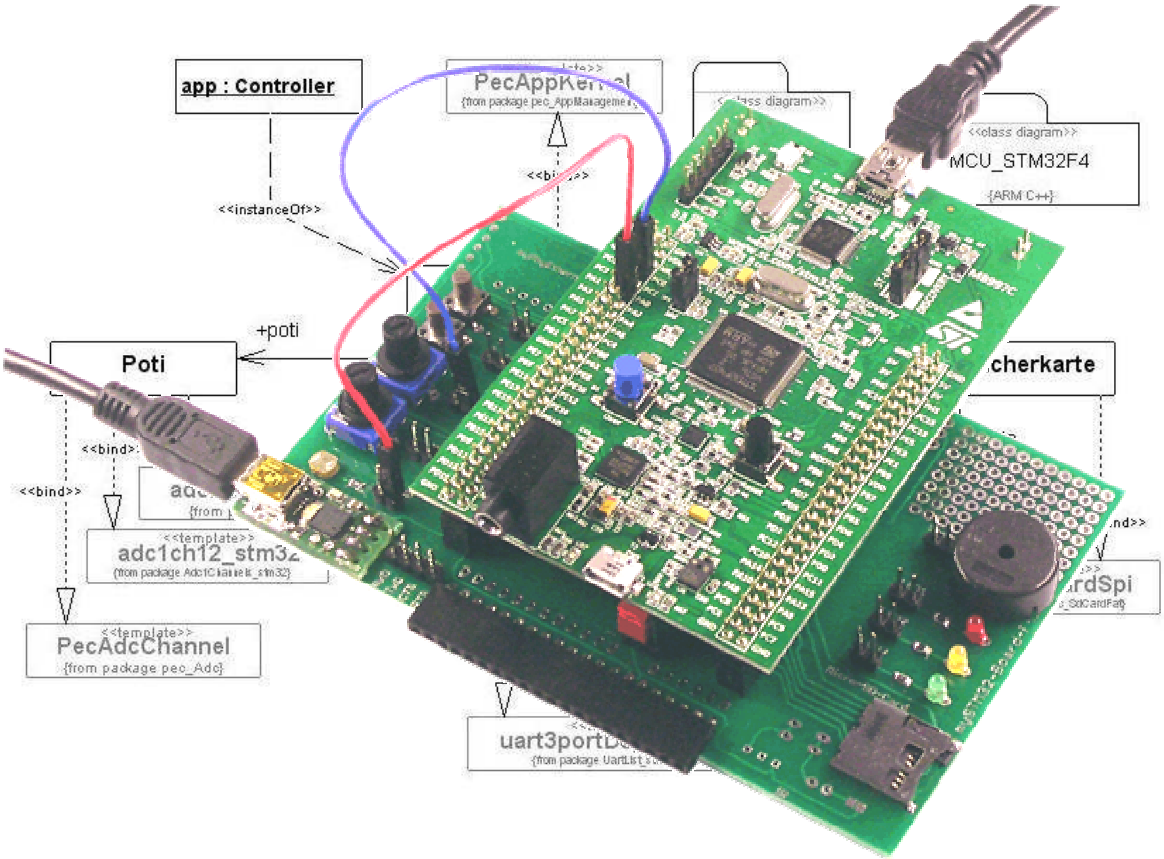
Sven Löbmann
Toralf Riedel
Alexander Huwaldt

# mySTM32 Lehrbuch

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




*Leseprobe*

5. Auflage: März 2016


www.laser-co.de
www.mySTM32.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 Programmierkenntnisse einer beliebigen Sprache verfügen. Es ist kein C++ oder ARM-Programmier-Lehrbuch im engeren Sinne und erhebt daher keinen Anspruch auf Vollständigkeit oder Allgemeingültigkeit in diesen Bereichen. Hier soll sich speziell mit ausgewählten Aspekten für den einfachen Einstieg in die objektorientierte Programmierung von ARM-Mikrocontrollern auseinandergesetzt werden.

Bevor wir uns dem STM32 zuwenden, möchte ich die Aufmerksamkeit auf die Objektorientierung lenken. Dieser Denkansatz 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 ///////////////////////
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
GPIO_InitTypeDef  GPIO_InitStructure;
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
GPIO_Init(GPIOD, &GPIO_InitStructure);
GPIO_SetBits(GPIOD,GPIO_Pin_12);
...
```

```
// objektorientierte Schreibweise //////////////////
Led ledGreen;
ledGreen.config(GPIOD,12);
ledGreen.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 diversen Bedenkenträgern und vielleicht auch 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*

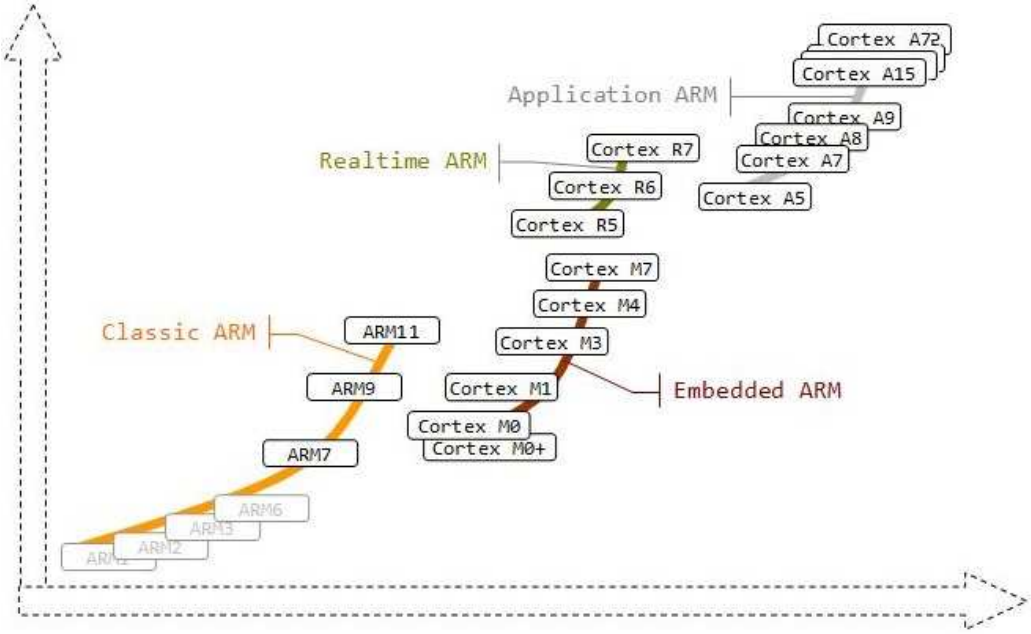
Weitere Informationen und Beispiele finden Sie im begleitenden Online-Tutorial zu diesem Lehrbuch unter www.mySTM32.de.

Wir wünschen Ihnen viel Erfolg beim Studium.

# 1 Einführung

## 1.1 ARM-Architektur

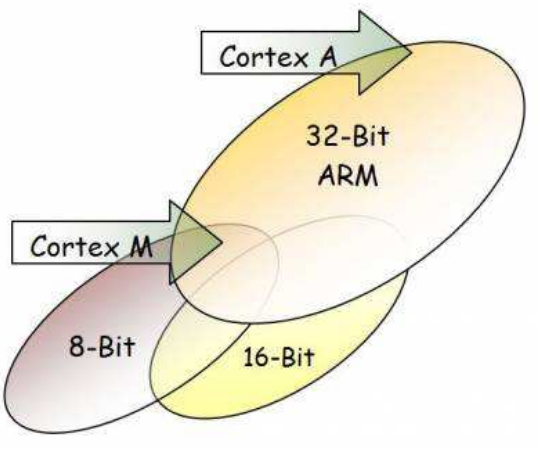
Die 1991 erstmals vorgestellte 32-Bit Architektur der Firma Advanced RISC Machines Ltd. aus Cambridge bildet die Basis für jeden ARM-Prozessor. Im Laufe der Jahre hat sich die ursprüngliche ARM-Architektur rasant entwickelt. Die neueste Version des ARM bildet die ARMv8 Architektur. Diese zeigt schon deutlich in Richtung 64-Bit Architekturen. Vielleicht werden Sie sich jetzt fragen, wozu Sie solche Leistung brauchen. Aber selbst Hobbyprojekte wie Quadcopter oder ein Hexapod können recht schnell an die Leistungsgrenze eines 8/16-Biter stoßen. Preis und Energieeffizienz sind längst keine Argumente mehr gegen den Einsatz eines ARM.



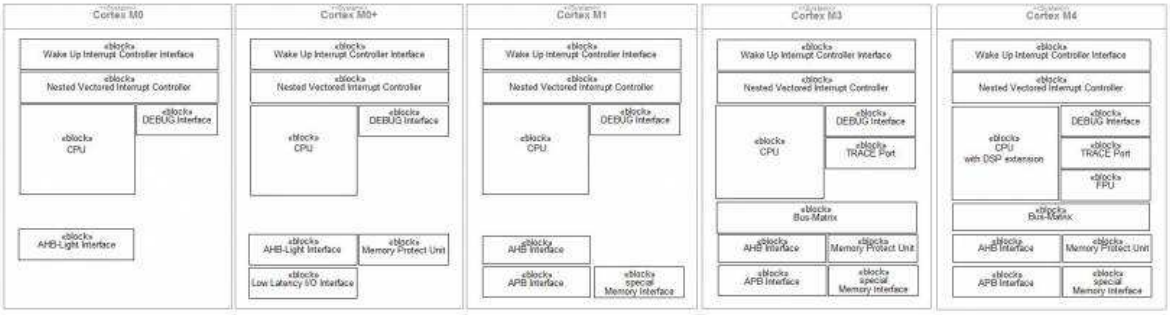

ARM Controller sind dem Wesen nach RISC (Reduced Instruction Set Computer) und unterstützen die Realisierung einer breiten Palette von Anwendungen. Inzwischen gilt ARM als führende Architektur in vielen Marktsegmenten und kann getrost als Industriestandard bezeichnet werden. Den Erfolg der ARM-Architektur kann man sehr gut an den aktuellen Trends bei Smartphone, 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.

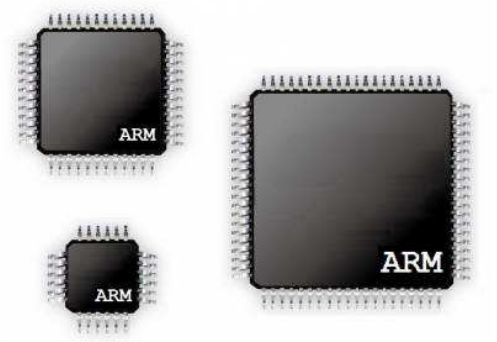
### 1.1.1 Cortex-M

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 setzen wird. Die folgende (mit Sicherheit nicht vollständige) Darstellung soll die Skalierung der Cortex-M Familie verdeutlichen:



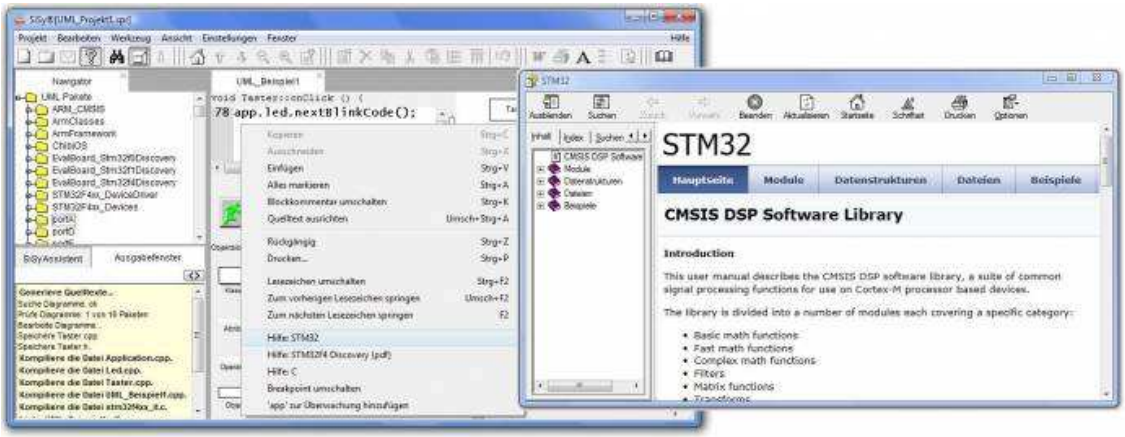

Der Formfaktor dieser 32-Bit Controller lässt sich durchaus mit den größeren Mega- und X-Mega Controllern der AVR-Familie von Atmel vergleichen. Für den blutigen Anfänger unter den Bastlern könnte jedoch die SMD-Bauweise eine nicht unerhebliche Einstiegshürde darstellen.
Die Standardisierung des ARM betrifft neben der Hardware, auch die gemeinsamen Aspekte aller ARM-Applikationen. Die ARM-Lizenznehmer halten sich strikt an die Architektur des ARM-Kerns und fügen nur ihre spezifische Peripherie hinzu. Alle den Kern betreffenden Softwarefunktionen lassen sich somit herstellerübergreifend standardisieren.

### 1.1.2 CMSIS

CMSIS - Cortex Microcontroller Software Interface Standard, 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 SiSy steht Ihnen eine umfangreiche Hilfe zum CMSIS zur Verfügung.



### 1.1.3 STM32 Peripherie Treiber

Es handelt sich hier um ein komplettes Firmware-Paket, bestehend aus Gerätetreiber für alle Standard-Peripheriegeräte der STM32F4 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 deren Bitkombinationen zu kennen. Es spart viel Zeit, die sonst bei der Codierung anhand des Datenblattes aufgewendet werden muss. Die STM32F4xx Peripherie Bibliothek umfasst 3 Abstraktionsebenen und beinhaltet:

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

Sie finden diese Bibliotheken als zip-Datei auf der STM32F4-Discovery Webseite unter

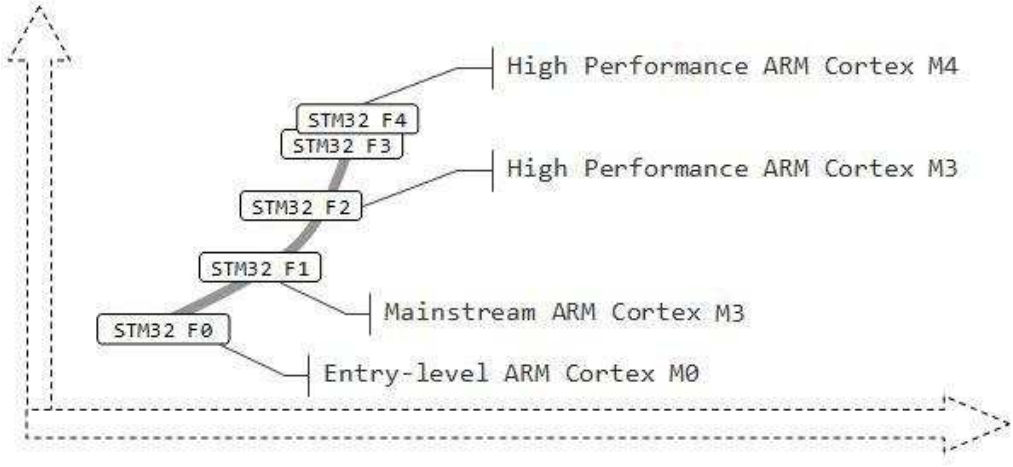
www.st.com/stm32f4-discovery

Während der Installation der im Abschnitt 1.3 vorgestellte Entwicklungsumgebung werden die Bibliotheken für das CMSIS und die Peripherie-Treiber bequem mit installiert.

