Embedded Software Engineering Kongress

ESE Kongress 2015

Vortragsskript:

Objektbasiert oder objektorientiert?
Moderne Low Level Treiberprogrammierung mit C/C++

Renate Schultes, MicroConsult GmbH

Die Programmiersprache C ist in der Embedded-Welt sehr weit verbreitet. Eine Vielzahl von Embedded-Systemen wurde in der Vergangenheit – und wird sicher auch noch in Zukunft – in C programmiert. Der Zugriff auf die im Mikrocontroller vorhandenen Peripheriemodule (Steuer-, Status- und Arbeitsregister) kann in unterschiedlichsten Formen durchgeführt werden. Für die modernen, zum Teil sehr komplexen Embedded-Systeme muss darauf geachtet werden, dass gut lesbarer, wiederverwendbarer, leicht erweiterbarer Code entsteht. Hier kommt als Programmier-Paradigma die „objektbasierte Programmierung“ ins Spiel. Und dann stellt sich als nächstes die Frage, ob nicht gleich „objektorientiert“ programmiert werden soll.

Wenn Sie sich jetzt fragen: „Was ist denn eigentlich ein Objekt?“, dann könnte die Antwort lauten: „Alles ist ein Objekt“. Zum Beispiel eben die Peripheriemodule eines Mikrocontrollers. In der Vergangenheit wurden die Namen für die einzelnen Register der verschiedenen Peripheriemodule mithilfe der C-Präprozessoranweisung **`#define`** in einem Headerfile so definiert, dass über diesen Namen direkt lesend und/oder schreibend auf die absolute Adresse im Mikrocontroller zugegriffen werden konnte. Bei modernen 16-Bit und 32-Bit Mikrocontrollern ist die Erstellung dieses Headerfiles durchaus eine Herausforderung Es sind tausende von Registernamen zu definieren. Das führt schnell zu Fehlern und ist auch für den Nutzer sehr unübersichtlich. Deshalb gehen Baustein- und Toolhersteller dazu über, stattdessen ein Abbild der Registerstruktur eines Peripheriemoduls (Objekt) im Baustein als C-Struktur in der Software zu realisieren.

```
#define P10_OUT     (*(volatile unsigned int*)0xF003B000u)
#define P10_OMR     (*(volatile unsigned int*)0xF003B004u)
#define P10_ID      (*(volatile unsigned int*)0xF003B008u)
#define P10_IOCR0   (*(volatile unsigned int*)0xF003B010u)
#define P10_IOCR4   (*(volatile unsigned int*)0xF003B014u)
#define P10_IOCR8   (*(volatile unsigned int*)0xF003B018u)
#define P10_IOCR12  (*(volatile unsigned int*)0xF003B01Cu)
#define P11_OUT     (*(volatile unsigned int*)0xF003B100u)
#define P11_OMR     (*(volatile unsigned int*)0xF003B104u)
#define P11_ID      (*(volatile unsigned int*)0xF003B108u)
#define P11_IOCR0   (*(volatile unsigned int*)0xF003B110u)
#define P11_IOCR4   (*(volatile unsigned int*)0xF003B114u)
#define P11_IOCR8   (*(volatile unsigned int*)0xF003B118u)
#define P11_IOCR12  (*(volatile unsigned int*)0xF003B11Cu)
```

```
#define P10_BASE    ((PORT*)(0xF003B000))
#define P11_BASE    ((PORT*)(0xF003B100))
typedef struct port
{
        volatile unsigned int OUT;
        volatile unsigned int OMR;
        volatile unsigned int ID;
        volatile unsigned int reserved0[1];
        volatile unsigned int IOCR0;
        volatile unsigned int IOCR4;
        volatile unsigned int IOCR8;
        volatile unsigned int IOCR12;
} PORT;
```

Abb. 1: C-Struktur statt vieler C-Makros

Kommt in einem Mikrocontroller ein und dasselbe Peripheriemodul mehrfach vor (zum Beispiel mehrere PORT-Module, siehe Abb. 1), so kann die Struktur wiederverwendet werden. Nur die Startadresse der Registerstruktur ist für die unterschiedlichen Module verschieden (MODULE_P10 und MODULE_P11). Über Zeiger auf den Typ der Struktur kann dann auf die verschiedenen Register im Modul zugegriffen werden.

```
PORT* pPort = P10_BASE;
// . . .
pPort->OUT = 0x0000FFFF;
```

Abb. 2: Zugriff auf ein Datenelement der Struktur über einen Zeiger

Der Zugriff auf die Register der Peripheriemodule erfolgt dann nur noch über Zugriffsfunktionen; direkte Zugriffe sollten generell verboten sein. Dies führt nicht zwangsläufig zu einer schlechteren Performance des Systems (mehr Code und/oder Laufzeit). Mithilfe des Schlüsselwortes **`inline`** können „normale Funktionen“ bei der Übersetzung durch den Compiler wie Makros übersetzt werden Das heißt es gibt keinen Funktionsaufruf und kein Return, sondern der Code der Inline-Funktion wird direkt an der Aufrufstelle eingesetzt.

