Предложения по дизайну

Полностью новый подход

Что не так с легаси архитектурой и кодом

Код плохо подаётся изменению в других проектах и плох в переиспользовании из-за нарушения принципов ООП, например, SOLID

Нарушение принципа Единственной ответственности (S — The Single Responsibility Principle)

> У программной сущности должна быть только одна причина для изменения.

> Если несколько программных сущностей изменяются вместе по одним и тем же причинам, то на самом деле это одна программная сущность.Объедините их немедленно.

Пример ниже показывает, что кроме расчёта давления происходит еще установка битов в Comprehensive Status. Т.е. если мы хотим использовать класс давление в другом проекте с другими статусами, нам придется бегать по нему и искать какие статусы где надо поменять и где вообще они меняются? Таким образом, изменение в устанавливаемых битов, приведет к изменению GlobalStatus.

```
void cPressure::notifyGoodUpdate(void)
{
   tF32 compensatedPressure;
   const tSaturationDirection eSatDir = oPressureCore.calculate(compensatedPressure);
   tDeviceVarStatus eVarStatus;
   ...
   oInternalValue.set(compensatedPressure, eVarStatus, eSatDir,
                      oRealTimeClock.getCurrentDateTime().timeSinceMidnight);

   notify();
   oPressureFilter.updateVal(compensatedPressure);

   if(isProcessAlertTriggered(getFilteredInternalValue()))
   {
      oGlobalStatus.setComprehensiveStatus(CS_pressureAlert);  # (1)
   }
   else
   {
      oGlobalStatus.clearComprehensiveStatus(CS_pressureAlert); # (2)
   }
}
```

NOTE Возможное решение, сохранять статус ошибки в самом объекте и дать интерфейс для их считывания.

По большому счёту можно выделить Pressure в отдельный компонент, с настройками, например, подписчики на обновление давления, биты для установки в случае ошибок, единицы измерения и так далее... Это конечно сложная задача, но вполне решаемая.

Не знал куда этот пример отнести, но точно плохо. Одновременно у нас не будет Taconite и Marble, поэтому логично, что иметь два метода, один из которых не особо нужен в одном классе не хорошо. Лучше наверное сделать два различных класса - один ответственнен за работу с Marble, другой с Taconite

```
class cAsic
{
public:
    tAsicStatus readCountsMarble(void);
    tAsicStatus readCountsTaconite(void);
....
}
```

NOTE Возможное решение, сделать базовый класс для Asic с чисто виртульной функцией readCounts.

Нарушение принципа открытости/закрытости (O — The Open/Closed Principle)

> Программные сущности должны быть открыты для расширения и закрыты для модификации.

Пример, если нам нужно добавить новый детальный статус, то нам придется переделать метод **findDetailedStatusRelation** в **cGlobalStatus**. Тоже самое касается других методов и статусов в классе **cGlobalStatus**.

```
#define DS_outputBoardNonCorrectableWarning  (tDetailedStatus)0x000800000000
#define DS_displayUpdateFailure              (tDetailedStatus)0x000400000000
//unused                                     (tDetailedStatus)0x000200000000
#define DS_boardTemperatureOutOfLimits       (tDetailedStatus)0x000100000000
 ...

class cGlobalStatus
{
   ...
  private:
    tDetailedStatus findDetailedStatusRelation(const tComprehensiveStatus bitId,
                                       tComprehensiveStatus& statusMask) const;
};
```

NOTE	Возможное решение, вынести маппинг битов в отдельную сущность.

Принцип подстановки Лисков (L — The Liskov Substitute Principle)

> Наследующий класс должен дополнять, а не замещать поведение базового класса.
>
> Если базовый класс проходит определённый юнит-тест, то его должны проходить все наследники базового класса тоже.


```
class cVariableExternal
{
   public:
      ...
      virtual tDeviceVarStatus getStatus(void) const; // Наследуемый класс может заместить поведение базового класса
      virtual tF32 getExternalMinSpan(void) const = 0;

      virtual tSec getDamping(void) const {return (tSec)NAN;} // Наследуемый класс может заместить поведение базового
      virtual tHartRespCode setDamping(const tSec newTimeConstant) { return HRC_invalidSelection; } //Тоже самое

      virtual tDeviceVarClassificationCode getClassification(void) const { return DVCC_notYetClassified; } //Также
      virtual tU24 getSensorSerialNumber(void) const; // И еще раз
      virtual tS8 getDisplayPrecision(void) const = 0;
      ...
};

// а потом еще и использовать "костыли" в виде виртуального наследования
class cVariableMappable : virtual public cVariableExternal, public cSubject
```

NOTE	Возможное решение, функции базового класса должны быть либо чисто виртуальные, либо не виртуальные.

