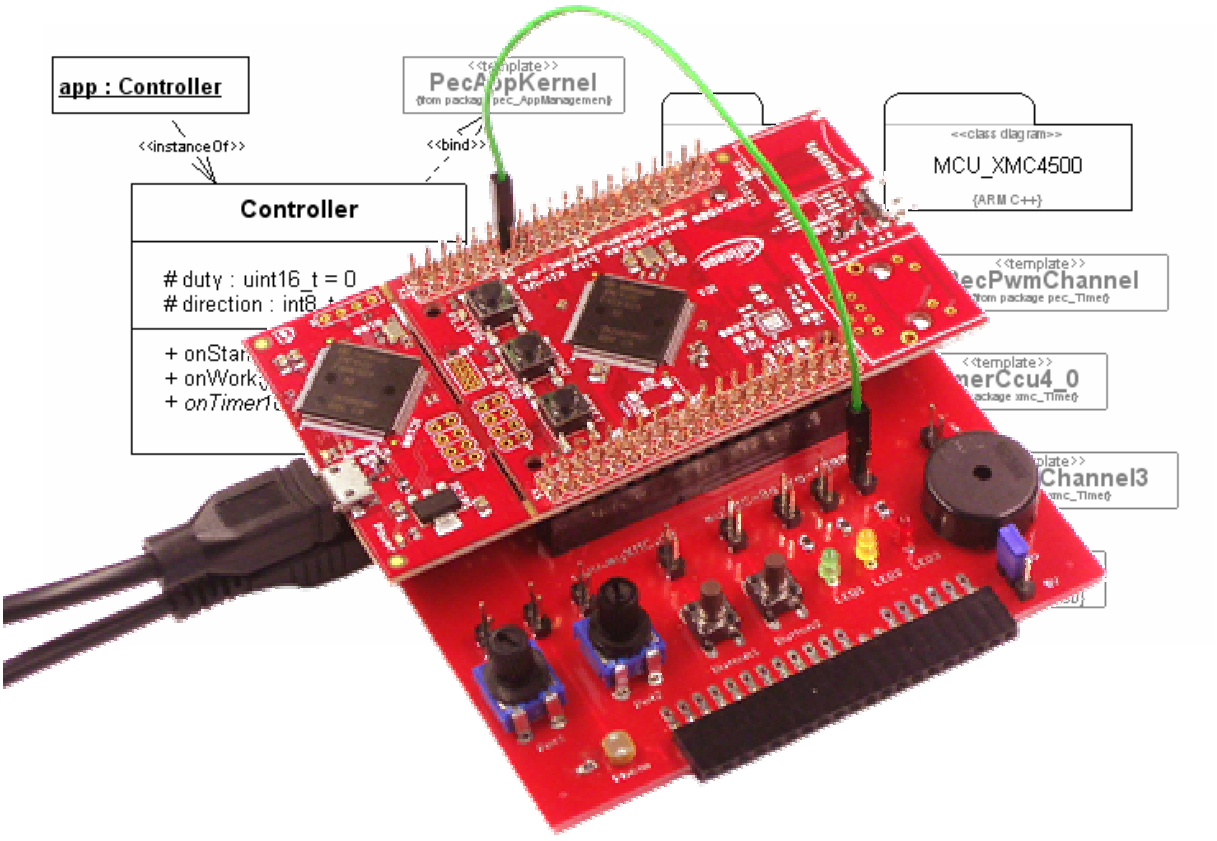
Sven Löbmann
Toralf Riedel
Alexander Huwaldt

myXMC Lehrbuch

Ein Lehrbuch für die praxisorientierte Einführung
in die Programmierung von XMC-Mikrocontrollern



Leseprobe

3. Auflage: April 2016

www.laser-co.de
www.SiSy.de
www.myXMC.de
www.myMCU.de
support@laser-co.de
Tel: ++49 (0) 3585 470 222
Fax: ++49 (0) 3585 470 233

Inhalt

Vorwort

Dieses Buch wendet sich an Leser, die bereits über Kenntnisse einer beliebigen Programmiersprache verfügen und sich auch mit der objektorientierten Programmierung von XMC Mikrocontrollern von Infineon beschäftigen möchten.
Hier soll sich speziell mit ausgewählten Aspekten für den **einfachen Einstieg in die objektorientierte Programmierung von XMC-Mikrocontrollern** auseinandergesetzt werden.
Infineon kombiniert in der XMC-Mikrocontroller-Familie seine langjährige Erfahrung mit den Vorteilen des ARM Cortex-M. Dabei arbeitet Infineon intensiv an den Herausforderungen, die durch die zunehmende Softwarekomplexität auf die Entwickler zukommen. Eine Entwicklungsrichtung ist DAVE. *DAVE* ist ein in *Eclipse* eingebetteter Codegenerator, der es dem Entwickler erleichtern soll, anwendungsfallspezifische Applikationen und Bibliotheken aus Low-Level-Drivern und Middleware zu generieren. Eine zweite Entwicklungsrichtung zur Beherrschung der zunehmenden Softwarekomplexität ist der objektorientierte Ansatz und die UML.

Die Objektorientierung ist ursprünglich angetreten, das Programmieren einfacher zu machen. Praktisch erscheinen jedoch objektorientierte Sprachen für viele eher als Hürde, nicht als Erleichterung. Das muss aber nicht so sein. Assembler und C sind nicht wirklich einfacher als C++. Bilden Sie sich zu folgenden Codeausschnitten selbst Ihre Meinung.

```
// "klassische" Schreibweise ////////////////////////////////
PORT0->IOCR4 &= ~0x0000f800UL;
PORT0->IOCR4 |= 0xC0U << 8;
PORT0->OMR = 0x00200020UL;

// objektorientierte Schreibweise ////////////////////////////
Led led;
led.config(PORT0,5);
led.on();
```

Ich glaube dieses kleine Beispiel zeigt deutlich, dass eine objektorientierte Vorgehensweise und Programmierung zu wesentlich verständlicherem Code führen kann. Man könnte auch sagen, dass man das Ziel der Objektorientierung erreicht hat, wenn sich der Programmcode wie Klartext lesen lässt. Wir wollen uns von Bedenken und inneren Hürden nicht abhalten lassen, objektorientiert zu arbeiten.

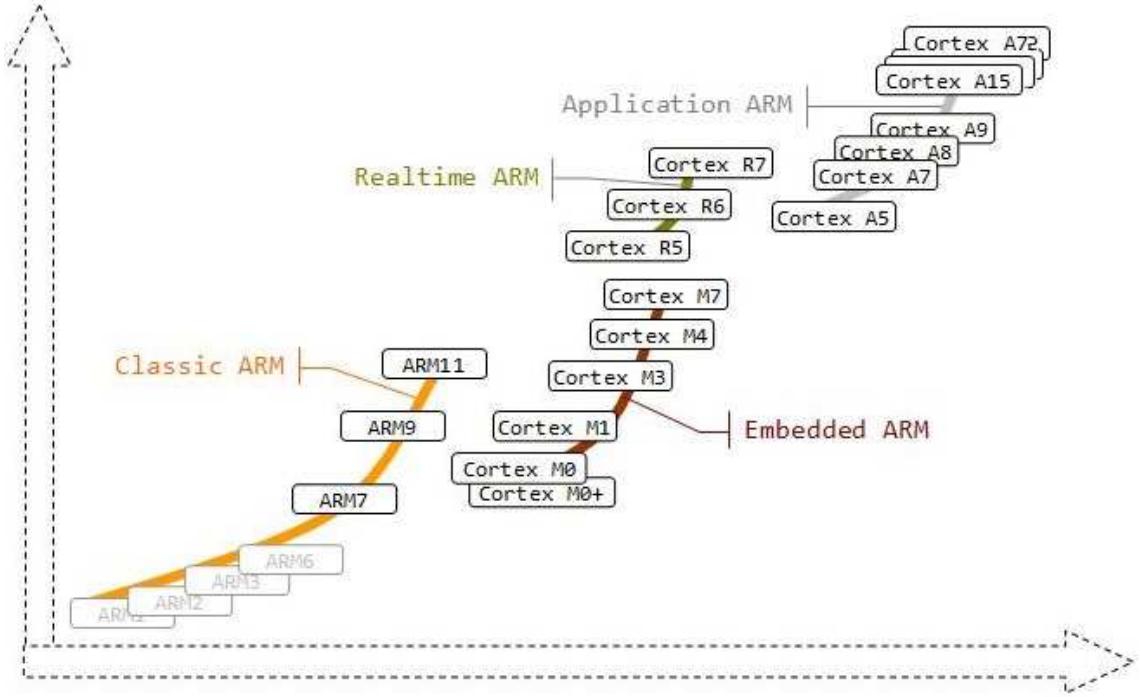
„Jede neue Sprache ist wie ein offenes Fenster,
das einen neuen Ausblick auf die Welt eröffnet
und die Lebensauffassung weitet.“
Frank Harris (1856-1931), amerikanischer Schriftsteller

Weitere Informationen und Beispiele finden Sie im begleitenden Online-Tutorial zu diesem Lehrbuch unter www.myXMC.de.
Wir wünschen Ihnen viel Erfolg beim Studium.

1 Einführung

1.1 ARM-Architektur

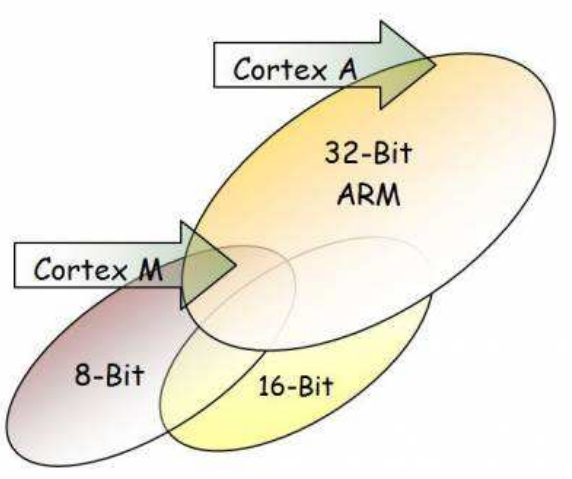
Die standardisierte 32-Bit ARM-Architektur der Firma ARM Ltd. aus Cambridge bildet die Basis für jeden ARM-Prozessor. Im Laufe der Jahre hat sich die ursprüngliche ARM-Architektur rasant entwickelt. Die neueste Version des ARM bildet die ARMv8 Architektur. Diese zeigt schon deutlich in Richtung 64-Bit Architekturen. Vielleicht werden Sie sich jetzt fragen, wozu Sie solche Leistung brauchen. Aber selbst Hobbyprojekte wie Quadcopter oder eine Hexapod können recht schnell an die Leistungsgrenze eines 8/16- Biter stoßen.



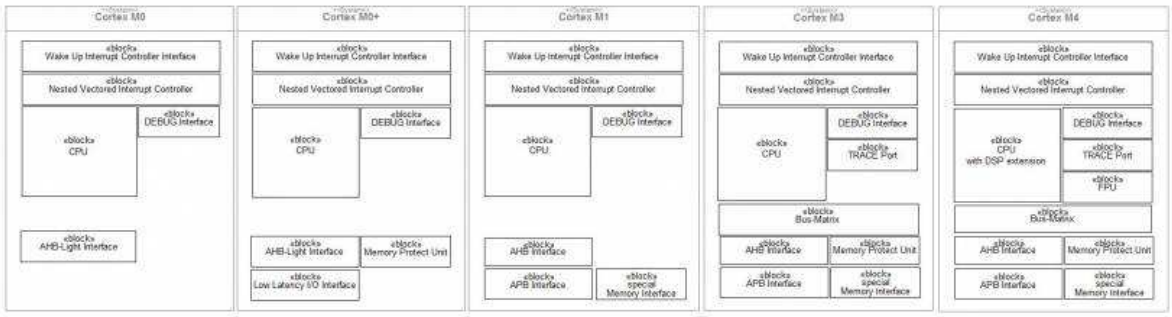
ARM Controller sind dem Wesen nach RISC (Reduced Instruction Set Computer) und unterstützen die Realisierung einer breiten Palette von Anwendungen. Inzwischen gilt ARM als führende Architektur in vielen Marktsegmenten und kann getrost als Industriestandard bezeichnet werden. Den Erfolg der ARM-Architektur kann man sehr gut an den aktuellen Trends bei Smart-Phone, Tablet und Co. ablesen. Mehr als 40 Lizenznehmer bieten in ihrem Portfolio ARM-basierende Controller an. Vor allem Effizienz, hohe Leistung, niedriger Stromverbrauch und geringe Kosten sind wichtige Attribute der ARM-Architektur.

1.1.1 Cortex-M

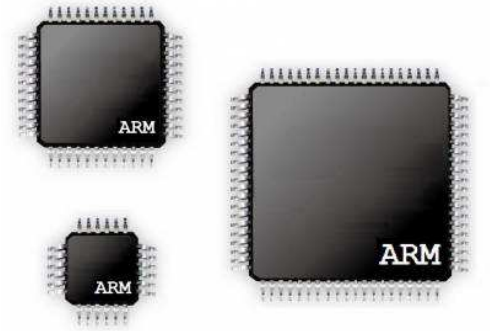
Die Cortex-M Prozessoren zielen direkt auf das Marktsegment der mittleren eingebetteten Systeme. Dieses wird bisher von 8-Bit und 16-Bit Controllern dominiert. Dabei scheut ARM auch nicht den direkten Vergleich mit der kleineren Konkurrenz bezüglich Effizienz, Verbrauch und Preis. Die Botschaft heißt: 32-Bit Leistung muss nicht hungriger nach Ressourcen sein, als ein 8-Bit Controller und ist auch nicht teurer. Natürlich vergleicht man sich besonders gern mit den guten alten



8051ern und zeigt voller Stolz seine Überlegenheit bei 32-Bit Multiplikationen. Bei aller Vorsicht bezüglich der Werbeargumente kann es jedoch als sicher gelten, dass der Cortex-M einige Marktverschiebungen in Gang gesetzt hat.
Die folgende (mit Sicherheit nicht vollständige) Darstellung soll die Skalierung der Cortex-M Familie verdeutlichen.



Der Formfaktor dieser 32-Bit Controller lässt sich durchaus mit den größeren Mega und X-Mega Controllern der AVR-Familie von Atmel oder anderer 8/16 Bit Controller vergleichen. Für den blutigen Anfänger unter den Bastlern könnte jedoch die SMD-Bauweise eine nicht unerhebliche Einstiegshürde darstellen.



1.1.2 CMSIS und Peripherie Treiber

CMSIS - **C**ortex **M**icrocontroller **S**oftware **I**nterface **S**tandard, ist ein herstellerunabhängiges Hardware Abstraction Layer für die Cortex-M Prozessoren und umfasst folgende Standards:

- CMSIS-CORE (Prozessor und Standardperipherie),
- CMSIS-DSP (DSP Bibliothek mit über 60 Funktionen),
- CMSIS-RTOS API (API für Echtzeitbetriebssysteme),
- CMSIS-SVD (Systembeschreibung in XML),

Damit sind grundlegende Funktionen aller ARM Controller kompatibel und lassen sich herstellerunabhängig und portabel verwenden. In der später vorgestellten Entwicklungsumgebung steht Ihnen eine umfangreiche Hilfe zum CMSIS zur Verfügung.

.
.
.
.

1.1.3 Low Level Peripherie Treiber

Es handelt sich hier um ein komplettes Firmware-Paket, bestehend aus Gerätetreiber für alle Standard-Peripheriegeräte der XMC 32-Bit-Flash-Mikrocontroller-Familie. Das Paket enthält eine Sammlung von Routinen, Datenstrukturen und Makros sowie deren Beschreibungen und eine Reihe von Beispielen für jedes Peripheriegerät.
Die Firmware-Bibliothek ermöglicht im Anwenderprogramm die Verwendung jedes Gerätes ohne die speziellen Einzelheiten der Register und Bitkombinationen zu kennen. Es spart viel Zeit, die sonst bei der Codierung anhand des Datenblattes aufgewendet werden muss.

Die XMC Peripherie Bibliothek umfasst 3 Abstraktionsebenen und beinhaltet:

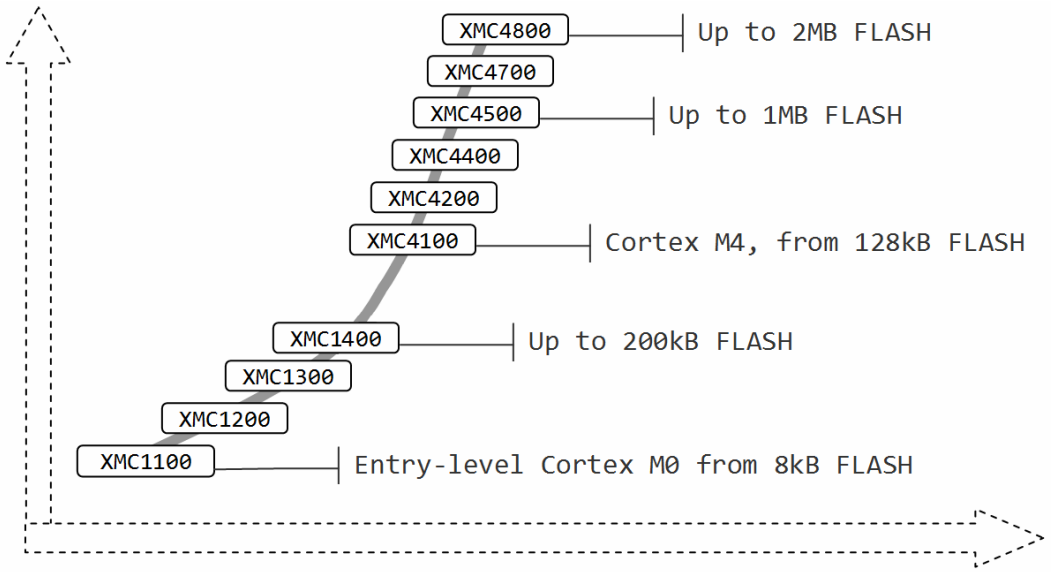
1. Ein vollständiges Register Adress-Mapping mit allen Bits, Bit-Feldern und Registern, in C deklariert.
2. Eine Sammlung von Routinen und Datenstrukturen für alle peripheren Funktionen, als einheitliche API.
3. Eine Reihe von Beispielen für alle gängigen Peripheriegeräte.

Während der Installation der im Abschnitt 1.3 vorgestellten Entwicklungsumgebung SiSy XMC werden die Bibliotheken für das CMSIS und die Peripherie-Treiber gleich mit installiert.

