Hochschule
Kaiserslautern
University of
Applied Sciences

Informatik und
Mikrosystemtechnik
Zweibrücken

Bericht Praxisprojekt SS 19

Bachelor-Studiengang (PO Version 2011)

Angewandte Informatik

Titel:

Anpassung eines C++ µC-Softwareframeworks auf eine neue µC-Generation

Adjustment of a C++ µC-softwareframework on a new µC-generation

vorgelegt von: Keven Klöckner

vorgelegt am: 31. Mai 2019

vorgelegt bei: Prof. Dr.-Ing. Wilhelm Meier

durchgeführt bei: HS Kaiserslautern - Fachbereich IMST
Amerikastraße 1
66482 Zweibrücken
Deutschland

Bericht Praxisprojekt

Inhaltsverzeichnis

1. Einleitung . . . 1
2. Tätigkeiten . . . 2
   2.1. Projektstart . . . 2
   2.2. Woche 1 . . . 3
   2.3. Woche 2 . . . 5
   2.4. Woche 3 . . . 7
   2.5. Woche 4 . . . 7
   2.6. Woche 5 . . . 9
   2.7. Woche 6 . . . 11
   2.8. Woche 7 . . . 12
   2.9. Woche 8 . . . 13
   2.10. Woche 9 . . . 14
   2.11. Woche 10 . . . 15
3. Design und Aufbau der Library . . . 16
   3.1. HW Files . . . 16
      3.1.1. Der Header . . . 16
      3.1.2. Die Komponente . . . 16
      3.1.3. Die Instanzen . . . 17
   3.2. Die Microcontrollerklassen . . . 19
      3.2.1. Header . . . 19
      3.2.2. Klasse . . . 20
      3.2.3. Spezialfall Port . . . 21
      3.2.4. Komponenten . . . 22
   3.3. AtmegaZero Struktur . . . 23
   3.4. RessourceController . . . 23
      3.4.1. RCComponent . . . 23
      3.4.2. Die Kernklasse . . . 24
   3.5. TWI Command und Callback Aufbau . . . 26
4. Benutzung der Library . . . 30
   4.1. Programmierung . . . 30
   4.2. Grundsätzliches . . . 30
   4.3. Basics . . . 30
   4.4. RessourceController . . . 32
   4.5. Eventsystem . . . 33
   4.6. TWI . . . 35
      4.6.1. Beispielbehandlung TWI Address-NACK . . . 35
      4.6.2. Lesen mit FiFo, keine Interrupts . . . 36
      4.6.3. Schreiben mit FiFo, keine Interrupts . . . 37

4.6.4. Lesen nicht blockierend, keine Interrupts, keine FiFo . . . 37
4.6.5. Lesen nicht blockierend, Scoped-Variante . . . 38
4.6.6. Schreiben nicht blockierend, Scoped-Variante . . . 39
4.6.7. Schreiben nicht blockierend, FiFo mit Interrupts . . . 40
4.6.8. Lesen nicht blockierend, FiFo mit Interrupts . . . 40
4.6.9. Lesen und schreiben nicht blockierend, FiFo mit Interrupts . . . 41
4.7. SPI . . . 42
4.7.1. Schreiben, nicht blockierend, FiFo ohne Interrupts . . . 42
4.7.2. Schreiben, nicht blockierend, FiFo mit Interrupts . . . 43
4.8. USART . . . 44
Quellen . . . 46

Kapitel 1. Einleitung

In diesem Projekt geht es darum eine Library für eine neue Microcontroller-Familie anzupassen und zu entwickeln, als Grundlage dient die Library aus BMCPP [BMCPP]. Da die Peripherie und die Microchip Library wesentlich unterschiedlich sind im Vergleich zu den älteren, wie z.B. dem Atmega328, mussten auch in den grundsätzlichen Strukturen wie z.B. Port diverse Änderungen vorgenommen werden. Die 0 Serie von Microchip besitzt eine wesentlich umfassendere Peripherie, diese beinhaltet vor allem auch ein Eventsystem welches im Rahmen dieses Projektes betrachtet wird. Ziel des Projektes ist es eine moderne und leicht erweiterbare Library für diese µC-Familie zu schaffen und diese dabei möglichst kompatibel zu der BMCPP Library zu gestalten. Des Weiteren soll das Projekt Cross-Plattform fähig sein, d.h. es soll auf Windows sowie Linux kompilieren, diese Vorraussetzung wird mit CMake als Build-Tool geschaffen. Zunächst wird hier die Tätigkeit im Rahmen des Projektes dargelegt, hierzu wird Chronologisch der Fortschritt beschrieben und anhand von Codebeispielen verdeutlicht. Als nächstes wird der Aufbau und das Design sowie spezielle Eigenschaften (wie z.B. die automatisch generierten Dateien) der Library genauer beleuchtet. Zum Schluss wird Betrachtet wie das Software-Framework benutzt wird, dies wird anhand von Codebeispielen und Erklärungen veranschaulicht.

Kapitel 2. Tätigkeiten

2.1. Projektstart

Zu Beginn des Projekts habe ich mir eine Struktur für die Ports und Register überlegt die mit dem BMCPP-Aufbau möglichst kompatibel ist, allerdings auch die Verwendung der von Microchip mitgelieferten C-Strukturen ermöglicht.

Abbildung 1. Auszug aus Register.hpp [ZLib]

```
    /*
    function to cast the structs to Register
    */
    [[nodiscard]] static inline volatile Register& getRegister(volatile mem_width&
reg) {
        return reinterpret_cast<volatile Register&>(reg);
    } ①
```

① Konstruktion des Registers durch die C-Struktur

Hier war der Hintergedanke die neue Library möglichst kompatibel sowie flexibel hinsichtlich Änderungen von Microchip zu gestalten. Des Weiteren wurde bei der Erstellung der Software-Architektur auf die leichte Erweiterbarkeit sowie Cross-Plattformfähigkeit Wert gelegt, deswegen wurde als Build-Tool CMake gewählt. Zunächst hatte ich nach dem Grundgerüst (Port, Pins und Register) das neue Eventsystem betrachtet.Das Eventsystem der 0 Serie ist jedoch Abhängig von der Peripherie eines µCs und somit für jeden µC einmalig. Aus dieser Erkenntnis heraus entwarf ich die Schnittstelle für das Eventsystem möglichst Benutzerfreundlich und sicher, jedoch sind bei der Implementierung einige Anpassungen von Hand notwendig (z.B. fallen je nach Gerät wegen fehlender Peripherie einige Generatoren oder auch User weg).

Abbildung 2. Auszug Atmega4808EventSystem.hpp [ZLib]

```
struct _generalGenerators { ①

    #define ic(x) typename utils::integralConstant<mem_width,x> ②

    using pdi = ic(EVSYS_GENERATOR_UPDI_gc);
    using rtc_ovf = ic(EVSYS_GENERATOR_RTC_OVF_gc);
    using rtc_cmp = ic(EVSYS_GENERATOR_RTC_CMP_gc);
    //lines omitted...
```

① Generatoren die in allen Channels vorhanden sind

② Makro für eine kürzere und übersichtlichere Schreibweise

Abbildung 3. Auszug aus Atmega4808EventSystem.hpp [ZLib]

```
template<uint8_t number>
struct generatorChannel;

template<>
struct generatorChannel<0> {

    using generals = _generalGenerators;
    using RTCDivGenerator = type1RTC; ①

    template<uint8_t pinNumber>
    struct PortAGenerator {
        static_assert(pinNumber < 8, "only pins [0,7] allowed");
        using portAPin = utils::integralConstant<mem_width,(0x40+pinNumber)>;
    }; ②

    template<uint8_t pinNumber>
    struct PortBGenerator {
        static_assert(pinNumber < 8, "only pins [0,7] allowed");
        using portBPin = utils::integralConstant<mem_width,(0x48+pinNumber)>;
    };
};
```

① Es gibt 2 Gruppen RTC Generatoren

② Pro Port gibt es 8 Einstellbare Generatoren

Das hier gezeigte Design ist leicht zu modifizieren und verhindert das der Benutzer den Kanälen falschen Werten übergibt (z.B. kann der Benutzer nicht versehentlich einen PortC-Pin als Generator in Channel 0 einstellen). Zum testen und validieren während des Projekts habe ich ein AVR-Development Board genutzt und die Programme mit dem eingebauten debug-Chip und Atmel-Studio auf den µC geflasht. Eine weitere Aufgabe war es daher andere Programmiermethoden für die 0 Serie zu testen, da dieser mit der neuen UPDI - Methode Programmiert werden.

Hierzu evaluierte ich zwei Methoden:

- https://github.com/ElTangas/jtag2updi hier wird ein Atmega328 oder ein vergleichbarer Chip zu einem Programmer umfunktioniert, hierzu kann z.B. ein fertiges Arduino-Board genutzt werden.
- https://github.com/mraardvark/pyupdi hier wird mithilfe eines USB-UART Adapters geflasht, hier muss darauf geachtet werden dass der Widerstand stimmt, sollte der Widerstand nicht funktionieren muss eine Diode mit der Anode zu UPDI und Kathode zu Tx verbunden werden.

2.2. Woche 1

Nach Besprechung des Projekts musste die bisherige Struktur abgeändert werden, sodass die Komponenten anhand von den verschiedenen Registertypen (Flag, Toggle, Data, Control, R, RW) beschrieben werden können. Zunächst habe ich die Port Komponente dahingehend angepasst, währenddessen fiel mir auf, dass durch mehrere static inline Variablen unnötiger Overhead

entstand. Das Problem war hier, dass der GCC doppelte Referenz-Variable durch eine jump-table auflöst statt eine Variable wegzuoptimieren (Clang entfernt eine der doppelten Variablen). Nach einigem herumprobieren löste ich das Problem durch die Ersetzung der Variablen durch eine function-reference.

Abbildung 4. Auszug aus ATmega4808Port.hpp [ZLib]

```
        struct ports{
            NoConstructors(ports);
            struct A{
                NoConstructors(A);
                // [[gnu::always_inline]] static inline auto& value { return PORTA; }
①
                [[gnu::always_inline]] static inline auto& value() { return PORTA; }
②
                struct pins {
                    static inline constexpr Pin pin0{0}, pin1{1}, pin2{2}, pin3{3},
pin4{4}, pin5{5},pin6{6}, pin7{7};
                };
                //lines omitted
```

① Alte Variante, produziert Overhead durch jump-tables

② Version als function-reference, wird immer inlined → kein Overhead

Danach habe ich die anderen Komponenten entsprechend angepasst, die SPI Implementierung wurde getestet mithilfe eines Logic Analyzers. Dann Habe ich einen Parser geschrieben der es ermöglicht durch String Parameter geeignete Strukturen für die Komponentenbeschreibungen zu erstellen. Der nächste Schritt war die Suche nach einem geeigneten cpp XML-Parser um die atdf (aus [ASF]) Dateien auszulesen, ich entschied mich dann für den pugi xml Parser ([PUGI]). Danach analysierte ich die Struktur der `atdf` Datei und entwarf einen Prototypen um die dort beschriebene Hardware in C++ Strukturen zu schreiben. Nach längerem einlesen habe ich schließlich herausgefunden wie das Problem der physisch nicht existenten IO-Ports gelöst werden kann (`signals` Node), das Programm wurde demnach erweitert. Somit können zur Compile-Zeit schon nicht existente Komponenten ausgeschlossen werden.

Abbildung 5. Auszug aus der Atmega4808.atdf [ASF]

```
  <instance name="USART0">
    <register-group address-space="data" name="USART0" name-in-module="USART" offset=
"0x0800"/>
    <signals>
      <signal field="PORTMUX.USARTROUTEA.USART0" function="USART0_ALT" group="RXD"
pad="PA5"/>
        #...
      <signal field="PORTMUX.USARTROUTEA.USART0" function="USART0" group="XDIR" pad=
"PA3"/>
    </signals>
  </instance>
```

Nachdem das Programm auf meinem Windows System korrekt funktionierte testete ich es auch auf Linux, da es auch dort ohne Probleme lief mussten hier keine Änderungen erfolgen.