Und damit zurück zum Programmier-Paradigma. Die oben beschriebene Art der Programmierung wird als „objektbasierte Programmierung“ bezeichnet.

Aus Wikipedia:
Zur besseren Verwaltung gleichartiger Objekte bedienen sich die meisten Programmiersprachen des Konzeptes der Klasse. Klassen sind Vorlagen, aus denen Instanzen genannte Objekte zur Laufzeit erzeugt werden. Im Programm werden nicht einzelne Objekte, sondern eine Klasse gleichartiger Objekte definiert. Existieren in der gewählten Programmiersprache keine Klassen oder werden diese explizit unterdrückt, so spricht man zur Unterscheidung oft auch von objekt-basierter Programmierung [1]

Von der objektbasierten Programmierung in C (Objekt abgebildet als Struktur) ist es dann nur noch ein kleiner Schritt zur objektorientierten Programmierung OOP in C++ (Objekt abgebildet als Klasse).

```c
#define P10_BASE    ((PORT*)(0xF003B000))
#define P11_BASE    ((PORT*)(0xF003B100))
typedef struct port
{
        volatile unsigned int OUT;
        volatile unsigned int OMR;
        volatile unsigned int ID;
        volatile unsigned int reserved0[1];
        volatile unsigned int IOCR0;
        volatile unsigned int IOCR4;
        volatile unsigned int IOCR8;
        volatile unsigned int IOCR12;
} PORT;

/*****************************************************
 * Port Function Prototypes
 *****************************************************/
void port_init(PORT* pBase, PInitStruct_t* pInitStruct);
void toggleLED(uint32_t Index);
```

```c
#define P10_BASE    ((PORT*)(0xF003B000))
#define P11_BASE    ((PORT*)(0xF003B100))
class Port
{
public:
    static void init(Port* pBase, volatile uint32_t* pInitStruct);

    static void toggleLED(uint32_t Index)
    {
        uint32_t uiMask = (1 << (Index + LEDCTL_LED_FIRST))
                        | (1 << (Index + LEDCTL_LED_FIRST + 16));
        ((Port*)(0xF003D300))->OMR = uiMask;
    } //  End of function toggleLED
private:
        volatile unsigned int OUT;
        volatile unsigned int OMR;
        volatile unsigned int ID;
        volatile unsigned int reserved0[1];
        volatile unsigned int IOCR0;
        volatile unsigned int IOCR4;
        volatile unsigned int IOCR8;
        volatile unsigned int IOCR12;
};
```

Abb. 3: Von der C-Struktur zur C++-Klasse

Die OOP mit C++ bietet dabei mehrere Vorteile:

- Der C++-Compiler ist das bessere statische Testtool, da strengere Regeln bezüglich der Sprachsyntax eingehalten werden müssen.
  - Weniger eigene Codierregeln, automatische Prüfung der Regeln
  - Die Datenkapselung in der Klasse wird durch die Zugriffsrechte (**`private`** und **`public`**) geregelt und vom Compiler überwacht. Auf die privaten Member einer Klasse kann nicht von außerhalb der Klasse zugegriffen werden.
  - Datenschutz, Datenkapselung

- Zusammenfassen der Datenelemente und darauf operierender Methoden innerhalb der Klasse:
  - Was logisch zusammengehört, wird in der Klasse zusammengefasst.

- Gleiche (sinnvolle) Methodennamen können in unterschiedlichen Klassen verwendet werden.
  - Aus **`port_init()`** wird **`init()`** der Klasse Port.

- Erweiterung von vorhandenen Klassen kann mithilfe der Vererbung realisiert werden. Das hat den Vorteil, dass vorhandene Klassen (und damit vorhandener und getesteter Code) nicht nachträglich verändert oder ergänzt werden muss.
  - Alles was bereits existiert wird vererbt, und durch zusätzliche Datenelemente und/oder Methoden erweitert (spezialisiert).