1.2 XMC Hardware

Die Firma Infineon bietet in ihrem breiten Produktspektrum ebenfalls Mikrocontroller auf der Basis der ARM Cortex-M Architektur an. Dabei lassen sich derzeit in der XMC Familie zwei Grundrichtungen erkennen:

- kleine Steuerungen, XMC1000-Serie, Cortex-M0, bisher 8/16-Bit Domäne
- High-Performance Systeme mit der XMC4000-Serie, Cortex-M4, 32-Bit Domäne



Infineon bietet, wie jeder Hersteller, für verschiedene Anwendungsfälle Referenzhardware zum Kennenlernen und Testen an. Beispiele für solche XMC-Evaluierungsbords sind:

- XMC1100 Boot Kit, Entry Level Applications
- XMC4500 Relax Kit , Mainstream Applications
- XMC4000 Application Kit, High Performance Applications

Alle weiteren Ausführungen in diesem Lehrbuch beziehen sich auf das *XMC4500 Relax Lite Kit* und *XMC4500 Relax Kit* von Infineon.

1.2.1 XMC4500 Relax Lite Kit

Das XMC4500 ist zwar nicht eines der neuesten, aber eines der preiswertesten und leistungsfähigsten Evaluierungsboards von Infineon. Es ermöglicht dem Anwender, besonders die Hochleistungs-Eigenschaften des Cortex-M4 zu erkunden und trotzdem Anwendungen einfach zu entwickeln. Mit dem im nächsten Abschnitt vorgestellten Erweiterungsboard „myXMC-Board-4500“ verfügen der Anfänger und der Umsteiger über alles, was für den schnellen Einstieg in die XMC-Programmierung, aber auch für anspruchsvolle Anwendungen, erforderlich ist.

Eigenschaften:

- XMC4500 Mikrocontroller (basierend auf ARM Cortex-M4)
- Spannungsversorgung über USB mit Micro USB-Kabel
- Spannungsversorgung regelbar von 5 V bis 3,3 V
- abnehmbarer on-board Debugger
- 4 LEDs
  - 1 Power-LED
  - 1 Debugg-LED
  - 2 LEDs, vom Anwender frei verfügbar
- 3 Taster
  - 1 Reset-Taster
  - 2 Taster, vom Anwender frei verfügbar
- Schnittstellen
  - 4x SPI-Master
  - 3x I²C
  - 3x I²S
  - 3x UART
  - 2x CAN
  - 17x ADC (12 Bit)
  - 2x DAC
  - 31x PMW auf 2 Pin-Reihen 2×20

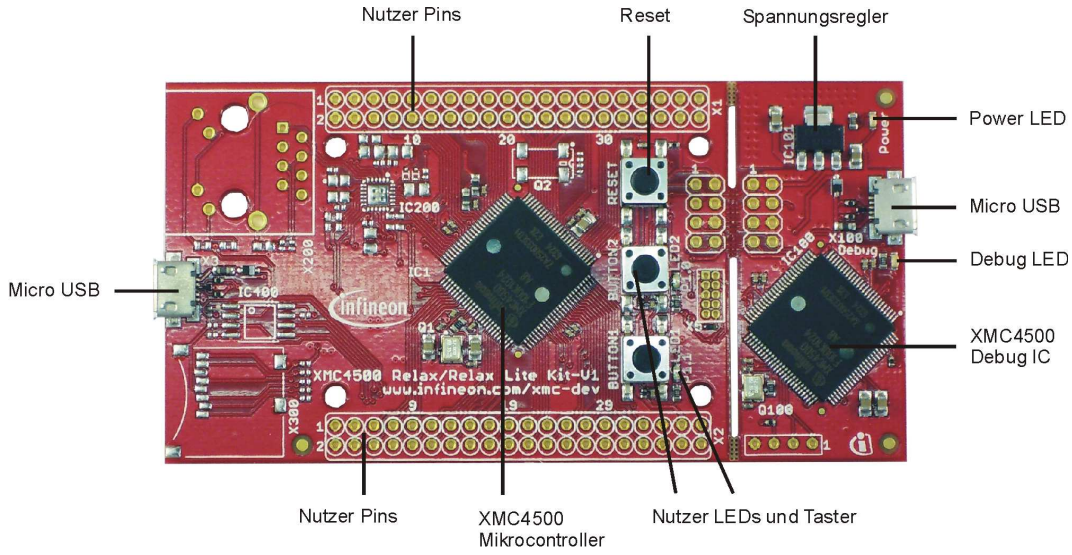


Abbildung: XMC4500 Relax Lite Kit

Die Bestückung mit simplen Eingabe- und Ausgabegeräten ist auf dem Board mit zwei Tastern und zwei frei verfügbaren LEDs doch eher spartanisch gehalten. In diesem Punkt bringt das Erweiterungsboard genügend Abhilfe. Hervorzuheben ist der abtrennbare J-LINK Programmer von Segger. Mit diesem können über den

nachrüstbaren SWD-Pfostenstecker (Serial Wire Debugging) andere XMC programmiert und debuggt werden.

.
.
.
.

1.2.2 myXMC-Board-4500

Das myXMC-Board-4500 fungiert als Add-On und ist eine ideale Ergänzung zum Board „XMC4500 Relax (Lite) Kit“. Sie erweitern mit diesem Add-On in einfacher Art und Weise die Möglichkeiten Ihres XMC4500 Relax (Lite) Kit. Zusätzliche digitale und analoge Ein- und Ausgabegeräte sowie die Möglichkeit einer optionalen USB-USART Bridge für die Kommunikation mit dem PC, komplettieren Ihre Experimentier- und Lernplattform. Desweiteren verfügt dieses Add-On über eine Schnittstelle für weitere myAVR Add-Ons.

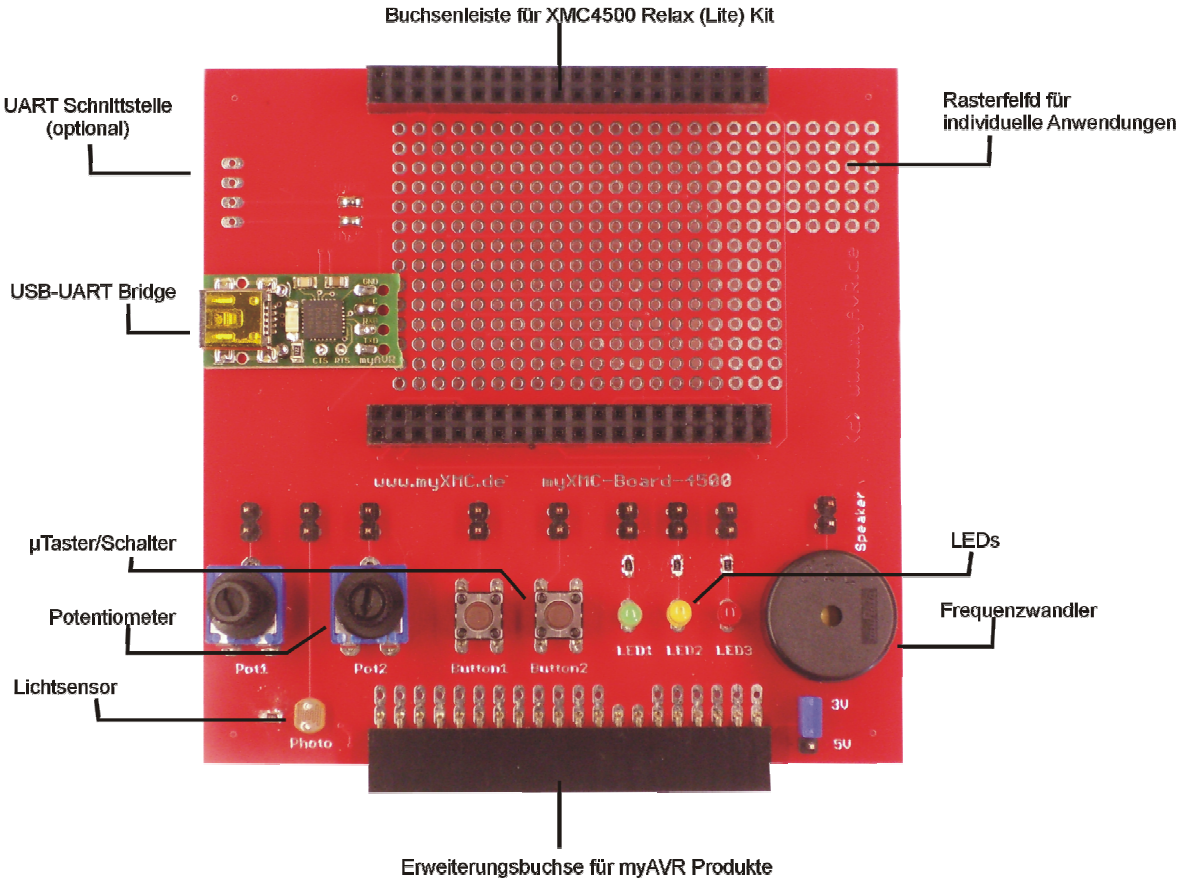


Abbildung: myXMC-Board-4500

Das myXMC-Board-4500 ist besonders darauf ausgelegt, Kennern der myAVR-Produkte und der 8-Bit AVR-Controller, den Umstieg und Anfängern den Einstieg in die Programmierung von 32-Bit ARM-Mikrocontrollern zu erleichtern.

Das myXMC-Board-4500 verfügt über einige typische, von der myAVR-Serie bekannte Ein- und Ausgabegeräte, wie zum Beispiel Potentiometer, Schalter, Frequenzwandler und LEDs. Ebenfalls ist auf dem Board ein analoger Lichtsensor zur Verwendung unterschiedlicher Helligkeitsgrade installiert. Der Formfaktor orientiert sich an den bewährten didaktischen Prinzipien der myAVR Lernsysteme.

Eigenschaften:

- Schnittstelle für XMC4500 Relax (Lite) Kit
- Schnittstelle für myAVR Produkte
- typische Ein- und Ausgabegeräte
  (2 Taster, 3 LEDs, 1 Speaker, 2 Potentiometer, 1 analoger Fotosensor)
- myUSBtoUART (USB-UART Bridge)
- Raster für flexible Anwendung (2,54 mm)
- UART Schnittstelle optional

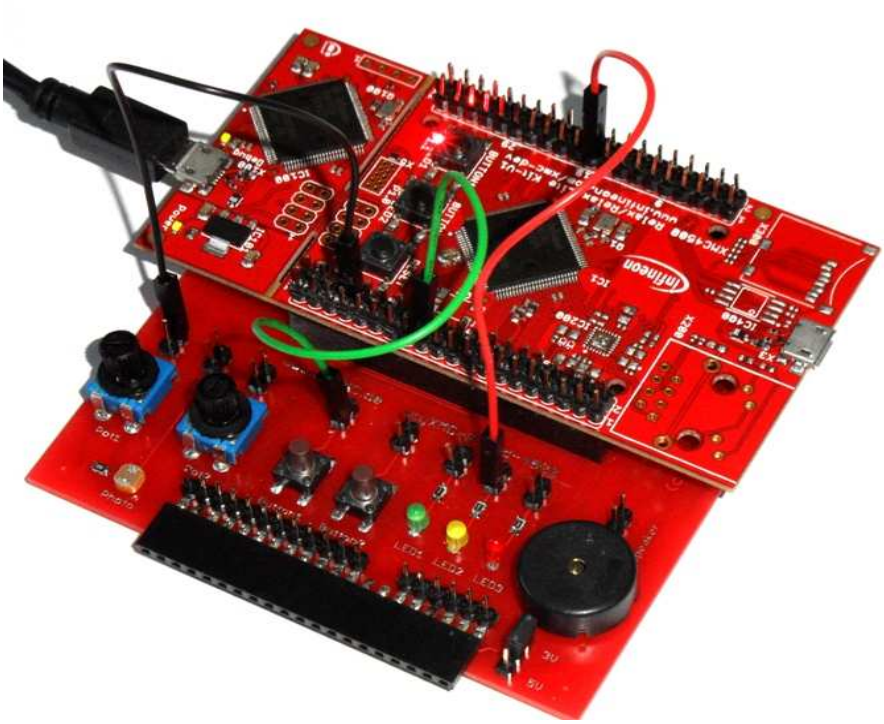


Abbildung: XMC4500 Relax Lite Kit in Kombination mit dem Erweitungsboard myXMC-Board-4500

1.3 Entwicklungsumgebung SiSy XMC

Die Basis des hier vorgestellten C++ für die XMC Mikrocontroller ist eine Portierung des GNU C/C++-Compilers. Damit liegt theoretisch schon mal ein Werkzeug zur Programmierung von ARM-Controllern sowohl in C als auch in C++ vor. Die Unterstützung objektorientierter Programmierung ist jedoch im Embedded-Bereich, selbst bei den 32-Bitern, noch nicht flächendeckend verbreitet. Es ist zwar in den verfügbaren Werkzeugen prinzipiell möglich in C++ zu programmieren, wird jedoch mehr schlecht als recht von den Entwicklungsumgebungen unterstützt. Da sich dieses Lehrbuch vor allem auch an Einsteiger wendet, soll auf eine Entwicklungsumgebung zurückgegriffen werden, die es dem Entwickler einfach macht seinen XMC in C++ und noch besser in UML zu programmieren.

1.3.1 Grundaufbau des Entwicklungswerkzeuges

Schauen wir uns als Nächstes kurz in der Entwicklungsumgebung SiSy um. An dieser Stelle wird die Handhabung nur kurz beschrieben. Bei den ausführlichen Beispielen in diesem Lehrbuch wird auf Besonderheiten explizit eingegangen. Eine ausführliche Beschreibung zur Handhabung von SiSy finden Sie im „SiSy Benutzerhandbuch“ und in der online-Hilfe von SiSy.

SiSy ist, wie bereits erwähnt, ein allgemeines Entwicklungswerkzeug, mit dem man von der Konzeption eines Systems bis zur Realisierung die verschiedensten Arbeitsschritte unterstützen kann. Für die Eingabe von Programmcode mit oder ohne Modellen bzw. Diagrammen bietet SiSy als Basiskomponente einen Zeileneditor mit Syntaxfarben und Hilfefunktionen an. Modelle werden als Diagramme erstellt bzw. abgebildet.

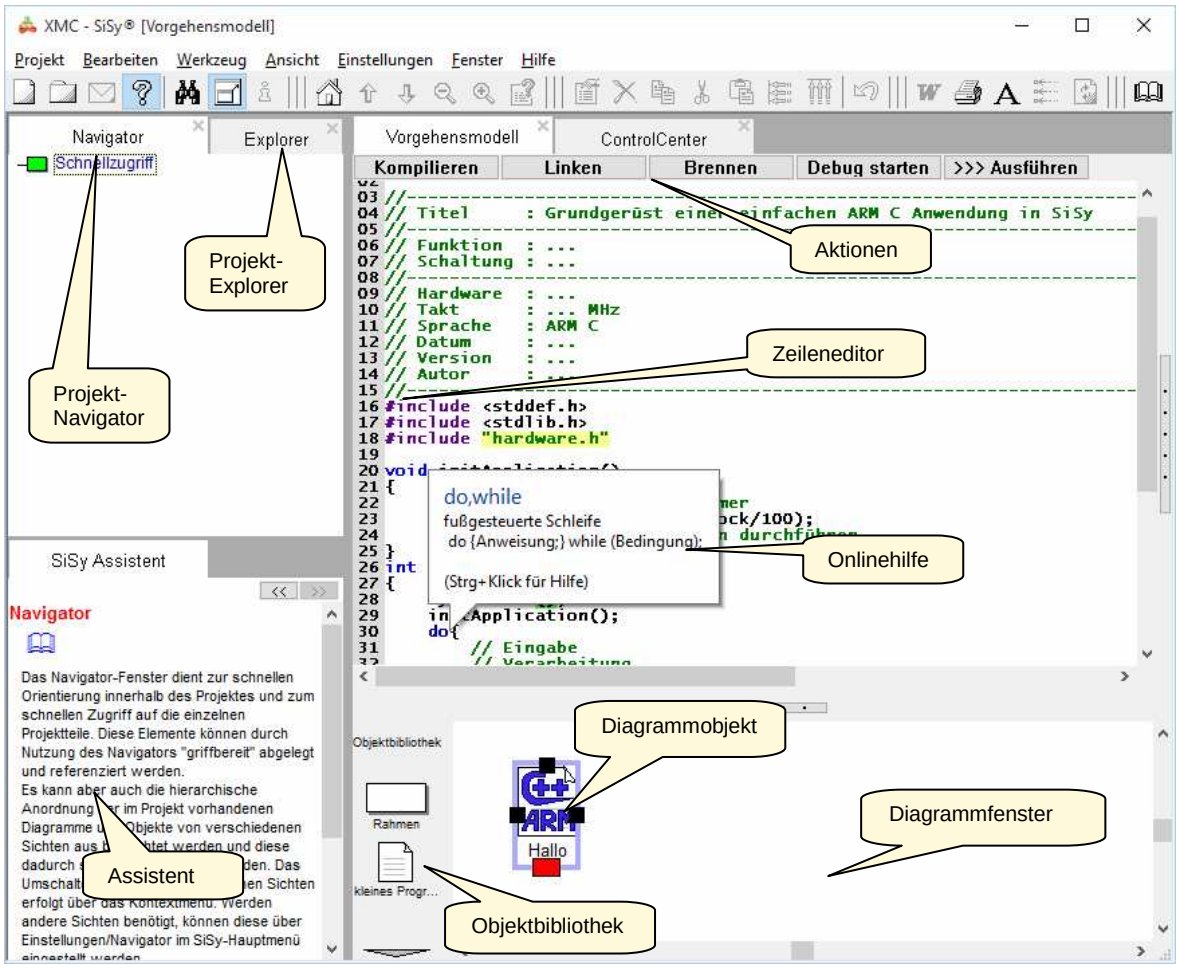


Abbildung: Bildschirmaufbau der Entwicklungsumgebung SiSy

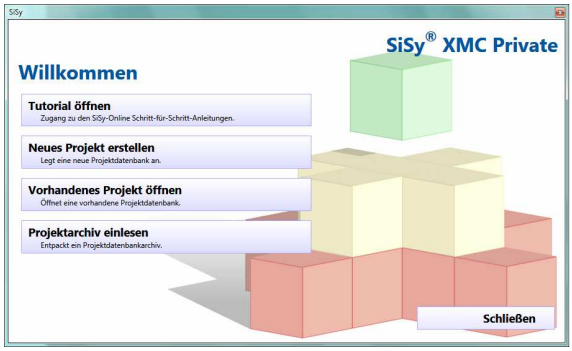
Beim Kompilieren, Linken oder auch Brennen öffnet sich ein Ausgabefenster und zeigt Protokollausgaben der Aktionen an. Wenn die Hardware ordnungsgemäß angeschlossen, von der Software erkannt und das Programm erfolgreich übersetzt sowie auf den Programmspeicher des Mikrocontrollers übertragen wurde, muss die letzte Ausschrift in Abhängigkeit der Konfiguration folgenden bzw. ähnlichen Inhalt haben:

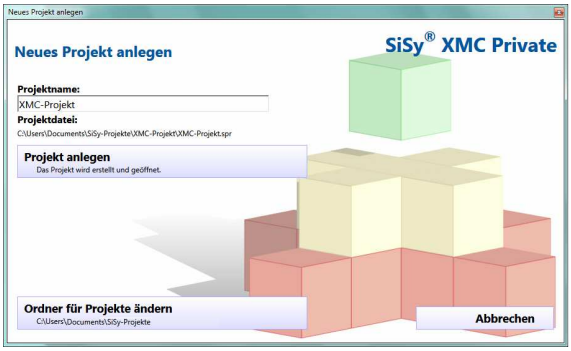
```
vorbereiten ...
brennen ...
benutze: mySmartUSB MK2 an COM2 mit ATmega8
USB-Treiber installiert, aktiv (V ), Port: COM2
Prozessor: ATmega8
schreibe 50 Bytes in Flash-Memory ...
... erfolgreich (0.32 s)
OK
```

Abbildung: ProgTool Ausgabefenster mit „Brenn“ - Protokoll

1.3.2 Grundstruktur einer XMC Anwendung

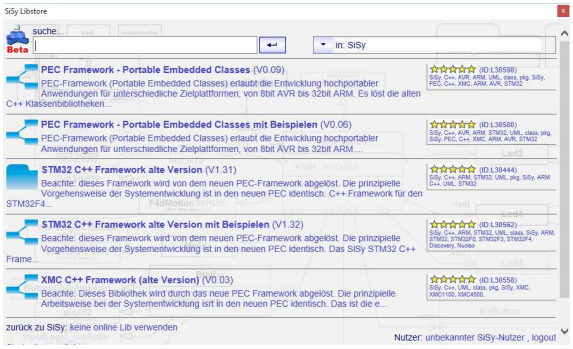
Die erste praktische Übung soll darin bestehen, dass ein XMC-Projekt angelegt und ein einfaches Programmgerüst erstellt wird. Danach schauen wir uns den Quellcode etwas näher an, übersetzen diesen und übertragen ihn in den Programmspeicher des XMC. Dazu muss SiSy XMC gestartet werden und die Experimentierhardware angeschlossen sein. Legen Sie ein *neues Projekt* mit dem Namen „XMC_Projekt“ an.