Danach folgten einige Bugfixes:

- Mapping der Bitfeld Namen in korrekte Präprozessor Makros der Microchip library: bisherige Prüfung ob die Maske gleich 0xff ist war unzureichend und nicht portabel. → Verwendung von std::bitset
- Benennung der Pins in den einzelnen Gruppen: Die Verwendung der Pin Nummer um diesen zu benennen könnte zu Problemen (mehrfache Definitionen) führen. → Verwendung eines gruppenlokalen Counters.
- Bei einem Test fiel ein logischer Bug durch den der Pfad im Namespace auftauchte. → Trennung von Name und Pfad.
- Einige "register-group"-Nodes hatten "mods"-Kinder vor Bitfield, führte zu fehlenden Enums. → Zusätzliche Prüfung und Schleife wurden notwendig um den Fall zu behandeln.
- Ein logischer Fehler führte dazu, dass immer alle Pins jeder Gruppe einer Funktion vorkamen. → Komplexeres Verfahren zum einsortieren wurde notwendig.

Abbildung 6. Auszug aus parser/main.cpp [ZLib]

```
auto bs = std::bitset<32>(static_cast<size_t>(node2.attribute("mask").as_int())); ①
//if (node2.attribute("mask").as_int() == 0xff) ②
if (bs.count() > 1) { ③
    for (uint32_t i = 0; bs.test(i) && i < 32; i++) {
        mbuilder.addEnumEntry(
                utils::toCamelCase(node2.attribute("name").as_string()) + std
::to_string(i),
                modName + "_" + node2.attribute("name").as_string() + std::to_string(
i) + "_bm");
    }
}
```

① std::bitset für eine Abstrakte Verwendung von Bit-Operationen

② Alte Version prüft nur ob alle Pins für das Enum verwendet werden

③ Neue Version testet ob es mindestens 2 sind, deutet dann auf ein Enum hin

Dann habe ich zum Evaluierungszweck die automatisch generierten Dateien für den Atmega 4808 (Port und SPI) implementiert, die Port Variante wurde zum Zweck der Lesbarkeit leicht angepasst. Abschließend habe ich noch einige Funktionen zum besseren Verwenden der Pins eingebaut.

2.3. Woche 2

Nach der letzten Besprechung habe ich das SPI interface leicht angepasst:

- doIfSet Methode wurde implementiert, diese führt eine Funktion aus wenn die angegebenen Bits im Flag Register gesetzt sind.
- Die Namensgebung der Spezialisierung wurde auf die treffenderen Varianten `blocking` und `notBlocking` geändert (mit doIfTest kann auch ohne Interrupt "nicht blockierend" und/oder

gelesen/geschrieben werden).

Abbildung 7. Auszug aus SPI.hpp [ZLib]

```
template<auto& funcRef, typename... FlagsToTest>
requires(utils::sameTypes<InterruptFlagBits, FlagsToTest...>() && etl::Concepts
::Callable<decltype(funcRef)>)
static inline auto doIfSet(FlagsToTest... flags) {
    using retType = decltype(funcRef());
    if (reg<InterruptFlags>().areSet(flags...)) { ①
        if constexpr (! std::is_same_v<retType,void>) ②
            return funcRef();
        else
            funcref();
    }
    if constexpr (! std::is_same_v<retType,void>)
        return retType{}; ③
}
```

① Im Flag Register nachsehen ob das Flag gesetzt ist

② Nachschauen ob der Rückgabetyp `void` ist

③ Rückgabe Default Wert des Rückgabetyps

Für die Flag Register habe ich noch eine RW Spezialisierung hinzugefügt weil es auch flag-Register gibt welche getogglt werden können.Im Parser gab es einen "Fehler" der Dazu führte dass wenn ein DEFAULT im Attribut `values` Stand, das Präprozessormakro inkorrekt gedeutet wurde (DEFAULT musste ausgeschnitten werden). Ein weiterer Bug wurde behoben bei der Auswertung der Registertypen: es musste noch überprüft werden ob Control und Flag Register auch tatsächlich Bitfelder als Kinder haben. Der nächste Schritt war die Vereinfachung der internen Struktur, hierzumussten innerhalb der HW Beschreibung die `hw_abstraction` includes sowiedie Vorwärtsdeklarationen entfernt werden. Damit wurden für die bisherigenSchnittstellen SPI und Eventsystem weitreichende Bearbeitungen notwendig,danach wurden diese Schnittstellen auf das BMCPP Model angepasst.In diesem Zuge wurden weitere Anpassungen beim Parser notwendig, dadieser nun nicht mehr die AVR::port Schnittstelle nutzen konnte, der Parser hat hierzu den Port als Sonderfall zu behandeln. Dann wurde ich noch auf ein weiteres Problem aufmerksam, dies betrafwieder die Microchip HW-Beschreibung, auch bei anderen Komponenten alsdem Port sind teilweise Pins beschrieben welche physisch nicht existent sind.→ Lösung durch ausklammern einer Bedingung.

Abbildung 8. Auszug aus parser/main.cpp [ZLib]

```
//if (modName != "PORT" && utils::contains(pins_available, (sig_pad)))
if (utils::contains(pins_available, (sig_pad)))
```

Die Abfrage auf ungleich "PORT" hatte zur Folge dass auch nicht existente Pins hinzugefügt werden konnten (z.B. beim ADC).Als nächstes wandte ich mich an die Implementierung des TWI Interfaces, daich vorher noch nicht damit in Kontakt gekommen bin musste ich michzunächst einlesen. Der nächste Schritt war das testen des TWI nach demDatenblatt mithilfe eines Logic

Analyzers. Nachdem ich die TWI Komponentekorrekt Konfiguriert hatte begann damit das Interface festzulegen.

2.4. Woche 3

Zunächst führte ich die Implementierung des TWI fort,hierzu musste ich verschiedene Tests mit dem Logic Analyzerdurchführen.Bei den Tests hatte ich einige Bugs feststellen können, so z.B. hatte icham Ende einer Transaktion keine Stop-Condition ausgelöst.Das nächste Problem welches mir auffiel ist dass ich keine Möglichkeiteingebaut hatte den `portmux` zu benutzen.dies regelte ich indem ich die Möglichkeit einbaute eine "Alternative" alsTypnamen zu wählen.Ein weiterer Parser Bug führte dazu dass Makro Namen falsch geparsedwurden, dieser wurde behoben durch eine weitere Abfrage.Des Weiteren änderte ich die Komponenten so ab dass man zwischen `Blocking` und `nonblocking` wählt und daraus dann der Zugriff resultiert(`nonblocking` Funktionen entfallen).Dann begann ich damit die Funktionen umzugestalten, sodass die neueSchnittstelle möglichst kompatibel zu der BMCPP Schnittstellewird.Die nächste Änderung betraf die Einführung einesResssource-Controller, hier wurden noch einige Änderungen innerhalbder Atmega und deren HW Klassen notwendig. Der Ressource-Controller hat zum Ziel die Ressourcen der einzelnen Komponenten auf Kollisionen zu prüfen und somit zur Compile-Zeit eine Doppelbelegung auszuschließen.

Abbildung 9. Auszug aus ATmega4808SPI.hpp [ZLib]

```
template<bool dummy>
struct inst<0,dummy> { ①
    [[nodiscard,gnu::always_inline]] static inline auto& value()  { return SPI0;}

    template<auto N, bool dummy1 = true>
    struct alt;

    template<bool dummy1>
    struct alt<1, dummy1> { ②
    //lines omitted
```

① Die Instanz z.B. SPI0

② Die Portmux Variante z.B. SPI0 variante 0 (Default Portmux Einstellung)

Die HW Strukturen mussten auf das obige Schema abgeändert werden, sonst ist es nicht möglich generisch auf diese zuzugreifen, dies wiederum ist aber die Grundvorraussetzung für den Ressource-Controller.

2.5. Woche 4

Zunächst hatte ich das CMake-file angepasst sodass es aufunterschiedlichen Plattformen ohne Probleme Kompiliert, dazu mussten diefesten Pfade entfernt werden. In der neueren CMake Version werden dieSystemvariablen (Path) genutzt um den Compiler zu finden. Im CMake Filewurde ein Bug gefixt der dazu führte, dass (u.a.) das Root-directory als includePfad hinzugefügt wurde und zu Permission Problemen führte. Als Lösung wurde die Verzeichnisstruktur leicht abgeändert.

Abbildung 10. Auszug aus CMakeLists.txt [ZLib]

```
set(INC_PATH     "${BASE_PATH}/inc")
# set(LIB_DIR_PATH "") ①
set(LIB_DIR_PATH "${BASE_PATH}/lib")
set(SRC_PATH     "${BASE_PATH}/src")
```

① Der Leere Pfad wird von CMake als root interpretiert

Der Ressourcecontroller wurde für TWI und SPI implementiert, somit **muss** dieser genutzt werden um auf entsprechende Library Funktionenzuzugreifen.Für den Ressource-Controller wurde noch eine Testdatei hinzugefügt, dieseüberprüft die Richtigkeit des RC zur Compile-Zeit. Bei der Implemetierungwurde u.a. Ein Bug behoben welcher dazu führte dass die Rekursion durchdas Parameterpack frühzeitig endete und somit fehlerhaft war. Nach demRessourceController begann ich damit die SPI-Komponente umzuschreibensodass diese mit der Fifo aus der vorhandenen Library betrieben werdenkann. Dabei fiel mir auf dass Spezialisierungen für gewisse Klassen fälligwerden, das bedeutet es muss zwischen den "neuen" und den "alten"Microcontrollern unterschieden werden. Grundsätzlich wird dies mithilfe eines Concepts erledigt, dieses stellt fest ob es sich um einen MCU der 0 Serie handelt, folglich wird dann die entsprechende Spezialisierung ausgewählt.

Abbildung 11. Auszug aus scoped.h [BMCPP]

```
    template<typename T = Transaction, bool Active = true, typename MCU =
DefaultMcuType, typename F1 = void, typename F2 = void>
    using Scoped = std::conditional_t<etl::Concepts::ZeroAVR<MCU> && ①
                                      !std::is_same_v<DisbaleInterrupt<ForceOn>,T> &&
                                      !std::is_same_v<DisbaleInterrupt<
NoDisableEnable>,T>, ②
                                      details::Z_Scoped<T, Active, MCU, F1, F2>, ③
                                      details::_Scoped<T, Active, MCU, F1, F2>>; ④
```

① Prüfung ob die MCU zur 0 Serie gehört

② Prüfung auf ForceOn/NoDisableEnable → hier werden nur `sei()` und `cli()` genutzt

③ Spezialisierung für die neue MCU-Familie

④ Spezialisierung für die alte MCU-Familie

Nachdem ich dann die Klasse Scoped (benutzt in FiFo) kompatibel zur neuenLibrary gemacht hatte, baute ich noch ein, dass SPI als -read, writeOnly oderim readWrite Modus betrieben werden kann (spart Ressourcen). Als nächstesbegann ich damit USART einzubauen, dabei fiel (nach importieren dergenerierten Header Datei) ein weiterer Bug des Parsers auf.Der Parser identifizierte Komponenten fehlerhaft wenn innerhalb einer Instanzauch nur 1 Pin Invalide war, jedoch sollte dann nur die gruppe nichthinzugefügt werden.

Abbildung 12. Auszug aus parser/main.cpp [ZLib]

```
if (utils::contains(pins_available, (sig_pad)) && grpValid) {
    tmp.push_back(utils::triple<>{sig_func, sig_group, sig_pad});
} else {
    tmp.clear(); ①
    break;
}
```

① Wenn irgendein Pin fehlerhaft ist → entferne alle Einträge und breche Generierung ab

Aus Lesbarkeits- und Portierbarkeitsgründen tauschte ich (soweit möglich)eigene Metafunktionen durch die std-Varianten [libcpp] aus. Des Weiteren separierteich Port und PortPin (PortPin bekam eine eigene HeaderDatei). Für die PortKlasse habe ich wesentlich mehr abstrakte Funktionen hinzugefügt und dieFunktionen um spezielle Register anzufordern entfernt. Dieser Schritt warnotwendig um eine abstraktere Ebene zu erreichen, des weiteren kann durchdie Convenience-Funktion "get<typename Register>()" falls nötig auf die Register zugegriffenwerden.

2.6. Woche 5

Nach Besprechung hatte ich Zwecks Kompatibilität und Portabilität einegetAddress-Funktion für die 0-Reihe implementiert, in diesem Zugemusste der Parser angepasst werden, sodass Strukturen erzeugtwerden können wie sie in der “alten” Library vorhanden sind.

