Vorlesungsskript zu „Vertiefung Programmieren“ Software Module

Dozent: Dipl.-Inf. (FH) Andreas Schmidt

DHBW
Duale Hochschule
Baden-Württemberg

Layered Architecture

Standard Layered Architecture

Layered Architecture

Mögliche Architekturen Pattern für Embedded Systeme

- **Monolithic Architecture**
  Die Software besteht aus einem großen Block ohne klare Trennung von Funktionalitäten

- **Layered Architecture**
  Die Software ist in Schichten mit spezifischen Funktionalitäten und Schnittstellen aufgeteilt.

- **Microkernel Architecture**
  Es gibt eine Kern-funktionalität (Microkernel) und Service Module, die auf diese Funktionalität zurückgreifen

- **Event-Driven Architecture**
  Software Komponenten reagieren asynchron auf Ereignisse und Nachrichten

- **Message Passing Architecture**
  Software Komponenten kommunizieren mit Hilfe von Nachrichten (z.B. Message Queues)

- **Component Based Architecture**
  Das System besteht aus losen gekoppelten, wiederverwendbaren Komponenten, die über eine definierte Schnittstelle miteinander interagieren.

- **Client-Server Architecture**
  Das System ist in Server und Client Komponenten aufgeteilt, wobei die Client-Komponenten bei den Server-Komponenten Services nutzen

Quelle: intern

Layered Architecture

Eigenschaften und Aufbau einer Layered Architecture

- In Embedded Systemen (insb. sog. Bare-Metal Systemen) wird sehr häufig eine Layered Architecture eingesetzt.
- Hierbei wird die Software in einzelne, aufeinander aufbauende „Schichten“ aufgeteilt
- Jeder Schicht wird eine bestimmte Verantwortung/Funktionalität zugewiesen. Software Module, welche eine bestimmte Funktionalität realisieren, werden hierbei genau einer definierten Schicht zugewiesen.
- Je weiter „oben“ eine Software Schicht in der Layered Architecture angesiedelt ist, desto höher ist i.d.R. auch der Abstraktionsgrad dieser Schicht.

Quelle: intern

Layered Architecture

Eigenschaften und Aufbau einer Layered Architecture

- Embedded System verwenden häufig folgende Schichten:
  - **HAL – Hardware Abstraction Layer**
    Abstrahiert die darunter liegende Hardware bzw. den Microcontroller und dessen Peripherie (z.B. GPIO, UART, PWM etc.). Diese Schicht stellt in der Regel ein API bereit, mit dem ohne Kenntnisse über Controller-spezifische Details (z.B. Config-Register etc.) mit der Hardware interagiert werden kann.