Принцип разделения интерфейсов (I — The Interface Segregation Principle)

> Программные сущности не должны зависеть от частей интерфейса, которые они не используют (и знать о них тоже не должны).

Вроде с интерфейсами все не плохо, но есть пару возможностей

```
class cVariableExternal
{
   public:
      ...
      tF32 getExternalValue(void) const;
      virtual tDeviceVarStatus getStatus(void) const;

      HART::t32thsOfMs getTimestamp(void) const; // а всем переменным нужна эта хрень? поэтому можно в отдельный интерфейс
```

NOTE	Возможное решение, сделать отдельные интерфейсы для каждого случая. Выделить в базовый класс действительно только общие вещи.

Принцип инверсии зависимости (D — The Dependency Inversion)

> Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
>
> Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

```
class cPressureTrim
{
   public:
      cPressureTrim(cIirFilterConfig& oFilterConfig, const tF32 minTrimSpanValue) : oFilter(oFilterConfig) {};

   private:
      cIirFilter oFilter;    //Жетская зависимость от cIirFilter и его реализации
};
```

NOTE	Возможное решение, создать объект фильтр вне, а в cPressureTrim передать ссылку на cFilter

Переиспользование компонентов

В прошлых проектах, мы переиспользовали код путём простого копирования в другой проект. Это с одной стороны позволяло быстро сделать новый проект, с другой стороны между кодовой базой не было никакой синхронизации и если что-то менялось в базе, нужно было те же самые изменения делать и в других проектах.

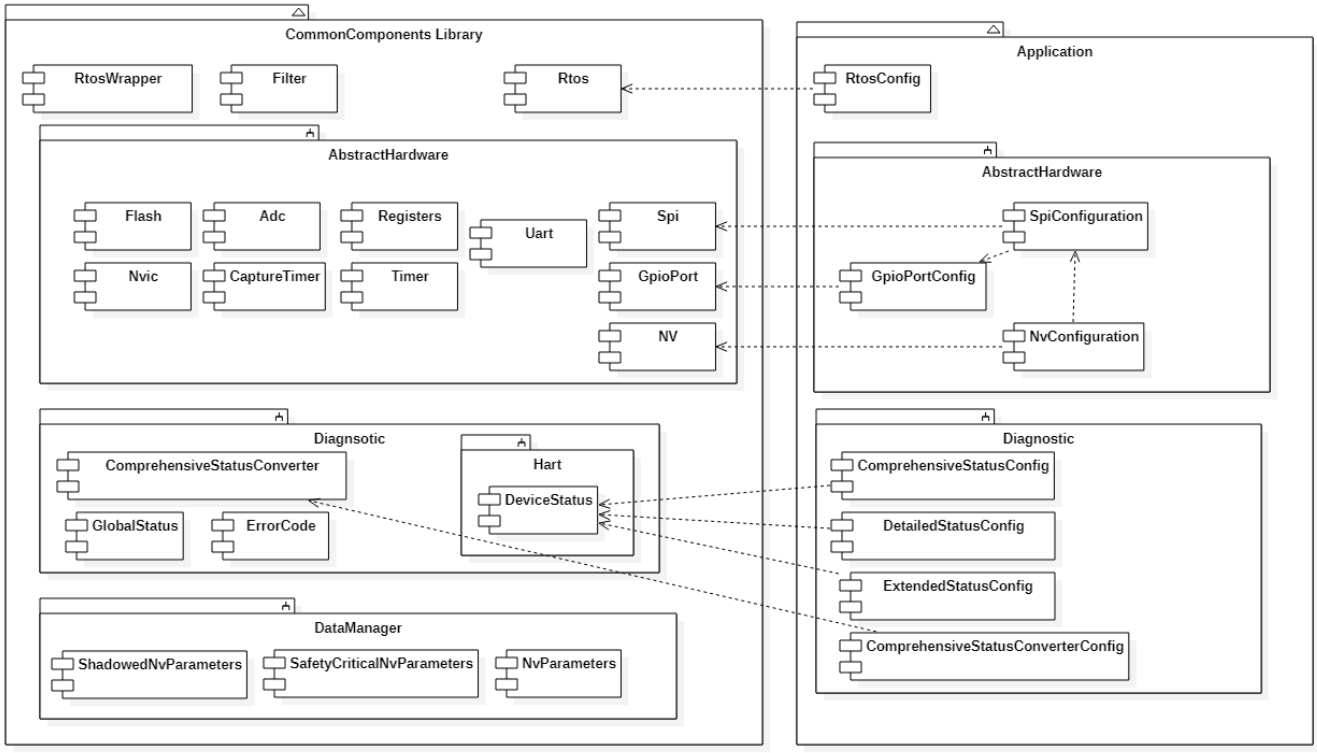
Кроме того, так как структура была не модульной, а практически цельной, очень много зависимостей, которые требовали изменения в разных частях кода. Поэтому в новых проектах различия было трудно искать в коде, трудно переделывать юнит тесты и так далее…

