参赛队员姓名：傅易

中学：上海中学

省份：上海

国家/地区：中国

指导教师姓名：王亚娟

论文题目：模块化单片机开发系统

本参赛团队声明所提交的论文是在指导老师指导下进行的研究工作和取得的研究成果。尽本团队所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果。若有不实之处，本人愿意承担一切相关责任。

参赛队员：傅易　　指导老师：王亚娟

2019 年 8 月 31 日

模块化单片机开发系统

傅易

摘　要　本文在 8/16 位单片机及基于此的开发板盛行的背景下，发现了市售开发板与模块使用不方便的问题，由此设计并制作了一个名为 MEDS 的单片机开发系统的原型以解决。这一系统在硬件与软件上完全模块化；模块有统一的硬件参数，通过 I^2C 总线通信；基于事件驱动架构，扩展了 C++标准库中的 `std::function` 作为回调接口；构造继承体系，将实体模块抽象为对象；开源并使用 C 语言编写统一接口的硬件驱动，使广大开发者都能参与系统的扩展；通过了应用实例的可行性验证，展现出程序结构清晰、使用方便的优点；是一种解决问题的方案，并有广阔的应用前景。

关键词　单片机，模块化，I^2C 总线，C++，泛型编程，事件驱动

Modular MCU Development System

Fu Yi

Abstract In the context of 8/16-bit MCUs and development boards based on them prevail, this paper detects the inconvenience of using commercially available development boards and modules, and designs and makes a prototype of an MCU development system called MEDS to solve it. This system features complete modularity in both hardware and software. Modules have unified hardware parameters, communicating through I^2C bus. Based on event-driven architecture, the system uses an extension to `std::function` in the C++ standard library as callback interface. It constructs an inheritance hierarchy and abstract module entities to objects. The open-source system uses C to provide hardware drivers within a unified interface, and thus general developers can participate in extending it. The system passes the feasibility testing of an application example, showing the clear program structure and the convenience of using. It is a solution to the problem and has a broad application prospect.

Keywords MCU, Modular, I^2C bus, C++, Generic programming, Event-driven

目录

第一章 研究背景

2018 年，全球微控制器市场规模达到 18.60 亿美元[10]。尽管近年来的趋势是 32 位单片机的市场占有率不断增长，但 8/16 位单片机仍有一席之地，并且规模有增无减[12]。

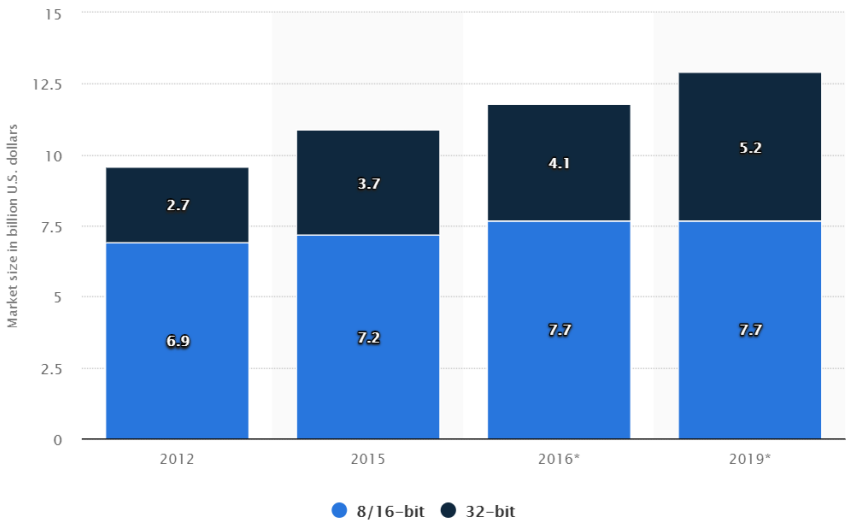


图 1 单片机市场规模与份额

在 Arrow 电子评选出的 2018 年十大开发板中，基于 8 位 AVR 单片机的 Arduino Uno 摘得桂冠[9]。8/16 位单片机以及基于此的开发板市场具有广阔前景，是值得重视的。

随着工艺提升、成本下降，8/16 为单片机的发展趋势是更高的频率、更大的 Flash 与 RAM，以及更多的外围设备。开发者们的关注点从性能与成本转移到了使用体验上，因而产生了适度的、可接受的浪费。比如，官方的 Arduino Uno 开发板使用一颗 Atmega16U2 单片机进行 USB 与串口的转换[1]，增加了不少成本，但开发板的价格仍能被用户接受；Arduino 库中的 `digitalWrite` 等函数比直接操作寄存器慢了十几倍[6]，但大量用户仍在使用，并且很少受到性能上的影响。对于开发者与业余爱好者而言，完善的开发环境比设备的性能与成本更加重要。

但是，现有的开发环境仍不是完美的。无论何种开发板，板载资源总是有限的，为了加入更多功能，开发板会留出足够多的接口，供用户通过杜邦线连接面包板或其他模块。在笔者使用 Arduino 的过程中，总是发现杜邦线连接比较繁琐，不够美观，并且当连接的器件或模块的数量多了以后会显得杂乱，同时在程序设计中也难以协调

