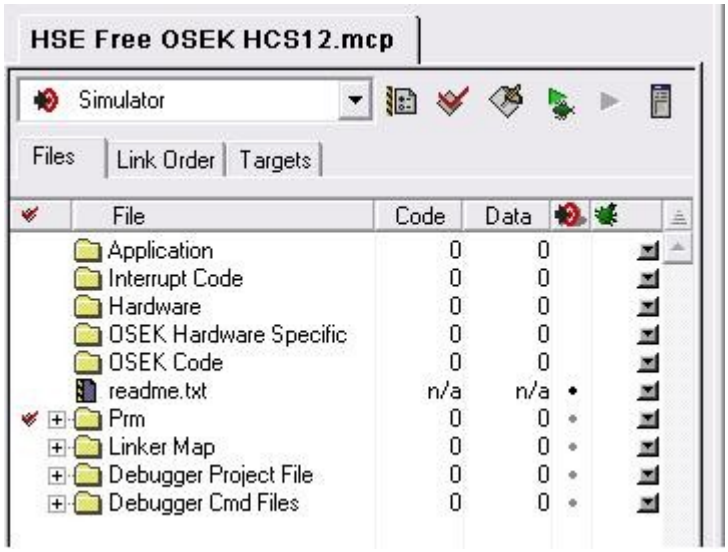


Abbildung 14: HSE OSEK Implementieren 1

Als nächstes kopiert man den kompletten Inhalt für den HCS12 aus dem Ordner HSE Free OSEK V0.1 in sein Projektverzeichnis in den Unterordner „../Sources“. Zu beachten ist, dass jeder Ordner mehrere Unterordner besitzt, welche für die einzelnen Derivate vorhanden sind. Beim diesem Vorgang ist zu beachten, dass die Verzeichnisse den selben Namen besitzen wie die manuell erstellten Ordner im Codewarrior Projekt. Es ist nun auch notwendig die kopierten Dateien dem Projekt zuzuweisen. Letztendlich sollte sich folgende Projektstruktur ergeben (s.h. Abb. 15):

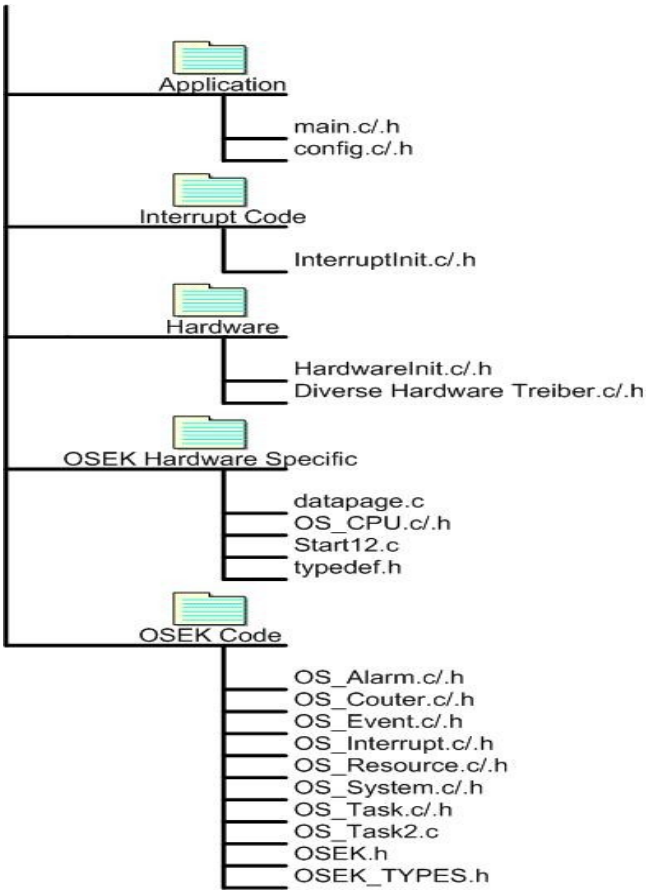


Abbildung 15: OSEK Verzeichnisbaum

Nun hat man alle Schritte durchgeführt welche zur Erstellung des ersten Projekts benötigt werden. Falls man sich die Arbeit ersparen möchte, findet man auf der CD im Ordner CD:\\DefaultProject\HCS12 ein fertiges Projekt welches man nur kopieren muss und die MCP-Projektdatei öffnet.

Damit die Zeitgeber (OS Counter) von HSE Free OSEK aktiviert werden ist es notwendig die Interrupt Service Routine ISROSTimer der entsprechenden Adresse zuzuweisen. Dies geschieht durch den folgenden Eintrag in den PRM-Files Simulator_Linker.prm und Monitor_Linker.prm:

VECTOR ADDRESS 0xFFEE ISROSTimer.

```
STACKSIZE 0x100

VECTOR 0 _Startup /* reset vector: this is the default entry point for a C/C++ application. */
VECTOR ADDRESS 0xFFEE ISROSTimer
//VECTOR 0 Entry  /* reset vector: this is the default entry point for a Assembly application.
//INIT Entry      /* for assembly applications: that this is as well the initialisation entry
```

Abbildung 16: OSTimer Interrupt bekannt machen

Somit ist die Erzeugung des Projekts abgeschlossen und man kann seine Applikation erstellen. Ein Testdurchlauf ohne Fehlermeldung sollte als Bestätigung gelten, dass die Projekterzeugung erfolgreich war.

2.2 Installation der Arbeitsumgebung unter Keil µVision

Wie bei Codewarrior wird hier ebenfalls zuerst das Projekt erzeugt. Dies geschieht indem man ***Project→New Project*** auswählt. Als nächstes erscheint einem ein Fenster in dem man den Projektnamen und den Arbeitsordner wählt. Im hier dargestellten Beispiel nennt sich das Projekt „***HSE Free OSEK LPC2148***“. Darauf erscheint ein Fenster in welchem der Prozessor zu wählen ist. Dazu wählt man den Hersteller NXP und das Modell LPC2148. Früher wurde die LPC Serie von Philips direkt vertrieben und entwickelt, also kann es bei älteren Versionen von µVision sein, dass der LPC2148 im Herstellerordner Philips zu finden ist.

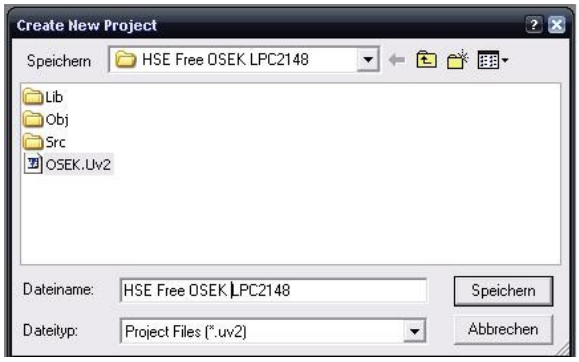


Abbildung 17: Projekt erzeugen 1

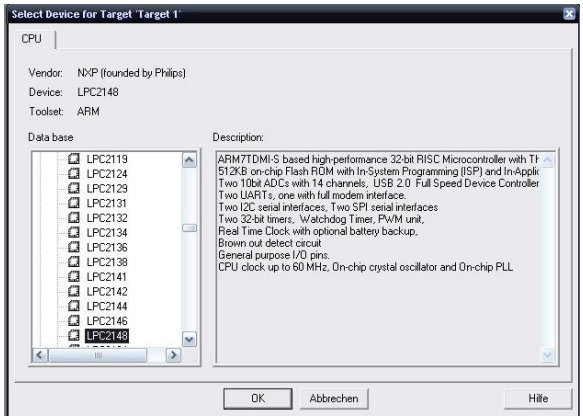


Abbildung 18: Projekt erzeugen 2

Die folgende Abfrage ob die Startup File mit einzufügen ist muss mit „Nein“ beantwortet werde. Nun hat man die Arbeitsumgebung eingestellt.

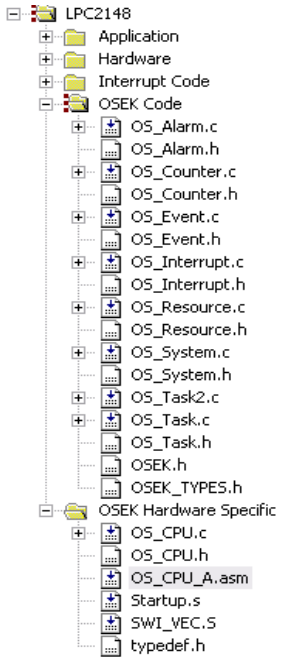


Abbildung 19: µVision Verzeichnisbaum

Der nächste Schritt wäre es folgende Ordner im Projekt zu erstellen (s.h. Abb. 19):

OSEK Code,
OSEK Hardware Specific,
Application,
Hardware und
Interrupt Code.

Als nächstes kopiert man den kompletten Inhalt für den HCS12 aus dem Ordner HSE Free OSEK V0.1 in sein Projektverzeichnis in den Unterordner „../Sources“. Zu beachten ist, dass jeder Ordner mehrere Unterordner besitzt, welche für die einzelnen Derivate vorhanden sind.

Es ist nun notwendig die kopierten Dateien dem Projekt zuzuweisen.

Als nächstes sind einige Projekteigenschaften einzustellen. Dies geschieht indem man ***Project→Options for Target*** auswählt. Es erscheint ein Fenster wo mehreren Registerkarten auswählbar sind.

Zunächst wählt man die ***Registerkarte Ouput*** und darunter den Ordner für die Objektdateien. In diesem Beispiel würde sich im Projektverzeichnis der Ordner „Obj“ anbieten.
Das selbe gilt für die ***Registerkarte Listing***. Hier würde sich der Ordner ***Lst*** anbieten.

Im Vergleich zur Codewarrior IDE, ist das Einfügen der Projektdateien nicht ausreichend um den Linker die Verzeichnisse in denen er suchen muss bekannt zu geben. Diese müssen ebenfalls noch manuell vom User eingetragen werden. Dies geschieht in der ***Registerkarte C***. Unter dem Punkt ***Include Paths*** müssen die selben Ordner nochmal bekannte gemacht werden.

Wichtig ist es in der ***Registerkarte C*** das Häckchen bei ***USE THUMB-Mode*** zu deaktivieren. Da das HSE Free OSEK nur den 32 Bit ARM-Mode unterstützt, würde es im Thumb Mode zu unerwünschten Fehlermeldungen kommen. Weitere Informationen zu den Betriebsmodies und deren Eigenschaften finden man in den Datenblättern des LPC2148 Prozessors.

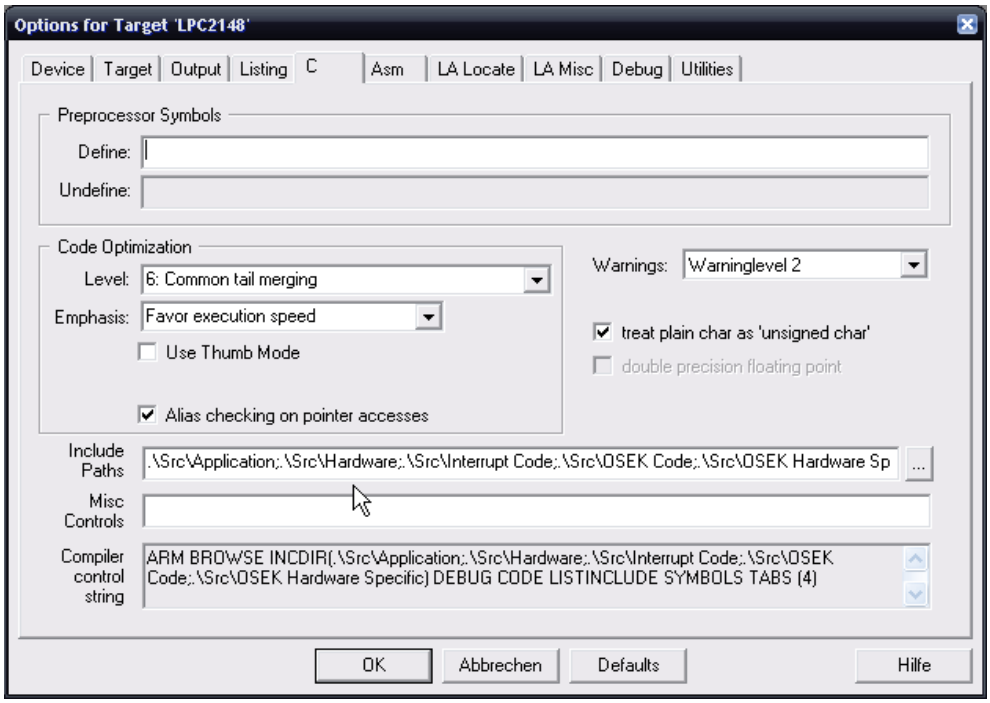


Abbildung 20: Einstellung des C Compilers und Linkers

Da auch die Assembler-Datei OS_CPU_A.asm im Ordner „***..Src/OSEK HardwareSpecific/***“ existiert, muss diese ebenfalls dem Linker bekannt gemacht werden. Dieser Eintrag ist unter der ***Registerkarte ASM*** unter dem Punkt ***Include Paths*** einzutragen.

Zum Schluss sollte man sich nur noch vergewissern ob auch wirklich der CARM Compiler und Linker verwendet werden.

3 Aufbau des Kernels

3.1 Kritische Regionen

Wie bei allen Echtzeitbetriebssystemen müssen auch bei HSE Free OSEK Interrupts gesperrt und danach wieder freigegeben werden, wenn man in kritische Regionen eintritt. Es muss verhindert werden, dass während der Ausführung von System kritischen Code, Interrupts oder Tasks diesen Vorgang unterbrechen. Dabei ist zu beachten das die Zeit in der die Interrupts unterbrochen werden sehr kurz sein muss, da sonst die Echtzeitfähigkeit des Betriebssystems beeinträchtigt wird.

HSE Free OSEK bietet dazu bisher zwei API Funktionen und eine System Funktion um Interrupts zu sperren bzw. freizugeben. Das sind zum einen die OSEK konformen Funktionen
EnableAllInterrupts() / ***DisableAllInterrupts()*** und
ResumeAllInterrupts() / ***SuspendAllInterrupts()***
und zum anderen die Systemfunktionen
SaveSR() / ***RestoreSR()***

Die OSEK Spezifikation gibt für die Interrupt Funktionen vor, dass in einem kritischen Bereich keine API Funktion aufgerufen werden darf. Die Aufgabe der OSEK API Funktionen besteht im wesentlichen darin, dass Interrupt Flag Bit im Prozessorstatuswort zu setzen bzw. zu löschen.
Da es nun in den API Funktionen zu einem Context Switch kommen kann und dieser ebenfalls vor Unterbrechungen von Interrupts geschützt werden muss, hat man sich bei HSE Free OSEK entschlossen das Prozessorstatuswort lokal in der Task zu sichern. Der Vorteil ist das man das Sperren der Interrupts verschachteln kann und man, unabhängig von den anderen Tasks die im System laufen, für jede einzelne Task die Interrupts sperren oder freigeben kann.
Aus diesem Grund verwendet HSE Free OSEK eigene Systemfunktionen (***SaveSR()*** / ***RestoreSR()***), welche nicht zur API gehören, um einen fehlerfreien Ablauf zu gewährleisten. Diese Funktionen speichern das gesamte Prozessorstatuswort für jeden einzelnen Aufruf in einer Variablen ab. Somit wird nach jedem RestorSR das jeweilige PSW wiederhergestellt.

```
{
        ...
        PSW = SaveSR();
        /* Bereich für System kritischen Code */
        RestoreSR(PSW);
        ...
}
```

Die OSEK spezifischen Interrupt Funktionen rufen alle die Funktion ***__EnAllInt()*** bzw. ***__DisAllInt()*** auf. Hierbei handelt es sich um einen so genannten Wrapper, eine Schnittstelle zwischen HSE Free OSEK und dem Prozessor spezifischen Code. Damit soll eine Kompatibilität zwischen verschiedenen Prozessor Architekturen gewährleisten werden.

3.2 Tasks

Eine Task ist im Grunde das gleiche wie eine Funktion mit einer Endlosschleife.

```
void TaskName(void)
{
        while(1)
        {
                /*Anwendungs Spezifischer Code*/
        {
{
```

Eine Task besitzt nie einen Rückgabewert. Es besteht zwar generell die Möglichkeit einer Task einen Wert zu übergeben, jedoch nicht in einem OSEK konformen Betriebssystem. Durch bestimmte defines wurde festgelegt wie der Funktionskopf einer Task auszusehen hat.

```
TASK(TaskName)
{
        /*Anwendungs Code*/
}
```

Es besteht auch die Möglichkeit mehrere Tasks zu erzeugen, die alle unterschiedliche ID's und demzufolge auch jeder seinen eigenen Task Control Block besitzt (s.h. 3.4), jedoch alle Tasks sich ein und die selbe Task Routine teilen. Die Tasks müssen nur alle den gleichen Namen haben.
Es ist zu beachten das sich eine Task am Ende immer selber beenden muss. Das geschieht mit der Funktion ***TerminateTask()*** (s.h. 4.3 Beenden einer Task). Wenn das nicht geschieht, kann die Task nicht mehr gestartet werden. Eine Task kann auch von einer anderen, höher Prioren Task, unterbrochen werden. Je nach dem ob es sich hierbei um eine unterbrechbare Task handelt oder nicht.
HSE Free OSEK besitzt noch zwei System Tasks (s.h. 3.11 Betriebssystem Tasks) die bereits fest im Betriebssystem implementiert sind. Sie besitzen einmal die niedrigste Priorität (IdleTask) und die höchste Priorität (KernelTask).

3.3 Taskzustände unter OSEK

In einem OSEK konformen Betriebssystem kann es bis zu vier verschiedene Zustände geben die eine Task einnehmen kann. Dabei wird grundsätzlich zwischen Basic- und Extended Mode unterschieden. Im Basic Mode gibt es drei Zustände und im Extended Mode sind es vier (s.h. 1.2.3 Task-Verwaltung).

3.4 Der Task Control Block

Der wesentliche Unterschied zwischen sequenziellen Programmen und Multitasking ist, dass ein Programm jederzeit von einem anderen unterbrochen werden kann und nach einer bestimmten Zeit an der Stelle der Unterbrechung fortgesetzt wird. Bei der sequenziellen Programmierung ist dies nur durch Interrupts möglich, die durch ein Ereignis z.B. Timer Overflows oder das Empfangen eines Bytes über die UART, ausgelöst werden kann. Diese Art von Unterbrechung wird von der Hardware ausgelöst und man spricht von einem Foreground/Background System.

In einem Multitasking System existieren mehrere Prozesse die scheinbar parallel zueinander laufen. In Wirklichkeit unterbrechen sich die einzelnen Prozesse ständig und sind auch teilweise abhängig voneinander. Beispielsweise könnte ein Prozess auf eine Berechnung warten für die ein anderer Prozess zuständig ist. Um dies zu ermöglichen ist es notwendig jedem einzelnen Prozess eine übergeordnete Struktur zuzuweisen, welche alle relevanten Daten des Prozesses kennt. Diese wären z.B. die Position im Speicher, die Priorität oder auch welche Betriebssystemmittel der Prozess verwendet.
Allgemein gilt das jedes Betriebssystemmittel, welches eine Unterbrechung eines Prozesses auslösen kann, einer Struktur zugewiesen wird. Diese Art von Strukturen nennt man Control Block auf die nun näher eingegangen wird.

In der OSEK Spezifikation sind alle Kontrollstrukturen eindeutig definiert. Als Anwender beschäftigt man sich letztendlich dann nur mit der Initialisierung der Kontrollstrukturen was in der OIL-File geschieht. Die OIL-Files ist eine normierte Darstellung der Initialisierung von Kontrollblöcken die für alle OSEK Distributionen gilt. Meistens erhält der Anwender vom Distributor eine GUI-Software mit welcher er alle Betriebssystemmittel definiert und initialisiert. Da es für das HSE Free OSEK keine OIL-File Generator zur Verfügung stand, findet die Initialisierung in den Configuration-Files statt.

Der Task Control Block (TCB) ist die Kontrollstruktur in welcher die Eigenschaften der einzelnen Task beschrieben werden.

```
typedef struct os_tcb
   {
    //Task Stack
    OS_STK   *TCBStackPtr;          //akt. Stack Adresse;                    (1)
    OS_STK   *TCBTaskStack;         //Stack Start                            (2)

    #ifdef LPC2148
      OS_STK   *TCBContextStack;  //CTX Stack Start                          (3)
    #endif

    void *TCBTaskAdr;               //Funktionsadresse                       (4)

    INT8U TCBStat;                  //Task Zustand                           (5)
    INT8U TCBPrio;                  //Priorität                              (6)
    INT8U TCBStdPrio;
    INT8U TCBId;                    //Task ID                                (7)

    //Conformitätsklasse (BCC1,BCC2,ECC1,ECC2)
    INT8U TCBConfClass;                                                      (8)

    INT8U TCBScheduled;                                                      (9)

    //Events
    struct os_event TCBEvent;  //anliegendes Event                          (11)
   //angabe auf welches Event gewartet wird
    struct os_event TCBWaitEvent;

    //verkettung zu anderen TCB's
    struct os_tcb *nextTCB;   //Pointer auf folgende Task                   (12)
    struct os_tcb *prevTCB;   //Pointer auf vorherige Task

    //Wenn Bitmasken-Scheduling aktiv
    #if (SCHEDULER == 1 )                                                    (13)

      INT8U TCB_Y;
      INT8U TCB_BIT_Y;
      INT8U TCB_X;
      INT8U TCB_BIT_X;

    #endif

   }OS_TCB;
```

Erklärung zu TCB:

(1): **TCBStackPtr** beinhaltet die aktuelle Adresse des Stackpointers (SP) einer Task. In HSE Free OSEK besitzt jede Task ihren eigenen Stack der vom Anwender angelegt wird. Beim erzeugen einer Task zeigt **TCBStackPtr** auf die Startadresse des Stacks und bei einer Unterbrechung wird die aktuelle SP-Adresse hier gespeichert.

(2): **TCBTaskStack** beinhaltet die Startadresse des Stacks. Wenn eine Task beendet wird soll sie beim Wiederaufruf wissen wo der Stack anfängt.

(3): **TCBContextStack** ist eine Erweiterung auf die erst später genauer eingegangen wird. Im allgemeinen handelt es sich um einen Zeiger auf einen Container welche Prozessorspezifische Daten beinhaltet.

(4): **TCBTaskAdr** beinhaltet die Startadresse einer Task. Wenn eine Task aufgerufen wird ist dies die Adresse wo nach einem ContextSwitch hin gesprungen wird.

(5): **TCBStat** beinhaltet den Status einer Task (Running,Suspended,Wait,Ready)

(6): **TCBPrio** gibt Auskunft über die Priorität einer Task. Möglich sind die Prioritäten 1-62. In HSE Free OSEK ist es möglich das mehrere Tasks die selbe Priorität haben.

(7): **TCBId** ist der Task Identifier. Jede Task hat eine eindeutige ID (0-255).

(8): **TCBConfClass** beschreibt um was es für einen Typen es sich bei der Task handelt. Nach der OSEK Spezifikation wird zwischen Basic und Extended Task unterschieden.

(9): **TCBScheduled** beschreibt ob es sich um eine preemptive oder nicht preemptive Task handelt.

(10): **TCBEvent** ist eine Eventmaske welche beschreibt welche Events gerade anliegen.

(11): **TCBWaitEvent** ist eine Eventmaske welche beschreibt auf welche Events die Task wartet.

(12): **nextTCB/prevTCB** sind Zeiger auf weitere TCBs. HSE Free OSEK verwendet 2 Schedulingmechanismen in welchen die beiden Pointer verwendet werden. Alle Tasks die sich im Zustand Ready befinden liegen in einen der beiden Verfahren in der ReadyQueue und werden nach dem FIFO Prinzip in den Zustand Running versetzt. Beim 2. Scheduling verfahren ist diese Funktionalität identisch jedoch nur bei Tasks welche die selbe Priorität besitzen.

(13): **TCB_X-Y/BIT_X-Y** sind Bitmasken welche für das Bitmap-Scheduling relevant sind. Über die Priorität jeder einzelnen Task werden die Bitmasken dementsprechend gefüllt. Wenn sich eine Task im Zustand Ready befindet, wird ein Bit in einem zweidimensionales Array namens ReadyTable gesetzt. Die Position des gesetzten Bits in der ReadyTable wird über die Bitmasken bestimmt.

Da ein OSEK OS ein statisches Echtzeitbetriebssystem ist, müssen alle Tasks vornherein bekannt sein und die jeweiligen Attribute der TCBs von Anwender initialisiert werden. Dies geschieht in den config.c/.h Dateien. Alle Tasks werden unabhängig von ihrem Zustand in der TaskList[] aufgeführt. Der Index in der TaskList ist der Task eigene Identifier (TCBId). Der Anwender hat theoretisch die Möglichkeit 253 Tasks zu erzeugen wobei die ID 0 und ID 1 bereits von Betriebssystemtasks reserviert sind.

3.4.1 Initialisierung des Task Control Blocks

Im Projektverzeichnis des HSE Free OSEK Projekts findet man im Verzeichnis „../Src/Application/Configuration/“ die Dateien config.c und config.h. In diesem beiden Dateien findet die statische Initialisierung der Betriebssystemmittel statt. Zuerst wird Task Konfiguration in der config.h bearbeitet:

```
/*-----------------------------------------------------------------------*/
//TASK-KONFIGURATION
/*-----------------------------------------------------------------------*/

//Maximale Anzahl der Tasks ( IDLE + KERNEL TASK)
#define         MAX_NR_OF_TASKS       (1+2)                          (1)
// ID der einzelnen Tasks
#define         TASK1         2       //Task 1 beginnt mit der ID 2  (2)
/* Tasktype der einzelnen Tasks */
#define         TASK1_CONF_TYPE       BCC1                           (3)
/* Priorität der einzelnen Tasks */
#define         TASK1_PRIO            50                             (4)
/* STACK-Größen der einzelnen Tasks     */
#define         TASK1_STACK_SIZE      80                             (5)

/* Definition der Tasks */
TASK(TASK1);
```

(1) Die Konstante **MAX_NR_OF_TASKS** ist fest vorgegeben und muss mindestens den Wert 2 besitzen. Mit ihr gibt man an wie viele Tasks im System vorhanden sind. In diesem Beispiel wird für die Anwendung nur 1 Tasks erzeugt.

(2) Die Konstante TASK1 ist der Name der Task die man erzeugen möchte und bekommt eine eindeutige ID zugewiesen. Gültige ID's sind 2-255.

(3) Die Konstante TASK1_CONF_TYPE beschreibt die Konformitätsklasse der Task. Gültige Werte sind BCC1, BCC2, ECC1, ECC2.

(4) Die Konstante TASK1_PRIO definiert die Priorität der Task. Gültige Prioritäten sind 1 bis 62. Die Idle-Task besitzt die Prio 0 (minimale Priorität) und die Kernel Task die Prio 63.

(5) Da jede Task ihren eigenen Stack besitzt muss die Stackgröße angegeben werden. Dies geschieht mit der Konstanten TASK1_STACK_SIZE welche in diesem Fall 80 beträgt. Je nach dem wie breit der Stack ist kann die Gesamtgröße bei unterschiedlichen Controllern variieren. Bei einem 16 Bit Controller währen es 2*80 also 160 Byte und bei einem 32 Bit Controller 4*80 also 320 Byte.

(6) Zuletzt ist noch die Definition der Task selber anzugeben. Mit dem OSEK konformen Signalwort TASK(Taskname) wird bekannte gemacht das es sich um eine Task handelt.