Предлагаемый подход - это полное развязывание компонентов друг от друга, и использование их в разных проектах. Использование реально общей кодовой базы и выбор нужных компонентов для нового проекта с последующей настройкой.

Например:

- CortexM4 Аппаратные модули
- Обертка, позволяющая осуществлять безопасный доступ к регистрам
- Диагностика
- Nv параметры
- Simple Tasker (Rtos)
- Общая обертка над любыми RTOS
- Другие модули, например, Фильтры и так далее

Такой подход был опробован в RML (И где он сейчас :)) Подход RML позволял использовать компоненты в разных приложениях без модификации и минимальными требованиями к ресурсам микроконтроллера. Приложение должно только сконфигурировать необходимый компонент.



Пример настройки таймеров RTOS для задач:

```
// rtostimersconfig.hpp

#pragma once
#include "rtostimer.hpp"        // For RtosTimer common component
#include "rtostimerservice.hpp" // For RtosTimerService commmon component
#include "tasksconfig.hpp"      // For led1Task, measTask, frequencyTransmissionTask


//Timer for Led1 task
using tLed1Timer = RtosTimer<
  led1Task, // подписчик на событие от таймера
  1000U, //time in ms
  tTaskEvents(Led1TaskEvents::togglePin)> ; //событие

//Timer for MeasurementDirector task
using tMeasTimer = RtosTimer<
  measTask, // подписчик на событие от таймера
  100U, //time in ms
  tTaskEvents(MeasurementDirectorTaskEvents::calculate)> ;

//Timer for Frequency Transmition task
using tFrequencyTransmissionTimer = RtosTimer<
  frequencyTransmissionTask, // подписчик на событие от таймера
  1000U, //time in ms
  tTaskEvents(FrequencyTransmissionEvents::transmitFrequency)> ;

using tRtosTimerService =
  RtosTimerService<tLed1Timer, tMeasTimer, tFrequencyTransmissionTimer> ;
```

Настройка задач

```
//Filename      : tasksconfig.hpp

#pragma once

#include "itask.hpp"                            // For ITask
#include "rtos.hpp"                             // For Rtos
#include "testtasks.hpp"                        // For Led1Task
#include "measurementdirector.hpp"              // For MeasurementDirector
#include "measurementdirectorconfig.hpp"        // For SensorBoardFrequencyProcessing
#include "frequencytransmissiondirector.hpp"    // For FrequencyTransmissionDirector

//Tasks: global objects
inline Led1Task led1Task ;
inline MeasurementDirector<SensorBoardFrequencyProcessing> measTask ;
inline FrequencyTransmissionDirector frequencyTransmissionTask ;

enum class TaskPriorities : tTaskPriority
{
  lowest = 1U,
  low = 2U,
  medium = 3U,
  high = 4U,
  highest = 5U
} ;

//Configuration of Tcb block of task Led1
inline constexpr TaskControlBlock tcb1
{
  &led1Task,
  TaskPriorities::low
} ;

//Configuration of Tcb block of MeasurementDirector task
inline constexpr TaskControlBlock tcb2
{
  &measTask,
  TaskPriorities::low
} ;

//Configuration of Tcb block of Frequency Transmission task
inline constexpr TaskControlBlock tcb3
{
  &frequencyTransmissionTask,
  TaskPriorities::low
} ;

using tRtos = Rtos<&tcb1, &tcb2, &tcb3> ; // 3 задачи
```