Abbildung 13. Auszug aus ATmega4808SPI.hpp [ZLib]

```
struct registers {
    using ctrla = utils::Pair<reg::Register<reg::accessmode::RW,reg::specialization
::Control,CTRLAMasks>,0x0>;
    //...
    using intflags = utils::Pair<reg::Register<reg::accessmode::RW,reg::
specialization::Control,INTFLAGSMasks>,0x3>; ①
    //...
    intflags::type Intflags;
    data::type Data; ②
};
```

① Bisherige Register-Typdefinitionen

② Zusätzliche Variablendeklarationen

Als nächstes hatte ich mir eine eine Struktur einfallen lassen um dieKommunikation über die verschiedenen Schnittstellen zuvereinheitlichen. Die verschiedene Peripherie lässt sich über dieOberklasse Communication jeweils als Blockierend/ nichtBlockierend(mit oder ohne Interrupts[mit Protokoll Adapter oder FiFo]),mit oder ohne Fifo sowie als RW, R-only oder W-only konfigurieren.

Abbildung 14. Auszug aus Components.hpp [ZLib]

```
template<typename RW, typename accesstype, typename bit_width>
struct Communication {
    using Use_Fifo = typename accesstype::fifo;
    static constexpr bool fifoEnabled = Use_Fifo::value > 0;
    static constexpr bool InterruptEnabled = accesstype::intEnabled;
    static constexpr bool isBlocking = std::is_same_v<accesstype,blocking>;
    static constexpr bool isReadOnly = std::is_same_v<RW,ReadOnly>;
    static constexpr bool isWriteOnly = std::is_same_v<RW,WriteOnly>; ①

    using fifo_t = std::conditional_t<InterruptEnabled && fifoEnabled,
            volatile etl::FiFo< bit_width,Use_Fifo::value> ,
            std::conditional_t<! fifoEnabled,NoFifo,etl::FiFo< bit_width,Use_Fifo
::value>>>; ②
    static inline std::conditional_t<isReadOnly,NoFifo,fifo_t> fifoOut{};
    static inline std::conditional_t<isWriteOnly,NoFifo,fifo_t> fifoIn{}; ③
};
```

① Boolsche Konstanten für enable_if- und andere Funktionen

② Fifo Typbestimmung (volatile/nicht volatile/ keine Fifo)

③ In-/Output Fifos

Im CMake File wurden unnötige Einträge entfernt sowie ein Bug gefixt:Cmake hatte in der Release Version von sich aus mit O3 statt Os kompiliert, dashat dazu geführt dass alle Funktionen inlined wurden und die Hex-Datei immens vergrößert wurde. In der SPIImplementierung waren noch einige Fehler in den enable_if Konditionen,dies wurde während diversen Tests behoben. Danach habe ich dieUSART Schnittstelle implementiert, dabei habe ich noch einigeAnpassungen im Design der Library vorgenommen und Testsdurchgeführt. Während den Tests an der USART Schnittstelle fiel mirwegen der Baudrate auf dass die CPU-Frequenz falsch eingestellt istund ein Fehler bei der Kalkulation der getRegister Methode in Port, hierhatte ich bei der Offset Kalkulation fälschlicherweise auf die Registergröße statt auf 8-Bit gecasted (16-Bit Registeraddressen wurden falsch berechnet).

Abbildung 15. Auszug aus Port.hpp [ZLib]

```
template<typename T, auto &inst>
[[nodiscard, gnu::always_inline]] static inline auto &getRegister() {
    using reg_t = typename T::type;
    // auto offset = (typename reg_t::reg_size *) &inst() + T::value; ①
    auto offset = (uint8_t *) &inst() + T::value;
    return reg_t::getRegister(*((typename reg_t::reg_size*)offset));
}
```

① Alte Version

Die alte Version hat vor der Berechnung auf die Registergröße gecasted, allerdings sind die Offsets immer in Bytes angegeben, somit kamen bei Offsets > 0 falsche Adressen heraus.

Nachdem die USART Implementierung funktioniertebaute ich für das Board mit dem ich getestet habe (AVR Atmega 4808 IOTNano Curiosity) ein Debug Output ein, damit lassen sich sehr einfachAusgaben an den PC senden. Bei den USART Tests für denInterrupt-Mode fiel mir ein Fehler in meiner Scoped Implementierung auf,diese führte dazu das die Globalen Interrupts nicht wieder aktiviertwurden im Destruktor. Des Weiteren fiel mir auf dass die _delayFunktion der Microchip Library durch die Interrupts gestört wird, bei denTests war die Verzögerung viel zu hoch. Deswegen habe ich eigeneDelay-Funktionen eingebaut, unter anderem auch eine Safe DelayFunktion, während dem Delay sind dann Interrupts deaktiviert. Der eigentliche Grund für die Störung der Delay-Funktion war, dass der DRE Vector statt der TX-Vector benutzt werden muss wenn gesendet werden soll. Zum Schluss habe ich noch die Funktionen verschoben und etwasaufgeräumt damit man sich in der Header-Datei besser zurecht findet,außerdem habe ich die Struktur von USART auf SPI angewendet undgetestet ob alle Validen Konfigurationen Compilieren.

2.7. Woche 6

Zunächst habe ich das CMake-File so abgeändert dass bei einemmake-all Aufruf alle Atmega4808/09 Dateien generiert und in denpassenden Ordner gelegt werden.

Abbildung 16. Auszug aus CMakeLists.txt [ZLib]

```
add_custom_target(atmega4809 COMMAND untitled ${DEVICEFILE_4809} ${HWFOLDER_4809}
        WORKING_DIRECTORY ${PARSER_PATH} DEPENDS hwfilesexe)
add_custom_target(atmega4808 COMMAND untitled "${DEVICEFILE_4808}" "${HWFOLDER_4808}"
        WORKING_DIRECTORY ${PARSER_PATH} DEPENDS hwfilesexe) ①
add_custom_target(hwfiles DEPENDS atmega4808 atmega4809 ) ②
add_custom_target(hwfilesexe COMMAND make all
        WORKING_DIRECTORY ${PARSER_PATH}
        DEPENDS hwfilescmake
        )  ③
add_custom_target(hwfilescmake COMMAND cmake -G "MinGW Makefiles" ./
        WORKING_DIRECTORY ${PARSER_PATH}
        ) ④

add_dependencies(${PROJECT_NAME} hwfiles) ⑤
```

① Erstellung der HW-Dateien

② Festlegen welche HW-Dateien generiert werden

③ Erstellen des Parsers

④ Make-Datei für Parser erstellen

⑤ Festlegen das Parser erstellt wird

Der Parser hatte noch kleinere Bugsund unnötige Ausgaben die ich noch entfernt habe. Der Nächste Schrittwar die Implementierung von TWI, diese ist soweit fertig für dieBenutzung mit FiFos. Die TWI Schnittstelle habe ich so Designt dassdie Klasse einen Command Stack und Output/Input FiFos verwaltet. DerHintergrund hier ist die hohe Flexibilität und die Angenehme Benutzungder Schnittstelle. Des Weiteren kann die Schnittstelle so konfiguriertwerden, dass

viel Code eingespart wird z.B. wenn der Master nur liest(dann fallen diverse Prüfungen und/oder Datenmember weg). Beim testen fielen mir einigeSchwierigkeiten auf, ich hatte zunächst den Bus Status nicht abgefragtweil ich davon ausging dass die Abfrage der Wif Flag reicht (ohneAbfragen des Zustands hatte der Bus niemals eine Stop Conditiongesendet). Dieses Verhalten ist begründet durch den Smart Mode der neuen Microcontroller, dieser führt bestimmte TWI Funktionen aus wenn auf gewisse Register zugegriffen wurde. Nachdem der Bus korrekt gesendet hatte baute ich noch dieAuswahl der Frequenz ein. Anschließend hatte ich die SPI Schnittstellenoch mit Interrupts getestet und einen kleinen Bug behoben (FlagNamen waren falsch geschrieben und führten zu einem Compiler-Error).

2.8. Woche 7

Zuerst habe ich den TWI-Master fertig gestellt und damit einige Testsdurchgeführt (Funktion ohne Interrupts, mit Interrupts, scopedWritesowie mit und ohne FiFo). Als Nächstes hatte ich das CMake-File sowieeinige Header dahingehend angepasst, dass die Geräte-Auswahl nurnoch an einer Stelle im CMake-File passiert, somit wird das abändern inden Header-Dateien überflüssig. Als nächstes habe ich die Atmega4809Klasse auf den selben Stand wie die Atmega4808 gebracht. Alsnächstes war geplant die ADC-Schnittstelle zu implementieren, hier fielmir eine Schwachstelle meines Ressource-Controllers auf: DerRessource-Controller ging bisher davon aus, dass immer alleRessourcen einer Schnittstelle benutzt werden (beim ADC dann bis zu16 Pins). Wegen dieser Annahme musste ich ein Workaround für denRC hinzufügen um solche Komponenten anders zu überprüfen.

Abbildung 17. Auszug aus RessourceController.hpp [ZLib]

```
template<typename instances, typename Component_t, typename instances2 = void>
class RCComponent {
//...
    template<typename Alias, typename T = instances2>
    struct comps {
        using inst  = typename  instances::inst;
        using alt  = typename  inst::alt;

        static_assert(Meta::contains_all<typename instances2::template inst<Alias
::Instance>::template alt<Alias::Alternative>::list,typename alt::list>::value, "not
available pin was set up");
    }; ①

    template<typename Alias>
    struct comps<Alias,void>{
        using inst  = typename  instances::template inst<Alias::Instance>;
        using alt  = typename  inst::template alt<Alias::Alternative>;
    }; ②
    //lines omitted...
```

① neue Struktur für festgelegte Pins

② alte Struktur für die HS-Dateien

Das Workaround bestand hauptsächlich daraus die oben gezeigte Spezialisierung hinzufügen,

weiterhin ermöglichte diese Änderung die Einführung einer GenericRessource, diese kann verwendet werden um vom Benutzer festgelegte Pins zur Prüfung an den Ressource-Controller weiterzugeben. Bei der ADC Schnittstelle muss jetzt vom Benutzer angegeben werden welchePins benutzt werden, der RC überprüft dann nur diese. Später wird inder ADC Schnittstelle beim Auswahl des Kanals wieder überprüft ob derPin auch überprüft wurde (diese Funktion kann auch ausgeschaltetwerden, falsche Pins würden dann einen in der Klasse definiertenDefault-Wert für den ADCMux setzen), somit kann der Benutzer keinefalschen Pins übergeben. Zum Ende der Woche hatte ich ein Gerüst fürdie ADC-Klasse eingebaut und die Funktionalität desRessource-Controllers im Hinblick auf die Änderungen validiert.

2.9. Woche 8

Zunächst hatte ich für TWI bezüglich des Lesens ein Callback eingebaut,somit kann der Benutzer außerhalb der Schnittstelle nach Empfang derDaten festlegen was passiert.

Abbildung 18. Auszug aus TWI.hpp [ZLib]

```
template<>
struct Command<true,true> {
    uint8_t address = 0;
    uint8_t bytes = 0;
    void (*Callback)() = (void (*)()) noop; ①

    void operator=(const volatile Command &other) volatile {
        address = other.address;
        bytes = other.bytes;
        Callback = other.Callback;
    }
};
```

① Default Value ist eine Funktion die nichts tut

Zunächst hatte ich std::variant in Betrachtgezogen, da aber zur Laufzeit auch entschieden werden können muss,ob jetzt gerade gelesen oder geschrieben werden soll, schied diese Variante aus. Das Callback feature wird in der Command Strukturals function-pointer gespeichert und zur Laufzeit aufgerufen, der Nutzerkann jedoch auch eine Lambda-Referenz übergeben. Als nächstes begann ich ein Testsystem für das Lesen überTWI aufzubauen, hierzu verwende ich den Atmega4808, einenLogic-Analyzer und meinen STM32F446RE.

Abbildung 19. Auszug aus main.c

```
int main(void) {
  HAL_Init();

  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_I2C1_Init();
  MX_SPI1_Init();
  MX_UART4_Init();

  unsigned char* testData = (unsigned char*)"Hallo Master";
  unsigned char arr[12];
  while (1)
  {
      //HAL_I2C_Slave_Transmit(&hi2c1, testData,12, 500);
      HAL_TWI_Transmit(&htwi1, testData,12,500);
      //HAL_SPI_Transmit(&hspi1, testData, 12, 50);
      //HAL_SPI_Receive(&hspi1, arr, 12, 50); ①
      HAL_Delay(120);
  }
}
```

① HAL Funktionen der STM-Library [STM]

Der STM schickt in einer Quasi-Dauerschleife immer wieder einen String, der Atmega4808 mussden Bus korrekt Steuern und die Daten speichern sowie per USARTausgeben (kann dann einfach über Putty auf dem Terminal ausgegebenwerden). Der Logic-Analyzer hängt zusätzlich noch zwischendrin um dieBaudrate sowie Start-/Stop-Conditionen und etwaige Testausgabenaufzunehmen.

