03/2016

Von C nach Embedded-C: Das Ziel bestimmt den Weg

Autorin: Renate Schultes

Natürlich ist Embedded-C auch C. Der Umstieg von C auf Embedded-C bedeutet aber, dass der Programmierer sich beim Einsatz an den Erfordernissen der jeweiligen Embedded-Anwendung orientieren muss. Dies sind beispielsweise Echtzeitfähigkeit, geringer Speicherbedarf oder hohe Betriebssicherheit. Gleichzeitig stellt die Wiederverwendbarkeit von Software auch in der Embedded-Welt ein wichtiges Qualitätsmerkmal dar. Die richtige Anwendung der C-Schlüsselwörter spielt dabei eine bedeutende Rolle.

Wer ein Haus bauen möchte, hat eine Vorstellung auf welchem Grund es gebaut wird, welche Fläche oder Raumeinteilung es haben sollte und vieles mehr. Aus den Vorgaben entsteht mit Hilfe des Architekten der Bauplan. Und danach wird – in einer vorher festgelegten Reihenfolge der Gewerke – das Haus gebaut.

Wer ein Embedded-System bestehend aus Hardware und Software entwickelt, braucht auch einen Plan. Für die Hardware besteht an dieser Notwendigkeit kein Zweifel, aber wie sieht es mit der nicht sichtbaren oder greifbaren Software aus?

Ja, auch das muss geplant werden. Ein Vorgehensmodell (Prozessmodell) legt fest, WER WANN WOFÜR zuständig ist.

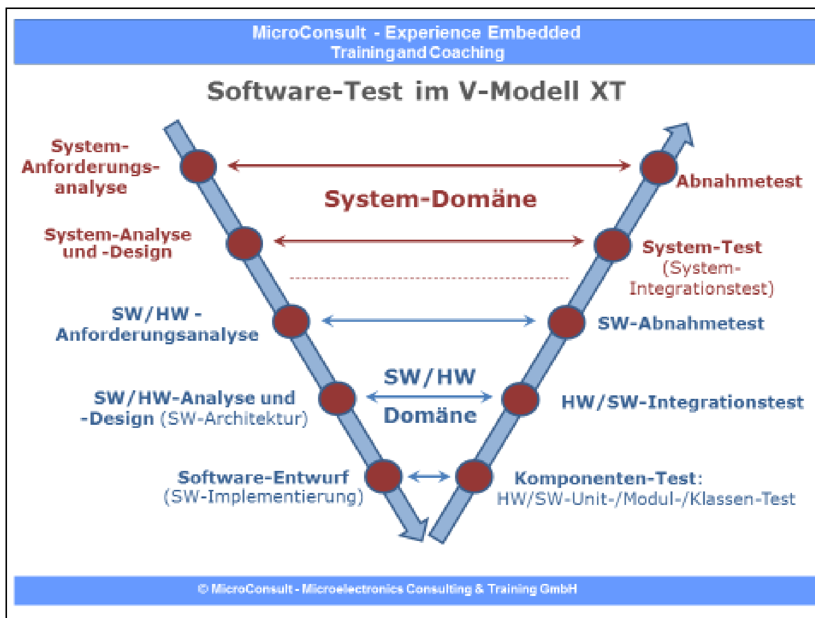

Bild 1: Das V-Modell wird im Umfeld von Embedded-Entwicklungen häufig als Vorgehensmodell eingesetzt. Der linke Schenkel zeigt von oben nach unten die Arbeitsschritte der Entwicklungsphase, der rechte Schenkel die verschiedenen Tests.

Der Weg, den die Softwareentwickler von der Anforderungsanalyse über Design und Implementierung bis hin zum Test gehen, wird durch die Aufgabenstellung vorgegeben. Und was hat das alles mit der Programmiersprache C oder Embedded-C zu tun?

Nun, die Anforderungen müssen vom Programmierer richtig umgesetzt werden, und dabei helfen unter anderem eine Reihe von C-Schlüsselwörtern.

Beginnen wir mit dem ersten Schritt, dem Festlegen der Anforderungen an die Software.

Anforderungen an die Software

In der Software-Analysephase müssen die unterschiedlichen Anforderungen an die Software - die funktionalen und die nicht-funktionalen Qualitätsmerkmale - festgelegt werden.