И потом используем это так:

```
int main()
{
  //Start three tasks: frequencyTransmissionTask, measTask, led1Task
  tRtos::Start(); // Никаких накладных, задачи уже сделаны на этапе компиляции

  return 0;

}
```

C++ 17 преимущества

Меньше кода, как исходного, так и бинарного

Использование constexpr

```
//in displaydriverloi.h file
class cDisplayDriverLoi
{
  ...
  static const tCharacterSegmentTable segmentTable;
}

// in *.cpp file
const tCharacterSegmentTable cDisplayDriverLoi::segmentTable = {
   {0xA8, 0xA8}, {0x00, 0x28}, {0xA4, 0xB0}, /* 0,1,2 */
   {0x84, 0xB8}, {0x0C, 0x38}, {0x8C, 0x98}, /* 3,4,5 */
   ...
   {0x40, 0xB0}, {0x10, 0x40}, {0x00, 0x00}, /* ,?,/,' ' */
   {0x57, 0x54}, {0x1E, 0x5C}, {0x00, 0x54}, /* *,%,left arrow */
   {0x00, 0x80}                              /* ovrscr */
   };
```

```
//in *.h file
class cDisplayDriverLoi
{
  ...
   static constexpr tCharacterSegmentTable segmentTable = {
   {0xA8, 0xA8}, {0x00, 0x28}, {0xA4, 0xB0}, /* 0,1,2 */
   {0x84, 0xB8}, {0x0C, 0x38}, {0x8C, 0x98}, /* 3,4,5 */
   ...
   {0x40, 0xB0}, {0x10, 0x40}, {0x00, 0x00}, /* ,?,/,' ' */
   {0x57, 0x54}, {0x1E, 0x5C}, {0x00, 0x54}, /* *,%,left arrow */
   {0x00, 0x80}                              /* ovrscr */
   };
}
```

Использование inline для переменных и атрибутов

Пример: Использование inline переменных

```
//pressure.h
class cPressure : public cVariableMappable, public cVariableTrimmable,
                    public cVariableAlertable, public cVariableForceable
{
   public:
      explicit cPressure(cIirFilterConfig& oFilterConfig);
...
} ;

extern cPressure oPressure;

// somewhere in the cpp file
cPressure oPressure; // определение объекта oPressure
int main()
{
}
```

C++17

```
//pressure.h
class cPressure : public cVariableMappable, public cVariableTrimmable,
                    public cVariableAlertable, public cVariableForceable
{
   public:
      explicit cPressure(cIirFilterConfig& oFilterConfig);
...
} ;

inline cPressure oPressure; // определение и объявление объекта oPressure. Так как это
inline переменная, то объявление будет только 1 раз, не зависимо от того, сколько раз
заголовочник подключался в cpp файлы.

// somewhere in the cpp file. Не нужно уже определять.
int main()
{
}
```

Пример: Использование inline статических переменных

```
// in sequencescreenset.h
class cSequenceScreenSet: public cScreenSet
{
   ...
   //Variables shared by the functions listed below.
   static tSaveState eSaveState;
}

// in sequencescreenset.h.cpp
tSaveState cDynamicMenuScreenSet::eSaveState = SS_none;
```

C++17

```
// in sequencescreenset.h
class cSequenceScreenSet: public cScreenSet
{
  ...
  //Variables shared by the functions listed below.
  inline static tSaveState eSaveState = SS_none;;
}
```

Использование std::pair и std::tupple

Пример: Упростить интерфейс. Например, одновременно получать и статус и значение параметра.

