ESE Kongress 2010

Vortragsskript:

CMSIS für ARM Cortex-Mx Microcontroller

Cortex™ Microcontroller Software Interface Standard defined by the ARM Company

Ingo Pohle, MicroConsult

Die Anforderungen zur Reduktion des Energieverbrauches und verbesserte Sicherheitssysteme resultieren in komplexere Steuerungen von Maschinen und Modulen in Fahrzeugen.

Ergebnis dieser Anforderungen ist der Entwurf von komplexeren Software-Designs. Damit steigen die Aufwendungen bei der Software-Entwicklung und dem Software-Test. Zur Vermeidung von Fehlern und Ausfällen der neuen, komplexen Systeme muss das Software-Design eine klar strukturierte Architektur aufweisen.

Prozessmodelle wie z. B. das V-Modell definieren eine klare Vorgehensweise bei der Software Entwicklung (Anforderungs-Analyse-> System-Spezifikation->System-Design->Coding->Unit-Test ->Modultest->Systemtest->Abnahmetest).

Mit dem Einsatz dieser modularen Entwicklungstechniken resultieren übersichtliche, zuverlässige und leichter testbaren Software-Architekturen. Ein weiterer Gesichtspunkt einer erfolgreichen und zukunftsorientierten Entwicklung von Applikationssoftware ist der Aspekt der **Wiederverwendbarkeit** der Software-Module. Dies kann durch den strikten Einsatz von **Abstraktionsschichten** (Abstraction Layer) in der Software erreicht werden:

Hardware Module
sollten nur über eine Hardware-Abstraktionsschicht (**Hardware Abstraction Layer HAL**) angesprochen werden.

Betriebssysteme
sollten nur über eine Betriebssystem-Abstraktionsschicht (**Operating System Abstraction Layer)** angesprochen werden.

Es sind zu diesem Zweck einige Standards definiert worden, die die Abhängigkeiten zwischen der Applikationssoftware und implementierter Hardware und Betriebssystemen vermeiden. Dies geschieht durch den Einsatz von Abstraktionsschichten.

Einige Beispiele für Standards aus dem Bereich **Applications-Entwicklung für Automobile**:

AUTOSAR	**AUT**omotive **O**pen **S**ystem **AR**chitecture (General Standard)
OSEK	**O**ffene **S**ysteme und deren Schnittstellen für die **E**lektronik im **K**raftfahrzeug

Applikationsunabhäniger Standard für die Software-Entwicklung:

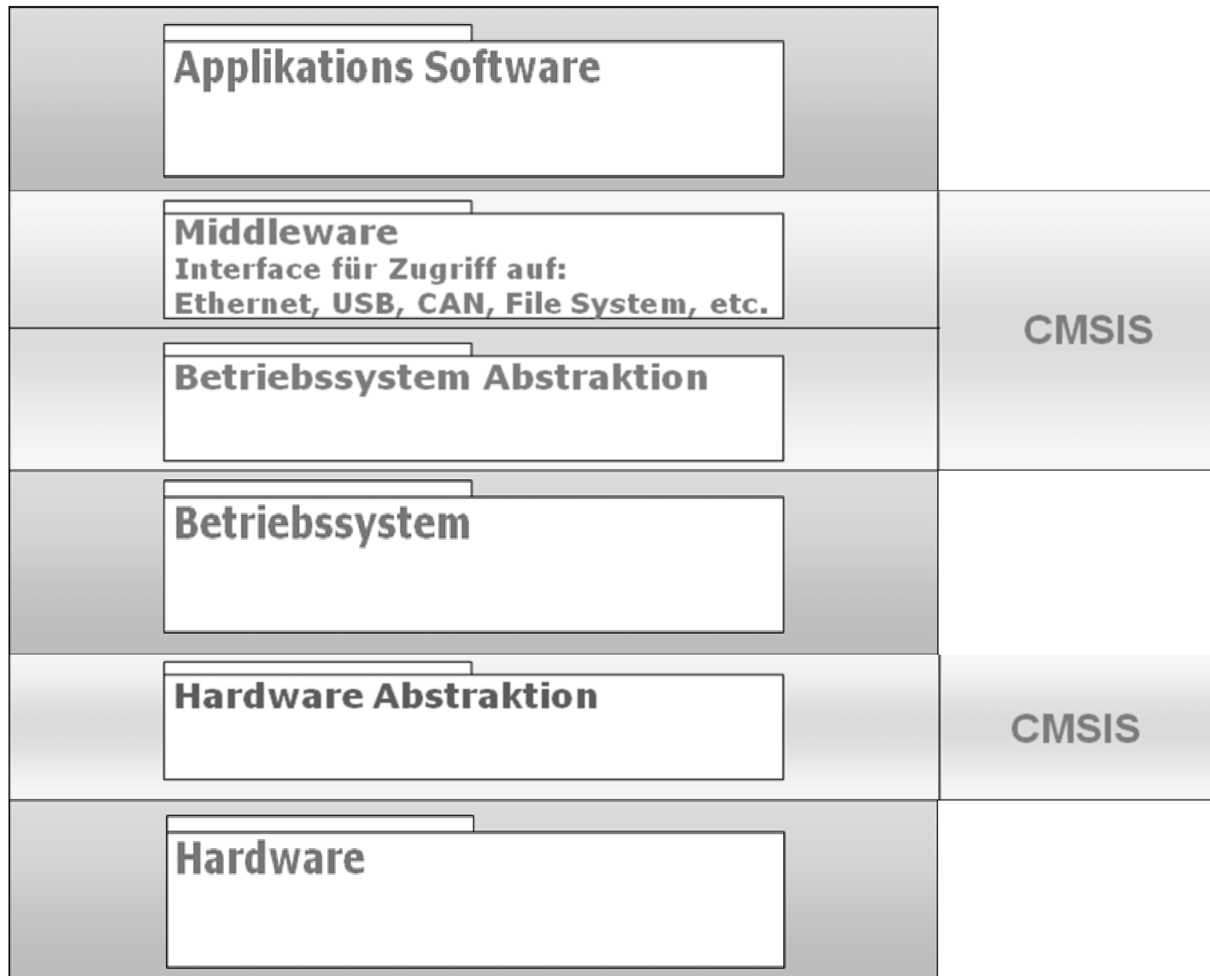
CMSIS Cortex™	**M**icrocontroller **S**oftware **I**nterface **S**tandard