## 1.2 STM32 Hardware

Die Firma ST-Microelectronics bietet in ihrem breiten Produktspektrum auch Mikrocontroller auf der Basis der ARM Cortex-M Architektur an. Dabei lassen sich vier Anwendungsfelder erkennen, auf die die STM32-Familie abzielt:

- Entry-Level-MCU, STM32-F0, Cortex-M0, bisher 8/16-Bit Domäne
- Mainstream-MCU, STM32-F1, Cortex-M3, bisher 16-Bit Domäne
- High-Performance MCU, STM32-F2/3/4, Cortex-M3/4, 32-Bit Domäne
- Spezialanwendungen, STM32-W/L, Cortex-M3



Die Firma ST-Microelectronics bietet, wie jeder Hersteller, für verschiedene Anwendungsfälle Referenzhardware zum Kennenlernen und Testen an. Beispiele für solche STM32-Evaluierungsbords sind:

- STM32F0-Discovery, Entry Level MCU
- STM32VL-Discovery , Mainstream MCU
- STM32F4-Discovery, High Performance MCU
- STM32W RF-Control-Kit, Spezialanwendung

Alle weiteren Ausführungen in diesem Lehrbuch beziehen sich auf das Board STM32F4-Discovery von der Firma ST-Microelectronics.

### 1.2.1 STM32F4-Discovery

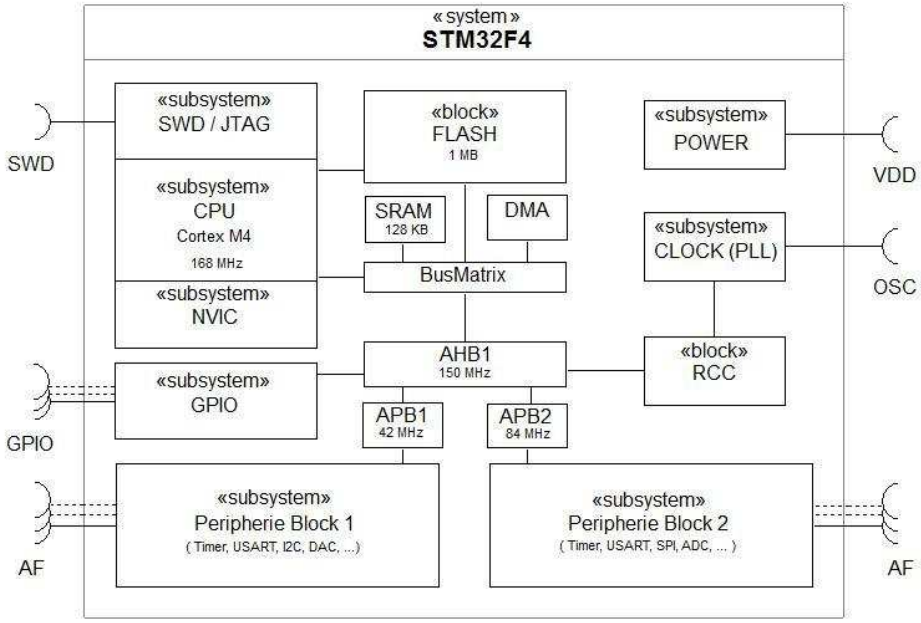
Das „STM32F4-Discovery“ ist derzeit eines der neuesten Evaluierungsboards von ST-Microelectronics. Es ermöglicht dem Anwender besonders die Hochleistungseigenschaften des Cortex-M4 zu erkunden und trotzdem Anwendungen einfach zu entwickeln. Mit der im nächsten Kapitel vorgestellten Erweiterungsplatine „mySTM32-Board-F4D“ verfügen der Anfänger und der Umsteiger über alles, was für den schnellen Einstieg in die Programmierung von STM-Controllern, aber auch für anspruchsvolle Anwendungen erforderlich ist.

Eigenschaften:

- Mikrocontroller STM32F407VGT6 im LQFP100 Gehäuse mit
  - 32 Bit ARM Cortex-M4 Kern
  - 1 MByte FLASH
  - 192 KByte RAM
  - 168 MHz
- ST MEMS 3-Achs Beschleunigungssensor, Typ versionsabhängig
- MP45DT02: ST MEMS ungerichtetes digitales Mikrofon
- CS43L22: Audio Digital-Analog-Konverter mit integriertem Verstärker
- 8 LEDs
  - 1 für USB Kommunikation
  - 1 Power-LED für 3,3 V
  - 4 durch den Anwender nutzbare LEDs
  - LEDs für USB on-the-go
- 2 Taster
  - 1 für Reset
  - 1 frei verfügbar für Anwender
- onBoard Programmer ST-LINK V2

Die Bestückung mit simplen Ein- und Ausgabegeräten ist auf dem Board mit einem Taster und vier frei verfügbaren LEDs doch eher spartanisch gehalten. In

diesem Punkt bringt jedoch das Erweiterungsboard mySTM32-Board-F4D genügend Abhilfe. Interessant für anspruchsvollere Anwendungen sind auf dem STM32F4-Discovery natürlich die Audio-Features und der Lagesensor. Hervorzuheben ist ebenfalls der integrierte ST-LINK/V2 Programmer. Mit diesem können über den vorhandenen SWD Pfostenstecker (Serial Wire Debugging) auch andere STM32 programmiert und debuggt werden. Doch zunächst einige wichtige Aspekte zum Inneren des ARM Controllers.




Für das Verständnis des ARM Cortex Controllers sind einige grundlegende Strukturmerkmale wichtig. Neben dem Programmier- und Debug-Interface, den getrennten Programm- und Datenspeichern sind für den Anfänger, aber auch für Umsteiger vom AVR, folgende Bausteine von besonderer Bedeutung:

- **RCC (Real-Time Clock Control)**
  Dieser Baustein liefert den Takt für jede einzelne Komponente, die benutzt werden soll. Im Gegensatz zum AVR ist faktisch die gesamte Peripherie nach dem RESET zunächst ausgeschaltet. Jeder einzelne Baustein muss durch Zuweisung eines Taktsignals erst eingeschaltet werden, bevor man diesen initialisieren und benutzen kann.

- **AHB (Advanced High-performance Bus)**
  ARM Controller besitzen mindestens einen sehr schnellen Haupt-Bus mit Busmatrix. Über diesen leistungsfähigsten Bus im System werden ausgewählte extrem schnelle Bausteine, wie die GPIO-Ports und die Peripherie, über ihre eigenen Bussysteme angesprochen. Kleinere Cortex-M verfügen über eine Light-Variante des AHB, größere können auch mal zwei oder drei davon haben (AHB1, AHB2, AHB3).

- **APB (Advanced Peripheral Bus)**
  Die Peripherie, wie Timer, ADC, USART usw. werden über ein eigenes Bus-Interface angesprochen. Die gerätespezifische Nutzung von Port-Pins wird als *alternativ function* (AF) bezeichnet. Je nach Geräteklasse sind diese einem schnellen oder auch langsameren Peripherie-Bus zugeordnet. Mit dem gesamten System von Busmatrix, AHB und APB ist es möglich, recht flexibel einzelne Geräte auf sehr verschiedene Pins des Controllers aufzuschalten.

- **NVIC (Nested Vectored Interrupt Controller)**
  Die Interrupts des 32-Bit ARM sind gegenüber dem AVR nüchtern als erwachsen zu bezeichnen. Was jedoch auch deren Nutzung für den Programmierer nicht unbedingt einfacher macht. Der NVIC ist die Schaltstelle für alle Interrupts und muss vom Programmierer in Kombination mit den Konfigurationen von RCC, AHB, APB und der Ereignisquelle sowie der Programmierung der ISR sauber gehandhabt werden.

Diese Bausteine werden öfter eine Rolle spielen. Es ist einfach im Sinne des Lernens durch Wiederholung zweckmäßig, schon jetzt davon gehört zu haben.

### 1.2.2 mySTM32-Board-F4D

Das *mySTM32-Board-F4D* fungiert als Add-On und ermöglicht in einfacher Art und Weise die Funktionen des *STM32F4-Discovery* zu erweitern. Analoge und digitale Ein- und Ausgabegeräte sowie eine USB-USART Bridge für die Kommunikation mit dem PC komplettieren das Evaluierungsboard STM32F4-Discovery. Darüber hinaus sind weitere optionale Schnittstellen implementiert, so z.B. für Infrarot-Sender und -Empfänger. Zusätzlich verfügt dieses Add-On über eine Schnittstelle für myAVR Produkte. Somit bietet Ihnen das mySTM32-Board-F4D die Chance, die neue 32-Bit Technologie in Kombination mit vorhandenen myAVR Produkten einzusetzen.

Das mySTM32-Board-F4D ist besonders darauf ausgelegt, Kennern und Anwendern 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.

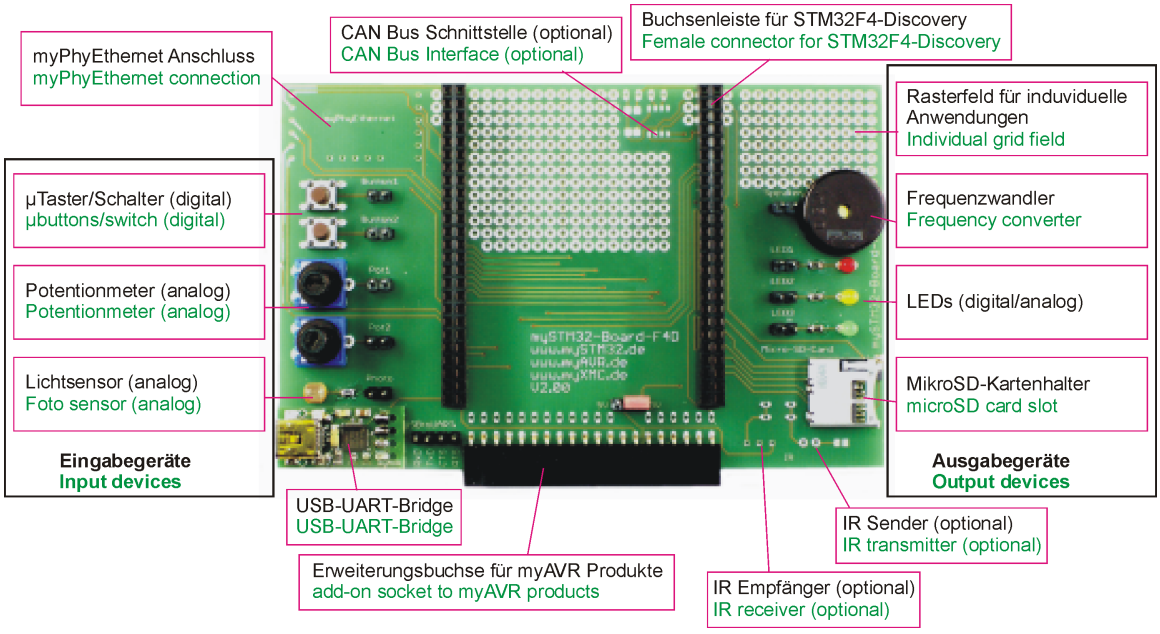



***Abbildung: mySTM32-Board-F4D***

Eigenschaften:

- Schnittstelle für STM32F4-Discovery
- Schnittstelle für myAVR Produkte
- typische Ein- und Ausgabegeräte (Taster, LEDs, usw.)
- analoger Fotosensor zum Experimentieren mit unterschiedlichen Helligkeitsgraden
- MicroSD-Kartenhalter
- Raster für flexible Anwendung (2,54 mm)
- USB-UART-Bridge
- optionale Infrarot Schnittstelle (Sender und Empfänger)
- optionale Schnittstelle CAN Bus

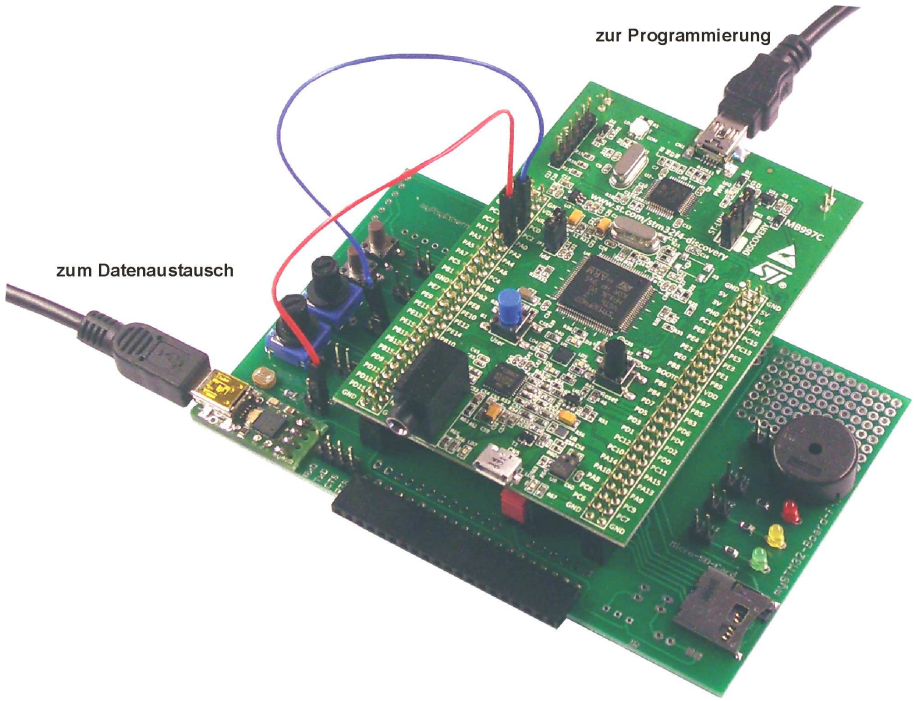



***Abbildung: Anwendungsbeispiel STM32F4-Discovery mit mySTM32-Board-F4D***

## 1.3 Entwicklungsumgebung SiSy

SiSy ist die Abkürzung für Simple System. Dabei steht System dafür, dass Systeme, egal ob klein, mittel oder groß, strukturiert und methodisch mit standardisierten Darstellungsmitteln konstruiert werden. Simple steht für eine einfache Vorgehensweise und übersichtliche Darstellung. SiSy bildet die Darstellungsmittel zur Konstruktion eines Systems individuell und aufgabenspezifisch ab. Das bedeutet, dass für jede spezifische Konstruktionsaufgabe auch spezielle Darstellungstechniken zur Verfügung stehen. Die Art der mit SiSy zu konstruierenden Systeme kann sehr vielfältig sein. Die Einsatzmöglichkeiten reichen von der Konstruktion von Softwaresystemen für Mikrocontroller über Datenbanklösungen auf Arbeitsstationen oder Servern bis hin zu betriebswirtschaftlichen Managementsystemen. SiSy ist ein allgemeines Modellierungswerkzeug für beliebige Systeme.