Abbildungen: Willkommensbildschirm in SiSy XMC und neues Projekt in SiSy XMC anlegen

Wählen Sie das ARM Vorgehensmodell aus. Damit sind alle wichtigen Einstellungen für das Projekt und die darin enthaltenen Übungen als Default-Werte gesetzt. Nach Auswahl des Vorgehensmodells öffnet SiSy LibStore und bietet vorhandene Vorlagen für die weitere Arbeit an.



Abbildungen: Vorgehensmodell auswählen; Anzeige von Vorlagen im SiSy LibStore

Wir brauchen für die ersten Schritte noch keine UML Bibliotheken. Damit können wir „zurück zu SiSy: keine online Lib verwenden“ aktivieren. Sie erhalten somit ein leeres Projekt.
Die typische Aufteilung der SiSy-Oberfläche besteht aus Navigator, Explorer, Assistent, Diagrammfenster und Editor. Die Aufteilung zwischen Diagrammfenster und Editor können Sie sich je nach Bedarf anpassen.

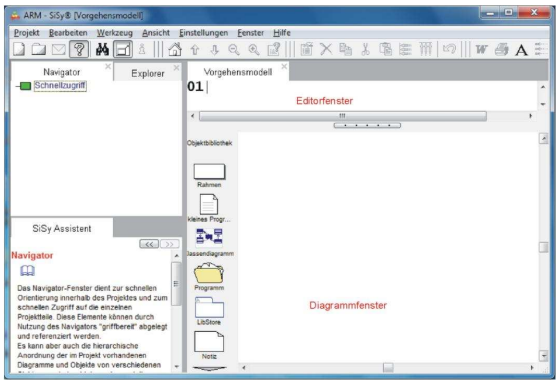
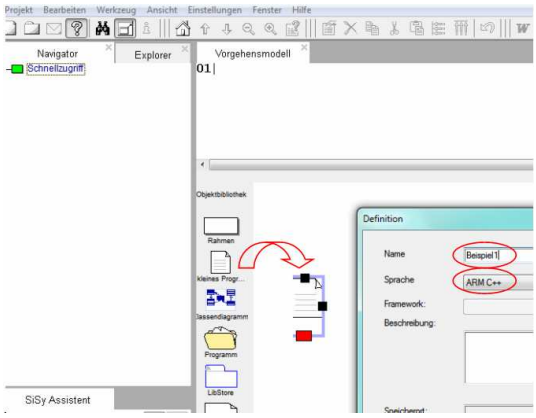


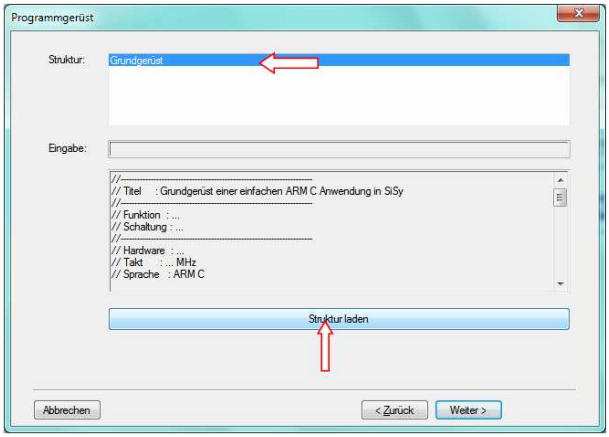
Abbildung: Bildschirmaufteilung in SiSy

Legen Sie Ihr erstes *kleines Programm* an, indem Sie das entsprechende Objekt aus der Objektbibliothek per *Drag&Drop* in das Diagrammfenster ziehen. In dem sich öffnenden Dialogfenster geben Sie dem Programm den Namen *Beispiel1* und überprüfen, ob die Zielsprache auf *ARM C++* eingestellt ist.
Im nächsten Schritt wird die Hardware ausgewählt. Wir benutzen ein Entwicklerboard von Infineon, das XMC4500 Relax (Lite) Kit, und den Programmer „J-Link“.



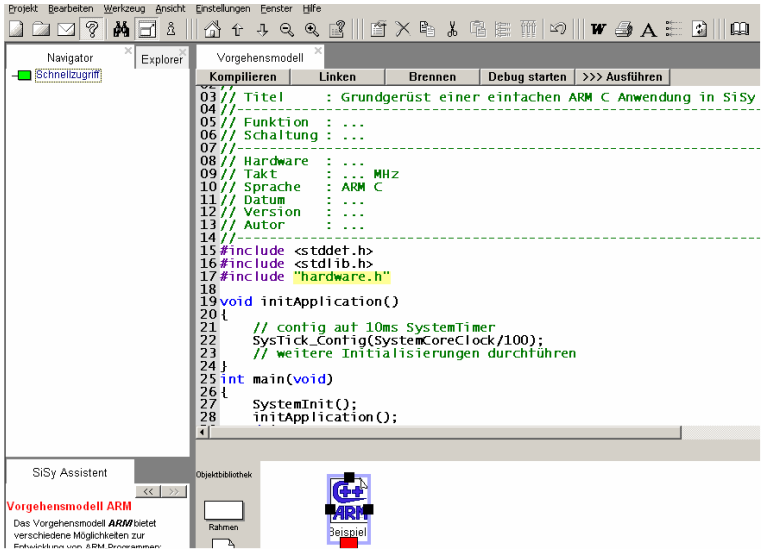
Abbildungen: Objekt in das Diagramm ziehen und Hardware auswählen

Bevor wir uns dem Stress aussetzen fast 40 Zeilen Programmcode abzutippen, benutzen wir lieber eines der Features von SiSy, die Programmgerüste.
Selektieren Sie das Grundgerüst für ein ARM C++ Programm und laden die Struktur über die Schaltfläche „Struktur laden“. Aber Achtung, bitte nicht mehrfach ausführen. SiSy fügt die aus-gewählten Programmstrukturen jeweils an das Ende des bestehenden Quellcodes an.



Das nächste Dialogfeld mit Code-Wizzard überspringen Sie und wählen die Schaltfläche „Fertig stellen“.

Sie gelangen wieder in das Diagrammfenster von SiSy; im Editorfenster wird der geladene Quellcode angezeigt.



.
.
.
.

Schauen wir uns den geladenen Quellcode etwas genauer an. Dieser lässt sich in mehrere Bereiche unterteilen. Zum einen ist da der Programmkopf mit Dokumentation und Deklarationen. Hier werden unter anderem die Deklarationen, zum Beispiel die hardwarespezifischen Registernamen und die Funktionsdeklarationen des CMSIS sowie der Peripherietreiber für den XMC4500, aus externen Dateien in den Programmcode eingefügt (*#include).* Die *stddef* und *stdlib* sind exemplarisch eingefügte C-Standardbibliotheken.

```
//-------------------------------------------------------------
// Titel     : Grundgerüst einfache ARM C Anwendung in SiSy
//-------------------------------------------------------------
// Funktion  : ...
// Schaltung : ...
//-------------------------------------------------------------
// Hardware  : ...
// Takt      : ... MHz
// Sprache   : ARM C++
// Datum     : ...
// Version   : ...
// Autor     : ...
//-------------------------------------------------------------
#include <stddef.h>
#include <stdlib.h>
#include "hardware.h"
```

Die Dokumentation sollte immer gewissenhaft ausgefüllt werden. Vor allem die Beschreibungen von Funktion und Hardware sind sehr wichtig. Das richtige Programm zur falschen Schaltung oder umgekehrt kann verheerende Folgen haben. Es folgt der Definitionsteil. Hier finden sich globale Variablen oder eben Unterpro-

gramme, besser gesagt Funktionen. Diese müssen vor dem ersten Benutzen deklariert sein.
Das bedeutet in unserem Fall, dass die Funktion initApplication noch vor dem Hauptprogramm der Funktion *main* steht. Besonders der C-Neuling beachte den Funktionskopf, in dem Fall *mit ohne* Typ und Parametern sowie den Funktionskörper, begrenzt durch die geschweiften Klammern.

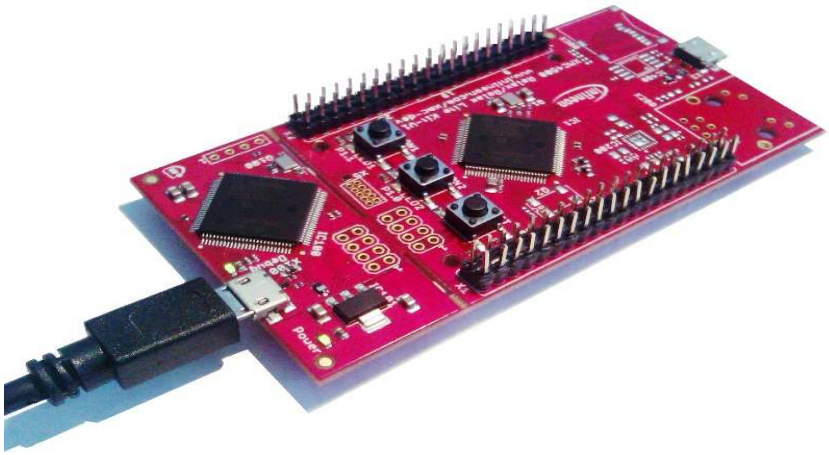
```
void initApplication()
{
    SysTick_Config(SystemCoreClock/100);
    // weitere Initialisierungen durchführen
}
```

Als vorgegebenen Funktionsaufruf finden wir dort die Initialisierung des SysTick-Timers. Dieser liefert uns schon mal ein regelmäßiges Timer-Ereignis. In den Übungen werden wir dies recht schnell benötigen. An dieser Stelle können noch weitere Funktionen eingefügt werden.
Es folgt jetzt das Hauptprogramm. Es ist durch das Schlüsselwort mai*n* gekennzeichnet. Auch hier sehen wir wieder die Begrenzung des Funktionskörpers durch die geschweiften Klammern. Innerhalb des Hauptprogramms findet sich zuerst die Initialisierungssequenz. Dabei sollte als erstes die Funktion SystemInit aufgerufen werden. Diese ist im Treiberfundus enthalten und übernimmt die Grundinitialisierungen des XMC-Kerns. Die Funktion ist quell-codeoffen und kann bei Bedarf durch den Entwickler für ein Projekt angepasst werden. Als Einsteiger nehmen wir diese, wie sie vorgefertigt ist. Danach initialisieren wir die Peripherie. Das erfolgt durch Aufruf der bereits besprochenen Funktion *initApplication*.

```
int main(void)
{
    SystemInit();
    initApplication();
    do{
        // Eingabe
        // Verarbeitung
        // Ausgabe
    } while (true);
    return 0;
}
```

Zum Schluss folgen die Interrupt Service Routinen und Ereignishandler. Da diese nicht explizit zum Beispiel aus der main aufgerufen werden, sondern in der Regel an für unser Anwendungsprogramm quasi externe Hardware-Ereignisse gebunden sind und automatisch auslösen können, stehen sie hinter dem Hauptprogramm als Letztes.

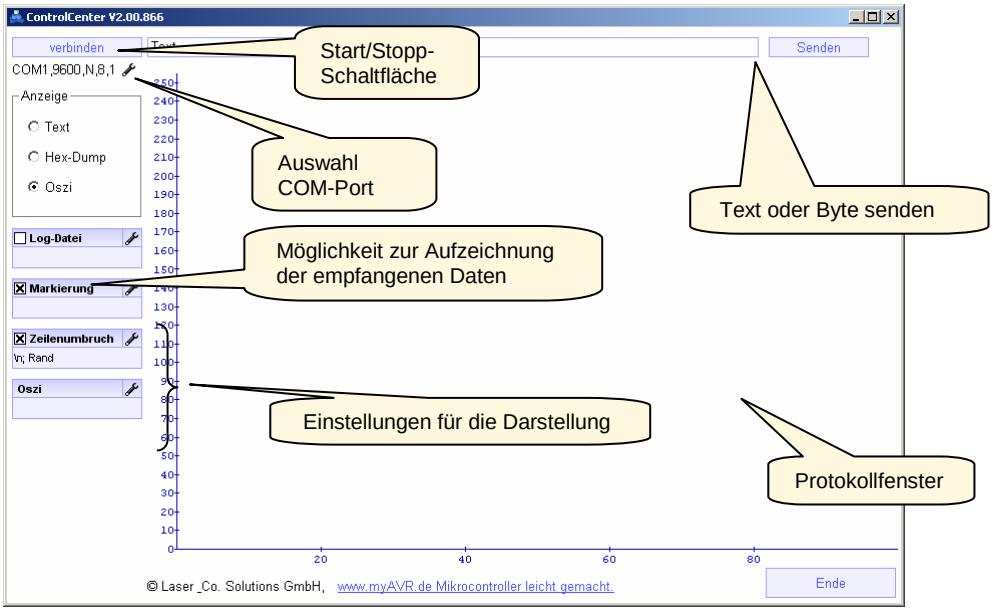
```
extern "C" void SysTickFunction(void)
{
    // Application SysTick
}
```



.
.
.
.

1.3.3 Das SiSy ControlCenter

Die Inbetriebnahme, der Test und die Datenkommunikation mit der Mikrocontrollerlösung erfolgen über das SiSy ControlCenter. Dabei wird über die Schaltfläche „Start“ das Testboard mit der nötigen Betriebsspannung versorgt und der Controller gestartet. Der Datenaustausch mit dem Board ist möglich, wenn das USB-Kabel an Rechner und Testboard angeschlossen, sowie die Mikrocontrollerlösung dafür vorgesehen ist. Es können Texte und Bytes (vorzeichenlose ganzzahlige Werte bis 255) an das Board gesendet und Text empfangen werden. Die empfangenen Daten werden im Protokollfenster angezeigt.



2 Erste Schritte mit dem XMC

Die Programmierung im klassischen C kann man sich ruhig einmal antun. Umso mehr wird man die Klassen aus dem myXMC Framework schätzen lernen. Des Weiteren finden sich im Netz auch jede Menge Beispiele in klassischem C. Die folgenden Abschnitte befähigen Sie, sich diese zugänglich zu machen. Falls Sie lieber gleich objektorientiert in C++ und UML anfangen möchten, dann überspringen Sie diesen Abschnitt einfach.

2.1 Hallo XMC in einfachem C

Die erste Übung in jedem Programmierkurs ist das berühmte „Hallo Welt“. Damit wird versucht, dem Lernenden ein motivierendes **„AHA-Erlebnis“** zu vermitteln. OK mal sehen, ob wir das auch hin bekommen. Bei der Programmierung von eingebetteten Systemen besteht oft das Problem, dass kein Bildschirm oder Display zur Textausgabe angeschlossen ist. Dann stehen für das „sich bemerkbar machen“ dem System nur LEDs zur Verfügung. Also leuchten und blinken eingebettete Systeme somit ihre Botschaft in die Welt. Ganz nebenbei lernen wir in diesem Abschnitt sehr viel über die digitale Ausgabe des XMC.
Zuerst betrachten wir die Programmierung des XMC auf der Ebene der internen Register, also die hard core Variante. Dann schauen wir uns an wie die Lösung aussieht wenn wir die *Infineon XMC Low Level Treiber* benutzen.

2.1.1 Eine LED einschalten

Aufgabe

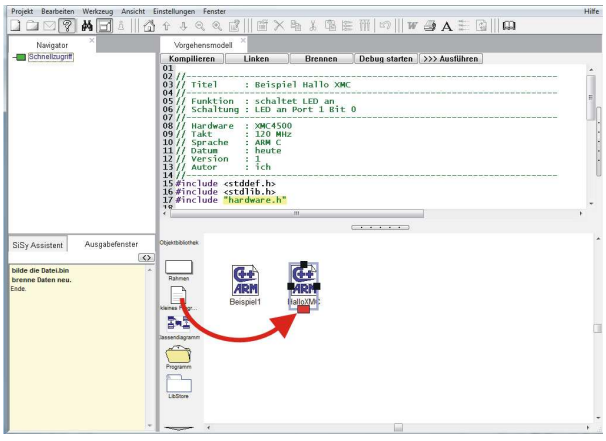
Die erste Übung soll das typische LED einschalten sein. Dazu nutzen wir eine der LEDs auf dem XMC4500 Relax Kit. Die LED ist bereits fest mit dem *Pin 1.0* verbunden.