Struktur der CMSIS Software-Schichten
Der ARM® Cortex™ Microcontroller Software Interface Standard (CMSIS) definiert eine anbieter-unabhängige Hardware-Abstraktionsschicht für die **Cortex-M** Prozessoren:

Cortex-M0 and **Cortex-M3**.

Der CMSIS Standard definiert die Software-Interfaces zwischen der Applikation und:

CPU / Core
Peripherie
Realzeit-Betriebssysteme (Real-Time Operating Systems RTOS)
Middleware



Aspekte für Software-Implementierung basierend auf dem CMSIS Standard

CMSIS ist eine Standardisierung für Software-Module für **Cortex-M0, Cortex-M3** und in der Zukunft auch für **Cortex-Mx** Bausteine und ist die Basis für:

Standardisierung für Zugriffe auf einfache und komplexe Peripherie Module
Verbessert die **Software-Wiederverwendbarkeit**, die CMSIS-basierende SW ist unabhängig von: Chip-Hersteller Compiler-Hersteller
Reduktion der **Produkteinführungszeit** (time to market)
Austauschbarkeit und **Ersetzbarkeit** von **Software-Modulen** für **Peripherie** und **Systemsteuerung**

Die CMSIS-basierende Software-Struktur wird durch den CMSIS Standard definiert:

Der CMSIS Standard wurde in Kooperation mit Mikrocontroller- und Tool-Herstellern definiert. Er enthält Definitionen für die Interfaces zwischen Applikationssoftware und dem System (CPU/Core), den System-Peripherien (wie den SysTick, NVIC, etc.), dem Real-Time Operating Systems RTOS, und den Middleware-Komponenten (wie USB, Ethernet, CAN, File-System etc.).

Folgende **generelle Definitionen** sind im CMSIS-Standard enthalten:

CMSIS - Generelle Definitionen	
Generelle Namensdefinitionen	Default Regeln für die Symbol-Definitionen
Generelle Adressdefinitionen	Definition of pointer to absolute address

Folgende Definitionen sind im CMSIS Standard für den **Zugriff auf** die **Cortex-Mx Core Register-** und die **Core-Peripherien** enthalten:

Zugriff auf ARM Cortex-Mx Core Peripherien über den **Core Peripheral Access Layer**, dieser enthält folgende Regeln:	
Core-Register-Zugriffsdefinitionen	**Makros** für **Core-Register-Zugriff**, sie enthalten Spezial-Cortex-M Assemblerbefehle (Treiber)
Core-Peripherie-Zugriffs-funktionsdefinitionen	Zugriff auf SysTick Timer, Nested Vectored Interrupt Controller NVIC, etc. (Diese Definitionen gelten für die Cortex-Architekturen Cortex-M0 und Cortex-M3)
Core Exception Vector Definitionen	Definitionen für die Vektortabelle für die Core-Exceptions
Hilfsfunktions-Definitionen	Makro-Definitionen für den Zugriff auf Core-Register und Core-Peripherien
Baustein-unabhängiges Interface für **RTOS Kernel-Funktionen**	Incl. Debug Channel Definitions

Folgende Definitionen sind im CMSIS Standard für den **Zugriff auf herstellerspezifische Peripherien** enthalten:

Zugriff auf herstellerspezifische Peripherien über den **Device Peripheral Access Layer**, dieser enthält folgende Regeln:	
Peripherie-Register-Zugriffsdefinitionen	Regeln für die Peripherie-Register-Zugriffsfunktionen (Treiber)
Peripherie-Zugriffsfunktionsdefinitionen	Regeln für die Peripherie-Zugriffsfunktionen (Treiber)
Peripherie Exception Vector Definitionen	Definitionen für Interruptvektoren von Peripherien
Hilfsfunktions-Definitionen	**Inline-Funktionen** oder **bausteinspezifische Bibliotheken**, diese werden von den Bausteine-Herstellern zur Verfügung gestellt.
Compiler-Hersteller und b**austeinspezifische Startup-Dateien**	**System Startup Dateien (startup_device.c, system_device.c)**

Beispiel für den Core Peripherie zugriff basierend auf dem CMSIS.Standard

Generelle Definitionen:

Der CMSIS C-Code ist konform zu den MISRA 2004 Regeln.
Die ANSI Standard Daten Typen werden im ANSI C Standard Library Header-File **<stdint.h>** definiert.
#define Konstanten müssen die Ausdrücke in runden Klammern darstellen, z. B.: #define PERIPH_BASE ((u32)0x40000000)
Variable und Parameter werden mit ihrem kompletten Datentyp definiert.
Alle Funktionen für den **Core Peripheral Access Layer** müssen **re-entrant** sein.
Der **Core Peripheral Access Layer** enthält keine blockierende Codesequenzen (Warteschleifen wie „wait loops“ werden in anderen Software Schichten durchgeführt).
Für **Exceptions** bzw. **Interrupts** gibt es spezielle Definitionen.

CMSIS Regeln für Exception/Interruptvektoren und Handler-Routinen:

Namen für **Exception Handler** enthalten den Post fix:	**_Handler**	Beispiel für ein Handler Symbol: **SysTickHandler**
Namen für **Interrupt Handler** enthalten den Post fix:	**_IRQHandler**	Beispiel für ein Interrupt Symbol: **TIM2_IRQHandler**
Default Exception Handler **Default Interrupt Handler**	Default-Handler enthalten eine **Endlos-Schleife**	Beispiel: **void UsageFaultException(void){** **while (1) // run infinite loop if** **// Usage Fault exception occurs** **}** **}**
Interrupt-Nummern	Werden definiert als:	**#define** mit dem Post fix **_IRQn**, z. B.: **// TIM2 globaler Interrupt** **#define TIM2_IRQChannel ((u8)0x1C)**

CMSIS-Dateien
Folgende CMSIS-Dateien stehen für einen Zugriff auf die Cortex-M-Hardware und Peripherie zur Verfügung:

Datei	**Anbieter**	**Beschreibung**
device.h *(bausteinspezifisch)*	Bausteinhersteller	Definiert die Peripherien für den aktuellen Baustein. Diese Datei kann für die Definition der Peripherie eine oder mehrere Dateien inkludieren.
core_cm0.h **core_cm3.h**	ARM (verfügbar für verschiedene Compiler Typen)	Definiert die für die Bausteine Cortex-M0 / Cortex-M3 CPU und Core-Peripherien.
core_cm0.c **core_cm3.c**	ARM (verfügbar für verschiedene Compiler Typen)	Enthalten Hilfsfunktionen für Zugriff auf die Core-Register der Bausteine Cortex-M0 / Cortex-M3.
startup_*device*	ARM (adaptiert durch den Compiler- bzw. Bausteinhersteller)	Enthält den Cortex-M Startup-Code und die komplette (bausteinspezifische) Interrupt-Vektortabelle.
system_*device*	ARM (adaptiert durch den Baustein-hersteller)	Enthält eine bausteinspezifische Konfigurationsdatei, diese konfiguriert den Baustein und optional die On-Chip PLL.