Qualitätsmerkmal	Qualitäts-Teilmerkmal
Funktionalität	Angemessenheit, Richtigkeit, Interoperabilität, Ordnungsmäßigkeit, *Sicherheit*
Zuverlässigkeit	Reife, Fehlertoleranz, Wiederherstellbarkeit
Benutzbarkeit	Verständlichkeit, Erlernbarkeit, Bedienbarkeit
Effizienz	Zeitverhalten, Verbrauchsverhalten
Änderbarkeit	Analysierbarkeit, Modifizierbarkeit, Stabilität, Prüfbarkeit
Übertragbarkeit	Anpassbarkeit, Installierbarkeit, Konformität, Austauschbarkeit
Sicherheit	Betriebssicherheit, Gefahr für den Anwender vom System ausgehend Zugriffssicherheit, Gefahr für das System vom Anwender ausgehend

Bild 2: Tabelle der Software-Qualitätsmerkmale nach ISO 9126

Funktionale Anforderungen sind vergleichsweise einfach zu definieren: WAS soll/muss die Software leisten? Die Herausforderung sind bei Embedded-Systemen häufig die nicht-funktionalen Anforderungen. Sie beziehen sich oft auf die Randbedingungen (Vorschriften, Normen, Zeitvorgaben, etc.) eines Embedded-Systems.

Ein Beispiel dafür ist das Software-Qualitätsmerkmal Effizienz. Über wie viel Programm- und Datenspeicher verfügt das System? Muss bei der Programmierung „gespart“ werden, weil sehr wenig Speicher zur Verfügung steht? Kann es Laufzeitprobleme geben, weil die Echtzeitanforderungen nur unter bestimmten Bedingungen einhaltbar sind? Stellen die Spannungsversorgung bzw. Stromaufnahme oder die Wärmeerzeugung ein Problem dar, das durch die Art der Programmierung beeinflusst werden kann oder muss?

Wenn eine oder mehrere dieser Fragen mit JA beantwortet werden, dann MUSS das in den Anforderungen stehen. Der Programmierer setzt dann die Anforderungen mit geeigneten Maßnahmen um, und der Tester muss sie testen.

Diese Anpassungen an die spezifischen Erfordernisse der Systemoptimierung hat nicht nur Einfluss auf die Art der Programmierung sondern auch auf die Wiederverwendbarkeit der Software. Funktionsaufruf und -rückkehr kosten zum Beispiel zusätzlich Laufzeit, also gilt es, diese zu vermeiden. Darunter leidet die Modularisierung, die ihrerseits jedoch für gut wiederverwendbare Software wichtig ist.

Auch die Zuordnung von Programmcode in den physikalisch vorhandenen Programmspeicher kann eine wichtige Rolle spielen. Verfügt der Mikrocontroller über Cache, muss schon im C-Quellcode dafür gesorgt werden, dass im Linker/Locator die Zuordnung der Funktionen zum cacheable und nicht-cacheable Speicher erfolgen kann. Gibt es unterschiedliche Zugriffszeiten auf verschiedene Arten von Programm- oder Datenspeicher, muss ebenfalls schon im C-Quellcode vorgesehen werden, dass im Linker/Locator die Zuordnung richtig durchgeführt werden kann.

Hat eine CPU nichts zu tun, außer zum Beispiel auf ein Ereignis zu warten, kann sie entweder in einer Warteschleife weiter arbeiten, oder abgeschaltet werden. Nichts tun in einer Warteschleife kostet Strom, und Abschalten reduziert die Stromaufnahme erheblich, doch das Zuschalten kostet natürlich Zeit. Was ist wichtiger – Strom sparen oder die sehr schnelle Reaktion auf ein Ereignis? Die Antwort auf diese Frage hat bedeutende Auswirkungen darauf, wie die Software implementiert wird.

Eine ebenfalls sehr wichtige Anforderung an viele Embedded-Systeme stellt die Betriebssicherheit (Safety) dar. Schauen wir uns als erstes an, was der Programmierer dazu beitragen kann.

Betriebssicherheit – Safety

Die Maßnahmen, um eine komplexe Steuerung für sicherheitskritische Aufgaben ausreichend betriebssicher zu machen, sind sehr umfangreich und betreffen das Konzept (Anforderungsanalyse, Design) sowie den gesamten Entwicklungs- und Testprozess.