Вместо:

```
class cInternalValue
{
...
 tF32 getValue(void) const;
 tDeviceVarStatus getStatus(void) const;
};

auto SomeMethod()
{
    const varStatus = internalValue.getStatus() ;
    if (varStatus != tDeviceVarStatus::DVS_goodNotLimited)
    {
       return  internalValue.getValue();
    }
}
```

C++17

```
using tVarValueAndStatus = std::pair<tF32, tDeviceVarStatus> ;

class InternalValue
{
...
 tVarValueAndStatus Get() const;
};


....

auto SomeMethod()
{
    const auto var = internalValue.Get() ;
    if (get<tDeviceVarStatus>(var) != tDeviceVarStatus::DVS_goodNotLimited)
    {
       return std::get<tF32>(var) ;
    }
}
```

Constexpr конструктор. Константные строки с размером сразу

```
struct StringView
{
  const char* str;
  const size_t size;

  template<size_t N>
  explicit constexpr SusuStringView(const char (& s)[N]): str(s), size(N - 1)
  {
  }
  ... //добавить операторы, итераторы и будет вообще красота
};

class KelvinUnits: IUnits
{
private:
  static constexpr SusuStringView unitsStr = StringView("K");
  static constepxr tF32 offset = 273.15f ;

public:
  tSensorValue Get(tF32 value) const override
  {
    return std::make_pair(unitsStr, value + offset);
  }
} ;
```

Использование шаблона с переменным количеством аргументов

Помогает избавиться от необходимости следить за размером массива, его можно вычислить на этапе компиляции:

```
template <const auto&... units>
class Temperature
{
private:
  static constexpr size_t UnitsCount = sizeof...(units) ; //Узнаем количество
аргументов
  static constexpr std::array<const IUnits*, UnitsCount> unitsList = { &units...};

  size_t index = 0U;
public:

  void SetUnits(size_t value)
  {
    (value < UnitsCount) ? index = value ;
  }

  tSensorValue Get(float value)
  {
    auto& currentUnits = *unitsList[index] ;
    return currentUnits.Get(value) ;
  }
};

// Use in some class
class SomeMediator
{
private:
// все создаться на этапе компиляции.
  static constexpr KelvinUnits kelvin = KelvinUnits();
  static constexpr CelsiusUnits celsius = CelsiusUnits();

  Temperature<kelvin, celsius> temperature;
...
}
```

Можно сделать все статически: Пример можно посмотреть здесь

Статическая подписка

Позволяет на этапе компиляции подписать необходимые объекты или классы на события без лишнего кода и гемора.

```
template<tU8 size>
class cRtosHwTimer
{
   public:
      virtual void isrHandler(void);
      void subscribe(cHwTimerSubscriber* pSubscriber);
      ...
   private:
      cRtosHwTimer(const cRtosHwTimer& other);
      const cRtosHwTimer& operator=(const cRtosHwTimer& other);

      static const tU8 maxSubscribersNumber;
      cHwTimerSubscriber* subscribers[size];
      tU8 subscribersNumber;
};
// ненужный метод вообще
template<tU8 size>
void cRtosHwTimer<size>::subscribe(cHwTimerSubscriber* pSubscriber)
{
   ASSERT(subscribersNumber < maxSubscribersNumber); // проверка на длину массива
   subscribers[subscribersNumber] = pSubscriber; //дурацкая подписка
   subscribersNumber++; //лишний счетчик
}

template<tU8 size>
void cRtosHwTimer<size>::isrHandler(void)
{
   ...
   for(tU8 index = (tU8)0; index < subscribersNumber; index++)
   {
      subscribers[index]->timerExpiredNotify();
   }
}

//затем для каждого таймера нужно вызвать метод для подписки. Лишний код и работа
// Мозг можно сломать....
cRtosTimerService::cRtosTimerService(void) : ...
{
   ...

   cRtosHwTimer<TIMER_MULTIPLE_SUBSCRIBERS>& oHalfSecondTimer = oRtosHwTimerService
.getTimerHalfSecond();

   oHalfSecondTimer.subscribe(&timerSensorTemperature);
   oHalfSecondTimer.subscribe(&timerTaskExecutionMonitor);
   oHalfSecondTimer.subscribe(&timerLoiDirector);
   oHalfSecondTimer.subscribe(&timerLoiDirectorMenuMode);
   oHalfSecondTimer.subscribe(&timerLoiDirectorMenuModeExitTimeout);
   oHalfSecondTimer.subscribe(&oTestFixedCurrent);
}
```