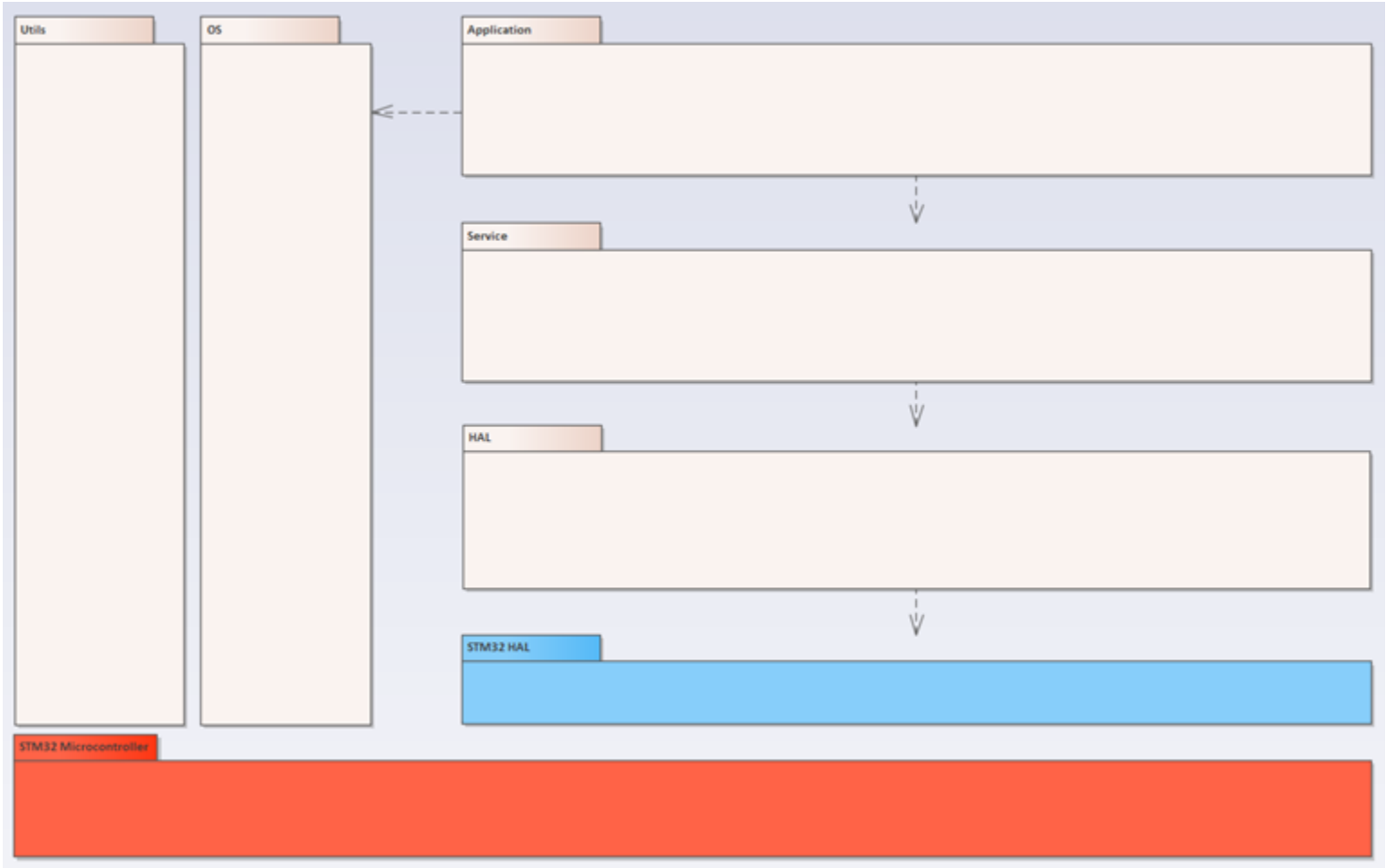
  - **Service – Service Layer**
    Abstrahiert mit Hilfe des darunter liegenden HAL Layers weitere Hardware und Funktionalitäten. Dies kann z.B. die Abstraktion von Hardware-Komponenten außerhalb des Microcontrollers sein. (Z.B. Sensoren, Speicherbausteine, Kommunikationsschnittstellen etc.). Dennoch muss diese Schicht nicht zwingend Hardware abstrahieren. Eine weiter Funktionalität die häufig auf diesem Layer liegt, ist z.B. ein Fehlerspeicher, Kommunikationsprotokolle, Signalauswertung etc. Ein häufiger Grund eine bestimmte Komponente in dem Service Layer zu platzieren ist, dass die Funktionalität z.B. nicht projekt- oder Applikation-spezifisch ist und u.U. in anderen Kontexten wieder verwendet werden kann.

  - **Application – Application Layer**
    Der Application Layer beinhaltet alle projekt- bzw. Applikations-spezifischen Funktionalitäten.

Quelle: intern

Layered Architecture

Aufbau Software Module



Quelle: intern

Layered Architecture

Eigenschaften und Aufbau einer Layered Architecture

- Kern-Ideen der Layered Architecture sind:
  - Module/Funktionalitäten werden genau einer definierten Schicht zugewiesen
  - Module einer Schicht können nur Funktionalitäten der darunter liegenden Schicht verwenden
  - Eine direkte Nutzung von Funktionalitäten der darüberliegenden Schicht ist nicht gestattet (Ein Sonderfall stellen hier sog. Callback-Funktionen z.B. via Funktions-Pointer dar)
  - Ein Modul einer Schicht, kann auch darunter liegenden Schichten „überspringen“. Z.B. kann ein Modul auf dem Application Layer auch Funktionen direkt aus der HAL Schicht nutzen. (Auch wenn dies die Wiederverwendbarkeit/Portierbarkeit einschränkt)