好各个模块之间的关系，既增加了难度，也降低了效率。其他用户也有类似的感受[4][8]。

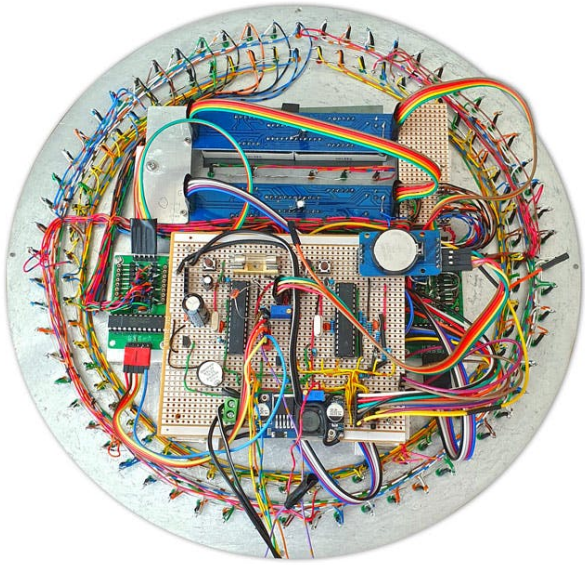


图 2 复杂的杜邦线（来自 Arduino 官网）

针对这个问题，一个可行的解决方案是将资源完全模块化。所谓完全模块化，包含硬件与软件两个方面：硬件上，模块之间直接或使用较少的线连接；软件上，将模块抽象为对象进行操作，运用回调机制处理输入事件。参考文献[18]和[17]中的设计分别实现了硬件与软件的模块化，而 M5Stack 则是一款已进入市场的完全模块化的开发系统。它支持使用 Arduino IDE 开发，也可以使用自主研发的图形化开发工具 UIFlow，后者将图形转化为 MicroPython 代码[3]。然而，这一系列产品基于 32 位单片机，价格昂贵，并且仅有用 MicroPython 才能使用回调机制，在软件模块化的方面与主流的 Arduino 等开发系统不兼容。

因此，8/16 位单片机市场存在一个空缺，需要一种与现有产品兼容的、完全模块化的系统来填补。

第二章 需求与方案

本文的目的是设计一个完全模块化的单片机开发系统的原型，以改善以 Arduino 为代表的 8/16 位单片机开发系统在项目复杂时的使用体验。笔者将这一系统称为 MEDS (Modular Electronics Development System)。

2.1 硬件

模块化要求硬件与软件相结合。硬件上，用户编程的核心模块需要连接到各个有具体功能的应用模块上。连接方式有两类，分别是应用模块连接到核心模块的不同端口上，或相同端口上。对于前者，核心模块上的端口数量即决定了可以连接的应用模块数量，由于单片机 IO 引脚数量固定，这种连接方式显得有些局促。而后一种方式是每一个模块都有一颗单片机，它们之间通过总线连接，可连接的模块数量取决于选用的是何种总线。在笔者熟悉的 AVR 单片机上，有 3 种总线可供选用：SPI、I^2C 与 USART。

SPI (Serial Peripheral Interface)总线是一种高速全双工同步串行总线，在单向传输时最少只需 3 根信号线，但是如果要多设备通信，每多加一个设备，就需要多一根片选信号线，这又限制了模块数量，因此不可选用。

USART (Universal Synchronous and Asynchronous Receiver and Transmitter)总线是一种全双工同步或异步串行总线，常用的是异步通信，2 根信号线即可支持全双工传输，但是总线传输速率较低，而且很多单片机提供的 USART 或 UART 组件不支持多设备通信，也不可选用。

I^2C (Inter-Integrated Circuit)总线是一种半双工同步串行总线，只需 2 根信号线即可通信，支持 7 位或 10 位地址，允许多主机并可以自动仲裁，每一字节传输后都有应答。但它的缺点在于每个设备都通过开集或开漏输出连接到总线上，稳定性有所欠缺，线与逻辑使总线不允许短路到地的硬件故障。然而，在前两种总线不可用的前提下，这是 MEDS 的唯一选择，同时在设计上是可以避免这些问题的。大多数单片机都支持 400kbps 的全速模式，这个速率是可以接受的。根据 I^2C 规格，7 位地址下总线上最多可以接 112 个设备[5]，对于开发系统来说也足够了。

在决定了使用 I^2C 总线连接模块之后，另一个问题是模块之间通过什么介质来连接。杜邦线或其他连接线是一种方案，但与 MEDS 设计的目的相矛盾，因此选用直接连接的方式。每个模块都是圆角矩形，长度和宽度都是一个最小值的整数倍，模块之间通过 4 针的排针和排母连接。这样的设计使得模块可以直接拼插连接。

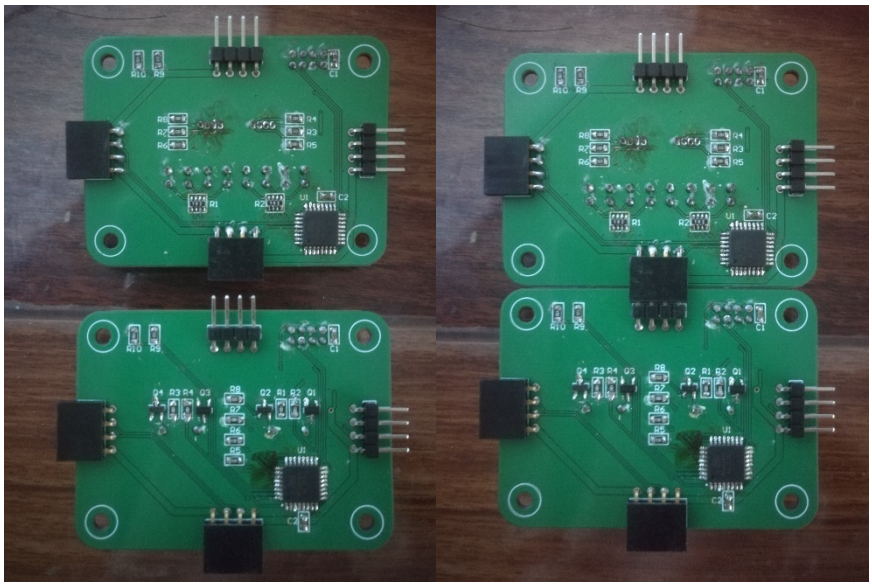


图 3 模块直接拼插

2.2 软件

单片机市场规模已相当大，每个用户都有成千上万种选择，在这样的环境下，一个从零开始的系统是很难融入市场的，仅凭一人或一个小团队的力量，无法提供足够数量的模块，因此开源是必然的选择。唯有开源，才能让广大开发者协同参与 MEDS 的开发与扩展。