Nachdem man nun die wichtigsten Attribute für den TCB einer Task festgelegt hat folgt nun die Initialisierung im System selber. In der Datei config.c ist die Funktion void TCBInit(void) zu finden, in welcher alle vom Anwender definierten Tasks initialisiert werden müssen. Später wird die Funktion vom Betriebssystem aus aufgerufen.

```
/*-----------------------------------------------------------------------*/
// Initialisierung der einzelnen TCB's
/*-----------------------------------------------------------------------*/
void TCBInit(void)
{
  TaskList[TASK1].TCBTaskAdr = (void *)Main2;                          (1)

#ifdef  LPC2148
  TaskList[TASK1].TCBStackPtr = &StackTask1CTX[CONTEXTSIZE-1];         (2)
  TaskList[TASK1].TCBContextStack = &StackTask1CTX[CONTEXTSIZE-1];
#else // HCS12
  TaskList[TASK1].TCBStackPtr = &(StackTask1[TASK1_STACK_SIZE-1]);     (3)
#endif

  TaskList[TASK1].TCBTaskStack    = &(StackTask1[TASK1_STACK_SIZE-1]); (4)

..TaskList[TASK1].TCBStat = READY;                                     (5)

  TaskList[TASK1].TCBPrio = TASK1_PRIO;                                (6)
  TaskList[TASK1].TCBStdPrio = TASK1_PRIO;

  TaskList[TASK1].TCBId   = TASK1;                                     (7)
  TaskList[TASK1].TCBConfClass = TASK1_CONF_TYPE;

  TaskList[TASK1].TCBScheduled = NONE_SCHEDULED;                       (8)

  TaskList[TASK1].nextTCB = NULL;                                      (9)
  TaskList[TASK1].prevTCB = NULL;

  TaskList[TASK1].TCBEvent.Mask  = EVENTMASK_NONE;                     (10)
  TaskList[TASK1].TCBWaitEvent.Mask |= EVENTMASK_NONE;

#if (SCHEDULER == 1 )                                                  (11)
  TaskList[TASK1].TCB_Y = TASK1_PRIO >> 3;
  TaskList[TASK1].TCB_BIT_Y = OSMapTbl[TaskList[TASK1].TCB_Y];
  TaskList[TASK1].TCB_X = TASK1_PRIO & 0x07;
  TaskList[TASK1].TCB_BIT_X  = OSMapTbl[TaskList[TASK1].TCB_X];
#endif

  TaskStackInit(&TaskList[TASK1]);                                     (12)
}
```

Der Zugriff auf die Taskattribute findet über die TaskList[] und den Task Namen statt. Die TaskList ist ein Array aus TCB's welches ebenfalls in der config.c erzeugt wird.

(1): Zuerst wird die Adresse des Codesegment in welchem sich die Task befindet dem TCB zugewiesen. In der config.h haben wir die Task(TASK1) deklariert. Zu einem späteren Zeitpunkt muss die Task irgendwo auf Anwendungsebene definiert werden ansonsten wird es zu Fehlern beim kompilieren kommen. Das Signalwort TASK(Taskname) ist ein von HSE Free OSEK definiertes Makro, welches eine Konkatenation aus MAIN und der TASKID liefert (MAIN##TASKID). Z.B. haben wir in der config.h den Tanknamen TASK1 die ID 2 zugewiesen. Durch das Makro Task(TASK1) wird immer die Konkatenation MainID gebildet. In diesem Fall Main2.

(2): Wenn man den LPC2148 Mikrocontroller HSE Free OSEK nutzt, ist der Stack vom Prozessorspezifischen Context getrennt. Normalerweise wird der Context auf dem Stack abgelagert und beim Wiedereintritt vom Stack geladen. Diese Lösung war beim LPC2148 nicht möglich doch dazu später mehr. Hier wird der Context-Speicher der zu der Task gehört zugewiesen.

(3): Wenn man den HCS12 Mikrocontroller nutzt, liegt der Context am Anfang auf der Spitze des Stacks. Also zeigt der SP der Task auf den Anfang des Stacks.

(4): **TCBTaskStack** beinhaltet die Startadresse des Stacks welche hier zugewiesen wird.

(5): Es wird der Task Zustand initialisiert welcher beim Starten von HSE Free OSEK vorliegt. Gültige Werte sind Suspended und Ready.

(6): Es folgt die Zuweisung der Task Priorität. Hier wird zwischen **TCBPrio** und **TCBStdPrio** unterschieden. **TCBPrio** ist die Priorität welche die Task im Moment besitzt. Die Priorität lässt sich zwar während der Laufzeit nicht mehr vom Anwender ändern, aber im Fall das eine Task eine Ressource in Anspruch nimmt, wird diese verändert. Bei Freigabe der Ressource wird diese mit Hilfe des Attributs **TCBStdPrio** die eigentliche Priorität wieder hergestellte.

(7): Es werden die ID und die Konformitätsklasse der Task festgelegt.

(8): **TCBScheduled** gibt Auskunft darüber ob es sich um eine voll preemptive (**FULL_SCHEDULED**) oder um eine nicht unterbrechbare (**NONE_SCHEDULED**) Task handelt. HSE Free OSEK ist so geschrieben das man sich ständig im Mixed Mode Scheduling befindet. D.h. in einem System kann es unterbrechbare sowie auch nicht unterbrechbare Task geben.

(9): Die beiden Pointer nextTCB und prevTCB werden nur vom Scheduler verwendet. Bei der Initialisierung sollten beide den Wert NULL haben. Sie dienen zur doppelten Verkettung der Tasks welche sich im Zustand Ready befinden.

(10): Zuweisung der Eventmasken. Beim Starten des Betriebssystems hat noch keine Task ein Event erhalten können. Von daher erhalten beide Attribute den Wert EVENTMASK_NONE (s.h. 3.8 Events).

(11): Falls man das Bitmap-Scheduling verwendet werden hier die Bitmasken initialisiert.

(12): Zum Abschluss folgt noch die Initialisierung des Stacks. Hier wird der Context erstellt wie er beim Starten einer Task auszusehen hat. Dazu aber ebenfalls später mehr.

3.5 Schedulingverfahren

Wie in allen Betriebssystemen ist der Scheduler eines der wichtigsten Elemente der Task Verwaltung. Der Scheduler bestimmt welche Task als nächstes an die Reihe kommt und welche nicht. Es gibt viele verschiedene Arten wie man einen Scheduler realisieren kann. Diese ist ausschlaggebend für die Geschwindigkeit eines Betriebssystems.
Das HSE Free OSEK besitzt zwei verschiedene Scheduling Verfahren. Beide sind jedoch Priorität basierend. Die beiden Schedulingverfahren unterscheiden sich nur darin wie man die höchst priori Task findet. Beim Bitmapscheduling existieren nur 64 Priortitätsstufen, beim Scheduling mit verketteten Listen 255.

3.5.1 Scheduling mit verketteten Listen

Aus den Grundlagen der Informatik und vielen verschiedenen Büchern die sich mit der C Programmierung beschäftigen, ist die Theorie der „Verketten Listen“ bekannt. Bei einer doppelt verketteten Liste ist es nur notwendig zu wissen wo sie anfängt und wo sie aufhört. Es ist egal ob die Strukturen im Speicher hintereinander liegen oder im RAM verstreut sind. Das war die Grundlage auf der wir uns dafür entschlossen haben die TCB's miteinander zu verketten. Durch einfache Kontrollabfragen ist es auch möglich neue Elemente in diese Kette einzugliedern oder zu entfernen. Um HSE Free OSEK mit diesem Schedulingverfahren zu betreiben, ist es notwendig die „../Src/OSEK Hardware Specific/typedef.h“ zu bearbeiten.
Man sucht die Konstante #define SCHEDULER und weißt ihr den Wert 0 zu.

Nun ein wenig mehr zu den Grundlagen des Schedulingverfahrens. Der Zustand Ready bedeutet das eine Task ablaufbereit ist doch im Moment eine andere Task eine höhere Priorität besitzt und läuft (Running). Es können mehrere Task ablaufbereit sein. Diese werden nach ihrer Priorität in einer Warteschlange namens ReadyQueue geordnet.

Der TCB der sich am Anfang der Liste befindet wird als nächster bei einem Taskwechsel in den Zustand Running versetzt und aus der Liste entfernt (s.h. Abb. 21).

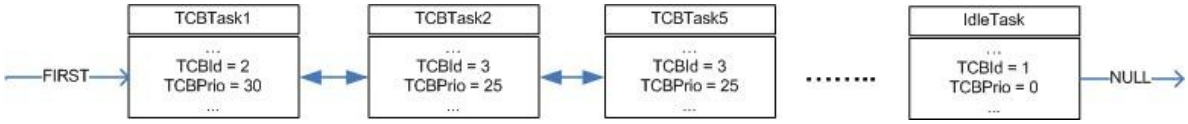


Abbildung 21: Die ReadyQueue

Es ist aus allen Zuständen möglich in den Zustand Ready zu gelangen. Der üblichste Fall ist es, das eine Task inaktiv ist (SUSPENDED) und von einer anderen Task oder auch einen Alarm in den Zustand Ready versetzt wird.
Wenn die gerade aktivierte Task die höchste Priorität in der Liste besitzt, oder die Liste leer ist, wird der TCB der Task einfach nur an die spitze der ReadyQueue eingefügt.

Das folgende Beispiel (s.h. Abb. 22) zeigt die Aktivierung einer Task mit der höchsten Priorität in der ReadyQueue.

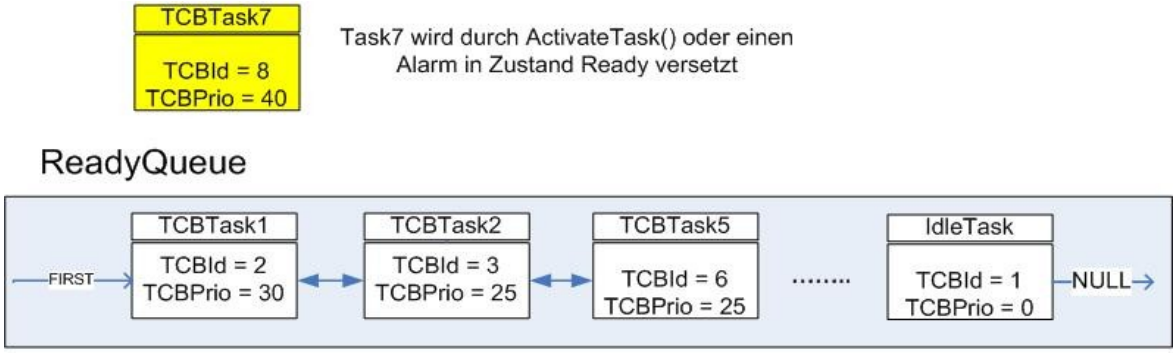


Abbildung 22: Höchstpriori Task wird aktiviert

Nach der Aktivierung folgt das Einfügen in die ReadyQueue (s.h. Abb. 23).

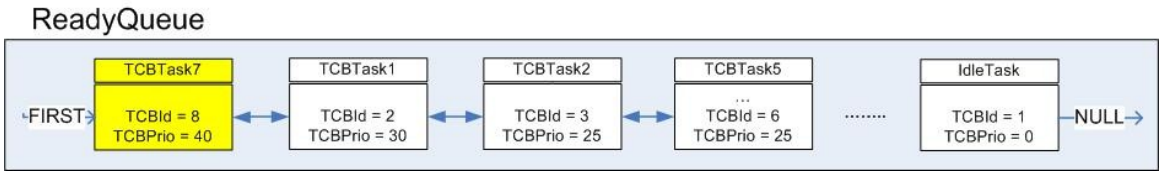


Abbildung 23: Höchst priori Task an die Spitze der ReadyQueue gesetzt

Wenn die neu aktivierte Task allerdings eine niedere Priorität besitzt, wird eine Suche innerhalb der Queue durchgeführt. Der Suchalgorithmus geht die Queue von Anfang an durch bis er auf einen TCB trifft der eine kleinere Priorität besitzt als die neu aktivierte Task. Dann wird der TCB mitten in die Liste eingefügt (s.h. Abb. 24).

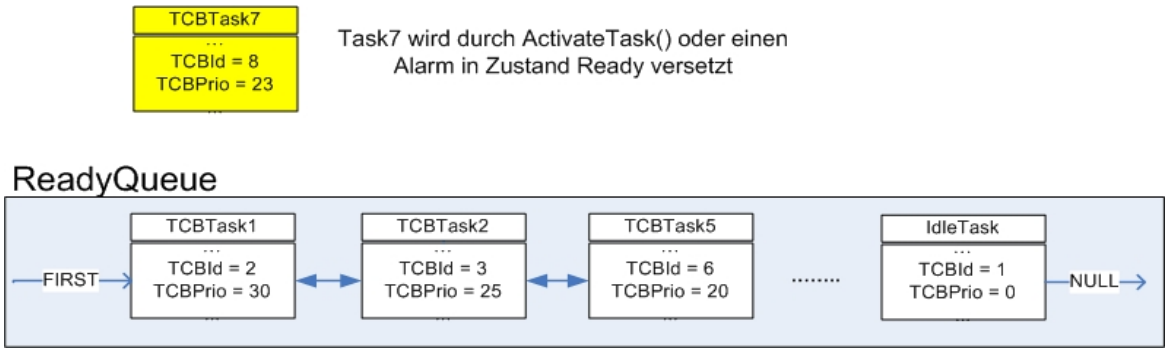


Abbildung 24: Task mit mittlerer Priorität in die Queue einfügen

Wenn die Suche auf eine Task mit niederer Priorität trifft, wird die neu aktivierte Task vor diese in der Queue eingefügt (s.h. Abb. 25).

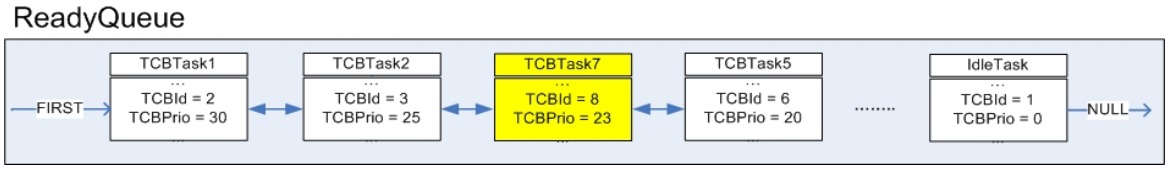


Abbildung 25: Task in der Queue eingefügt

Im allgemeinen Fall wird in allen Funktionen die einen Taskwechsel zur Folge haben könnten (Point of Rescheduling) folgende Abfragen gemacht (s.h. Abb. 26):

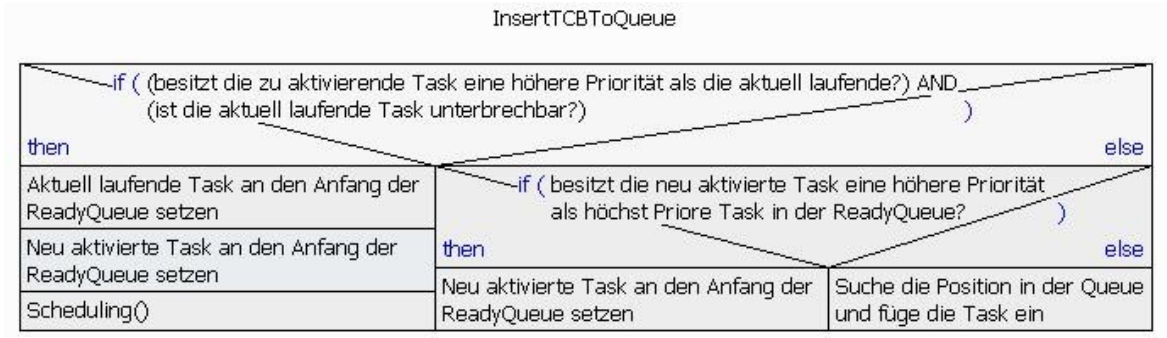


Abbildung 26: Allgemeine Abfragestruktur für den Taskwechsel

Je nachdem welche Betriebssystemfunktion aufgerufen wird, liegen die Unterschiede zu der in Abb. dargestellten Abfragestruktur im Detail. Z.B. beim Aufruf der Funktion ***ChainTask()*** würde sich die aktuell laufende Task sich selbst beenden und eine neue Task aus den Zustand Suspended in den Zustand Ready versetzen. Also würde die Abfrage nach der Priorität der laufenden Task wegfallen. Die genauen Unterschiede jeder einzelnen Funktion werden später im Kapitel 4 näher verdeutlicht.

Die Taskzustände Suspended, Wait und Running erfordern keine Verkettung von daher sind die Pointer ***nextTCB*** und ***prevTCB*** in diesen Zuständen immer NULL.

Das Scheduling mit den verketteten Listen ist ein sehr anschauliches Verfahren und auch relativ einfach zu implementieren. Es bringt allerdings einen Nachteil mit sich. Je mehr Tasks sich im Zustand Ready befinden desto länger wird die Liste. Wenn eine Task mit einer niederen Priorität aktiviert wird muss die Suche nach der Position in der Queue durchgeführt werden. Je länger die Liste ist desto länger dauert die Suche nach der Position was auch eine relativ lange Ausführungszeit mit sich bringen kann.

Die Dauer der Ausführungszeit für das Positionieren einer Task in der ReadyQueue hängt von verschiedenen Parametern wie z.B. dem CPU-Takt, der Speicherzugriffszeit und eben der Anzahl Tasks welche sich im Zustand Ready befinden. Um diesen Vorgang genau zu bestimmen wären auf jeder Entwicklungsplattform Messungen notwendig welche in Rahmen dieser Studienarbeit nur in einer Simulationssoftware in µVision3 am LPC2148 durchgeführt worden ist. In der OSEK Code/Task.c findet sich die Funktion Reschedule und in ihr findet die Suche statt. Die Simulation ergab bei einem Durchlauf die Zeit von 517 ns (s.h. Abb. 27).

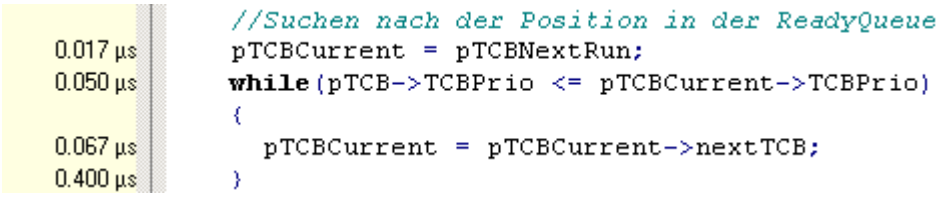


Abbildung 27: Messung am LPC2148 bei einer CPU-Taktfrequenz von 60MHz

Beispielrechnung:

Angenommen es währen 9 Tasks bereits in der ReadyQueue und eine neue würde hinzukommen welche die niederste Priorität besitzt mit Ausnahme der IdleTask.
Wie lange würde die Suche nach der Position in der Queue dauern?

Anzahl Tasks im Ready Zustand ($Task_{Ready}$): 10 inklusive IdleTask
Vergleichszeit ($t_{compare}$): 517ns
Suchlaufzeit (t_{search})

Formel: $t_{search} = (Task_{Ready} - 1) \times t_{compare} = 9 * 517ns$

Ergebnis Beispielrechnung **$t_{search} = 4{,}653\mu s$**

Die Zeit erscheint unter den Bedingungen der Beispielrechnung nicht lang doch unter bestimmten Umständen wie einem langsameren CPU-Takt und mehreren Tasks im Zustand Ready, könnte das Schedulingverfahren auch soviel Zeit in Anspruch nehmen das die Echtzeitfähigkeit des Systems nicht mehr gewährleistet werden kann. Dennoch ist es für die meisten Applikationen anwendbar und funktionsfähig.

In HSE Free OSEK existiert aber noch ein weiteres Verfahren namens Bitmapscheduling mit dem man die Zeit mittels einer BitOSMapTbl[] um ein wesentliches verkürzen kann.

3.5.2 Scheduling mit einer BitOSMapTbl[] und mit verketteten Listen

Das wichtigste bei diesem Verfahren ist, dass es nur 64 Prioritäten gibt. Da die Prio 0 (Idle Task) und die Prio 63 (Kernel Task)für Betriebssystem Anwendungen vergeben sind, stehen dem User nur noch 62 Prioritäten zur Verfügung. Die OSEK Spezifikation gibt vor, dass die Prioritäten aufsteigend von der kleinsten Prio = 0 zur größten Prio = 63 verlaufen. Hierbei ist es auch möglich mehreren Tasks die gleiche Priorität zu vergeben.

Bei diesem Verfahren wird eine BitOSMapTbl[] verwendet, mit deren Hilfe die Task gefunden wird, welche die höchste Priorität besitzt und sich im Zustand Ready befinden.

```
INT8U const OSBitMapTbl[] = {
   0, 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3,          //0x00 bis 0x0F
   4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,          //0x10 bis 0x1F
   5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,          //0x20 bis 0x2F
   5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,          //0x30 bis 0x3F
   6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,          //0x40 bis 0x4F
   6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,          //0x50 bis 0x5F
   6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,          //0x60 bis 0x6F
   6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,          //0x70 bis 0x7F
   7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,          //0x80 bis 0x8F
   7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,          //0x90 bis 0x9F
   7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,          //0xA0 bis 0xAF
   7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,          //0xB0 bis 0xBF
   7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,          //0xC0 bis 0xCF
   7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,          //0xD0 bis 0xDF
   7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,          //0xE0 bis 0xEF
   7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7           //0xF0 bis 0xFF
};
```

Ein kurzes Beispiel wie eine solche BitOSMapTbl[] funktioniert:

Im HSE Free OSEK Betriebssystem sind die **OSReadyGrp** und die **ReadyTable** 8 Bit groß und somit ist die **OSBitMapTbl** wesentlich größer. Und zwar 256 Byte, wie im oberen Bild zu sehen ist. In diesem Beispiel werden nur 4 Bit für die **OSReadyGrp** und die **ReadyTable** verwendet um es besser zu verdeutlichen.

Das Ergebnis auf der linken Seite der Tabelle ist die BitOSMapTbl[] für eine absteigende Priorität. Die auf der rechten Seite für die aufsteigende Priorität.

					absteigende Priori					aufsteigende Priori			
	3	2	1	0	0	1	2	3		0	1	2	3
0	0	0	0	0	X					X			
1	0	0	0	1	X					X			
2	0	0	1	0		X					X		
3	0	0	1	1	X						X		
4	0	1	0	0			X					X	
5	0	1	0	1	X							X	
6	0	1	1	0		X						X	
7	0	1	1	1	X							X	
8	1	0	0	0				X					X
9	1	0	0	1	X								X
10	1	0	1	0		X							X
11	1	0	1	1	X								X
12	1	1	0	0			X						X
13	1	1	0	1	X								X
14	1	1	1	0		X							X
15	1	1	1	1	X								X

Diese 4 Bit Maske kann man sich als eine Art X und Y Koordinate für die ReadyTable vorstellen. Je nach dem ob es sich um absteigende oder aufsteigende Prioritäten handelt, wird immer dort ein Kreuz gesetzt wo sich das derzeit hochwertigste Bit in der Bit Maske auf 1 bzw. 0 befindet. Der Zustand Null muss hier nicht berücksichtigt werden, da dieser Zustand nicht erreicht werden kann. Es muss immer mindestens eine Task Ready sein. Im HSE Free OSEK ist z.B. immer die Idle Task Ready. Nun wird das Ergebnis welches man haben will in einem Array zusammen gefasst.

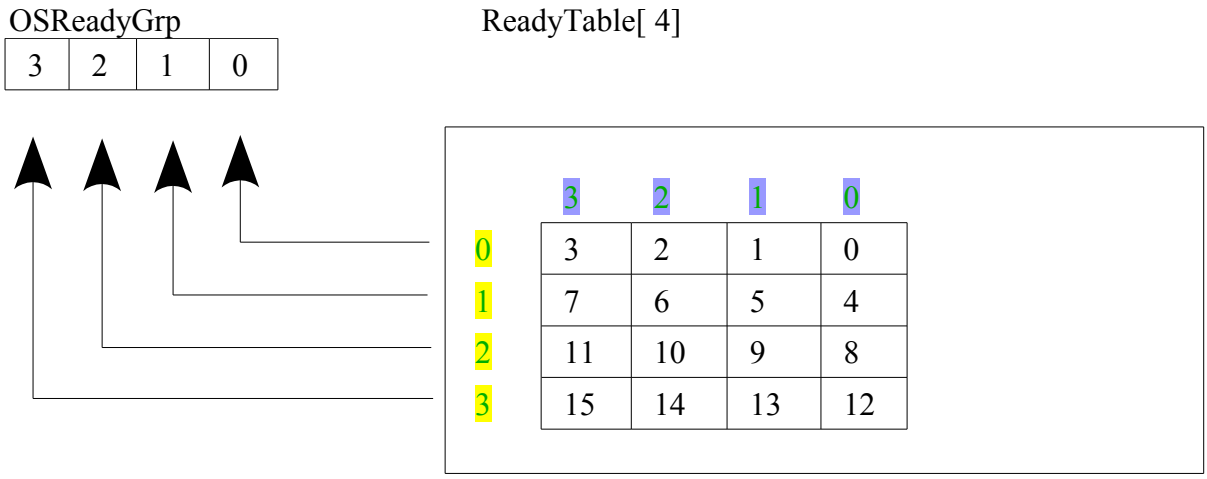
Absteigende Priori

0	0	1	0
2	0	1	0
3	0	1	0
2	0	1	0

Aufsteigende Priori

0	0	1	1
2	2	2	2
3	3	3	3
3	3	3	3

Die ReadyGroup ist in diesem Fall vier Bit groß. Jedes Bit in dessen Gruppe mindestens eine Task Ready ist wird zu 1 gesetzt. Die ReadyTable ist ein Array vom Type char.



Nun ein kleines Beispiel, um das ganze noch ein bisschen zu veranschaulichen. Angenommen drei Tasks sind Ready und somit in der ReadyTable eingetragen. Task1 Priori = 3, Task2 Priori = 5 und Task3 Priori = 9. Die 15 ist hier die größte Priorität und die 0 die kleinste. Somit muss die OSReadyGrp = 1101b = 13d sein. Die ReadyTable muss dementsprechend wie folgt aussehen:
ReadyTable [1000, 0000, 0010, 1000];
Wenn nun in der BitOSMapTbl[] das Byte an der Stelle 13 genommen wird, Y=OSBitMapTbl[13] kommt für Y = 3 heraus. Da bei aufsteigenden Prioritäten und bei dieser Bit folge das hochwertigste Bit welches 1 ist das 2^3 Bit ist. Somit ist jetzt klar, dass die Task die die höchste Priorität besitzt in der dritten Gruppe zu finden ist. Da in diesem Beispiel die Task mit der höchsten Priori die Task3 mit der Priori = 15 ist, muss das Ergebnis bisher richtig sein. Also wird nun mit dem Bit Muster aus der dritten Gruppe weiter gearbeitet ReadyTable[3] = 1000b = 8d.
Nun noch einmal das selbe für die X Koordinaten X=OSBitMapTbl[8] == 3. Das entspricht, in der ReadyTable, der 15.
Um nun auch rechnerisch auf die Priorität zu kommen, müssen die beiden Koordinaten verbunden werden. Das geschieht mit folgender Rechnung:

Priorität = (Y << 2) + X
Da es in diesem Beispiel nur 16 Prioritäten gibt, also nur vier Bit groß ist, muss die Y Koordinate nur um 2 verschoben werden. Nun würde im obigen Beispiel für X und Y 15 = (3 << 2) + 3 herauskommen.

Im HSE Free OSEK sind es nicht 16 Prioritäten sonder 64. Aus diesem Grund ist die ReadyTable und die BitOSMapTbl[] (= 256 Byte) um einiges größer. Jedoch ist die Funktionsweise die Gleiche.

```
//Höchstpriore Task finden
y = OSBitMapTbl[OSReadyGrp];
prioHighReady = (INT8U)((y << 3) + OSBitMapTbl[ReadyTable[y]]);
```

Das Setzen einer Task in den Ready Zustand, verläuft vom Prinzip her gleich wie das obere Beispiel nur umgekehrt.

```
//Aktiviere Bit in ReadyTable und in der Ready Goup --> eine Task Ready machen
OSReadyGrp |= OSMapTbl[pTCB->TCBPrio >> 3];
ReadyTable[pTCB->TCBPrio >> 3] |= OSMapTbl[pTCB->TCBPrio & 0x07];
```

Die Priorität der Task wird wieder aufgeteilt in eine X und eine Y Koordinate. Mit Hilfe der OSMapTbl[] wird nun die entsprechende Gruppe und daraus die entsprechende Priorität aktiviert.