CMSIS Cortex Core Auswahl

Die Datei **core_cm3.h** enthält den CMSIS Cortex-M3 Baustein-Typ, definier wie folgt:
#define __CORTEX_M (0x03)

Die Datei **core_cm3.h** enthält den CMSIS Cortex-M0 Baustein-Typ, definier wie folgt:
#define __CORTEX_M (0x00)

CMSIS Cortex CPU und Core-Peripherie Initialisierung

Die Datei ***device.h*** wird durch den Bausteine-Hersteller zur Verfügung gestellt und enthält:

Definitionen der Interrupt-Nummern:
enthält Interrupt-Nummern (IRQn) für alle core- und bausteinspezifischen Exceptions und Interrupts.

Die Datei **core_cm0.h** bzw. **core_cm3.h** enthält die **Konfiguration** des im aktuellen Baustein verwendeten **Cortex-M Prozessors**. Die Datei inkludiert die Datei für die Prozessorauswahl: **core_cm0.h bzw. core_cm3.h**.

Der **Device Peripheral Access Layer** enthält die Definitionen für alle **bausteinspezifischen Peripherien**. Hier sind **Datenstrukturen** und das **Adress-Mapping** für die bausteinspezifischen Peripherien enthalten.

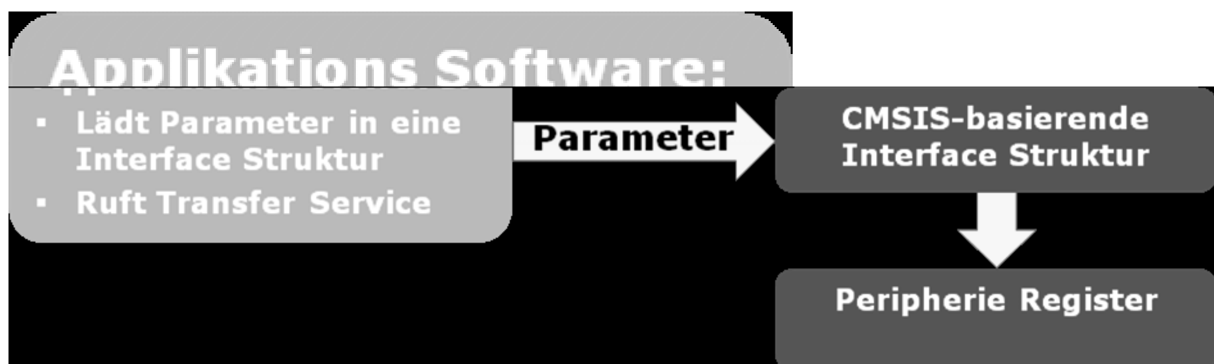
Peripherie-Zugriffsfunktionen:
Hier sind zusätzliche Hilfsfunktionen für die Programmierung der Kontrollfunktionen und den Zugriff auf Status-Informationen der Peripherien enthalten. Diese Zugriffsfunktionen werden vom Baustein-Hersteller zur Verfügung gestellt und enthalten:

Inline-Funktion oder **bausteinspezifische Bibliotheken**

Für die **Initialisierung** und **Steuerung** der **CPU,** der **Core-Peripherien** und der **bausteinspezifischen Peripherien** gibt es folgende Definitionen:

Definitionen der **Peripherie** **Modul- Register-Strukturen**	Hier wird der komplette Registersatz des physikalischen Speichers der Peripherie abgebildet.
Initialisierungsfunktionen	Initialisierungsfunktionen für die Einstellung des Modes vom Core oder einer Peripherie.
Kontroll-Funktionen	Steuerungsfunktionen für Core/CPU/Peripherie (werden für den Start und Stopp von Core/CPU/ Peripherie verwendet.
Zugriffsroutinen für Status-Informationen	Lese-/Schreibfunktionen für den Zugriff auf Status-Informationen (z. B. zum Pollen oder Rücksetzen von Status-Flags)

Für die Übertragung von Initialisierungswerten in die physikalischen Register der Peripherie werden zunächst **temporäre Interface-Speicherstrukturen** mit den notwendigen Setup-Werten geladen. Mit speziellen **Initialisierungsroutinen** werden dann die vorgeladenen Werte in die physikalischen Register des Mikrocontrollers kopiert/transferiert.



Temporäre Interface-Struktur
Für die Initialisierung von Core/CPU/Peripherie Registern werden **temporäre Interface-Strukturen** verwendet.