Im Zuge der Tests (welche später auch als Beispiel-Code verwendet werden) sind einige Bugs aufgefallen:

- Quick-Command war per Default angeschaltet, dieser hatteProbleme bereitet da er scheinbar nicht I²C kompatibel ist.
- Das Blockierende Lesen war noch nicht fertig eingebaut.
- Das Transactionsende beim Lesen von Master muss explizit mitdem Kommando Stop und dem setzen des NACK Bits im MasterControlB Register beendet werden (war nicht sehr ersichtlich ausder Doku).

2.10. Woche 9

Die Kernaufgabe dieser Woche war es den Userguide zu erstellen, indiesem Zuge wurden noch einige Bugs behoben und etwas am Designverändert. .Folgende Bugs habe ich entdeckt und behoben:* Die Current-Commando Variable muss als volatile deklariert werden.* Bei SPI wurde nicht beachtet dass für ein lesen als Master auchwenigstens eine 0 in das Datenregister geschrieben werden muss.* Bei SPI wurde der SS Pin nicht belegt, dies führt vor allem bei

derSlave Variante zu Problemen.* In PortPin wurden intern noch die C-Strukturen statt die Registerverwendet.Bei der Erstellung der Doku wurden einige Probleme sichtbar die sichauf die Benutzung (das Design) der Library beziehen:Bei TWI fehlte bei den ScopedTransactions für das Lesen einreceiveLast (um die Transaktion korrekt abzuschließen).Eine `Reset` Methode fehlte um bei einer Fehlerbehandlung o.Ä. denZustand der TWI-Schnittstelle zurückzusetzen.Beim Eventsystem werden ab jetzt User **und** Generator aus demChannel bezogen weil es für den Benutzer übersichtlicher ist. DieRegisterListener Methode des Eventsystems kann jetzt beliebig vieleNutzer registrieren. Des weiteren habe ich eine Funktion eingebaut die es ermöglicht die Software-Events für den Channel auszulösen. Für TWI kann jetzt ein NACK-Handler übergeben werden damit diese Fehlersituation korrekt abgefangen werden kann.

Abbildung 20. Auszug aus main.cpp

```
using _twi = AVR::twi::TWIMaster<AVR::blocking, twires, AVR::ReadWrite>; ①

static inline auto handler = [](){
    while (!_twi::endTransaction());
    while(!_twi::startTransaction<42,AVR::twi::access::Read>());
}; ②

using twi = AVR::twi::TWIMaster<AVR::blocking, twires, AVR::ReadWrite, handler>; ③
```

① Deklarieren der Schnittstelle ohne handler

② Definieren des handlers

③ Einsetzen des Handlers in die Schnittstelle

Bei dem Atmega4808 Dev-Board ist mir ein Designfehler aufgefallen der es verhindert das der SPI Anschluss im Slave-Modus betrieben werden kann und des Weiteren zu Verwirrung führt. Microchip hat auf dem Board neben dem MISO,MOSI und SCK der 0. Portmux Variante den SS der 1. Variante gesetzt, das erweckt den Eindruck der SS-Pin gehöre zu den anderen SPI Pins, aber das Design führt dazu, dass der Pin dort Wirkungslos ist, weil auf den UCs keine einzelnen Pins gerouted werden können.Nach Erkenntnis dieses Design-Fehlers bereitete ich noch eine entsprechende Fehlermeldung für Microchip vor und stellte diese zu.Aus Portabilitäts- sowie Ästheticgründen habe ich zum Schluss die Doku mit Asciidoctor erstellt.

2.11. Woche 10

In der letzten Woche der Praxisphase habe ich mich eindringend mit dem fertigstellen des User-Guide sowie der Erstellung eines kleinen Development-Guides beschäftigt. Hintergrund hierzu ist, dass später auch andere Entwickler die Funktionsweise zwecks Weiterentwicklung und Benutzung nachvollziehen können. Kern des Development-Guides war hier vor allem der Ressource-Controller und der Aufbau der HW-Dateien sowie die Interaktion dieser mit den µC-Klassen sowie des Ressource-Controllers.

Kapitel 3. Design und Aufbau der Library

Es werden hier einige spezielle Aspekte der Library erwähnt, der Großteil wie z.B. Register oder Port Strukturen setze ich als gegeben vorraus (s.a. [BMCPP]).

3.1. HW Files

Die Software Basiert Komplett auf den im folgenden erläuterten automatisch generierten Dateien. Aus der HW-Beschreibung von Microchip (*.atdf) wird mit einem mitgelieferten Parser eine C++ Struktur erzeugt, diese Struktur wird später in der Library verwendet. Das vorliegende Design ist begründet durch die hohe Flexibilität hinsichtlich Änderungen durch Microchip (durch die Defines der Microchip Library werden bei jedem Kompilieren die Änderungen mit übernommen) sowie die hervorragenden generischen Eigenschaften durch die automatische Erstellungen der Registerstrukturen. Anders als in BMCPP müssen von Hand keine Register in Header-Dateien geschrieben werden, das wird von dem Parser übernommen und spart somit Zeit und verhindert Flüchtigkeitsfehler.

Die EventSystem Dateien liegen zwar im selben Ordner wie die anderen HW-Dateien, jedoch sind diese wegen ihrer speziellen Struktur von Hand erstellt

3.1.1. Der Header

Jede HW-Datei beginnt nach dem obigen Schema, wegen der Namespaces sind die einzelnen Geräte und Komponenten eindeutig voneinander getrennt. Der erste include bezieht sich auf die Port-Abstraktion, diese wird innerhalb der HW-Dateien genutzt um übersichtliche Strukturen erzeugen zu können. Danach werden die Meta-Funktionen[BMCPP] inkludiert, hier wird die List genutzt um die Pins einer Komponente zu einer Liste zu packen, diese kann der Ressource-Controller verwenden um auf Kollisionen zu prüfen.

Abbildung 21. Auszug aus ATmega4808SPI.hpp [ZLib]

```
#pragma once
 #include "../../hw_abstractions/Port.hpp"
 #include "../../tools/meta.h" ①

namespace mega4808 { ②
    namespace spi_details { ③
```

① Notwendige includes

② Device-Namespace → jeweils `mega` + die Nummer

③ Komponenten-Namespace → jeweils die Komponente und `_details`

3.1.2. Die Komponente

Die Komponenten Struktur besteht im Grunde aus 2 wesentlichen Teilen:

- Einige Enum-Klassen die Später den Registern als typ dient um Funktionen bereit zustellen,

somit ist das übergeben von unbenannten Werten nur erschwert möglich.

- Die `registers` Struktur welche die Registertyp-Deklarationen und Register-Variablen Deklarationen enthält.

Die `registers` Struktur enthält alle definierten Register der Komponente, diese sind aus der atdf-Datei abgeleitet und somit sind auch die Attribute des Typs gewählt (Schreibschutz, Größe ....). Die Variablen Deklarationen sind in der Register Struktur vorhanden um die Legacy Funktionen "getBaseAddress" aus der BareMetal-CPP Library zu ermöglichen, in der eigentlichen Library finden sie keine Anwendung. Die Registertypen sind als Pair angegeben und habe noch den Offset-Wert mit angegeben für eine spezielle Funktion innerhalb der Port Abstraktion.

Abbildung 22. Auszug aus ATmega4808SPI.hpp [ZLib]

```
struct spiComponent {
    //lines omitted....
    enum class INTFLAGSMasks : mem_width {
        Bufovf = SPI_BUFOVF_bm,
        //...
        Wrcol = SPI_WRCOL_bm
    }; ①

    struct registers {
        using ctrla = utils::Pair<reg::Register<reg::accessmode::RW,reg
::specialization::Control,CTRLAMasks>,0x0>;
        using ctrlb = utils::Pair<reg::Register<reg::accessmode::RW,reg
::specialization::Control,CTRLBMasks>,0x1>; ②
        //...
        intflags::type Intflags;
        data::type Data; ③
    };
};
```

① Spezial-Bits für das Register `intflags`

② Registertypen und Namen

③ Register Variablen Deklarationen

Das Attribut `packed` muss nicht für die einzelnen Strukturen gesetzt werden, dem Compiler wird das Flag `fpack-struct` übergeben.

3.1.3. Die Instanzen

Der letzte Teil der HW Dateien besteht aus der `spis` Struktur, diese ist eine Sammlung an Ressourcen-Instanzen der Komponente (z.B. USART0, USART1 ...). Die `value` Funktion wird in der Library an verschiedenen Stellen verwendet, z.B. in der Port Klasse um dann auf die verschiedenen Register des Ports zuzugreifen. Hier werden zunächst die Instanzen aufgelistet, auf die verschiedenen Instanzen kann dann mit der entsprechenden Nummer zugegriffen werden (z.B. USART0 → usarts::template inst<0>). Innerhalb der verschiedenen Instanzen befinden sich dann noch die Mappings für die unterschiedlichen Portmux alternativen, somit wird es einem Benutzer

auch ermöglicht verschiedene Alternativen auszuwählen. Erst innerhalb der Portmux Alternativen finden sich dann die unterschiedlichen Pin-Gruppen und die konkreten Pins. Wenn also auf die Konkreten Pins zugegriffen werden soll geschieht dies mit der Angabe einer konkreten Instanz sowie der Portmux Alternative.

Abbildung 23. Auszug aus ATmega4808SPI.hpp [ZLib]

```
//lines omitted...

struct spis {
    //...
    template<bool dummy>
    struct inst<0, dummy> ①
     {
        //...
        [[nodiscard,gnu::always_inline]] static inline auto& value()  { return SPI0;}
②

        template<bool dummy1>
        struct alt<1, dummy1> ③
         {
            struct Miso {
                using pin0 = AVR::port::details::PortPin<port_details::port
<port_details::ports::portc>,1>;
            };
            //...
            struct Ss {
                using pin0 = AVR::port::details::PortPin<port_details::port
<port_details::ports::portc>,3>;
            }; ④

            using list = Meta::List<typename Miso::pin0, typename Mosi::pin0, typename
Sck::pin0, typename Ss::pin0>; ⑤
        };

        template<bool dummy1>
        struct alt<0, dummy1>
         {
            //lines omitted...
        };

    };

};
```

① Konkrete Instanz einer Ressource

② Funktion welche die von Microchip definierte Struktur zurückgibt

③ Die Portmux-Variante

④ die verschiedenen Gruppen innerhalb einer Portmux-Variante

⑤ Liste von allen Pins die von der Ressource belegt werden können

Die hier beschriebenen Struktur ist nur relevant wenn die Komponente I/O Pins verwendet

3.2. Die Microcontrollerklassen

In den Microcontrollerklassen werden die HW-files inkludiert, des Weiteren stellen die Klassen die Komponenten und deren Konfigurationen zur Verfügung (für die Top-Level Funktionen im AVR-Namespace). Die Microcontrollerklassen stellen die Konfigurationen für die Library zur Verfügung damit innerhalb der einzelnen Klassen schon vorab spezifische Einstellungen angepasst werden können. Somit können innerhalb dieser Familie Spezialfälle in den Microcontrollerklassen behandelt werden. Die Microcontrollerklassen können hinzugefügt werden indem eine fertige kopiert und in eine neue Header-Datei hinzugefügt wird. Anschließend müssen nur die entsprechenden namespaces, Port-Mappings und includes angepasst werden. Allerdings muss dann auch die entsprechende .atdf Datei hinzugefügt werden und im CMake-File muss die Datei-Generierung ergänzt werden.

3.2.1. Header

Zu Beginn werden in den Microcontrollerklassen die benötigten Include-Dateien und Typdefinitionen vorgenommen. Dies ist zunächst der Typ `mem_width`, dieser bezeichnet die Speicherbreite (z.B. bei den 8-Bit µCs sind das immer 8 Bit), die zweite Typdefinition ist ptr_t und bezeichnet Zeigertyp. Der erste Hardware-Include muss der Port sein, der Port besitzt innerhalb der Library eine Sonderfunktion, da dieser verwendet wird um andere Funktionen zu ermöglichen. Daraus folgt dann auch dass der Port eine Vorraussetzung für die anderen Library Funktionen ist. Nach dem Port können die verschiedenen HW-Dateien in beliebiger Reihenfolge inkludiert werden, da diese immer nur in der entsprechenden Abstraktion benötigt werden. .Danach kommen die Feststehenden Abstraktionen: * Die CPU: Ermöglicht das auslesen des Statusregister, diese Funktion wird innerhalb der Scoped-Klasse benutzt. * Der Resssource-Controller: die dort vorhandene RCComponent-Klasse wird verwendet um qualifizierte Klassen zu identifizieren. * Die AtmegaZero Datei: Hier werden die Enum-Klassen aus den HW-Dateien nochmals in Atomare Einstellungen unterteilt damit die Schnittstelle besser konfiguriert werden kann.