Die Programmiersprache C lässt unterschiedliche Syntaxformen zu, Schlüsselwörter können zum Teil ganz weggelassen werden, Datentypen sind nicht typsicher, Variable und Zeiger müssen vor ihrer Nutzung nicht initialisiert werden. All das kann zu Missverständnissen, Fehlinterpretation oder sogar schwerwiegenden Fehlern führen (beispielsweise nicht initialisierte Zeiger). Eines von vielen Beispielen dafür ist die Nutzung eines booleschen Ausdrucks in verschiedenen C-Kontrollstrukturen (if, while, ...).

```
int32_t i = 5;
static int32_t a;   // Variable a ist nicht initialisiert

if (a = i)
{
    a = STARTVALUE;
}
```

Bild 3: Die if-Anweisung prüft einen booleschen Ausdruck auf TRUE oder FALSE, hier wird aber eine Zuweisung (Variable a wird mit dem Inhalt von Variable i beschrieben) ausgeführt und anschließend das Ergebnis der Zuweisung (ein ganzzahliger 32-Bit Wert) auf TRUE/FALSE geprüft; außerdem fehlt hier der else-Zweig.

Solche unsicheren Syntaxformen sind unbedingt zu vermeiden, da sie leicht zu Folgefehlern führen können. Hier empfiehlt sich der Einsatz von Programmierregeln. Fertige Programmier-Regelwerke für C-Programmierer gibt es zum Beispiel in Form der MISRA[1] C-Regeln.

Bekanntermaßen sind Regeln jedoch nur gut, wenn sie auch eingehalten werden. Deshalb ist die Einhaltung der Programmierregeln zu überwachen, z.B. durch unterschiedliche Arten von Tests. Eine Möglichkeit sind Reviews, eine andere, sehr effiziente (und unbestechliche), sind statische Testtools, wie sie von verschiedenen Toolherstellern angeboten werden. Teilweise sind sie schon in den C-Compiler integriert.

Die Anwendung von Regelwerken, Standards und Normen hilft auf der einen Seite, qualitativ bessere Softwaresysteme zu bauen, führt aber andererseits zu einem Mehraufwand in allen Entwicklungsphasen und Beeinträchtigungen der Effizienz. Das führt uns zur Frage, wie bedeutend das Software-Qualitätsmerkmal Effizienz tatsächlich ist.

Effizienz

Die Embedded-C-Compiler oder der dazugehörige C-Präprozessor verfügen über besondere Schlüsselwörter (zum Beispiel __attributes__ bzw. #pragma), um die gezielte Zuordnung zu bestimmten Adressbereichen durch den Linker/Locator zu unterstützen.

Mit dem Compiler-Schlüsselwort inline können C-Funktionen wie Makros direkt an der Aufrufstelle eingefügt werden. Bei sehr kleinen Funktionen mit wenigen Assemblerbefehlen kann das viel Laufzeit sparen, kostet bei häufigen Aufrufen aber auch mehr Speicherplatz.

Ein weiteres Problem stellt bei vielen Mikrocontrollern die Nutzung von Floating-Point-Arithmetik dar, weil die einfache ALU (arithmetisch-logische Einheit der CPU) keine geeignete Logik für die schnelle Abarbeitung von Floating-Point-Operationen bietet. Die Verwendung von C-Bibliotheksfunktionen kostet sehr viel Laufzeit. Hat ein Mikrocontroller eine Floating-Point-Unit FPU in Hardware implementiert, beherrscht diese oft nur einfache Genauigkeit. In diesem Fall sollten alle Operationen nur in einfacher Genauigkeit ausgeführt werden, damit die Hardware-FPU auch genutzt wird. Diese Einschränkungen müssen für Floating-Point-Variable, Floating-Point-Konstanten und Floating-Point-Berechnungen angegeben werden.