### 1.3.1 Grundaufbau des Entwicklungswerkzeuges

Schauen wir uns als Nächstes kurz in der Entwicklungsumgebung SiSy STM32 um. SiSy STM32 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.

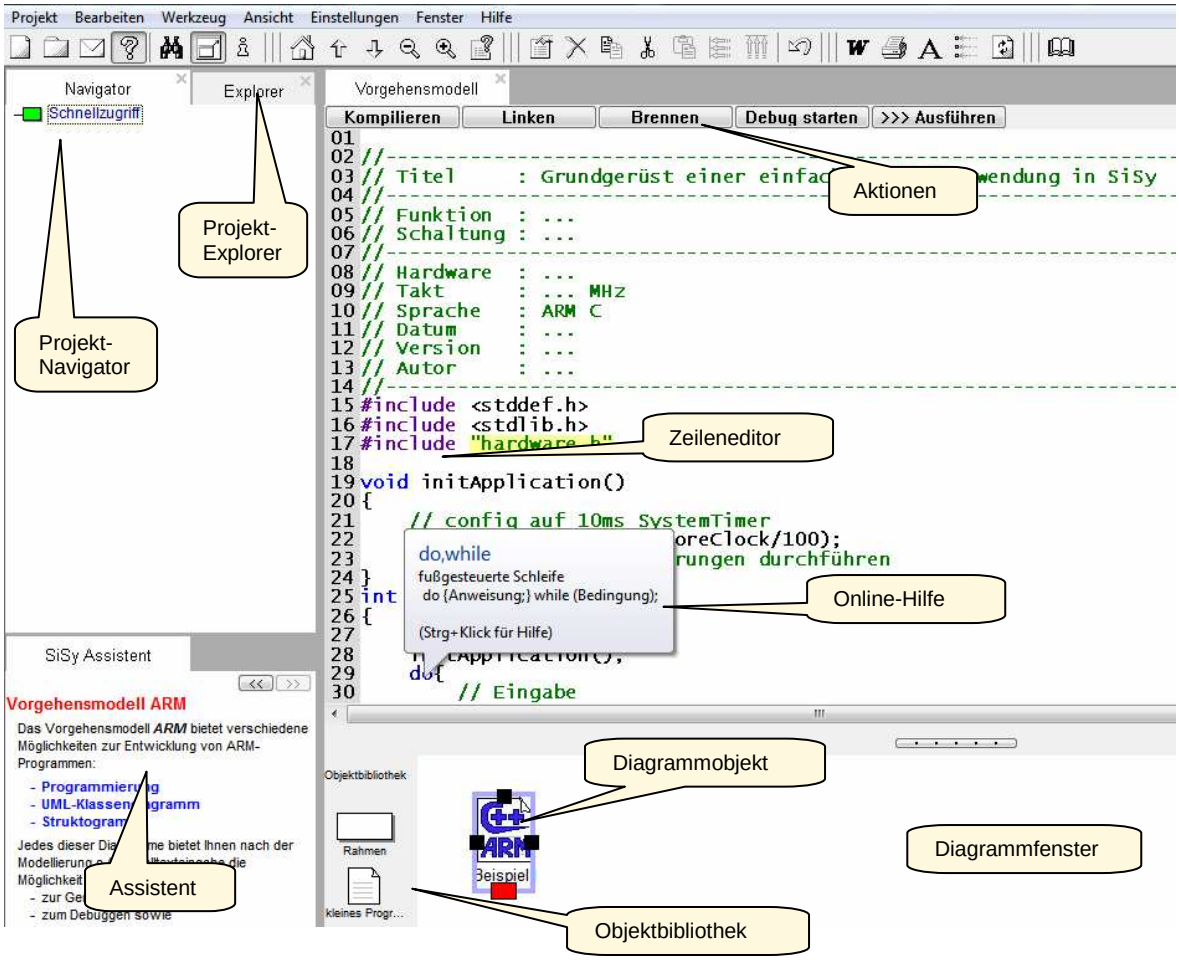



***Abbildung: Bildschirmaufbau der Entwicklungsumgebung SiSy***

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

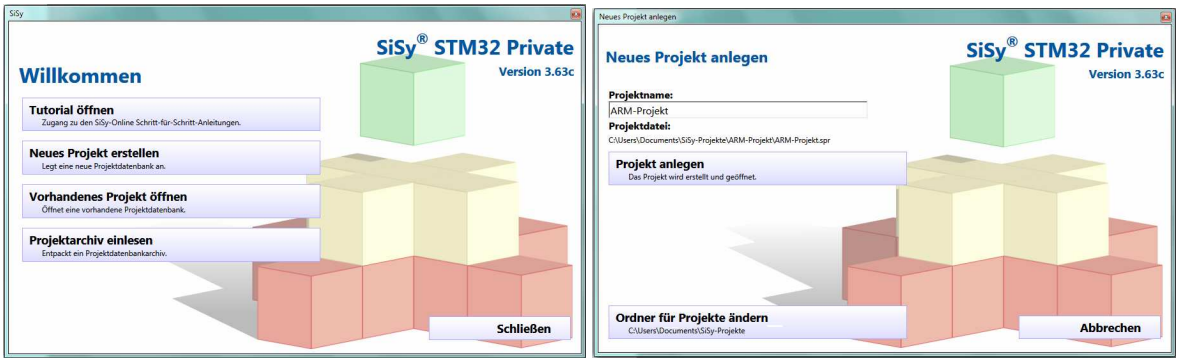
sowie auf den Programmspeicher des Mikrocontrollers übertragen wurde, muss die letzte Anzeige in Abhängigkeit der Konfiguration folgenden bzw. ähnlichen Inhalt haben:

```
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***

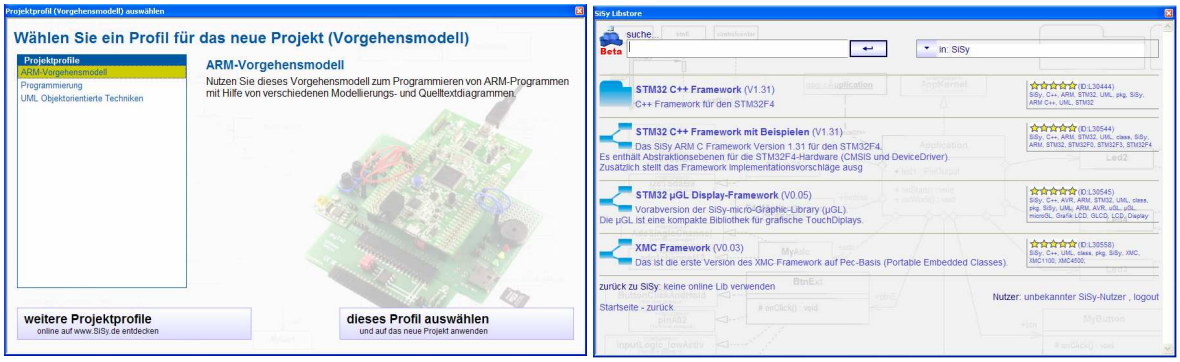
### 1.3.2 Grundstruktur einer ARM Anwendung

Die erste praktische Übung soll darin bestehen, dass ein ARM-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 Controllers. Dafür muss SiSy STM32 gestartet werden und die Experimentierhardware angeschlossen sein. Legen Sie ein neues Projekt mit dem Namen „ARM-Projekt“ an.




***Abbildungen: SiSy STM32 starten und Projekt anlegen***

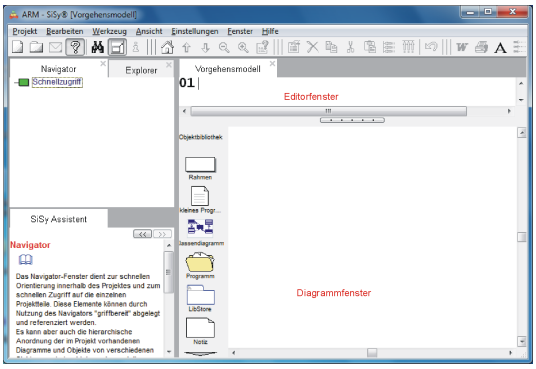
Wählen Sie das ARM-Vorgehensmodell aus. Damit sind alle wichtigen Einstellungen für das Projekt und die darin enthaltenen Übungen als Default-Werte gesetzt. Nach Auswahl des Vorgehensmodells öffnet SiSy LibStore und bietet vorhandene Vorlagen für die weitere Arbeit an.

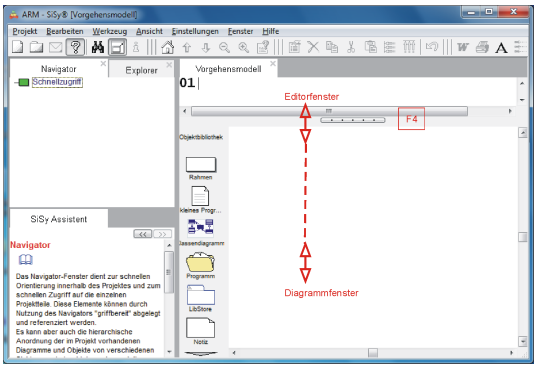



***Abbildungen: Vorgehensmodell auswählen, Anzeige SiSy LibStore***

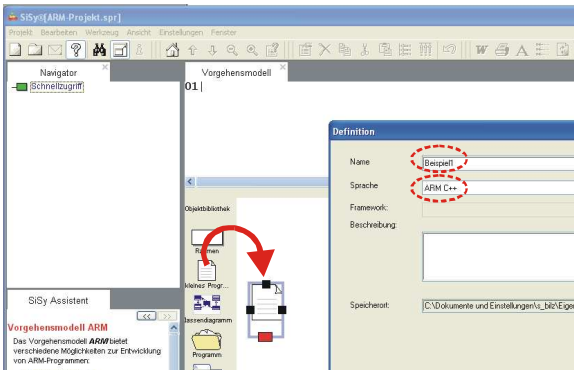
Wir brauchen für die ersten Schritte noch keine UML Bibliotheken. Damit können wir „zurück zu SiSy: keine online Lib verwenden“ aktivieren. Sie erhalten somit ein leeres Projekt.

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.

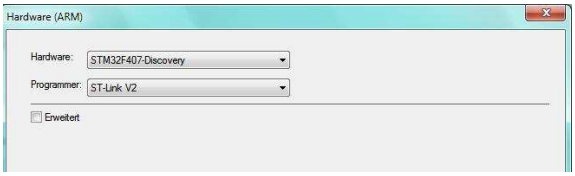


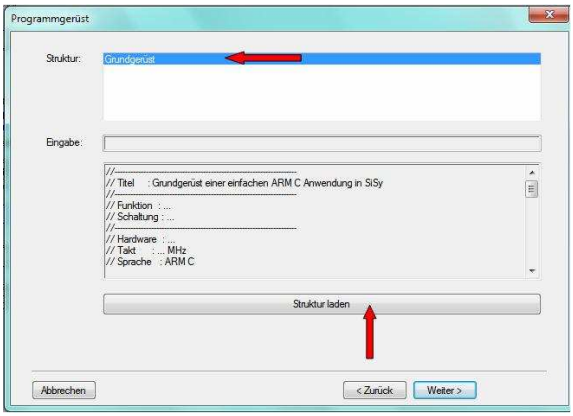

Legen Sie Ihr erstes kleines Programm an, indem Sie das entsprechende Objekt aus der Objektbibliothek per Drag&Drop in das Diagrammfenster ziehen. Geben Sie dem Programm den Namen „Beispiel1“ und überprüfen Sie, ob die Zielsprache auf ARM C++ eingestellt ist.




Im nächsten Schritt wird die Hardware ausgewählt. Wir benutzen das Entwicklerboard STM32F407-Discovery, und den Programmer „ST-Link V2“.

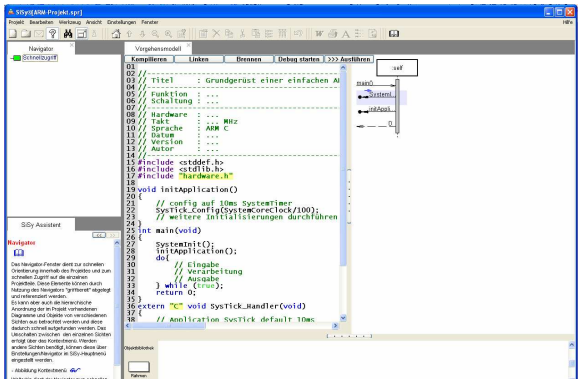



Bevor wir uns dem Stress aussetzen fast 40 Zeilen Programmcode abzutippen, benutzen wir lieber eines der Features von SiSy, die Programmgerüste. Selektieren Sie das Grundgerüst für ein ARM C++ Programm und laden die Struktur über die Schaltfläche „Struktur laden“. Aber Achtung, nicht mehrfach ausführen. SiSy fügt die ausgewählten 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 Diagramm-fenster von SiSy, im Editorfenster wird der geladene Quellcode angezeigt.



***Abbildungen: typisches Vorgehen beim Anlegen eines SiSy-Projektes***

Schauen wir uns den geladenen Quellcode etwas genauer an. Dieser lässt sich in mehrere Bereiche unterteilen. Zum einen ist da der Programmkopf mit Dokumentationen und Deklarationen. Hier werden unter anderem die Deklarationen, zum Beispiel die Registernamen und die Funktionsdeklarationen des CMSIS und der Peripherietreiber für den STM32F4, aus externen Dateien in den Programmcode eingefügt *(#include*). Die *stddef* und *stdlib* sind exemplarisch eingefügte C-Standardbibliotheken.