Abbildung 24. Auszug aus Atmega4808.hpp [ZLib]

```
#pragma once
using mem_width = uint8_t;
using ptr_t = uintptr_t; ①
#include "../hw_abstractions/Basics.hpp"

//hw includes
#include "hal/ATmega4808PORT.hpp"
#include "mega4808/hal/Atmega4808EventSystem.hpp"
#include "hal/ATmega4808SPI.hpp"
//...
#include "hal/ATmega4808ADC.hpp" ②
#include "../hw_abstractions/CPU.hpp"
#include "../hw_abstractions/RessourceController.hpp"
#include "../DeviceFamilys/ATmegaZero.hpp" ③
```

① In folgenden includes benötigte Typdefinitionen

② Hardware-Includes

③ Von der Library benötigte includes

3.2.2. Klasse

Innerhalb der Komponenten der Microcontrollerklassen befinden sich private Elemente, der RessourceController und die Metafunktion benötigen Zugriff auf diese. Wie zuvor schon erwähnt wird die Verwendung des Ressource-Controller mehr oder weniger erzwungen, dies geschieht mit diesen private-Deklarationen, im Prinzip kann sich nur der Ressource-Controller auf die benötigten Elemente Zugriff verschaffen. Umgehen kann man den Ressource-Controller entweder mit den ungeprüften Ressourcen (siehe RessourceController) oder durch die Verwendung der HW-Klassen in den entsprechenden namespaces.

Abbildung 25. Auszug aus Atmega4808.hpp [ZLib]

```
template<auto frequency>
class Atmega4808 {

    template<typename Alias>
    friend struct AVR::rc::details::resolveComponent; ①
//...
    public:
    struct isZero{}; ②
    static constexpr auto clockFrequency = frequency;
    template<typename T>
    static inline constexpr bool is_atomic(){
        return false;
    } ③
```

① Friend-Deklaration für eine Metafunktion zur Bestimmung der Instanzen von Komponenten

② Typdefinition für ein Concept

③ Legacy-Funktion aus BMCPP

Damit bestimmte Tools aus BMCPP verwendet werden können müssen dort Spezialisierungen erstellt werden, dies wird ermöglicht durch die Concepts. ein Concept überprüft vor der Spezialisierung ob die `isZero` Struktur vorhanden ist, dies hat dann den Vorteil dass innerhalb der älteren Microcontrollerklassen nichts verändert werden muss. Die is_atomic Funktion musste aus kompatibilitätsgründen übernommen werden, diese Funktion wurde z.B. auch in der Scoped-Klasse aufgerufen.

3.2.3. Spezialfall Port

Innerhalb der Microcontrollerklasse ist der Port als Komponente ein Ausnahmefall, weil er nur dazu dient den Portnamen zuzuordnen (und ggf. die legacy BMCPP-Funktion ermöglicht). Der Port als Komponente wird in der Port-Abstraktion genutzt um die Port-Strukturen der Library zu erstellen hierzu wird das gezeigte Mapping angewandt, die daraus entstehenden Strukturen werden in der gesamten Library und vom Benutzer verwendet.

Abbildung 26. Auszug aus Atmega4808.hpp [ZLib]

```
template<typename p> ①
struct Port {

    using port = typename utils::condEqual<AVR::port::A, p, port_details::port
<port_details::ports::porta>,
            typename utils::condEqual<AVR::port::C, p, port_details::port<
port_details::ports::portc>,
                    typename utils::condEqual<AVR::port::D, p, port_details::port
<port_details::ports::portd>,
                            typename utils::condEqual<AVR::port::F, p, port_details
::port<port_details::ports::portf>, void
                            >::type
                    >::type
            >::type
    >::type; ②

    static constexpr auto baseAddress = port::port; ③

    private:
    using Component_t = port_details::portComponent; ④
};
```

① Übergabe des Portnamen

② Übersetzung des Portnamen in den HW-Port

③ Hilfsvariable für die getBaseAddress Funktion

④ Typdeklaration der HW-Komponente

3.2.4. Komponenten

Diese Struktur zeigt den Aufbau einer Microcontrollerklasse mit der Eigenschaft, dass die Komponente vom RessourceController verwaltet wird. Die Setting Struktur ermöglich eine einfache und flexible Konfiguration der Schnittstelle, aus dieser werden später dann in den entsprechenden `init` Methoden die Register eingestellt.

Abbildung 27. Auszug aus Atmega4808.hpp [ZLib]

```
struct USART : public AVR::rc::details::RCComponent<usart_details::usarts,
usart_details::usartComponent>, ①
               public AVR::details::AtmegaZero::template USART_C< usart_details
::usartComponent> ②
               {

    template<USART::RS485Mode RSMode, USART::ReceiverMode receiverMode,
            //...
            >
    struct USARTSetting {
        using AConf = usart_details::usartComponent::CTRLAMasks;
        using BConf = usart_details::usartComponent::CTRLBMasks;
        using CConf = usart_details::usartComponent::CTRLCMasks;

        static constexpr AConf rsmode = static_cast<AConf>(RSMode);
        static constexpr AConf loopbackmode = LoopBackMode ? AConf::Lbme :
static_cast<AConf>(0);
        static constexpr BConf opendrainmode = OpenDrainMode ? BConf::Odme :
static_cast<BConf>(0);
        //...
        static constexpr CConf msb = !Msb ? CConf::Udord : static_cast<CConf>(0);
    }; ③
private:
    using Component_t = usart_details::usartComponent;
};
```

① Legt fest dass die Komponente vom RessourceController gesteuert wird

② Importiert die Enum-Klassen der 0 Familie aus der entsprechenden Komponente

③ Konfigurationen der Schnittstelle

Das Erben von RCComponent ermöglicht es dem Ressource-Controller die verschiedenen Instanzen sowie die Register zu sehen (hier z.B. usarts und usartComponent). Der Ressource Controller braucht Zugriff auf die verschiedenen Instanzen des bestimmten µCs um Zugriff auf die Pin-Liste zu bekommen. Die zweite Oberklasse ermöglicht dann die Konfiguration über die Enum Klassen der 0 Serie, somit werden gut lesbare und Atomare Einstellungen für den Benutzer zur Verfügung gestellt. Der Vorteil der Oberklasse besteht hier darin, dass die Enums für die ganze Familie zur Verfügung stehen und nicht nur für einen µC (somit muss der Vorgang nicht für jeden µC einzeln vorgenommen werden).

3.3. AtmegaZero Struktur

Diese Struktur dient dazu die verschiedenen Einstellungen aus den HW-Dateien aufzugliedern und in sinnvolle Gruppen zu packen, sodass in den Setting Strukturen vom Benutzer klare Einstellungen möglich sind. In C-Style müssten die Benutzer den entsprechenden Registern per Makros und Bit-Operationen dann die entsprechenden Werte zuweisen, diese Vorgehensweise ist unübersichtlich, fehleranfällig und wird hier ganz klar vermieden.

Abbildung 28. Auszug aus ATmegaZero.hpp [ZLib]

```
namespace AVR {
    namespace details {
        struct AtmegaZero {
            //...
template<typename twiComponent>
struct TWI_C {
    enum class SDASetup: mem_width {
        SDASetup_off = static_cast<mem_width>(twiComponent::CTRLAMasks::Sdahold_off),
        SDASetup_50ns = static_cast<mem_width>(twiComponent::CTRLAMasks::Sdahold_50ns)
,
        SDASetup_300ns = static_cast<mem_width>(twiComponent::CTRLAMasks
::Sdahold_300ns) ,
        SDASetup_500ns = static_cast<mem_width>(twiComponent::CTRLAMasks
::Sdahold_500ns) ,
    };

    enum class SDAHold: mem_width {
    //...
};
```

3.4. RessourceController

Der Ressourcecontroller hat die Aufgabe die Ressourcen und ihre Pinbelegungen miteinander zu Vergleichen, hierzu verwendet der Ressourcecontroller die in den HW-files enthaltenen Pin-lists, der Parser fügt diese Listen automatisch hinzu, dies hat also den Vorteil dass ohne großen Aufwand während der Compile-Zeit schon Kollisionen ausgeschlossen werden können. Zusätzlich ist der Ressource-Controller so konzipiert, dass bei normaler Verwendung der Library das Verwenden des Ressource-Controller notwendig ist. Die Verwendung des Ressource-Controller kann auch Umgangen werden, diese Funktionalität wurde eingebaut damit man als erfahrener Benutzer sämtliche Möglichkeiten ausschöpfen kann.

3.4.1. RCComponent

Diese Struktur wird im RessourceController genutzt um z.B. auf die HW-Instanzen zuzugreifen um schließlich die IO Pins zu vergleichen. Je nach Komponente ist e möglich dass nicht alle Pins überprüft werden die in den Komponenten verfügbar sind, z.B. der ADC.

Abbildung 29. Auszug aus RessourceController.hpp [ZLib]

```
template<typename instances, typename Component_t, typename instances2 = void>
class RCComponent {

    template<typename Alias>
    friend struct AVR::rc::details::resolveComponent;

    template<typename Alias, typename T = instances2>
    struct comps {
        using inst  = typename  instances::inst;
        using alt  = typename  inst::alt;

        static_assert(Meta::contains_all<typename instances2::template inst<Alias
::Instance>::template alt<Alias::Alternative>::list,typename alt::list>::value, "not
available pin was set up");
    }; ①

    template<typename Alias>
    struct comps<Alias,void>{
        using inst  = typename  instances::template inst<Alias::Instance>;
        using alt  = typename  inst::template alt<Alias::Alternative>;
    }; ②
public:
    static constexpr bool isRCComponent = true;

    template<auto number>
    static inline auto getBaseAddress(){
        return (typename Component_t::registers*) &instances::template inst<number
>::value();
    } ③

};
```

① Spezialisierung für eine Teilprüfung der IO Pins

② Spezialisierung für eine Vollprüfung der IO Pins

③ Realisierung der BCMPP legacy-Funktion getBaseAddress

Hier wird die Comps-Struktur verwendet um zwischen den verschiedenen Komponenten (alle Pins überprüfen oder nicht) zu unterscheiden, wenn der Typ `void` entspricht so werden die Pins aus den HW-Dateien verwendet. Im zweiten Fall werden nicht die Pins aus den HW-Dateien, sondern die vom Benutzer übergebenen Pins verwendet.

3.4.2. Die Kernklasse

Hier wird als erstes die MCU übergeben (idr. DEFAULT_MCU), danach folgen eine bis beliebig viele Ressourcen (0 Ressourcen macht keinen Sinn). Die beiden Alias Metafunktionen sollen jeweils die erste oder 2. Ebene in der HW-Struktur einer Instanz zurückgeben und eine kürzere und übersichtlichere Schreibweise innerhalb des RC zur Folge haben.

Abbildung 30. Auszug aus RessourceController.hpp [ZLib]

```
template<typename MCU,typename FIRST,typename... INSTANCES>
class ResController {

    template<typename Alias>
    using resolveAlt = typename details::resolveComponent<Alias>::alt;
    template<typename Alias>
    using resolveInst = typename details::resolveComponent<Alias>::inst;
```

Hier zu sehen sind die einzelnen Prüfungen die der RessourceController durchführt während er die angeforderte Ressource frei gibt.

Abbildung 31. Auszug aus RessourceController.hpp [ZLib]

```
template<typename N>
struct getRessource {
    using type = utils::tuple<typename get_ressource_help<N, FIRST, INSTANCES...
>::inst,typename get_ressource_help<N, FIRST, INSTANCES...>::alt>;
        static_assert(!std::is_same<typename type::t2, void>::value, "portmux not
found"); ①
        static_assert(checkRessource<FIRST, INSTANCES...>(), "I/O Pins conflicting");
②
        static_assert(checkInstance<FIRST, INSTANCES...>(), ③
                      "only 1 alternative from a single instance permitted");
};

template<typename N>
using getRessource = typename getRessource<N>::type;
```

① Prüfung ob die Portmux-Variante existiert

② Prüfung ob die I/O Pins Kollisionsfrei sind

③ Prüfung auf doppelt Belegung einer Instanz durch mehrere Portmux-Varianten

Diese Prüfungen haben zu Folge dass der Benutzer keine doppelt belegten Pins, keine 2 verschiedenen Portmux-Alternativen aus derselben Instanz und keine nicht existierenden Ressourcen belegen kann. Die erste Prüfung wird einen Fehler produzieren wenn der Rückgabetyp der `get_ressource_help` Funktion `void` ist, d.h. die Funktion hat keine Ressource `N` gefunden. ===== Zu Punkt 2

Dies ist die Startfunktion um zu Prüfen ob die Pins Kollisionsfrei sind, hierzu wird die Liste der Pins innerhalb der konkreten Instanzen benötigt. Falls `pins` 0 Elemente enthält gibt es nur eine Ressource und somit keine Kollision, ansonsten muss die rekursive Hilfsfunktion aufgerufen werden.

