Embedded Software Engineering Kongress

ESE Kongress 2017

Vortragsskript:

Moderne Low-Level-Treiberprogrammierung

CMSIS, MCAL und Co. – Low Level Treiber von der Stange

Renate Schultes, MicroConsult GmbH

Embedded-Systeme sind heute allgegenwärtig und stellen in vielen Bereichen einen wichtigen Faktor dar, wenn es um Komfort, Sicherheit, Nachhaltigkeit und Innovation geht. Der Anteil der Software in Embedded-Systemen steigt immer mehr an. Die Hardware, ob Mikroprozessor mit externer Peripherie oder Mikrocontroller, wird immer komplexer. Multicore ist bereits Realität, und immer mehr Hersteller bringen neue Multicore-Derivate auf den Markt. Eine Abstraktion der Hardware ist unumgänglich, denn diese komplexe Hardware selbst bis in das letzte Bit zu kennen – und zu programmieren – ist in der dafür zur Verfügung stehenden Zeit nicht mehr möglich.

Bei den 8-Bit und 16-Bit Mikrocontrollern war es noch üblich, dass die Bausteinhersteller und/oder die Toolhersteller Headerfiles geliefert haben, die Symboldefinitionen für alle Steuer-/Status- und Arbeitsregister der Peripheriemodule enthielten. Nicht selten waren diese Dateien bei 16-Bit Mikrocontrollern schon mehrere tausend Zeilen lang, wie das folgende Beispiel mit insgesamt 12 000 Zeilen zeigt.

```
7648  // Capture/Compare Register for Channel CC60
7649  #define CCU61_CC60R          (*((uword volatile *) 0xEA98))
7650
7651  // Capture/Compare Shadow Reg. for Channel CC60
7652  #define CCU61_CC60SR         (*((uword volatile *) 0xEAA0))
7653
7654  // Capture/Compare Register for Channel CC61
7655  #define CCU61_CC61R          (*((uword volatile *) 0xEA9A))
7656
7657  // Capture/Compare Shadow Reg. for Channel CC61
7658  #define CCU61_CC61SR         (*((uword volatile *) 0xEAA2))
7659
7660  // Capture/Compare Register for Channel CC62
7661  #define CCU61_CC62R          (*((uword volatile *) 0xEA9C))
7662
7663  // Capture/Compare Shadow Reg. for Channel CC62
7664  #define CCU61_CC62SR         (*((uword volatile *) 0xEAA4))
```

Abb. 1 Ausschnitt aus der Datei XE16xREGS.H [1]

Komplexe 32-Bit (Singlecore/Multicore) Mikrocontroller enthalten Peripheriemodule, bei denen ein einzelnes Modul bereits mehrere hundert Steuer-/Status- und Arbeitsregister zur Verfügung stellt. Und dann gibt es häufig mehrere gleichartige Peripheriemodule in einem Baustein.

Unterschiedliche Bausteinvarianten enthalten oft eine unterschiedliche Anzahl der jeweiligen Peripheriemodule des gleichen Typs. Die Anzahl der Register und der dafür notwendigen Symboldefinitionen wird dadurch immer unübersichtlicher.

Hardware-Abstraktion

Anstelle dieser einzelnen Symboldefinitionen für jedes Peripherieregister kann der komplette Satz an Steuer-/Status-/Arbeitsregistern eines Peripheriemoduls mit Hilfe einer C-Struktur abgebildet werden. Die Register eines Peripheriemoduls liegen in einem festgelegten Adressraum – eventuell mit Lücken – hintereinander. Die Struktur bildet diese Register in der richtigen Reihenfolge und mit den Lücken ab.

STM32F10x.h

```
906  typedef struct
907  {
908    __IO uint16_t CR1;
909    uint16_t  RESERVED0;
910    __IO uint16_t CR2;
911    uint16_t  RESERVED1;
912    __IO uint16_t OAR1;
913    uint16_t  RESERVED2;
914    __IO uint16_t OAR2;
915    uint16_t  RESERVED3;
916    __IO uint16_t DR;
917    uint16_t  RESERVED4;
918    __IO uint16_t SR1;
919    uint16_t  RESERVED5;
920    __IO uint16_t SR2;
921    uint16_t  RESERVED6;
922    __IO uint16_t CCR;
923    uint16_t  RESERVED7;
924    __IO uint16_t TRISE;
925    uint16_t  RESERVED8;
926  } I2C_TypeDef;
```

Abb. 2 Struktur für das I2C Modul eines STM32 Bausteins

Um dieses Abbild der Peripherieregister für mehrere Module des gleichen Typs im Baustein mehrfach wiederverwenden zu können, wird zusätzlich noch die Startadresse des jeweiligen Registerblocks benötigt.