Neben dem Steuerparameter des Compilers für das Abschalten der Berechnungen mit doppelter Genauigkeit gibt es in der Programmiersprache C auch noch die Möglichkeit, Floating-Point-Konstanten (standardmäßig doppelte Genauigkeit) mit dem Buchstaben f für float (einfache Genauigkeit) zu kennzeichnen.

```
// Single-Precision Floating Point
static float f = 2.3f;

f = f + 1.2f;
```

Bild 4: Beispielcode für die Kennzeichnung von Floating-Point-Konstanten in C

Aber nicht nur der Programmierer kann etwas zur Effizienz beitragen. Moderne C-Compiler optimieren den zu erzeugenden Maschinencode. Typisch gibt es mehrere Optimierungsstufen (-O0/-O1/-O2/-O3), die vom Zusammenfassen von Konstanten-Ausdrücken bis hin zum Umstellen der Assembler-Befehlsreihenfolge unterschiedliche Stufen von Code-Optimierung durchführen. Viele Embedded-C-Compiler bieten darüber hinaus Speed- und Size-Optimierung an. Wie gut C-Compiler wirklich optimieren, hängt allerdings maßgeblich vom Compiler-Hersteller ab.

C-Compiler versuchen unter anderem auch, Zugriffe auf Speicherzellen oder Steuer-/Status-/Arbeitsregister von Peripheriemodulen zu optimieren.

Ein Beispiel:

Der Mikrocontroller enthält eine serielle Schnittstelle. In einer Software-Warteschleife soll gewartet werden, bis über diese serielle Schnittstelle ein Zeichen vollständig empfangen wurde. Der C-Compiler übersetzt diese Warteschleife in der Regel als folgende Befehlsfolge:

1. Lies einmal das Statusregister der seriellen Schnittstelle in ein CPU-Arbeitsregister ein.
2. Frage im Arbeitsregister der CPU, ob sich das entsprechende Statusbit verändert.

Das Statusbit ändert sich allerdings nur im Statusregister, nicht aber im CPU-Arbeitsregister. Die Schleife wird aufgrund der Compiler-Optimierung zu einer unendlichen Warteschleife.
Abhilfe schafft hier das C-Schlüsselwort volatile. Es sorgt dafür, dass immer die Original-Speicherzelle gelesen/beschrieben wird. Sobald sich das Statusbit ändert, wird der veränderte Wert im Arbeitsregister der CPU zu sehen sein. Die Schleife wird verlassen.

```
static volatile int32_t status = STARTVALUE;

void waitForEvent(void)
{
    while (status == STARTVALUE)    // wait for status change
        ;
}
```

Bild 6: Die Variable status wird asynchron, zum Beispiel in einer Interrupt-Service-Routine, verändert und muss deshalb volatile sein. So ist sichergestellt, dass bei jedem Durchlauf der Warteschleife die Originalspeicherzelle gelesen wird.

Mit Hilfe eines weiteren C-Schlüsselwortes, nämlich inline (das im Übrigen von C++ übernommen wurde), kann der C-Programmierer einen Beitrag zur Code- und/oder Laufzeit-Optimierung leisten. Bei C-Funktionen mit kompaktem Code kann inline zu wesentlichen Laufzeitverkürzungen führen. Der Compiler fügt dann an der Aufrufstelle der Funktion direkt den kompletten Funktionscode ein. Das geht natürlich nur, wenn der Compiler den C-Quellcode kennt. Deshalb müssen inline-Funktionen im Headerfile implementiert und als inline gekennzeichnet werden. Allerdings hängt die Umsetzung des Schlüsselwortes inline von der Compiler-Optimierung ab. Infolgedessen hat es der Programmierer wieder nicht selbst in der Hand, wie optimiert wird.

Viele Compiler-Hersteller haben das Problem erkannt und mittlerweile Abhilfe geschaffen. Sie bieten zusätzliche Optimierungseinstellungen (zum Beispiel in Form von Pragma-Direktiven) an, mit deren Hilfe der Compiler gezwungen wird, inline-Funktionen unabhängig von der eingestellten Optimierung immer als solche zu übersetzen.

inline-Funktionen können auch als C-Makros programmiert werden. Der Vorteil besteht darin, dass sich der Aufruf der Funktionen nicht vom Aufruf „richtiger“ Funktionen unterscheidet. Deshalb kann jederzeit ohne großen Aufwand zwischen inline und nicht-inline umgestellt werden.
Aber Vorsicht! Das Debuggen ändert sich durch inline-Funktionen. Es gibt keinen Breakpoint in der Funktion, da der Funktionskörper ja nicht mehr nur an einer Stelle, sondern an vielen Stellen existiert. Der Breakpoint muss demnach an der Aufrufstelle gesetzt werden.

```
#ifndef TESTEMBC_H_
#define TESTEMBC_H_

#include <stdint.h>

typedef struct
{
    uint32_t state;
} STATE_t;

inline void setState(STATE_t* const pSTATE_t, uint32_t value)
{
    if (value > 0 && value <= 100)
    {
        pSTATE_t->state = value;
    }
}

inline int32_t getState(const STATE_t* const pSTATE_t)
{
    return pSTATE_t->state;
}

#endif /* TESTEMBC_H_ */
```