Quelle: intern

DHBW
Duale Hochschule
Baden-Württemberg

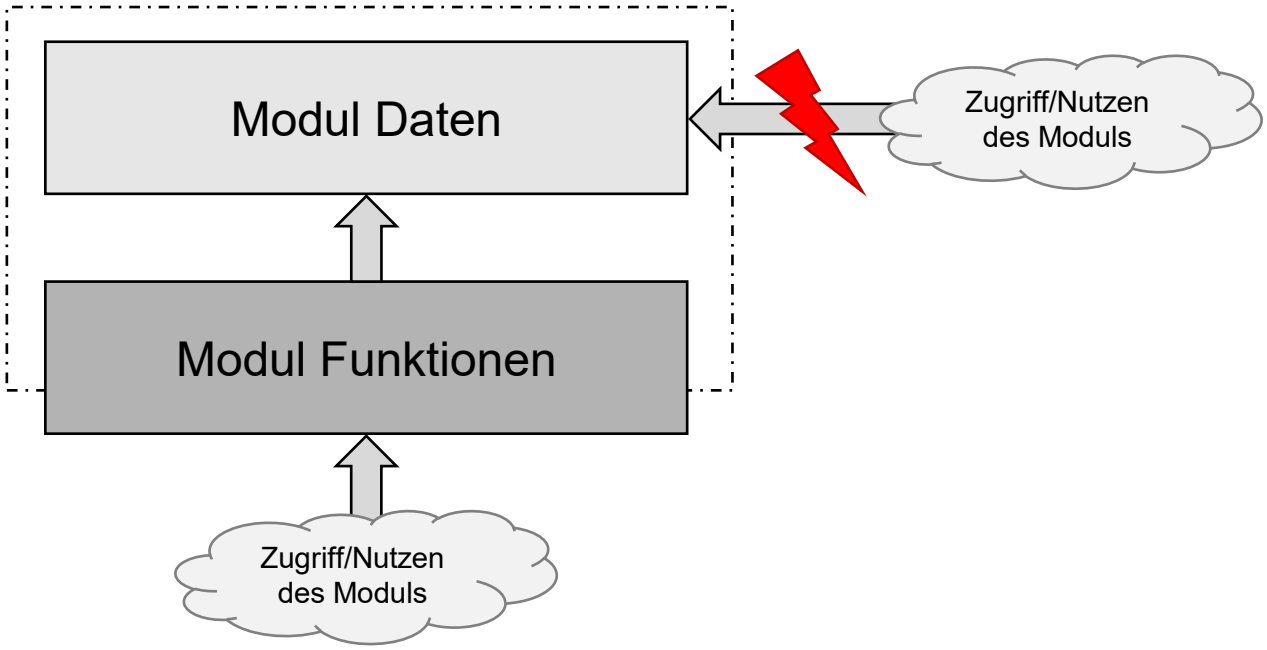
Software Module

Modul-Struktur und Interfaces

Software Module

Aufbau Software Module

- Ein Software-Modul kapselt bzw. fasst Daten und Funktionen zusammen, welche eine definierte Funktionalität bereitstellen.
- In der Regel operieren nur die Funktionen eines Moduls auf den Modul-Daten



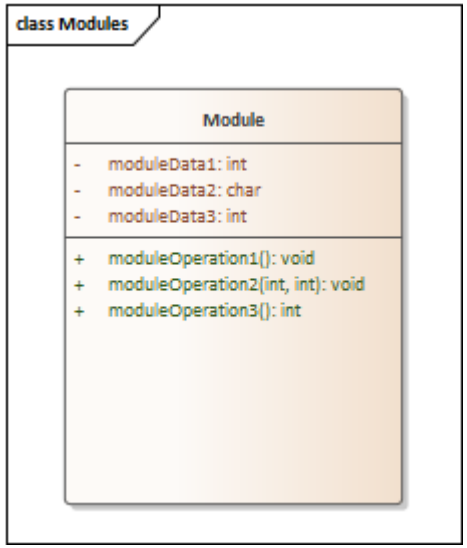
Quelle: intern

Software Module

Aufbau Software Module

- Konzept ist ähnlich von „Klassen“ einer objektorientierten Programmiersprache

- Wichtiger Unterschied zu Klassen/Objekten: **Instanziierung**
  - Daten eines Moduls werden als globale Variablen realisiert
  - Können nicht wie eine Klasse instanziiert werden, daher auch nicht mehrere Instanzen eines Moduls möglich
  - Klassen-Konzept kann aber mit Hilfe von Strukturen (struct) nachgebildet werden



Quelle: intern

Software Module

„Arten“ eines Software Modules

- **Multi-Instanz Module**
  - Konzept ähnlich der Klassen
  - Kapselung der Modul-Daten in Strukturen (struct)
  - Keine Modul-Globalen Daten/Variablen (aber u.U. Konstanten)
  - Funktionen des Moduls operieren i.d.R. auf den Parametern
  - Sehr flexible Wiederverwendung (Instanz wird als Parameter übergeben)
  - Gut testbar

- **Single-Instanz Module („Singleton“)**
  - Daten eines Moduls sind als globale Variablen realisiert ➔ Compiler/Linker reservieren Speicher ➔ festgelegt während der Compile-Zeit
  - Funktionen des Moduls operieren i.d.R. auf den globalen Variablen des Moduls
  - Wiederverwendung möglich, allerdings nur „alles oder nichts“
  - Testbarkeit hängt von Modul-Einsatz/Zweck ab

Quelle: intern

Software Module

Realisierung eines Moduls

- In C/C++ besteht ein Modul in der Regel aus:
  - einem Header-File mit öffentlichen Funktionen und Konstanten/Datentypen ➔ Public Interface
  - einem Source-File mit der Implementierung der Funktionen. Hier können u.U. auch „private“ Funktionen deklariert und definiert sein
- Sollte ein „Software Modul“ sehr komplex sein, so empfiehlt sich u.U. die Aufteilung der Header- und Source-Files auf mehrere Dateien (Teil-Funktionalitäten)

Quelle: intern

Software Module

Best Practices

- Header- und Source-File sollten eine feste Struktur haben.
  - Header File
    - File Comment Header
    - Public Constants and Defines
    - Public Data Types
    - Public Functions
  - Source File
    - File Comment Header
    - Include Files
    - Private Constant and Defines
    - Private Data Types
    - Private Function Declarations
    - Global Data
    - Function Implementation (Private + Public)

Quelle: intern