Abbildung 32. Auszug aus RessourceController.hpp [ZLib]

```
template<typename _first,typename... pins>
static constexpr bool checkRessource() {
    using first = typename resolveAlt<_first>::list;

    if constexpr(sizeof...(pins) == 0)
    return true;
    else return checkRessourceHelper<first,pins...>();
}
```

Die Prüfung auf Kollisionsfreiheit Funktioniert so, dass auf der aktuellen Stufe immer überprüft wird ob irgendein Element der aktuellen Pinliste in der nächsten enthalten ist, danach wird die aktuelle Liste mit der nächsten vereinigt und als erste Liste übergeben, das geht solange bis es in `pins` keine weiteren Listen mehr gibt.

Abbildung 33. Auszug aus RessourceController.hpp [ZLib]

```
template<Meta::concepts::List first,typename _second, typename... pins>
static constexpr bool checkRessourceHelper(){

    using second = typename resolveAlt<_second>::list;

    if constexpr(sizeof...(pins) > 0){
        constexpr bool current = !Meta::contains_any<first, second>::value; ①
        constexpr bool next =  checkRessourceHelper<Meta::concat_t<first,second>,
pins...>(); ②
        return  current && next ;
    }
    else {
        return !Meta::contains_any<first,second>::value; ③
    }
}
```

① Aktuelle Rekursionsstufe

② Nächste Rekursionsstufe

③ Rekursionsende

Die CheckInstance Methode funktioniert Analog (bloß das die Pinlisten nicht überprüft werden sondern die HW-Instanzen)

3.5. TWI Command und Callback Aufbau

Die TWI Schnittstelle ist grundsätzlich so Aufgebaut dass sie (wenn mit Fifos benutzt) genauso verwendet werden kann wie die anderen Kommunikationsschnittstellen, die einzige Besonderheit ist das Callback beim lesen. Damit diese einfache Bedienung der Schnittstelle möglich ist, müssen Intern die Command Strukturen gespeichert werden.

Abbildung 34. Auszug aus TWI.hpp [ZLib]

```
static inline auto noop = []{}; ①

template<bool singlemode = false, bool callback = false>
struct Command {
    uint8_t address = 0; ②
    uint8_t bytes = 0; ③
    readWrite access = write; ④
    void (*Callback)() = (void (*)()) noop; ⑤

    void operator=(const volatile Command &other) volatile {
        address = other.address;
        bytes = other.bytes;
        access = other.access;
        Callback = other.Callback;
    }
};
```

① Leerer Funktionsaufruf (wenn geschrieben wird)

② Adresse des anderen Device

③ Anzahl zu lesender/schreibender Bytes

④ Lesen oder Schreiben

⑤ Callback für lesen

Die Command Struktur wird benötigt um gewisse Informationen zur Laufzeit zu speichern und auszuwerten. Diese Struktur wiederum wird in einer Fifo gespeichert, wenn z.B. Bytes gelesen werden sollen so wird eine entsprechende Command Struktur erzeugt und in der Fifo abgelegt. Eine statische Callback Funktion wäre hier nicht sinnvoll, da auch zur Laufzeit entschieden wird ob das aktuelle Kommando ein Lese-oder Schreibvorgang ist.

Die oben gezeigte Command Struktur besitzt noch einige Spezialisierungen um bei verschiedenen Konditionen nicht benötigte Member auszulassen (z.B bei einem WriteOnly wird weder das Callback- noch das Accessmember benötigt).

Abbildung 35. Auszug aus TWI.hpp [ZLib]

```
static inline void put(bit_width *data, uint8_t size) {

    if constexpr(_TWI::isWriteOnly)
        CommandStack.push_back(command{(address << 1), size}); ①
    else
        CommandStack.push_back(command{(address << 1), size, write}); ②
    for (uint8_t i = 0; i < size; i++)
        _TWI::fifoOut.push_back(data[i]);
}
```

① Command Konstruktor für WriteOnly

② Command Konstruktor für ReadWrite

In diesem Beispiel wird gezeigt wie die Command Struktur genutzt wird um Befehle in eine Fifo zu legen, während dem Periodic Aufruf oder der Interrupt Routine können die Befehle dann aus der Fifo entnommen und verarbeitet werden. Das Kommando wird hier erzeugt mit dem Konstruktor: uint8_t Adresse und uint8_t Größe, wenn die Schnittstelle als ReadWrite konfiguriert ist, muss noch die Information hinzugefügt werden, dass es ein Schreibvorgang ist. Beim Lesen verhält es sich Analog, jedoch mit dem Zusatz, dass als letztes Member noch ein Callback in die Command-Struktur gelegt wird.

Abbildung 36. Auszug aus TWI.hpp [ZLib]

```
//RW Branch
bool newCommand = false; ①

if(TWIMaster::current.access == read){ ②
    if ( TWIMaster::current.bytes > 1) {
        TWIMaster::_TWI::receive();
    } else if( TWIMaster::current.bytes == 1) {
        stopTransactionNack();
        TWIMaster::_TWI::receive();
        TWIMaster::current.bytes = 0;
        TWIMaster::current.Callback();
        typename TWIMaster::command tmp;
        if (TWIMaster::CommandStack.pop_front(tmp)) {
            TWIMaster::current = tmp;
            newCommand = true;
        }
    }
}
```

① Boolsche Variable zur Feststellung ob ein neues Kommando folgt

② Prüfung auf Zugriffsart

Zunächst wird im IH unterschieden ob es sich beim aktuellen Element um ein Lese- oder Schreibvorgang handelt, je nachdem wird dann die receive() oder die transfer() Methode aufgerufen, diese Methoden führen dann noch Aktionen auf dem `current` Member der Oberklasse aus (dekrementieren der bytes Variable). Im Lesevorgang muss speziell geprüft werden ob noch ein Element zu versenden ist, wenn ja muss vor der receive() Methode ein Nack mit einer Stop-Condition gesendet und abschließend der Callback ausgeführt werden, alle anderen Vorgehensweisen führen zu Buskollisionen oder anderen ungewünschten Verhaltensweisen. Der nächste Schritt ist das herausnehmen des neuen Kommandos aus der Fifo, nebenbei wird die Variable newCommand auf `true` gesetzt, diese wird anschließend genutzt um zu prüfen ob noch auf die Stop-Condition gewartet werden muss.

Abbildung 37. Auszug aus TWI.hpp [ZLib]

```
if(newCommand){
    while(! (alt::Scl::pin0::isOn() && alt::Sda::pin0::isOn())); ①
    if(TWIMaster::current.access == read){
        readCondition();
    } else {
        writeCondition();
    } ②
}
```

① Warten auf die Beendigung der Stop-Condition

② Auslösen der neuen Read oder Write Start-Condition

Nachdem das neue Kommando übernommen wurde ist es notwendig auf das Beenden der Stop-Condition zu warten, direkt danach kann dann die neue Start-Condition gesendet werden, ansonsten wird der IH nicht wieder aufgerufen und das nächste Element nicht gesendet.

Kapitel 4. Benutzung der Library

4.1. Programmierung

Um die Programme auf die Chips der Atmega 0 Serie zu flashen gibt es grundsätzlich 2 Methoden:

- https://github.com/ElTangas/jtag2updi hier wird ein Atmega328 oder ein vergleichbarer Chip zu einem Programmer umfunktioniert, hierzu kann z.B. ein fertiges Arduino-Board genutzt werden.
- https://github.com/mraardvark/pyupdi hier wird mithilfe eines USB-UART Adapters geflashed, hier muss darauf geachtet werden dass der Widerstand stimmt, sollte der Widerstand nicht funktionieren muss eine Diode mit der Anode zu UPDI und Kathode zu Tx verbunden werden.

4.2. Grundsätzliches

- Die Library stützt sich auf den AVR-GCC 8.3+, es werden intern unter anderem Concepts genutzt, die Verwendung eines aktuellen cpp Compilers ist daher notwendig.
- Am besten wird das enthaltene CMakefile genutzt um das Projekt zu erstellen, um den UC zu wechseln muss lediglich die Variable `Mnumber` und `F_CPU` abgeändert werden.
- [Optional] um ein schnelleres Compilieren zu ermöglichen kann nach dem ersten Compilieren die Variable GenerateHWFiles auf FALSE gestellt werden.
- Die Library kann auf Windows und Linux genutzt werden, sollte auf dem Ziel-Windows "sh.exe" in der Shell vorhanden sein wird sich CMake beschweren und beim ersten Versuch abbrechen, zur Behebung muss dann ein zweites Mal CMake aufgerufen werden.
- Es sollten nur die HW-Abstraktionen inkludiert werden, ansonsten kann kein ordnungsgemäßes funktionieren gewährleistet werden.
- Der Erste Include MUSS MCUSelect.hpp sein, hier findet die Auswahl des Gerätes statt.
- Port, PortPin und der Ressource Controller sind Standardmäßig inkludiert und müssen nicht nochmal explizit inkludiert werden.
- Es ist notwendig, dass die von Microchip bereitgestellten Header, Library-files sowie device-specs der Toolchain hinzugefügt werden.

4.3. Basics

Zunächst wird betrachtet wie die Ports oder auch spezifische Pins über die Library angesteuert werden.

Abbildung 38. Auszug aus main.cpp

```
#include "MCUSelect.hpp"
#include "hw_abstractions/Delay.hpp"

using portA = AVR::port::Port<AVR::port::A>;
using portC = AVR::port::Port<AVR::port::C>; ①
using led1 = AVR::port::Pin<portA,2>;
using led3 = AVR::port::Pin<portA,3>;
using led4 = AVR::port::Pin<portA,4>;
using led5 = AVR::port::Pin<portA,5>; ②

using led2 = typename portC::pins::pin3; ③

int main() {

    portA::setDir<AVR::port::Out>(); ④
    led2::setOutput(); ⑤
    while(true){
        led1::toggle();
        if(led2::isOn())
            led3::toggle();
        led2::toggle();
        AVR::port::pinsOutToggle<led4,led5>(); ⑥
        AVR::delay<AVR::ms,500>(); ⑦
    }

}
```

① Portauswahl

② Pinauswahl

③ Sichere Pinauswahl über den Port

④ Pin auf Output setzen mit Port Funktion

⑤ Pin auf Output setzen mit Pin Funktion

⑥ Mehrere Pins mit freier Funktion togglen

⑦ Delay-Funktion

Zunächst werden die Ports ausgewählt welche angesprochen werden sollen, hier in diesem Falle Port A und C. Sollte bei der Auswahl ein Compiler Fehler auftreten wurde ein nicht existenter Port ausgewählt (der Atmega4808 besitzt z.B. keinen Port B oder E obwohl es Defines dafür gibt). Der nächste Schritt ist die Auswahl von spezifischen Pins, dafür wird dem Template-Alias der zuvor gewählte Port sowie die Pin Nummer angegeben. Hier muss darauf geachtet werden dass die spezifischen Pins NICHT auf Verfügbarkeit geprüft werden, um sicher zu gehen können die Pins auch aus dem Port entnommen werden (s.a. Led2), die hier enthaltenen Pins sind definitiv physikalisch vorhanden. Nachdem die Auswahl für Ports und/oder Pins getroffen wurden können auf diesen Operationen ausgeführt werden. Es könne sowohl auf den Ports als auch auf den Pins spezifische Operationen ausgeführt werden (z.B. toggle, on, off,setDir ….). Sollen auf mehreren Pins

gleichzeitig Operationen ausgeführt werden, so können die freien Funktionen in AVR::port genutzt werden. Bei den freien Funktionen ist zu beachten, dass nur Pins des gleichen Ports übergeben werden können, verschiedene Ports lösen einen Compiler-Fehler aus. Die Port Klasse besitzt noch die Convenience-Methode „get“ um auf Register zuzugreifen, damit kann von außerhalb auf ein spezifisches Register (z.B. pin0ctrl) zugegriffen werden. Wenn möglich sollte diese Methode aber vermieden und vorhandene abstraktere Klassen-Methoden genutzt werden. Im obigen Beispiel findet sich die `delay-Methode`, diese wrappt die C _delay Methoden und rechnet je nach Einheit (us, ms, s) korrekt um. Des weiteren gibt es noch eine `safeDelay-Methode`, diese schaltet die Interrupts für die Dauer des Wartens ab und funktioniert somit immer korrekt (sollten die Interrupts angeschaltet gewesen sein werden sie nach Ablauf des Delays wieder aktiviert).

