WHITEPAPER:
MODERNE C++ SOFTWAREARCHITEKTUR

Wiederverwendbare Module

Wir beschreiten seit einigen Jahren den Weg von „Wir machen Embedded C++“ zu „Wir verstehen Embedded C++“. Dabei begegnen uns regelmäßig Stolpersteine auf unserem Weg, die sich bisher nach und nach in Wohlgefallen auflösten. In diesem Artikel möchte ich unsere Erfahrungen teilen und Ihnen Lust auf modulares Architekturdesign und Denken in Modulen machen.

von Florian Seibold

Schon während meinem Elektrotechnik-Studium, habe ich nicht verstanden, warum die Embedded-Software Welt bei „Tools und Programmierhilfen“ der App- und Webwelt gefühlt 20 Jahre hinterherhinkt. Ein Vergleich zwischen App- und Embedded-Frameworks: Man stelle gedanklich auf die App-Seite react-(native) und auf die Embedded-Seite den STM32CubeMX .

Ich bin unseren Chipherstellern sehr dankbar für Ihre Tools, die uns Entwickler bei der Hardwarekonfiguration unterstützen. Sie werden trotzdem zugeben: Das sind zwei Welten.

Das interessante an diesem Vergleich sind nicht die Möglichkeiten, die die jeweiligen Tools bieten, sondern die Vorgehensweise. Chiphersteller binden uns an ihre Software-Services, damit sie mehr Silizium verkaufen können, react hingegen ist eine gigantische Sammlung aus kleinen Helferlein-Algorithmen, die man als Entwickler einfach nutzen kann, um nicht jedes Mal das Rad neu zu erfinden.

Als wir 2017 unsere Vorgehensweise in der Embedded-Software-Entwicklung umgekrempelt haben, gab es noch keine Library für das tägliche Brot im Embedded-Bereich. Daher mussten wir fast alles selbst machen.

Seitdem stellen wir uns vor der Implementierung einer Aufgabe folgende Fragen:

- Gibt es in der Standard-Library Hilfen, mit denen ich meine Aufgabe weitestgehend lösen kann? Wenn ja, hat sich das Problem meistens schon von selbst erledigt.

- Ist mein Lösungsansatz generisch genug, damit auch andere meine Implementierung sinnvoll nutzen können und sich der Aufwand rentiert?

- Ist mein Lösungsansatz für alle Anwendungsfälle, für die das Modul beschrieben ist, effizient in Footprint und CPU-Last?

Warum stellen wir uns diese Fragen?

In den letzten Jahren haben sich intern immer wieder unterschiedliche Trends abgezeichnet, die es zu kontrollieren galt. Zum Beispiel gab es eine Zeit, in der das Implementer-Pattern Teil jedes zweiten Lösungsansatzes war. Mit diesen vier einfach zu stellenden Fragen versuchen wir Module für uns zu schaffen, die sinnhaft sind. Sinnhaft, weil sie nicht die Standard Library nachimplementieren, an anderen Stellen wiederverwendet werden können und möglichst effizient sowie Embedded-tauglich umgesetzt sind.

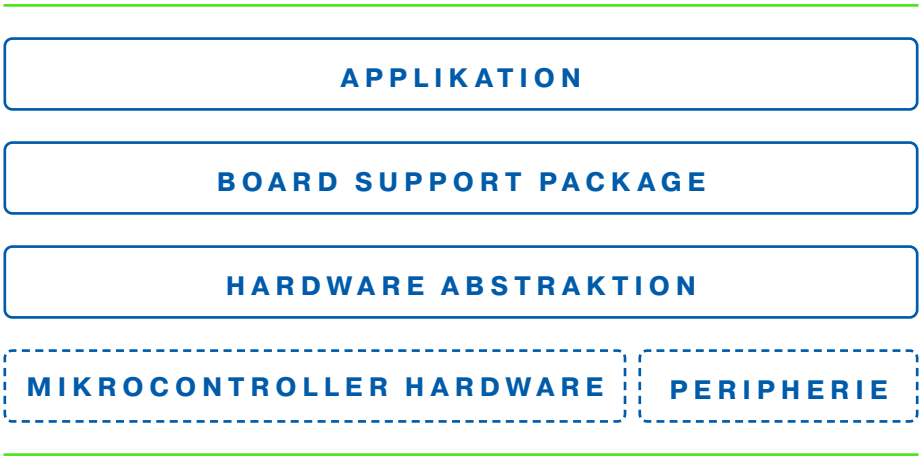
Das alles hat unser Vorgehen erst einmal auf den Kopf gestellt und dann neu geordnet. Die Wertigkeit, Robustheit und Verständlichkeit von Entwicklungsergebnissen ist spürbar gestiegen. Auch unsere Kunden danken es uns.