OSMapTbl[]

Index	Bit Maske (Binär)
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

Beim Entfernen einer Task aus der ReadyTable, muss zuerst abgefragt werden ob noch eine andere Task Ready ist und sich in der selben Gruppe befindet. Erst wenn es die einzige Task in der Gruppe ist, kann auch das Bit in der OSReadyGrp gelöscht werden.

```
//Löschen der Bits für ReadyTable
if((ReadyTable[pTCBCurRun->TCBPrio >> 3] &=
    ~OSMapTbl[pTCBCurRun->TCBPrio & 0x07]) == 0x00)
{
//wenn in selben Gruppe kein Task Ready ist, dann auch Gruppen Bit auf Null setzen
    OSReadyGrp &= ~OSMapTbl[pTCBCurRun->TCBPrio >> 3];
}
```

Auch hier wird zu aller erst die Priorität in die X und Y Koordinaten aufgeteilt, um dann die entsprechende Bit Maske aus der ReadyTable mit der negierten Bit Maske aus der OSMapTbl[] zu konjugieren. Wenn dann etwas ungleich Null herauskommt, sind in der Gruppe noch andere Tasks Ready. Somit darf der weitere Schritt nicht mehr vollzogen werden.

Im HSE Free OSEK Betriebssystem ist es auch erlaubt mehreren Tasks die gleiche Priorität zuzuweisen. Aus diesem Grund müssen alle Tasks die eine gleiche Priorität besitzen und Ready sind, mit Hilfe von verketteten Listen, nach dem FIFO Prinzip aneinander gekettet werden. In den jeweiligen Task Control Blöcken (s.h. 3.4 Der Task Control Block) wird mit Hilfe von Pointern auf die Adresse des nächsten Task Control Blocks gezeigt (s.h. Abb. 28).

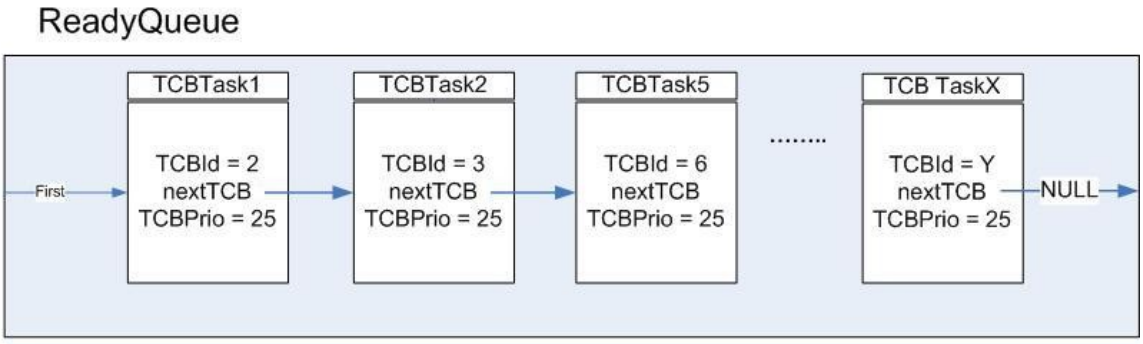


Abbildung 28: Verkettete Liste beim Bitmapscheduling

Wenn eine Task aus der ReadyTable genommen wird, muss zuerst überprüft werden ob es noch weitere Task gibt die ready sind und die gleiche Priorität besitzen. Das wird überprüft in dem man die nextTCB Zeiger mit der NULL vergleicht. Wenn ungleich NULL dann sind noch weitere Tasks ready und somit darf dieses Bit nicht gelöscht werden. Jedoch muss in der Task, die aus der Liste genommen wird, der nextTCB Pointer auf NULL gesetzt werden und die Adresse der nächsten Task in die TCBPrioTbl geschrieben werden.

3.6 Der Context Switch

Das Herz eines jeden Betriebssystems ist der Context Switch. Die im vorherigen Kapitel beschriebenen Schedulingverfahren entscheiden welche Task als nächste läuft, doch erst der Context Switch oder auch Dispatcher ermöglicht den Taskwechsel an sich. Dieser Vorgang wird nun im weiteren Verlauf näher beschrieben.

Mikrocontroller können von der Architektur und den Features sehr unterschiedlich zueinander sein aber im Kern besitzen sie alle einen Pointer der auf die aktuelle Adresse im Stackzeit (SP), Arbeitsregistern (R0,R1,..,RX) in denen alle Werte geladen und verarbeitet werden und ein Prozessorstatuswort(PSW) was Auskunft über den momentanen Prozessorstatus liefert. Wenn eine Task am laufen ist zeigt der SP des Prozessor auf den Stack Bereich dem der Task zugewiesen ist. Alle Werte und Adresse die in der Task verwendet werden, werden teilweise in den Registern abgelegt und von dort aus bearbeitet. Zusätzlich kann z.B. bei einem Überlauf das Prozessorstatuswort Auskunft liefern. Wenn man nun eine Task mitten in der Ausführung unterbricht muss gewährleistet werden das beim Wiedereintritt der SP, die Arbeitsregister und das Prozessorstatuswort die selben Werte besitzen wie vor dem Taskwechsel.

HSE Free OSEK hat sich da an den Grundlagen von µC/OS II angelehnt. Die folgende Abbildung soll das Verfahren veranschaulichen.

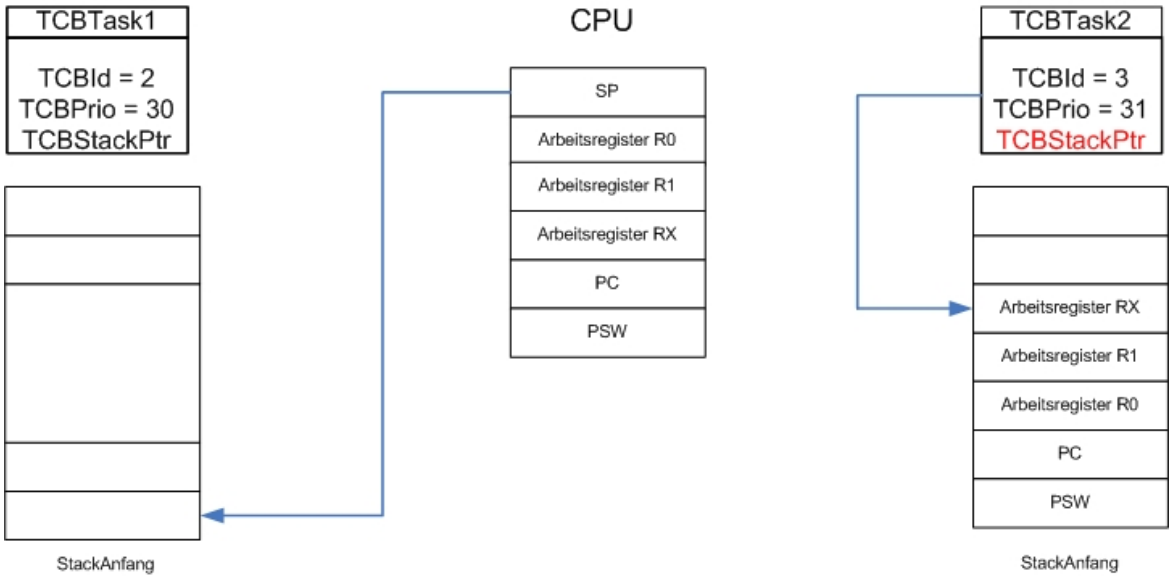


Abbildung 29: Betrachtung des allgemeinen Taskswechsels 1

Das Beispiel in Abb. 29 Zeigt 2 Tasks. Task1 läuft gerade und Task2 befindet sich im Zustand Suspended. Nun hat eine ISR die Task2 aktiviert und der Scheduler entscheidet das die Task2 laufen soll.

Abbildung 30 zeigt das zuerst alle prozessorspezifischen Register auf dem Stack der Task abgelegt werden. Im 2. Schritt wird die aktuelle SP-Adresse der Task1 dem TCBStackPtr Attribut zugewiesen. Im 3. Schritt wird der SP der Task2 aus dem jeweiligen TCB und dem Attribut TCBStackPtr geladen.

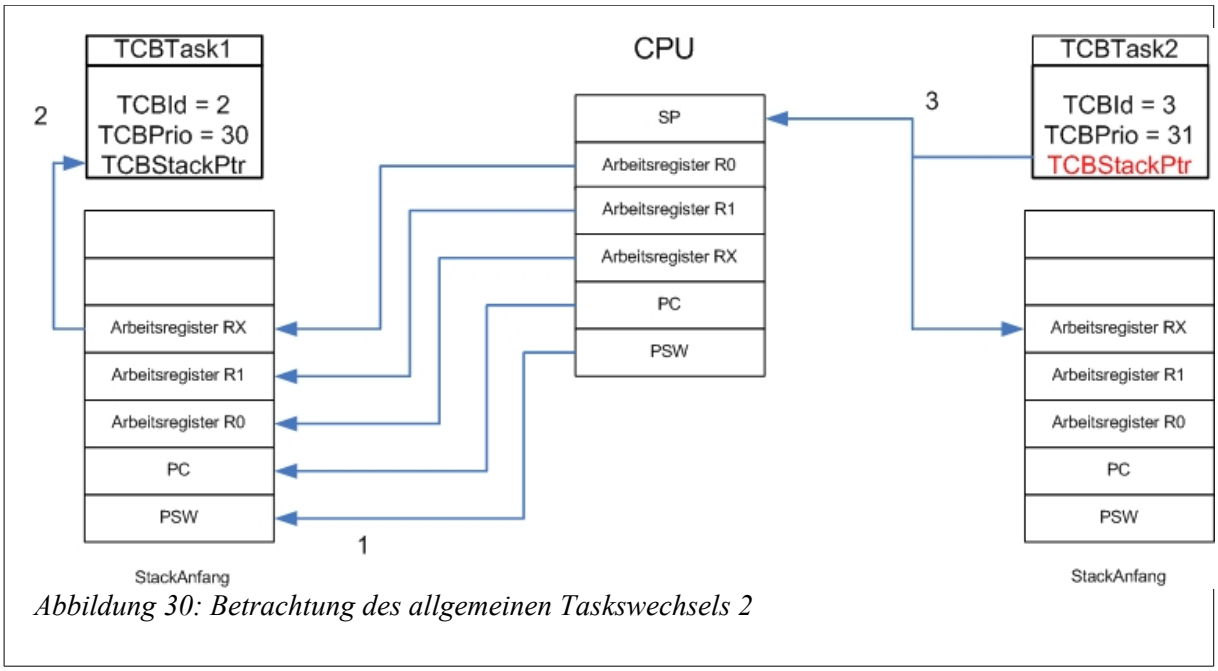


Abbildung 30: Betrachtung des allgemeinen Taskswechsels 2

Abbildung. 31 zeigt den ausgeführten Context Switch. Von Task1 sind alle Daten gesichert und die prozessorspezifischen Register sind aus dem Stack von Task2 geladen worden. Der Taskwechsel war erfolgreich.

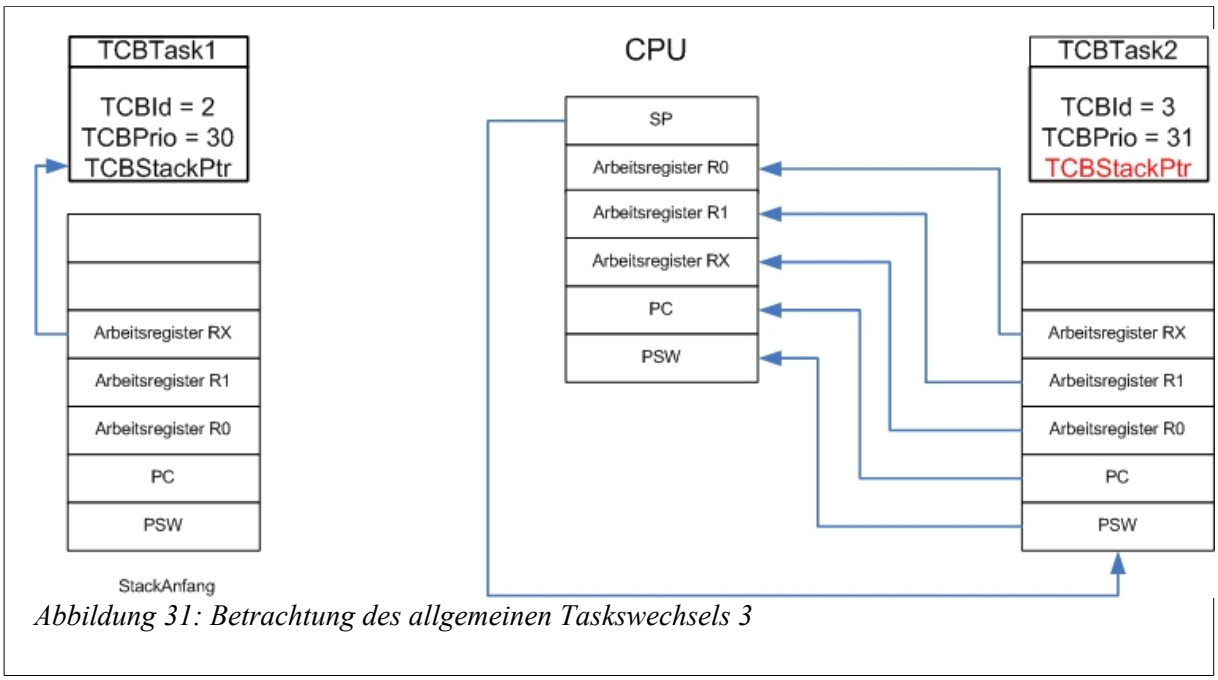


Abbildung 31: Betrachtung des allgemeinen Taskswechsels 3

3.6.1 Freescale HCS12

Auf einem freescale HCS12 Derivat sieht ein Context Wechsel wie folgt aus.

Task1 setzt ein Event, was zur Folge hat dass Task2, welches auf dieses Event wartet und eine höhere Priorität besitzt, in den Zustand Running versetzt wird. Dies hat wiederum zur Folge, dass ein Context Wechsel stattfindet.

```
Task1
{
        ...
        /*Event wird gesetzt*/
        SetEvent(Task2, Mask);
        ...
}
```

Bei einem freescale HCS12 Derivat werden bei einem Context Wechsel folgende Register auf dem Stack gesichert (s.h. Abb. 32).

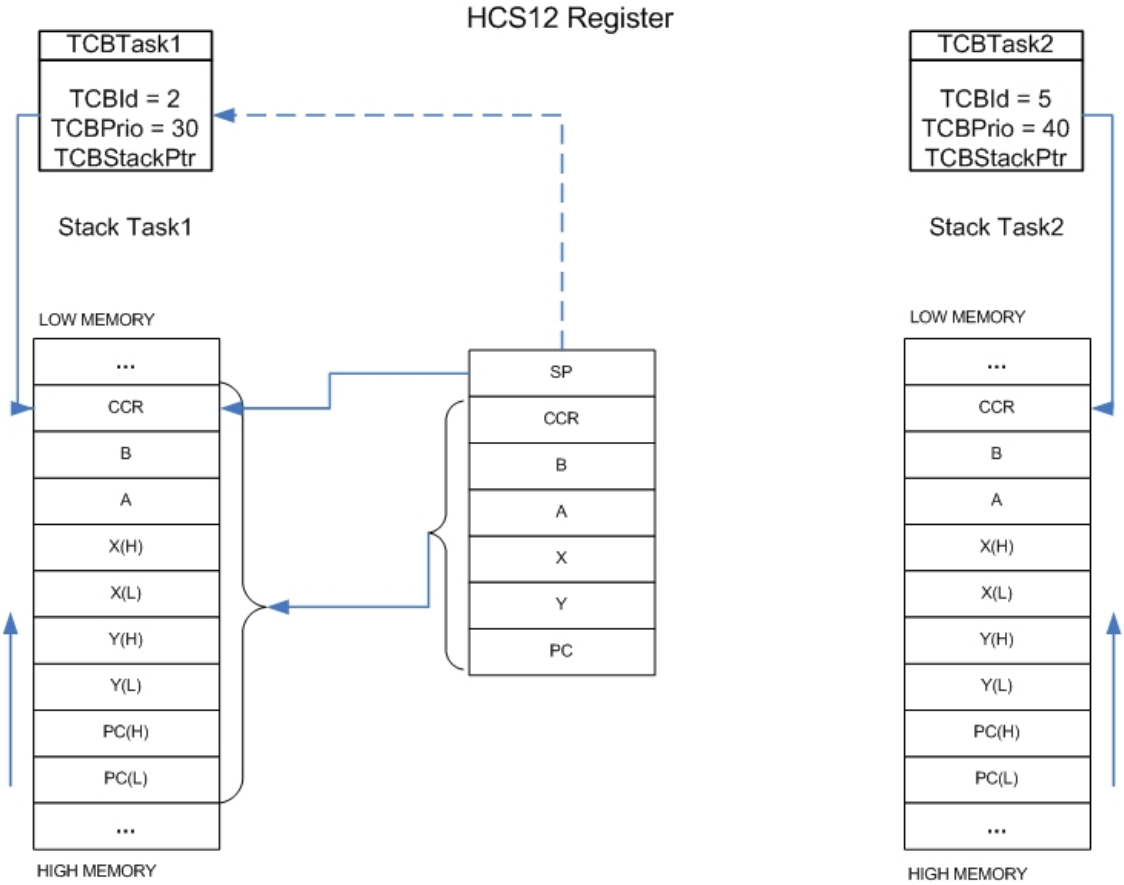


Abbildung 32: Betrachtung Taskwechsel bei HCS12 1

Nachdem nun alle Register von der unterbrochenen Task auf dem Stack gesichert sind, wird dem Stackpointer die Adresse des neuen Stacks zugewiesen und die Register werden geladen (s.h. Abb. 33).

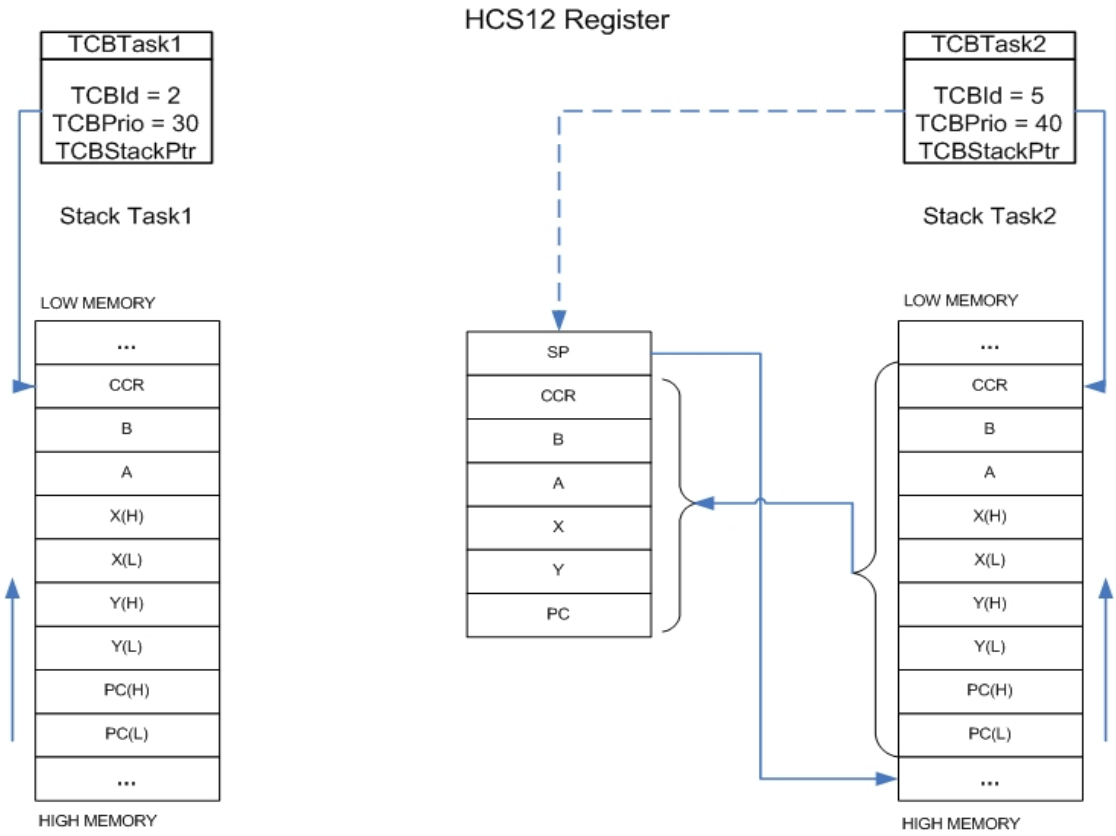


Abbildung 33: Betrachtung Taskwechsel bei HCS12 2

Nun wird die neue Task gestartet. Nach dem Terminieren wird die Task wieder beendet und die gleiche Prozedur, wie oben beschrieben, wird durchlaufen.

```
Task2
{
        ...
        printf("Hallo");
        /*Die Task beendet sich selbst*/
        TerminateTask();
        ...
}
```

```
#pragma TRAP_PROC                                          (1)
void ContextSwitch(void)
{
  __asm
  {
    pshy
    pshx
    pshd
    pshc                                                   (2)

    ldx   pTCBPreRun
    sts   0,x                                              (3)

    ldx   pTCBCurRun
    lds   0,x                                              (4)
  }
}
```

Erklärung zu ContextSwitch:

(1): Der **TRAP_PROC** Befehl dient dazu die Assembler Anweisung Return to Interrupt (RTI) an das Ende der Funktion zu setzen. Diese Anweisung führt automatisch die am Ende der Funktion relevanten POP Befehle durch. Aus diesem Grund sind in dieser Funktion keine Assembler POP Anweisungen zu sehen.

(2): Die einzelnen Register werden auf dem Stack gesichert. Dies geschieht mit Hilfe der verschiedenen PUSH Anweisungen in Assembler.

(3): Der aktuelle Stack Pointer wird, mit Hilfe des globalen Zeigers **pTCBPreRun**, gesichert.

(4): Zum Schluss wird der Stack Pointer auf die aktuelle Adresse des Stacks gesetzt

Wenn der System Interrupt von HSE Free OSEK auftritt, wird immer eine Kernel Task (s.h. 3.11.2 Kernel Task) aufgerufen. Aus diesem Grund findet hier immer ein Context Wechsel statt. Jedoch muss im Gegensatz zum normalen Task Wechsel, die ContextSwitch Funktion abfragen ob es sich bei der unterbrochenen Task um eine preemptive oder nicht preemptive Task handelt. Deshalb wird bei Interrupts immer eine etwas veränderte Context Wechsel Funktion aufgerufen als bei normalen Task Wechsel.

```
#pragma TRAP_PROC
void INTCTXSW(void)
{
  __asm
  {
     pshy
     pshx
     pshd
     pshc

     ldaa   OSScheduleMode     //ScheduleModus herausfinden                    (1)
     cmpa    #0                                                               (2)
     bne    Full_Sched

     //Adresse der unterbrochenen Task holen
     ldx    pTCBInterruptedTask                                               (3)
     sts    0,x
     bra   Set_Stack

Full_Sched:
     ldx    pTCBPreRun
     sts    0,x
     //Stack Pointer auf neue Task setzen
Set_Stack:
     ldx    pTCBCurRun
     lds    0,x
  }
}
```

Erklärung zu_INTCTXSW:

(1): **OSScheduleMode** ist eine globale Variable die nach jedem Task Wechsel gesetzt wird. Die entsprechende Information wird aus dem Task Control Block der zu aktivierenden Task geholt. Die Variable gibt an ob es sich bei der Task um eine unterbrechbare oder nicht unterbrechbare Task handelt.

(2): Bei **OSScheduleMode** handelt es sich um eine Boolesche Variable. Wenn sie ungleich Null ist, dann ist die zurzeit laufende Task unterbrechbar und somit kann eine höher priorisierte Task gestartet werden. Wenn gleich Null, dann wird die durch den Interrupt unterbrochene Task wieder gestartet.

(3): **pTCBInterruptedTask** ist global deklariert und beinhaltet die Adresse des TCB's der Task die vom Interrupt unterbrochen wurde. Jedoch geschieht das nur in dem Fall wenn die Task nicht unterbrechbar ist. Ansonsten ist **pTCBInterruptedTask** gleich Null.

Folgendes Szenario soll den Context Wechsel bei einem Interrupt etwas näher erläutern:

Task1 befindet sich im Zustand Running. Während der Abarbeitung tritt ein Interrupt auf. Beim HCS12 würde der Vorgang folgendermaßen aussehen (s.h. Abb. 34):

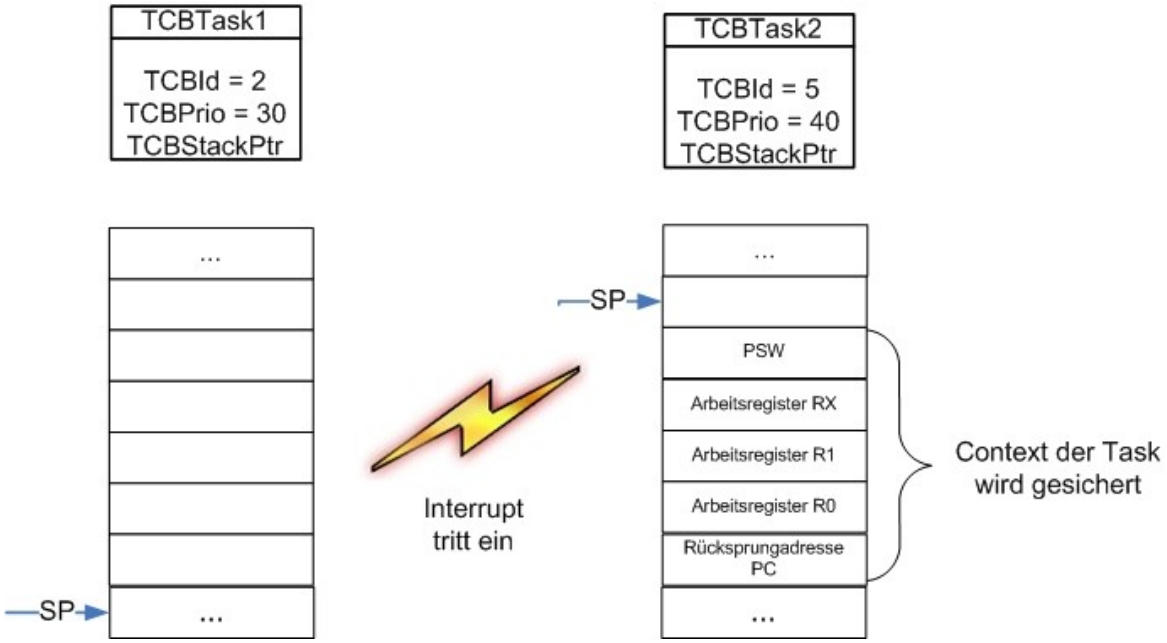


Abbildung 34: Vorgang beim Eintreten eines Interrupts(HCS12)

Der Interrupt unterbricht die aktuell laufende Task. Um zu gewährleisten das die Task, nach Ablauf der ISR, an der Stelle der Unterbrechung fortgesetzt wird, muss der Context auf den taskeigenen Stack gesichert werden. Die ISR selbst arbeitet auf dem Stack der Task weiter.
Angenommen die aktuell laufende Task wäre unterbrechbar und würde nun innerhalb der ISR eine Task aktivieren die eine höhere Priorität als die aktuell laufende Task besitzt. Das hätte dann zur Folge, dass die Rücksprungadresse, die Arbeitsregister (R0 – RX) und das Prozessorstatuswort ebenfalls auf den Stack gelegt werden (s.h. Abb. 35).