然而，各个开发者擅长的单片机型号是不同的。如果系统中用到了一种单片机中的某一个功能，而这个功能在另一种单片机中不可用，那么后者就不能接入系统。为了充分利用开发者的资源，MEDS 中使用的功能应该是各种常见单片机功能集合的交集。同时，对于不同的单片机型号，应该用统一的 API 把底层硬件操作封装起来，给系统以跨平台支持。

8/16 位单片机平台可用的编程语言有汇编、C 和 C++。汇编是无法跨平台的。C 是面向过程的语言，在单片机开发中功能已经足够，但模块化在软件方面的体现在于抽象，这方面 C 有所不足，而 C++是更加合适的选择。具体来讲，模块化表现为将实体模块抽象为面向对象软件中的对象，将模块上的输入输出抽象为对象的消息，运用事件驱动机制进行系统与用户之间的交互。

对于核心模块而言，事件驱动机制的实现可以通过核心模块向应用模块轮询，或应用模块通知核心模块的方式实现。由于 I^2C 总线速率有限，轮询需要占用大量总线资源，并且响应时间取决于轮询速率，而应用模块主动通知占用总线资源少，并能即时响应，是一种更好的选择。

为了实现软件模块化，回调函数不能硬编码在事件处理函数中，需要引入一层间接，即回调函数接口。函数指针是一种常见的实现，但接口单一、固定，不方便对对象进行操作，而 C++11 提供的 lambda 表达式，允许以值或引用的方式捕获变量，标准库提供的 `std::function` 可以简化回调函数的写法，是一种更优雅的实现，同时也免去了给函数命名的麻烦。

模块化要求组件之间的耦合尽量降低，每个回调函数应该只负责一个对象的操作，如果有多个对象需要注册同一个事件，应该引入多播以解耦。C#中的 `delegate` 是一个典型的例子，每一个 `delegate` 实例都可以包装多个方法，在调用时每个方法都会被调用。这些方法可以是函数也可以是对象与方法的组合，并可以动态添加与删除[7]。动态删除建立在委托实例可判定相等的基础上，然而 C++的 `std::function` 不支持 `operator==`，boost 库的文档也明确指出，因为不能实现好，所以不予实现[2]。

然而，要实现模块化的软件系统，回调与多播都是必需的工具。让 8/16 位单片机支持 C#编程显然是凭笔者一己之力无法完成的事，而扩展 C++标准库中的 `std::function`，写一个适合模块化单片机开发系统的类模板作为回调接口是一个可行的方法。

2.3 小结

至此，MEDS 的需求分析已完整。要实现 MEDS 的原型，有许多工作要完成。第三章将介绍标准库扩展的接口与实现中的一些技巧；第四章将描述 MEDS 硬件规格与总线等设备的驱动；第五章将介绍 MEDS 框架的模块化、层次化设计；第六章想象 MEDS 的应用场景与优势，并用项目实例验证 MEDS 系统的可行性。

第三章 标准库扩展的接口与实现

std::function 因不支持 operator==而无法用作多播委托的函数接口。同时，它只能绑定一元对象，不能是对象与成员函数的组合，这限制了它的应用范围。在标准库中，std::bind 解决了对象与成员函数组合的问题，同时还能绑定参数，然而 std::bind 的实现比较复杂，难以添加对 operator==的支持，因此，对对象与成员函数的组合的支持应该直接添加在 std::function 的扩展，即 meds::function 中。在此基础之上，meds::event 用于包装多个 meds::function 实例，并实现动态删减与多播的功能。

虽然 meds::function 和 meds::event 是为模块化单片机开发系统而写，但它们在桌面环境也可以应用，因此这个设计应该具有跨平台的特性。附录 1 和 2 分别为 meds::function 和 meds::event 的源代码。

3.1 meds::function 的接口

meds::function 是一个类模板，其模板参数为函数类型，包括返回值与参数：

```
template <typename>
class function;
template <typename R, typename... Args>
class function<R(Args...)>;
```

其成员方法与辅助函数如下：

```
function() noexcept = default;
function(std::nullptr_t) noexcept;
function(const function& _other);
function(function&& _other) noexcept;
function& operator=(std::nullptr_t) noexcept;
function& operator=(function _other);
~function() noexcept;
void swap(function&) noexcept;

template <typename F>
function(F _functor);
template <typename F>
function& operator=(F _functor);

template <typename T, typename M>
function(T _obj, M _mem);
template <typename T, typename M>
void assign(T _obj, M _mem);
```

```
19.
20.  explicit operator bool() const noexcept;
21.
22.  R operator()(Args...) const;
23.
24.  template <typename T>
25.  T* target() noexcept;
26.  template <typename F>
27.  const F* target() const noexcept;
28.
29.  template <typename T, typename M>
30.  std::pair<T*, M*> target() noexcept;
31.  template <typename T, typename M>
32.  std::pair<const T*, const M*> target() const noexcept;
33.
34.  template <typename R, typename... Args>
35.  bool operator==(const function<R(Args...)>& _lhs,
36.                  const function<R(Args...)>& _rhs) noexcept;
37.  template <typename R, typename... Args>
38.  bool operator!=(const function<R(Args...)>& _lhs,
39.                  const function<R(Args...)>& _rhs) noexcept;
40.
41.  template <typename R, typename... Args>
42.  void swap(function<R(Args...)>& _lhs, function<R(Args...)>& _rhs) noexcept;
```