```
//------------------------------------------------------------------
// Titel     : Grundgerüst einer einfachen ARM C Anwendung in SiSy
//------------------------------------------------------------------
// Funktion  : ...
// Schaltung : ...
//------------------------------------------------------------------
// Hardware  : STM32F4 Discovery
// Takt      : 168 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 Unterprogramme, 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. Es gibt durchaus die Möglichkeit, dies auch anders zu tun, aber das ist dann doch eher Bestandteil eines reinen C/C++ Lehrbuchs. Besonders der C-Neuling beachte den Funktionskopf, in dem Fall ohne Typ und Parameter sowie den Funktionskörper, begrenzt durch die geschweiften Klammern.

```
void initApplication()
{
    // config auf 10ms SystemTimer
    SysTick_Config(SystemCoreClock/100);
    // weitere Initialisierungen durchführen
}
```

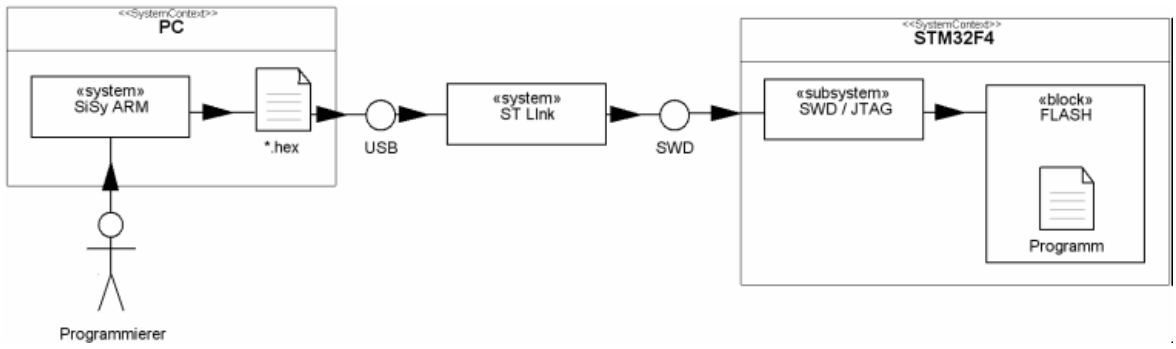
Als vorgegebenen Funktionsaufruf finden wir dort die Initialisierung des *SysTick-Timers*. Dieser liefert uns schon mal ein regelmäßiges Timerereignis. In den Übungen werden wir dies recht schnell benötigen. An dieser Stelle können noch weitere Funktionen eingefügt werden.
Es folgt jetzt das Hauptprogramm. Dies ist durch das Schlüsselwort *main* 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 von ST enthalten und übernimmt die Grundinitialisierungen des ARM Kerns. Die Funktion ist quellcodeoffen 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 Hardwareereignisse gebunden sind und automatisch auslösen können, stehen sie hinter dem Hauptprogramm als Letztes.

```
extern "C" void SysTick_Handler(void)
{
    // Application SysTick default 10ms
}
```

Rekapitulieren wir kurz was bisher getan wurde. Wir haben ein neues Projekt, ohne Vorlagen zu importieren, angelegt; ein kleines Programm in das Diagrammfenster gezogen und für die Zielsprache ARM C++ ein Grundgerüst geladen. Diesen Quellcode können wir jetzt übersetzen (kompilieren, linken) und in den Programmspeicher des Controllers (FLASH) übertragen (brennen).




Während der Übertragung sieht man die Status-LED des integrierten ST-LINK flackern. Nach der Übertragung leuchtet diese in der Regel grün für den Status RUN. Der ARM „arbeitet“ jetzt. Da unser Programm selbst aber nur ein leeres Grundgerüst ist, läuft der Controller faktisch im Leerlauf. Es sind keine Bausteine bzw. keine Peripherie aktiv.

### 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 Entwicklungboard ist möglich, wenn das USB-Kabel an Rechner und Entwicklungboard 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.

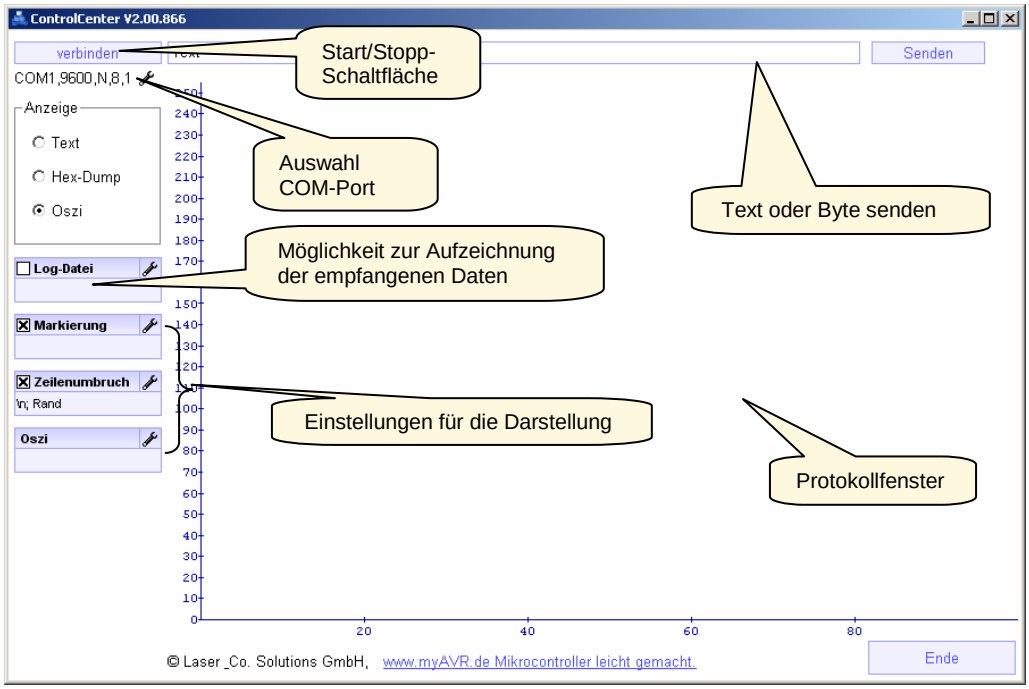



***Abbildung: SiSy ControlCenter***

### 1.3.4 Hilfen in SiSy

Nutzen Sie die zahlreichen Hilfen und Vorlagen, die SiSy bietet! Diese sind im Benutzerhandbuch von SiSy ausführlich beschrieben. Hier folgt ein kurzer Überblick.

#### *SiSy LibStore*

Der SiSy LibStore ist eine online-Sammlung von Vorlagen, Beispielprogammen und Bibliotheken. Diese speziellen Hilfen werden bei der Arbeit mit SiSy angeboten, sobald bei der Modellierung im jeweiligen Diagramm LibStore verfügbar ist und Sie online sind.

#### *Online-Hilfe*

Bei der Eingabe von Quellcode im Editorfenster werden reservierte Worte der gewählten Programmiersprache durch Syntaxfarben hervorgehoben. In der Regel existiert zu den hervorgehobenen Bezeichnern eine kurze online-Hilfe, welche in einen Pop-Up Fenster automatisch eingeblendet wird.

#### *SiSy Code-Vervollständigung*

Der Codegenerator ist eine integrierte Hilfe in SiSy. Er fungiert als Assistent zum Erstellen von Assembler- und C-Codes für die Programmierung von Mikrocontrollern. Bei der Eingabe von drei zusammenhängenden Buchstaben bei der Quellcodeerfassung springt die Codevervollständigung an und der gewünschte Befehl kann selektiert werden.

# 2 Programmierung in C mit dem STM32

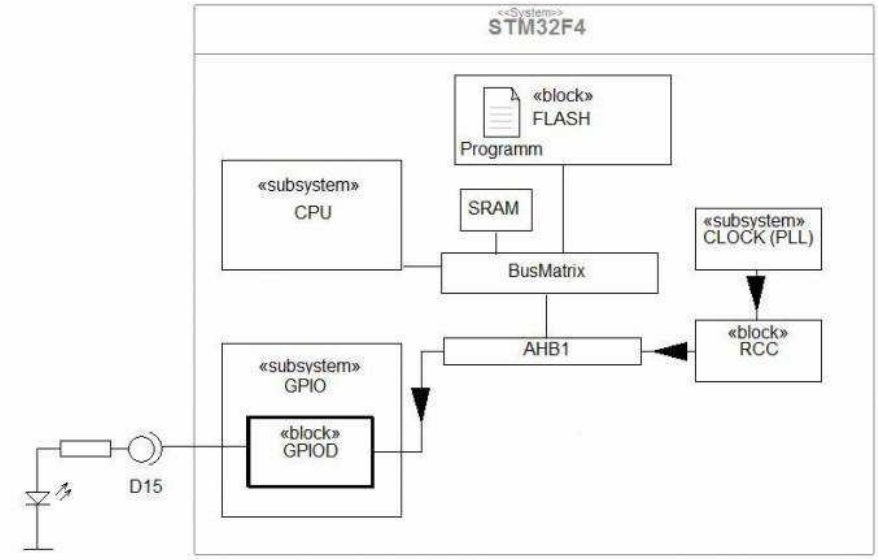
Die Programmierung im klassischen C kann man sich ruhig einmal antun. Umso mehr wird man die Klassen aus dem mySTM32 Framework schätzen lernen. Des Weiteren finden sich im Netz auch jede Menge Beispiele in klassischem C. Die folgenden Abschnitte befähigen Sie, sich diese zugänglich zu machen. Falls Sie lieber gleich objektorientiert in C++ und UML anfangen möchten, dann überspringen Sie diesen Abschnitt einfach.

## 2.1 „Hallo ARM“ in 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.

***Aufgabe***

Die erste Übung soll das typische LED einschalten sein. Dazu nutzen wir die blaue LED auf dem STM32F4-Discovery Board. Die blaue LED ist bereits fest mit dem Pin PD15 verbunden.




Aus dem Datenblatt des STM32F4xx (schauen Sie auf Seite 18) können wir entnehmen, dass dieser über drei AHB verfügt. Für uns ist erst einmal nur der AHB1 interessant. An AHB2 und AHB3 hängen spezielle Geräte, wie die Kameraschnittstelle oder ein externer Speicher. Über AHB1 erreichen wir die GPIO Ports und die zwei Peripheriebusse APB1 und APB2. Die digitalen Eingabe- und Ausgabeleitungen hängen direkt an AHB1 und sind zu 9 Ports (A bis I) mit jeweils 16 Leitungen (Bit 0 bis 15) zusammengefasst. Digitale Ein- und Ausgabe sind die primären Funktionen der Pins und im Pin-Out entsprechend als Pin-Namen gekennzeichnet.

Die Aufgabe besteht also darin:

- über den AHB1 Bus den GPIO Port D zu aktivieren, indem dieser mit einem Taktsignal versorgt wird
- das Bit 15 des GPIOD als Ausgang zu konfigurieren
- und das Pin auf High zu schalten

### *Vorbereitung*

Falls das SiSy-Projekt nicht mehr offen ist, öffnen Sie dies. Legen Sie ein neues kleines Programm mit dem Namen „HalloARM“ an und laden das Grundgerüst ARM C++ Anwendung.
Beachten Sie die Einstellungen für die Zielplattform STM32F4-Discovery.

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

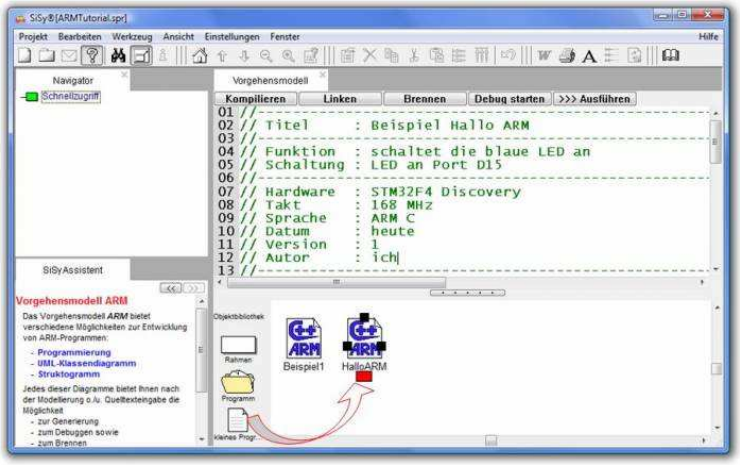
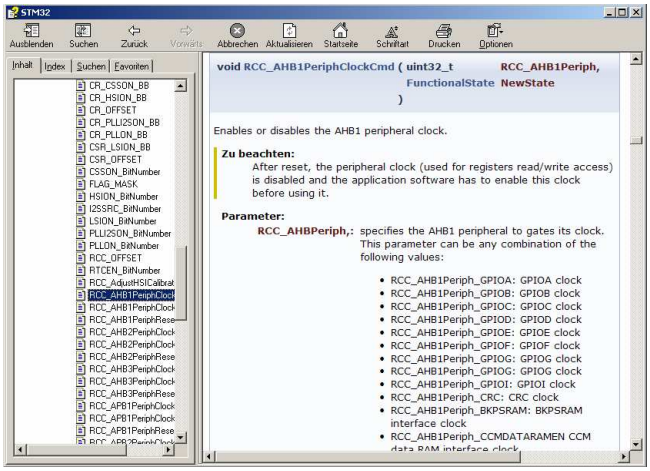



***Abbildung: Objekt „kleines Programm“ aus der Objektbibliothek in das Diagrammfenster ziehen***

```
//------------------------------------------------------------------
// Titel     : Beispiel Hallo Welt mit SiSy STM32
//------------------------------------------------------------------
// Funktion  : schaltet die blaue LED an
// Schaltung : blaue LED an GPIO Port D15
//------------------------------------------------------------------
// Hardware  : STM32F4 Discovery
// Takt      : 168 MHz
// Sprache   : ARM C++
// Datum     : heute
// Version   : 1
// Autor     : ich
//------------------------
```

### *Lösungsansatz*

Als Erstes diskutieren wir kurz die nötigen Lösungsschritte. Wie bereits ausgeführt, sind nach dem RESET alle Peripheriegeräte ausgeschaltet. Demzufolge ist der I/O-Port, an dem die LED angeschlossen ist, erst einmal einzuschalten.
Jetzt schauen wir uns das Blockbild zum STM32F4, zum Beispiel im Datenblatt Seite 18, oder das vereinfachte Blockbild an. Man erkennt, dass der RCC-Unit (Reset & Clock Control) mitgeteilt werden muss, dass GPIOD über den AHB1 mit Takt zu versorgen ist. Dazu nutzen wir die Funktion *RCC_AHB1PeriphClockCmd* aus den STM32F4-Peripherie-Treibern. Die Hilfe zu der Funktion können wir uns über den Editor, rechte Maustaste, Hilfe STM32 ansehen.