C++17

```
template <auto& ...Timers> //если объекты то  template <auto& ...Timers>
struct TaskerTimerService {
    static void OnSystemTick()  //Прерывание по системному тику
    {
        (Timers.timerExpiredNotify(), ...) ; //вызов методов подписчиков
    }
} ;

//Подписываем таймера на сервис от системного таймера. Количество подписчиков не
ограничено...
using tRtosTimerService = TaskerTimerService<timerSensorTemperature,
timerTaskExecutionMonitor, timerLoiDirector, timerLoiDirectorMenuMode,
timerLoiDirectorMenuModeExitTimeout> ;
```

Пример конфигурации UART:

```
// Filename     : uartconfig.hpp

#pragma once

#include "system.hpp"                   // For systemClock
#include "hwuart.hpp"                   // For HwUart
#include "uart.hpp"                     // For Uart4
#include "uart4registers.hpp"           // For Uart4 registers

class tFrequencyTransmitter ; //subsriber for Uart events
using tUart =  Uart<HwUart<Uart4, System::systemClock / System::apbPrescaler>,
tFrequencyTransmitter> ;
```

Использование:

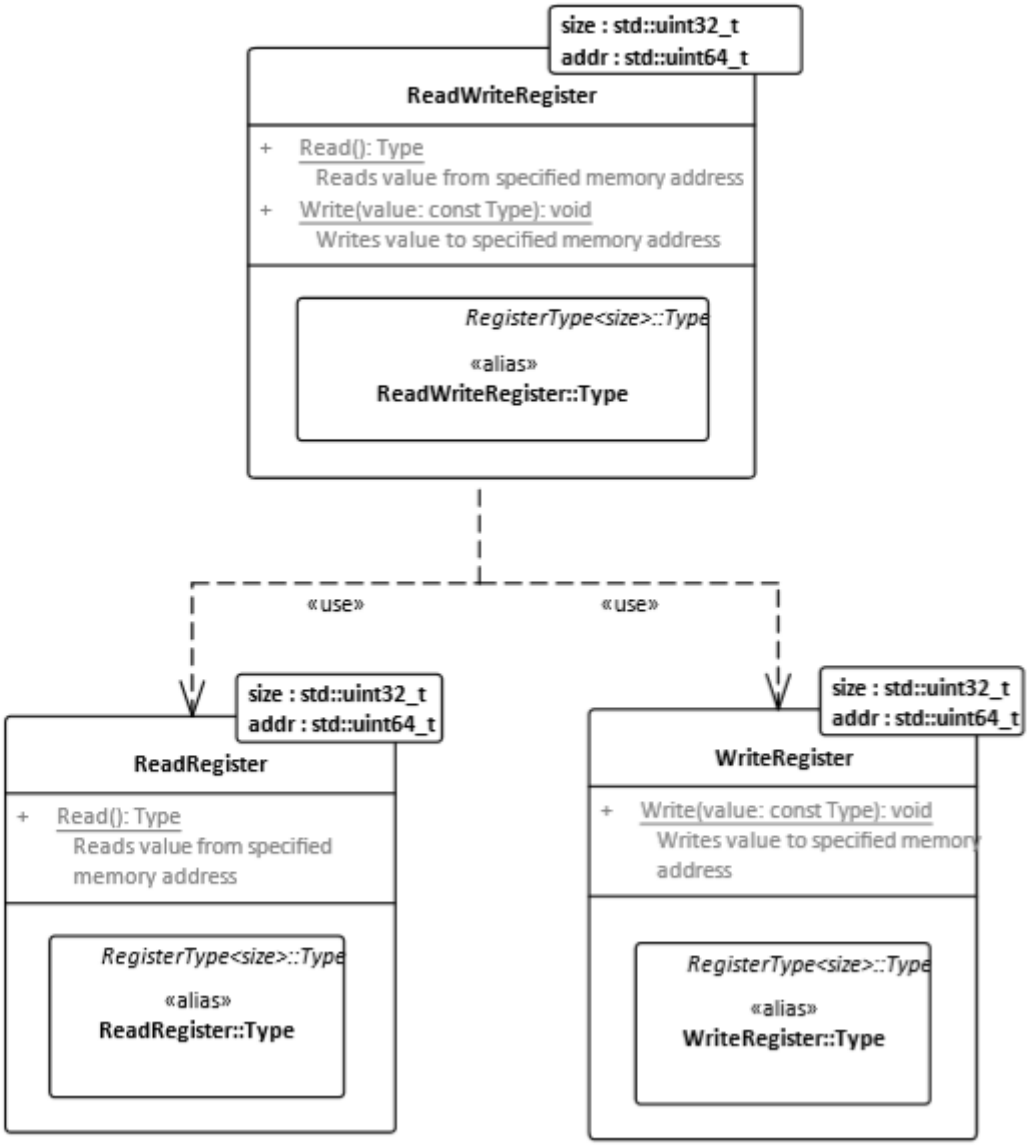
```
int main()
{
  tUart::Enable() ;
  tUart::SetBaudRate(tU32{9600U}) ;
  tUart::SetParity(UartParity::none) ;
  tUart::SetWordLength(UartWordLength::eightDataBits) ;
  tUart::SetStopBitsNumber(UartStopBits::oneBit) ;
  tUart::EnableTransmitter() ;

  return 0;

}
```