Wird eine Instanz der Klasse angelegt (ein Objekt erzeugt), wird Speicherplatz für die Datenelemente angelegt. Da die Peripherieregister bereits in der Hardware des Mikrocontrollers mit fest vorgegebenen Adressen existieren, wird in diesem Fall keine Instanz der Klasse erzeugt. Damit die Methoden der Klasse trotzdem aufrufbar sind, müssen sie statisch (C++ Schlüsselwort **`static`**) sein. Der Aufruf erfolgt dann über den Klassennamen in Verbindung mit dem Scope Operator (**`Port::init()`**).

Methoden, die direkt in der Klasse implementiert werden, sind implizit **`inline`**. Das heißt dass beim Aufruf der Methode der übersetzte Assemblercode direkt an der Aufrufstelle eingebettet wird (wie bei C-Makros). Dies hat den Vorteil, dass bei sehr kurzen Methoden keine zusätzliche Laufzeit für Aufruf (Rückkehradresse im Stack sichern) und Rückkehr (Rückkehradresse aus dem Stack holen) verbraucht wird. Damit lassen sich ähnlich laufzeiteffiziente Applikationen entwickeln, wie sie aus der C-Programmierung bekannt sind. Allerdings muss darauf geachtet werden, dass bei häufigen Aufrufen mehr Programmspeicher benötigt wird, da bei jedem Aufruf der komplette übersetzte Assemblercode eingesetzt wird. Deshalb sollte **`inline`** generell nur für kleine Methoden verwendet werden (zum Beispiel reine Lese- und Schreiboperationen).

Natürlich können auch Interrupt Service Routinen in C++ geschrieben werden. Bietet der C Compiler des Toolherstellers ein spezielles Schlüsselwort für die Implementierung (zum Beispiel **`__interrupt`**), so steht dieses auch im C++ Compiler zur Verfügung. Interrupt Service Routinen müssen in C++ statische (**`static`**) Methoden sein, da sie ohne Objektbezug aufgerufen werden. Da typisch jedes Peripheriemodul eine oder mehrere eigene Interruptquellen hat, sollten Interrupt Service Routinen nicht als Methoden der Klasse, die das Peripheriemodul abbildet, definiert werden. Denn dann müsste die eine Klasse für zwei oder drei (oder mehr) Peripheriemodule alle Interrupt Service Routinen immer mit enthalten, unabhängig davon, ob sie je genutzt werden.

Eine eigene Klasse nur für die Interrupt Service Routinen ist hier auf jeden Fall sinnvoller. Da C++ auch für die prozedurale Programmierung verwendet werden kann, wäre eine weitere Möglichkeit, die Interrupt Service Routinen als globale Methoden zu definieren. Im Sinne der OOP sollte das allerdings vermieden werden.

Zusammenfassung

Durch die verbesserte Typsicherheit, Syntaxvereinfachungen und die Unterstützung der objektorientierten Programmierung OOP ist C++ auch für Embedded-Applikationen die bessere Alternative zu C. C++ nur als besseres statisches Testtool zu verwenden, ist aber nur der erste mögliche Schritt für die Nutzung in Embedded-Systemen. Denn die Nutzung der Techniken und erweiterten Sprachmöglichkeiten der OOP führt zusätzlich zu besserer Wiederverwendbarkeit und leichterer Anpassbarkeit der Software. Der Ressourcenverbrauch muss sich durch die Nutzung der OOP nicht zwangsläufig verschlechtern. Deshalb sollte vor allem beim Start in ein neues Projekt eine grundsätzliche Überlegung sein: „Objektbasiert oder objektorientiert?“.

Abkürzungen

OOP – Objektorientierte Programmierung

Quellenverzeichnis

[1] https://de.wikipedia.org/wiki/Objektorientierte_Programmierung

Autorin

Renate Schultes - kaum jemand kennt die Welt der Mikrocontroller so gut wie sie. Als Trainerin, Beraterin und Autorin von Fachbüchern und Publikation zu 8-, 16- und 32-Bit Architekturen hat Renate Schultes schon vielen Entwicklern den Weg in die praktische Anwendung erleichtert. Sie hat eine große „Fangemeinde“ an Entwicklern, die sich darauf freuen, wenn sie wieder mit ihr gemeinsam in die Welt einer neuen Prozessorfamilie oder Programmiersprache eintauchen dürfen.

Kontakt

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