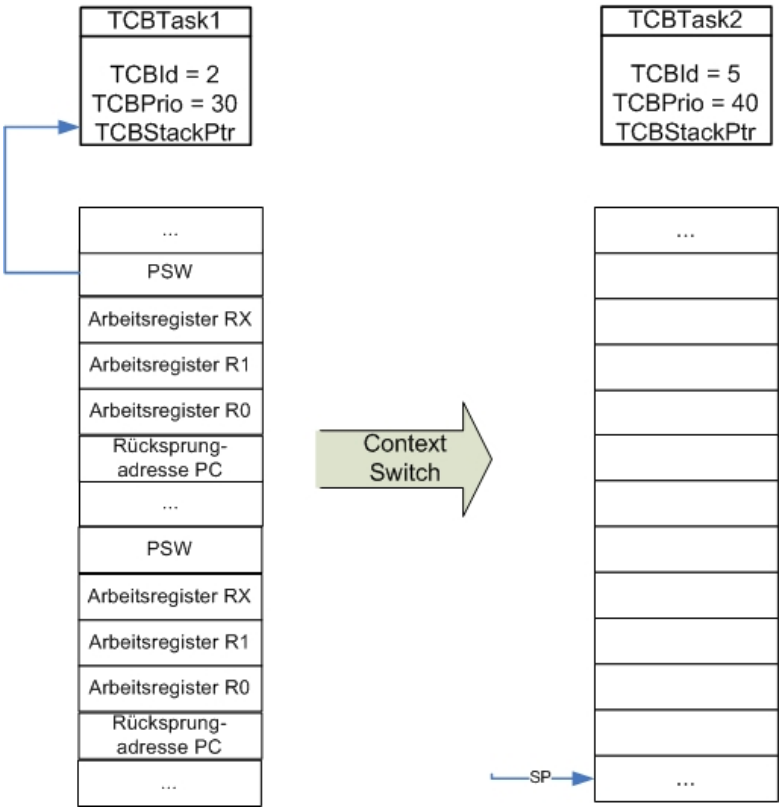


Abbildung 35: Abgespeicherter Context einer TASK

Nach einer bestimmten Zeit befindet sich Task1 wieder im Zustand Running. Der Wiedereintrittspunkt der Task liegt in diesem Fall in der noch nicht vollständig abgearbeiteten ISR. Erst nachdem sich diese beendet, ist der vollständige Context der Task1 wieder hergestellt.

3.6.2 Philips LPC2148

Beim LPC2148 konnte das Verfahren zum Context Switch welches in den vorherigen Kapitel beschrieben worden ist, nicht ohne eine Erweiterung übernommen werden. Im Vergleich zum HCS12 wird das Auftreten eines Interrupts anders abgearbeitet.

Die Interrupts und Fast Interrupts auf dem LPC2148 besitzen ihren eigenen Stack. D.h. wenn nun ein Interrupt auftritt, verändert sich das PSW (CPSR) und der Prozessor wechselt in den IRQ-Modus. Das hat zur Folge, dass der SP auf die Adresse gesetzt wird wo sich der Stack der ISR befindet. Zusätzlich werden die Arbeitsregister (R0-R12) auf den Stack der ISR abgelegt, da diese in allen Prozessormodies verwendet werden (s.h. Abb. 36).

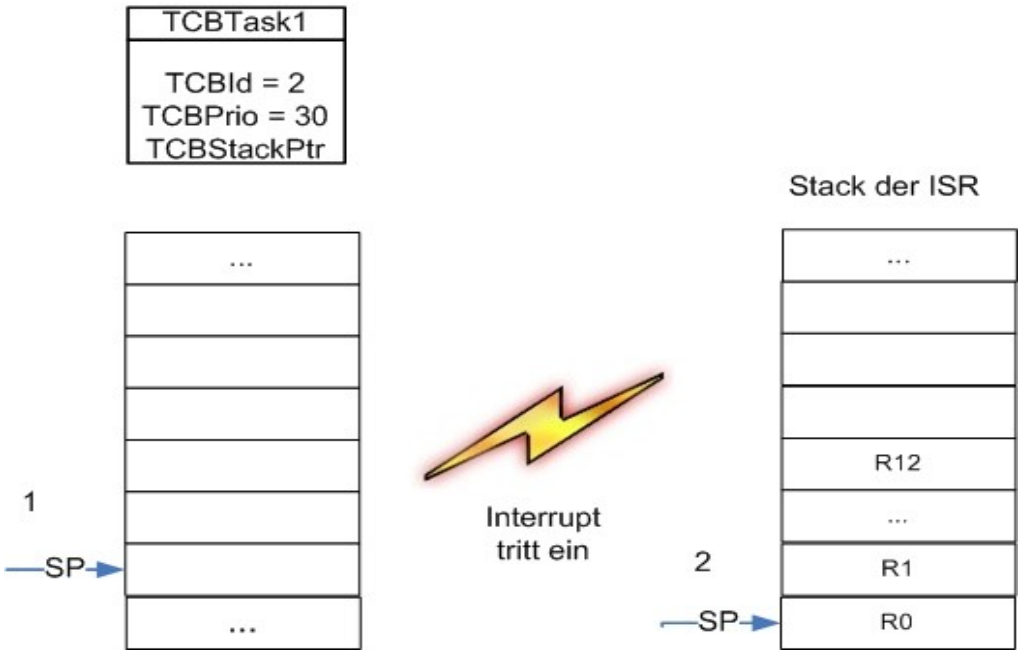


Abbildung 36: Vorgang beim Eintretten eines Interrupts(LPC2148)

Das Problem ergibt sich nun wenn es innerhalb der ISR zu einem Context Switch kommt. Wenn bis nächsten Auftreten des Interrupts die unterbrochene Task nicht mehr in den Zustand Running gelangt, wird der Stack überschrieben. Somit ist der Context der unterbrochenen Task verloren.

HSE Free OSEK hat zur Lösung des Problems auf dem LPC2148 eine Erweiterung vorgenommen und zwar den Context einer jeder einzelnen Task einen festen Speicherplatz zuzuweisen. D.h. Der Context wird nicht mehr auf den Stack gelegt sondern wird immer auf dem Context Speicher verwaltet. Der Vorteil der Methode ist das der Stack nicht unnötig wächst und der Context beim Debuggen immer ohne Aufwand zu betrachten ist. Um nicht unnötig Speicher zu verschwenden muss man nur noch die Größe des Stacks anpassen.

Um die Erweiterung besser zu verstehen soll das nächste Beispiel den Vorgang verdeutlichen. Wie man bereits im Kapitel 3.4.1 beim betrachten des TCBs erkennen konnte, ist eines der Attribute die sich darin befinden der Pointer TCBContextStack. Dieser Pointer zeigt auf die Anfangsadresse im Speicher wo sich der Context Speicher (ContextStorage) befindet. Der Context Speicher hat für jede Task die selbe Größe und könnte bei einer Portierung mit dieser Methode durch das Ändern der Konstanten CONTEXTSIZE (in der typedef.h) angepasst werden. Beim LPC2148 sind es 18 Register die auf Context Speicher abgelegt werden.

Angenommen Task1 würde sich im Zustand Running befinden und ein Event setzten welches die Task2 aktiviert. Darauf hin wird der Context von Task1 in den Task eigenen Contextspeicher abgelegt(s.h.. Abb. 37 und Abb. 38).

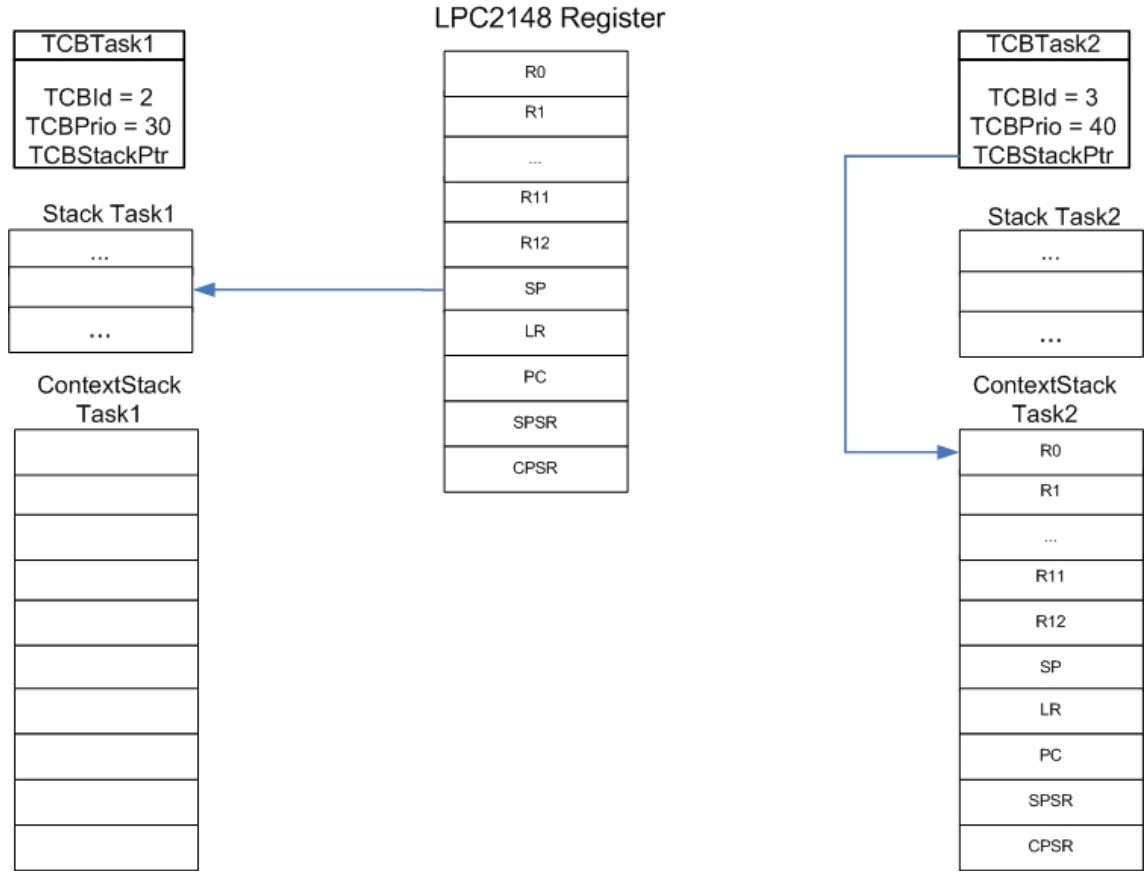


Abbildung 37: Task1 befindet sich im Zustand Running

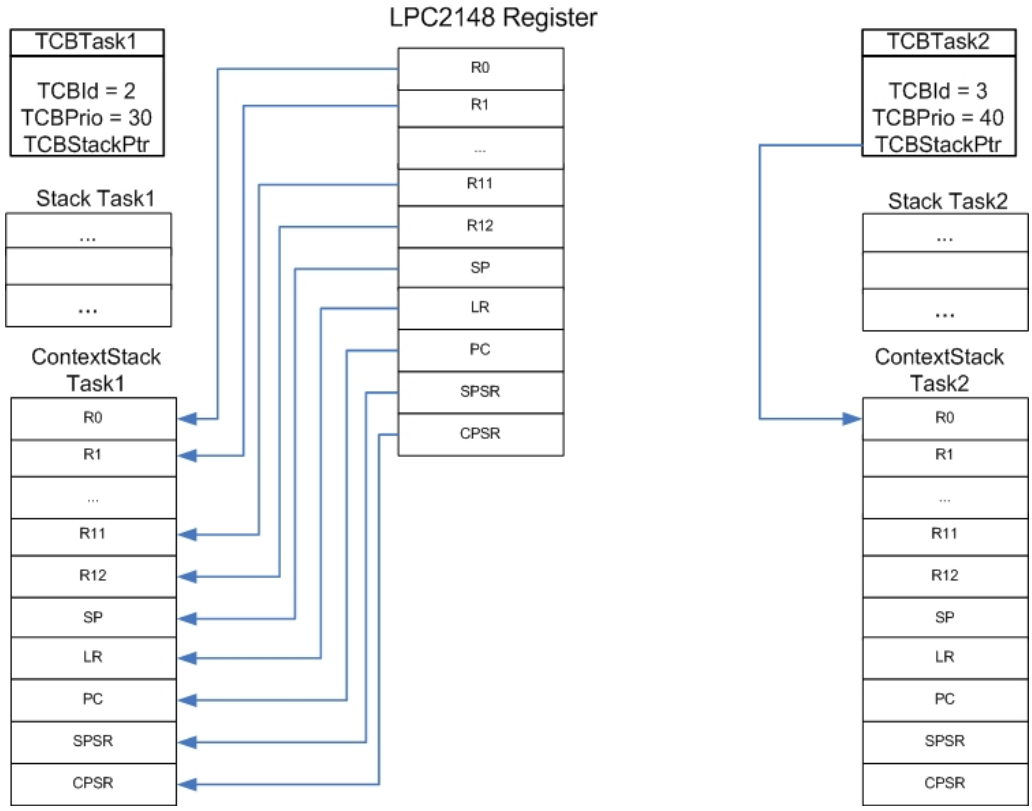


Abbildung 38: Der Scheduler hat entschieden das Task2 laufen soll. Der Context von Task1 wird im Contextspeicher abgelegt

Nachdem nun der Context gesichert worden ist kann der Context von Task2 geladen werden (s.h. Abb. 39). Es ist noch notwendig dem Pointer TCBStackPtr die Endadresse des Contextspeichers von Task1 zuzuweisen.

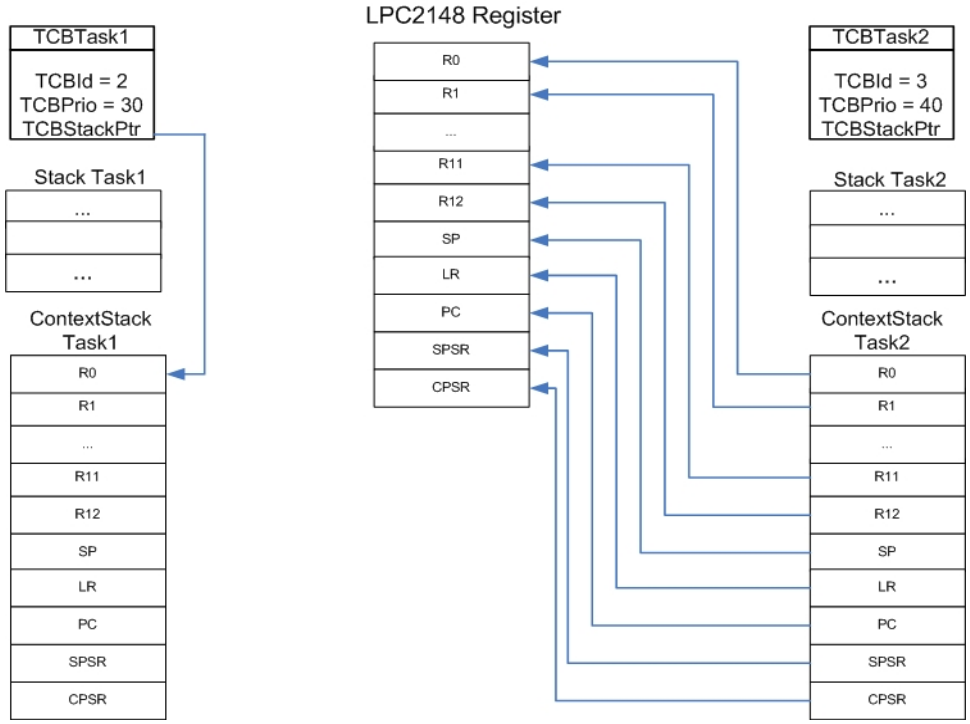


Abbildung 39: Context von Task 2 wird geladen

Im Context Speicher von Task2 ist auch die SP Adresse abgelegt von der derzeitigen Position. Das Task2 zuvor nicht unterbrochen worden ist sondern das erste Mal aufgerufen wurde, zeigt dieser auf die Startadresse des wahren Stacks von Task2. Somit ist der Context Switch abgeschlossen (s.h. Abb. 40).

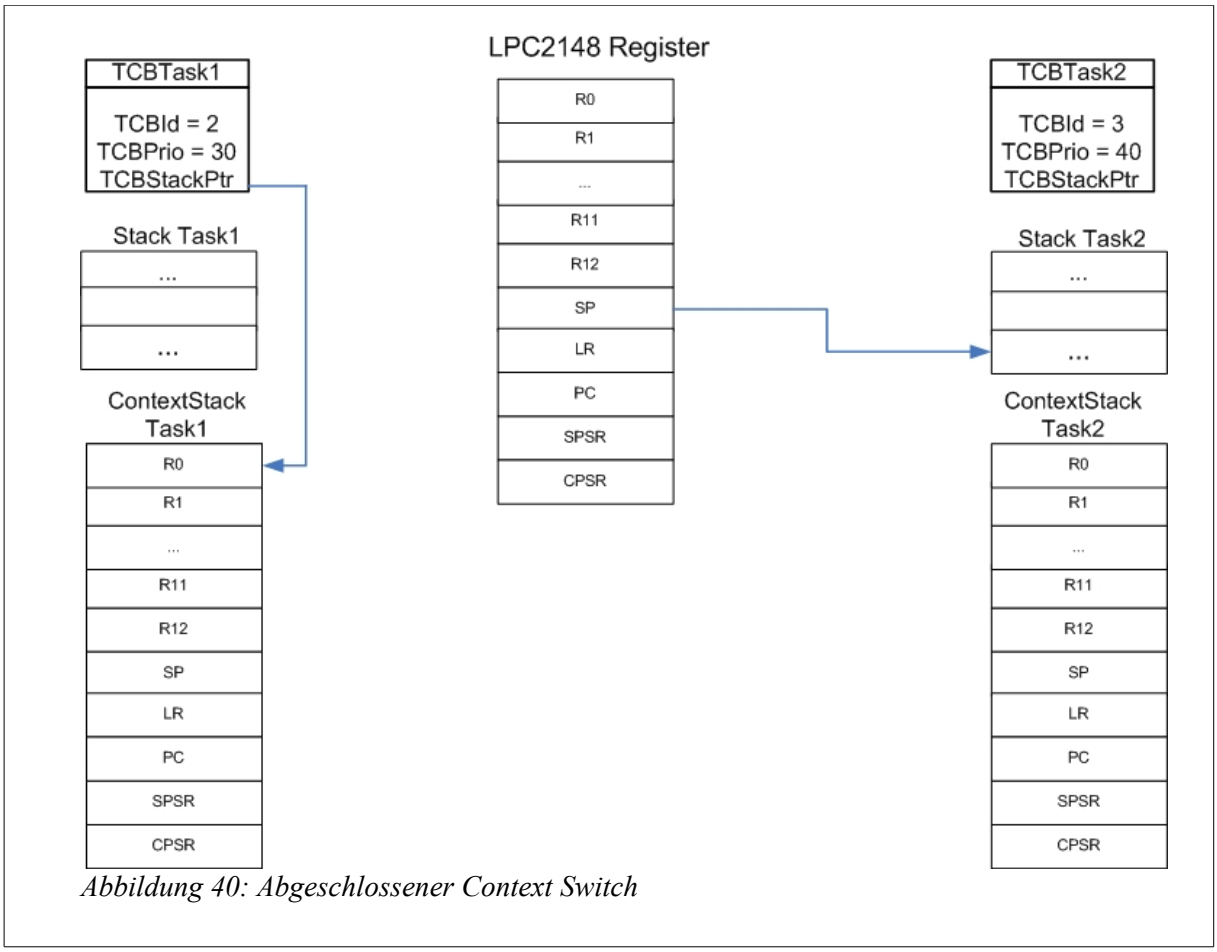


Abbildung 40: Abgeschlossener Context Switch

Der Vorgang hat sich als sehr zuverlässig erwiesen und kann in den Beispielprojekten im Assemblercode nachvollzogen werden in der Datei

„../Src/OSEK Hardware Specific/OS_CPU_A.asm“

Dort sind die Funktion ***__ContextSwitch()*** und ***__ISRCtxSw()*** zu finden und der Vorgang nochmals nachzuvollziehen. Es war notwendig eine zusätzliche Methode zu schreiben, die den Context Switch innerhalb eines Interrupts durchführt, da beim Auftreten eines Interrupts noch der Stack der ISR hinzukommt und der Vorgang dadurch an Aufwand gewinnt.

3.7 Interrupts unter HSE Free OSEK

Ein OSEK konformes Betriebssystem besitzt zwei Kategorien von Interrupts. Das sind zum einen die Betriebssystem Interrupts und zum anderen normale Interrupts die keine Betriebssystem Funktionen aufrufen kann.

- Kategorie 1
  Kein Aufruf von Betriebssystem Funktionen möglich
  Kein Einfluss auf die Tasks

- Kategorie 2
  Betriebssystem Funktionsaufrufe in der ISR Möglichkeit
  Wird über ein bereits definiertes Makro aufgerufen

Im HSE Free OSEK sind Interrupts der Kategorie 2 noch nicht vollständig implementiert. Interrupts der Kategorie 1 werden vollständig unterstützt.

```
ISR(ISRName)                                    (1)
{
        EnterInterrupt();                       (2)
        //Syntax der ISR
        ExitInterrupt();                        (2)
}
```

(1): Hinter dem Makro **ISR(ISRName)** verbirgt sich der Compiler abhängige Code für eine Interrupt Deklaration. Je nach dem welcher Compiler verwendet wird, muss entsprechend auch die Deklaration geändert werden.

(2): Es ist sehr wichtig das am Anfang jeder Interrupt Service Routine die Funktion ***EnterInterrupt()*** und am Ende die Funktion ***ExitInterrupt()*** steht. Dem HSE Free OSEK Betriebssystem muss die Information übergeben werden ob man sich in einer ISR befindet oder nicht. Aus diesem Grund wird beim Aufruf der Funktion ***EnterInterrupt()*** eine globale Variable inkrementiert und beim Aufruf der Funkion ***ExitInterrupt()*** diese wieder decrementiert.

Die OSEK Spezifikation bietet drei verschiedene Funktionen um Interrupts zu unterbrechen bzw. zu aktivieren.

EnableAllInterrupts() / ***DisableAllInterrupts()*** und
ResumeAllInterrupts() / ***SuspendAllInterrupts()***

Die Funktionen ***ResumeOSInterrupts()*** / ***SuspendOSInterrupts()*** dienen dem sperren bzw. freigeben der Interrupts der Kategorie 2. Diese werden unter HSE Free OSEK nicht unterstützt.

```
void EnableAllInterrupts(void)
{
  DecOSNesting();                                                    (1)
  __EnAllInt();                                                      (2)
}
```

Erklärung zu EnableAllInterrupts:

(1): Die Funktion ***DecOSNesting()*** decrementiert eine globale Variable. Diese Funktion hat nur einen statistischen Zweck. Mit Hilfe dieser Funktion kann die Anzahl der hintereinander aufgerufenen Funktionen, um Interrupts zu deaktivieren, ermittelt werden.

(2): In dieser Funktion werden die Interrupts freigegeben. Es handelt sich um Assemblercode, welcher je nach Prozessor anders aussieht und somit in einer extra Funktion verlagert wurden.

Erklärung zu DisableAllInterrupts:

Die Funktion ***DisableAllInterrupts()*** sperrt die alle Interrupts. Die Funktion besitzt ebenfalls eine statistische Funktion, nur das hier die Variable inkrementiert wird. Danach wird die Funktion ***__DisAllInt()*** aufgerufen, welche mit Hilfe von Assemblercode die Interrupts sperrt.

```
void ResumeAllInterrupts(void)
{
        DecOSNesting();

        if (SUS_ALL_CNT == 0)                                   (1)
        {
          //Der Minimale Nesting Counter ist erreicht,
          //kein runterzaehlen mehr moeglich
        }
        else if (SUS_ALL_CNT == 1)                              (2)
        {
                __EnAllInt();
                SUS_ALL_CNT--;                                  (3)
        }
        else
        {
         SUS_ALL_CNT--;                                         (4)
        //ResumeAllInterrupts wurde schon mal vorher aufgerufen
        //und bisher noch nicht wieder gesperrt => Nesting!
       }
}
```

Erklärung zu ResumeAllInterrupts:

(1): Da diese Funktion Nesting unterstüzt, wird hier überprüft ob bereits Minimale Nesting Couter Wert erreicht ist und nicht mehr weiter runtergezählt werden kann. Das kann nur dann der Fall sein wenn die Funktion ***ResumeAllInterrupts()*** öfter aufgerufen wurde als die Funktion ***SuspendAllInterrupts()***, die den Nesting Counter inkrementiert.

(2): Wenn der Nesting Counter genau eins ist, dann wurde die Funktion zum ersten mal aufgerufen ohne das danach wieder ***SusendAllInterrupts()*** aufgerufen wurde. Somit kann nun das eigentliche freigeben der Interrupts erfolgen.

(3): Hierbei handelt es sich um die Nesting Variable die in dieser Funktion Decrementiert wird.

(4): Hier wird die Nesting Variable ebenfalls Decrementiert, jedoch werden keine Interrupts freigegeben. An dieser Stelle wurde die Funktion schon mal aufgerufen und somit sind die Interrupts schon gesperrt.

Erklärung zu SuspendAllInterrupts:

Die Funktion ***SuspendAllInterrupts()***, macht das gleiche wie die Funktion ***ResumeAllInterrupts()***, nur das hier die Interrupts gesperrt und der Nesting Counter inkrementiert wird. Auch hier werden die gleichen if Bedingungen abgefragt wie in der Funktion ***ResumeAllInterrupts()***.

3.8 Events

Um sinnvolle Echtzeitapplikationen zu erstellen ist es notwendig gewisse Routinen auf Ereignisse warten zu lassen. Beispielweise bei der Kommunikation über die UART würde es sich anbieten, dass die ISR der UART Daten empfängt und in einem Buffer ablegt. Wenn man die empfangen Daten auswerten möchte, würde die ISR ggf. zu viel Zeit in Anspruch nehmen und damit das Echtzeitverhalten von HSE Free OSEK beeinträchtigen. Es könnte auch sein das man 2 Task's miteinander synchronisieren möchte und die TaskX erst auf die Abarbeitung von TaskY wartet. Um dies zu ermöglichen wird das Betriebssystemmittel Event eingeführt.
Events repräsentieren ein 1:1-Kommunikationsmittel mit dem Informationsgehalt von einem Bit. Sie tragen lediglich die Information „Event empfangen“ oder „kein Event empfangen“.Events können zur synchronen und zur asynchronen Kommunikation eingesetzt werden.

Das folgende Beispiel zeigt wie Events zur Synchronisation zwischen 2 Tasks verwendet werden können.

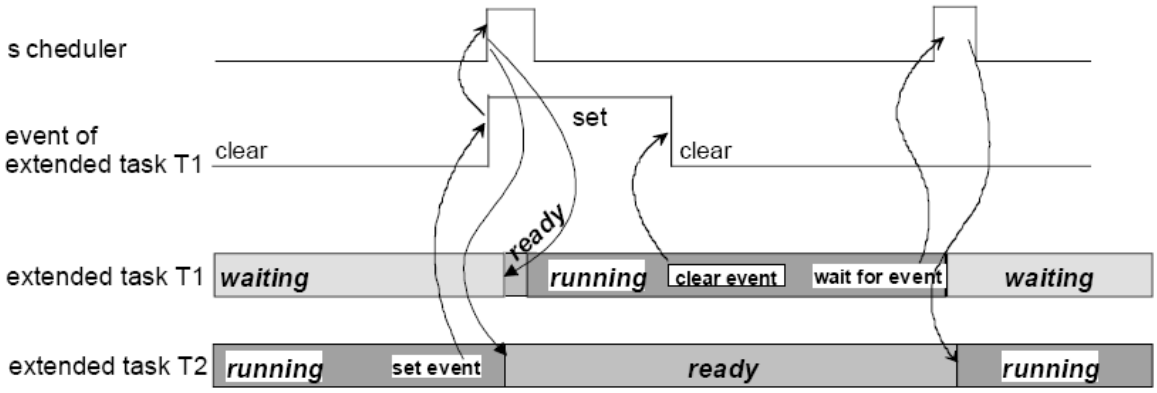


Abbildung 41: Beispiel mit Events
© OSEK/VDX Operating System Specification 2.2.3

Abbildung 41 zeigt 2 Tasks. Task1 wartet auf ein Event und befindet sich somit im Zustand Waiting. Task2 läuft und besitzt eine niedere Prioritäten als Task1. Zum Zeitpunkt t_x setzt Task2 das Eventflag und aktiviert somit den Scheduler. Dieser merkt das Task1 das Event erhalten hat auf das gewartet worden ist und entscheidet das Task1 laufen soll. Nach der Abarbeitung von Task1 setzt diese sich wieder in den Zustand Waiting und Task2 läuft weiter.