Bild 7: Einfache Schreiboperationen wie setState() oder Leseoperationen wie getState() bieten sich als `inline`-Funktionen an.

```c
#include "TestEmbC.h"

int main(void)
{
    uint32_t temp;
    static STATE_t locObj;
    setState(&locObj, 20);
    temp = getState(&locObj);

    return temp;
}
```

Bild 8: inline-Funktionen werden wie „normale“ Funktionen aufgerufen; nur am übersetzten Code ist erkennbar, ob es sich um inline-Funktionen handelt.

Sichtbarkeit und Gültigkeit von Variablen

C-Programmierer neigen dazu, viele – oder alle – Variable als globale Variable anzulegen. So sind sie für alle Funktionen von außen sichtbar. Verfügt ein Softwaresystem über viele solcher globalen Variablen, stellt jede Änderung oder Erweiterung ein Risiko dar. Es ist bei komplexer Software schwer zu durchschauen, wer wann und warum auf welche Variable lesend oder schreibend zugreift? Der einzelne Entwickler verliert dabei den Überblick, da er selten alle Systemzusammenhänge kennt.

Deshalb muss der Programmierer sorgfältig bedenken, was global sein muss bzw. lokal bleiben sollte. Die Programmiersprache C stellt dafür das Schlüsselwort static zur Verfügung. Damit können Variable von „für jeden sichtbar“ auf „nur im Modul sichtbar“ eingeschränkt werden, oder lokale Variable innerhalb von Funktionen können von dynamisch auf statisch umgestellt werden.

Dynamisch bedeutet, dass die Variable im Stack (oder einem Arbeitsregister der CPU) abgelegt wird und bei Beenden des Blocks wieder freigegeben wird. Statisch bedeutet, dass die Variable eine feste RAM-Adresse bekommt und die Speicherzelle über das Blockende hinaus belegt bleibt.

Lokale statische Variable haben Vor- und Nachteile. Der Vorteil ist, dass der Inhalt erhalten bleibt und beim nächsten Eintritt in den Block wiederverwendet werden kann. Der Nachteil ist, dass RAM-Speicher belegt wird (und bleibt), auch wenn auf den Inhalt nicht zugegriffen werden kann.

```c
int32_t globalVar = STARTVALUE;  // Application-global visibility,
                                 // global validity
static int32_t localVar = 0;     // Module-local visibility,
                                 // global validity

int main(void)
{
   int32_t i;          // Function-local visibility, local validity
   static int32_t a;   // Function-local visibility, global validity

   return 0;
}
```

Bild 5: Globale und lokale Variable können mit static deklariert werden, um die Sichtbarkeit und Gültigkeit zu verändern.

Auch Variable, die von mehreren Funktionen genutzt werden, können lokal bleiben, wenn die Funktionen, die auf sie zugreifen, in dasselbe Modul gepackt werden. Falls das nicht möglich ist, muss über globale Variable (oder über Zeiger) nachgedacht werden.

Eine Programmierregel sollte festlegen, dass globale Variable gut dokumentiert werden müssen:

- Welchen Zweck hat die Variable, und wer stellt sie zur Verfügung?
- Gibt es Einschränkungen (z.B. Wertebereich, etc.)?
- Wer darf nur lesend, wer nur schreibend oder wer lesend und schreibend zugreifen?

Wenn der Aufwand für die Dokumentation hoch ist, wird der Programmierer gerne auf die übertriebene Nutzung verzichten.

Auch C-Funktionen können static, also modul-lokal sein. Sie lassen sich dann nur innerhalb eines Moduls aufrufen. Dies sollte für reine Hilfsfunktionen festgelegt werden, die nicht von außerhalb des Moduls genutzt werden. Wären sie global, verführt das den einen oder anderen Programmierer dazu, sie doch zu nutzen. Änderungen an diesen Funktionen können jedoch schwer absehbare Auswirkungen auf das System nach sich ziehen.