Die Funktion *RCC_AHB1PeriphClockCmd* benötigt zwei Parameter. Parameter eins bestimmt das Gerät und Parameter zwei den neuen Status des Gerätetaktes. Daraus ergibt sich folgende Befehlszeile:

```
 /* GPIOD Takt einschalten  */
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
```

Jetzt ist der GPIO Port D angeschaltet. Das ist eine wichtige Voraussetzung damit wir diesem Initialisierungskommandos senden können. Die Initialisierung eines Gerätes, selbst eines einfachen I/O-Ports des 32-Bit ARM Cortex, ist um einiges aufwendiger als beim kleinen 8-Bit AVR. Zur Vereinfachung gibt es für den Programmierer die Möglichkeit, Geräte über Strukturen und Treiberfunktionen zu initialisieren. Diese abstrahieren die Hardware und fassen alle nötigen Einstellungen kompakt zusammen. Die STM32 Peripherie-Treiber und auch CMSIS haben in der Regel für jedes Gerät mindestens eine Initialisierungsstruktur und zwei korrespondierende Funktionen. Die erste Funktion initialisiert eine angelegte leere Initialisierungsstruktur mit den Grundeinstellungen für das Gerät und die zweite führt die Initialisierung nach den Vorgaben des Entwicklers aus. Damit ergibt sich folgendes Strickmuster zur Initialisierung von Geräten:

- **Takt einschalten**, *RCC_xxxClockCmd*
- **Initialisierungsstruktur anlegen**, *xxx_InitTypDef initStruct*
- **Struktur mit Standardwerten füllen**, *xxx_StructInit (&initStruct)*
- **Spezifische Anpassungen vornehmen**, *initStruc.xxx_Mode = wert*
- **Gerät initialisieren**, *xxx_Init(xxx, &initStructure)*

Der Takt für Port D ist bereits aktiviert. Demzufolge ist als nächstes die Initialisierungsstruktur anzulegen und mit Standardwerten zu füllen. Strukturen und Funktionen finden sich in der Hilfe im Abschnitt des jeweiligen Gerätes. Der entsprechende Quellcode für den GPIO Port D sieht wie folgt aus:

```
GPIO_InitTypeDef  GPIO_InitStructure;
GPIO_StructInit (&GPIO_InitStructure);
```

Als Nächstes müssen die anwendungsspezifischen Einstellungen angegeben werden. Das erfolgt durch Zuweisung der entsprechenden Werte zu den einzelnen Elementen der Initialisierungsstruktur. Die möglichen Strukturelemente und Werte sind wiederum der Hilfe entnehmbar.

Bei den Werten für die Strukturelemente handelt es sich um Aufzählungen bzw. Bitdefinitionen, welche als selbsterklärende Bezeichner deklariert wurden.

- Strukturelement *GPIO_Pin:*
  Die Werte können angegeben werden mit *GPIO_Pin_0* bis *GPIO_Pin_15*. Hier handelt es sich um Bitdefinitionen. Diese können ODER-verknüft werden, um zum Beispiel mehrere Pins gleichzeitig anzusprechen.

- Strukturelement *GPIO_Mode*:
  Dabei handelt es sich um eine Aufzählung. Diese Werte können nicht kombiniert werden, sondern schließen sich gegenseitig aus.
  - *GPIO_Mode_IN*: GPIO Input Mode,
    der/die Pins werden als Eingang betrieben
  - *GPIO_Mode_OUT*: GPIO Output Mode,
    der/die Pins werden als Ausgang betrieben
  - *GPIO_Mode_AF*: GPIO Alternate function Mode,
    der/die Pins werden nicht als GPIO betrieben, sondern bekommen eine alternative Peripheriefunktion zugewiesen
  - *GPIO_Mode_AN*: GPIO Analog Mode,
    der/die Pins werden als Analogeingang betrieben

- Strukturelement *GPIO_Otype*:
  Dieser Wert bezieht sich auf den Output Mode. Die Einstellungen schließen einander aus.
  - *GPIO_OType_PP*: Push Pull,
    der Ausgangstreiber arbeitet als Gegentaktstufe, gibt definiert entweder High oder Low aus
  - *GPIO_OType_OD*: Open Drain,
    der Ausgangstreiber schaltet nur gegen Masse und ist ansonsten hochohmig, vgl. Open Collector, ist mit PullUp Widerstand kombinierbar

- Strukturelement *GPIO_Speed*:
  Gibt an, mit welcher Zykluszeit die Register des Ports aktualisiert werden, also wie schnell zum Beispiel Änderungen zwischen Register und Treiberstufen durchgeschaltet werden.
  - *GPIO_Speed_2MHz*
  - *GPIO_Speed_25MHz*
  - *GPIO_Speed_50MHz*
  - *GPIO_Speed_100MHz*

- Strukturelement *GPIO_PuPd*:
  Der STM32 verfügt über interne PullUp und PullDown Widerstände. Diese können wahlweise aktiviert werden. Die Werte schließen sich gegenseitig aus.
  - *GPIO_PuPd_NOPULL*: kein PullUp oder PullDown aktiviert
  - *GPIO_PuPd_UP*: PullUp Widerstand aktivieren
  - *GPIO_PuPd_DOWN*: PullDown Widerstand aktivieren

Für unsere LED ergibt sich, dass diese an Pin15 angeschlossen ist, dieser als Ausgang betrieben werden soll und keine PullUp oder PullDown benötigt werden. Der mögliche Quellcode sieht wie folgt aus:

```
GPIO_InitStructure.GPIO_Pin =  GPIO_Pin_15;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
```

Damit sind alle relevanten Einstellungen in der Initialisierungsstruktur vorbereitet. Jetzt kann die eigentliche Initialisierung erfolgen. Das geschieht mit der Funktion *GPIO_Init*. Diese erfordert die Angabe des GPIO-Port und der vorbereiteten Initialisierungsstruktur.

```
GPIO_Init(GPIOD, &GPIO_InitStructure);
```

Für das An- oder Ausschalten von GPIO-Pins stehen die Funktionen *GPIO_SetBit* und *GPIO_ResetBit* zur Verfügung. Diese Funktionen erwarten den Port und die Pins, welche zu schalten sind.

```
GPIO_SetBits(GPIOD,GPIO_Pin_15);
```

Es wird wohl deutlich, dass selbst die Initialisierung eines einfachen Ausgangs, um eine LED einzuschalten, recht aufwendig ist. Dabei war dies nur das Minimum im Umgang mit einem GPIO-Port. Der ARM gibt dem Anwendungsentwickler noch viel umfangreichere Möglichkeiten.

***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     : Beispiel Hallo Welt mit SiSy STM32
//------------------------------------------------------------------
// Funktion  : schaltet die blaue LED an
// Schaltung : blaue LED an GPIO Port D15
//------------------------------------------------------------------
// Hardware  : STM32F4 Discovery
// Takt      : 168 MHz
// Sprache   : ARM C++
// Datum     : heute
// Version   : 1
// Autor     : ich
//------------------------------------------------------------------
#include <stddef.h>
#include <stdlib.h>
#include “hardware.h“

void initApplication()
{
    SysTick_Config(SystemCoreClock/100);
     // GPIOD Takt einschalten
    // Konfiguriere GPIO Port D15 für die LED
 }

int main(void)
{
    SystemInit();
    initApplication();
    do{
         // LED anschalten,
     } while (true);
    return 0;
}

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

Jetzt nehmen wir die Finger von der Tastatur, atmen tief durch und schauen noch mal in Ruhe über unseren Entwurf. Dann kann es los gehen.

***Realisierung***

Ergänzen Sie den Quellcode des Beispiels „HalloARM“ wie folgt: Nutzen Sie die Codevervollständigung des Editors. Die in den Treibern systematisch festgelegten Bezeichner folgen einem einfach einzuprägenden Muster:

```
Gerät_TeilKomponente_Was ( Parameter ) ;
```

Der Bezeichner beschreibt einen Pfad vom Allgemeinen (dem Gerät) zum Speziellen (z.B. einer konkreten Funktion oder einem Bit). Zum Beispiel finden Sie alle Funktionen zur *Reset and Clock Control Unit* unter *RCC_* **.**

Nach drei zusammenhängenden Buchstaben springt die Codevervollständigung an und listet alle Bezeichner fortlaufend gefiltert nach dem Stand der Eingabe.

Wählen Sie jetzt die Taste CUD (Cursor/Pfeil nach unten), können Sie in der Liste rollen und per Enter einen Eintrag auswählen.

Also dann, viel Erfolg bei den ersten richtigen Programmierschritten auf dem ARM.

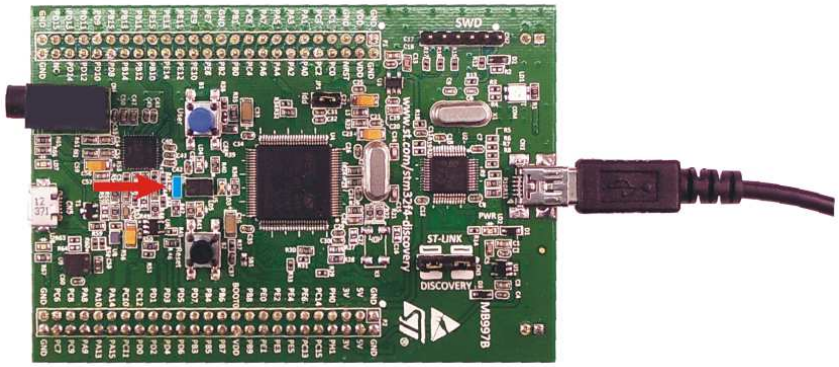
```
//-------------------------------------------------------------
// Titel     : Beispiel Hallo Welt mit SiSy STM32
//-------------------------------------------------------------
// Funktion  : schaltet die blaue LED an
// Schaltung : blaue LED an GPIO Port D15
//-------------------------------------------------------------
// Hardware  : STM32F4 Discovery
// Takt      : 168 MHz
// Sprache   : ARM C++
// Datum     : heute
// Version   : 1
//-------------------------------------------------------------
#include <stddef.h>
#include <stdlib.h>
#include “hardware.h“

void initApplication()
{
    SysTick_Config(SystemCoreClock/100);
    // weitere Initialisierungen durchführen
     /* GPIOD Takt einschalten  */
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);

    /* Konfiguriere GPIO Port D15 */
    GPIO_InitTypeDef  GPIO_InitStructure;
    GPIO_StructInit (&GPIO_InitStructure);
    GPIO_InitStructure.GPIO_Pin =  GPIO_Pin_15;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOD, &GPIO_InitStructure);
 }
int main(void)
{
    SystemInit();
    initApplication();
    do{
        GPIO_SetBits(GPIOD,GPIO_Pin_15);
    } while (true);
    return 0;
}
 extern "C" void SysTick_Handler(void)
{
    // Application SysTick
}
```

***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

Gratulation! Sie haben Ihre erste Ausgabe realisiert.
Die blaue LED auf dem STM32F4-Discovery leuchtet jetzt.




***Zusammenfassung***

Fassen wir noch mal kurz zusammen, was es sich einzuprägen gilt:

**Initialisierungssequenz für Geräte:**

- Takt einschalten, *RCC_xxxClockCmd*
- Initialisierungsstruktur anlegen, *xxx_InitTypDef initStruct*
- Struktur mit Standardwerten füllen, *xxx_StructInit (&initStruct)*
- spezifische Anpassungen vornehmen, *initStruc.xxx_Mode = wert*
- Gerät initialisieren, *xxx_Init(xxx, &initStructure)*

**Initialisierungsstruktur für Digitalports:** *GPIO_InitTypeDef initStruct;*

- `initStruct.GPIO_Pin = GPIO_Pin_0..15;`
- `initStruct.GPIO_Mode = GPIO_Mode_OUT|IN|AF;`
- `initStruct.GPIO_OType = GPIO_OType_PP|OD;`
- `initStruct.GPIO_Speed = GPIO_Speed_2..100MHz;`
- `initStruct.GPIO_PuPd = GPIO_PuPd_NOPULL|UP|DOWN;`

**wichtige Funktionen für Digitalports:**

- `RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOx, ENABLE|DISABLE);`
- `GPIO_StructInit (&initStruct);`
- `GPIO_Init(GPIOD, &initStruct);`
- `GPIO_SetBits(GPIOD,GPIO_Pin_x);`
- `GPIO_ResetBits(GPIOD,GPIO_Pin_x);`

. . . . .

# 3 Ausgewählte Paradigmen der Softwareentwicklung

## 3.1 Basiskonzepte der Objektorientierung

Ausgewählte Basiskonzepte der objektorientierten 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.


*Wir geben den Dingen Namen.*

***Objekt***
Womit wir beim nächsten Punkt sind. Dinge bezeichnet der Fachmann als Objekte und mal ehrlich, Dingorientierung klingt auch nicht wirklich sexy. Also dann doch lieber Objektorientierung. 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. Programmieren wir objektorientiert, müssen wir dafür sorgen, dass auch das Programm aus genau diesen Objekten besteht und die Attribute und Operationen diesen Objekten zugeordnet sind.


*Die Bausteine, aus denen das System besteht, sind der Ausgangspunkt der Softwareentwicklung. Diese Bausteine bezeichnen wir als Objekte.*

***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 zwei Stück. Um diese zu unterscheiden, geben wir jedem noch einen individuellen Namen nämlich „Taster-1“ und „Taster-2“. Taster steht also als Begriff für alle Schalter mit den entsprechenden gleichen Eigenschaften. Der Fachmann bezeichnet so etwas als Kategorie oder auch Klasse. Die beiden Objekte „Taster-1“ und „Taster-2“ sind Bausteine unseres Systems und gehören zur Klasse (Gruppe) der Taster. Unser oberschlauer Fachmann bezeichnet diese beiden konkreten Objekte auch gern als Instanzen der Klasse „Taster“. Übrigens kennen wir diese Problematik schon aus der klassischen Programmierung in Form von Typen und Variablen. Klassen sind die Typen, und die Objekte so etwas wie die Variablen.

*Wir geben einer Menge gleichartiger Bausteine einen Gruppennamen (Klassennamen) und beschreiben die gemeinsamen Merkmale (Attribute und Operationen). Objekte sind Instanzen einer Klasse.*


.....