Die Aufgabe besteht darin:

1. Port 1 Bit 0 als Ausgang zu konfigurieren
2. und das Pin auf High zu schalten

Vorbereitung

Im Kapitel 1.3.1 und 1.3.2 wurde bereits die Handhabung von SiSy kurz beschrieben; sowohl der Grundaufbau als auch eine erste Handhabung.
Falls das Projekt aus dem Einführungskapitel nicht mehr offen ist, öffnen Sie dies. Legen Sie bitte ein neues *kleines Programm* an und laden das *Grundgerüst ARM C++ Anwendung*. Beachten Sie die Einstellungen für die Zielplattform *XMC4500 Relax Kit*.



Erstellen Sie die Programmkopfdokumentation. Übersetzen und übertragen Sie das noch leere Programm auf den Controller, um die Verbindung zu testen.

```
//-----------------------------------------------------------
// Titel     : Beispiel Hallo Welt mit SiSy XMC
//-----------------------------------------------------------
// Funktion  : schaltet eine LED an
// Schaltung : LED an Port 1 Bit 0
//-----------------------------------------------------------
// Hardware  : XMC4500
// Takt      : 120 MHz
// Sprache   : ARM C++
// Datum     : heute
// Version   : 1
// Autor     : ich
//-----------------------------------------------------------
```

Grundlagen

Unter *GPIO (General Purpose Input/Output)* verstehen wir zunächst einmal einen allgemeinen Pin der als Kontakt aus dem Controllergehäuse herausgeführt ist. Dieser hat nach dem Einschalten bzw. RESET also dem Start des Controllers keine konkrete Funktion. Die gewünschte Funktion des GPIO-Pins kann vom Programmierer eines ARM verhältnismäßig frei festgelegt werden. Die Freiheit bewegt sich natürlich nur innerhalb der Möglichkeiten der Bus- und Cross-Connect-Matrix des jeweiligen Controllers.

.
.
.
.

Für das Einschalten der LED benötigen wir:

`XMC_GPIO_MODE_OUTPUT_PUSH_PULL`

Das Ausgaberegister *OUT* ist für die unmittelbare Ausgabe verantwortlich. Jedes Bit repräsentiert den geforderten Zustand am Pin (0=low, 1=high). Zusätzlich kann die Ausgabe auch indirekt über das Register ***OMR*** (Output Modification Register) erfolgen. Dieses Register kann wie folgt auf das *OUT* Register wirken.:

- ein oder mehrere Pins setzen
- ein oder mehrere Pins zurücksetzen
- ein oder mehrere Pins umschalten

.
.
.
.

Entwurf

Gewöhnen wir uns gleich daran einigermaßen systematisch vorzugehen. Bevor wir die Befehle in unseren Code wild hineinhacken, schreiben wir erst die Kommentare, was wir an dieser oder jener Stelle im Code tun wollen.


```
// Titel     : Hallo XMC
//-----------------------------------------------------
// Funktion  : LED leuchtet
// Schaltung : LED an Port 1, Bit 0
//-----------------------------------------------------
// Hardware  : XMC4500 Relax Kit
// Takt      : 120 MHz
// Sprache   : ARM C++
// Autor     : ich
//-----------------------------------------------------
#include <stddef.h>
#include <stdlib.h>
#include "hardware.h"
#include "xmc_common.h"
#include "xmc_gpio.h"

void initApplication()
```

.
.
.
.

Realisierung

Die Konfiguration eines Pins erfolgt dadurch, dass die korrekte Bitkombination (laut Referenzhandbuch `0b10000000 = GPIO_OUTPUT_TYPE_PUSH_PULL`) auf die entsprechende Position des IOCR-Blocks geschrieben wird. Für Pin1.0 ergibt sich die Position IOCR0 Offset 0. Eine einfache Möglichkeit der Konfiguration ist die direkte Zuweisung der erforderlichen Werte an das Register.

.
.
.
.

Ergänzen Sie den Quellcode des Beispiel *HalloXMC* wie folgt:

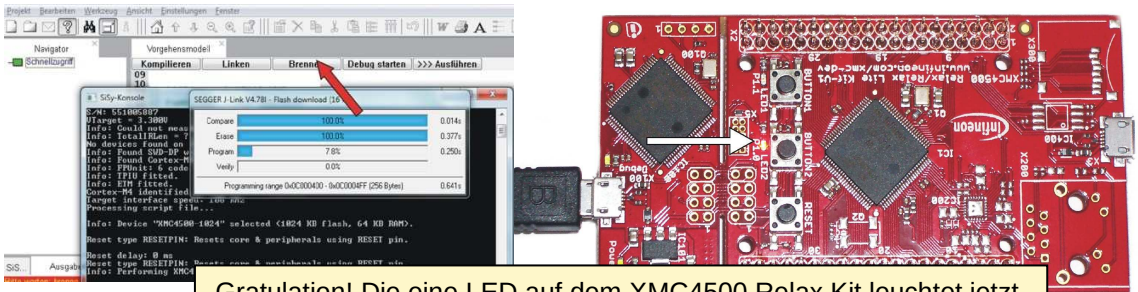
```
//-----------------------------------------------------
// Titel     : Hallo XMC
//-----------------------------------------------------
// Funktion  : LED leuchtet
// Schaltung : LED an Port 1, Bit 0
//-----------------------------------------------------
// Hardware  : XMC4500 Relax Kit
// Takt      : 120 MHz
// Sprache   : ARM C++
// Autor     : ich
//-----------------------------------------------------
#include <stddef.h>
#include <stdlib.h>
#include "hardware.h"
#include "xmc_common.h"
#include "xmc_gpio.h"
void initApplication()
```

.
.
.
.

Test

Übersetzen Sie das Programm. Korrigieren Sie ggf. Schreibfehler. Übertragen Sie das lauffähige Programm in den Programmspeicher des Controllers.

- Kompilieren
- Linken
- Brennen



Gratulation! Die eine LED auf dem XMC4500 Relax Kit leuchtet jetzt.

2.1.2 Erweiterung zum Klassiker „Blinky“

In den meisten Fällen gehen Beispiele für die erste einfache Ausgabe schon weiter und lassen gern mal eine oder mehrere LEDs blinken. Manchmal konfiguriert man dafür den *SysTick* so langsam, dass man dort das Blinken codieren kann. Diesen Weg werden wir nicht gehen. Der *SysTick* bleibt als System-Ereignis mit 10 Millisekunden unverändert. Der zweite Weg, der in Beispielen oft beschritten wird ist der, eine kleine Wartefunktion zu bauen, die es ermöglicht, das Blinken in der *Mainloop* zu realisieren. In SiSy gibt es bereits vorgefertigte Warteroutinen. Wir verwenden die Funktion *WaitMs*.

.
.
.
.

```
//---------------------------------------------------------------------------
// Titel     : Blinky mit dem XMC4500
//---------------------------------------------------------------------------
// Funktion  : LED blinkt
// Schaltung : LED an Port 1, Bit 0
//---------------------------------------------------------------------------
// Hardware  : XMC4500 Relax Kit
// Takt      : 120 MHz
// Sprache   : ARM C++
// Autor     : ich
//---------------------------------------------------------------------------
#include <stddef.h>
#include <stdlib.h>
#include "hardware.h"
#include "xmc_common.h"
#include "xmc_gpio.h"

void initApplication()
{
    // u.a. nötig für waitMs(..) und waitUs(..)
```

.
.
.
.

2.2 Einfache Ein- und Ausgaben mit dem XMC

Mit der zweiten Übung tasten wir uns im wahrsten Sinne des Wortes an die eigentliche Aufgabe eines jeden Controllers heran. Dieser soll Zeit seines Lebens fortlaufend Eingaben aus seiner Umgebung nach einer vorgegebenen Logik verarbeiten und entsprechend dieser Verarbeitung Ausgaben erzeugen. Es ist das allseits beliebte EVA-Prinzip, **gähn**. Spaß beiseite! Zum Gähnen ist das nur solange, wie man mit der EVA nichts Praktisches anfängt **grins**.

2.2.1 Ein- und Ausschalten einer LED

Aufgabe

Es soll ein Taster auf dem *XMC4500 Relax Kit* ausgewertet und bei Tastendruck eine LED eingeschaltet werden.
Es ist zunächst wieder wichtig, sich mit der konkreten Schaltung zu beschäftigen. Diese entnehmen wir der Produktbeschreibung von Infineon zum *XMC4500 Relax Kit* (Schaltplan Seite 15). In der folgenden vereinfachten Darstellung ist das Wesentliche zusammengefasst.

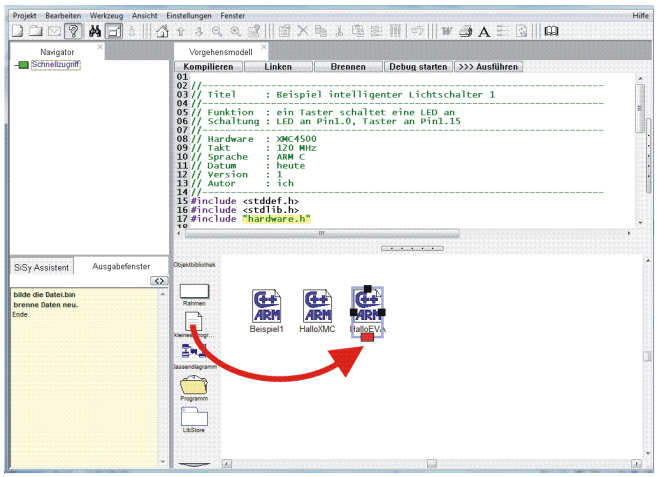
.
.
.
.

Die gewünschte LED hängt überraschenderweise *immer noch* an Port 1 Bit 0 und der Taster ist an Port 1 Bit 15 angeschlossen. Die tatsächliche Tasterlogik wurde hier etwas vereinfacht als Umschalter abgebildet. Es ist also Folgendes zu tun, um die Aufgabe zu erfüllen:

1. das Pin 1.0 als Ausgang konfigurieren
2. das Pin 1.15 als Eingang ohne PullUp konfigurieren
3. das Bit 15 von Port 1 einlesen
4. wenn der Taster gedrückt ist, also Pin 1.15 == 1
   - LED an Pin 1.0 auf High schalten
   - sonst auf Low schalten

Vorbereitung

Falls das Projekt nicht mehr offen ist, öffnen Sie dies. Legen Sie bitte ein neues kleines Programm an und laden das Grundgerüst ARM C++ Anwendung. Beachten Sie die Einstellungen für die Zielplattform *XMC4500 Relax Kit*.



Erstellen Sie die Programmkopfdokumentation. Übersetzen und übertragen Sie das noch leere Programm auf den Controller, um die Verbindung zu testen.

```
//-----------------------------------------------------------
// Titel     : Beispiel intelligenter Lichtschalter 1
//-----------------------------------------------------------
// Funktion  : ein Taster schaltet eine LED an
// Schaltung : LED an Pin1.0, Taster an Pin1.15
//-----------------------------------------------------------
// Hardware  : XMC4500
// Takt      : 120 MHz
// Sprache   : ARM C++
// Datum     : heute
// Version   : 1
// Autor     : ich
//-----------------------------------------------------------
```

Grundlagen

Zur Wiederholung hier noch einmal die Struktur eines GPIO beim XMC.

.
.
.
.

Der Taster schaltet gegen Masse. Das heißt, die direkte Eingabe liefert bei gedrücktem Taster eine logische 0, welche in C den Wahrheitswert *FALSE* repräsentiert. Die unter Umständen nötige Umkehrung der Logik kann in der Software oder eben listigerweise in der Hardware erfolgen. Einen controllerinternen PullUp oder PullDown brauchen wir nicht zu aktivieren, da dieser auf dem *XMC Relax Kit* extern, warum auch immer, bereits diskret bestückt ist. Nun ja, warum ist schon klar, die Benutzertaster auf dem XMC *Relax Kit* werden mit einem PullUp-Widerstand von 10 kΩ bereit gegen High gezogen und sind mit einem Kondensator von 100 nF hardwareseitig entprellt. Das macht es dem Anfänger etwas leichter. Er braucht den internen PullUp nicht per Software zu aktivieren und was viel schwerer wiegt, er braucht nicht per Software den Taster zu entprellen.

Entwurf

Zuerst wieder der Entwurf in Form von Kommentaren:

```
//-----------------------------------------------------------
// Titel     : Beispiel intelligenter Lichtschalter 1
//-----------------------------------------------------------
// Funktion  : ein Taster schaltet eine LED an
// Schaltung : LED an Pin1.0, Taster an Pin1.15
//-----------------------------------------------------------
// Hardware  : XMC4500
// Takt      : 120 MHz
// Sprache   : ARM C++
// Datum     : heute
// Version   : 1
// Autor     : ich
//-----------------------------------------------------------

#include <stddef.h>
#include <stdlib.h>
#include "hardware.h"
#include "xmc_gpio.h"
```



```
void initApplication()
{
    SysTick_Config(SystemCoreClock/100);
    // Konfiguriere Pin1.0 als Ausgang
    // Konfiguriere Pin1.15 als Eingang ohne PullUp
}

int main(void)
{
    SystemInit();
    initApplication();
    do{
        // WENN Taster an Pin1.15 gedrückt DANN
        // LED an Pin1.0 einschalten
        // SONST
        // LED an Pin1.0 ausschalten
    } while (true);
    return 0;
}

extern "C" void SysTickFunction(void)
{
    // hier nichts tun
}
```

Realisierung

Nachdem wir den Entwurf in Ruhe rekapituliert haben, kann es an die Umsetzung gehen. Die Konfiguration für die LED können wir einfach vom voran gegangenen Beispiel übernehmen.

.
.
.
.

Wenn Sie Quelltexte lieber kopieren, können Sie das gern mit dem obigen Entwurf machen, aber die eigentlichen Befehle sollten Sie aus lernpsychologischen Überlegungen tatsächlich selbst und bewusst, demzufolge selbstbewusst, eintippen.

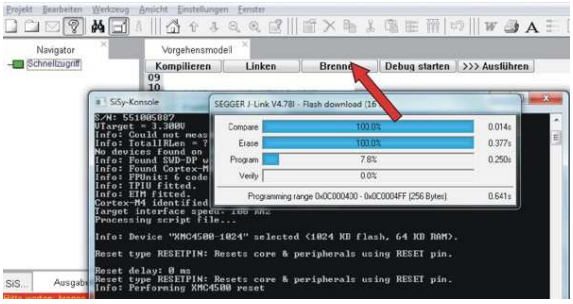
```
//---------------------------------------------------------------------
// Titel     : Beispiel intelligenter Lichtschalter 1 mit SiSy XMC
//---------------------------------------------------------------------
// Funktion  : ein Taster schaltet eine LED an
// Schaltung : LED an Pin1.0, Taster an Pin1.15
//---------------------------------------------------------------------
// Hardware  : XMC4500
// Takt      : 120 MHz
// Sprache   : ARM C++
// Datum     : heute
// Version   : 1
// Autor     : ich
//---------------------------------------------------------------------
#include <stddef.h>
#include <stdlib.h>
#include "hardware.h"
#include "xmc_gpio.h"
```

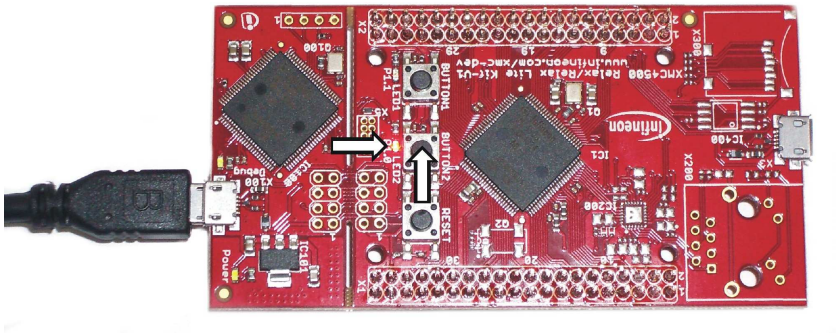
.
.
.
.

Test

Übersetzen Sie das Programm. Korrigieren Sie ggf. Schreibfehler. Übertragen Sie das lauffähige Programm in den Programmspeicher des Controllers.

1. Kompilieren
2. Linken
3. Brennen



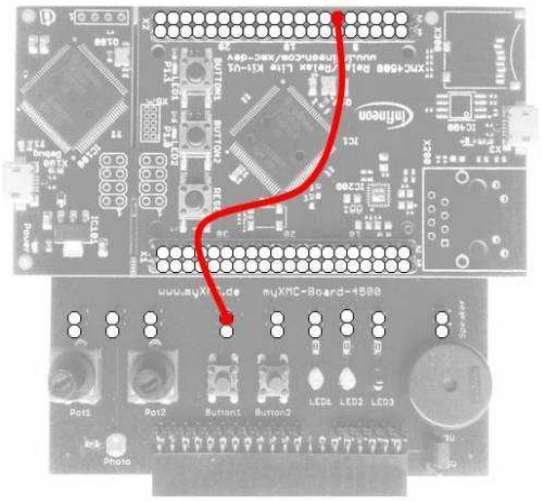


Die LED auf dem *XMC Relax Kit* leuchtet jetzt immer solange, wie der Taster gedrückt ist.

.
.
.
.

2.2.2 Variante mit internem Pull-Up

Wir wollen uns den Umstand zu Nutze machen, dass auf dem Erweiterungsboard *myXMC-Board-4500* die Taster ohne PullUp und Kondensator bestückt sind. Damit können wir den internen PullUp des XMC ausprobieren. Dazu verbinden wir mit einem der Patchkabel den Taster 1 auf dem Erweiterungsboard mit Pin 1.13 des XMC.



.
.
.
.

2.3 Der SystemTick in C

ARM Controller sind prädestiniert für den Einsatz spezieller Laufzeitumgebungen oder bestimmter Betriebssysteme. Solche basieren oft auf einer timer-getriggerten Verteilung von Ressourcen, vor allem der Ressource Rechenzeit. Dafür steht beim ARM ein spezieller Timer zur Verfügung, der ausschließlich die Aufgabe hat, ein System-Trigger-Ereignis zu generieren. Auch ohne Echtzeitbetriebssystem ist dieser SystemTick für den Anwendungsentwickler sehr interessant. Die verwendeten Programmvorlagen, und insbesondere das später verwendete myARM C++ Framework, sind bereits auf die Nutzung des *SysTick* vorbereitet bzw. basieren darauf.