Events (Ereignisse) sind nach der OSEK/OS Spezifikation folgendermaßen definiert:

- Events sind immer Task-lokal, d.h. ein Event kann in einer Task A gesetzt sein, während das Event mit dem selben Namen in einer TaskB gelöscht ist.
- Alle Events die eine Task empfangen kann, müssen zuvor im TCB der Task bekannt gemacht werden
- nur Extended Task können Events empfangen und verarbeiten. Basic Tasks dürfen Events erzeugen

Da die Events im TCB einer Task gespeichert werden müssen, muss dafür Speicherplatz reserviert werden. Zur Speicherung von eingegangenen Events werden Bitfelder innerhalb des TCB benutzt.

In HSE Free OSEK müssen alle Events die man verwenden möchte, wie allen anderen Betriebssystemmittel, statisch erzeugt werden. Events besitzen einen Control Block welcher deklariert und initialisiert werden muss.

```
/*-----------------------------------------------------------------------*/
// Event Controll Block (ECB)
/*-----------------------------------------------------------------------*/
typedef struct os_event
{
  EventMaskType Mask;
}OS_EVENT;
```

Der Control Block ist im Verhältnis zu den anderen relativ winzig. Er enthält nur eine einziges Attribut (**MASK**) vom Typ ***EventMaskType***. Beim Typ von ***EventMaskType*** handelt es sich um einen OSEK konformen Typ der entweder 8,16,32 oder 64 Bit groß ist. Der Anwender hat die Möglichkeit die Größe selbst festzulegen. Die Größe der Eventmaske wird in der Datei „*../Src/OSEK Hardware Specific/typedef.h*“ festgelegt. Dort befindet sich die Konstante **EVENT_MASK_SIZE**. Gültige Werte sind 8,16,32 oder 64.

```
#define         EVENT_MASK_SIZE                 8               // 8/16/32/64 Bit
```

Nachdem man sich für die Größe der Eventmasken entschieden hat wird der Typ ***EventMaskType*** in der Datei „*../Src/OSEK Code/OSEK_TYPES*“ folgendermaßen festgelegt:

```
/*-----------------------------------------------------------------------*/
//  Event Typendefinition
/*-----------------------------------------------------------------------*/
#if    (EVENT_MASK_SIZE == 8)
    typedef INT8U EventMaskType;
#elif  (EVENT_MASK_SIZE == 16)
  typedef INT16U  EventMaskType;
#elif  (EVENT_MASK_SIZE == 32)
  typedef INT32U  EventMaskType;
#else  (EVENT_MASK_SIZE == 64)
  typedef INT64U  EventMaskType;
#endif
```

Hinweis:

Im unterschied zur OSEK Spezifikation kann eine erzeugte Eventmaske in allen Extended Tasks lokal verwendet werden. HSE Free OSEK weicht hiermit von der Spezifikation ab.

3.8.1 Initialisierung von Eventmasken

Im Projektverzeichnis des HSE Free OSEK Projekts findet man im Verzeichnis „../Src/Application/Configuration/“ die Dateien config.c und config.h. In diesem beiden Dateien findet die statische Initialisierung der Betriebssystemmittel statt. Zuerst betrachten wird uns die Task Konfiguration in der config.h:

```
/*-----------------------------------------------------------------*/
//EVENT-KONFIGURATION
/*-----------------------------------------------------------------*/
#define     MAX_NR_OF_EVENTS        1                   (1)

#define     EVENT1                  0                   (2)
#define     EVENT1_MASK             0x03                (3)

#define     EVENT1_MASK_BIT0        0x01
#define     EVENT1_MASK_BIT1        0x02
```

(1): Die Konstante **MAX_NR_OF_EVENTS** legt fest wie viele Eventmasken vorhanden sind. In HSE Free OSEK ist eine Liste von Event Control Blocks vorhanden namens EventList[] über welche die ECB's zugegriffen wird. **MAX_NR_OF_EVENTS** legt die Größe der EventList fest.

(2): **EVENT1** ist der vom Anwender vergebene Name des Events und hinter diesem verbirgt sich auch der Identifier(ID) welche als Index von EventList[] verwendet wird. Sinngemäß muss die Vergabe der ID aufsteigend von 0 bis **MAX_NR_OF_EVENTS-1** vergeben werden.

(3): Die Konstante **EVENT1_MASK** ist die Eventmaske auf die eine Task wartet. Diese könnte in diesem Beispiel auch 0xFF sein. Je nach Anwendung wird diese vom Anwender festgelegt.

An dieser Stelle hat der Anwender auch die Freiheit die einzelnen Bits der Gesamtmaske selbst zu erzeugen z.B. **EVENT1_MASK_BIT0** und **EVENT1_MASK_BIT1**. Es wäre ja möglich das unterschiedliche Tasks, ein Alarm oder eine ISR nur ein Bit der Gesamtmaske setzen und eine Task befindet sich solange im Zustand Waiting bis die Maske vollständig gesetzt worden ist.

Nachdem nun die Eventmaske angelegt worden ist muss nun der Event Control Block initialisiert werden. In der Datei config.c findet man die Funktion ***void EventInit(void)***.

```
// Initialisierung der einzelnen Events
void EventInit(void)
{
        EventList[EVENT1].Mask = EVENT1_MASK;
        ...
}
```

Letztendlich wird die Eventmaske festgelegt. Im Kapitel X.X werden die Funktionen zur Verwendung der Events detaillierter vorgestellt. Intern verwenden die Event-Methoden immer die EventList[] und die vom Anwender festgelegten Event-Namen und Identifier. Im Kapitel 3.4.1 Initialisierung des Task Control Blocks wurde ja bereits auf die beiden Attribute des Task Control Blocks **TCBEvent** und **TCBWaitEvent** erwähnt.
Da eine Event-Maske Task lokal ist, wird die Gesamtmaske auf das die Task wartet in **TCBWaitEvent** beschrieben. Das setzten der einzelnen Eventbits durch den Aufruf von ***SetEvent(...)*** wird im Attribut **TCBEvent** abgelegt und mit **TCBWaitEvent** verglichen.

3.9 Resource management

Ein Problem das Multitasking mit sich bringt ist die gemeinsame Verwendung von globalen Variablen, Programm-Sequenzen, Speicher und Hardwarekomponenten. Es ist zwingend notwendig bei der Verwendung von gemeinsamen Ressourcen sicherzustellen das nur 1 Task eine Ressource verwendet und erst nach der Freigabe eine andere Task die Ressource belegen kann. Der Zugriff auf eine Ressource sieht daher folgendermaßen aus:

```
void Task(TaskX)
{
...
//Ressource belegen
GetResource(RS232);
...
        RS232Send("HALLO");
...
//Ressource wieder freigegeben
ReleaseResource(RS232);

...
}
```

Somit kann die verwendete Ressource nur noch von TaskX verwendet werden. Angenommen TaskX hätte eine niedere Priorität und würde die Ressource belegen und kurz darauf vom Scheduler unterbrochen werden, der eine Task mit einer höheren Priorität (TaskY) aufruft. Die TaskY ist nun im Zustand Running (TaskX im Waiting-Zustand) und möchte die Resource ebenfalls belegen. Sie würde nach dem Aufruf von GetResource die Fehlermeldung **E_OS_ACCESS** erhalten was bedeutet das die Ressource belegt ist und TaskY keinen Anspruch darauf hat.

Unter Umständen kann es bei der Verwendung von Ressourcen (binäre Semaphore) zu Deadlocks kommen. Um Deadlocks zu vermeiden, schreibt die OSEK Spezifikation das Priority Celling Protokoll vor. Dieses legt folgende Punkte fest:

- Bei der Generierung des Systems wird vom Anwender jeder Ressource eine statische Priorität zugewiesen
- Die Priorität für die Ressource ist nach der höchst priori Task zu vergeben, welche auf die Ressource zugreift. Sie muss entweder Identisch sein oder größer. Die Priorität muss allerdings kleiner sein als die kleinste Priorität aller Tasks die nicht auf die Ressource zugreifen
- Wenn eine Task nun auf die Ressource zugreift, erhält die Task die Priorität der Ressource.
  Wenn die Task die Ressource freigibt, wird ihre ursprüngliche Priortitätsstufe wiederhergestellt.

Das folgende Beispiel soll das Priority Celling veranschaulichen.
Angenommen es liegt ein System mit 5 Tasks vor (T0-T4). T0 hat die höchste Priorität und belegt keine Ressource. T1-T4 verwenden alle eine gemeinsame Ressource. T1 hat dabei die höchste ,T4 die niederste Priorität (s.h. Abb. 42).

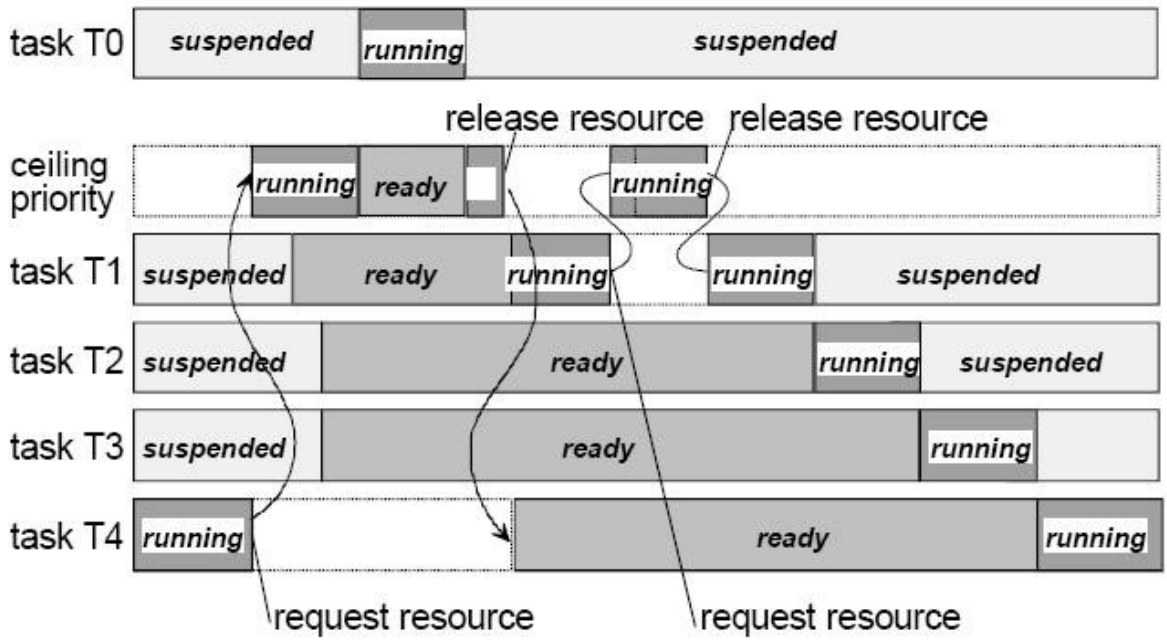


Abbildung 42: Verwendung von Ressourcen
©OSEK/VDX Operating System Specification 2.2.3

Angenommen T4 würde laufen und als erste die Ressource belegen. Sie erhält nun eine höhere Priorität als T1. Daraufhin entscheidet der Scheduler das T1-T3 aktiviert werden. Da jedoch T4 die Ressource belegt, ist ihre Priorität im Moment höher. D.h. T1 kann erst in den Zustand Running wechseln nachdem T4 die Priorität wieder freigegeben hat.

Im Kapitel 3.4 Der Task Control Block wurden die beiden Attribute des Task Control Blocks **TCBPrio** und **TCBStdPrio** bereits erwähnt. Der im Beispiel beschriebene Vorgang wird in HSE Free OSEK durch die API-Funktionsaufrufe ***GetResource(ResourceID)*** und ***ReleaseResource(ResourceID)*** realisiert. Das Attribut **TCBPrio** erhält die Priorität der Ressource beim aufruf von ***GetResource(...)***. Erst bei der Freigabe der Ressource durch ***ReleaseResource(ResourceID)*** wird die originale Priorität über das Attribut **TCBStdPrio** wiederhergestellt.

3.9.1 Initialisierung von Ressourcen

Ressourcen müssen in HSE Free OSEK statisch erzeugt werden. Sie besitzen wie alle anderen Betriebssystemmittel auch einen Resource Control Block der folgendermaßen aufgebaut ist:

```
// Resource Controll Block (RCB)
 typedef struct os_resource
 {
  INT8U    ResID;                                    (1)
  INT8U    ResourceProperty;                         (2)
  BOOLEAN  Locked;                                   (3)

  INT8U    ResourcePrio;                             (4)

 }OS_RESOURCE;
```

(1): Das Attribut **ResID** ist der Identifier der Ressource. Der Identifier dient als Index für das ResourceList[] Array, welches alle Resource Control Blocks verwaltet.

(2): Das Attribut **ResourceProperty** beschreibt die Eigenschaft der Ressource. Es gibt 3 Typen von Ressourcen (STANDART, LINKED & INTERNAL). Da HSE Free OSEK nur den Standardtyp verwendet, wird hier darauf nicht näher eingegangen.

(3): Das Attribut **Locked** sagt aus ob die Ressource frei (false) oder belegt (true) ist.

(4): Das Attribut **ResourcePrio** enthält die Priorität der Ressource.

Im Projektverzeichnis des HSE Free OSEK Projekts findet man im Verzeichnis „../Src/Application/Configuration/“ die Dateien config.c und config.h. In diesem beiden Dateien findet die statische Initialisierung der Betriebssystemmittel statt. Zuerst wird Resource Konfiguration in der config.h bearbeitet:

```
/*-----------------------------------------------------------------------*/
//RESOURCE-KONFIGURATION
/*-----------------------------------------------------------------------*/

#define MAX_NR_OF_RESOURCES      1

#define RESOURCE1                0
#define RESOURCE1_PRIO           51
```

(1): Die Konstante **MAX_NR_OF_ RESOURCES** legt fest wie viele Ressourcen vorhanden sind. In HSE Free OSEK ist eine Liste von Resource Control Blocks vorhanden namens ResourceList[] über welche auf die RCB's zugegriffen wird. **MAX_NR_OF_ RESOURCES** legt die Größe der EventList fest.

(2): **RESOURCE1** ist der vom Anwender vergebene Name der Ressource und hinter diesem verbirgt sich auch der Identifier(ID) welcher als Index von ResourceList[] verwendet wird. Sinngemäß muss die Vergabe der ID aufsteigend von 0 bis **MAX_NR_OF_RESOURCES-1** vergeben werden.

(3): Die Konstante **RESOURCE1_PRIO** definiert die Priorität der Ressource

Nachdem nun die Ressource angelegt worden ist muss nun der Ressource Control Block initialisiert werden. In der Datei config.c findet man die Funktion ***void RessourceInit(void)***.

```
/*----------------------------------------------------------------------*/
// ResourceInit():
/*----------------------------------------------------------------------*/
void ResourceInit(void)
{

  ResourceList[RESOURCE1].ResourceProperty = STANDARD;
  ResourceList[RESOURCE1].ResourcePrio = RESOURCE1_PRIO;
  ResourceList[RESOURCE1].Locked = FALSE;
  */
}
```

Damit wäre die Initialisierung der Ressource abgeschlossen.

3.10 Timer, Counter und Alarme

Zu einem Echtzeitbetriebssystem gehören auch Instanzen welche sich nach der Zeit richten. Im allgemeinen besitzen alle µC Timer (Zeitgeber). Timer sind Steuerbausteine, die zur Realisierung der unterschiedlichsten zeitbezogenen Funktionen sowie als Zähler eingesetzt werde. Timern können unabhängig voneinander betrieben werden. Mögliche Einsatzgebiete ergeben sich beispielsweise als Impulsgenerator, als Taktgeber, als Ereigniszähler oder einfach nur zur Zeitmessung.
Jeder µC besitzt eine unterschiedliche Anzahl an Hardware-Timern welche meistens auch für die Anwendungen selbst verwendet werden. HSE Free OSEK und die meisten anderen Echtzeitbetriebssysteme benötigen 1 Timer der als Taktgeber der Betriebssystems dient. Da es vorkommen kann das zu wenig Hardware-Timer vorhanden sind besitzt HSE Free OSEK den Counter Mechanismus. Counter sind im von der Funktionalität identisch zu den Timern, jedoch werden sie per Software getriggert und der Betriebssystemtimers gibt den Takt für alle Counter vor. Das bedeutet das alle Counter die maximale Auflösung des Betriebssystemtimers besitzen können.
In HSE Free OSEK arbeitet der Timer mit einer Taktrate von 10ms. Das heißt das jeder Counter alle 10ms getriggert werden kann oder einem ganzzahligen vielfachen der Zykluszeit (20ms, 30ms, .. 1000ms (1s), usw.).

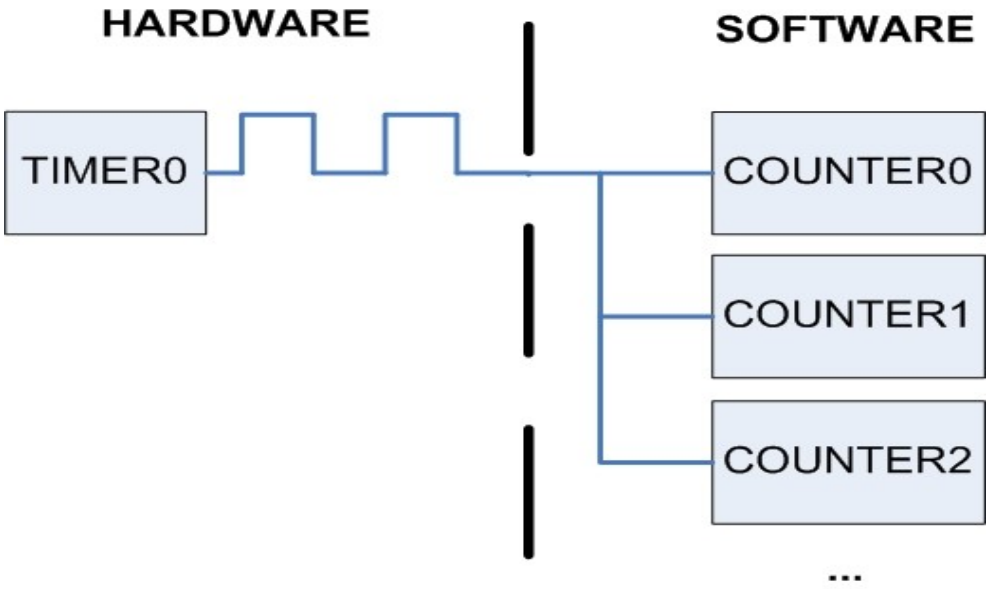


Abbildung 43: Hardware Timer gibt den Takt für Counter (Software-Timer) vor

Um nun der Echtzeit gerecht zu werden, muss ein Echtzeitbetriebssystem die Fähigkeit besitzen, auf zeitgesteuerte Ereignisse, welche zyklisch eintreffen oder während der Laufzeit entstehen, reagieren zu können. Beispielsweise soll eine Task alle 100ms aktiviert werden. Um diese Feature anzubieten wird das Betriebssystemmittel „Alarm“ verwendet, welches je nach Anwendung entweder ein Event erzeugen kann oder eine Task aktiviert bzw. in den Zustand Ready/Running versetzt.
Ein Alarm wird vom Anwender einem Counter zugewiesen, welcher die Entsprechende Zykluszeit erfüllt. Alarme können ab dem Starten von HSE Free OSEK aktiviert sein oder

auch während der Anwendung gestartet werden. Die ist vom Anwender anzugeben sowie auch ob es sich um zyklische Alarme oder einmalige handelt. Für das triggern der Counter (Software-Timer) benötigt HSE Free OSEK eine ISR, welche über den für das Betriebssystem zur Verfügung gestellten Timer getriggert werden.

Hardware-Timer können vom unterschiedlichen Typ sein (8/16/32 Bit). Der Datentyp TickType ist OSEK konform und dient dazu den Wertebereich des Timer festzulegen. In der Typedef.h ist der Timer Typ mit der Konstanten **COUNTERSIZE** festzulegen. Darauf hin wird der Typ TickType definiert (s.h. OSEK_TYPES.h):

```
/*-----------------------------------------------------------------------*/
//Definition von Ticks
/*-----------------------------------------------------------------------*/
#if(COUNTERSIZE == 8)
  typedef INT8U     TickType;
#elif(COUNTERSIZE == 16)
  typedef INT16U    TickType;
#elif(COUNTERSIZE == 32)
  typedef INT32U
  TickType;
#endif
```

Alle Counter besitzen einen Counter Control Block welcher in einer Struktur namens OS_Counter definiert ist (s.h. OSEK_TYPES.h) und folgende Attribute besitzt:

```
/*-----------------------------------------------------------------------*/
// Counter Control Block
/*-----------------------------------------------------------------------*/
typedef struct os_counter
{
  INT8U      CounterID;                                          (1)

  TickType  MAXALLOWEDVALUE;                                     (2)
  TickType  TICKSPERBASE;                                        (3)
  TickType  MINCYCLE;                                            (4)

  TickType  COUNTERVALUE;                                        (5)
  TickType  TICKSPERVALUE;                                       (6)

  AlarmBaseRefType  AlarmHead;                                   (7)

}OS_COUNTER;
```

(1): **CounterID** ist der Identifier des Counters. Alle im HSE Free OSEK existenten Counter sind in einer statischen Liste (Array) angelegt namens CounterList[]. Der Identifier entspricht dem Index des Arrays. Der Betriebssystem Counter OS_CTR besitzt die **CounterID = 0**. Es ist ebenfalls möglich Alarme dem Betriebssystem Counter zuzuweisen.

(2): Das Attribut **MAXALLOWEDVALUE** gibt den maximal erreichbaren Wert für den Counter an. Wird er bei diesem Wert inkrementiert, erhält er danach den Wert 0 (wrap-around).

(3): Mit dem Attribut **TICKSPERBASE** wird angegeben, wie oft der Timer getriggert werden muss, um ihn einmal inkrementieren zu werden – der Standardwert dafür ist 1. Getriggert wird ein Counter durch den Aufruf der OSEK-Systemfunktion ***CounterTrigger()***, welche sich in der „../Src/OSEK Code/OS_Counter.c“ befindet. Die Funktion ***CounterTrigger()*** wird aus der SystemTask aufgerufen welche später genauer erläutert wird.

(4): Das Attribut **MINCYCLE** gibt an, wie viele Ticks des Counters die Periode eines zyklischen Alarms mindestens umfassen muss. Der standardmäßig eingestellte Wert von 1 sagt somit aus,dass die minimale Periode eines zyklischen Alarms einen Countertick lang ist. Der Wert 0 sagt aus das es sich um einen einmaligen Alarm handelt.

Bemerkung: Die Attribute (2- 4) sind Attribute welche in der OSEK-Spezifikation festgelegt sind.

(5): **COUNTERVALUE** ist der Zählerstand des Counters.

(6): Das Attribut **TICKSPERVALUE** wird bei jedem Aufruf von ***CounterTrigger()*** inkrementiert. Erst wenn **TICKSPERVALUE** den Wert **TICKSPERBASE** erreicht, wird das Attribut **COUNTERVALUE** inkrementiert. **TICKSPERVALUE** wird dann auf 0 zurückgesetzt.

(7): **AlarmHead** ist ein Pointer vom Typ ***AlarmBaseRefType.*** Der Pointer zeigt auf einen Alarm Control Block welcher im Anschluss erklärt wird. Alarme werden Counter zugewiesen. Es ist auch möglich das einem Counter mehrere Alarme zugewiesen werden. Der Pointer AlarmHead zeigt also entweder ins Leere oder auf eine Verkettete Liste von Alarmen welche dem Counter zugewiesen sind.

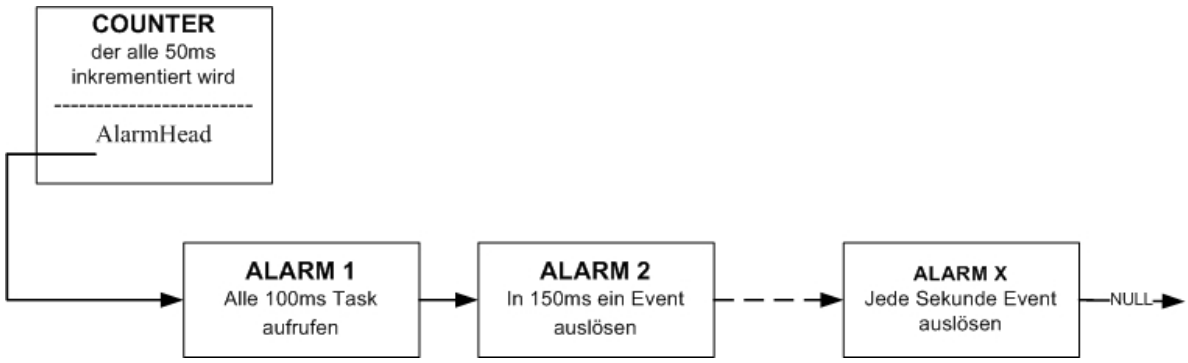


Abbildung 44: Alarme die einem Counter zugewiesen sind

Abbildung 44 zeigt ein Beispiel in welchem 3 Alarme einem Counter zugewiesen sind. Der Zählwert des Counters wird alle 50ms inkrementiert. Angenommen der Hardware Timer würde alle 10 ms den Betriebssystem Interrupt ***ISR(OSTimer)*** aufrufen (OS_System.c). Dann muss das Counter Attribut **TICKSPERBASE** den Wert 5 besitzen.
Sobald ein Alarm einem Counter zugewiesen ist, kann er nur mit dessen Auflösung arbeiten. In diesem Beispiel wäre es nicht möglich einen Alarm bei 75ms auszulösen. Auch wenn ein Alarm inaktiv ist bleibt er weiterhin dem Counter zugewiesen.

Alle Alarme besitzen einen Alarm Control Block welcher in einer Struktur namens OS_Alarm definiert ist (s.h. OSEK_TYPES.h) und folgende Attribute besitzt:

```
/*-------------------------------------------------------------------*/
// Alarm Controll Block
/*-------------------------------------------------------------------*/

typedef struct os_alarm
{
  INT8U    AlarmID;              //ID des Alarms                    (1)
  INT8U    CounterID;            //ID des Counters                  (2)
  TASKID AlarmTaskID;            //ID der Task                      (3)

  INT8U    AlarmAction;          //Alarm Aktion                     (4)
  EventMaskType AlarmEvent;      //Eventmaske                       (5)

  TickType  AlarmTime;           //Alarm Zeit                       (6)
  TickType  AlarmCycleTime;      //Alarm ZyklusZeit                 (7)

  //Beschreibt ob Alarm aktiv oder nicht
  BOOLEAN   AlarmStat;                                            (8)

  //pointer auf nächsten Alarm
  struct os_alarm *nextAlarm;                                     (9)

}OS_ALARM;
```

(1): **AlarmID** ist der Identifier des Alarms. Alle im HSE Free OSEK existenten Alarme sind in einer statischen Liste (Array) angelegt namens AlarmList[]. Der Identifier entspricht dem Index des Arrays.

(2): Das Attribut **CounterID** bezieht sich auf den Counter welchem der Alarm zugewiesen ist.

(3): Das Attribut **AlarmTaskID** bezieht sich auf die Task welche vom Alarm aktiviert wird oder auf ein Event wartet das der Alarm auslöst.

(4): Das Attribut **AlarmAction** enthält die Information ob der Alarm eine Task aufrufen (in den Zustand Ready/Running versetzen) soll oder ein Event für eine Task auslösen soll. Es sind 2 Konstanten dafür vorgesehen, ALARM_ACTION_ACTIVATE_TASK und ALARM_ACTION_SET_EVENT.

(5): Das Attribut **AlarmEvent** enthält eine Eventmaske. Falls ein Alarm ein Event auslösen soll, wird dem **AlarmEvent** die jeweilige Eventmaske zugewiesen.

(6): **AlarmTime** ist die Zeitangabe wann der Alarm ausgelöst werden soll. In der SystemTask werden vom jeweiligen Counter das Attribut **COUNTERVALUE** und **AlarmTime** miteinander verglichen und falls diese gleich sind die entsprechende Aktion durchgeführt.

(7): Falls es sich um einen zyklischen Alarm handelt, enthält das Attribut **AlarmCycleTime** die Angabe wie viele Ticks bis zum nächsten Auftreten vergehen sollen. Bei einmaligen Auftreten des Alarms ist die **AlarmCycleTime** gleich null.

(8): Das Attribut **AlarmStat** sagt aus ob der Alarm aktiv oder inaktiv ist.