`meds::function` 的接口兼容大部分与 C++11 中 `std::function` 的接口[11][13]，除了：

- 不支持单独的成员函数作为函数对象，这是因为 `meds::function` 支持对象加成员函数指针的模式，不再需要这个方法；
- 构造函数与 `assign` 中不支持 allocator；
- 不支持 `target_type` 方法，这是因为 `meds::function` 可能包装两个对象。

相比 `std::function`，`meds::function` 有两个额外的特性：

- 二元构造函数与 `assign` 方法，接受一个对象或 `std::reference_wrapper` 实例与一个成员函数作为参数；
- `operator==`用于相等性判定。

其中，`operator==`判定的方法是：

1. 如果两个 `meds::function` 实例的模式不同，返回 `false`；
2. 如果包装的函数对象类型不同，返回 `false`；

3. 如果函数对象是 std::reference_wrapper 实例，则返回其包装的指针是否相等；
4. 如果函数对象属于其他不支持 operator==的类型，则返回 false；
5. 如果函数对象的类型支持 operator==，则返回其结果；
6. 如果包装了对象与成员函数指针，还要判定此指针是否相同。

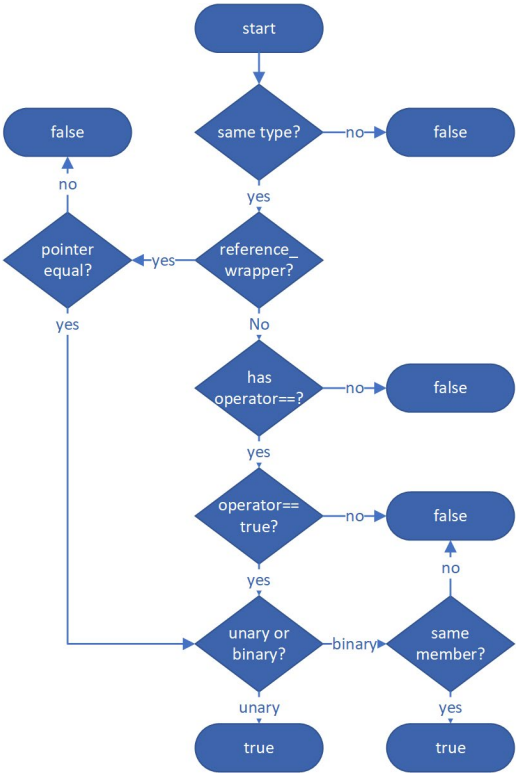


图 4 meds::function::operator==执行流程

3.2 meds::event 的接口

meds::event 也是一个类模板，模板参数与 meds::function 相同。成员方法如下：

```
1.  event();
2.  event(const event& _other);
3.  event(event&& _other);
4.  event& operator=(event _other);
5.  ~event();
6.  void swap(event& _other);
7.
8.  template <typename F>
9.  event(F _functor);
10.
11. bool add(const function_type& _func);
```

```
12.  bool add(function_type&& _func);
13.  event& operator+=(const function_type& _func);
14.  event& operator+=(function_type&& _func);
15.  template <typename... A>
16.  bool emplace(A&&... _args);
17.
18.  template <typename... A>
19.  bool remove(A&&... _args);
20.  event& operator-=(const function_type& _func);
21.
22.  bool empty() const;
23.  bool empty_removing(const function_type& _func) const;
24.
25.  void clear();
26.
27.  void operator()(Args... _args) const;
28.
29.  template <typename F>
30.  void for_each(F&& _functor) const;
```

meds::event 的接口与 C#中的 delegate 类似，通过 add 和 remove 方法添加或删除一个 meds::function 类型的委托，并重载了 operator+=与 operator-=，使添加与删除的操作更加形象。operator()多播调用 meds::event 实例保存的委托实例，for_each 方法允许客户遍历委托链。

meds::event 的构造函数接受一个函数对象，用于管理委托实例的添加与删除。当客户添加或删除一个委托时，该函数对象会被调用，这样的设计允许用户在委托实例变化时执行一些相应的操作，比如当客户向一个空 meds::event 添加一个委托时可以向事件源发送通知。

3.3 函数指针的使用

当 meds::function 实例把一个函数对象包装起来以后，它就不知道这个对象的类型了，因此在 meds::function 实例拷贝与析构的时候，不能直接调用函数对象的相应操作，需要引入虚函数或函数指针。由于后面要提到的存储问题，虚函数在这个实现中不可用，只能使用函数指针来包装这些与类型相关的操作。