Das führt uns zum Aspekt der Änderbarkeit. Er tritt häufig gepaart mit Wiederverwendbarkeit und Anpassbarkeit auf, weil diese Software-Qualitätsmerkmale oft durch die gleichen Maßnahmen bei der Programmierung angestrebt werden.

Wiederverwendbarkeit, Anpassbarkeit, Änderbarkeit

In der heutigen schnelllebigen Zeit gilt auch für Software: Für die Erstellung steht sehr begrenzt Zeit zur Verfügung, für Änderungen und Ergänzungen noch weniger. Also sind gute Wiederverwendbarkeit sowie leichte Anpassbarkeit und Änderbarkeit ein MUSS, um im Wettbewerb bestehen zu können.

Allerdings kostet Wiederverwendbarkeit Speicherplatz und Laufzeit. Bei modernen 32-Bit-Multicore-Mikrocontrollern steht relativ viel Speicherplatz und Rechenleistung zur Verfügung, bei 8-Bit-Singlecore-Mikrocontrollern ist beides nur sehr begrenzt verfügbar.

Für gute Wiederverwendbarkeit und Erweiterbarkeit eignet sich der objektorientierte Ansatz sehr gut. Aber wir sprechen hier über prozedurale Programmierung in C, nicht über objektorientiert und C++.

Die Lösung könnte lauten: Wir kombinieren prozedurale und objektorientierte Ansätze und arbeiten objektbasiert. Das heißt praktisch, dass Strukturen und Zeiger in den Vordergrund rücken. Dabei werden mehrere Variable, auch unterschiedlichen Typs, die einen logischen Zusammenhang haben, zu einer Struktur zusammengefasst. Der Zugriff erfolgt ausschließlich über dafür zur Verfügung gestellte Funktionen. Und diese wiederum verwenden Zeiger, um auf die im Speicher erzeugten Objekte zuzugreifen.

```
typedef struct stm
{
    volatile unsigned int CLC;        // Clock Control Register
    volatile unsigned int RESERVED0;  // reserved (0x04)
    volatile unsigned int ID;         // ASCLIN Identification Register
    volatile unsigned int RESERVED1;  // reserved (0x0C)
    volatile unsigned int TIM0;        // STM[31:0]
    volatile unsigned int TIM1;        // STM[35:4]
    volatile unsigned int TIM2;        // STM[39:8]
    volatile unsigned int TIM3;        // STM[43:12]
    volatile unsigned int TIM4;        // STM[47:16]
    volatile unsigned int TIM5;        // STM[51:20]
    volatile unsigned int TIM6;        // STM[63:32]
    volatile unsigned int CAP;         // STM[63:32]
    volatile unsigned int CMP0;        // Compare Register 0
    volatile unsigned int CMP1;        // Compare Register 1
    volatile unsigned int CMCON;       // Compare Match Control Register
    volatile unsigned int ICR_;        // Flags Clear Register
    volatile unsigned int ISCR;        // Flags Enable Register
    volatile unsigned int RESERVED2[3];  // reserved (0x44 - 0x4C)
    volatile unsigned int TIM0SV;     // Timer 0 Register Second View
    volatile unsigned int CAPSV;      // Capture Register Second View
    volatile unsigned int RESERVED3[2];   // reserved (0x58, 0x5C)
    volatile unsigned int RESERVED4[9*4]; // reserved (0x60 ... 0xD0)
    volatile unsigned int RESERVED5[2];   // reserved (0xE0, 0xE4)
    volatile unsigned int OCS;        // OCDS Control and Status Register
    volatile unsigned int KRSTCLR;    // Reset Status Clear Register 12
    volatile unsigned int KRST1;      // Kernel Reset Control Register 1
    volatile unsigned int KRST0;      // Kernel Reset Control Register 0
    volatile unsigned int ACCEN1;     // ASCLIN Access Enable Register 1
    volatile unsigned int ACCEN0;     // ASCLIN Access Enable Register 0
} STM_t;
```

Bild 9: Definition einer Struktur für die Register des System Timers (STM) des Infineon 32-Bit-Multicore-Mikrocontrollers AURIX TC277