2.3.1 Zyklisches Blinken von LEDs

Aufgabe

Diese Übung wird eine einfache Verwendung der *SysTickFunction* zur Generierung zyklischer Ausgaben demonstrieren. Wir lassen die LEDs auf dem Board abwechselnd blinken. Das folgende Blockbild verdeutlicht, welche Bausteine bei dieser Aufgabe eine Rolle spielen.

.
.
.
.

Die zwei LEDs auf dem *XMC4500 Relax Kit* sind immer noch fest mit den Pins 1.0 und 1.1 verbunden. Der SystemTick soll so konfiguriert werden, dass dieses Ereignis alle 10 Millisekunden eintritt. Fassen wir die Aufgaben zusammen:

1. das *SysTick-Ereignis* auf 10 ms konfigurieren
2. die Pins 1.0 und 1.1 als Ausgang konfigurieren
3. wenn das *SysTick-Ereignis* eintritt, LEDs unterschiedlich blinken lassen

Vorbereitung

Legen Sie bitte ein neues kleines Programm an und laden Sie das Grundgerüst für eine ARM C++ Anwendung. Beachten Sie die Einstellungen für die Zielplattform *XMC4500 Relax Kit*.

Erstellen Sie die Programmkopfdokumentation. Übersetzen und übertragen Sie das noch leere Programm auf den Controller, um die Verbindung zu testen.

```
//------------------------------------------------------
// Titel     : Beispiel einfache SysTick-Nutzung
//------------------------------------------------------
// Funktion  : lässt die LEDs zyklisch blinken
// Schaltung : LEDs an Pin1.0 und Pin1.1
//------------------------------------------------------
// Hardware  : XMC4500
// Takt      : 120 MHz
// Sprache   : ARM C++
// Datum     : 08.10.2012
// Version   : 1
// Autor     : ich
//------------------------------------------------------
```

Grundlagen

Der XMC verfügt wie jeder ARM Cortex-M über einen 24 Bit System Timer. Dieser ist explizit für Betriebssysteme vorgesehen die einen Trigger für das Prozess-

Scheduling benötigen. Da wir bei unseren Beispielen noch nicht auf ein Betriebssystem zurückgreifen, gehört der SysTick voll und ganz uns. Wir können damit im weiteren Verlauf schon ein bisschen einfache Nebenläufigkeit (Parallelverarbeitung) programmieren oder wie hier den SysTick als Trigger für zyklische Aufgaben nutzen, statt die Prozessorzeit mit Warteroutinen zu verbraten. Standardmäßig konfigurieren wir den SysTick auf 10 Millisekunden. Dieser Wert bewegt sich im Bereich der typischen Zeitscheibe von Betriebssystemen für eingebettete Systeme. Der SysTick-Timer löst einen entsprechenden Interrupt aus. Die *ISR* (Interrupt Service Routine) oder auch Interrupt-Handler genannt wird beim Laden eines Grundgerüstes für den XMC bereits vorgefertigt angeboten.

```
extern "C" void SysTick_Handler(void)
{
    // Application SysTick
}
```

.
.
.
.

Entwurf

Die 100 Hz oder auch 10 ms, mit der das *SysTick*-Ereignis ausgelöst wird, ist für das Blinken einer LED viel zu schnell. Das könnten wir mit unserem Auge nicht mehr wahrnehmen. Wir müssen uns etwas einfallen lassen, nur jeden 10ten oder 20ten SysTick für das Umschalten der LED zu nutzen.
Auch wenn es uns in den Fingern kribbelt sofort die nötigen Befehle einzugeben, kasteien wir uns mit dem folgenden Entwurf.

```
//-------------------------------------------------
// Titel     : Beispiel einfache SysTick-Nutzung
//-------------------------------------------------
// Funktion  : lässt die LEDs zyklisch blinken
// Schaltung : LEDs an Pin1.0 und Pin1.1
//-------------------------------------------------
// Hardware  : XMC4500
// Takt      : 120 MHz
// Sprache   : ARM C++
// Datum     : heute
// Version   : 1
// Autor     : ich
//-------------------------------------------------

#include <stddef.h>
#include <stdlib.h>
#include "hardware.h"
#include "xmc_gpio.h"

void initApplication()
{
    SysTick_Config(SystemCoreClock/100);
    // weitere Initialisierungen durchführen
    // Konfiguriere Pin1.0 und Pin1.1 als Ausgang
}

int main(void)
{
    SystemInit();
    initApplication();
    while(true)
```



```
    {
        //leer
    }
    return 0;
}

extern "C" void SysTickFunction(void)
{
    // Zähler anlegen
    // Zähler eins hoch zählen
    // wenn Zähler = 10 dann Pin1.0 umschalten
    // wenn Zähler = 20 dann Pin1.1 umschalten
}
```

Realisierung

Nachdem wir den Entwurf bei einem kräftigen Schluck Kaffee auf uns haben wirken lassen, kann der Code erstellt werden.

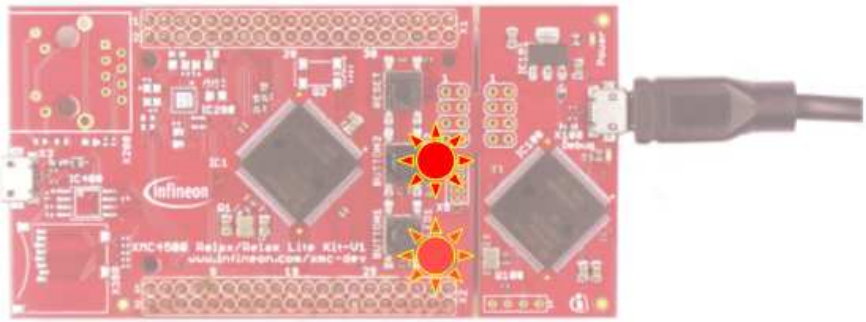
```
//-----------------------------------------------------------------------
// Titel     : Beispiel einfache SysTick-Nutzung in SiSy XMC
//-----------------------------------------------------------------------
// Funktion  : lässt die LEDs zyklisch blinken
// Schaltung : LEDs an Pin1.0 und Pin1.1
//-----------------------------------------------------------------------
// Hardware  : XMC4500
// Takt      : 120 MHz
// Sprache   : ARM C++
// Datum     : heute
// Version   : 1
// Autor     : ich
//-----------------------------------------------------------------------
#include <stddef.h>
#include <stdlib.h>
#include "hardware.h"
#include "xmc_gpio.h"

void initApplication()
```

.
.
.
.

Test

Übersetzen Sie das Programm. Übertragen Sie das lauffähige Programm in den Programmspeicher des Controllers.

1. Kompilieren
2. Linken
3. Brennen



.
.
.
.

3 Ausgewählte Paradigmen der Softwareentwicklung

3.1 Basiskonzepte

Ausgewählte Basiskonzepte objektorientierter Programmiersprachen sollen hier kurz umrissen werden. Sie müssen diesen Teil nicht unbedingt lesen, um das Lehrbuch nachzuvollziehen. Es lohnt jedoch in jedem Falle, sich intensiver mit dieser Problematik zu beschäftigen. Zum objektorientierten Paradigma zählen folgende Konzepte:

- **Abstraktion**
- **Objekte** mit Eigenschaften, Verhalten und Zuständen
- **Klassen** als abstrahierte Objekte
- **Vererbung**, auch Generalisierung oder Spezialisierung
- **Kapselung** und **Nachrichten**, um Merkmale zu schützen
- **Assoziation**, **Aggregation** und **Komposition**
- **Polymorphie**

Abstraktion

lat. abstractus „abgezogen“, von abs-trahere „abziehen, entfernen, trennen“
Bedeutung: von der Gegenständlichkeit losgelöst

Nicht erschrecken. Die Herleitung des Begriffs ist wichtig. Verweilen Sie einen Moment bei dem Gedanken: „von der Gegenständlichkeit losgelöst“. Das bedeutet nichts anderes, als dass wir in der Lage sind, mit etwas umzugehen, ohne dass es da sein muss. Frauen reden über Männer sogar am eifrigsten, wenn diese nicht anwesend sind! Und damit haben wir auch schon den Bogen zur Sprache geschlagen. Sprache ist Ausdruck der uns von Natur aus gegebenen Fähigkeit zu abstrahieren. Das Gegenständliche bilden wir in Begriffen ab. Und mehr soll an dieser Stelle dazu auch nicht gesagt werden.

Objekt

Dinge bezeichnet der Fachmann als Objekte. Objekte, das sind die Bausteine, aus denen die Systeme, welche wir programmieren wollen, bestehen. Für uns sind das zum Beispiel der *ARM-**Controller***, vielleicht ein ***Taster*** und eine ***LED*** usw.. Diese Objekte besitzen konkrete Eigenschaften und typisches Verhalten. Der Controller hat eine bestimmte Speicherkapazität, der Taster prellt etwas und die LED leuchtet grün. Die Eigenschaften bilden wir in Programmen als Variablen (Attribute) und das Verhalten als Funktionen (auch Methoden bzw. Operationen genannt) ab.

.
.
.

Klasse

Der Name, welchen wir für ein Ding benutzen, bezeichnet meist nicht nur das einzelne Ding, sondern eine Menge (Gruppe) gleichartiger Dinge. Nehmen wir zum Beispiel den *Taster*. Davon haben wir auf unserem Experimentierboard schon mal drei Stück. Um diese zu unterscheiden, geben wir jedem noch einen individuellen Namen nämlich *„Taster-1“, „Taster-2“* und *„Taster-3“.* Taster steht also als Begriff für alle Schalter mit den entsprechenden gleichen Eigenschaften.

.
.
.

Vererbung

Taster haben mit DIP-Schaltern und Bausteinen der 74er Reihe etwas gemeinsam, sie liefern digitale Signale. Wesentliche Aspekte der Programmierung für Ein- und Ausgaben sind bei all diesen Bausteinen gleich. In der objektorientierten Systementwicklung ist man bestrebt, gemeinsame Merkmale (Attribute und Operationen) nur einmal zu programmieren, um Arbeitsaufwand zu sparen. Das einmal programmierte Merkmal soll aber in allen Bausteinen wieder verwendet werden.

.
.
.

Kapselung (Sichtbarkeit)

Besonders dann, wenn der Benutzer einer Klasse nicht deren Entwickler ist, kann es wichtig sein, bestimmte interne Sachverhalte der Klasse vor versehentlich falscher Benutzung zu schützen. Dafür kennt die Objektorientierung das Konzept der Sichtbarkeit. Attributen und Operationen können zum Schutz vor unsachgemäßem Zugriff unterschiedliche Sichtbarkeiten zugewiesen werden. Man unterscheidet in der Theorie zwischen den Sichtbarkeiten *public, package, protected und privat*.

.
.
.

Aggregation

Um Komplexität zu beherrschen, kann man größere Dinge aus kleineren zusammenbauen. Dabei ist das Ganze *verantwortlich* für seine Einzelteile. Das ist nicht nur beim Programmieren so. Es verwundert also nicht, dass in einer objektorientierten Programmiersprache komplexe Klassen aus einfachen zusammengebaut werden können. Die ***Verantwortlichkeit*** kann man vereinfacht mit ***„hat“*** ausdrücken.

Nachricht

Das Konzept der Nachrichten müssen wir aus zwei Blickwinkeln betrachten. Zum einen im Zusammenhang mit der oben genannten Kapselung. Es hat sich bewährt, vor allem Attribute vor dem direkten Zugriff zu schützen und das Schreiben oder Lesen von Attributen nur über den Aufruf von Operationen zu ermöglichen. Dieser Aufruf von Operationen bedeutet, dem betreffenden Objekt eine Nachricht zu senden. Das Objekt kann auf die Nachricht antworten; diese Antwort ist dann der Rückgabewert der Operation. Somit kommuniziert also das System mit seinen Bausteinen über diese Nachrichten.

.
.
.

Assoziation

Manchmal ist es erforderlich, dass Einzelteile direkt miteinander Nachrichten austauschen; der kleine Dienstweg sozusagen. Dabei besteht als wesentlicher Unterschied zur Aggregation, dass die Einzelteile nicht füreinander verantwortlich sind, sich aber ***kennen***.

Lange Rede, kurzer Sinn

Unsere natürliche Sprache ist objektorientiert!

```
Wenn der Taster gedrückt ist schalte die LED an.
If the button is pressed the LED will turn on.
if button.isPressed then led.on
if ( button.isPressed() ) led.on();
```

3.2 Grundzüge der Objektorientierung

Wie eingangs schon beschrieben kann und soll dieses Lehrbuch kein C-Lehrbuch sein. Es ist für das Verstehen auf jeden Fall von Vorteil, wenn Kenntnisse in einer höheren Programmiersprache vorhanden sind; am besten natürlich C. Für jeden, der über keine oder noch wenig Programmierkenntnisse verfügt, ist zu empfehlen, ein entsprechendes C/C++ Nachschlagewerk (Lehrbuch oder online-Tutorial) diesem Lehrbuch beizustellen und jede Klammer sowie jeden Ausdruck, der in den angebotenen Quelltexten unklar ist, nachzuschlagen.

.
.
.
.

3.2.1 Wesentliche Merkmale von C

Auszug von Sprachumfang in C

```
// Schlüsselworte ........................................
break           double          int             struct
case            else            long            switch
char            extern          return          unsigned
const           float           short           signed
continue        for             void            sizeof
default         if              static          volatile
do              while           main
// Operatoren ...........................................
+               -               *               /
=               ++              --
<<              >>              !               &
|               ^               ~               %
==              >               <               <=
>=              &&              ||              !=
```

Wesentliche C-Steuerflusskonstrukte

C folgt dem Basiskonzept der Strukturierung, d.h dass auf die Sprunganweisung verzichtet wird, um einen Algorithmus zu erstellen. Es werden nur drei algorithmische Grundstrukturen benötigt, um jede Programmlogik zu realisieren. Das sind:

- die *Sequenz*, Anweisungen werden nacheinander ausgeführt
- die *Iteration*, Anweisungen werden wiederholt ausgeführt
- die *Alternative*, bestimmte Anweisungen werden entsprechend einer Bedingung, anstatt anderer Anweisungen ausgeführt

In den strukturierten Sprachen gibt es zwar noch weitere Spielarten dieser drei Grundstrukturen, aber dem Wesen nach gibt es tatsächlich nur diese drei. Im Folgenden soll dargestellt werden wie diese Algorithmusbausteine in C und dann natürlich auch in C++ notiert werden. Beachten Sie die geschweiften Klammern! Diese begrenzen einen Anweisungsblock und sollten immer, besonders am Anfang, diszipliniert gesetzt werden.

```
// Blöcke in C und C++
      { // BEGIN
        // Blockinhalt
      } // END
```

Blöcke können zusätzlich einen Namen und Parameter erhalten.

```
      name ( parameter )
      { // BEGIN
        // Blockinhalt
      } // END
```


Diese Namen können sogenannte reservierte Worte, also Schlüsselworte der Sprache C/C++ sein. Damit wird dem bezeichneten Block eine bestimmte Funktionalität zugewiesen.

.
.
.
.

3.2.2 C++: die objektorientierte Erweiterung der Sprache C

Zusätzlicher Sprachumfang von C++ (Auswahl)

```
bool        catch       false       enum
class       new         delete      public      template
virtual     operator    private     protected   this
namespace   using       true        throw       try
```

Deklarieren von Klassen in C++

Es ist ein Anwendungsprogramm mit dem Namen *Applikation* (englisch: *Application*) zu entwickeln. Die Anwendung soll zunächst geplant und dann programmiert werden und letztlich benötigen wir noch eine Instanz von dem Programm.

```
// Klasse Name { Bauplan } Instanz;
class Application
{

} app;
```

Vererbung in C++

Die Applikation **ist eine** Mikrocontrolleranwendung. Diese soll alle Möglichkeiten der vorhandenen Klasse *Controller* besitzen. Somit erbt die Applikation am besten alle Merkmale vom Controller.

```
//Klasse Name:Sichtbarkeit Basisklasse { Bauplanerweiterung } Instanz;
class Application : public Controller
{

} app;
```

Operationen in C++

Der Controller wird eingeschaltet und arbeitet dann fortlaufend taktgesteuert. Oh ja, wir erinnern uns dunkel. Subjekt und Prädikat. WER (der Controller) macht WAS (wird eingeschaltet, arbeitet)… Dafür sollte es jetzt Operationen in der Klasse geben.

```
//Klasse Name:Sichtbarkeit Basisklasse { Bauplanerweiterung } Instanz;
class Application : public Controller
{
   // Sichtbarkeit : RückgabeTyp name (Parameter) { Code; }
   public: void onStart()
   {
      // alles was beim Hochfahren getan werden muss
      // dann gehe zur Mainloop
   }
   // Sichtbarkeit : RückgabeTyp name (Parameter) { Code; }
   public: void onWork()
   {
      // alles was fortlaufend getan werden muss
      // die Mainloop liegt in der Controllerklasse
      // von dort aus wird onWork fortlaufend aufgerufen (getriggert)
```

```
      // hier also KEINE Unendlichschleife !!!
   }
} app;
```

Aggregationen und Kapselung in C++

Es soll eine LED angeschlossen werden. An diese LED wollen wir niemand anderen heran lassen. Wir schützen diese vor unberechtigtem Zugriff.

```
//Klasse Name:Sichtbarkeit Basisklasse { Bauplanerweiterung } Instanz;
class Application : public Controller
{
   // Sichtbarkeit : Typ name;
   protected: LED led;

   public: void onStart()
   {
      // ...
   }
   public: void onWork()
   {
      // ...
   }
} app;
```

Nachrichten in C++

Die LED ist eine fertige Klasse aus dem Framework. Wir müssen der LED mitteilen, an welchem Port-Pin sie angeschlossen ist und wir wollen sie einschalten.

```
//Klasse Name:Sichtbarkeit Basisklasse { Bauplanerweiterung } Instanz;
class Application : public Controller
{
   // Sichtbarkeit : Typ name;
   protected: LED led;

   public: onStart()
   {
      // instanzName . nachricht ( Parameter );
      led.config(pin22);
   }
   public: onWork()
   {
      // instanzName . nachricht ( );
      led.on();
   }
} app;
```

Zwischenfazit

Bei diesem kurzen Ausflug in die objektorientierte Art und Weise Programme zu schreiben ist wohl deutlich geworden, dass es sehr darauf ankommt, sich ein bestimmtes Muster anzugewöhnen, Systeme zu betrachten und darüber nachzudenken.