与函数对象类型相关的方法有两类，一类是拷贝、析构、相等性判定等无需其他参数的，另一类是函数对象调用，是需要其他参数的。由此，可以将这些方法包装进两个模板函数。下面的代码片段展示了包装函数对象拷贝等操作的函数。

```
1.   enum class function_manager_operation
```

```
2.  {
3.      target, clone, destroy, compare
4.  };
5.  template <typename F>
6.  bool function_manager::unary_manager<F>::
7.  manager(function_data_wrapper* _dst, const function_data_wrapper* _src,
8.      function_manager_operation _op)
9.  {
10.     switch (_op)
11.     {
12.     case function_manager_operation::target:
13.     {
14.         _dst->reinterpret_as<const F*>() = reinterpret_cast<const F*>(
15.             object_manager<function_data_wrapper, F>::target(_src));
16.         break;
17.     }
18.     case function_manager_operation::clone:
19.         object_manager<function_data_wrapper, F>::clone(_dst, _src);
20.         break;
21.     case function_manager_operation::destroy:
22.         object_manager<function_data_wrapper, F>::destroy(_dst);
23.         break;
24.     case function_manager_operation::compare:
25.         return function_object_compare(*reinterpret_cast<const F*>
26.             (object_manager<function_data_wrapper, F>::target(_dst)),
27.                                         *reinterpret_cast<const F*>
28.             (object_manager<function_data_wrapper, F>::target(_src)));
29.     }
30.     return false;
31. }
```

这个函数有 3 个参数，分别为一个数据域指针、一个常量数据域指针与一个指示操作类型的枚举变量。函数中对枚举变量的可能值用 swtich-case 语句枚举，并分别调用相应的操作。

类模板 unary_manager 用于解决函数对象的情况，对象加上成员函数指针的情况由 binary_manager 处理，两个类的对外接口是完全一致的，使 meds::function 可以统一地处理两种情况。

meds::function 实例在初始化或赋值时保存函数指针，在后续的操作中通过指针调用这些函数，将对函数的调用组合起来，成为 meds::function 的方法：

```
1.  template <typename R, typename... Args>
2.  template <typename F>
```

```
function<R(Args...)>::function(F _functor)
{
    if (detail::function_manager::valid_object(_functor))
    {
        detail::function_manager::unary_manager<F>::initialize
            (&data_, std::move(_functor));
        manager_ = &detail::function_manager::
            unary_manager<F>::manager;
        handler_ = &detail::function_handler::
            unary_handler<F, R, Args...>::invoke;
    }
}

template <typename R, typename... Args>
function<R(Args...)>::function(const function& _other)
    : manager_(_other.manager_), handler_(_other.handler_)
{
    if (manager_)
        manager_(&data_, &_other.data_,
            detail::function_manager_operation::clone);
}
```

3.4 Tag Dispatching

meds::function 包装函数对象可以分为两大类情况：函数对象保存在 meds::function 实例内，以及函数对象保存在 heap 上，而 meds::function 实例中保存指针。这种设计使得小函数对象不需要在 heap 上开辟空间，提升了性能。对于一个具体的函数对象，meds::function 以何种方式保存它只取决于该对象的类型特征。

```
using local_storage = std::integral_constant<bool,
       std::is_trivially_copy_constructible<T>::value
    && sizeof(T) <= sizeof(S)
    && alignof(S) % alignof(T) == 0
>;

static void clone(S* _dst, const S* _src)
{ clone(_dst, _src, local_storage()); }
static void clone(S* _dst, const S* _src, std:: true_type)
{
    new (reinterpret_cast<T*>(_dst)) T(_src->template reinterpret_as<T>());
}
static void clone(S* _dst, const S* _src, std::false_type)
{
    _dst->template reinterpret_as<T*>() =
        new T(*_src->template reinterpret_as<T*>());
```

```
17.  }
```

在这段代码中，local_storage 是一个由编译器常量产生的 traits 类，对于具体的模板参数，它会成为 std::true_type 或 std::false_type。第一个 clone 函数产生一个 local_storage 类的实例，根据重载决议调用第二或第三个 clone 函数，分别处理函数对象存储在本地和 heap 上的情况。这种技巧称为 tag dispatching[14]。

3.5 SFINAE

SFINAE (Substitution Failure Is Not An Error) 是 C++模板元编程中的一种常见技巧，意为匹配失败不是错误，是指如果模板参数替换失败，此特化仅被忽略而不作为一个编译错误[14]。

前面提到的 operator==的判定法则中，需要检测一个类型是否有 operator==，可以利用 SFINAE 来实现。

```
1.   template <typename T>
2.   class function_equality_comparable
3.   {
4.   private:
5.       using one = int;
6.       struct two
7.       {
8.           one unused[2];
9.       };
10.      template <typename U,
11.          typename = decltype(std::declval<U>() == std::declval<U>())>
12.      static one test(int);
13.      template <typename>
14.      static two test(...);
15.  public:
16.      static constexpr bool value = sizeof(decltype(test<T>(0))) == sizeof(one);
17.  };
```

在这个实现中，类模板内定义了两个类型 one 与 two，并确保它们大小不相等。test 是一个重载模板函数，一个版本参数为 int，返回值为 one 类型，另一个版本接受不定参数，返回值为 two 类型。对于一个具体的 T 类型，在客户获得静态成员 value 的过程中，编译器会检查 test<T>(0)的返回值类型。当 T 重载了 operator==时，第一个重载函数没有错误，根据重载决议法则，第一个重载优于第二个，因此 value 为 true；当 T 没有重载 operator==时，第一个重载函数匹配失败，test<T>(0)决议为第二个版本，value 为 false，由此实现了类型是否有 operator==的判定。