STM32F10x_MAP.h

```
36  #define CAN1_BASEADR         0x40006400
37  #define CAN2_BASEADR         0x40006800
38  #define USB_CAN_SRAM_BASEADR  0x40006000
39  #define USB_FS_BASEADR       0x40005C00
40  #define I2C2_BASEADR         0x40005800
41  #define I2C1_BASEADR         0x40005400
42  #define UART5_BASEADR        0x40005000
43  #define UART4_BASEADR        0x40004C00
44  #define USART3_BASEADR       0x40004800
45  #define USART2_BASEADR       0x40004400
46  #define SPI3_I2S_BASEADR     0x40003C00
47  #define SPI2_I2S_BASEADR     0x40003800
48  #define IWDG_BASEADR         0x40003000
49  #define WWDG_BASEADR         0x40002C00
```

Abb. 3 Basisadressen verschiedener Peripheriemodule eines STM32 Bausteins

Ein Zeiger auf den entsprechenden Strukturtyp wird auf die Basisadresse des Registerblocks gesetzt. Gibt es zwei oder mehr Module des gleichen Typs, gibt es zwei oder mehr Zeiger von diesem Typ. Über die Zeiger kann auf alle Elemente der zuvor definierten Struktur (Register des Peripheriemoduls) zugegriffen werden.

```
1  /* Includes ------------------------------------------
2  #include "stm32f10x_i2c.h"
3  #include "stm32f10x_map.h"
4
5  int main(void)
6  {
7    I2C_TypeDef* pI2C1 = (I2C_TypeDef*)I2C1_BASEADR;
8    I2C_TypeDef* pI2C2 = (I2C_TypeDef*)I2C2_BASEADR;
9
10   pI2C1->
11         CCR
12         CR1
13         CR2
14         DR
15         OAR1
16
17
```

Abb. 4 Zeigerdefinition und Zugriff auf die Register des Typs I2C_TypeDef

In diesem Beispiel wird der Zugriff auf die Peripherieregister direkt in der Applikation durchgeführt. Der Nachteil dieser Art der Nutzung liegt auf der Hand – jede Änderung und Erweiterung, aber insbesondere die Wiederverwendbarkeit der Applikation in anderen Systemen ist problematisch, da jede Codezeile in der gesamten Applikation überprüft und unter Umständen angepasst werden muss. Jede Codezeile kann einen Verweis auf Peripherieregister enthalten und muss deshalb für andere Bausteine oder andere Aufgabenstellungen verändert oder entfernt werden. Der Applikationsprogrammierer muss außerdem wissen, was er mit den Peripherieregistern machen muss oder darf. Jeder lesende oder schreibende Zugriff auf eines der Register erfordert detaillierte Hardwarekenntnisse. Außerdem ist der Testaufwand sehr hoch, weil die gesamte Applikation direkt auf Register des Bausteins zugreift. Wer wann was und wie nutzt, muss bei jeder kleinen Änderung komplett neu getestet werden.

Software-Schichtenmodell

Besser ist eine saubere Trennung zwischen Applikationscode und Low-Level-Treibercode. Dadurch entstehen zunächst unabhängige Software-Schichten (Software Layer – Software-Subsysteme), die über Schnittstellen kommunizieren. Die Subsysteme können getrennt voneinander entwickelt und getestet werden.

Die obere Schicht kann auf die darunterliegende Schicht über eine vordefinierte Schnittstelle (Interface) zugreifen. Dieses Interface ist in der Programmiersprache C nichts anderes als ein Headerfile. Allerdings darf die untere Schicht nicht auch über ein Interface auf die obere Schicht zugreifen, das würde zu bidirektionalen Abhängigkeiten führen. Der Zugriff von unten nach oben kann über Callback realisiert werden. In der Programmiersprache C werden dafür Funktionszeiger verwendet.