***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.
Ausgangspunkt für das Verstehen einer objektorientierten Programmiersprache ist immer das objektorientierte Paradigma. Das Basiskonzept stellt sozusagen das SOLL und die konkrete Sprache das IST dar. Gehen Sie davon aus, dass eigentlich in keiner derzeit verfügbaren objektorientierten Sprache auch alle in der Theorie formulierten Konzepte bereits komplett umgesetzt sind. Man sollte sich auf jeden Fall davor hüten, von den Möglichkeiten und den Einschränkungen einer konkreten Sprache auf das Konzept zu schließen. Wesentliche Aspekte objektorientierter Sprachen sollen im Folgenden anhand der Sprache C++ aufgezeigt werden. Für das Verständnis von C++ ist weiterhin wichtig zu wissen, dass C++ die Sprache C beinhaltet. C++ ist die objektorientierte Erweiterung der Sprache C.

### 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 ............................................
+               -               *               /
=               ++              --
<<              >>              !               &
|               ^               ~               %
==              >               <               <=
>=              &&              ||              !=
```

**.....**

### 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.

***Ein einfaches C++ Programm für ARM Mikrocontroller:***

```
//-------------------------------------------------------
#include <stddef.h>
#include <stdlib.h>
#include “hardware.h“

class Controller
{
  public: void onStart()
  {
    SysTick_Config(SystemCoreClock/100);
    // weitere Initialisierungen durchführen

    this->run();
  }

  protected: void run()
  {
    do {
       // Eingabe
       // Verarbeitung
       // Ausgabe
       } while (true);

  }
 public: void onSysTick()
  {
      // Application SysTick
  }

} app;

//-------------------------------------------------------
// StartUp in old C-Style
int main(void)
{
   SystemInit();
   app.onStart();
   return 0;
}