3.6 多播的实现

一个 meds::event 实例需要保存多个 meds::function 实例，需要一种合适的数据结构来存储。meds::function 实例需要在结构的末尾插入，从任意位置删除，不需要随机访问，因此链表是一种可行的方案。事实上，C#中的 delegate 也是用链表来存储委托实例的。在内存空间足够的环境中，当委托实例较多，并且客户频繁地增删委托实例时，用散列表存储可能会有更好的性能。

std::list 是一个可用的工具，但在内存不充足的单片机中，既然只需要前向遍历，就没有必要维护一个双向链表。std::forward_list 不能常数时间内在链表尾端增加一个节点，如果维护一个指向尾端的迭代器，则需要考虑各种边界情况，比较麻烦，在复制过程中也会出现问题。因此，在 meds::event 类中定义嵌套类型作为链表的节点，实例中直接保存两个指针，分别指向链表的头结点和尾节点。

```
1.  struct node
2.  {
3.      template <typename... A>
4.      node(A&&... _args)
5.          : function_(std::forward<A>(_args)...) { }
6.      function_type function_;
7.      node* next_ = nullptr;
8.  };
```

在 add 与 emplace 方法中，一个 meds::function 实例被根据实参构建出来，如果委托链中没有与之相同的实例，并且存在管理器且其返回值为 true 或不存在，则在链表末尾添加一个节点，由该 meds::function 实例移动得到，并改变指向尾结点的指针。

```
1.  template <typename R, typename... Args>
2.  template <typename... A>
3.  bool event<R(Args...)>::emplace(A&&... _args)
4.  {
5.      function_type func(std::forward<A>(_args)...);
6.      for (auto cur = delegate_; cur; cur = cur->next_)
7.          if (cur->function_ == func)
8.              return false;
9.      if (!manager_ || manager_(this, event_modifier_operation::add, &func))
10.     {
11.         auto next = new node(std::move(func));
12.         if (back_)
13.         {
14.             back_->next_ = next;
```

```
            back_ = next;
        }
        else
        {
            delegate_ = next;
            back_ = next;
        }

        return true;
    }
    return false;
}
```

remove 方法先在链表中寻找与参数相等的 meds::function 实例，使用与上述相同的判定方法，从链表中删除这一节点。

```
template <typename R, typename... Args>
template <typename... A>
bool event<R(Args...)>::remove(A&&... _args)
{
    auto func = function_type(std::forward<A>(_args)...);
    auto cur = delegate_;
    node* prev = nullptr;
    for (; cur; prev = cur, cur = cur->next_)
        if (cur->function_ == func)
            break;
    if (!cur)
        return false;
    if (!manager_ || manager_(this, event_modifier_operation::remove, &func))
    {
        if (!cur->next_)
            back_ = prev;
        auto next = cur->next_;
        delete cur;
        if (cur == delegate_)
            delegate_ = next;
        else
            prev->next_ = next;
        return true;
    }
    return false;
}
```

函数调用运算符遍历链表，并依次调用每一个节点中的 meds::function 实例。由于多播情况下委托的返回值没有意义，因此此方法返回 void。

```
1.  template <typename R, typename... Args>
2.  void event<R(Args...)>::operator()(Args... _args) const
3.  {
4.      for (auto cur = delegate_; cur; cur = cur->next_)
5.          cur->function_(_args...);
6.  }
```

meds::event 还提供了 for_each 方法，接受一个函数对象作为参数，遍历链表并将每一个 meds::function 实例给该对象调用，允许客户对每个委托进行自定义操作。比如，对于返回值为整数情况，传入一个适当的 lambda 可以实现累加返回值的功能。

```
7.  template <typename R, typename... Args>
8.  template <typename F>
9.  void event<R(Args...)>::for_each(F&& _functor) const
10. {
11.     auto cur = delegate_;
12.     for (auto cur = delegate_; cur; cur = cur->next_)
13.         _functor(cur->function_);
14. }
```

第四章 硬件接口与软件驱动

模块之间要可以自由拼插，需要模块尺寸与接口有一个统一的规格。尽管在目前的原型阶段模块种类很少，但应该要考虑到未来的情况，有必要对模块进行分类。

作为一个开源的、开放的系统，广大开发者都可以参与 MEDS 的扩展，但是大部分开发者以 C 为主要编程语言，同时还存在一些平台不支持 C++编程。为了使 MEDS 有更广的适用范围，底层 API 均使用 C 语言编写。

4.1 硬件规格

模块的最小长度为 6cm，最小宽度为 4.5cm，所有模块的实际大小都是这个尺寸的整数倍。对于最小尺寸的模块，左边与上边的正中间放置 4 针的 90 度弯排针，右边与下边的正中间放置 4 针的 90 度弯排母。对于更大的模块，需要放置多个排针与排母，位置与最小尺寸的对应。在这样的设计下，模块之间可以自由拼插，不会冲突。

为了安全，模块的四角被设计成半径为 5mm 的圆角，每个模块的四角各设置一个直径 3mm 的孔，其中心距离两边均为 5mm，即在圆角的圆心处。这些孔位可以安装 M3 螺丝，PCB 可由此固定在一块相应规格的亚克力底板上，固定后模块连接更加牢固，也更美观。