Eine Änderung in einem der Subsysteme hat keine Auswirkung auf das andere Subsystem, sofern die Schnittstelle nicht geändert wird.

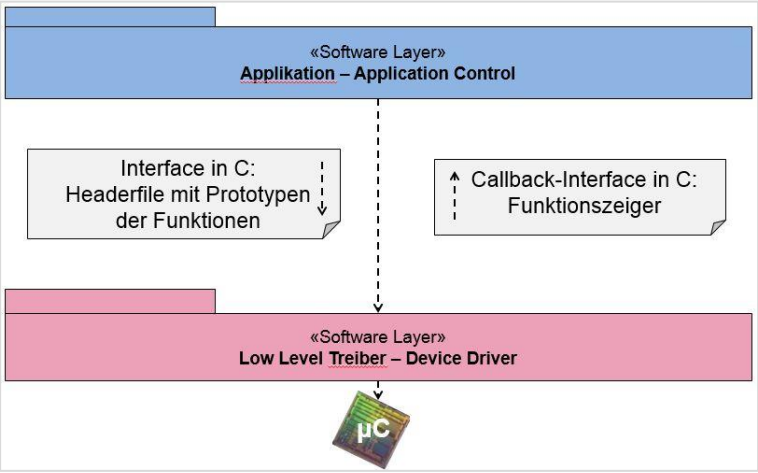

Abb. 5 Software-Schichten-Modell: 2-Schichten-Modell

Aus Sicht der Applikation könnte jetzt der Austausch des Headerfiles und der darunterliegenden Low-Level-Treiberschicht genügen, um bei einem Bausteinwechsel ohne viel Aufwand mit der gleichen Applikation weiterarbeiten zu können. Aber so einfach ist es leider meist nicht. Denn die Namen und die Parameterschnittstellen der Funktionen der Low-Level-Treiberschicht sind oft komplett unterschiedlich definiert. Der Austausch alleine reicht deshalb in den meisten Fällen nicht aus. Alte Funktionsnamen müssen in der Applikation ersetzt, die Übergabeparameterliste für jeden Aufruf überprüft und angepasst werden.

Hat ein anderer Baustein ein spezielles Peripheriemodul vielleicht gar nicht direkt zur Verfügung (zum Beispiel I2C Modul), muss es durch ein anderes Modul (zum Beispiel SPI Modul) emuliert werden. Dann müssen ganz andere Funktionen für die Initialisierung und Nutzung dieses Moduls in die Applikation eingebaut werden, und das bedeutet wieder zusätzlichen Zeitaufwand für die Implementierung und den Test.

Und hier kommt das 3-Schichten-Modell ins Spiel. Zwischen die Low-Level-Treiberschicht und die Applikationsschicht wird noch eine Low-Level-Treiberabstraktionsschicht geschoben. Diese Schicht verbirgt für den Nutzer nach oben (die Applikation) die tatsächlich vorhandene Hardware - also ob zum Beispiel eine echte I2C Schnittstelle in der Hardware zur Verfügung steht oder ob diese über eine andere serielle Schnittstelle emuliert werden muss.

Die Applikation liefert die Parameter für die richtige Einstellung, die Abstraktionsschicht darunter gibt diese an die passende Low-Level-Treiberkomponente weiter. Natürlich muss bei einem Bausteinwechsel die Abstraktionsschicht entsprechend angepasst werden.