Common Library

All design used static polymorphism, but it is possible to use the traditional approach with objects

Register approach

RML does not used the CMSIS at all. Instead of that is used registers which are generated from *.svd files, base on the article It allows to have safe access to the registers.



And example of usage

Register wrappers are static classes that provide methods to read/write values to/from specified memory address.
Do not use these classes directly - use special generator tool to convert SVD file to set of registers descriptions of device.
Example of generated description of peripheral TIM2:

```
struct Tim2
{
 struct Cr1 : public ReadWriteRegister<32, 0x40000000>
 {
  using Ckd = ReadWriteRegisterField<Tim2::Cr1, 8, 2>;
  using Arpe = ReadWriteRegisterField<Tim2::Cr1, 7, 1>;
  using Cms = ReadWriteRegisterField<Tim2::Cr1, 5, 2>;
  using Dir = ReadWriteRegisterField<Tim2::Cr1, 4, 1>;
  using Opm = ReadWriteRegisterField<Tim2::Cr1, 3, 1>;
  using Urs = ReadWriteRegisterField<Tim2::Cr1, 2, 1>;
  using Udis = ReadWriteRegisterField<Tim2::Cr1, 1, 1>;
  using Cen = ReadWriteRegisterField<Tim2::Cr1, 0, 1>;
 };
 ...
};
```

Example of usage:

```
int main()
{
 using tCr1 = Tim2::Cr1::Type;

 auto cr1Value = Tim2::Cr1::Read();

 Tim2::Cr1::Cen::Write(tCr1(0));
 Tim2::Cr1::Ckd::Write(tCr1(3));
 Tim2::Cr1::Cen::Write(tCr1(1));
 auto currentCms = Tim2::Cr1::Cms::Read();
}
```

In the begginig RML uses CMSIS and this how they works with registers:

```
GPIOA->AFR[1] = 0xBB000U ;
BitUtils::SetMask(RCC->APB1ENR,
                  RCC_APB1ENR_TIM2EN | RCC_APB1ENR_TIM3EN |
                  RCC_APB1ENR_TIM4EN | RCC_APB1ENR_UART4EN) ;
NVIC_EnableIRQ(TIM2_IRQn) ;
```

And then RML decided to use autogenerated registers from svd file:

```
Gpioa::Afrh::Afrh11::Write(11U) ;

Rcc::Apb1Enr::Tim2En::Write(RccApb1EnrTim2EnValues::clockEnabled) ;
Rcc::Apb1Enr::Tim5En::Write(RccApb1EnrTim5EnValues::clockEnabled) ;
Rcc::Apb1Enr::Uart4En::Write(RccApb1EnrUart4EnValues::clockEnabled) ;

NvicManager::EnableIrq<Irqn::tim2>() ;
```

It is possible to use another approach for safety access to the registers:

```
 GPIOB::AFRHPack<
        GPIOB::AFRH::AFRH13::Af5,
        GPIOB::AFRH::AFRH15::Af5
    >::Set() ;

 RCC::APB1ENRPack<
        RCC::APB1ENR::TIM4EN::Enable,
        RCC::APB1ENR::TIM2EN::Enable,
        RCC::APB1ENR::TIM3EN::Enable,
        RCC::APB1ENR::USART4EN::Enable
    >::Set() ;

NvicManager::EnableIrq<Irqn::tim2>() ;
```