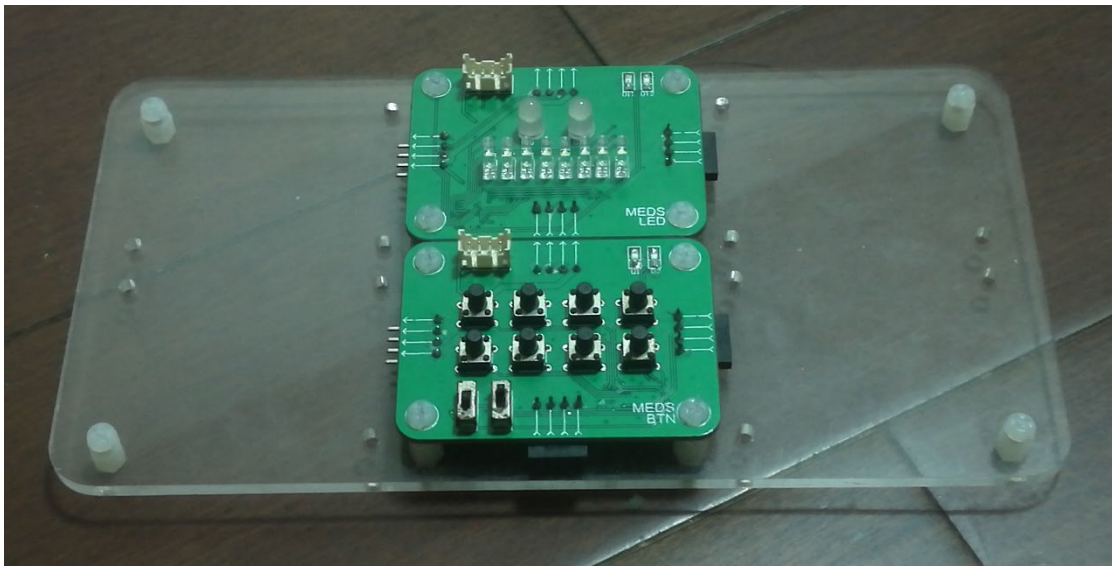


图 5 PCB 固定在亚克力底板上

排针与排母上都有 4 个引脚，从上到下或从左向右分别为 GND（地）、SCL（I^2C 时钟）、SDA（I^2C 数据）与 VCC（+5V 电源）。所有模块都并联在总线上。每个模块上都有一个下载器接口，只对该模块有效，但其中的电源可以给所有连接的模块供电。

4.2 模块分类

从用户的角度，模块可以分为两类：用户可编程的核心模块与可以直接使用的应用模块。用户写代码、编译、烧写的对象都是核心模块，而应用模块只需要接收到指令就能工作，包括回调。值得强调的是，在其他单片机系统，如 Arduino 等开发板上，使用 MEDS 库并连接应用模块，是不需要核心模块就能工作的。或者说，此时开发板充当了核心模块的角色。

在原型阶段，MEDS 暂时只有 5 个模块：

核心模块，使用 ATmega328P 单片机，引出了若干 IO 口，另有两个按键；

LED 模块，板载 8 个白色 LED 与 2 个 RGB LED；

蜂鸣器模块，板载 2 个蜂鸣器，并有 4 个 LED 可用于指示音量；

按键模块，板载 8 个 6mm 按键与 2 个拨动开关；

蓝牙模块，可通过 BLE 4.0 与其他蓝牙设备连接，包括另一个蓝牙模块。

应用模块具有两种属性：输入与输出。这两种属性是相对 MEDS 而言的，如 LED 模块有输出属性而按键模块有输入属性。一些模块同时拥有输入输出两种属性，如蓝牙模块。

在这种分类方法下，所有模块可以如下表所示分为 4 类：

表 1 模块分类

类型	核心模块	应用模块		
		输出模块	输入模块	输入输出模块
示例	核心模块	LED 模块 蜂鸣器模块	按键模块	蓝牙模块

4.3 总线操作

在与 I^2C 总线操作相关的头文件 i2c.h 中，有以下声明式：

```
1.  typedef struct
2.  {
3.      uint8_t size;
4.      uint8_t* data;
5.  } i2c_data_t;
6.  typedef void (*i2c_callback_t)(i2c_data_t*);
```

```
7.
8.    void i2c_init(uint8_t _address, i2c_callback_t _func);
9.    void i2c_interrupt();
10.   void i2c_address(uint8_t _address);
11.   void i2c_send(uint8_t _address, uint8_t _length, const uint8_t* _data);
12.   uint8_t i2c_receive(uint8_t _address, uint8_t _length, uint8_t* _data);
```

两个 typedef 定义了两个类型：i2c_data_t 是一种结构体，包含一个无符号 8 位整数表示大小，一个指针指向数据；i2c_callback_t 是函数指针类型，参数为 i2c_data_t 指针，返回 void。

i2c_init 用于初始化 I²C 总线相关资源，有两个参数：_address 为无符号 8 位整型，表示 I²C 地址，最低位必须为 0；_func 为回调函数，要处理总线从机事件。当其参数 i2c_data_t 指针所指的 size 为 0 时，表示这是一个主收从发操作，此时回调函数应该把要发送的信息写到 data 所指的空间中去，并修改 size 为数据大小；当 size 非 0 时，表示这是一个主发从收操作，此时回调函数应该从 data 读取 size 个字节，并处理指令。回调函数在中断中被调用，如果全局中断开启，回调函数随时可能被调用，是被动的。

i2c_interrupt 是 I²C 中断的入口。如果中断服务程序被编译在静态库文件中，当链接器执行链接并生成可执行程序时，由于中断服务程序没有被主函数调用过，链接器会忽略它，导致可执行程序中没有中断服务程序，此时触发的中断将导致错误。因此，接口只提供了中断的入口，需要应用程序或高层库编写中断服务程序，其中直接调用 i2c_interrupt 函数。

i2c_address 允许程序在初始化完成后改变本机 I²C 总线地址。

i2c_send 用于向指定设备发送一定长度的数据，有三个参数：address 为目标设备的 I²C 地址；_length 为发送数据的长度；_data 为指向待发送数据的指针。总线发送是非阻塞的，待发送的数据先被复制到缓冲区中，在中断中逐字节发送。

i2c_receive 用于从指定设备读取一定长度的数据，有三个参数：_address 为目标设备的 I²C 地址；_length 为读取数据的长度；_data 为指向保存所读取数据空间的指针。总线读取是阻塞的，必须等缓冲区清空后、读取完后才能返回，返回值为实际读到的数据长度。