Objektorientierung beginnt im Kopf!

Übrigens ist es hilfreich, das zu programmierende System in kurzen einfachen Sätzen zu beschreiben oder diese laut vor sich hin zu sagen.

3.3 Einführung in die UML

Die Unified Modeling Language ist ein Satz von Darstellungsregeln (Notation) zur Beschreibung objektorientierter Softwaresysteme. Ihre ursprünglichen Autoren Grady Booch, James Rumbaugh und Ivar Jacobson verfolgten mit der eigens gegründeten Firma Rational anfangs vor allem kommerzielle Ziele. Sie übergaben die UML jedoch im weiteren Verlauf der Entwicklung als offenen Standard an eine nicht kommerzielle Organisation, der Object Management Group (www.omg.org). Im Jahre 1996 wurde die UML durch die OMG und inzwischen auch durch die ISO (International Organization for Standardization) mit der ISO/IEC-19505 zu einem internationalen Standard erhoben. Die OMG entwickelt die UML und auf der UML basierende Konzepte und Standards weiter. Die UML soll nach den Wünschen der Autoren eine Reihe von Aufgaben und Zielen verfolgen, so zum Beispiel:

- Bereitstellung einer universellen Beschreibungssprache für alle Arten objektorientierter Softwaresysteme und damit eine Standardisierung,
- Vereinigung der beliebtesten Darstellungstechniken (best practice),
- ein für zukünftige Anforderungen offenes Konzept
- Architekturzentrierter Entwurf

Durch die UML sollen Softwaresysteme besser

- analysiert
- entworfen und
- dokumentiert werden

die Unified Modeling Language …

- ist NICHT perfekt, wird aber immer besser
- ist NICHT vollständig, wird aber immer umfangreicher
- ist KEINE Programmiersprache, man kann mit ihr aber programmieren
- ist KEIN vollständiger Ersatz für eine Textbeschreibung, man kann mit ihr aber immer mehr beschreiben
- ist KEINE Methode oder Vorgehensmodell, mit ihr wird die Systementwicklung aber viel methodischer
- ist NICHT für alle Aufgabenklassen geeignet, sie dringt jedoch in immer mehr Aufgabengebiete vor

Die UML spezifiziert selbst keine explizite Diagrammhierarchie. Die Diagramme der UML werden verschiedenen semantischen Bereichen zugeordnet.

.
.
.
.

Wichtige UML Notationen für Strukturen

Die Strukturdiagramme bilden das Fundament für alle weiteren Darstellungsmittel der UML. Das ergibt sich natürlich nicht zuletzt aus dem theoretischen Ansatz der Objektorientierung (kein Verhalten ohne Struktur). Es kann nur dann konkretes Verhalten geben, wenn die dafür nötigen Strukturen (Instanzen/Objekte) existieren. Sie können nur dann Auto fahren, wenn Sie eine Instanz vom Typ Auto zur Verfügung haben. Die Beschreibung, also den Bauplan eines Autos, können Sie nicht zum Fahren benutzen. Das Auto muss nach dem Plan gebaut werden, es muss eine Instanz von dem Auto erstellt werden. Daraus wird ersichtlich, dass es zwei Blickwinkel auf Strukturen gibt, die Sichtweise auf Typen (Klassen) und die Sichtweise auf Instanzen (Objekte). Die Typebene ist die der Beschreibung wie es sein soll, und die Instanzebene wird immer dann eingenommen, wenn etwas passieren soll.

Die Basis für alles Weitere in der UML ist die Beschreibung von Klassen. Das Klassendiagramm ist die Konstruktionszeichnung des Systems.

.
.
.
.

Eine ausführlichere Notationsübersicht finden Sie in unserem Lehrbuch „Software Engineering für Embedded Systems“.
Im Weiteren werden wir ausschließlich Entwurf und Realisierung einer ARM-Anwendung im Klassendiagramm durchführen bzw. über die UML darstellen.

.
.
.
.

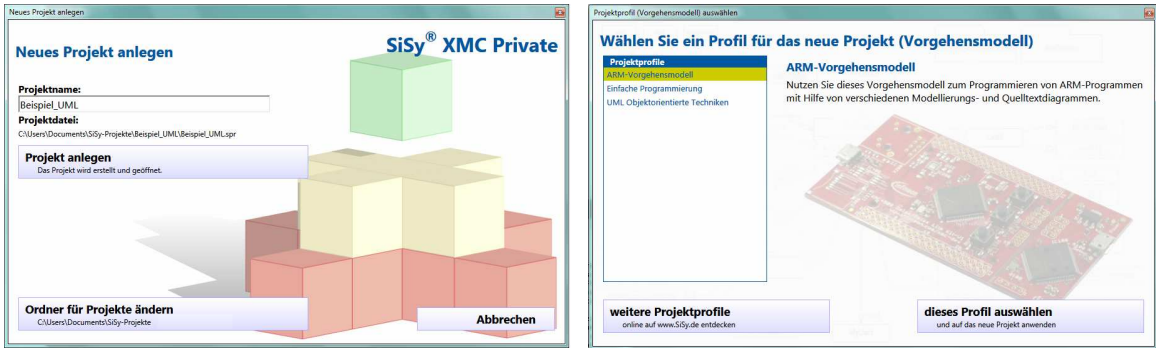
.

4 ARM Programmierung in C++ mit der UML

4.1 Grundstruktur

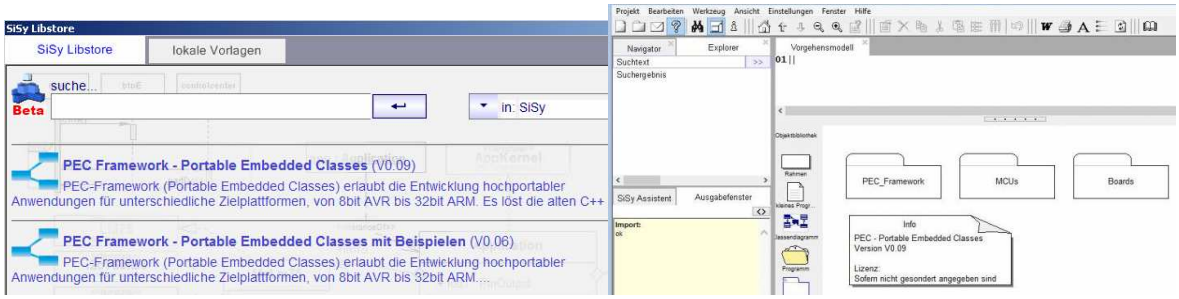
Ein UML Projekt anlegen

Für die weitere Arbeit in diesem Lehrbuch verwenden wir als Entwicklungsumgebung das UML Klassendiagramm und portable Klassenbibliotheken (PEC), welche auch die XMC-Controllerfamilien unterstützen. Im Kapitel 3.4.2 wurde bereits auf die Arbeit mit dem UML Klassendiagramm eingegangen.
Es ist nötig, ein neues Projekt anzulegen und eine Projektvorlage mit den gewünschten Bibliotheken auszuwählen. Zur Übung schauen wir uns diesen Vorgang noch einmal ausführlicher an. Legen Sie ein neues SiSy-Projekt an und wählen Sie das ARM-Vorgehensmodell.



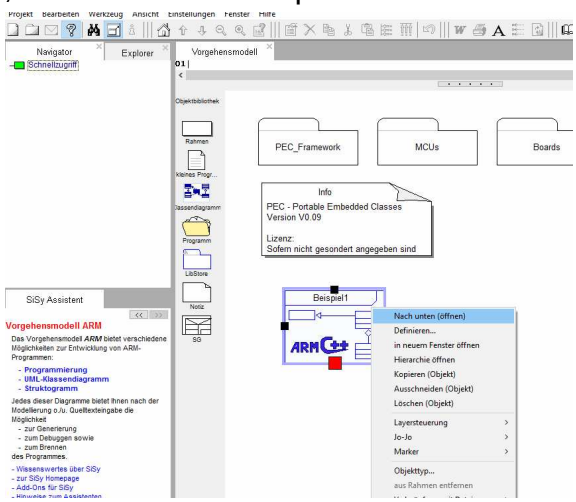
Abbildungen: Ein neues Projekt in SiSy anlegen und Vorgehensmodell auswählen

Sie erhalten im SiSy LibStore verschiedene Vorlagen zur Auswahl. Bitte laden Sie die Vorlage für das *PEC FrameWork* ohne Beispiele. Schränken Sie die Vorlagensuche ggf. mit dem Suchbegriff „PEC“ ein.



Abbildungen: Vorlage in SiSy LibStore auswählen und ausgeführter Download

Legen Sie ein neues Klassendiagramm an, indem Sie das entsprechende Element per Drag&Drop aus der Objektbibliothek in das Diagrammfenster ziehen. Achten Sie auf die Einstellung der Zielsprache „ARM C++“ und die Einstellungen für die Zielplattform „XMC4500 Relax Kit“.



Öffnen Sie das Klassendiagramm, indem Sie auf diesem das Kontextmenü (rechte Maustaste) öffnen und den Menüpunkt *nach unten (öffnen)* wählen.

Laden Sie aus dem SiSy LibStore die Diagrammvorlage „*Application Grundgerüst für PEC Anwendungen (XMC, STM32, AVR)*“.

Weisen Sie dem Diagramm das Treiberpaket für den konkreten Controller *MCU_XMC4500* zu. Sie finden dieses Paket über den Navigator (UML-Pakete) oder über die Suchfunktion im Explorer.

Hinweis: Aktivieren Sie im Diagrammfenster die Schaltfläche „Suche MCUs im Explorer“. Oben links erscheint das Fenster „MCU-Explorer“. Ziehen Sie das Objekt *MCU_XMC4500* in das Diagrammfenster.

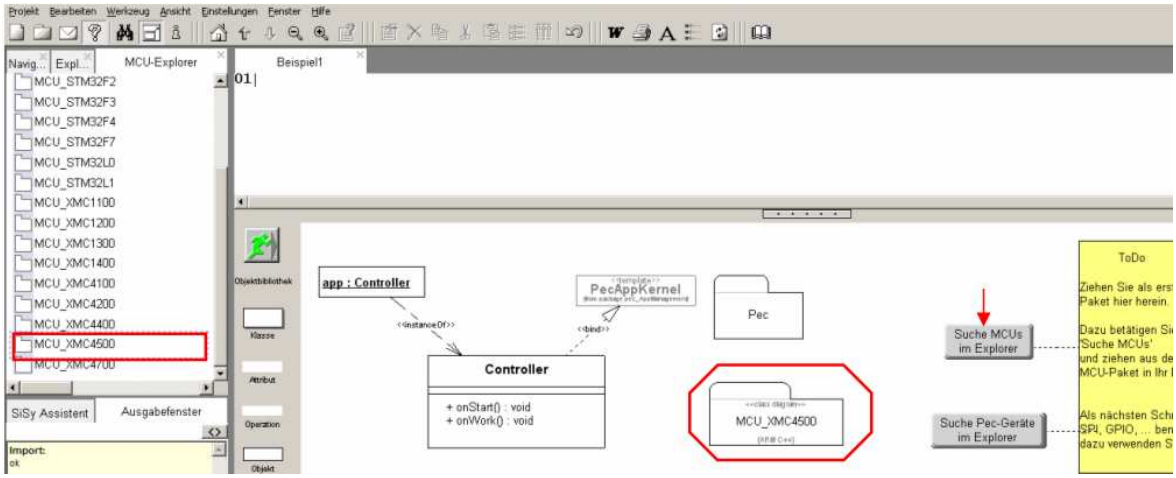


Abbildung: Treiberpaket für MCU_XMC4500 im MCU-Explorer auswählen

Grundstruktur einer objektorientierten XMC Anwendung

Bei dem so erhaltenen Diagramm handelt es sich um die typische Grundstruktur einer objektorientierten Anwendung auf der Basis des SiSy XMC Framework.

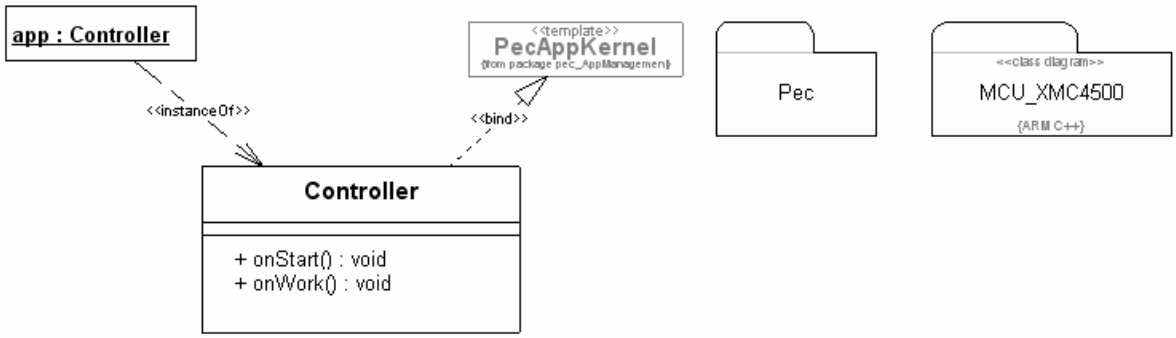
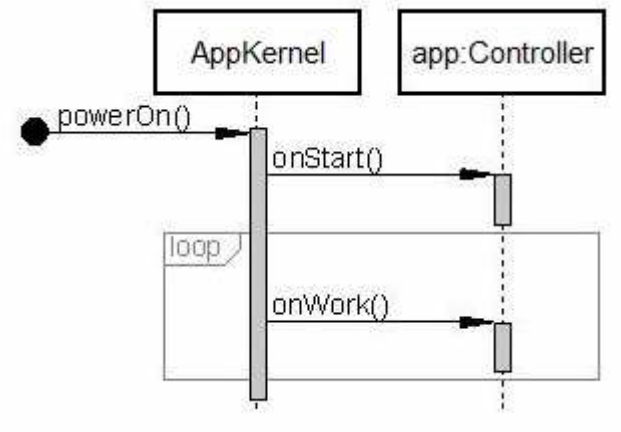


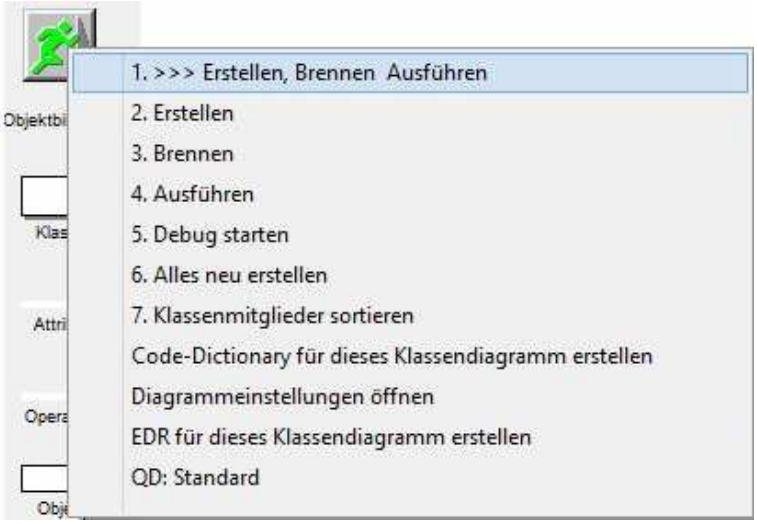
Abbildung: Grundstruktur für die weitere Arbeit einer objektorientierten Anwendung

Die Klasse *Controller* abstrahiert genau diesen in unserem System und ist eine Realisierung eines *PecAppKernel*. Es handelt sich um die sogenannte Anwendungsklasse. Diese nimmt die Rolle der gesamten Anwendung ein und muss als erstes ausgeführt werden. Das Objekt *app:Controller* ist die Instanz der Anwendungsklasse. Über die Referenzen der Pakete *Pec* und *MCU_XMC4500* werden alle benötigten Klassen aus der Bibliothek importiert.
Das Template *PecAppKernel* stellt bereits eine Reihe von nützlichen Struktur- und Verhaltensmerkmalen einer XMC-Anwendung bereit. Zwei Operationen sind in der Klasse *Controller* zur Realisierung vorbereitet. Die Operation *onStart* dient der Initialisierung nach dem Systemstart, bildet also die Initalisierungssequenz. Die

Operation *onWork* wird durch das Framework zyklisch aufgerufen. Damit nimmt diese die Position der *Mainloop* ein. Beachten Sie, dass die *Mainloop* jetzt selbst im Framework vor unseren Augen verborgen läuft und nicht mehr von uns geschrieben werden muss. Zur Verdeutlichung und zur Gewöhnung hier das grundsätzliche Verhalten der Anwendung als UML-Sequenzdiagramm.



So wie die Anwendung jetzt vor uns liegt tut das Programm nichts, sondern läuft im Leerlauf. Trotzdem wollen wir aus dem Klassendiagramm den Quellcode generieren, diesen übersetzen und auf den Controller übertragen. Das erfolgt über das Aktionsmenü in der Objektbibliothek. Wählen Sie dort den Menüpunkt „Erstellen, Brennen Ausführen“.



4.2 Hallo XMC-Welt in C++ und UML

So, dann frisch ans Werk. Die erste Übung mit der wahrscheinlich ungewohnten Umgebung soll wieder das einfache Einschalten einer LED sein. Der Sinn und Zweck von Klassenbibliotheken ist natürlich vor allen auch der, dass Dinge die öfters gebraucht werden oder typische Problemstellungen die einfach schon mal gelöst wurden, dem Anwender komfortabel zur Wiederverwendung zur Verfügung stehen.

Die Aufgabe

Die Objektorientierung zeichnet sich durch zunehmende Abstraktion von der tatsächlichen inneren Struktur und dem internen Verhalten der Maschine, hin zu einer anwenderbezogenen Sichtweise aus.

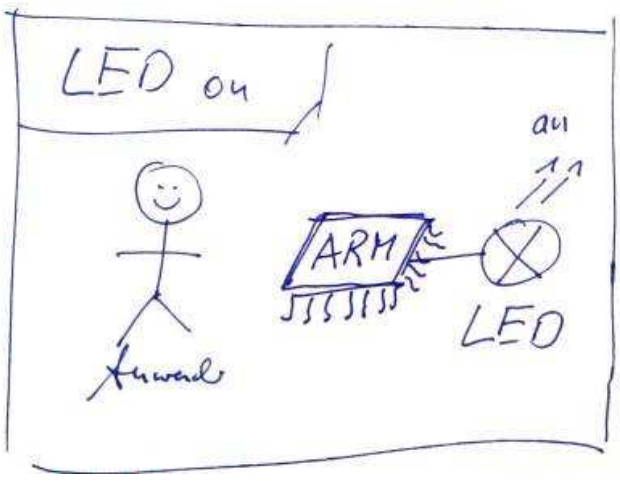
Die Aufgabe lautet:
Entwickeln Sie eine Mikrocontrolleranwendung, die eine LED anschaltet.