4.4. RessourceController

Die Library stellt eine Klasse zur Verfügung mit welcher die Ressourcen auf ihre Kollisionsfreiheit überprüft werden können. Wenn diese Klasse genutzt wird ist es ausgeschlossen dass darin angeforderte Ressourcen sich überschneiden, wenn z.B TWI PortA Pin 3 braucht und eine weitere Ressource diesen Pin belegen wollen würde, so würde das Programm nicht kompilieren und eine entsprechende Fehlermeldung ausgegeben.

Abbildung 39. Auszug aus main.cpp

```
using twiInstance = AVR::rc::Instance<
        AVR::twi::TWI,
        AVR::rc::Number<0>,
        AVR::portmux::PortMux<0>>;

using usartInstance = AVR::rc::Instance<
        AVR::usart::USART_Comp,
        AVR::rc::Number<2>,
        AVR::portmux::PortMux<0>>; ①

using RC = AVR::rc::RessourceController<twiInstance,usartInstance>; ②

using twiRessource = RC::getRessource<twiInstance>;
using usartRessource = RC::getRessource<usartInstance>; ③

using usart = AVR::usart::USART<AVR::blocking,usartRessource>;

using twi = AVR::twi::TWIMaster<AVR::blocking,twiRessource , AVR::ReadWrite>; ④
```

① Auswahl der gewünschten Ressourcen Instanzauswahl

② Instanziieren des RessourceController mit den gewünschten Ressourcen

③ Anfordern einer Ressource

④ Übergeben einer Ressource

Zuerst wird die spezifische Instance einer Ressource festgelegt, diese besteht aus der Ressource selbst, diese sind immer in den dazugehörigen namespaces abgelegt (z.B. TWI/USART). Danach wird der RessourceController mit den zu benutzenden Instanzen belegt, danach kann mit der

getRessource Klassen-Methode die Ressource angefordert werden. Sollten anderweitig noch Pins genutzt werden (z.B vom Benutzer für led`s oder sonstiges) so kann/sollte eine GenericRessource erzeugt und übergeben werden (Auch hier wird die physikalische Existenz der Pins nicht überprüft!).

Abbildung 40. Auszug aus main.cpp

```
using led1 = Pin<PortA, 0>;
using led2 = Pin<PortA, 1>;
using led3 = Pin<PortD, 6>;

using gRes = AVR::rc::GenericRessource<led1,led2,led3>; ①

using RC = AVR::rc::RessourceController<twiIntance,usartInstance,gRes >; ②
```

① Instanziieren einer generischen Ressource

② Instanziieren des RessourceControllers mit der generischen Ressource

Wenn alle Ressourcen mit dem RessourceController überprüft werden ist ein Konflikt der Ressourcen Belegung zur Compile-Zeit ausgeschlossen. Zu beachten ist allerdings dass bei den meisten Ressourcen wie z.B. TWI, USART und SPI alle Pins welche dort benutzt werden können auch geprüft werden. Sollte also z.B. MasterSPI als ReadOnly konfiguriert werden so wird logischerweise der Miso-Pin nicht benutzt, aber im RessourceController trotzdem geprüft.

Abbildung 41. Auszug aus main.cpp

```
using spiRessource = AVR::rc::Instance<
AVR::spi::SPI,
AVR::rc::Number<0>,
AVR::portmux::PortMux<0>>;

using uncheckedSPI = AVR::rc::UncheckedRessource_t<spiRessource>; ①
using spi = AVR::spi::SPIMaster<AVR::notBlocking<AVR::UseFifo<42> ,AVR::Interrupts<>
>,uncheckedSPI , AVR::ReadOnly>; ②
```

① Instanziieren einer ungeprüften Ressource

② Übergabe der ungeprüften Ressource an SPI

Sollte der besagte Pin trotzdem von einer anderen Ressource verwendet werden besteht die Möglichkeit eine Ressource von der Prüfung auszuschließen indem UncheckedRessource verwendet wird.

4.5. Eventsystem

Das Eventsystem der 0 Serie ermöglicht Peripherie Abhängige Funktionalitäten mit deren Hilfe die Peripherie ohne Interrupts Synchron und Asynchron miteinander kommunizieren kann. Das Eventsystem kann nur so konfiguriert werden dass je Kanal ein Generator läuft und dieser Kanal jeweils von beliebig vielen Users abgehört werden kann. Das Eventsystem ist hauptsächlich dazu gedacht Interrupts und den damit verbundenen Overhead zu vermeiden.

Generatoren:

- Je Kanal kann ein Generator eingeschaltet werden, diese haben bestimmte Bedingungen um Events auszulösen, z.B. löst der SPI-Generator immer zum Master-Takt ein Event aus.
- Beachtet werden müssen hier auch die Constraints, Wenn z.B. ein PortPin als Generator festgelegt wird, soll ein Event ausgelöst werden bei einem High-Pegel, dazu muss aber der Pegel für mindestens einen Taktzyklus Stabil sein.
- Des weiteren muss auch die Eventlänge beachtet werden, ein Generator kann ein Event z.B. als Pulse oder auch als Level ausgeben (hat wiederum Auswirkungen auf den User).
- Die Bedingungen, die Eventlänge sowie die Constraints müssen vor der Verwendung im Handbuch nachgelesen werden.

Es können auch Software-Events generiert werden (immer Synchron als Strobe).

User:

- Je Kanal können beliebig viele User registriert werden, die Benutzer reagieren dann auf die Events welche von dem Generator erzeugt werden.
- Wenn z.B. ein Pin als User gewählt wird, so wird das Event auf den Pin abgebildet, also falls ein Event generiert wird, so ist der Pegel High, sonst Low (beachte Eventlänge).

Wichtig ist zu beachten ob die Events die der Generator erzeugt Synchron oder Asynchron sind, dies muss im Handbuch nachgelesen werden.

Abbildung 42. Auszug aus main.cpp

```
using ch0 = AVR::eventsystem::Channel<0>; ①
using ch0_gen = typename ch0::generators::generals::spi0_sck; ②
using user0 = typename ch0::users::evtca0;
using user1 = typename ch0::users::evtcb0; ③

int main() {
    ch0::template registerListener<user0,user1>(); ④
    ch0::template setGenerator<ch0_gen >(); ⑤
    ch0::softwareEvent(); ⑥
}
```

① Kanalauswahl

② Generatorauwahl

③ Userauswahl

④ User auf Kanal registrieren

⑤ Generator an Kanal zuweisen

⑥ Software-Event auslösen

Zunächst wird der Kanal ausgewählt welcher benutzt werden soll (Kanal 0- maximal 8), sollte hier ein ungültiger Kanal gewählt werden wird das Programm nicht kompilieren. Als nächstes wird der Generator und der/die Benutzer ausgewählt. Dann werden die Benutzer auf den Kanal eingestellt

und der Generator wird für den Kanal eingestellt. Das Eventsystem ermöglicht des Weiteren das Auslösen von Software-Events, dies wird mir der entsprechenden Funktion ermöglicht.

4.6. TWI

Die TWI Schnittstelle ist aus Aufwandsgründen nur für den Master Modus eingerichtet, die Quick-Command Funktionalität wurde aus Gründen der Kompatibilität zwischen TWI und I²C abgeschaltet, hier wurden Probleme bei der Kommunikation zwischen verschiedenen Geräten sichtbar. Ansonsten kann die TWI-Master Schnittstelle mit allen möglichen Eigenschaften konfiguriert werden, empfehlenswert ist aber lediglich die Anpassung der Zugriffsart (RW / R / W) sowie Blockierend / nicht blockierend oder auch Interruptdriven, ob eine FiFo genutzt wird und Instanzauswahl (wenn die anderen Eigenschaften angepasst werden so ist das Verhalten NICHT getestet), nachfolgend einige Beispiele. Die Frequenzrate wird über die init Methode festgelegt, sofern fastmode plus nicht aktiviert ist (Default Einstellung ist aus), so ist die rate auf 400khz beschränkt.

Es wird nicht kontrolliert ob der Slave die Adresse bestätigt hat ,dieser Fall sollte/muss abgefangen werden!

4.6.1. Beispielbehandlung TWI Address-NACK

Abbildung 43. Auszug aus main.cpp

```
auto rd = twi::scopedRead<43>();

while(! twi::writeComplete()) { ①
    while(! twi::slaveAcknowledged()) { ②
        while (!twi::endTransaction());  ③
        rd = twi::scopedRead<43>(); ④
    }
}
```

① Warten auf Beendigung der Start-Condition

② Acknowledgment abfragen

③ Stop-Condition auslösen

④ Neuer Versuch starten

Hier wird außerhalb der Schnittstelle zuerst effektiv auf das Beenden der Start-Condition gewartet, danach wird eine Stop-Condition ausgelöst damit die Start-Condition wiederholt werden kann. Die Behandlung besteht hier darin es solange zu versuchen bis der Slave bestätigt, andere Behandlungsmethoden sind z.B. das versuchen einer anderen Adresse oder eine andere Programm-Routine auszuführen. Eine weitere Möglichkeit ist der Schnittstelle als 4. Parameter einen Handler in form eines Lambda oder einer Funktion mitzugeben, per Default Parameter ist keiner festgelegt (`AVR::DefaultNackHandler`). Intern wird nach der Start-Condition gewartet bis die Adresse geschrieben wurde und der Slave geantwortet hat, sollte hier ein NACK festgestellt werden so wird der NackHandler so lange ausgeführt bis ein ACK erhalten wurde oder der Handler ein true zurückgibt (z.B. könnte hier Punk 3 und 4 als Lambda und return false an die TWI-Schnittstelle

übergeben werden).

Abbildung 44. Auszug aus main.cpp

```
using _twi = AVR::twi::TWIMaster<AVR::blocking, twires, AVR::ReadWrite>; ①

static inline auto handler = [](){
    while (!_twi::endTransaction());
    while(!_twi::startTransaction<42,AVR::twi::access::Read>());
}; ②

using twi = AVR::twi::TWIMaster<AVR::blocking, twires, AVR::ReadWrite, handler>; ③
```

① Deklarieren der Schnittstelle ohne Handler

② Definieren des Handlers

③ Einsetzen des Handlers in die Schnittstelle

Hier wird nochmal gezeigt wie so ein built-in Handler realisiert werden kann.

4.6.2. Lesen mit FiFo, keine Interrupts

Abbildung 45. Auszug aus main.cpp

```
using twi = AVR::twi::TWIMaster<AVR::notBlocking<AVR::UseFifo<42>,AVR::NoInterrupts
>,twires , AVR::ReadOnly>;

volatile bool wasread = false;

static inline void Callback (){
    wasread=true;
    uint8_t item;
    while(twi::getInputFifo().pop_front(item))
        AVR::dbgout::put(item);
} ①

int main() {
    twi::init<100000>(); ②
    AVR::dbgout::init();
    while(true){
        twi::get<42,Callback>(12); ③
        while(!wasread)
            twi::periodic(); ④
        AVR::dbgout::flush();
        AVR::delay<AVR::ms,200>();
        wasread = false;
    }
}
```

① Definition des Read-Callbacks

② Initialisieren der TWI-Schnittstelle mit 100 KHZ-Frequenz

③ Lese-Anforderung mit Callback

④ Periodic Aufruf

Dieses Beispiel zeigt die Konfiguration der Schnittstelle als nicht blockierend, ReadOnly, mit Fifo und ohne Interrupts, bei der TWI Schnittstelle wurde auf eine nicht blockierende Variante ohne FiFo verzichtet, da diese Version mangels Abstraktion zu fehleranfällig wäre. Die get Methode wird mit den Template Parametern der Adresse und einer Callback Funktion (oder auch Lambda) versehen, als Parameter wird die Anzahl der zu lesenden Bytes übergeben. Ohne Interrupts muss in der Schleife `periodic` aufgerufen werden, sonst werden weder Daten empfangen noch gesendet. Wenn der Lesevorgang vollendet wurde (die angegebene Anzahl an Bytes wurde gelesen), wird die Callback Funktion aufgerufen und der Benutzer kann festlegen was passieren soll (sinnvoll ist auf jeden fall die Fifo zu leeren wie im Beispiel).