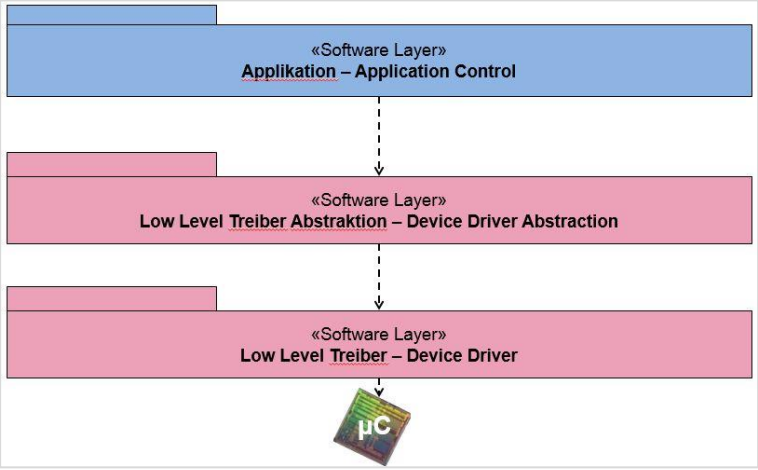

Abb. 6 Software Schichten Modell: 3-Schichten Modell

Aber unabhängig davon ob 2- oder 3-Schichten-Modell, der Low-Level-Treiber wird in jedem Fall benötigt. Und die entscheidende Frage ist jetzt: Selbst schreiben oder fertigen Treiber des Bausteinherstellers nutzen?

Selbst schreiben bedeutet, dass es einen „Hardware Experten“ geben muss, der die Funktionsweise der Peripheriemodule, die Register, die Bitfelder und Bits in den Registern genau kennt und die Initialisierung und Nutzung in die Programmiersprache umsetzt.

Da es keine allgemein gültigen Regeln (Namensregeln, Aufbau der Parameterschnittstelle, etc.) gibt, kann die Umsetzung für jeden Baustein oder sogar für jedes Peripheriemodul vollkommen unterschiedlich aussehen, und die Abstraktionsschicht muss jeweils angepasst werden.

```
void initI2C(void)
{

/*---------------------------- I2Cx CR2 Configuration ------------------------*/


/*---------------------------- I2Cx CCR Configuration ------------------------*/

  /* Configure speed */

  /* Write to I2Cx CCR */

/*---------------------------- I2Cx CR1 Configuration ------------------------*/

/* Configure I2Cx: mode and acknowledgement */

/* Write to I2Cx CR1 */

/*---------------------------- I2Cx OAR1 Configuration ------------------------*/
  /* Set I2Cx Own Address1 and acknowledged address */

}
```

Abb. 7 Low-Level-Treiberfunktion mit beliebigem Namen, keine Parameter

Werden keine Übergabeparameter an Low-Level-Treiberinitialisierungsfunktionen geliefert, müssen in der Funktion vorgegebene Werte für die Konfiguration verwendet werden. Das schränkt die Wiederverwendbarkeit sehr stark ein, weil für jede Art der Nutzung unterschiedliche Funktionen existieren müssen.

Um den Aufwand an dieser Stelle möglichst klein zu halten, muss es Regeln geben, wie viele und welche Parameter in welcher Reihenfolge an diese Funktionen übergeben werden. Außerdem ist es sinnvoll, die Funktionsnamen immer nach einem fest vorgegebenen Schema aufzubauen. Und das wiederum ist einer der Vorteile bei der Nutzung der „Treiber von der Stange“.

Spezifikationen und Standards

Bausteinhersteller und Softwareanbieter schließen sich zusammen und definieren Schnittstellen für den Zugriff auf Peripheriemodule, Echtzeitbetriebssysteme und Middleware Komponenten. Wird dann ein anderer Bausteintyp der Serie (zum Beispiel ein Cortex Derivat) eingesetzt, kann die Applikation unverändert bleiben, und die Abstraktion muss nur mit kleineren bausteinspezifischen Anpassungen versehen werden.

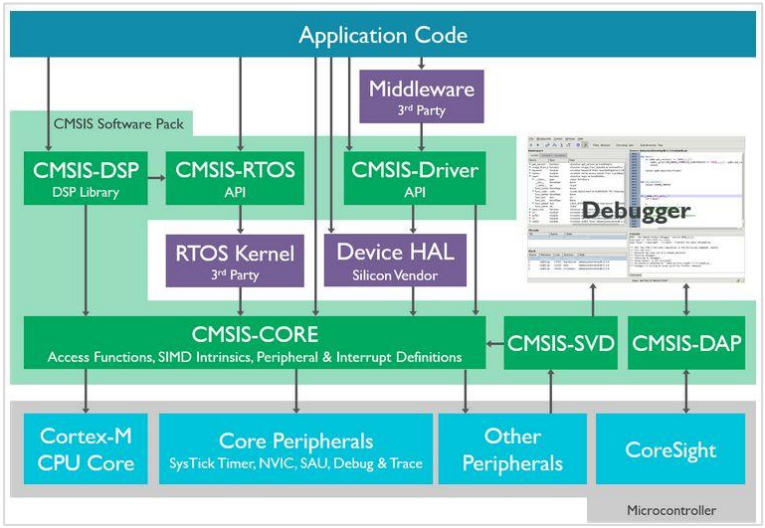

Abb. 8 CMSIS-Schichtenmodell

Die Standardisierung erleichtert also die Portierbarkeit und Wiederverwendbarkeit.