STANDARDISIERTE UND LEICHT VERSTÄNDLICHE ARCHITEKTUR

Über Architektur kann man streiten – und das ist auch gut so. Ich glaube daran, dass jede konstruktive Diskussion, auch wenn sie teilweise hitzig ist, früher oder später einen Sprung an Source-Code-Professionalität nach sich zieht.

Ob eine Architektur auch verständlich und nutzerorientiert ist, zeigt sich erst, wenn Menschen damit arbeiten, die an der Entwicklung nicht beteiligt waren. In unserem Fall holen wir regelmäßig Feedback von unseren Kunden ein und fragen nach, inwieweit sie verstehen, was in dem Code passiert und wo sie welche Funktionalität vermuten. Noch eine Stufe tiefer gehen wir mit neuen Mitarbeitern, die noch eine externe Sichtweise haben und uns helfen können Strukturprobleme aufzulösen, die wir selbst nicht mehr sehen. Iterativ sind wir zu folgendem Zwischenergebnis gekommen: Architektur, die immer wieder für unsere Kunden und uns funktioniert, ist in den Basis-Layern immer gleich aufgebaut und damit generisch.

GENERISCHE BASISARCHITKEKTUR



1. Applikation:

Die Applikation ist in den meisten Projekten über viele Klassen und Files verteilt. Unabhängig dessen gibt es einen Entry- und Steuerpunkt in allen mir geläufigen Applikationen. Wir betiteln diesen oft mit App.

2. Board Support Package:

Das BSP (Board Support Package) spiegelt die gesamte Hardware wieder, die Teil der projektspezifischen Baugruppe ist. Den größten Teil des BSPs beansprucht in der Regel die Peripherie des Mikrocontrollers wie GPIOs, ADCs, USB oder CAN. Daneben finden sich im BSP alle weiteren Komponenten wieder, die ebenfalls auf dem Board verbaut sind, wie eine externe RTC (Real Time Clock), externe Speicher oder Sensoren die z. B. via SPI oder I2C angebunden sind. Anspruch des BSPs ist, dass es die Gesamtheit der Hardware wiederspiegelt und dass bei Hardwareänderungen auch nur dieses verändert werden muss. Ein übliches Beispiel ist der Tausch des Speichertreibers, da der Vorgänger Chip abgekündigt wurde.

3. Hardware Abstraktion:

Der HAL (Hardware Abstraktion Layer) beschreibt mittels dessen Treiber die Funktionalität der Chip-Hardware. Fast alle Mikrocontrollerhersteller stellen Treiberpakete für ihre Derivate bereit und selbst Sensorhersteller fangen an, Treiber für Ihre Produkte zur Verfügung zu stellen.

Folgend ein minimalistisches Beispiel eines Board Support Packages mit einem Stm32F4 und lediglich einem GPIO. Die zugehörige Applikationsklasse wäre analog aufgebaut. Die Applikation ist in den meisten Projekten über viele Klassen und Files verteilt. Unabhängig dessen gibt es einen Entry- und Steuerpunkt in allen mir geläufigen Applikationen. Wir betiteln diesen oft mit App.

```
#include "HardwareAbstraction/Stm32/stm32f4gpio.h"
#include "main.h"
#include "stm32f4xx_hal.h"
class Bsp {
public:
  static Bsp* instance() {
       static Bsp instance;
       return &instance;
  }
  struct Gpios {
       semf::Stm32F4Gpio gpio1;
  };
  Gpios& gpios() {
       return m_gpios;
  }
  void init() { … }
private:
  Bsp() { … }
  Gpios m_gpios;
};
```

Als Lesson Learned folgend zwei Ansätze, die wir versucht haben, aber für uns nicht funktioniert haben:

Abstraktion des Microcontrollers und des Boards trennen:

Es klingt charmant, die Abstraktion des Mikrocontrollers und des gesamten Boards in zwei Schichten zu teilen: in ein MSP (Microcontroller Support Package) und BSP. Es kann intuitiv verstanden werden und die einzelnen Module sind übersichtlicher. Das Problem bei dieser Struktur kristallisiert sich heraus, wenn Funktionalitäten des Mikrocontrollers ausgelagert werden oder der Mikrocontroller durch ein größeres Derivat getauscht und damit externe Funktionalität im Mikrocontroller selbst gelöst werden kann. Beispiele sind RTCs oder Flash Speicher. So wäre die Struktur stark projektabhängig und die Auffindbarkeit der Objekte nicht mehr eindeutig, was dem generischen Ansatz widerspricht und sich somit für uns nicht bewährt hat.