Gibt es mehrere Peripheriemodule vom gleichen Typ, muss hier nur die Anfangsadresse des jeweiligen Moduls in einem Zeiger vom Typ der Struktur verwendet werden, um auf alle Register eines Moduls zugreifen zu können. Die Struktur lässt sich ohne Änderung für weitere Module dieses Typs wiederverwenden. Wenn es eine neue Bausteinvariante gibt, kann die Struktur häufig ebenfalls ohne Änderung, unter Umständen mit nur kleinen Anpassungen, wiederverwendet werden.

```
void stmInit(STM_t* const pSTM, const STMInitStruct_t* const pInit)
{
        pSTM->CMP0    = pInit->compVal0;
        pSTM->CMP1    = pInit->compVal1;
        pSTM->CMCON   = pInit->cmcon0 | pInit->cmcon1;
        pSTM->ICR_    = pInit->cmp0Icr | pInit->cmp1Icr;

        if(pInit->isr0 != 0)
        {
                SRC_STM0SR0 = pInit->isr0;
        }

        if(pInit->isr1 != 0)
        {
                SRC_STM0SR1 = pInit->isr1;
        }
}
```

Bild 10: Initialisierungsfunktion für den im Headerfile definierten Datentyp STM_t

Die Zugriffsfunktionen haben als Übergabeparameter konstante Zeiger vom Typ der Struktur. Für die Initialisierung gibt es zusätzlich eine Struktur, die die Initialisierungswerte enthält. So kann eine Initialisierungsfunktion für unterschiedliche Betriebsarten genutzt werden.

Bei den für die Parameterübergabe genutzten Zeigern ist das C-Schlüsselwort const sehr wichtig. Der Zeiger auf die Register-Struktur soll nicht verändert werden können – auch nicht unbeabsichtigt. Deshalb sollte er selbst immer konstant (also read-only) sein. Der Quellspeicher selbst, aus dem die Initialisierungswerte gelesen werden, sollte ebenfalls als konstant (read-only) gekennzeichnet werden. So kann der Compiler sicherstellen, dass dort, wo nur gelesen werden darf, auch wirklich nur gelesen wird. Ohne das Schlüsselwort const führt ein versehentlich eingebauter schreibender Zugriff oder eine Zeigerveränderung unter Umständen zu aufwändiger Fehlersuche.

Fazit

Das Ziel bestimmt den Weg! Das richtig funktionierende (sichere) Embedded-System ist das Ziel. Der Weg dahin muss geplant werden, und wer auf dem richtigen Weg bleibt, erreicht sein geplantes Ziel einfacher und schneller. Embedded-Systeme unterscheiden sich voneinander und von klassischen IT System durch ihre speziellen Qualitätsmerkmale. Standards, Normen und insbesondere die nichtfunktionalen Anforderungen stellen für den Embedded-C-Programmierer eine besondere Herausforderung dar. Wer ihnen gewachsen sein will, sollte die richtige Anwendung der C-Schlüsselwörter, die passenden Optimierungseinstellungen und die korrekte Compilersteuerung beherrschen.

Vor der Implementierung ist eine sorgfältige Planung auf Basis der Anforderungen und im Sinne einer änderungsfreundlichen Software-Architektur empfehlenswert. Nachträgliche Änderungen und Erweiterungen dürfen keine Gefahr für das Gesamtsystem darstellen. Programmierregeln sind sinnvolle Hilfen bei der Erstellung von Quellcode; ihre Einhaltung muss strikt gefordert und natürlich auch überprüft werden.

Wenn der Tester schließlich die geforderte Qualität des Systems ohne großen Aufwand für Korrekturen und weitere Tests bestätigen kann, hat sich der Mehraufwand im Vorfeld mehrfach bezahlt gemacht.

Autorin

Renate Schultes ist als Mitarbeiterin der Firma MicroConsult GmbH unter anderem für die Trainings zu den Programmiersprachen C, C++ und C# zuständig. Sie hat mehr als 30 Jahre Erfahrung mit der Softwareentwicklung für Embedded-Systeme. Sie berät und schult unter anderem bedeutende Zulieferer und OEMs der Automobiltechnik.

Literatur

[1] MISRA C ist ein Programmierstandard aus der Automobilindustrie für die Sprache C
http://www.misra.org.uk/

MicroConsult GmbH - Experience Embedded:

MicroConsult ist Ihr Partner für Embedded Systems Engineering - professionelle Schulungen, Beratung und Projektunterstützung.
www.microconsult.com