Low-Level-Treiber (CMSIS, MCAL, …)

Die fertig implementierten Low-Level-Treiber der Bausteinhersteller vereinfachen die Nutzung der Bausteinressourcen. Natürlich muss der Nutzer immer noch wissen, was er mit welchem Baustein realisieren kann. Aber die Details dieser Realisierung, welches Register, Bitfeld oder Bit mit welchem Wert beschrieben werden muss, sind im fertig implementierten Treiber verborgen.

Über eine Initialisierungsstruktur werden einzustellende Parameter an die Initialisierungsfunktion geliefert.

```
/**
  * @brief  I2C Init structure definition
  */

typedef struct
{
  uint32_t I2C_ClockSpeed;          /*!< Specifies the clock frequency.
                                         This parameter must be set to a value lower than 400kHz */

  uint16_t I2C_Mode;                /*!< Specifies the I2C mode.
                                         This parameter can be a value of @ref I2C_mode */

  uint16_t I2C_DutyCycle;           /*!< Specifies the I2C fast mode duty cycle.
                                         This parameter can be a value of @ref I2C_duty_cycle_in_fast_mode */

  uint16_t I2C_OwnAddress1;         /*!< Specifies the first device own address.
                                         This parameter can be a 7-bit or 10-bit address. */

  uint16_t I2C_Ack;                 /*!< Enables or disables the acknowledgement.
                                         This parameter can be a value of @ref I2C_acknowledgement */

  uint16_t I2C_AcknowledgedAddress; /*!< Specifies if 7-bit or 10-bit address is acknowledged.
                                         This parameter can be a value of @ref I2C_acknowledged_address */
}I2C_InitTypeDef;
```

Abb. 9 Initialisierungsstruktur für ein I2C-Modul

Die Applikation füllt die Initialisierungsstruktur mit den einzustellenden Werten und gibt die Basisadresse des zu initialisierenden Moduls und die Adresse der Initialisierungsstruktur an die untere Schicht weiter. Der Aufruf der Low-Level-Treiberfunktion steht im folgenden Beispiel wieder direkt im Applikationscode.

```
int main(void)
{
  I2C_TypeDef* pI2C1 = (I2C_TypeDef*)I2C1_BASEADR;

  I2C_InitTypeDef i2cInitStruct;

  i2cInitStruct.I2C_ClockSpeed = 100;
  i2cInitStruct.I2C_Mode = I2C_Mode_I2C;
  i2cInitStruct.I2C_DutyCycle = I2C_DutyCycle_2;
  i2cInitStruct.I2C_OwnAddress1 = 0x7;
  i2cInitStruct.I2C_Ack = I2C_Ack_Enable;
  i2cInitStruct.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;
  I2C_Init(pI2C1, &i2cInitStruct);
```

Abb. 10 Initialisierungsstruktur und Aufruf der Initialisierungsfunktion

Die nächste Abstraktionsebene wäre wieder die Trennung von Applikationscode und Low-Level-Treiberzugriffen. Der Applikationsschicht wird vom Hardware Abstraction Layer HAL eine Struktur zur Verfügung gestellt, die alles enthält, was für die Initialisierung und Nutzung eines Peripheriemoduls benötigt wird. Das heißt, neben der Initialisierungsstruktur weitere Strukturen, von denen eine zum Beispiel Funktionszeiger enthält, in die in der unteren Schicht die tatsächlich benötigten Funktionen des Low-Level-Treiber eingetragen werden. Die Applikation ruft über den Funktionszeiger auf und muss nicht mehr wissen, wer sich dahinter verbirgt.

```
int main(void)
{
    // open led HAL modules
    ledOpen( &led1.ctrl, &led1.cfg );
    ledOpen( &led2.ctrl, &led2.cfg );

    // open timer HAL modules
    timer1.pAPI->open( &timer1.ctrl, &timer1.cfg);
    timer2.pAPI->open( &timer2.ctrl, &timer2.cfg);

     // start timer
    timer1.pAPI->start( &timer1.ctrl );
    timer2.pAPI->start( &timer2.ctrl );
```