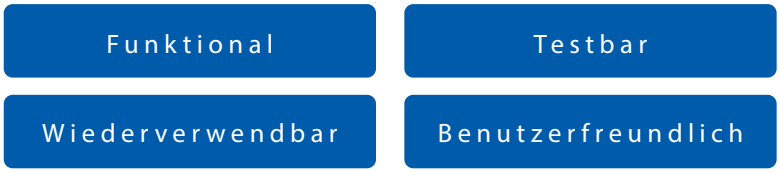
BSP und APP nicht als Klassen umsetzen:

Die BSP und APP Klassen könnten theoretisch in Form von statischen Objekten in einem jeweils eigenen Namespace schneller und für C++-Einsteiger verständlicher geschrieben werden. In erster Näherung klingt das gut. Kritisch ist jedoch der Zeitpunkt der Objekterstellung während der Initialisierung. Objekte, die von der Mikrocontroller-Hardware abhängig sind und diese benutzen, dürfen erst nach erfolgter Hardwareinitialisierung angelegt werden. Der Zeitpunkt der Erstellung kann einfach und stabil kontrolliert werden, indem man die BSP und APP Klassen als Singelton implementiert.

MODULE, DIE MAN GERNE WIEDERVERWENDET

Unser Ziel ist es, kleine und verständliche Software-Module zu produzieren, die in unterschiedlichen Projekten und Anwendungsfällen eindeutig Mehrwert bieten und die man als Entwickler gerne wiederverwendet.

Soweit die Theorie. In der Praxis braucht es eine Philosophie und klare Regeln, nach denen entwickelt wird. Für mich hat sich herauskristallisiert, dass alle Module folgende vier Kriterien klar erfüllen müssen:



Funktional:

Software ist funktional, wenn sie effektiv und exakt diejenige Aufgabe erledigt, für die sie beschrieben ist. Zusätzlicher „Schnick-Schnack", der nicht eindeutig Mehrwehrt für das Modul und dessen Nutzer erzeugt, hat auch nichts darin zu suchen.

Wiederverwendbar:

Software ist wiederverwendbar, wenn sie so generisch aufgebaut ist, dass sie nicht nur für exakt einen Anwendungsfall sinnvoll genutzt werden kann, sondern für eine ganze passende Kategorie an Anwendungsfällen.

Testbar:

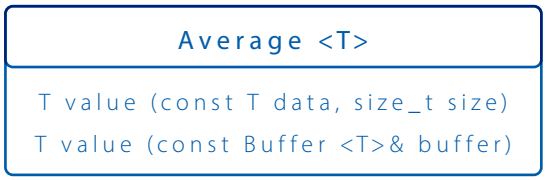
Unit-, Integrations- und Systemtests sind auch im Embedded-Software Bereich nicht mehr wegzudenken. Egal ob es sich um Haftungsthemen für Sicherheitsfunktionen oder die Reduktion von Reklamationen geht, gut gemachte Unit-Tests können viel bewirken. Dabei sind Unit-Tests nur so gut, wie auch die Struktur der Klasse selbst, die sie abtesten sollen. Nur Funktionen mit geringer Komplexität können stabil und nachvollziehbar abgetestet werden.

Benutzerfreundlich:

Mein Lieblingskriterium, wenn auch vermutlich das schwierigste der Vieren: Es muss Spaß machen, das Modul zu benutzen. Die API, die Dokumentation und die Beispiele müssen so gut verständlich sein, dass Externe das Modul lieber benutzen, als sich selbst Gedanken über eine mögliche Implementierung zu machen.