Beispiel einer **temporären Interface-Register-Struktur** für die **SysTick-Initialisierung**:

```
typedef struct
{
vu32 CTRL; // SysTick Control und Status-Register
vu32 LOAD; // SysTick Register für den Reload-Wert
vu32 VAL; // SysTick Register für den aktuellen Zählwert
vuc32 CALIB; // SysTick Register für den Kalibrierungswert
} SysTick_TypeDef;
```

CSMSIS Firmware Library Funktionen
Für jeden Cortex-Core und jede bausteinspezifische Peripherie gibt es einen Set von Bibliotheks-funktionen:

Verbinden des internen Clock-Takts mit dem Modul (spezifische Peripherie)

Initialisation/De-Initialisation des **Moduls**

Zugriff auf die **Register** des **Modules:** read-only, write-only bzw. read/write Zugriff

Zugriff auf **Status-Register**

Beispiel aus der **STMicroelectonics Firmware Library Funktion** für den **SysTick Timer**:

Funktionsname	**Beschreibung**
SysTick_CLKSourceConfig	Konfiguriert die SysTick Taktquelle
SysTick_SetReload	Initialisiert den SysTick Nachladewert
SysTick_CounterCmd	Gibt den SysTick-Zähler frei oder sperrt ihn
SysTick_ITConfig	Gibt den SysTick-Interrupt frei oder sperrt ihn
SysTick_GetCounter	Liest den aktuellen SysTick-Zählerwert
SysTick_GetFlagStatus	Testet ob das spezifische SysTick Status-Flag gesetzt oder nicht gesetzt ist

CSMSIS Firmware Library Funktions-Parameterliste

Für die Firmware Library Funktionen ist eine **Parameterliste** definiert, sie enthält:

Library Function Parameterliste:
- Funktionsname
- Funktions-Prototyp
- Verhaltensbeschreibung
- Eingabeparameter
- Ausgabeparameter
- Rückgabewert
- Benötigte Vorbedingung
- Aufgerufene Funktionen

Beispiel der **Parameterliste** für die **Funktion: SysTick Set Reload Value**

Parameterliste	**Bedeutung**
Funktionsname	SysTick_SetReload
Funktions-Prototyp	void SysTick_SetReload (u32 Reload)
Verhaltensbeschreibung	Setzt den SysTick Nachladewert
Eingabeparameter	Reload: SysTick Reload Neuwert. Dieser Parameter muss eine Zahl zwischen 1 und 0x00FFFFFF sein.
Ausgabeparameter	Keiner
Rückgabewert	Keiner
Benötigte Vorbedingungen	Keine
Aufgerufene Funktionen	Keine

Beispiel für den **Funktionsaufruf**: **SysTick Set Reload Value**

SysTick_SetReload(0xFFFF); // Setze den SysTick Nachladewert auf 0xFFFF

Speicherbedarf für die CMSIS Software-Abstraktionsschicht

Bei einer CMSIS-basierenden Software-Entwicklung werden die Zugriffe zwischen der Applikations-Software und CPU/Core/Peripherien und RTOS-Treibern über die **CMSIS Software-Abstraktionsschicht** durchgeführt. Diese Software-Schicht benötigt z. B. für den Zugriff auf die Core-Peripherie weniger als 1 Kbyte im Code-Speicher und weniger als 10 Bytes im RAM.

Der CMSIS-Standard ist für folgende Compiler verfügbar:
RealView ARMCC, IAR, und GNU GCC

Wichtiger Hinweis für die Verwendung von CMSIS

CMSIS schützt die Applikationssoftware nicht vor einem direkten Hardware-Zugriff.
Die Peripherie-Register werden über Speicherzugriffe angesprochen; hierbei werden mit Kopierroutinen vorgeladene CMSIS-basierende Speicherstrukturen in die physikalischen Register der entsprechenden Peripherie des Bausteines kopiert.

Download des CMSIS-Standards
Den Download der CMSIS-Spezifikation ist auf folgender **Webseite** möglich:
http://ww.onarm.com/download/download395.asp

Dort finden Sie die CMSIS-Datei mit der Version V1.30 (30. Oktober 2009): **CMSIS_V1P30.ZIP**

Quellenangaben:
Cortex Microcontroller Software Interface Standard
Version: 1.30 - 30. October 2009
Download der Firma ARM Ltd.: http://ww.onarm.com/download/download395.asp

Autor:
Ingo Pohle, Manager für Training & Coaching bei der MicroConsult GmbH in München