Abb. 11 Applikation mit Hardware Abstraction Layer HAL Aufrufen

Im HAL sind drei verschiedene Strukturen definiert. Sie stehen für die Initialisierung (Configuration – cfg), die Steuerung zur Laufzeit (Control – ctrl) und die verschiedenen Funktionsaufrufe (Application Programmers Interface – API) zur Verfügung.

```
typedef struct
{
    void (*open)(timerControl_t *pCtrl, timerConfig_t *pCfg);
    void (*close)(timerControl_t *pCtrl);
    void (*start)(timerControl_t *pCtrl);
    void (*stop)(timerControl_t *pCtrl);
    void (*clear)(timerControl_t *pCtrl);
    void (*setCycleTime)(timerControl_t *pCtrl, uint16_t cycleTime);
} timerAPI_t;
```

Abb. 12 HAL Struktur mit Funktionszeigern

Die verschiedenen Strukturen, die für die Nutzung eines Moduls zuständig sind, werden in einer umgebenden Struktur zusammengefasst.

```
typedef struct
{
    timerConfig_t  cfg;
    timerControl_t ctrl;
    timerAPI_t     *pAPI;
} timerModule_t;
```

Abb. 13 HAL Struktur mit Elementen vom Typ Struktur

Das detaillierte -Knowhow steckt also im Low-Level-Treiber. Der Hardware Abstraction Layer verbirgt die Zugriffe auf diese untere Softwareschicht in Strukturen, und die Applikation muss nur noch die Art der Nutzung der Peripheriemodule festlegen.

Darunter wird zwar die Effizienz (Laufzeit, Speicherplatzbedarf) leiden, aber die Wiederverwendbarkeit, Änderbarkeit und Portierbarkeit werden deutlich verbessert. Vor allem bei komplexen Systemen spielt das keine so große Rolle mehr. Vielmehr gilt es, in möglichst kurzer Zeit ein lauffähiges System zu entwickeln. Und dazu tragen fertige Low-Level-Treiber sicherlich bei.

Zusammenfassung

Durch die saubere Trennung der Software in Subsysteme oder Schichten wird die Wiederverwendbarkeit und Austauschbarkeit der Hardware oder der Applikation deutlich verbessert. Die Nutzung fertig implementierter Low-Level-Treiber vereinfacht das Handling komplexer Bausteine. Es spart Zeit sowohl in der Entwicklungsphase als auch in der Vorbereitung bei der Einarbeitung in die Funktionalität des genutzten Bausteins.

Ein Nachteil kann die Komplexität des entstandenen Softwaresystems sein. Speicherbedarf und Laufzeiten verändern sich, deshalb muss in der Analysephase geklärt werden, was für das System wichtiger ist – Effizienz oder Wiederverwendbarkeit, Austauschbarkeit, Anpassbarkeit.

Abkürzungsverzeichnis

API – Application Programmers Interface
CMSIS – Cortex Microcontroller System Interface Standard
CMSIS-SVD – CMSIS System View Description
DAP – Debug Access Port
DSP – Digital Signal Processing
HAL – Hardware Abstraction Layer
LLD – Low Level Driver
MCAL – Microcontroller Abstraction Layer
RTOS – Real-Time Operating System

Literatur- und Quellenverzeichnis

[1] Infineon XE16x Register Definition File, 28.07.2008

[2] ARM Embedded Software Development (CMSIS)
https://developer.arm.com/embedded/cmsis

Autorin:

Renate Schultes ist Senior Trainer, Consultant und Coach für Mikrocontroller-Hardware & -Software bei der MicroConsult GmbH. Neben Begeisterung für Innovation und Leidenschaft für Embedded-Systeme verfügt sie über langjährige Projekterfahrung in Softwareentwicklung, Systems Engineering und Debugging für Mikrocontroller. Ihr fundiertes Wissen rund um Embedded-Systeme gibt sie in Trainings und aktuellen Veröffentlichungen und Vorträgen an Kunden weiter.