(9): Für die Realisierung der bereits erwähnten verketteten Liste, ist noch der Pointer **nextAlarm** notwendig. Die Liste ist abgeschlossen wenn dieser den Wert NULL enthält.

Die bisher erwähnten Betriebssystemmittel Counter und Alarme besitzen nach der OSEK Spezifikation Methoden um während der Laufzeit Einstellungen und Veränderungen der Attribute der Control Blocks vorzunehmen. Diese werden in Kapitel 4 detaillierter beschrieben.

3.10.1 Initialisierung von Countern

Im Projektverzeichnis des HSE Free OSEK Projekts findet man im Verzeichnis „../Src/Application/Configuration/“ die Dateien config.c und config.h. In diesem beiden Dateien findet die statische Initialisierung der Betriebssystemmittel statt. Zuerst betrachten wird uns die Counter Konfiguration in der config.h:

```
/*-----------------------------------------------------------------------------------------*/
// COUNTER-KONFIGURATION
/*-----------------------------------------------------------------------------------------*/

//Anzahl Soft-Counter + OS TICK COUNTER
#define MAX_NR_OF_COUNTERS        (2+1)                          (1)


#define CTR1_ID                   1                              (2)
#define CTR1_MAXALLOWEDVALUE      0xFFFFFFFF                     (3)
#define CTR1_TICKSPERBASE         5                              (4)
#define CTR1_MINCYCLE             1                              (5)

//jede 1s                                                      (6)
#define CTR2_ID                   2
#define CTR2_MAXALLOWEDVALUE      59
#define CTR2_TICKSPERBASE         100
#define CTR2_MINCYCLE             1
```

(1): Die Konstante **MAX_NR_OF_COUNTERS** ist fest vorgegeben und muss mindestens den Gesamtwert 1 besitzen, da der Betirebssystem Counter OS_CTR intern in HSE Free OSEK vorhanden ist. **MAX_NR_OF_COUNTERS** beschreibt wie viele Coter im System vorhanden sind. In diesem Beispiel wird für die Anwendung nur 2 Counter erzeugt.

(2): **CTR1_ID** ist der Identifier des erzeugten Counters. Alle erzeugen Counter sind in der globalen Liste CounterList[] auffindbar. Der Identifier entspricht dem Array Index.

(3): Das Attribut **CTR1_MAXALLOWEDVALUE** gibt den maximalen Zählstand des Counters an. Wenn der Zählwert (**COUNTERVALUE**) den maximalen Zählwert erreicht hat, wird **COUNTERVALUE** wieder auf 0 gesetzt (wrap arround).

(4): Das Attribut **CTR1_TICKSPERBASE** gibt vor wie oft der Hardware-Timer auftreten muss um den Counter ein Mal zu inkrementieren. Angenommen der Timer tritt alle 10ms auf, dann würde sich **COUNTERVALUE** erst nach 50ms den Wert 1 besitzen.

(5): Das Attribut **CTR1_MINCYCLE** gibt die Zykluszeit vor. Bei einem einmaligen Alarm beträgt **CTR1_MINCYCLE** den Wert 0.

(6): Der Counter CTR2 ist ein weiteres Beispiel zur Initialisierung eines Counters. Ausgegangen von einem Tick nach 10 ms, würde dieser das Attribut **COUNTERVALUE** nach einer Sekunde inkrementieren.

Nachdem man nun die wichtigsten Attribute für den Counter Control Block festgelegt hat folgt nun die Initialisierung im System selber. In der Datei config.c ist die Funktion ***void CounterInit(void)*** zu finden, in welcher alle vom Anwender definierten Counter initialisiert werden müssen. Später wird die Funktion vom Betriebssystem aus aufgerufen.

```
/*-----------------------------------------------------------------------*/
// Initialisierung der einzelnen COUNTER
/*-----------------------------------------------------------------------*/
void CounterInit(void)
{

CounterList[CTR1_ID].CounterID = CTR1_ID;

CounterList[CTR1_ID].MAXALLOWEDVALUE = CTR1_MAXALLOWEDVALUE;
CounterList[CTR1_ID].TICKSPERBASE = CTR1_TICKSPERBASE;
CounterList[CTR1_ID].MINCYCLE = CTR1_MINCYCLE;

CounterList[CTR1_ID].COUNTERVALUE = 0;
CounterList[CTR1_ID].TICKSPERVALUE = 0;
CounterList[CTR1_ID].AlarmHead = NULL;                                  (1)
...
}
```

(1): Dem Pointer AlarmHead werden alle Alarme die den Counter verwenden in einer einfach verketteten Liste zugewiesen.

Damit wäre die Erstellung von Counters abgeschlossen.

3.10.2 Initialisierung von Alarmen

Im Projektverzeichnis des HSE Free OSEK Projekts findet man im Verzeichnis „../Src/Application/Configuration/“ die Dateien config.c und config.h. In diesem beiden Dateien findet die statische Initialisierung der Betriebssystemmittel statt. Zuerst betrachten wir uns die Alarm Konfiguration in der config.h:

```
/*-----------------------------------------------------------------------------------------*/
//ALARM-KONFIGURATION
/*-----------------------------------------------------------------------------------------*/
#define MAX_NR_OF_ALARMS        2                                   (1)

#define ALARM1_ID               0                                   (2)
#define ALARM2_ID               1

#define ALARM1_TIME             5                                   (3)
#define ALARM2_TIME             10
```

(1): Der Anwender muss die Anzahl der Alarme dem Betriebssystem bekannt machen. Dies tut man über das Attribut **MAX_NR_OF_ALARMS**.

(2): Des weiteren ist für jeden einzelnen Alarm ein eindeutiger Identifier festzulegen. Jeder Alarm wird in der AlarmList[] verwaltet und über den Identifier angesprochen.

(3): In Kapitel 3.7.1 sind im Beispiel 2 Counter erzeugt worden. Angenommen Alarm1 würde dem Counter1 zugewiesen werden und Alarm2 an Counter2. Counter1 wird alle 50ms inkrementiert, d.h. Alarm1 würde nach 5*50ms = 250ms auftreten. Alarm2 erst nach 10*1000ms = 10s. Die Festlegung der Alarmzeit obliegt dem Anwender.

Nachdem man nun die wichtigsten Attribute für den Alarm Control Block festgelegt hat folgt nun die Initialisierung im System selber. In der Datei config.c ist die Funktion ***void AlarmInit(void)*** zu finden, in welcher alle vom Anwender definierten Alarme initialisiert werden müssen. Später wird die Funktion vom Betriebssystem aus aufgerufen.

```
void AlarmInit(void)
{
 AlarmList[ALARM1_ID].AlarmID                = ALARM1_ID;
 AlarmList[ALARM1_ID].CounterID              = CTR1_ID;
 AlarmList[ALARM1_ID].AlarmTime              = ALARM1_TIME;
 AlarmList[ALARM1_ID].AlarmCycleTime         = ALARM1_TIME;
 AlarmList[ALARM1_ID].AlarmAction    = ALARM_ACTION_ACTIVATE_TASK;
 AlarmList[ALARM1_ID].AlarmTaskID            = TASK1_ID;
 AlarmList[ALARM1_ID].AlarmEvent             = EVENTMASK_NONE;

 AllocatAlarmToCounter(CTR1_ID, ALARM1_ID);
...
}
```

Die Attribute in diesem Listening sind nach den vorherigen Erklärungen festzulegen. Neu hinzugekommen ist die Funktion ***AllocatAlarmToCounter()***. Dies ist eine Betriebssystemfunktion die nicht zu OSEK API gehört. Sie dient lediglich dazu den erzeugten Alarm an den jeweiligen Counter zuzuweisen. Zur genaueren Betrachtung findet man die Funktion ***AllocatAlarmToCounter()*** im Ordner „../Src/OSEK Code/OS_Counter.c“.

3.11 Betriebssystem Tasks

Das HSE Free OSEK besitzt zwei Betriebssystem Task's, die Idle Task und die Kernel Task. Diese Task's haben Betriebssystem interne Funktionen und werden nie vom User selber aufgerufen.

3.11.1 Idle Task

Die Idle Task besitzt die kleinste Priorität (Priorität = 0) und ist immer dann aktive, wenn grad keine andere Task oder Interrupt am laufen ist. Die Idle Task ist eine Basic Task, d.h. sie kann nie in den Zustand Waiting gesetzt werden. Es handelt sich um eine preemptive Task die auch nie den Zustand Suspended haben kann. Die Task wird vom Betriebssystem automatisch aufgerufen, damit das Betriebssystem keinen nicht vorhersehbaren Zustand einnimmt.

```
/*-----------------------------------------------------------------------*/
// IdleTask():
/*-----------------------------------------------------------------------*/
TASK(IdleTask)
{
        while(1)
        {

        }
}
```

3.11.2 Kernel Task

Die Kernel Task besitzt die höchste Priorität (Priorität = 64) und wird immer durch den System Interrupt aufgerufen. Da der System Interrupt alle 10ms auftritt, muss auch die Kernel Task alle 10ms aufgerufen werden. Diese Task darf nie von einer anderen Stelle aufgerufen werden als aus dem System Interrupt. Diese Task hat die Aufgabe

```
/*-----------------------------------------------------------------------*/
// TASK(KernelTask):
/*-----------------------------------------------------------------------*/
TASK(KernelTask)
{
CtrType CtrId;
AlarmBaseRefType pAlarm;

//Inkrementieren der einzelnen Counter
for(CtrId = 0; CtrId < MAX_NR_OF_COUNTERS; CtrId++)                    (1)
{
   while( CounterTrigger(CtrId) != E_OK){}                             (2)
   pAlarm = CounterList[CtrId].AlarmHead;
```

```
    while(pAlarm != NULL)                                                    (3)
    {
      //Abfrage ob ein Alarm auftreten soll
      if((CounterList[CtrId].COUNTERVALUE == pAlarm->AlarmTime) &&
        (pAlarm->AlarmStat == TRUE))                                         (4)
      {
        if(pAlarm->AlarmAction == ALARM_ACTION_ACTIVATE_TASK)                (5)
        {
          while( ActivateTask(pAlarm->AlarmTaskID) != E_OK ){}               (6)
        }
        else if(pAlarm->AlarmAction == ALARM_ACTION_SET_EVENT)               (7)
        {
          while( SetEvent(pAlarm->AlarmTaskID,pAlarm->AlarmEvent) != E_OK ){} (8)
        }
        //wenn es sich um einen Zyklischen Alarm handel
        if(pAlarm->AlarmCycleTime != 0)                                      (9)
        {
          pAlarm->AlarmTime = CounterList[CtrId].COUNTERVALUE + pAlarm->
          AlarmCycleTime;
        }
        else
        {
          pAlarm->AlarmStat = FALSE;
        {
      }
      pAlarm = pAlarm->nextAlarm;
    }
}
//Das Beenden der System Task ist abhängig davon ob eine Unterbrechbare
//oder nicht Unterbrechbare (None_Scheduled) aufgerufen war

if(pTCBInterruptedTask == NULL)                                              (10)
{
    while (TerminateTask() != E_OK){}
}
else //NONE_SCHEDULED
{
    //Task in Zustand Suspended versetzen
    pTCBCurRun->TCBStat = SUSPENDED;

    #ifdef LPC2148
    //Stack der Task in Ausgangszustand versetzen
    pTCBCurRun->TCBStackPtr = pTCBCurRun->TCBContextStack;
    TaskStackInit(pTCBCurRun);
    #endif
    pTCBPreRun = pTCBCurRun;
    //Unterbrochene Task wieder herstellen
    pTCBCurRun = pTCBInterruptedTask;
```

```
    pTCBInterruptedTask = NULL;
    #ifdef LPC2148
    __SingleCtxSw();
    #endif

    #ifdef HCS12
    HCS12ResetStackAndSwitch();
    #endif
    }
}
```

(1): In der Konstanten **MAX_NR_OF_COUNTERS** steht die Anzahl der verwendeten Counters. Alle Counter werden in der while Schleife durchlaufen und Inkrementiert.

(2): Die Funktion ***CounterTrigger()*** zählt den übergebenen Counter hoch und gibt, wenn es sich um eine gültige ID handelt, den Wert E_OK zurück.

(3): pAlarm ist ein Zeiger auf den ersten Alarm in der Liste. Einem Counter können mehrere Alarme zugewiesen werden. Die Alarme sind wiederum mittels einer verketteten Liste miteinander verbunden. Nun wird die while Schleife so lange durchlaufen bis der letzte Alarm in der Liste der zu dem entsprechenden Counter gehört, abgearbeitet wurde.

(4): Hier wird abgefragt ob die Zeit bei dem der Interrupt aufgetreten ist mit der Zeit bei der ein Alarm auftreten soll, übereinstimmt.

(5): Wenn die Zeit übereinstimmt, dann wird überprüft was der Alarm machen soll. In diesem Fall wird danach gefragt ob der Alarm eine Task aktivieren soll.

(6): Wenn eine Task aktiviert werden soll, dann geschieht das mittels der Funktion ***ActivateTask()***.

(7): Abfrage ob der Alarm ein Event setzen soll.

(8): setzen des Events mittels der Funktion ***SetEvent()***.

(9): Alarme könne auch zyklisch sein, d.h. der Alarm wird immer nach einem bestimmten Zyklus, welcher in der Konfigurationsfile File statisch festgelegt wurde, aufgerufen. Aus diesem Grund muss dem Alarm die Zyklus Zeit dazu addiert werden.

(10): Wenn alle Counter abgearbeitet wurden, wird die Kernel Task beendet. Jetzt wird überprüft ob die Task, die durch den Interrupt unterbrochen wurde, preemptiv oder nicht preemptiv ist. Wenn die Task nicht unterbrechbar (NONE_SCHEDULED) ist dann muss die Task wieder an der Stelle gestartet werden wo sie unterbrochen wurde. Wenn die Task unterbrechbar ist (FULL_SCHEDULED) dann wird die Task gestartet welche die höchste Priorität besitzt und im Zustand Ready ist.

3.12 Starten von HSE Free OSEK

Um HSE Free OSEK letztendlich zu starten muss die Datei geöffnet werden in welcher sich die Main-Funktion befindet bzw. die Funktion welche nach dem Controller spezifischen Startup aufgerfuen wird. In diesem Beispiel wäre die Datei „../Src/Application Code/main.c“.

```
void main(void)
{
        //Initialisierung der Hardware
        HardwareInit();

        //Starten des Betriebssystems
        StartOS();
}
```

Vor dem Start von HSE Free OSEK sollten noch die Hardwarekomponenten initialisiert werden. HSE Free OSEK stellt in den Projektverzeichnis dafür eine Vorlage dar, welche aber beliebig verändert werden kann. Die Initialisierung der Hardwarekomponenten ist Hardware spezifisch und von daher kein Teil der OSEK Spezifikation.

Die OSEK API Funktion void StartOS(void) starten dann HSE Free OSEK. Sie ist im Verzeichnis „../Src/OSEK Code/OS_System.c“ zu finden und sieht folgendermaßen aus.

```
/*-----------------------------------------------------------------------*/
// StartOS:
/*-----------------------------------------------------------------------*/
void StartOS(void)
{
#ifdef LPC2148                                                          (1)
  SWI_SetSVCMode();
#endif

//Initialisierung des OSTimers
OSTimerInit();                                                          (2)

//Initialisierung von OS Komponenten
SystemTasksInit();                                                      (3)
SystemCounterInit();

//Aufruf der vom Anwender Konfigurieren Initialisierungsfunktionen
TCBInit();                                                              (4)
EventInit();
ResourceInit();
CounterInit();
AlarmInit();
```

```
#if (SCHEDULER == 0)                                                    (5)
  //Task's die Ready sind in die Ready Qeueu eintragen
  InitRdyList();
#elif (SCHEDULER == 1)
  InitPriorityList();
#endif

OSRunning = TRUE;                                                       (6)

#ifdef LPC2148                                                          (7)
  //Initialisierung der Interrupts
  InterruptInit();
#endif

  StartTask();       //höstpriore Task finden und in Zustand Running setzen   (8)

  __SingleCtxSw();    //Starten mit der höchstprioren Task            (9)
}
```

(1): Falls es sich um den LPC2148 handelt muss dieser im den SVC-Modus betrieben werden, da dies der einzigste Modus ist in welchem HSE Free OSEK auf das Prozessorstatuswort (CPSR) zugreifen kann. Dies geschieht mit dem Aufruf der Funktion SWI_SetSVCMode() welche sich in der Datei „../Src/OSEK Hardware Specific/OS_CPU.c“ befindet.

(2): Als nächstes wird die Funktion OSTimerInit() aufgerufen. Die Funktion befindet sich in der Datei „../Src/Interrupt Code/InterruptInit.c und ist hardwarespezifisch, da sich die Initialisierung von Controller zu Controller unterscheidet. Ihre Aufgabe ist es den für HSE Free OSEK ausgewählten Timer zu initialisieren.

(3): Der nächste Schritt ist die Initialisierung der Betriebssystemtask Idel und Kernel sowie die Initialisierung des OS_CTR (System Counters).

(4): Daraufhin folgt die eigentliche Systemerzeugung. Alle Betriebssystemelemente welche der Anwender erzeugt hat werden initialisiert.

(5): Daraufhin folgt die Auswahl des Scheduling-Algorithmus. Die Tasks die sich von Anfang an im Zustand Ready befinden werden entweder in der ReadyQueue oder in der ReadyTable aufgeführt. Beide Funktionen sind nur für das Betriebssystem intern verwendbar und befinden sich, wie StartOS() auch in der System.c.

(6): Die globale Variable OSRunning kennzeichnet ob das Betriebssystem läuft oder nicht.

(7): Die Betriebssystemfunktion StartTask() ist keine API-Funktion. Nur Betriebssystemfunktionen können sie aufrufen. Die Funktion ist in beiden Schedulingmechanismen definiert und findet die höchst prori Task. Somit ist sie der eigentliche Scheduler.

(8): Nun folgt der Single Context Switch ***__SingleCtxSw();.*** Dies ist eine etwas vereinfachte Funktion des eigentlichen Context Switch's. Sie wird selten verwendet, da der Context der Task die sie aufruft verloren geht. ***__SingleCtxSw()*** verwirft alle Daten wie SP,PC etc. und springt lediglich nur zu der Task welche sich im Zustand Ready befindet.

Somit ist die komplette Initialisierung von HSE Free OSEK abgeschlossen. Das Betriebssystem läuft. Die mitgelieferten Beispiele sollten als Referenz zu den bis hier hin beschrieben Abschnitten dienen.

3.13 Variablen, Strukturen und Funktionen von HSE Free OSEK

Die folgenden Tabellen sind für die Entwickler und Anwender gedacht die mit HSE Free OSEK arbeiten und die einen kurzen Überblick liefern der beim Entwickeln und Debuggen weiterhelfen sollen.

Globale Betriebssystem Variablen

Folgende Variablen/Arrays werden vom Betriebssystem intern verwendet

OS_TCB **TaskList**[MAX_NR_OF_TASKS]	Liste aller Tasks
OS_TCB ***pTCBCurRun**;	Pointer auf die Task die gerade läuft
OS_TCB ***pTCBPreRun**;	Pointer auf die Task welche zuvor gelaufen ist
OS_TCB ***pTCBNextRun**;	Pointer auf die Task welche als nächste an der Reihe ist
OS_ALARM **AlarmList**[MAX_NR_OF_ALARMS];	Liste aller Alarme
OS_COUNTER **CounterList**[MAX_NR_OF_COUNTERS];	Liste aller Counter
OS_EVENT **EventList**[MAX_NR_OF_EVENTS];	Liste aller Events
OS_RESOURCE **ResourceList**[MAX_NR_OF_RESOURCES];	Liste aller Resourcen
INT8U const **OSBitMapTbl**[]	Lookup Tabelle für das Bitmapscheduling
OS_TCB ***TCBPrioTbl**[MAX_PRIORITY + 1];	Bitmapscheduling Tabelle in welcher alle Ready Tasks aufgeführt sind
INT8U **OSReadyGrp**;	
INT8U **ReadyTable**[];	
INT8U **OSMapTbl**[]	
BOOLEAN **OSScheduleMode**	Gibt bekannt ob die gerade laufende Task sich preemptiv ist oder nicht

extern OS_STK ***pContextStack**;	Wird nur beim LPC2148 verwendet und zeigt auf den Context einer Task

Nur von Betriebssystemen verwendete Funktionen (keine API)

void **TCBInit**(void);	Initialisierung aller Task CB's
void **EventInit**(void);	Initialisierung aller Event CB's
void **CounterInit**(void);	Initialisierung aller Counter CB's
void **AlarmInit**(void);	Initialisierung aller Alarm CB's
void **ResourceInit**(void);	Initialisierung aller Resource CB's
void **AddFirstToReadyQueue**(OS_TCB *);	Scheduling mit verketteten Listen. Fügt einen TCB an die Spitze der Liste an
void **Reschedule**(TaskType TaskID);	Scheduling mit verketteten Listen. Sucht in der Liste nach der position der neu eingefügten Ready Task
void **InsertToPrioQueue**(OS_TCB *);	Fügt eine Task in die ReadyTabelle ein
void **SystemTasksInit**(void);	Initialisierung von SystemTask und IdleTask
void **SystemCounterInit**(void);	Initialisierung des Betriebssystem Counter OS_CTR
void **__ContextSwitch**(void);	ContextSwitch mit Taskwechsel
void ***__SingleCtxSw***(void)	Einfacher Context Switch
StatusType **AllocatAlarmToCounter** (CtrType CtrId, AlarmType AlarmId)	Fügt einem Counter einen Alarm hinzu
TickType **Add_Tick**(TickType almval, TickType incr, TickType maxval);	Einem Alarm seinen Zeitwert berechnen und zuweisen
StatusType **CounterTrigger**(CtrId)	Einen Counter inkrementieren

4 Task Management

4.1 Task Stack

Jede Task besitzt seinen eigenen Stack. Die Größe des Stacks muss für jede erstellte Task vom User statisch deklariert werden.

```
OS_STK  StackTask[TASK_STACK_SIZE];
```

Bei der Wahl der Stack Größe ist Vorsicht geboten. Es sollte nicht mehr als unbedingt nötig Speicher für den Stack allokiert werden (s.h. Abb. 45).

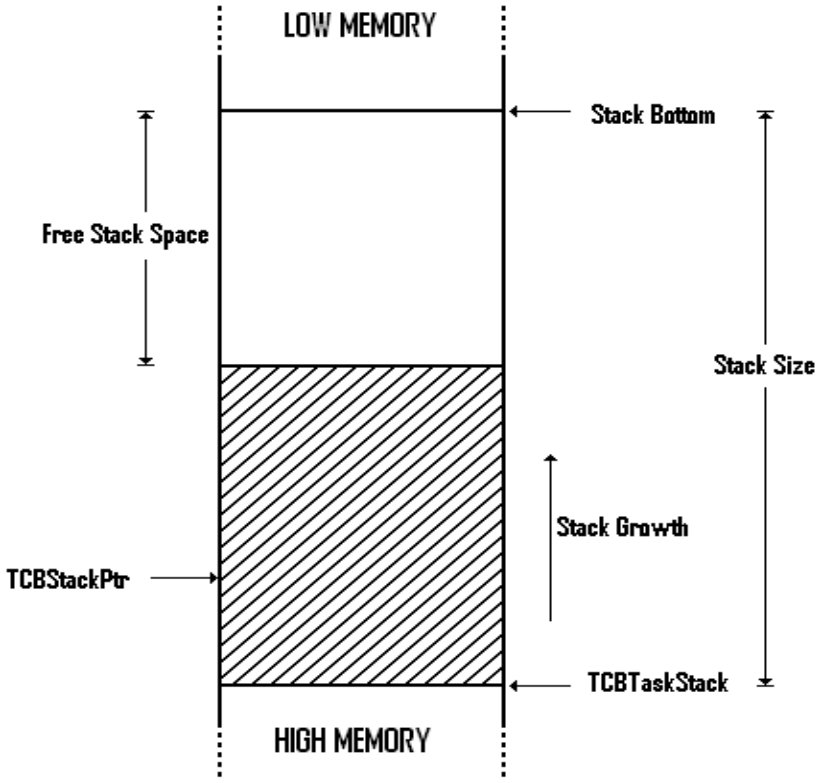


Abbildung 45: Betrachtung des Task-Stacks

4.2 Aktivieren einer Task, ***ActivateTask()***

Das setzen einer Task in den Ready Zustand wird mit der OS API Funktion ***ActivateTask()*** realisiert. Ob die entsprechende Task auch wirklich gleich nach dem Aufruf gestartet wird, hängt von der Priorität der Task ab.

Syntax

```
StatusType ActivateTask( TaskType TaskID );
```

Inputs

TaskID: Hier steht die jeweilige ID der Task die in der Konfigurations File deklariert wurde. Jede Task besitzt eine unterschiedliche ID.

Return

Status	Beschreibung
E_OS_ID	Ungültige Task ID
E_OK	Kein Fehler

Beschreibung der Funktion ***ActivateTask()***

Da kein Interrupt auftreten darf während die Funktion ***ActivateTask()*** ausgeführt wird, müssen zuallererst alle Interrupts gesperrt werden. Dies erfolgt durch die Funktion ***SaveSR()*** (s.h. 3.1 Kritische Regionen). Als nächstes wird abgefragt ob es sich bei der eingegebenen Task ID um eine Gültige Task ID handelt d.h. es handelt sich hierbei nicht um eine System Task (s.h. 3.11 Betriebssystem Tasks) und nicht um eine ID die es gar nicht gibt. Da die **MAX_NR_OF_TASKS** die gesamte Anzahl der vorhandenen Tasks beinhaltet, einschließlich der beiden System Tasks und die wiederum immer die ID 0 und 1 besitzen, muss die entsprechende Task ID dazwischen liegen.
Als nächstes wird abgefragt ob sich die entsprechende Task auch wirklich im Zustand Suspended befindet. Ist dies nicht der Fall, dann ist es nicht notwendig die Task zu aktivieren. Wenn es sich jedoch um eine Task handelt die sich zu dem Zeitpunkt im Zustand Suspended befindet, dann wird die Task erst einmal in den Zustand Ready gesetzt. Wenn die derzeit noch laufende Task eine kleinere Priorität besitzt als die zu aktivierende Task und auch noch **FULL_SCHEDULED** ist, d.h. dass die Task unterbrechbar ist, dann wird die aktuelle Task in den Zustand Ready gesetzt und die zu aktivierende Task in den Zustand Running.

Abbildung 46: Aktivieren einer Task (NSM-Diagramm)

4.2.1 ***ActivateTask()*** bei Scheduling mit verketteten Listen

```
//Abfrage ob zu aktivierende Task höhere Prio besitzt als akt. laufende
if((TaskList[TaskID].TCBPrio > pTCBCurRun->TCBPrio) && (OSScheduleMode ==
FULL_SCHEDULED))
{
    AddFirstToReadyQueue(pTCBCurRun);
    AddFirstToReadyQueue(&TaskList[TaskID]);

    Schedule();
    }
     //Abfrage ob zu aktivierende Task höhere Prio besitzt als HighestReady Task
    else if(TaskList[TaskID].TCBPrio > pTCBNextRun->TCBPrio)
    {
       AddFirstToReadyQueue(&TaskList[TaskID]);
    }
    //Ansonsten in ReadyQueue einfügen
    else
    {
       Reschedule(TaskID);
    }
```

Um die Task in den Zustand Running zu versetzen, muss die Task als erste in die ReadyQueue eingesetzt und dann ein Scheduling durchgeführt werden. Wenn die Task ID nicht höher ist als die der aktuell laufenden Task und/oder die aktuelle Task nicht

unterbrechbar ist, dann wird überprüft ob diese Task die nächst höhere Priorität besitzt die Ready ist. Wenn das der Fall ist, dann wird diese Task an erster Stelle in der ReadyQueue eingefügt, da diese Task dann als nächstes dran wäre wenn sich die aktuell laufende Task beendet.
Wenn auch das nicht der Fall ist, dann wird ***Reschedule()*** aufgerufen, was dazu führt das die Task ebenfalls in die ReadyQueue eingefügt wird, jedoch nicht als erstes Element sondern an die Stelle, an die eine Task mit der jeweiligen Priorität passt
(s.h. 3.5.1 Scheduling mit verketteten Listen).
Wenn alles ohne Fehler vonstatten ging, dann wird ein **E_OK** zurückgeliefert, ansonsten ein **E_OS_ID**. Zum Schluss werden die Interrupts wieder aktiviert.