extern "C" void SysTick_Handler(void)
{
    app.onSysTick();
}
//-------------------------------------------------------
```

## 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, 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.

. . . . .

## 3.4 Grafische Programmierung mit UML

Mit objektorientierten Programmiersprachen hat der Entwickler mächtige Sprachmittel, um komplexe Systeme realisieren zu können. C++ ist eine weit verbreitete objektorientierte Programmiersprache. Als Visualisierungsmittel objektorientierter Programme gilt die international standardisierte Beschreibungssprache UML (Unified Modeling Language).

SiSy bietet dem Entwickler das UML-Klassendiagramm mit Codegenerierung für unterschiedliche Plattformen, unter anderem auch für AVR- und ARM-Mikrocontroller.

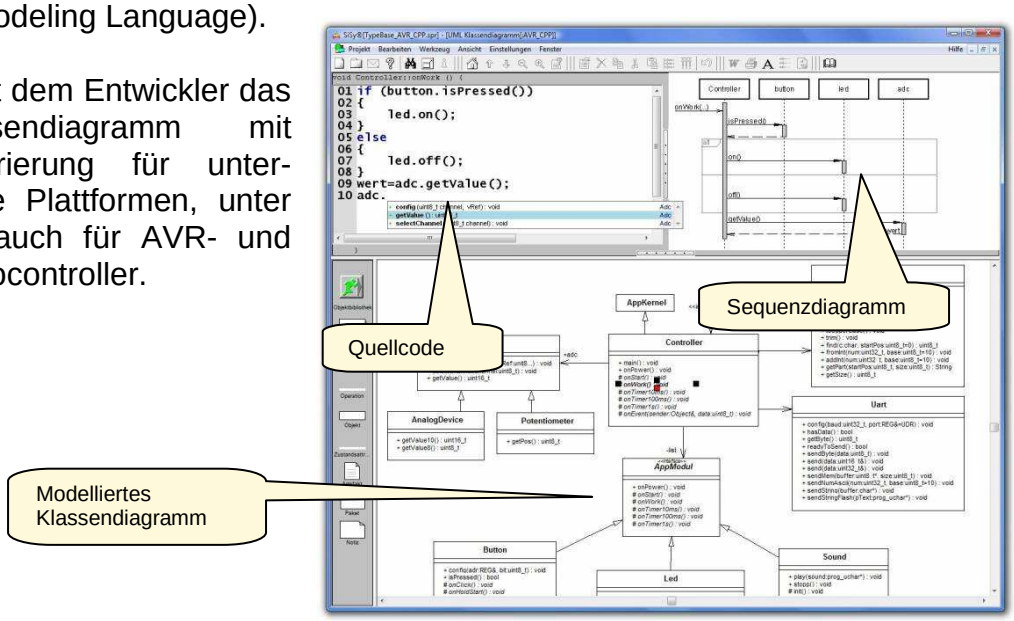
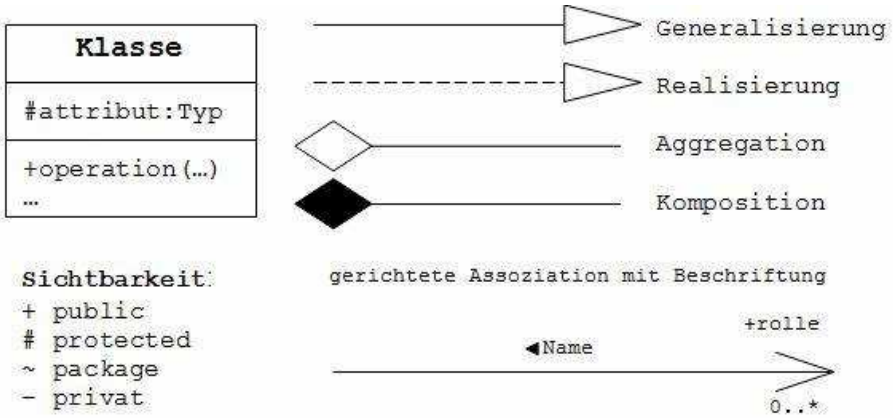



***Abbildung: Anordnung verschiedener Fenster bei der Arbeit mit Klassendiagrammen in SiSy***

Die folgende Abbildung zeigt Ihnen eine Kurzübersicht der Modellierungselemente des UML-Klassendiagramms.



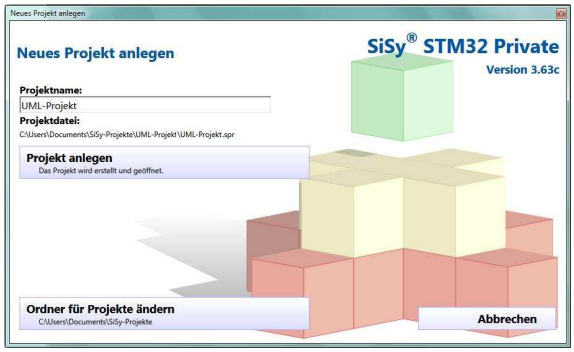

. . . . .

# 4 STM32 Programmierung in C++ mit der UML

## 4.1 Grundstruktur

### *Ein UML Projekt anlegen*

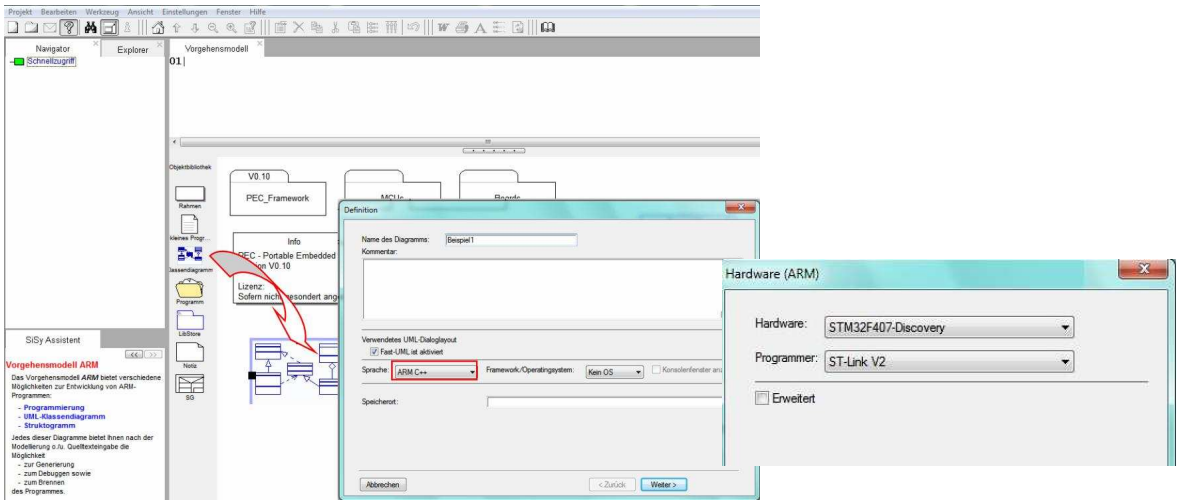
Für die weitere Arbeit in diesem Lehrbuch verwenden wir als Entwicklungsumgebung das UML-Klassendiagramm und Klassenbibliotheken für den STM32F4. Es ist nötig, dafür ein neues Projekt anzulegen und eine Projektvorlage mit den gewünschten Bibliotheken auszuwählen. Legen Sie ein neues SiSy-Projekt mit dem Namen „UML-Projekt“ an und wählen Sie das ARM-Vorgehensmodell.




Es öffnet SiSy LibStore und Sie erhalten verschiedene Vorlagen zur Auswahl. Laden Sie die Vorlage für das „*PEC Framework - Portable Embedded Classes*“.

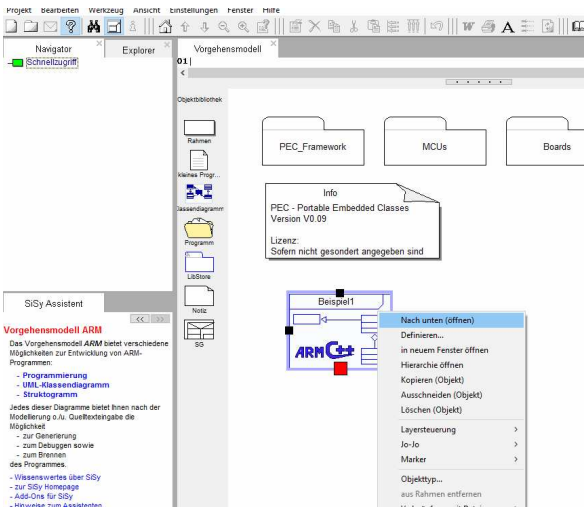

***Abbildungen: SiSy Projekt anlegen und Vorlage aus SiSy LibStore auswählen***

Legen Sie ein neues Klassendiagramm an, indem Sie das entsprechende Element per Drag&Drop aus der Objektbibliothek in das Diagrammfenster ziehen. Geben Sie dem Diagramm den Namen „Beispiel1“, achten Sie auf die Einstellung der Zielsprache ARM C++. Wählen Sie im nächsten Fenster die Hardware *STM32F407 Discovery* und den Programmer *ST-Link V2* aus.




***Abbildungen: Klassendiagramm anlegen und Einstellungen vornehmen***

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)“*. Schränken Sie die Vorlagensuche ggf. mit dem Suchbegriff *PEC* ein.
Weisen Sie dem Diagramm das Treiberpaket für den konkreten Controller *MCU_STM32F4* 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_STM32F4“ in das Diagrammfenster.

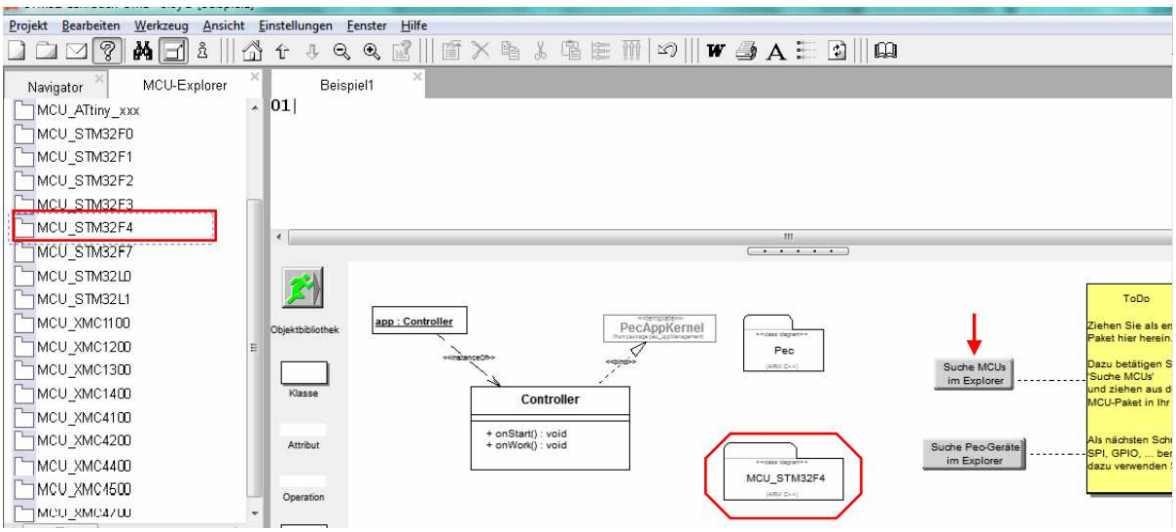
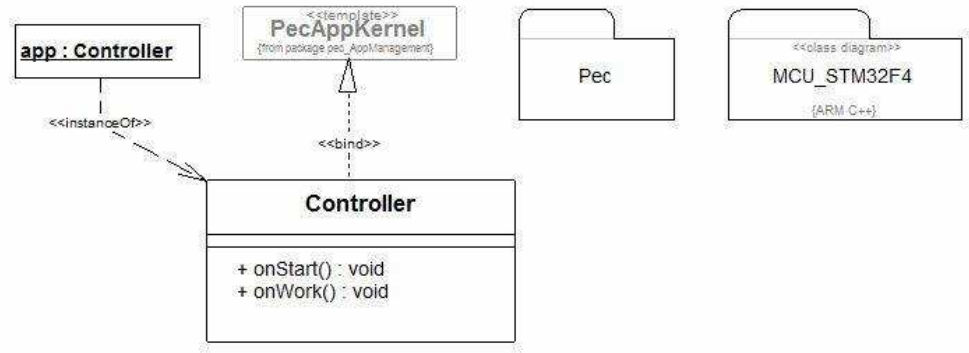



***Abbildung: Treiberpaket im MCU-Explorer auswählen und in das Diagramm ziehen***

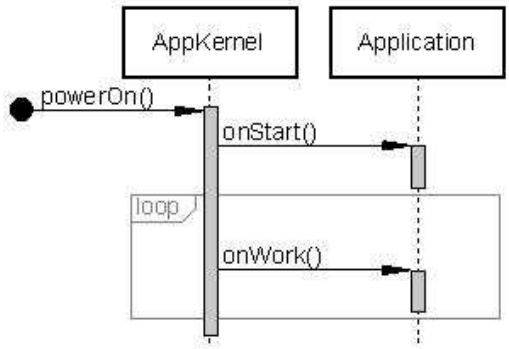
### *Grundstruktur einer objektorientierten Anwendung*

Sie erhalten das nachfolgende Diagramm. Dabei handelt es sich um die typische Grundstruktur einer objektorientierten Anwendung auf der Basis von SiSy ARM C++ Framework.



Die Klasse *Controller* 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 Referenz des Paketes *PecFramework* werden alle benötigten Klassen aus der Bibliothek importiert.

Das Template *PecAppKernel* stellt bereits eine Reihe von nützlichen Struktur- und Verhaltensmerkmalen einer ARM-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 ARM“ in C++

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 allem auch der, dass Dinge die öfter gebraucht werden oder typische Problemstellungen, die einfach schon mal gelöst wurden, dem Anwender komfortabel zur Wiederverwendung zur Verfügung stehen.

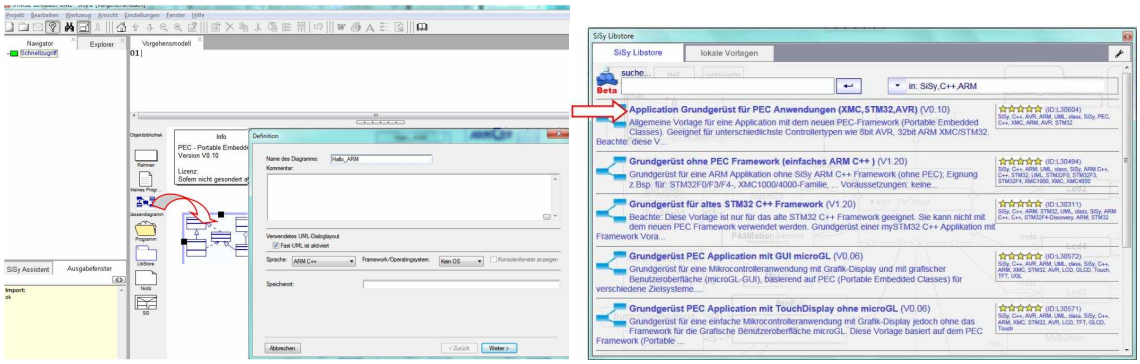
Die Objektorientierung zeichnet sich durch zunehmende Abstraktion von der tatsächlichen inneren Struktur und dem internen Verhalten der Maschine, hin zu einer anwenderbezogenen Sichtweise aus.

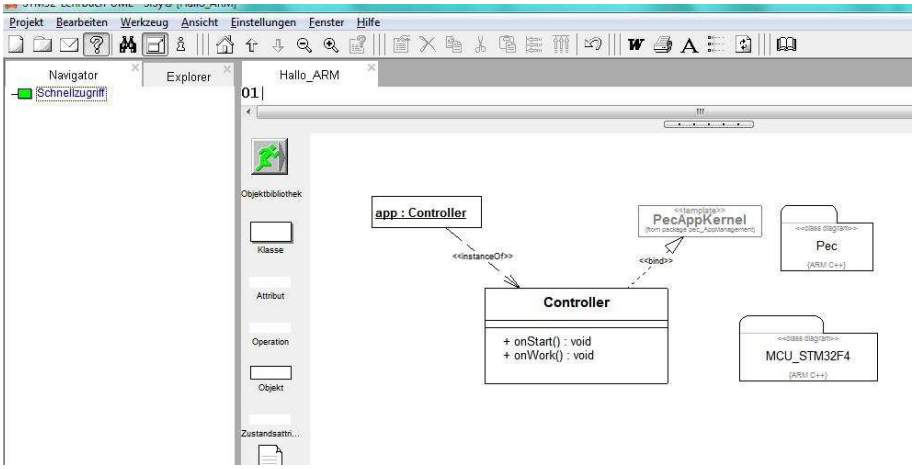
### *Aufgabe*

Entwickeln Sie eine Mikrocontrolleranwendung, die eine LED anschaltet.

### *Vorbereitung*

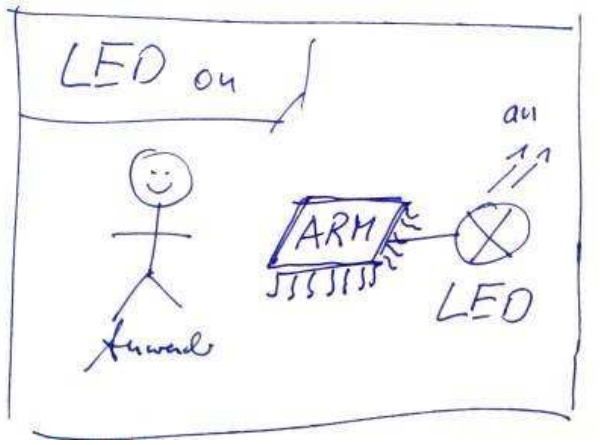
Falls Sie jetzt noch das Klassendiagramm geöffnet haben, wählen Sie im Kontextmenü (rechte Maustaste) des Diagramms den Menüpunkt „nach oben“. Falls das Projekt nicht mehr geöffnet ist, öffnen Sie das SiSy UML-Projekt wieder.
Legen Sie ein neues Klassendiagramm an, wählen Sie die Sprache ARM C++ und stellen Sie die Hardware (STM32F407) ein. Beim Öffnen des Diagramms (rechte Maustaste, nach unten) laden Sie aus dem SiSy LibStore die Diagrammvorlage *Application Grundgerüst für PEC Anwendungen (XMC, STM32, AVR).* Weisen Sie das Treiberpaket für *STM32F4* zu.





***Lösungsansatz***

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.

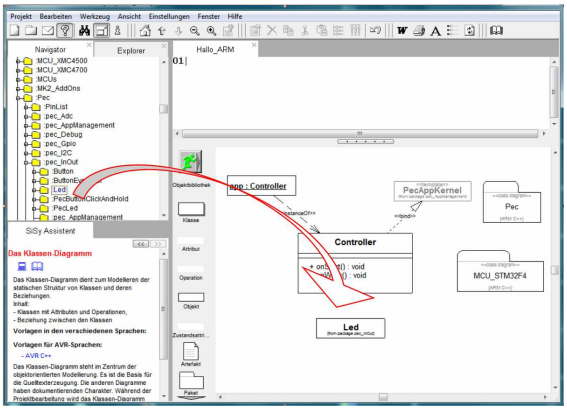


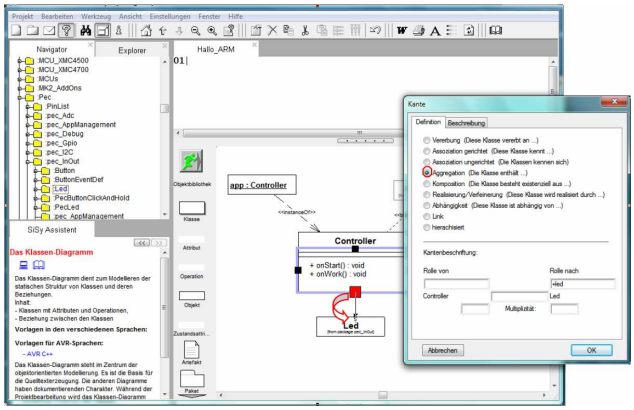

Die Abbildung genau dieser Verhaltensmerkmale fordern wir von der Klasse *Led*. Sich mit *GPIO_OTYP_PP* und *AHB1PeriphCmd* herumzuschlagen ist mit Sicherheit wichtig, um zu erlernen, wie ARM Controller intern funktionieren und um ggf. eigene Klassen für die Erweiterung der Bibliothek zu realisieren. Aber es ist für die Lösung eines konkreten Anwendungsproblems eher zeitfressend und vielleicht sogar kontraproduktiv. Welcher GUI-Entwickler kümmert sich noch wie vor 25 Jahren von Hand um Peek-, Get-, Translate- und DispatcheMessage, nur weil es grundlegende und extrem wichtige Funktionen in einer grafischen Benutzeroberfläche sind. Er möchte eine konkrete Anwendung schreiben. Dazu reichen das Grundverständnis der Funktionsweise einer GUI und eine leistungsfähige Klassenbibliothek.
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_Framework. Das Suchen geeigneter Klassen in den Bibliotheken erfolgt in SiSy am Besten über den Navigator oder den Explorer.
Im Navigator öffnen Sie das Kontextmenü mit der rechten Maustaste. Wählen Sie den Menüpunkt „UML-Pakete*“.* Wir werden recht schnell beim Durchstöbern der Pakete fündig. Im Paket *Pec/pec_InOut* findet sich neben anderen Klassen eine fertige Klasse *Led*.

Im Explorer geben Sie einfach den gesuchten Begriff als Suchtext ein.
In dieser fertigen Klasse *Led* sind all die mühseligen Kommandos zum Einschalten des Taktes und dem Füllen der Initialisierungsstrukturen bereits gelöst. Die Klasse kann auch wesentlich mehr als wir für die Aufgabe brauchen. Mit etwas Übung im SiSy-Handling kann man sich elegant zum Quelldiagramm der Basisklasse durchhangeln, um sich all das anzuschauen. Dort sind das Klassendiagramm und der Quellcode der Bibliotheksklasse einseh- und ggf. auch änderbar (Modell- und Quellcodeoffen).

Wir könnten 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 bzw. Explorer per Drag&Drop in das Klassendiagramm.





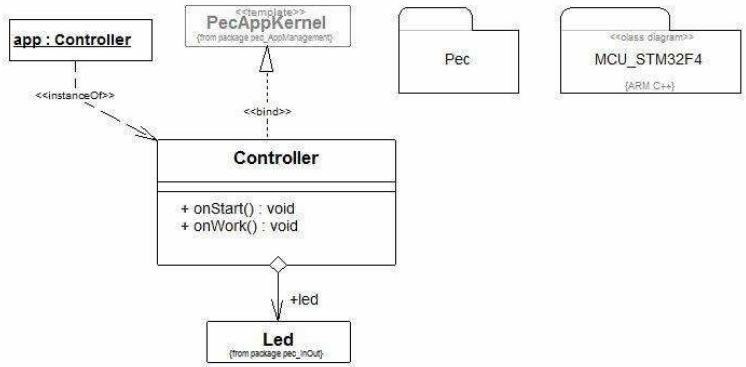
Wenn wir die *Led* hereingezogen haben müssen wir diese mit der Anwendung verbinden. Vergleichen Sie dazu die Ausführungen zur Aggregation im Grundlagenteil dieses Lehrbuches. Selektieren Sie dazu die Klasse *Controller*. Am unteren Rand erscheint eine große, rote Selektionsmarke. Das ist der Verteiler. 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*. Der Controller hat jetzt eine *Led*.