BEISPIEL: AVERAGE-KLASSE

Als kleines Beispiel soll hier die Average-Klasse dienen. Die beiden statischen Funktionen der Klasse bilden einen buchhalterisch korrekten Mittelwert aus dem übergebenen Datenarray oder aus dem Inhalt eines Puffer-Objekts, der zum Beispiel ein Ringpuffer sein kann, und gibt diesen zurück. Buffer ist in diesem Beispiel ein Interface für alle Puffer-Implementierungen. Die Rechenoperationen in den beiden value()-Funktionen sind für Embedded optimiert.



Klassendiagramm der Average-Klasse

Ich denke es ist klar, was man von der Klasse an Funktionalität erwarten kann und wie man sie verwendet. Was Sie jetzt nicht sehen, aber sicherlich vermuten, kann ich Ihnen versichern: Die Klasse ist leicht testbar und leicht wiederverwendbar.

HARDWAREABHÄNGIGKEITEN AUFLÖSEN

Die bisher größte Aufgabe für uns war und ist es, Hardwareanhängigkeiten performant und nachvollziehbar aufzulösen. Wie in Abbildung 1 sichtbar, haben wir einen Hardware-Abstraction-Layer zwischen den Mikrocontroller-Hersteller HAL und das Board Support Package eingefügt. Dessen Aufgabe ist es, die Hardwareabhängigkeiten in den unteren Layern zu kapseln, damit die Applikation nur noch hardwareunabhängige Interfacefunktionen zur Verfügung hat.

Als ein einfaches Beispiel soll hierdas GPIO Interface dienen:

```
class Gpio {
public:
	enum Direction {
		INPUT = 0,
		OUTPUT_PUSHPULL,
		OUTPUT_OPENDRAIN
	};
	enum PullUpPullDown {
		NO_PULLUP_PULLDOWN = 0,
		PULLUP,
		PULLDOWN
	};
	virtual void set() = 0;
	virtual void reset() = 0;
	virtual bool state() const = 0;
	virtual Direction direction() const = 0;
	virtual void setDirection(Direction direction) = 0;
	virtual PullUpPullDown pullUpPullDown() const = 0;
	virtual void setPullUpPullDown(PullUpPullDown pullUpPullDown) = 0;
};
```

Zugegebenermaßen steigt die Komplexität der Interfaces mindestens linear mit der Komplexität der zu abstrahierenden Hardware an. Errata-Sheet-Fehler sauber in einer Softwarearchitektur zu platzieren, verlangt uns immer wieder Geduld und Kreativität ab.

MODERNE C++ SOFTWAREARCHITEKTUR
ZUSAMMENFASSUNG UND AUSBLICK

Ich bin zufrieden und glücklich den Schritt zu einer überlegten Architektur vor über drei Jahren gemacht zu haben. Zum einen ist die dabei entstehende Qualität ein großer Motivator für das gesamte Team. Zum anderen befähigt uns das Wissen, das Vorgehen und die Codebasis sicherheitskritische Funktionen und Produkte effizient zu entwickeln. Neben den logischen Schlussfolgerungen bin ich mir sicher, dass die Mehrheit der Entwickler mir Recht gibt: „Es macht richtig Spaß, wenn etwas einfach und mit wenig Zeitaufwand funktioniert."

Aktuell sind wir an der Entwicklung eines automatischen Mikrocontroller HAL und Hardware-Testers und einer Low-Footprint Errorhandling-Lösung dran.

Florian Seibold, Inhaber und Geschäftsführer von querdenker engineering studierte an der Hochschule München Elektro- und Informationstechnik. Seither befasst er sich in seinen Unternehmen damit, wie man Entwicklung teil-automatisieren kann, um Ziele in kürzerer Zeit und mit höherer Qualität erreichen zu können.

KONTAKT

Sie interessieren sich für das Thema Softwarearchitektur im Embedded Bereich oder suchen Unterstützung für Ihre Projekte? Gerne evaluiere ich mit Ihnen diie Möglichkeiten bei einem kostenfreien Kennenlern-Gespräch. Sollten Sie auf der Suche nach einem Framework sein, das Ihnen das Leben in der C++ Welt einfacher macht, besuchen Sie die Webseite unserer hauseigenen C++ Middleware Library **semf**.

Tel.:
(0) 7807 / 890 80 10
Email:
info@querdenkerengineering.de

Soziale Medien:
www.linkedin.com/company/querdenker-engineering

Entwicklungsdienstleisung im Web
www.querdenkerengineering.de

semf – C++ Middleware Library im Web
www.semf.io