4.2.2 ***ActivateTask()*** bei Bitmap Scheduling

```
//Abfrage ob zu aktivierende Task höhere Prio besitzt als akt. laufende
if((pTCBNew->TCBPrio > pTCBCurRun->TCBPrio) && (OSScheduleMode ==
FULL_SCHEDULED))
{
    pTCBPreRun = pTCBCurRun;
    InsertToPrioQueue(pTCBPreRun);

    pTCBCurRun = pTCBNew;
    pTCBCurRun->TCBStat = RUNNING;
    OSScheduleMode = pTCBCurRun->TCBScheduled;

    __ContextSwitch();
}
else
{
    InsertToPrioQueue(pTCBNew);
}
```

Um hier die Task in den Zustand Running zu setzen, muss die Task mittels der Funktion ***InsertToPrioQueue()*** (s.h. !?!?!?!?) in der **ReadyTable** bekannt gemacht und gestartet werden. Wenn das nicht zutrifft, dann wird die Task über die Funktion ***InsertToPrioQueue()*** bekannt gemacht und in den Zustand Ready gesetzt. Zum Schluss werden die Interrupts wiederhergestellt und das Error Handling erfolgt.

4.3 Beenden einer Task, ***TerminateTask()***

Das Beenden einer Task, bzw. das Setzen einer bestimmten Task in den Suspended Zustand wird mit der OS API Funktion ***TerminateTask()*** realisiert. Die nächste Task die danach in den Zustand Running versetzt wird, ist die Task die zu dieser Zeit Ready ist und die höchste Priorität besitzt.

Syntax

```
StatusType TerminateTask( void );
```

Inputs

keine

Return

Status	Beschreibung
E_OS_RESOURCE	Die Task beansprucht noch ein Resource und darf somit nicht beendet werden
E_OS_CALLEVEL	Die Funktion wurde aus einem Interrupt her aufgerufen
E_OK	Kein Fehler

Beschreibung der Funktion ***TerminateTask()***

Auch in dieser Funktion werden zuallererst die Interrupts gesperrt. Wichtig ist hier, dass diese Funktion nie aus einem Interrupt aufgerufen werden darf. Das wird mittels der Funktion ***__CPUReadMode()*** überprüft. Nun folgt die Verifizierung der Richtigkeit der eingegebenen Task ID und ob die zu terminierende Task zurzeit keine Ressource belegt (s.h. 3.9 Resource management).

```
...
if(pTCBCurRun->TCBPrio == pTCBCurRun->TCBStdPrio)
...
```

Wenn alles zutrifft, dann wird die Task in den Zustand Suspended gesetzt, die Event Maske wird gelöscht und durch die Funktion ***StartTask()*** wird die nächste höchst Priorisierte Task in den Zustand Running gesetzt.
Wenn der LPC2148 Prozessor verwendet wird, dann muss der Task Stack wieder in den Ausgangszustand versetzt werden.
Nun wird der Context Wechsel vollzogen, der je nach verwendeten Prozessor eine andere Funktion aufruft.
Zum Schluss werden die Interrupts wiederhergestellt und eine Rückgabewert wird zurückgegeben.

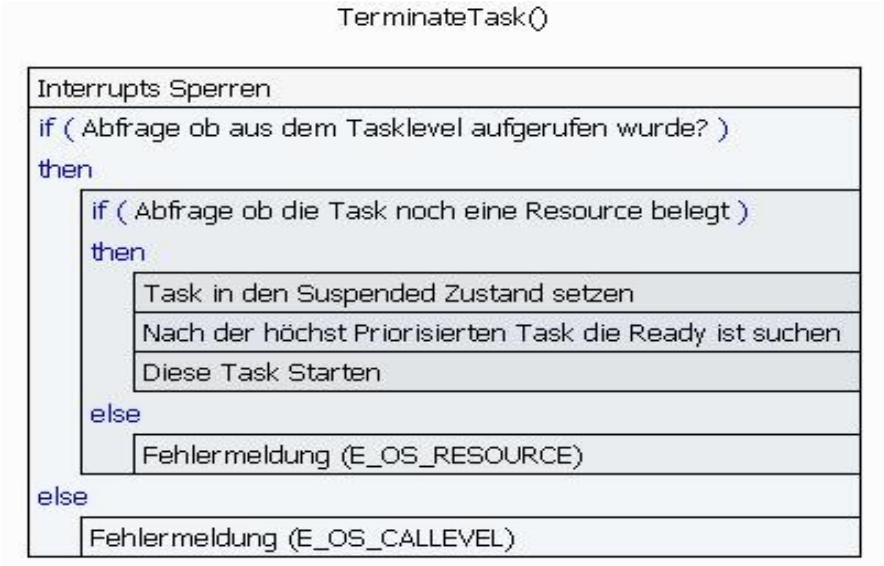


Abbildung 47: Beenden einer Task (NSM-Diagramm)

4.4 Beenden und gleichzeitig aktivieren einer Task, ***ChainTask()***

Das Beenden der aktuell laufenden Task und gleichzeitig das Aktivieren einer anderen Task, wird mit der OS API Funktion ***CainTask()*** realisiert. Diese Funktion kann als eine Zusammensetzung der Funktionen ***ActivateTask()*** und ***TerminateTask()*** gesehen werden. Aus diesem Grund ist diese Funktion sehr ähnlich wie die beiden zuvor genannten Funktionen.

Syntax

```
StatusType ChainTask( TaskType TaskID );
```

Inputs

TaskID: Hier steht die jeweilige ID der Task die in der Konfigurations File deklariert wurde. Jede Task besitzt eine unterschiedliche ID.

Return

Status	Beschreibung
E_OS_ID	Ungültige Task ID
E_OS_RESOURCE	Die Task beansprucht noch ein Resource und darf somit nicht beendet werden
E_OS_CALLEVEL	Die Funktion wurde aus einem Interrupt her aufgerufen
E_OK	Kein Fehler

Beschreibung der Funktion ***ChainTask()***

Hier gibt es nur einen kleinen Unterschied zwischen den beiden Scheduling Methoden und zwar beim aktivieren der neuen Task.

Der erste Teil dieser Funktion ist Identisch mit der API Funktion ***TerminateTask()*** und kann somit aus dieser Beschreibung entnommen werden.

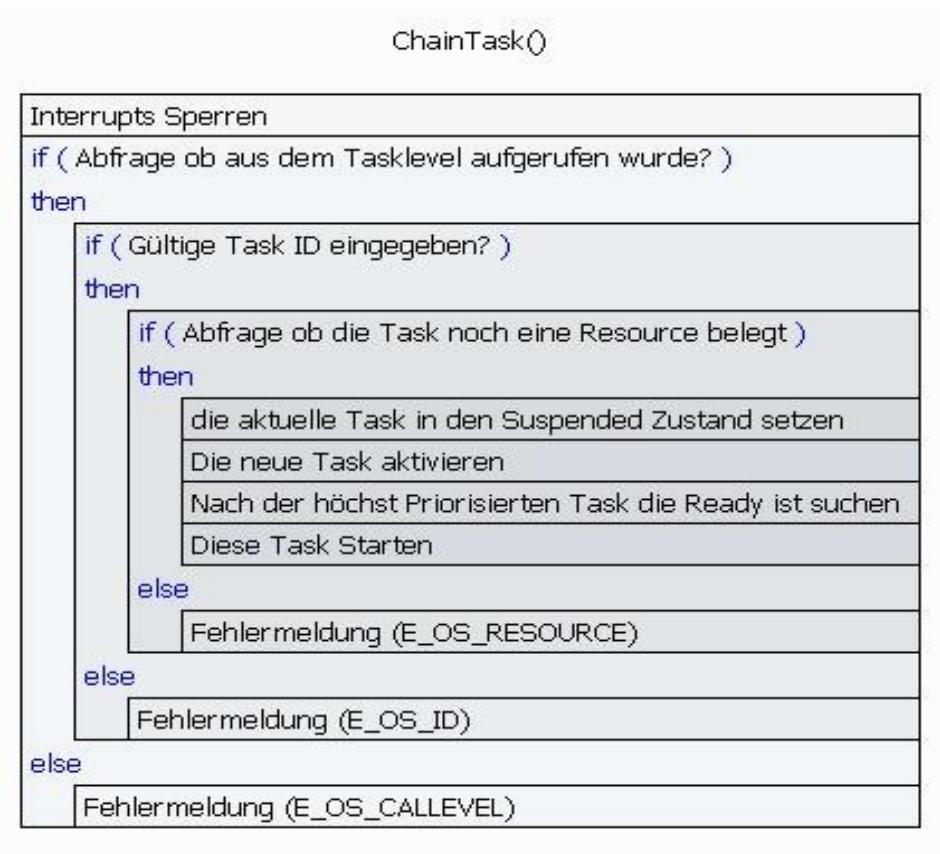


Abbildung 48: Beenden und aktivieren einer neuen Task (NSM-Diagramm)

4.4.1 ***ChainTask()*** bei Scheduling mit verketteten Listen

Hier wird das gleiche Verfahren verwendet wie es bei der Funktion ***ActivateTask()*** der Fall ist. Aus diesem Grund wird hier nicht näher darauf eingegangen. Im Grunde genommen wird überprüft ob die zu aktivierende Task sich noch im Suspended Zustand befindet. Wenn ja dann wird die Task in den Ready Zustand versetzt.

4.4.2 ***ChainTask()*** bei Bitmap Scheduling

```
pTCB =&TaskList[TaskID];

if(pTCB->TCBStat == SUSPENDED)
{
        InsertToPrioQueue(pTCB);
}
```

Adresse des Task Control Blocks wird geholt und es wird überprüft ob die zu aktivierende Task im Suspended Zustand ist. Wenn das zutrifft, wird diese Task in den Zustand Ready versetzt und in der Ready Table bekannt gemacht. Danach wird nach der höchst priorisierten Task die Ready ist gesucht und gestartet.

4.5 Unterbrechung einer nicht preemptiven Task, ***Schedule()***

Das Unterbrechen einer laufende Task d.h. in den Zustand Ready setzen und das Starten einer Task mit einer höheren Priorität erfolgt durch die OS API Funktion ***Schedule()***. Der Aufruf dieser Funktion aus einer unterbrechbaren Task macht keinen Sinn, da diese Task auch die Task ist die die derzeit höchste Priorität besitzt. Wenn die Funktion aufgerufen wird und es keine Task gibt die eine höhere Priorität besitzt als die aktuell laufende, dann wird die Task aus der die Funktion aufgerufen wurde, weiter geführt.

Syntax

```
StatusType Schedule( void );
```

Inputs

keine

Return

Status	Beschreibung
E_OS_CALLEVEL	Die Funktion wurde aus einem Interrupt her aufgerufen
E_OK	Kein Fehler

Beschreibung der Funktion ***Schedule()***

Wichtig, wie in jeder API Funktion, dass zu aller erst die Interrupts gesperrt werden. Nun wird überprüft ob die Funktion nicht aus einem Interrupt aufgerufen wird. Es wird nach der Task mit der höchsten Priorität gesucht die Ready ist. Danach wird überprüft ob diese Task eine höhere Priorität besitzt als die aktuell laufende. Wenn das nicht der Fall ist, dann wird die aktuelle Task wieder fortgesetzt. Andernfalls wird die neue Task gestartet. Zum Schluss werden die Interrupts wieder hergestellt.
Jedoch unterscheidet sich, je nach Art des Scheduling, die Funktion in ihrem Aufbau. Aus diesem Grund werden im folgenden beide Funktionen explizit beschrieben.

4.5.1 ***Schedule()*** bei Scheduling mit verketteten Listen

```
...
if((pTCBNextRun != NULL))                                          (1)
{
    //Prüfen ob Task mit höherer Prio vorliegt
    if(pTCBCurRun->TCBPrio < pTCBNextRun->TCBPrio)                 (2)
    {
        //Falls die Task selbst Schedule aufruft
        if ((pTCBCurRun->TCBStat == RUNNING) &&
          (OSScheduleMode == NONE_SCHEDULED))                      (3)
        {
            pTCBCurRun->TCBStat = READY;                           (4)
            Reschedule(pTCBCurRun->TCBId);                         (5)
        }
        //Auf Contextswitch vorbereiten, aktuell laufende Task sichern
        pTCBPreRun =pTCBCurRun;

        //Höchstpriore Task in den Zustand Run versetzen
        StartTask();                                               (6)
       __ContextSwitch();                                          (7)
...
```

Erklärung zu Schedule:

(1): Wenn pTCBNextRun gleich Null ist, dann gibt es keine Task die Ready ist und somit kann Schedule beendet werden.

(2): Falls es doch eine oder mehrere Tasks gibt die Ready sind, dann muss überprüft werden ob die zurzeit höchste Priorität der Task die Ready ist auch höher ist als die aktuell laufende.

(3): Da beim Scheduling mit verketteten Listen die Funktion Schedule() auch aus anderen Funktionen aufgerufen wird, besteht hier die Möglichkeit das der Zustand der aktuellen Task -> **pTCBCurRun** (Zeiger auf den TCB der aktuell laufenden Task) nicht Running ist. Außerdem muss es sich um eine nicht preemptive Task handeln.

(4): Hier wird die aktuelle Task in den Zustand Ready gesetzt.

(5): Aktuelle Task wieder in der ReadyQueue miteinander verketten.

(6): In der ***StartTask()*** wird nun die höchst Priorisierte Task in den Zustand Running versetzt und die Queue wird neu gesetzt.

(7): Der Task Wechsel wird durchgeführt.

4.5.2 ***Schedule()*** bei Bitmap Scheduling

```
...
//Höchstpriore Task finden
y = OSBitMapTbl[OSReadyGrp];
prioHighReady = (INT8U)((y << 3) + OSBitMapTbl[ReadyTable[y]]);                    (1)

pTCB = TCBPrioTbl[prioHighReady];                                                  (2)
//Prüfen ob Task mit höherer Prio vorliegt
if(pTCBCurRun->TCBPrio < pTCB->TCBPrio)                                            (3)
{
      pTCBPreRun = pTCBCurRun;
      InsertToPrioQueue(pTCBPreRun);                                               (4)

      StartTask();                                                                 (5)
      __ContextSwitch();                                                           (6)
}
...
```

Erklärung zu Schedule:

(1): Über die BitMap Tabelle wird die Task mit der höchsten Priorität und die Ready ist heraus gesucht.

(2): Die Adresse des Task Control Blocks wird geholt.

(3): Überprüfung ob diese Task eine höhere Priorität besitzt als die aktuell laufende Task.

(4): Die aktuell laufende Task wird in den Zustand Ready gesetzt, in der Ready Table bekannt gemacht und notfalls mit Tasks der gleichen Priorität verkettet.

(5): Die neue Task wird in den Zustand Running gesetzt und aus der Ready Table gelöscht.

(6): Der Task Wechsel wird durchgeführt.

4.6 Ermitteln der Task ID, ***GetTaskID()***

Um die ID der aktuell laufenden Task zu ermitteln wird die OS API Funktion GetTaskID() aufgerufen.

Syntax

```
StatusType GetTaskID( TaskRefType RefTask );
```

Outputs

RefTask ist ein Zeiger auf eine Variable vom Type TaskType. In dieser Variablen wird die ID der aktuell laufenden Task gespeichert.

Return

Status	Beschreibung
E_OK	Kein Fehler

Beschreibung der Funktion ***GetTaskID()***

Die Funktion überprüft ob sich die aktuelle Task auch im Zustand Running befindet. Ist das nicht der Fall, dann wird der Fehler **INVALID_TASK** an den Zeiger übergeben. Wenn sich die Task im Running Zustand befindet, dann wird die ID der Task zurückgegeben. Die Funktion kann auch aus einem Interrupt aufgerufen werden.

4.7 Task zustand ermitteln, ***GetTaskState()***

Um den Zustand einer Task zu erhalten, wird die OS API Funktion ***GetTaskState()*** verwendet.

Syntax

```
StatusType GetTaskState( TaskType TaskID, TaskStateRefType RefTask );
```

Inputs

TaskID: Hier steht die jeweilige ID der Task die in der Konfigurations File deklariert wurde. Jede Task besitzt eine unterschiedliche ID.

Outputs

RefTask ist ein Zeiger auf eine Variable vom Type TaskType. In dieser Variablen wird die ID der aktuell laufenden Task gespeichert.

Return

Status	Beschreibung
E_OS_ID	Ungültige Task ID
E_OK	Kein Fehler

Beschreibung der Funktion ***GetTaskState()***

Die Funktion überprüft erst ob die eingegebene Task ID eine gültige ID ist. Wenn nicht dann wird eine Fehlermeldung zurückgegeben. Wenn es sich um eine gültige ID handelt, dann wird der Zustand der jeweiligen Task an den Zeiger übergeben. Die Funktion kann auch aus einem Interrupt aufgerufen werden.

5 Event Management

5.1 Setzen eines Events, *SetEvent()*

Das Setzen eines Events für eine bestimmte Task wird durch die OS API Funktion ***SetEvent()*** realisiert. Wenn diese Task auf dieses gesetzte Event warten sollte, dann wird die Task in den Zustand Ready wechseln. Falls die Priorität der Task höher ist als die der aktuell laufenden Task, dann wird die Task gleich in den Zustand Running gesetzt.

Syntax

```
StatusType SetEvent( TaskType TaskID, EventMaskType Mask );
```

Inputs

TaskID: Hier steht die jeweilige ID der Task die in der Konfigurations File deklariert wurde. Jede Task besitzt eine unterschiedliche ID.

Mask: Das ist eine 8 Bit Maske die mit der Event Maske logisch UND verknüpft wird.

Return

Status	Beschreibung
E_OS_ID	Ungültige Task ID
E_OS_ACCESS	Es handelt sich nicht eine extended Task (d.h. Die Task kann nicht in den Waiting Zustand gesetzt werden)
E_OS_STATE	Task befindet dich im Zustand Suspended
E_OK	Kein Fehler

Beschreibung der Funktion ***SetEvent()***

Wie in jeder API Funktion in der eine Kontext Wechsel stattfinden kann, müssen erst die Interrupts gesperrt werden. Als nächstes wird überprüft ob es sich bei der Task ID um eine gültige ID handelt, ob es sich um eine Extended Task handelt (d.h. eine Task die in den Zustand Waiting gesetzt werden kann) und ob sich die Task nicht in diesem Augenblick im Zustand Suspended befindet. Falls eines dieser Überprüfungen fehlerhaft ist, dann wird die entsprechende Fehlermeldung zurückgegeben. Nun findet die Überprüfung der übergebenen Event Maske mit der Maske aus dem TCB der entsprechenden Task statt und ob diese Task sich auch im Zustand Waiting befindet. Wenn sich die Task nicht in dem Zustand Waiting befindet, dann ist es auch nicht notwendig diese Task in den Zustand Ready zu setzen, da die Task auf kein Event wartet. Ist dies alles der Fall, dann wird die Task in den Zustand Ready gesetzt und die Event Maske gelöscht. Wenn die Task eine höhere Priorität besitzt als die aktuell laufende Task und diese unterbrechbar ist, dann findet ein Task Wechsel statt der je nach Scheduling Methode etwas anders aussieht (s.h. 3.5 Schedulingverfahren). Zum Schluss werden die Interrupts wieder freigegeben und je nach dem wie der Verlauf dieser Funktion war, die entsprechende Meldung zurückgeliefert.

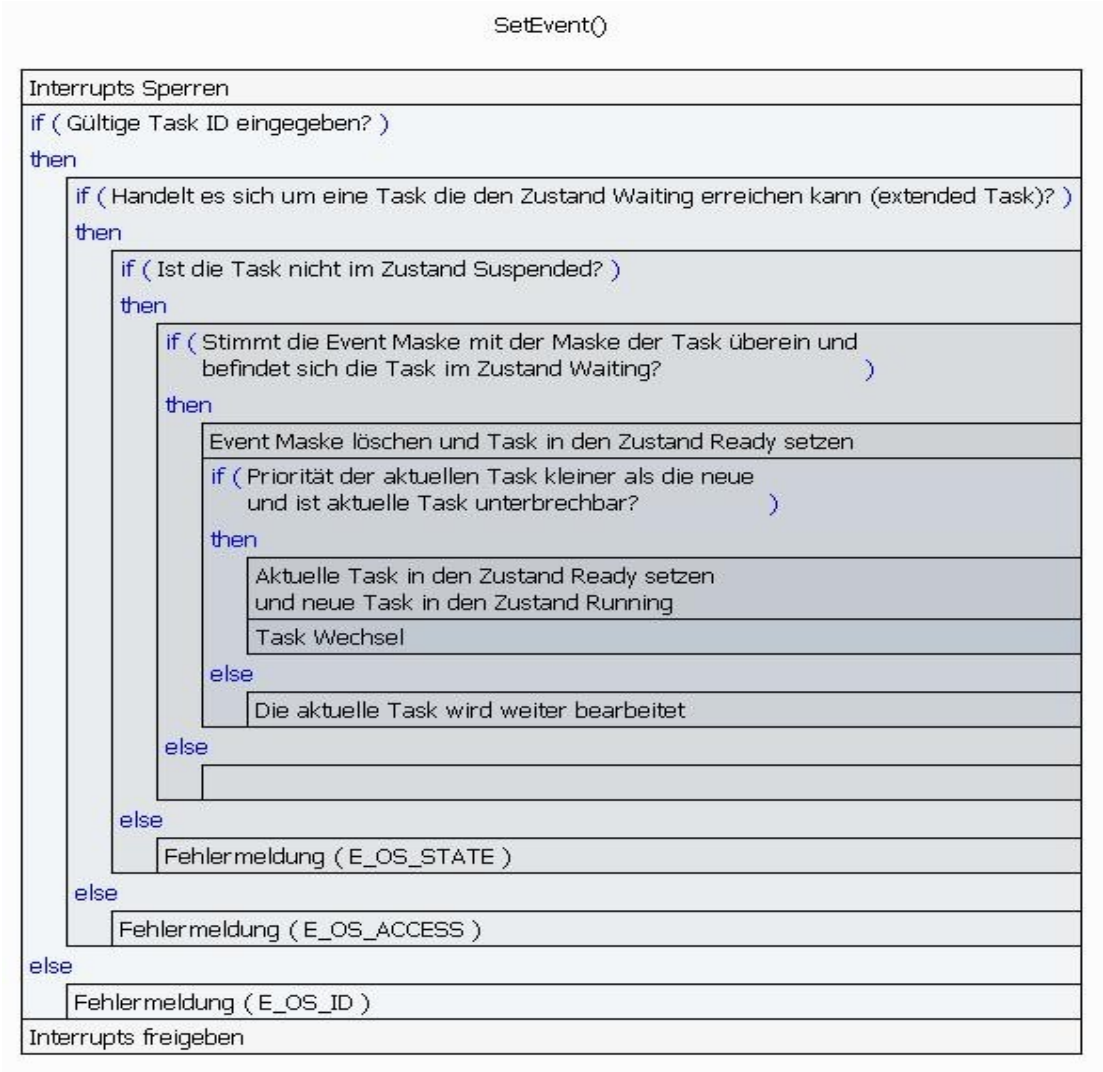


Abbildung 49: Setzen eines Events (NSM-Diagramm)

5.2 Löschen eines Events, ***ClearEvent()***

Das Löschen eines Events wird durch die OS API Funktion ***ClearEvent()*** realisiert. Nur die aktuell laufende Task kann die Events im entsprechenden TCB löschen. Zu beachten ist, das nur eine Task die im extended Mode läuft aufgerufen werden darf.

Syntax

```
StatusType ClearEvent( EventMaskType Mask );
```

Inputs

Mask: Das ist eine 8 Bit Maske die mit der Event Maske logisch UND verknüpft wird.

Return

Status	Beschreibung
E_OS_ACCESS	Es handelt sich nicht eine extended Task (d.h. Die Task kann nicht in den Waiting Zustand gesetzt werden)
E_OS_STATE	Task befindet dich im Zustand Suspended
E_OK	Kein Fehler

Beschreibung der Funktion ***ClearEvent()***

Diese Funktion darf nicht aus einem Interrupt aus aufgerufen werden, aus diesem Grund wird über die Funktion ***__CPUReadMode()*** überprüft ob aus dem Tasklevel aus die Funktion aufgerufen wurde. Danach wird sicher gestellt ob es sich bei der aufrufenden Task um eine extended Task handelt. Wenn alles in Ordnung ist, wird die entsprechende Maske aus dem jeweiligen TCB zurückgesetzt.

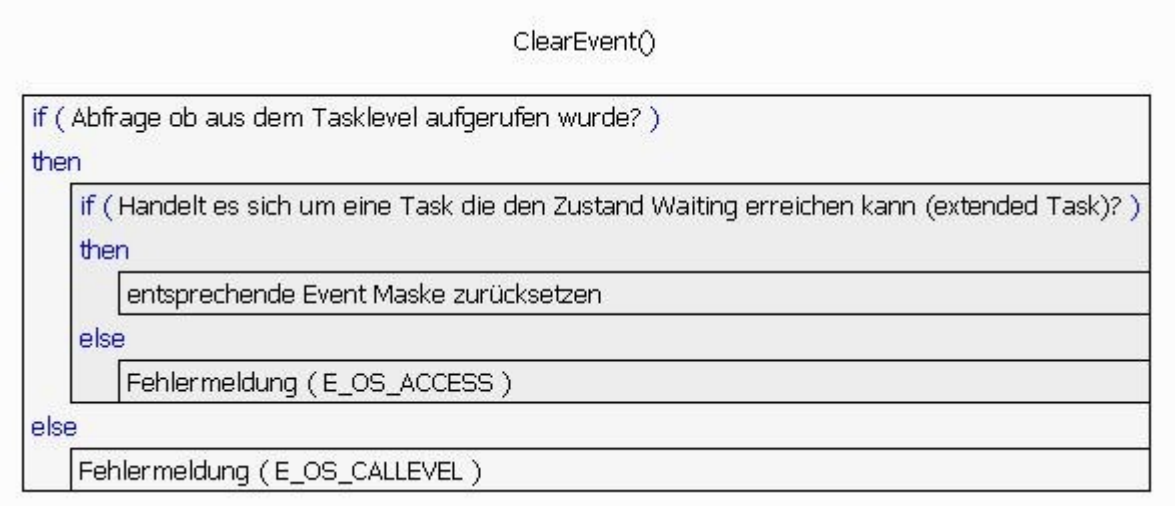


Abbildung 50: Löschen eines Events (NSM-Diagramm)

5.3 Einlesen von vorhandenen Events, ***GetEvent()***

Das Herauslesen von bestimmten Event Masken wird durch die OS API Funktion ***GetEvent()*** realisiert. Die Adresse der Event Maske wird dem Zeiger übergeben vom dem aus auf die Maske zugegriffen werden kann.

Syntax

```
StatusType GetEvent( TaskType TaskID, EventMaskRefType refEvent );
```

Inputs

TaskID: Hier steht die jeweilige ID der Task die in der Konfigurations File deklariert wurde. Jede Task besitzt eine unterschiedliche ID.

refEvent: Hier handelt es sich um eine Referenz auf eine Variable vom Type EventMaskType. In der wird die Adresse der Event Maske geschrieben.

Return

Status	Beschreibung
E_OS_ID	Ungültige Task ID
E_OS_ACCESS	Es handelt sich nicht eine extended Task (d.h. Die Task kann nicht in den Waiting Zustand gesetzt werden)
E_OS_STATE	Task befindet dich im Zustand Suspended
E_OK	Kein Fehler

Beschreibung der Funktion ***GetEvent()***

Zuerst wird überprüft ob es sich bei der Task ID um eine gültige ID handelt, ist diese Task eine extended Task und ist diese Task nicht um Zustand Suspended. Wenn das alles nicht zutrifft, dann wird die Adresse der Event Maske der jeweiligen Task dem Zeiger **refEvent** übergeben. Wenn bei den Abfragen etwas nicht übereinstimmt, dann wird die entsprechende Fehlermeldung zurückgegeben.

Abbildung 51: Eventmaske auslesen (NSM-Diagramm)

5.4 Auf ein Event warten, ***WaitEvent()***

Das Warten auf eine bestimmtes Event wird durch die OS API Funktion ***WaitEvent()*** realisiert. Nur extended Tasks dürfen diese Funktion aufrufen.