Vorbereitung

Falls Sie jetzt noch das Klassendiagramm geöffnet haben wählen Sie im Kontextmenü (rechte Maustaste) des Diagramms den Menüpunkt nach oben. Falls das Projekt nicht mehr geöffnet ist, öffnen Sie das SiSy UML-Projekt wieder. Legen Sie ein neues Klassendiagramm an und wählen Sie die Sprache ARM C++. Beachten Sie die Einstellungen für die Zielplattform XMC4500 Relax Kit. Beim Öffnen des Diagramms (rechte Maustaste, nach unten) laden Sie die Diagrammvorlage für eine *PEC Applikation* und weisen das Treiberpaket für den *XMC4500* zu.

Grundlagen

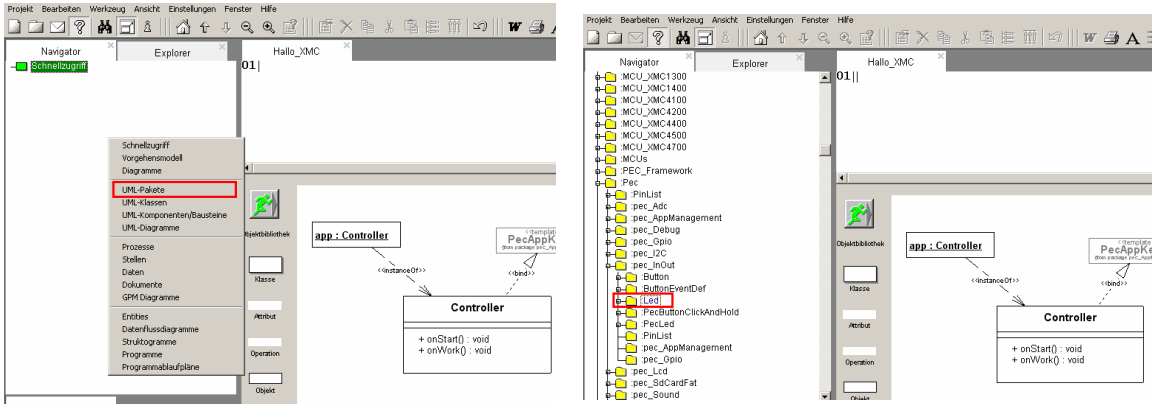
Die Aufgabe besteht darin eine LED anzusteuern. Folgen wir der objektorientierten Sichtweise, ist die *Led* unser Klassenkandidat. Eine Klasse *Led* soll die spezifische Initialisierung und Programmierung eines GPIO Pin auf der Anwenderebene abstrahieren. Also fragen wir uns, was eine LED denn aus Anwendersicht so tut. Sie kann an oder aus sein, vielleicht blinkt sie ja auch. Die Abbildung genau dieser Verhaltensmerkmale fordern wir von der Klasse *Led* **.**



.
.
.
.

Für die Ansteuerung von LEDs gibt es im „SiSy PEC Framework“ fertige Klassen. Die Kunst besteht jetzt darin, sich diese zugänglich zu machen. In klassischen Entwicklungsumgebungen schaut man in die Hilfe, ins Lehrbuch oder in ein Tutorial, schreibt die Zeilen ab, die dort erklärt werden und darf unter Umständen nicht vergessen benötigte Pakete per *include*, *using oder import* einzubinden.

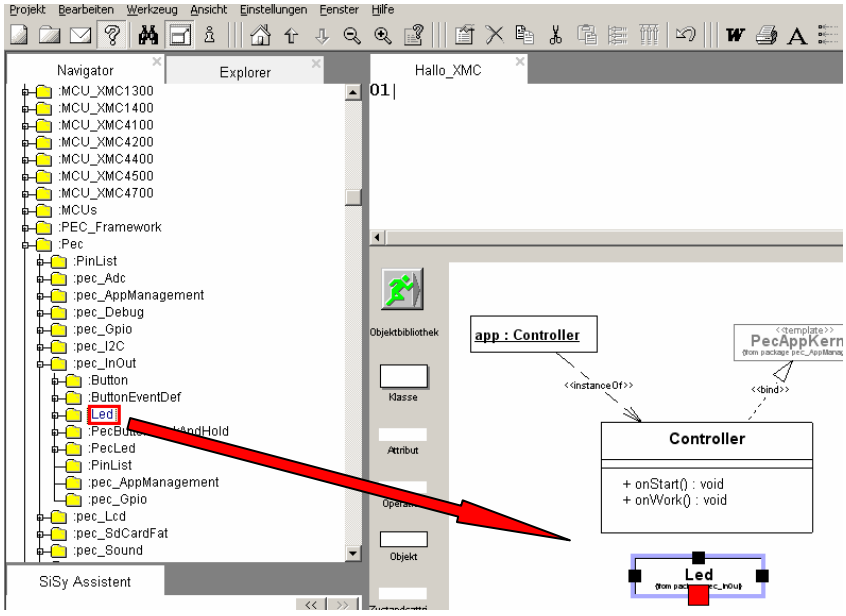
In unserem Klassendiagramm ist die Bibliothek bereits eingebunden. Sie steckt in dem Ordner *Pec*. Das Suchen geeigneter Klassen in den Bibliotheken erfolgt in SiSy am besten über den Navigator oder den Projekt-Explorer. Für die Suche im Navigator öffnen Sie das Kontextmenü des Navigators mit der rechten Maustaste. Wählen Sie den Menüpunkt „UML-Pakete“.



Abbildungen: Navigator umschalten auf „UML-Pakete“ und benötigte Klasse suchen

.
.
.
.

Wir könnten die eine einfachere Klasse benutzen, die einen simplen IO-Pin abbildet, aber was soll's … nehmen wir die sexy Led. Dazu ziehen wir die *Led* aus dem Navigator per Drag&Drop in das Klassendiagramm.

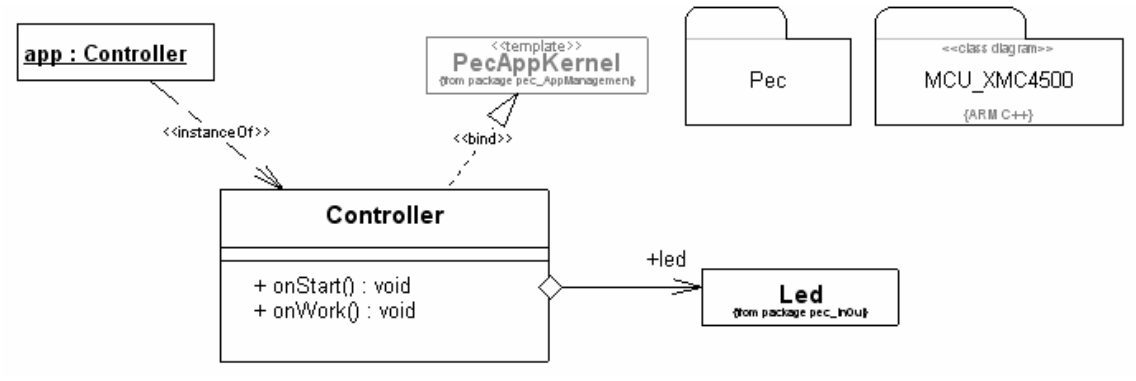


Wenn wir die Klasse *Led* hereingezogen haben, müssen wir diese mit der Anwendung verbinden. Vergleichen Sie dazu die Ausführungen zur Aggregation im Grundlagenteil dieses Lehrbuchs, Kapitel 3.1. Selektieren Sie dazu die Klasse *Controller*. Am unteren Rand erscheint eine große, rote Selektionsmarke. Das ist der Verteiler. Ziehen Sie von hier aus per Drag&Drop eine Verbindung von der Klasse *Controller* zur Klasse *Led*. Wählen Sie als Verbindungstyp die Aggregation und als Rollenbezeichner *+led*. Die Applikation hat jetzt eine *Led*.

.
.
.
.

Realisierung

Der Entwurf sollte der folgenden Darstellung entsprechen:



- die zentrale Klasse ist *Controller*
- diese verfügt über die Operationen *onStart* und *onWork*
- Die Applikationsklasse Controller ist ein *PecAppKernel*
- Das globale Objekt *app* ist die Instanz der Klasse *Controller*
- Die Klasse *Controller* verfügt über eine *Led* mit dem Attributnamen *led*
- Das Attribut *led* der Klasse *Controller* ist öffentlich

Die Aufgabenstellung fordert, dass eine LED einzuschalten ist. Diese ist bekanntlich an Pin 1.0 angeschlossen. Für die Initialisierung von Geräten steht die Operation *onStart* zur Verfügung. Selektieren Sie diese Operation und geben im Quelltexteditor folgenden Code ein:

```
Controller:: onStart
      led.config(PORT1,BIT0);
```

Beobachten Sie gleichzeitig das rechts neben dem Quelltexteditor befindliche Sequenzdiagramm. Es wird aus dem von Ihnen eingegebenen Code erzeugt. Das Sequenzdiagramm kann zum einen für Dokumentationszwecke genutzt werden zum anderen soll es gleichzeitig als Review für den Quellcode dienen.

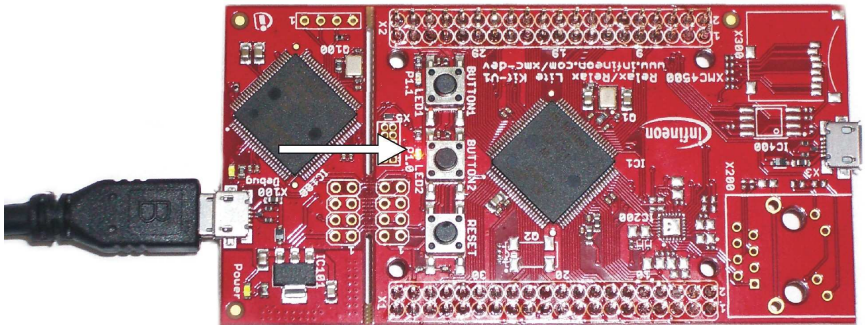
Selektieren Sie danach die Operation *onWork,* geben Sie den folgenden Code ein:

```
Controller:: onWork
      led.on();
```

Test

Übersetzen Sie das Programm. Korrigieren Sie ggf. Schreibfehler. Übertragen Sie das lauffähige Programm in den Programmspeicher des Controllers.

- Erstellen (Kompilieren und Linken)
- Brennen



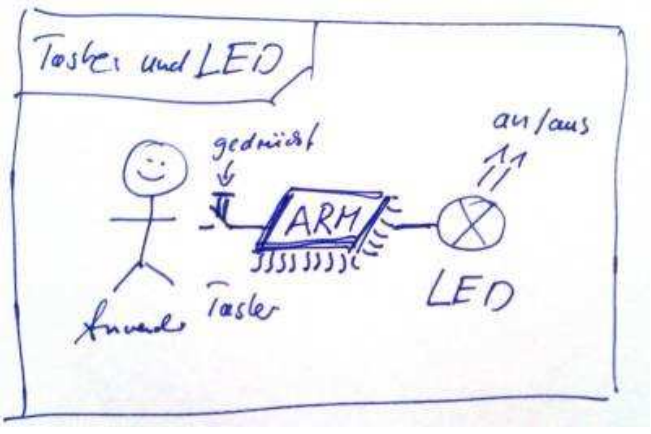
.
.
.
.

4.3 Die Klassen Button und Led

Lassen Sie uns diese Arbeitsweise vertiefen. Das zweite Beispiel in klassischem C war der intelligente Lichtschalter. Dabei sollte per Tastendruck eine LED eingeschaltet werden. Im letzten Abschnitt benutzten wir die vorhandene Klasse *Led*. Aufmerksame Beobachter haben gewiss schon die Klasse *Button* im selben Paket entdeckt.

Die Aufgabe

Es ist eine Mikrocontrolleranwendung zu entwickeln, bei der durch Drücken einer Taste eine LED eingeschaltet wird.



Vorbereitung

Falls Sie jetzt noch das Klassendiagramm geöffnet haben wählen Sie im Kontextmenü (rechte Maustaste) des Diagramms den Menüpunkt nach oben. Falls das Projekt nicht mehr geöffnet ist, öffnen Sie das SiSy UML-Projekt wieder.

Legen Sie ein neues Klassendiagramm an und wählen Sie die Sprache ARM C++. Beachten Sie die Einstellungen für die Zielplattform XMC4500 Relax Kit. Beim Öffnen des Diagramms (rechte Maustaste, nach unten) laden Sie die Diagrammvorlage für eine *PEC Applikation* und fügen das Treiberpaket für den *XMC4500* ein.

Grundlagen

Falls der Navigator nicht auf UML Pakete umgeschaltet ist, sollten Sie dies jetzt tun. Die benötigten Klassenkandidaten und die Anforderungen an diese ergeben sich aus der Aufgabenstellung:

- Led, an, aus
- Taster, ist gedrückt

.
.
.
.

Klassendiagramme sind die Konstruktionszeichnungen einer objektorientierten Anwendung.
Unter anderem können wir Aussagen aus der obigen Darstellung entnehmen.

- Die Klasse *Button* ist ein *PecButtonClickAndHold*.
- Es gibt eine Variante des Button die *low activ* arbeitet.
- Alle Button besitzen unter anderem folgende Operationen:
  - `isPressed`
  - `waitForPresson`
  - `onClick`

Damit lässt sich doch unsere Aufgabe locker lösen. Die Operation *isPressed* sollte genau das sein was wir suchen.

Ziehen Sie die Klassen *Button* und *Led* in den neuen Entwurf. Verbinden Sie diese mit der Klasse *Controller* und dem Verbindungstyp *Aggregation*.

.
.
.
.

Realisierung

Die Initialisierung erfolgt, wie gehabt, in der Operation *onStart*. Laut Schaltplan des Relax Kit ist der Taster an Pin 1.15 und die LED an Pin 1.0 angeschlossen.

```
Controller::onStart
      led.config(PORT1,BIT0);
      button.config(PORT1,BIT15);
```

Die Verarbeitungslogik erfolgt in der Operation *onWork* im Polling.

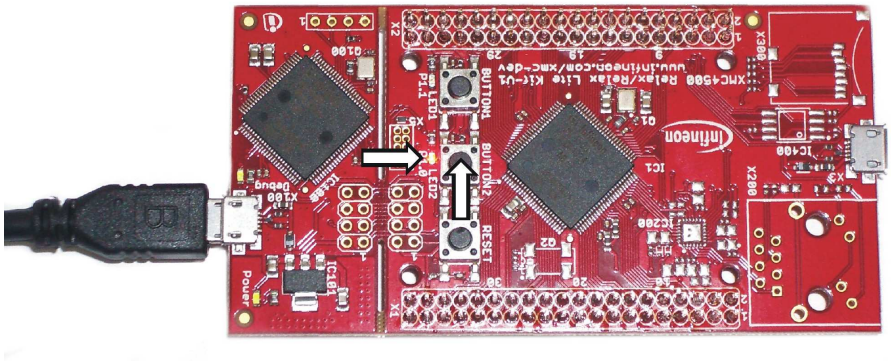
```
Controller::onWork:
```

.
.
.
.

Test

Übersetzen Sie das Programm. Korrigieren Sie ggf. Schreibfehler. Übertragen Sie das lauffähige Programm in den Programmspeicher des Controllers.

- Erstellen (Kompilieren und Linken)
- Brennen



.
.
.
.

4.4 Der SystemTick in C++

Den SysTick haben wir beim einfachen Programm bereits kennen gelernt. Dieser stellt einen einfachen Zeitgeber für das Anwendungssystem dar. Standardmäßig ist der SysTick auf *SystemCoreClock/100* konfiguriert. Damit verfügt unser System über ein vorgefertigtes 10 ms Ereignis.

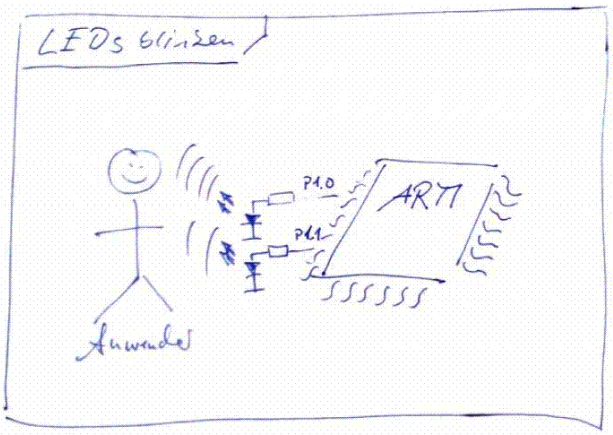
Die Aufgabe

Diese Übung wird wiederum eine einfache Verwendung der *SysTickFunction* zur Generierung zyklischer Ausgaben demonstrieren. Wir lassen die LEDs auf dem Board unterschiedlich blinken. Das folgende Blockbild verdeutlicht, welche Bausteine bei dieser Aufgabe eine Rolle spielen.

.
.
.
.

Lösungsansatz

Die Aufgabe besteht darin die zwei LED's anzusteuern. Folgen wir der objekt-orientierten Sichtweise, sind die beiden LEDs Objekte und können über eine Klasse mit dem Namen *Led* abstrahiert werden. Die Klasse *Led* soll die spezifischen Merkmale (Struktur und Verhalten) von typischen LEDs auf der Anwenderebene abstrahieren. Also fragen wir uns was die zwei LEDs denn aus Anwendersicht so tun sollen. Diese



können an oder aus sein, sie können ihren Zustand wechseln, also umschalten. Tun die LEDs das zyklisch, blinken sie.

.
.
.
.