Das ist dann unser Entwurf der Struktur unserer Anwendung in Form eines UML-Klassendiagramms. Die meisten UML-Werkzeuge bieten an dieser Stelle das Generieren der Klassenrümpfe, um diese dann in eine IDE einzufügen. Dort werden dann die konkreten Verhaltensmerkmale der so entworfenen Klassen fertig programmiert und ggf. die Klassen angepasst oder erweitert. Damit stimmen Konstruktionszeichnung und Quellcode jedoch nicht mehr miteinander überein. Das ist wie der richtige Schaltplan zur falschen Schaltung oder umgekehrt. Für diesen Defekt gibt es manchmal die Möglichkeit, dass das UML-Tool sich wiederum mit den Quelltextänderungen synchronisiert. In SiSy bleiben wir im Klassendiagramm und realisieren hier auch gleich das System.

### *Realisierung*

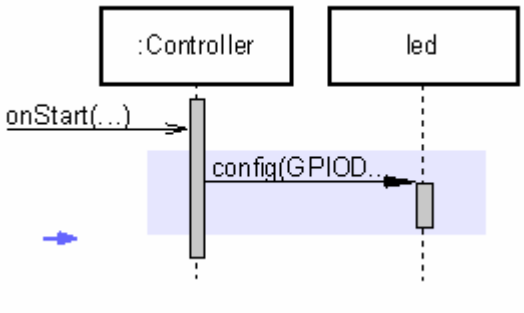
Der Entwurf sollte der folgenden Darstellung entsprechen:

- die zentrale Klasse ist *Controller*
- diese verfügt über die Operationen *onStart* und *onWork*
- Die Applikation 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 die blaue LED einzuschalten ist. Diese ist bekanntlich an Pin D15 angeschlossen. Für die Initialisierung von Geräten steht die Operation *onStart* zur Verfügung. Selektieren Sie diese Operation und geben im Quelltexteditor folgenden Code ein:

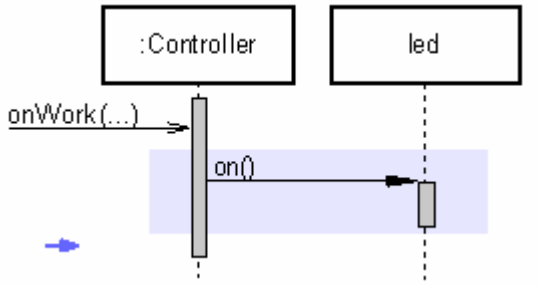
```
Controller::onStart:
      led.config(GPIOD,BIT15);
```




Beobachten Sie gleichzeitig das rechts neben dem Quelltexteditor befindliche Sequenzdiagramm. Dieses wird automatisch 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* und geben den folgenden Code ein:

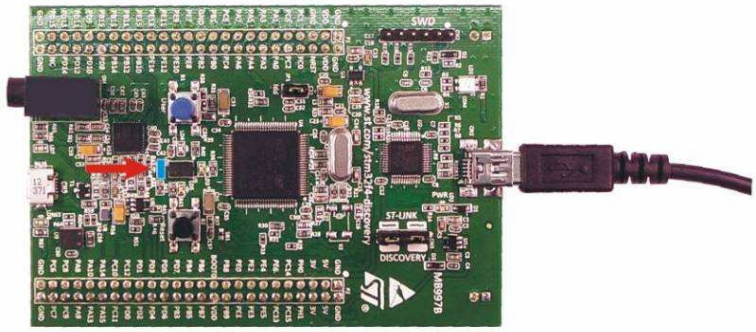
```
Controller::onWork:
      led.on();
```




***Test***

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

1. Erstellen (Kompilieren und Linken)
2. Brennen



***Zusammenfassung***

Fassen wir das Gelernte zusammen:

- Klassendiagramm anlegen und öffnen
- Diagrammvorlage für *PEC Applikation* auswählen, laden und Treiberpaket für STM32F4 einfügen
- ggf. Navigator auf UML Pakete umschalten
- gewünschte Klasse *Led* im Navigator oder Explorer suchen und in das Diagramm ziehen
- Klassen *aggregieren*
- den nötigen *Quellcode* in den Operationen erstellen
- Erstellen und Brennen einer ARM Applikation im Klassendiagramm

***Übung***

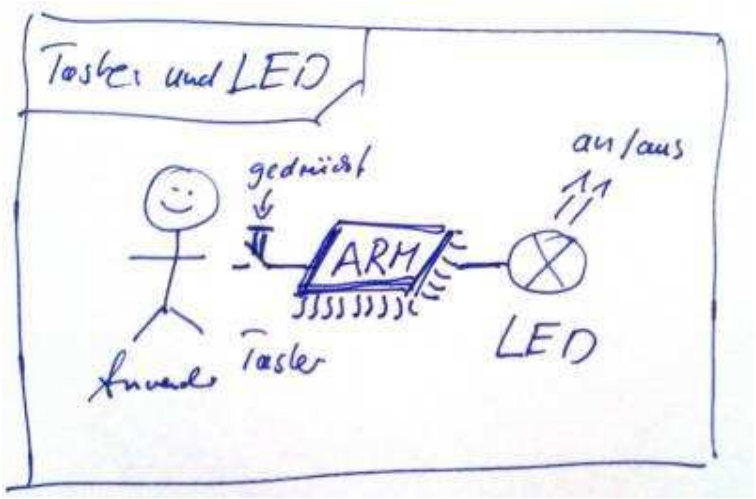
Erweitern Sie zur Übung die Anwendung um die grüne LED an D12. Orientieren Sie sich an den obigen Arbeitsschritten. Beachten Sie, dass die grüne LED nicht den gleichen Attributnamen (Rolle) wie die blaue LED besitzen darf.

## 4.3 Mit den Klassen *Button* und *Led* arbeiten

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.

***Aufgabe***

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




***Vorbereitung***

Wenn 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, wählen Sie die Sprache ARM C++ und stellen Sie die Hardware (STM32F407) ein. Beim Öffnen des Diagramms (rechte Maustaste, nach unten) laden Sie aus dem SiSy LibStore die Diagrammvorlage *Application Grundgerüst für PEC Anwendungen (XMC, STM32, AVR).* Weisen Sie das Treiberpaket für *STM32F4* zu.

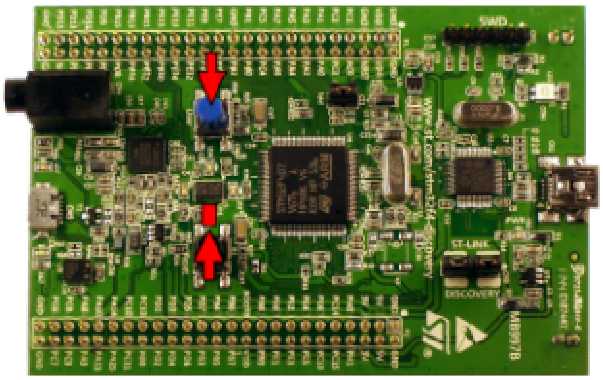
. . . . .

### *Test*

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

1. Erstellen (Kompilieren und Linken)
2. Brennen



Die rote LED auf dem STM32F4-Discovery leuchtet nun immer solange, wie der blaue Taster gedrückt ist.

### *Zusammenfassung*

- Klassendiagramm anlegen und öffnen
- Diagrammvorlage für PEC Applikation auswählen, laden und Treiberpaket für STM32F4 einfügen
- ggf. Navigator auf UML Pakete umschalten
- gewünschte Klassen Led und Button suchen und ins Diagramm ziehen
- Klassen aggregieren
- den nötigen Quellcode in den Operationen erstellen
- Erstellen und Brennen eine ARM Applikation im Klassendiagramm

### *Übung*

Erweitern Sie die Anwendung um die blaue LED. Diese soll leuchten, wenn der Taster nicht gedrückt ist und aus sein wenn er gedrückt wird.

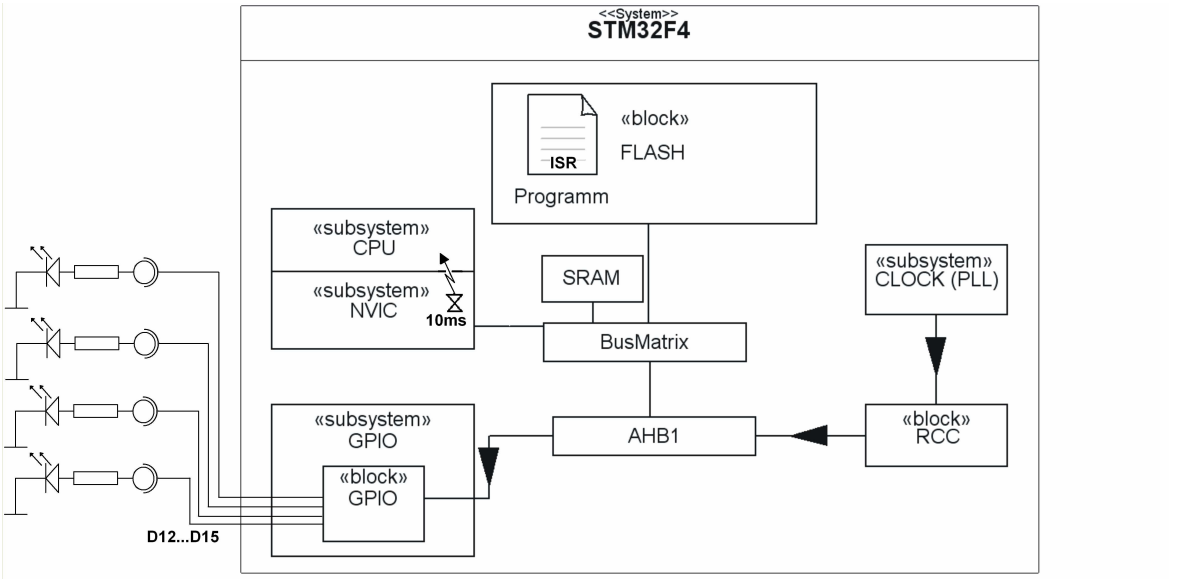
## 4.4 Der SystemTick in C++

Den *SysTick* haben wir beim einfachen Programm bereits kennengelernt. 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.

***Aufgabe***

Diese Übung wird wiederum eine einfache Verwendung von *SysTick_Handler* 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. Vergleichen Sie diese vereinfachte Darstellung mit dem Blockbild aus dem Datenblatt.




Die vier LEDs auf dem STM32F4-Discovery sind immer noch fest mit den Pins D12 bis D15 verbunden. Der SystemTick soll so konfiguriert werden, dass dieses Ereignis alle 10 Millisekunden eintritt. Die Takt-Versorgung des GPIO Ports D erfolgt, wie wir wissen, über AHB1. Fassen wir die Aufgaben zusammen:

- Das SysTick-Ereignis auf 10 ms konfigurieren
- über den AHB1 Bus den GPIO Port D mit Takt versorgen
- die Bits 12 bis 15 des GPIO Port D als Ausgang konfigurieren
- wenn das SysTick-Ereignis eintritt, die LEDs an GPIO Port D12 bis 15 unterschiedlich blinken lassen

. . . . .

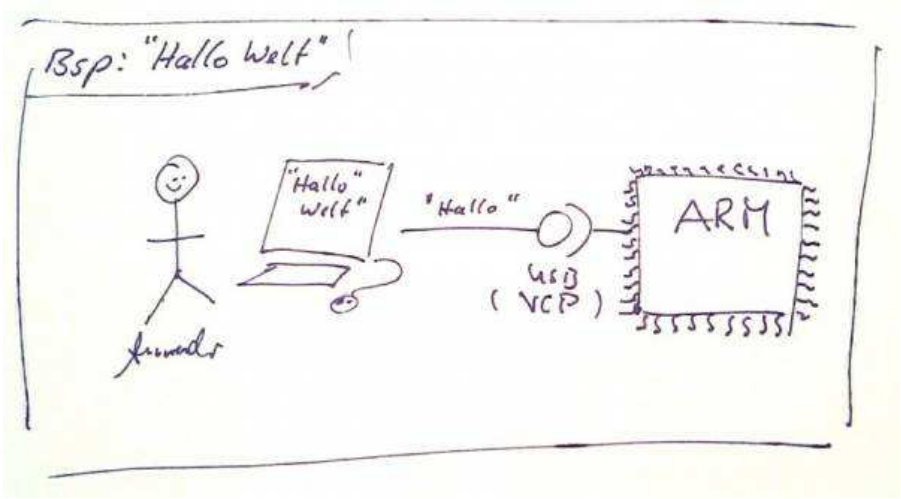
# 5 STM32 Anwendungsbeispiele in C++

## 5.1 Kommunikation mit dem PC

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 STM32F4 verfügt sogar über sechs UART bzw. USART. 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.

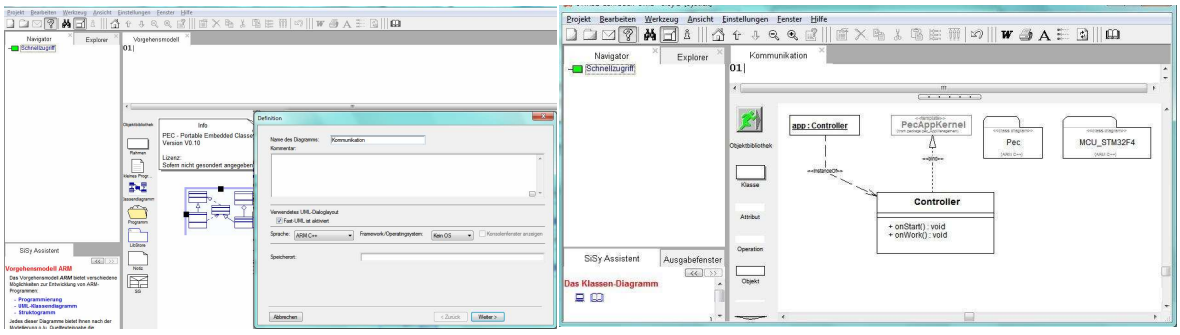
### *Aufgabe*

Die Aufgabe soll darin bestehen, die Zeichenkette „Hallo mySTM32!“ per UART an den PC zu senden. Dort werden die empfangenen Daten in einem Terminal-Programm angezeigt.



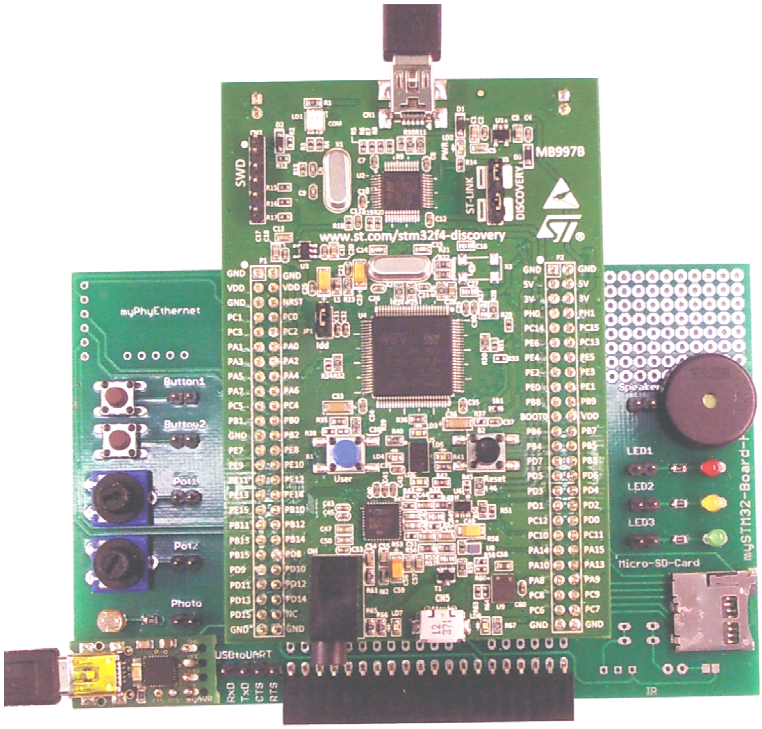

### *Vorbereitung*

Falls Sie jetzt noch das Klassendiagramm geöffnet haben, wählen Sie im Kontextmenü (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, wählen Sie die Sprache ARM C++ und stellen Sie die Hardware (STM32F407) ein. Beim Öffnen des Diagramms laden Sie aus dem SiSy LibStore die Diagrammvorlage *Application Grundgerüst für PEC Anwendungen (XMC, STM32, AVR).* Weisen Sie das Treiberpaket für *STM32F4* zu.



***Lösungsansatz***

Da heutige Rechner kaum noch über eine klassische RS232-Schnittstelle (COM) verfügen, benutzen wir eine USB-UART-Bridge. Auf dem Zusatzboard für das STM32F4-Discovery existiert dafür ein entsprechendes Modul, die USB-UART-Bridge „myUSBtoUART“.




Studieren wir das Datenblatt stellen wir fest, dass sich die Sendeleitung TxD (transmit data) und die Empfangsleitung RxD (receive data) der USART3 als Alternativfunktionen über die Busmatrix auf verschiedene GPIO-Pins legen lassen. Wir wollen die USART3 am Port D verwenden; damit lautet unsere Konfiguration:

- GPIO-Pin D8 ist die Sendeleitung TxD
- GPIO-Pin D9 ist die Empfangsleitung RxD.

▪ ▪ ▪ ▪ ▪

# 6 Fazit und Ausblick

## 6.1 Tutorial

An dieser Stelle verweisen wir auf das STM32 C++ Tutorial:
www.mystm32.de

In diesem Tutorial finden Sie ein weiteres kleines Projekt mit der Idee, einen kleinen Datenlogger zu bauen. Dieser soll über eine längere Zeit, zum Beispiel drei, vier Tage oder eine Woche, Umweltdaten, wie Temperatur und Helligkeit, erfassen sowie die Uhrzeit der Erfassung registrieren.
So wie in den vorangegangenen Beispielen ist auch dieses kleine Projekt didaktisch strukturiert, von der Aufgabenstellung bis zum Test.
Für alle hier aufgeführten Beispiele finden Sie im STM32 C++ Tutorial eine Videozusammenfassung.
Das STM32 C++ Tutorial ist nicht abgeschlossen; es gibt kontinuierlich Erweiterungen sowohl zu theoretischen Aspekten als auch praktischen Anwendungen.

# Literatur und Quellen

mySTM32-Homepage
www.mystm32.de

SiSy-Homepage
www.sisy.de

myMCU-Homepage
www.mymcu.de

Firmen-Homepage von ST
www.ST.com

Datenblätter der STM32-Controller
http://www.st.com/web/en/resource/technical/document/datasheet/DM00037051.pdf

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.microcontroller.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