Syntax

```
StatusType WaitEvent( EventMaskType Mask );
```

Inputs

Mask: Das ist eine 8 Bit Maske die mit der Event Maske logisch UND verknüpft wird.

Return

Status	Beschreibung
E_OS_ID	Ungültige Task ID
E_OS_CALLEVEL	Die Funktion wurde aus einem Interrupt her aufgerufen
E_OK	Kein Fehler

Beschreibung der Funktion ***WaitEvent()***

Die Interrupts werden gesperrt. Dann wird überprüft ob es sich um eine extended Task handelt, ob aus dem Tasklevel aus die Funktion aufgerufen wurde und ob die eingegebene Maske mit der Event Maske der Task übereinstimmt. Wenn die Event Maske gleich ist, dann wird die Task weiter bearbeitet. Wenn sie sich jedoch unterscheiden, dann wird die Task in den Zustand Waiting versetzt und eine Task Wechsel findet statt.

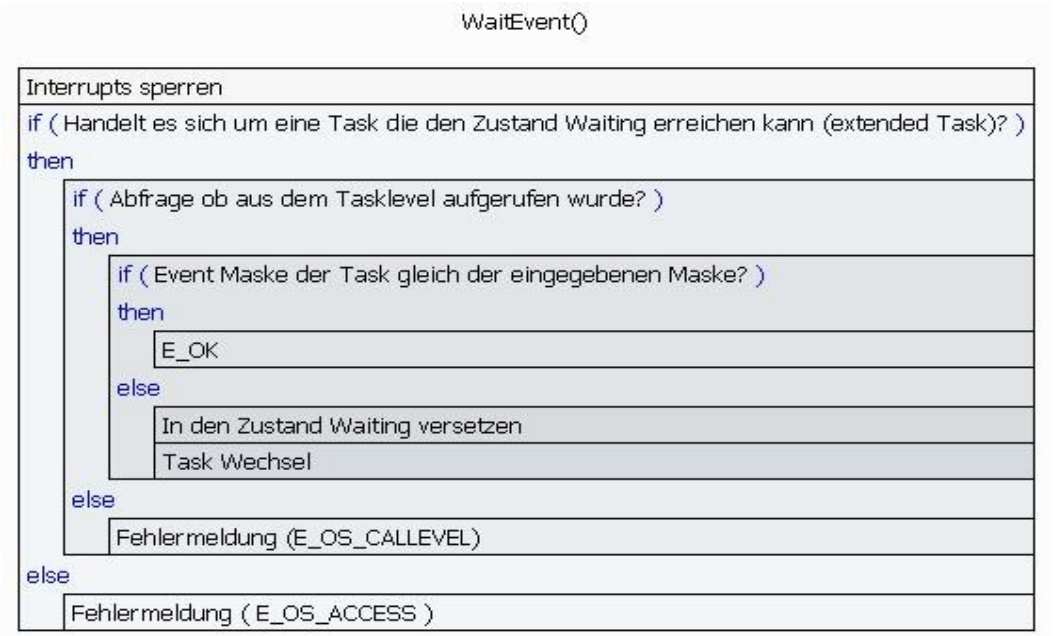


Abbildung 52: Warten auf ein Event, (NSM-Diagramm)

6 Time Management

In Kapitel 3.10 wurden bereits die Grundlagen von Timer, Counter und Alarmen erläutert. Die OSEK/VDX Spezifikation legt standardisierte API-Funktionen fest, mit deren Hilfe die Betriebssystemmittel verwendet werden können.
Die Versionen von OSEK/VDX Spezifikation welche kleiner 2.0 sind, beschrieben noch die Counter als Betriebssystemmittel. Ab der Version 2.0 sind diese kein OSEK Spezifisches Betriebssystemmittel. In HSE Free OSEK V.01 sind allerdings noch die alten API-Funktionen vorhanden doch sie werden nicht mit der „osek.h“ eingebunden. Die Verwendung bleibt jedem Anwender selber überlassen. Um die Counter-Funktionen zu nutzen ist es daher zusätzlich notwendig die „OS_Counter.h“ mit einzufügen („../Src/OSEK Code/Counter.h“).

6.1 Counter(Software-Timer)

6.1.1 Counter Initialisieren, *InitCounter()*

Syntax

StatusType ***InitCounter***(CtrType CounterId, TickType Ticks)

Inputs

CounterId: Name (Identifier) des Counters

Ticks: Neuer Zählstand des Counters

Return

Status	Beschreibung
E_OK	Funktionsabarbeitung erfolgreich
E_OS_ID	ID unbekannt (nicht existent)
E_OS_VALUE	Übergebener Wert Ticks ist größer als MAXALLOWEDVALUE
E_OS_CALLEVEL	Aufruf nur vom Task-Level zugelassen

Beschreibung der Funktion ***InitCounter()***

Um den Counter auf einen bestimmten Wert zu initialisieren, muss die Funktion ***InitCounter*** ausgeführt werden. Dabei wird in **CounterId** der Name des Counters übergeben und in Ticks der neue Wert für den Zählerstand. Ticks ist eine Variable des Datentyps **TickType**, es kann auch eine Konstante übergeben werden. Bei der Anwendung der Funktion ist darauf zu achten das der momentane Zählstand des ausgewählten Counters verloren geht.

6.1.2 Counterwert auslesen, *GetCounterValue()*

Syntax

StatusType ***GetCounterValue***(CtrType CounterId , TickRefType Ticks)

Inputs

CounterId: Name (Identifier) des Counters

Ticks: Referenz auf einer Variable vom Typ TickType in welcher der aktuelle Zählerstand zurückgegeben wird

Return

Status	Beschreibung
E_OK	Funktionsabarbeitung erfolgreich
E_OS_ID	ID unbekannt (nicht existent)
E_OS_CALLEVEL	Aufruf nur vom Task-Level zugelassen

Beschreibung der Funktion ***GetCounterValue()***

Der aktuelle Zählerstand eines Counters kann durch den Aufruf von ***GetCounterValue***() ermittelt werden. In **CounterId** wird wieder der Name des Counters übergeben und in Ticks der Zeiger auf eine Variable vom Typ **TickType**. Mit dem Aufruf wird der aktuelle Zählerstand in dieser Variable gespeichert.
Counterwert auslesen, ***GetCounterValue()***

Beispiel zur Verwendung der Funktion:

```
TASK(TaskX)
{
        TickType Tick;
        ...
        GetCounterValue(CTR1_ID , &Tick);
        ...
}
```

6.1.3 Auslesen der Counter-Parameter, ***GetCounterInfo()***

Syntax

StatusType ***GetCounterInfo***(CtrType CounterId, CtrInfoRefType pInfo)

Inputs

CounterId: Name (Identifier) des Counters

pInfo: Für pInfo ist der Zeiger auf eine Variable vom Type CtrInfoType zu übergeben. Bei CtrInfoType handelt es sich um eine Struktur mit den Attributen MAXALLOWEDVALUE, TICKSPERBASE und MINCYCLE.

Return

Status	Beschreibung
E_OK	Funktionsabarbeitung erfolgreich
E_OS_ID	ID unbekannt (nicht existent)
E_OS_CALLEVEL	Aufruf nur vom Task-Level zugelassen

Beschreibung der Funktion ***GetCounterInfo()***

Auch die Counterparameter MAXALLOWEDVALUE, TICKSPERBASE und MINCYCLE können zur Laufzeit ermittelt werden. Dazu ist die Funktion ***GetCounterInfo***() zu verwenden. Für Info ist der Zeiger auf eine Variable vom Type **CtrInfoType** zu übergeben. Bei **CtrInfoType** handelt es sich um eine Struktur mit den Attributen MAXALLOWEDVALUE, TICKSPERBASE und MINCYCLE. Diese Attribute werden mit dem Aufruf dieser Funktion mit den in der Config-Datei eingestellten Werten belegt.

Beispiel zur Verwendung der Funktion:

```
TASK(TaskX)
{
        OS_COUNTER_INFO  counterinfo;
        ...
        GetCounterInfo(0, &counterinfo);
        ...
}
```

6.1.4 Inkrementieren von Countern, *CounterTrigger ()*

Hinweis: Funktion weicht teilweise von der OSEK Spez. ab.

Syntax

StatusType ***CounterTrigger***(CtrType CounterId)

Inputs

CounterId: Name (Identifier) des Counters

Return

Status	Beschreibung
E_OK	Funktionsabarbeitung erfolgreich
E_OS_ID	ID unbekannt (nicht existent)

Beschreibung der Funktion ***CounterTrigger()***

Durch die Funktion ***CounterTrigger()*** wird der durch CounterId spezifizierte Counter getriggert. Falls der Counter dadurch inkrementiert wird und vor dem Aufruf schon den Wert **MAXALLOWEDVALUE** besessen hat, wird er auf den Wert Null gesetzt.
Der Counter wird erst dann inkrementiert, wenn er durch den Aufruf dieser Funktion so oft getriggert wurde, wie durch das Attribut TICKSPERBASE in der Config-Datei eingestellt wurde. Es kann lediglich der Fehlerwert E_ID zurückgeliefert werden, falls der Counter CounterId nicht deklariert wurde. Diese Funktion sollte nur von der KernelTask verwendet werden.

6.1.5 Alarm einem Counter zuweisen, ***AllocatAlarmToCounter()***

Hinweis**:** ***AllocatAlarmToCounter()*** ist keine API Funktion!

Syntax

StatusType ***AllocatAlarmToCounter***(CtrType CtrId, AlarmType AlarmId)

Inputs

CounterId: Name (Identifier) des Counters

AlarmId: Name (Identifier) des Alarms

Return

Status	Beschreibung
E_OK	Funktionsabarbeitung erfolgreich
E_OS_ID	ID unbekannt (nicht existent)

Beschreibung der Funktion ***AllocatAlarmToCounter()***

Ein Alarm basiert immer auf einem Counter. Er stellt lediglich Mechanismen zur Verfügung, um das System abhängig von einem absoluten oder relativen Zählerstand zu steuern. Doch ein Alarm muss einem Counter zugewiesen werden, den der Counter wird inkrementiert und der Zählstand mit dem Alarmzeitpunkt vergleichen. Die Funktion wird lediglich bei der Initialisierung von Alarm verwendet (***AlarmInit()***).

6.2 Alarme

6.2.1 Alarm-Parameter auslesen, ***GetAlarmBase()***

Hinweis: Funktion weicht teilweise von der OSEK Spez. ab

Syntax

StatusType ***GetAlarmBase***(AlarmType AlarmID, CtrInfoType pInfo)

Inputs

AlarmID: Name (Identifier) des Counters

pInfo: Für pInfo ist der Zeiger auf eine Variable vom Type CtrInfoType zu übergeben. Bei CtrInfoType handelt es sich um eine Struktur mit den Attributen MAXALLOWEDVALUE, TICKSPERBASE und MINCYCLE.

Return

Status	Beschreibung
E_OK	Funktionsabarbeitung erfolgreich
E_OS_ID	ID unbekannt (nicht existent)

Beschreibung der Funktion ***GetAlarmBase()***

Die Funktion ***GetAlarmBase()*** liefert die zugehörigen Counterparameter MAXALLOWEDVALUE, TICKSPERBASE und MINCYCLE zur Laufzeit. Somit verhält sich die ***GetAlarmBase()*** identisch zu ***GetCounterInfo***(). Der Unterschied ist lediglich das man zum zugehörigen Alarm die Parameter ausliest.

Beispiel zur Verwendung der Funktion:

```
TASK(TaskX)
{
        OS_COUNTER_INFO  counterinfo;
        ...
        GetAlarmBase(ALARM1_ID, &counterinfo);
        ...
}
```

6.2.2 Zeit bis zum nächsten Alarm bestimmen, ***GetAlarm()***

Syntax

```
StatusType GetAlarm ( AlarmType AlarmID, TickRefType pTick);
```

Inputs

AlarmID: Name (Identifier) des Counters

pTick: Ein Pointer, der auf eine Variable vom Typ Tick zeigt, liefert den Rückgabewert in Ticks zurück.

Return

Status	Beschreibung
E_OK	Funktionsabarbeitung erfolgreich
E_OS_ID	ID unbekannt (nicht existent)

Beschreibung der Funktion ***GetAlarm ()***

Die Funktion ***GetAlarm()*** liefert die Zeit in Ticks zurück bis zum Auftreten des Alarms. Der zurückgegeben Tick-Wert ist in der Abhängekit des zugewiesenen Counters zu interpretieren (TICKSPERBASE). Durch die Rechnung (pTick x TICKSPERBASE) kann der „wahre“ Wert bestimmt werden.

Beispiel zur Verwendung der Funktion:

```
TASK(TaskX)
{
        TickType Tick;
        ...
        GetAlarm (ALARM2_ID, &Tick);
        ...
}
```

6.2.3 Relative Zeit bis zum nächsten Alarm festlegen, ***SetRelAlarm()***

Syntax

StatusType ***SetRelAlarm*** (AlarmType AlarmID, TickType increment, TickType cycle)

Inputs

AlarmID: Name (Identifier) des Counters

TickType: Anzahl Ticks bis zum ersten Auftretten des Timers

cycle: Zykluszeit in Ticks

Return

Status	Beschreibung
E_OK	Funktionsabarbeitung erfolgreich
E_OS_ID	ID unbekannt (nicht existent)

Beschreibung der Funktion ***SetRelAlarm()***

Die Funktion ***SetRelAlarm()*** wird benutzt um einen Alarm zu aktivieren, der zeitlich relativ zum Aufruf der Funktion ausgeführt wird. Nach dem Auftreten des Alarms kann eine Task aktiviert oder ein Event ausgeführt werden. Der Alarm wird nach **increment** ausgelöst. Mit **cycle** wird die Periode des Alarms bestimmt. Wenn **cycle = 0** wird der Alarm einmalig ausgeführt.

Hierzu eine Beispiel: ***SetRelAlarm***(Alarm1_ID, 6, 8) (s.h. Abb. 53)

Dies würde bedeuten das es noch 6 Ticks sind bis der Alarm zum ersten Mal auftritt und danach zyklisch alle 8 Ticks. Der Zeitpunkt wann genau er das erste Mal auftritt, ist abhängig vom Zählerstand des Counters (t_{start}).

Beispiel: SetRelAlarm(ALARM1_ID, 6,8)

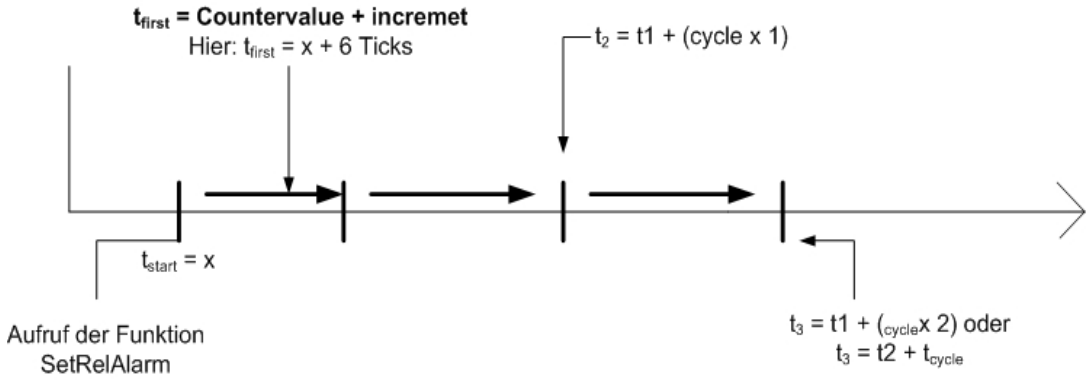


*Abbildung 53: Verwendung von **SetRelAlarm()***

6.2.4 Relative Zeit bis zum nächsten Alarm festlegen, ***SetAbsAlarm()***

Syntax

StatusType ***SetAbsAlarm*** (AlarmType AlarmID, TickType StartTime, TickType cycle)

Inputs

AlarmID: Name (Identifier) des Counters

StartTime: Genau festgelegter (absoluter) Zeitpunkt wann der Alarm das erste mal Auftrit.

cycle: Zykluszeit in Ticks

Return

Status	Beschreibung
E_OK	Funktionsabarbeitung erfolgreich
E_OS_ID	ID unbekannt (nicht existent)

Beschreibung der Funktion ***SetAbsAlarm()***

Die Funktion ***SetAbsAlarm()*** wird benutzt um einen Alarm zu aktivieren, der zu einem genau festgelegten Zeitpunkt eintreten soll. Nach dem Auftreten des Alarms kann eine Task aktiviert oder ein Event ausgeführt werden. Der Alarm wird zum Zeitpunkt **StartTime** ausgelöst. Mit **cycle** wird die Periode des Alarms bestimmt. Wenn **cycle = 0** wird der Alarm einmalig ausgeführt.

Hierzu eine Beispiel: ***SetAbsAlarm***(Alarm1_ID, 1000, 8)

Dies würde bedeuten das der Alarm genau zu dem Zeitpunkt auftritt wenn der zugewiesene Counter den Wert ***CounterValue*** = 1000 erreicht. Danach würde der Alarm zyklisch alle 8 Ticks erneut auftreten. Zu beachten ist, dass beim Aufruf der Funktion der Counter den Wert 1000 bereits überschritten haben kann. Somit würde der Alarm erst nach der Zeit t1+t2 auftreten (s.h. Abb. 54).

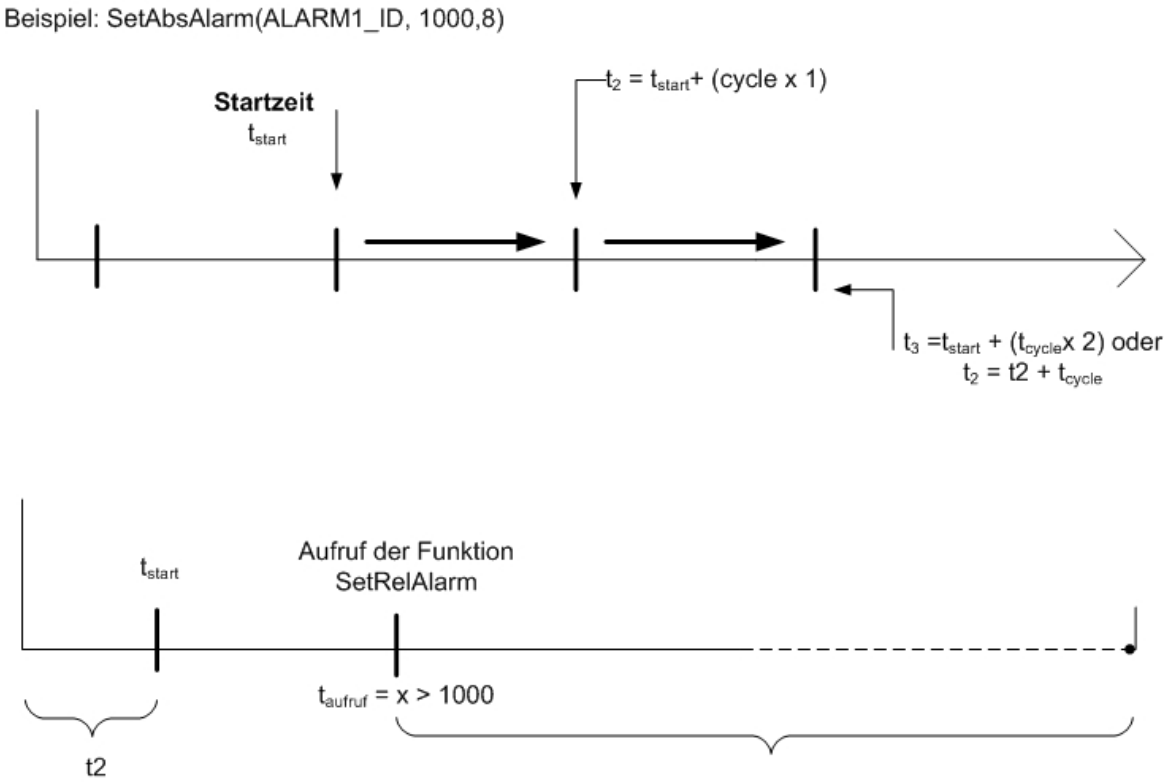


*Abbildung 54: Verwendung von **SetAbsAlarm()***

6.2.5 Alarm-Parameter auslesen, ***CancelAlarm()***

Syntax

StatusType ***CancelAlarm*** (AlarmType AlarmID)

Inputs

AlarmID: Name (Identifier) des Counters

Return

Status	Beschreibung
E_OK	Funktionsabarbeitung erfolgreich
E_OS_NOFUNC	Alarm ist bereits deaktivert

Beschreibung der Funktion ***CancelAlarm***

Die Funktion ***CancelAlarm()*** soll einen aktiven Alarm deaktivieren.

Beispiel zur Verwendung der Funktion:

```
TASK(TaskX)
{
        ...
        CancelAlarm(ALARM2_ID);
        ...
}
```

6.2.6 Berechnung der Alarm Zeit, ***Add_Tick()***

Hinweis**:** ***Add_Tick()*** ist keine API Funktion und nicht OSEK spezifisch

Syntax

TickType ***Add_Tick***(TickType almval, TickType incr, TickType maxval)

Inputs

almval: Aktueller Alarm (Counter) Wert in Ticks

incr: Wert der zum aktuellen Zeitpunkt hinzu addiert wird

maxval: Maximal zugelassener Wert des Alarms (MAXALLOWEDVALUE)

Return

Rückgabewert	Beschreibung
TickType	Neu berechneter Alarmswert

Beschreibung der Funktion ***Add_Tick***

Wie Abbildung X zeigt kann es bei setzen eines neuen Alarmwerts zu einem Overflow kommen wenn der Counter nahe an seinem Maximalwert liegt. Die Funktion ***Add_Tick()*** soll sicherstellen, dass beim Aufruf der Funktionen ***SetAbsAlarm()*** & ***SetRelAlarm()*** auch der exakte Wert eingestellt wird.

```
TickType Add_Tick(TickType almval, TickType incr, TickType maxval)
{
  //wenn das Ergebniss kleiner ist als die differenz von 255 - currentval
  if (incr <= (maxval - almval))
  {
    return(almval + incr);       //neuen Wert setzen
  }
  else
  {
    return( (almval + incr) - (maxval + 1) );
  }
}
```

7 Resource Management

7.1 Resource (binären Semaphor) belegen, *GetResource()*

Syntax

```
StatusType GetResource(ResourceType ResID);
```

Inputs

ResID: Name(Identifier) der Ressource

Return

Status	Beschreibung
E_OK	Funktionsabarbeitung erfolgreich
E_OS_ID	ID unbekannt (nicht existent)
E_OS_ACCESS	Ressource bereits belegt

Beschreibung der Funktion *GetResource()*

Um eine Ressource zu erwerben wird die Funktion *GetResource()* verwendet. Solange die Task die Resource belegt erhält sie die Priorität der Ressource welche größer sein muss als die eigene. Resourcen können globale Variablen, Systemfunkionen oder auch Hardwarebausteine sein. Der von der OSEK Spezifikation vorgegebene Funktionsumfang ist zur Zeit noch nicht vollständig erfüllt.

Beispiel zur Verwendung der Funktion:

```
TASK(TaskX)
{
        ...
        GetResource(Res1);
        ...
        //Resource verwenden
        ...
        ReleaseResource(Res1);
        ...
}
```

7.2 Resource (binären Semaphor) freigeben, ***ReleaseResource()***

Syntax

StatusType ***ReleaseResource***(ResourceType ResID)

Inputs

ResID: Name(Identifier) der Ressource

Return

Status	Beschreibung
E_OK	Funktionsabarbeitung erfolgreich
E_OS_ID	ID unbekannt (nicht existent)
E_OS_ACCESS	Ressource bereits belegt
E_OS_NOFUNC	Ressource ist nicht belegt

Beschreibung der Funktion ***ReleaseResource()***

Um eine Ressource wieder freizugeben wird die Funktion ***ReleaseResource()*** verwendet. Die Priorität der Task wird wieder hergestellt. Der Aufruf ist nur von der Task zugelassen welche die Ressource belegt hält. Der von der OSEK Spezifikation vorgegebene Funktionsumfang ist zur Zeit noch nicht vollständig erfüllt.

Beispiel zur Verwendung der Funktion:

```
TASK(TaskX)
{
        ...
        GetResource(Res1);
        ...
        //Resource verwenden
        ...
        ReleaseResource(Res1);
        ...
}
```

8 HSE Free OSEK V0.1

8.1 Zusammenfassung

Innerhalb dieser Studienarbeit ist es nicht gelungen ein vollwertiges OSEK Betriebssystem zu schreiben. Die meisten Funktionen und Methoden sind zwar vollständig OSEK spezifisch, allerdings noch nicht alle. Das Betriebssystemmittel Messages (Inter-Prozess-Kommunikation) fehlt komplett und der Unterschied zwischen den Konformitätsklassen beschränkt sich im wesentlichen nur darauf, ob die Tasks in den Zustand Waiting gelangen können oder nicht. Des weiteren ist noch keine Unterstützung von Interrupts der Kategorie 2 vorhanden und aus Interrupts heraus können nur bestimmte API Funktionen aufgerufen werden. Das Problem war einfach der Zeitmangel und der Aufwand den dieses Projekt mit sich bringt.

Nichts desto trotz kann man HSE Free OSEK verwenden und die wichtigsten Elemente eines Multitasking-Betriebssystems sind vorhanden. Für Privatanwender oder Studenten, die ein kostenloses RTOS anwenden möchten, ist es sehr geeignet. Der Vorteil zu anderen Echtzeitbetriebssystemen ist der, dass es nichts kostet. Im Rahmen der Studienarbeit wurde auf Elemente zugegriffen die sich jeder aus dem Internet besorgen kann. Ein weiterer Vorteil zu anderen freien Betriebssystemen ist die Dokumentation welche z.B. bei Free RTOS zum derzeitigen Stand nicht in dem Umfang für die Öffentlichkeit vorhanden ist.

Die Studienarbeit dient im allgemeinen als eine Grundlage zur Weiterentwicklung. Jeder hat die Möglichkeit den Sourcecode zu ändern oder ihn zu vervollständigen. Das allgemeine Ziel sollte dabei das Erreichen eines vollständig portablen und vollständigen OSEK Betriebssystems sein das nichts kostet und gut dokumentiert ist.

8.2 Ausblick

Wie es mit dieser Studienarbeit weitergeht ist im Moment unklar. Es ist zu hoffen, dass es weitere Studienarbeiten oder Diplomarbeiten geben wird. Zusätzlich besteht die Option eine Community zu Gründen die das Produkt weiter entwickelt. Die bisherigen Entwickler, Franjo Rupcic und Mario Penava, stehen allen Nachfolgenden Entwickler als Kontaktpartner zur Verfügung und haben die Motivation zumindest teilweise in den Weiterentwicklungsprozess einen Teil mit zu wirken. Nach unserer Auffassung sollten die weiteren Schritte folgende sein:

- Entwicklung des Betriebssystemmittels Messages
- Interrupts der Kategorie 2 vervollständigen
- Prüfen der OSEK konformität (ist auch wirklich alles OSEK spezifisch?)
- Funktionalität und Eigenschaften der Konformitätsklasse vervollständigen
- Eine reale Echtzeitapplikation die typisch für OSEK Betriebssysteme ist mit HSE Free OSEK zu betreiben
- Entwicklung einer GUI zur Systemerzeugung (s.h. ProOSEK und OIL-File Generator)
- Unterstützung der OIL-File
- HSE Free OSEK auf kostenlose Compiler/Linker portieren (s.h. GNU)
- Portierung auf verschiedene Derivate

Qullenangaben:

- Buch: MicroC/OS II. The Real Time Kernel
- OSEK/VDX Spek. V2.2.3
- Skript: Realzeitsysteme von Jens Härer
- Skript: Prozessdatenverarbeitung von Prof. Dr. Jörg Friedrich
- Buch: OSEK von Matthias Homann
- Studienarbeit: Portierung von MicroC/OS II auf ein HCS12 Derivat (Alexander Hess)
- Datenblätter der Hardwarederivate (HCS12, LPC2148)

Beiliegender Anhang:

CD die alle Datenblätter und weitere Skripte enthält welche im Laufe der Studienarbeit verwendet worden. Der Quellcode für die beiden Derivate ist ebenfalls darauf zu finden.