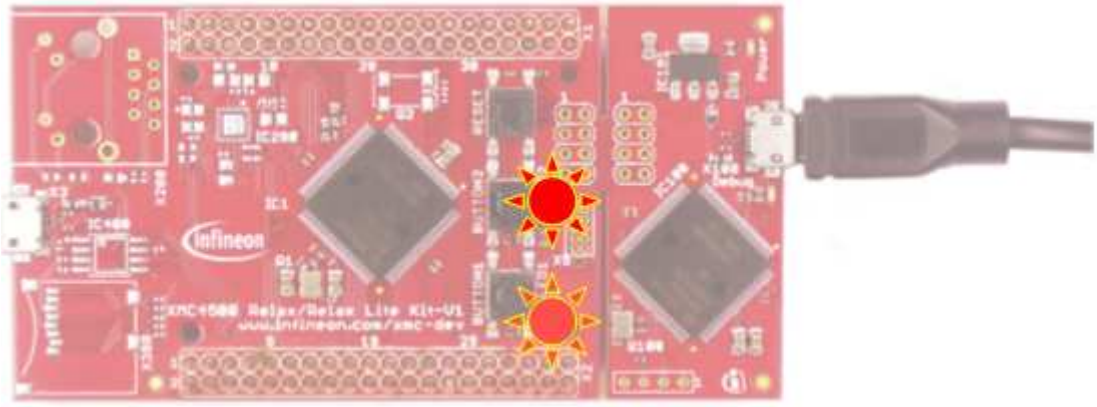
Realisierung

Den Entwurf unterziehen wir einem kurzen Review. Dann können wir mit der Realisierung beginnen. Zuerst initialisieren wir die Geräte, sprich die LEDs. Diese sind mit GPIO-Pins 1.0 und 1.1 verbunden. Die Initialisierung soll beim Start der Anwendung erfolgen.

.
.
.
.

Test

Erstellen, übersetzen und übertragen Sie das Programm. Die LEDs blinken jetzt in unterschiedlicher Geschwindigkeit.



.
.
.
.

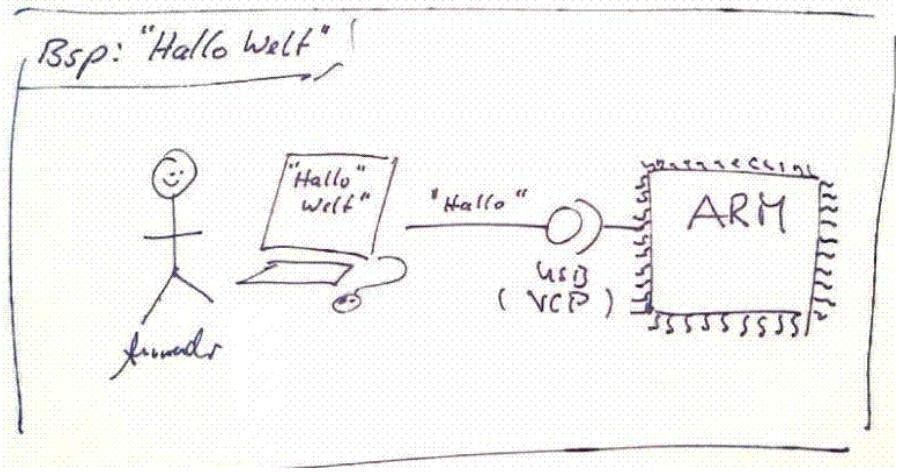
4.5 Kommunikation mit dem PC

4.5.1 Daten empfangen

Eingebettete Systeme besitzen oft eine Schnittstelle zur PC-Welt oder anderen eingebetteten Systemen. Das können ein USB-Anschluss, Ethernet, Infrarot, Bluetooth oder zum Beispiel auch WiFi sein. Eine nach wie vor oft verwendete Schnittstelle ist die gute alte UART. Obwohl diese schon recht betagt ist finden wir faktisch in jeder Controllerfamilie diese Schnittstelle wieder. Der XMC4500 verfügt über 3 universelle serielle Schnittstellen (USIC) mit je 2 Kanälen die unter anderem auch als UART bzw. USART arbeiten können. Die hohe Verfügbarkeit und die einfache Handhabung machen diese Schnittstelle sehr attraktiv. So können eingebettete Systeme über diese Schnittstelle Messwerte senden oder konfiguriert und debuggt werden.

Die Aufgabe

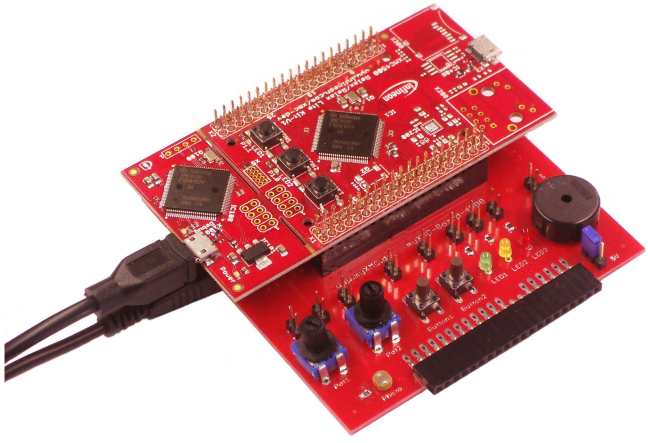
Die Aufgabe soll darin bestehen, die Zeichenkette „Hallo XMC!“ per UART an den PC zu senden. Dort werden die empfangenen Daten in einem Terminal-Programm angezeigt.



Vorbereitung

Falls Sie jetzt noch das Klassendiagramm geöffnet haben, wählen Sie im Kontextmenü des Diagramms den Menüpunkt nach oben. Falls das Projekt nicht mehr geöffnet ist, öffnen Sie das SiSy UML-Projekt wieder.

Legen Sie ein neues Klassendiagramm an und wählen Sie die Sprache ARM C++. Beachten Sie die Einstellungen für die Zielplattform XMC4500 Relax Kit. Beim Öffnen des Diagramms laden Sie die Diagrammvorlage für eine PEC Applikation und weisen das Treiberpaket für den XMC4500 zu.



Grundlagen

Da heutige Rechner kaum noch über eine klassische RS232-Schnittstelle (COM) verfügen, benutzen wir eine USB-UART-Bridge. Auf dem Erweiterungsboard für das XMC4500 Relax Kit ist dafür eine entsprechende Nachrüstung vorgesehen.

.
.
.
.

Damit lautet unsere Konfiguration: Pin 2.14 ist die Sendeleitung TxD und Pin 2.15 ist die Empfangsleitung RxD. Die Geschwindigkeit für die Datenübertragung legen wir mit 9600 Baud fest.

Zusätzlich ist die USB-UART-Bridge mit dem PC zu verbinden. Dazu benötigen Sie ein Mini USB-Kabel. Über die USB-Verbindung zum integrierten J-Link wird der Controller programmiert und das System mit Spannung versorgt.

Entwurf

Für die zu entwerfende Anwendung wollen wir vorhandene Bausteine (Klasse/Templates) für die UART nutzen. Dazu schauen wir uns in den UML Paketen des Frameworks um.

.
.
.
.

Realisierung

Nachdem wir die benötigten Bausteine (*PecUart, baudrate9600, usart1c0_P2.14/P2.15_xmc4500*) aus der Bibliothek bezogen und zu einer Komponente (Terminal) zusammengesetzt haben, wurde diese im System (*Controller*) als Instanz (terminal) aggregiert. Jetzt können wir die Instanz der Komponente benutzen.

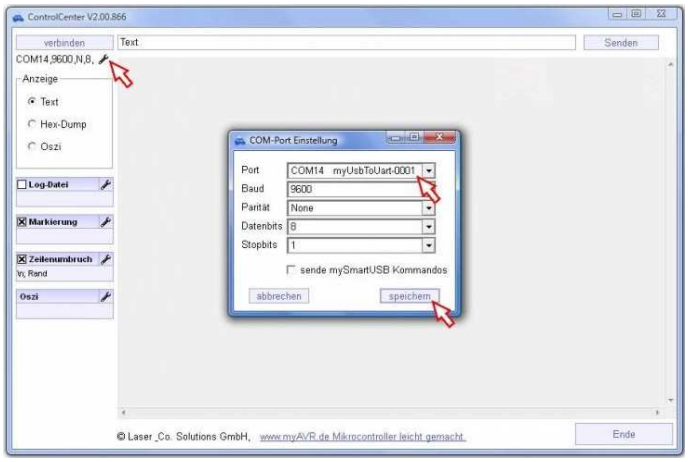
.
.
.
.

Test

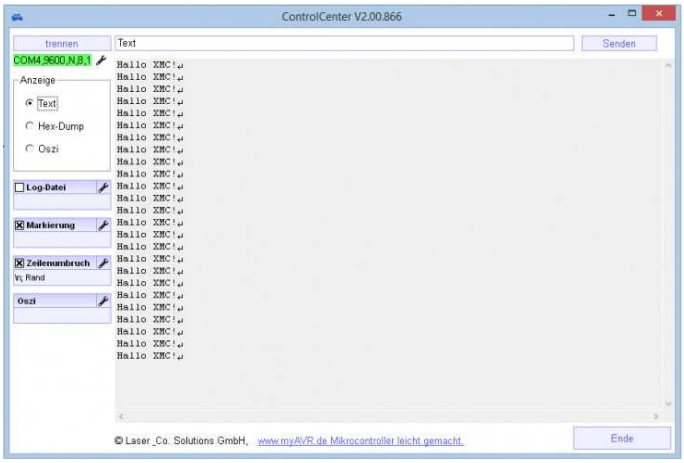
Übersetzen Sie das Programm. Korrigieren Sie ggf. Schreibfehler. Übertragen Sie das lauffähige Programm in den Programmspeicher des Controllers und Starten das Werkzeug „Controlcenter“.

- Erstellen (Kompilieren und Linken)
- Brennen
- Menü Werkzeuge → Controlcenter

Stellen Sie im Controlcenter die Parameter für die Verbindung mit dem Board ein. Achten Sie auf den richtigen COM-Port (COM-Port mit UsbToUart) und die korrekte Baudrate (der COM-Port ist ggf. dem Geräte-Manager zu entnehmen).



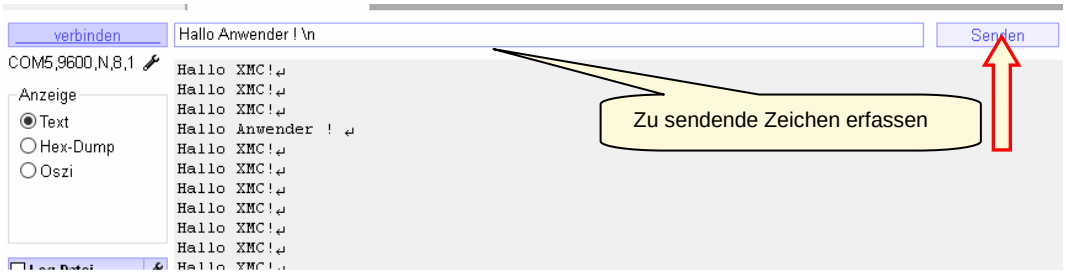
Jetzt können Sie die gewünschten Daten vom Controller empfangen.



4.5.2 Daten senden

Als Erweiterung der Anwendung sollen Zeichen vom PC empfangen und wieder zurückgesendet werden. Dabei ist zu beachten, dass nur dann, wenn Zeichen verfügbar sind, diese auch von der UART abgeholt werden dürfen.
Mit den Operationen *dataAvailable* und *readByte* stellt die UART die benötigte Funktionalität zur Verfügung.

.
.
.
.



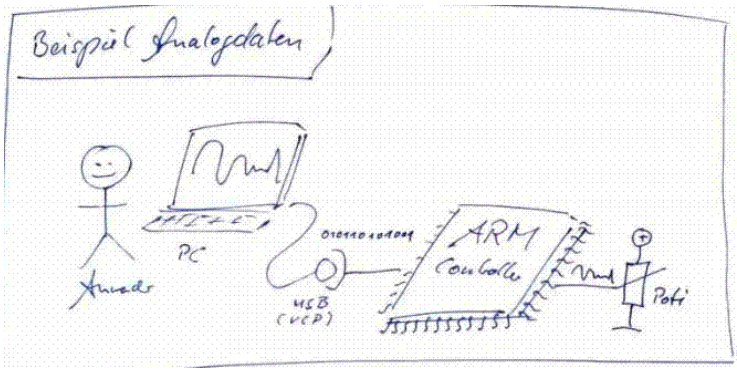
.
.
.
.

4.6 Analogdaten erfassen

Für eingebettete Systeme ist das Erfassen und die Verarbeitung analoger Daten oft essentiell. Das Framework stellt Klassen und Templates zur Verfügung, mit denen sich eine Vielzahl von Aufgabenstellungen rings um den Analog-Digital-Converter mit wenigen Handgriffen erledigen lassen. Viele Problemstellungen, bei denen ein Analogwert erfasst, verarbeitet und daraufhin etwas, was auch immer, getan werden soll, sind mit einer Auflösung von 8 Bit absolut hinreichend zu lösen. Somit soll in diesem Beispiel ein 8-Bit Wert erfasst und per UART an das Controlcenter gesendet werden. Der Charm dabei ist, dass sich 8-Bit Werte auch sehr reibungslos per UART übertragen lassen. In einer Erweiterung der Übung übertragen wir den Wert nicht direkt (binary raw data) sondern wandeln die Werte in lesbare Zeichen (ASCII) um und senden diese als Zeichenkette an den PC.

Die Aufgabe

Es soll eine Mikrocontrolleranwendung entwickelt werden, bei der die Analogwerte von einem Potentiometer digitalisiert und an den PC gesendet werden. Die Auflösung der Analog-Digital-Wandlung soll 8 Bit betragen.



.
.
.
.

Für die Erfassung des Analogwertes verbinden wir eines der Potentiometer auf dem Erweiterungsboard mit ADC0, Chanel 0 als Alternativfunktion des Pin 14.0 (vgl. ADC Pin Mapping).

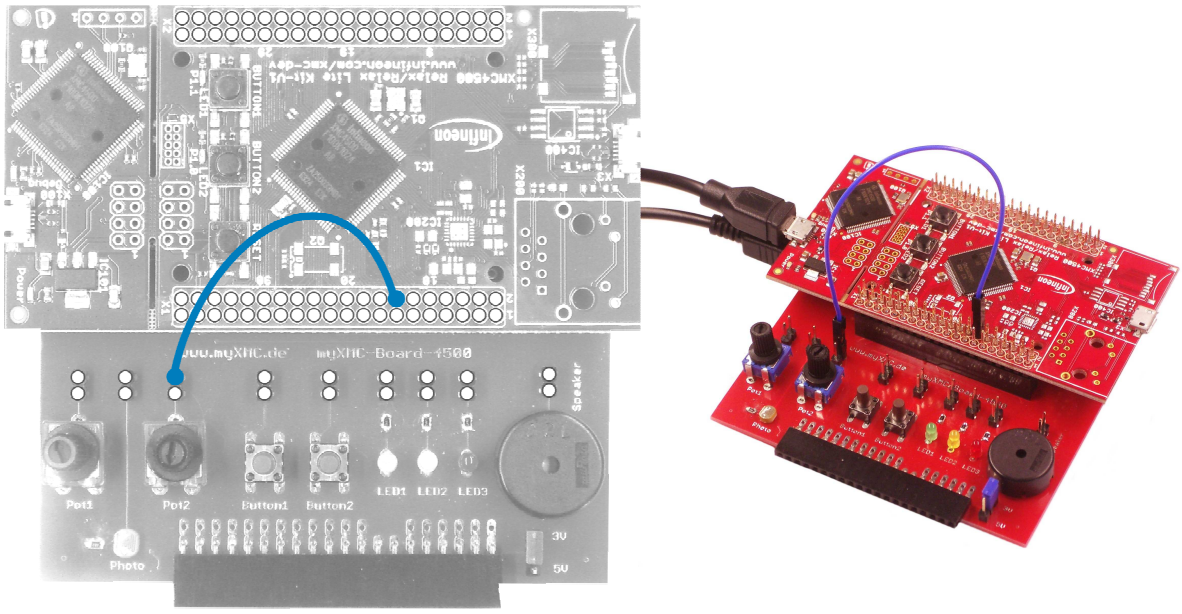
.
.
.
.

Test

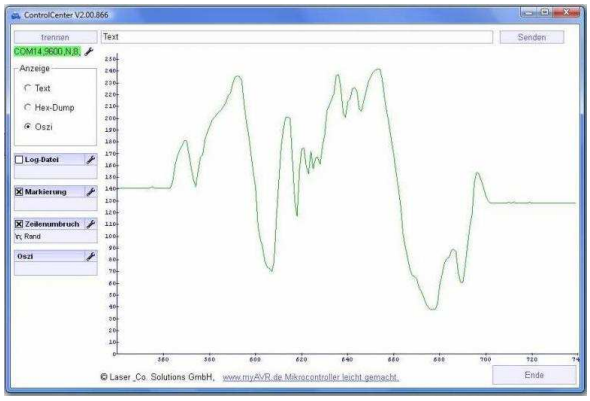
Übersetzen Sie das Programm. Übertragen Sie das lauffähige Programm in den Programmspeicher des Controllers.

- Erstellen (Kompilieren und Linken)
- Brennen
- Verbindungen herstellen
- ControlCenter starten

Verbinden Sie die USB-UART-BRIDGE auf dem Erweiterungs-board mit dem PC und patchen Sie das Potentiometer auf Pin 14.0 (vgl. Bild).



Starten Sie das ControlCenter und beachten die korrekten Einstellungen für den COM-Port. Nachdem Sie „verbinden“ aktiviert haben, stellen Sie die Ansicht auf „Oszi“ um. Sie können über die Optionen für die Oszi-Ansicht verschiedene Darstellungsvarianten auswählen. Nun können Sie jede Veränderung am Potentiometer auf dem Bildschirm verfolgen.



.
.
.
.

Literatur und Quellen

SiSy-Homepage
www.sisy.de

myMCU-Homepage
www.mymcu.de

Firmen-Homepage von Infineon
www.infineon.com

Datenblätter der XMC-Controller
www.infineon.com/cms/de/product/microcontrollers/

Eine Online-Dokumentation des GNU-Assemblers
www.delorie.com/gnu/docs/binutils/as.html#SEC_Top
www.delorie.com/gnu/docs/binutils/as_392.html

Mikrocontroller-Foren
www.roboternetz.de
www.mikrocontroller.net

Online-Lexikon
http://wiki.mikrocontroller.net/
http://www.roboternetz.de/wissen

Wolfgang Trampert
AVR – RISC Mikrocontroller
2. Auflage, Franzis, 2003

Bernd Österreich
Objektorientierte Softwareentwicklung. Analyse und Design mit UML 2
7. Auflage, Oldenbourg, 2004

Helmut Balzert
Lehrbuch der Software-Technik
Spektrum Akademischer Verlag, 2008

Peter Scholz
Softwareentwicklung eingebetteter Systeme
Springer-Verlag Berlin Heidelberg 2005

Christoph Kecher
UML 2 – das umfassende Handbuch
Galileo Press, Bonn 4. Auflage 2011

Tim Weilkiens
Systems Engineering mit SysML/UML
dpunkt.verlag GmbH, 1. Auflage 2006