4.6.3. Schreiben mit FiFo, keine Interrupts

Abbildung 46. Auszug aus main.cpp

```
static constexpr const char* hello = "Hello Slave";

int main() {
    twi::init();

    static constexpr auto len = utils::strlen(hello);

    while(true){
        twi::put<42>((uint8_t*)hello,len);
            while( twi::dataToSend()) ①
                twi::periodic();

        AVR::delay<AVR::ms,200>();
    }
}
```

① Überprüfen ob in der Fifo nicht gesendete Daten sind

Hier im Beispiel wird die Verwendung des Schreibens gezeigt, hier wird solange `periodic` aufgerufen bis keine Daten zum Schreiben mehr vorhanden sind.

4.6.4. Lesen nicht blockierend, keine Interrupts, keine FiFo

Abbildung 47. Auszug aus main.cpp

```
using twi = AVR::twi::TWIMaster<AVR::notBlocking<AVR::NoFifo,AVR::NoInterrupts>,twires
, AVR::ReadOnly>;
static constexpr uint8_t size = 12;
static uint8_t arr[size];

int main() {
    twi::init();
    while(true){
        while(!twi::startTransaction<42,AVR::twi::access::Read>()); ①
        uint8_t tmp = 0;
        while(tmp != size){
            tmp += twi::receive(arr,size-tmp); ②
        }

        AVR::delay<AVR::ms,200>();

    }
}
```

① Auslösen der Start-Condition für lesen

② Lesen mit Rückgabe der gelesenen Bytes

wird keine FiFo genutzt dann wird von der `receive` Methode die Anzahl gelesener Bytes zurückgegeben (bei einem einzelnen Lesevorgang ein Boolean ob ein Byte gelesen wurde), logischerweise muss der Anwender ohne FiFo eigene Maßnahmen treffen um entsprechend viele Bytes zu lesen (siehe Beispiel code). Auch das Auslösen der Start – sowie StopCondition müssen dann manuell eingepflegt werden, Ausnahme betrifft das lesen, hier wird die Stop-Condition mit dem lesen des letzten Bytes automatisch ausgeführt.

4.6.5. Lesen nicht blockierend, Scoped-Variante

Abbildung 48. Auszug aus main.cpp

```
static constexpr uint8_t size = 12;
static uint8_t arr[size];
int main() {
    twi::init();
    while(true){
        uint8_t tmp = 0;
        auto rd = twi::scopedRead<42>(); ①
        uint8_t data;
        while(tmp != 11){
            if(rd.receive(data)) ②
                arr[tmp++] = data;
        }
        while(!rd.receiveLast(data)); ③
        arr[tmp] = data;
}
```

① Erzeugen einer Scoped-Transaktion

② Lesen eines Bytes

③ Lesen des Letzten Bytes

Alternativ können bei nicht verwenden der nicht blockierenden NoFifo Varianten auch Scoped Varianten genutzt werden, somit muss sich nicht mehr explizit um die Start/Stop-Conditions gekümmert werden, auch hier gilt die Ausnahme des Lesens (siehe Beispiel).

4.6.6. Schreiben nicht blockierend, Scoped-Variante

Abbildung 49. Auszug aus main.cpp

```
static constexpr const char* hello = "Hello Slave";

int main() {
    twi::init();

    static constexpr auto len = utils::strlen(hello);

    while(true){
        auto wr = twi::scopedWrite<42>();
        for(uint8_t i = 0; i < len; ) {
            if (wr.send((uint8_t) *(hello + i)))
                i++;
        }

        AVR::delay<AVR::ms,200>();
    }
}
```

Hier wird noch die ScopedWrite Variante gezeigt, somit muss der Nutzer sich nicht um Start/Stop-

Conditions kümmern.

4.6.7. Schreiben nicht blockierend, FiFo mit Interrupts

Abbildung 50. Auszug aus main.cpp

```
static constexpr const char* hello = "Hello Slave";
using twi = AVR::twi::TWIMaster<AVR::notBlocking<AVR::UseFifo<42>,AVR::Interrupts
<>>,twires , AVR::WriteOnly>;

ISR(TWI0_TWIM_vect){ ①
    twi::intHandler(); ②
}

int main() {
    twi::init();
    static constexpr auto len = utils::strlen(hello);

    while(true){
        twi::put<42>((uint8_t*)hello,len);
        AVR::delay<AVR::ms,200>();
    }
}
```

① Definition des entsprechenden Interrupt-Vektors

② Aufruf des Interrupt-Handlers

Diese Interrupt gesteuerte Version wird im Intervall von 200 ms den String Hello Slave an den Slave mit der Adresse 42 senden.

4.6.8. Lesen nicht blockierend, FiFo mit Interrupts

Abbildung 51. Auszug aus main.cpp

```
using twi = AVR::twi::TWIMaster<AVR::notBlocking<AVR::UseFifo<42>,AVR::Interrupts
<>>,twires , AVR::ReadOnly>;

volatile bool wasread = false;

ISR(TWI0_TWIM_vect){
    twi::intHandler();
}

static inline void Callback (){
    wasread=true;
    uint8_t item;
    while(twi::getInputFifo().pop_front(item))
        AVR::dbgout::put(item);
}

int main() {
    twi::init();
    AVR::dbgout::init();
    while(true){
        twi::get<42,Callback>(12);
        while(!wasread)
            ; ①
        AVR::dbgout::flush();
        AVR::delay<AVR::ms,200>();
        wasread = false;
    }
}
```

① Kein Periodic da Interruptgesteuert

Diese Version erhält vom Slave mit der Adresse 42 je Iteration 12 Datenbytes, nachdem diese gelesen wurden wird der Callback ausgeführt (in diesem Falle werden die Datenbytes über die Debug-Schnittstelle an den PC gesendet → nur 4808 verfügbar und die innere While-Schleife der main Methode unterbrochen).

4.6.9. Lesen und schreiben nicht blockierend, FiFo mit Interrupts

Abbildung 52. Auszug aus main.cpp

```
using twi = AVR::twi::TWIMaster<AVR::notBlocking<AVR::UseFifo<42>,AVR::Interrupts
<>>,twires , AVR::ReadWrite>;

ISR(TWI0_TWIM_vect){
    twi::intHandler();
}

volatile bool wasread = false;
static inline void Callback (){
    wasread=true;
    uint8_t item;
    while(twi::getInputFifo().pop_front(item))
        AVR::dbgout::put(item);
}

int main() {
    twi::init();
    AVR::dbgout::init();
    while(true){
        twi::get<42,Callback>(12); ①
        while(!wasread)
            ;
        AVR::dbgout::flush();
        twi::put<42>('H'); ②
        AVR::delay<AVR::ms,200>();
        wasread = false;
    }
}
```

① Lesen von 12 Bytes

② Schreiben eines Bytes

Als abschließendes Beispiel wird hier noch eine kombinierte Version gezeigt (lesen und Schreiben).

4.7. SPI

Das SPI-Interface ist ähnlich gestaltet wie bei TWI, mit der Ausnahme dass hier in keinem Fall auf Start- oder Stop-Conditions oder dergleichen geachtet werden muss. Weiterhin wird auch ein Interrupthandler für die Benutzung ohne Fifo und der Slave Modus zur Verfügung gestellt. Hier gibt es keine Option in der init Methode die Frequenz einzustellen, dies geschieht über die Prescaler-Option während der Schnittstellen-Konfiguration.

4.7.1. Schreiben, nicht blockierend, FiFo ohne Interrupts

Abbildung 53. Auszug aus main.cpp

```
using spi = AVR::spi::SPIMaster<AVR::notBlocking<AVR::UseFifo<42> ,AVR::NoInterrupts
>,res, AVR::ReadWrite>;

int main() {
    spi::init();
    AVR::dbgout::init();
    while(true){
        spi::put('h');
        spi::put('e');
        spi::put('l');
        spi::put('l');
        spi::put('o');
        for(uint8_t i = 0; i < 5; i++) {
            spi::periodic();
            AVR::delay < AVR::us, 5 > ();
        }
        uint8_t item;
       while(spi::getInputFifo().pop_front(item)) {
           AVR::dbgout::put(item);
       }
        AVR::dbgout::flush();
    }
}
```

Die Nicht blockierende Variante ohne Interrupts lässt sich ebenso verwenden wie die TWI-Schnittstelle.

4.7.2. Schreiben, nicht blockierend, FiFo mit Interrupts

Abbildung 54. Auszug aus main.cpp

```
using spi = AVR::spi::SPIMaster<AVR::notBlocking<AVR::UseFifo<42> ,AVR::Interrupts<>
>,res, AVR::ReadWrite>;

ISR(SPI0_INT_vect){
    spi::intHandler();
}

int main() {
    spi::init();
    AVR::dbgout::init();
    while(true){
        spi::put('h');
        spi::put('e');
        spi::put('l');
        spi::put('l');
        spi::put('o');
        uint8_t item;
       while(spi::getInputFifo().pop_front(item)) {
           AVR::dbgout::put(item);
       }
        AVR::dbgout::flush();
    }
}
```

Hier zu sehen ist eine Interrupt-Version des SPI, um die gelesenen Bytes auszugeben muss lediglich die Fifo ausgelesen werden (siehe Beispiel). Wird eine ReadWrite Variante genutzt so wird die Library eine Null transferieren sofern keine Nutzdaten vorhanden sind. Sollte keine Fifo verwendet und kein WriteOnly eingestellt sein so muss ein Protokolladapter festgelegt werden.

4.8. USART

Die USART Schnittstelle lässt sich im Grunde genauso bedienen wie SPI und TWI.

Hier müssen besonders 2 Dinge beachtet werden:

- Baudrate: Hier ist es essentiell wichtig dass vor allem der F_CPU Wert stimmt und das eine zu dem Ziel-Gerät passende Baudrate eingestellt ist (default 115200).
- Besonderheit Interrupts: Hier gibt es 2 Interrupt Vectoren (der TX-Vector wird nicht beachtet) und deswegen auch 2 Interrupt-Handler (tx/rxHandler).

Für den Atmega4808 ist ein Debug-Interface in der Library enthalten (AVR::dbgout → include Datei = inc/Boards/CuriosityNanoIOT.hpp), mit diesem kann für das IoT-Devboard von Microchip über das USB-Interface bequem kommuniziert werden, dabei ist zu beachten das dieses blockierend arbeitet und USART2 benutzt.

Abbildung 55. Auszug aus main.cpp

```
using usart =AVR::usart::USART<AVR::notBlocking<AVR::UseFifo<42>, AVR::Interrupts
<>>,usartres, AVR::ReadWrite>;

ISR(USART2_DRE_vect){
    usart::txHandler();
} ①
ISR(USART2_RXC_vect){
    usart::rxHandler();
} ②

int main() {
    usart::init<115200>(); ③
    while(true){
        usart::put('h');
        usart::put('e');
        usart::put('l');
        usart::put('l');
        usart::put('o');
        AVR::delay<AVR::ms,200>();
    }
}
```

① Transfer Vektor (DRE)

② Receive Vektor (TXC)

③ Initialisierung der Schnittstelle mit 115,2 KHZ-Baudrate

Hier im Beispiel wird die Verwendung der Interrupt Variante für Read und Write Zugang beschrieben, beachtet werden muss dass der TX Vector nicht verwendet werden darf, stattdessen muss der DRE Vector verwendet werden.

Quellen

- [ZLib] Keven Klöckner. Atmega 0 Series Lib: https://github.com/Keven1994/Atmega0SeriesLib
- [libcpp] Free Software Foundation, Inc. Libstdc++: https://gcc.gnu.org/
- [BMCPP] Prof. Dr.-Ing Wilhelm Meier. BmCPP https://sourceforge.net/projects/wmucpp/
- [MDok] Microchip. Atmega 0-series Family Data Sheet http://ww1.microchip.com/downloads/en/DeviceDoc/megaAVR0-series-Family-Data-Sheet-DS40002015B.pdf
- [ASF] Microchip. Advanced Software Framework: http://ww1.microchip.com/downloads/en/DeviceDoc/asf-standalone-archive-3.46.0.94.zip
- [PUGI] Arseny Kapoulkine. Pugi XML-Parser: https://pugixml.org/
- [STM] STMicroelectronics. STM Hal Dokumentation: https://www.st.com/content/ccc/resource/technical/document/user_manual/2f/71/ba/b8/75/54/47/cf/DM00105879.pdf/files/DM00105879.pdf/jcr:content/translations/en.DM00105879.pdf