以上为总线操作的接口，至于总线操作的实现细节，如缓冲区大小、是否使用 DMA 等，由实现决定，与接口无关。对于不同的单片机平台，总线操作的实现是不同的，但接口是统一的，使得 MEDS 可以运行在不同的平台上。

4.4 其他组件

除了必需的总线操作以外，还有一些硬件资源是单片机开发中很常用的，比如定时器、串口、ADC 等。驱动层提供了这些硬件的基本操作，功能较少，比如串口只提供了初始化与发送字符等的功能，至于格式化输出与字符串解析等功能，应该由上层库来完成。但重要的是，驱动层提供了一个平台无关的接口，使构建基于这些硬件操作的程序相对容易。

第五章 MEDS 框架的设计

MEDS 是由多个模块组成的系统，模块之间的通信，即模块上单片机之间的通信，有一定的格式。这需要对驱动层再进行一次封装。

核心模块控制所有应用模块，后者在前者的程序中抽象为对象。应用模块有功能相同、方法不同的操作，面向对象程序设计中类与继承的概念与工具被用来处理这些问题。

5.1 通用驱动层

上一章中介绍的与硬件相关的库即为 MEDS 的通用驱动层，核心模块与应用模块的程序都建立在这一层之上。这一层将不同单片机平台的硬件操作统一成 API，提供了 MEDS 运行所必需的功能，使其拥有跨平台的特性。

将核心模块与应用模块更多的共用部分提取出来加入通用层可以减小工作量，但是后文中将会介绍，核心模块中直接建立在驱动层之上的库因为涉及到模块抽象只能使用 C++编写，而应用模块的程序不应该出现 C++，因此通用的部分到硬件操作为止，更高层的封装由两类模块分别完成。

5.2 应用模块管理层

应用模块有 3 项工作：接收核心模块从总线上发来的数据；控制板载资源；在适当的时候通过总线向核心模块发送数据。总线相关的操作由管理层来包装，板载资源由可执行程序直接控制。总线上共有 3 种操作：核心模块向应用模块发送（MT）、核心模块从应用模块读取（MR）、应用模块向核心模块发送（SR），每一种的数据都具有一定的格式，由模块的开发文档详细描述，并硬编码在双方的程序中。

Instruction:

byte 0	description
0x00~0x0F	reserved
0x10~0x11	turn red LED off/on
0x14~0x15	turn green LED off/on
0x20~0x21	disable/enable buttons released event
0x22~0x23	disable/enable buttons pressed event
0x30~0x37	disable button 0~7 released event
0x38~0x3F	enable button 0~7 released event
0x40~0x47	disable button 0~7 pressed event
0x48~0x4F	enable button 0~7 pressed event
0x50~0x51	disable switch 0~1 changed event
0x58~0x59	enable switch 0~1 changed event

Requested:

index	range	description
byte 0	0x00~0xFF	button 0~7 pressed
byte 1	0x00~0x03	switch 0~1 status

Callback:

byte 0	description
0x00~0x0F	reserved
0x10~0x17	button 0~7 released
0x18~0x1F	button 0~7 pressed
0x20~0x21	switch 0~1 turned off
0x28~0x29	switch 0~1 turned on

图 6 按键模块指令

MT 数据的第一字节表示要执行的操作，随后可能有若干字节的数据，由第一个字节决定。这些数据称为指令。指令的第一个字节可由模块使用的范围为 `0x10` 到 `0xFF`，`0x00` 至 `0x0F` 为保留指令，可用于设置 I^2C 地址等操作，由管理层直接处理。

MR 数据的格式由模块自行定义。如果应用模块有多种可发送的数据，需要核心模块在读取前加一个发送的操作，以表示要读取的数据类型。

SR 数据的第一字节为应用模块在 I^2C 总线上的地址，随后的数据格式由模块自行定义。

《设计模式》在关于 Observer Pattern 的讨论中提出了推/拉模型的概念[15]，即回调数据中包含的信息量大小。在 MEDS 中，由于单片机处理的数据量一般比较小，一般使用推模型，即 SR 中传输全部信息，无需再进行 MR 操作。然而，对于蓝牙等数据量可能较大的模块来说，拉模型也是可以使用的，具体的阈值、格式等，由模块自行定义。

管理层头文件 app.h 声明的接口如下：

```
1.  void app_init(uint8_t _address);
2.  extern void receive(uint8_t _size, uint8_t* _data);
3.  extern uint8_t request(uint8_t* _data);
4.  void send(uint8_t _size);
5.  extern uint8_t* buffer;
```

`app_init` 用于初始化，包括 I^2C 初始化与回调函数注册。

`receive` 和 `request` 为需要应用程序实现的接口，分别处理 MT 与 MR 操作中的数据。`_data` 为数据，`_size` 和 `request` 的返回值为大小。

`send` 函数在总线上发送数据，`_size` 为大小，数据应存储在 `buffer` 中，从下标为 `0` 的位置开始存放数据，无需考虑地址。

利用管理层与定时器等硬件资源的 API，可以快捷地编写出应用模块程序。

5.3 核心模块抽象层

模块化在软件方面的体现是抽象。下图为核心模块库的 UML 类图，附录 3 以代码形式给出了这些类的完整定义。

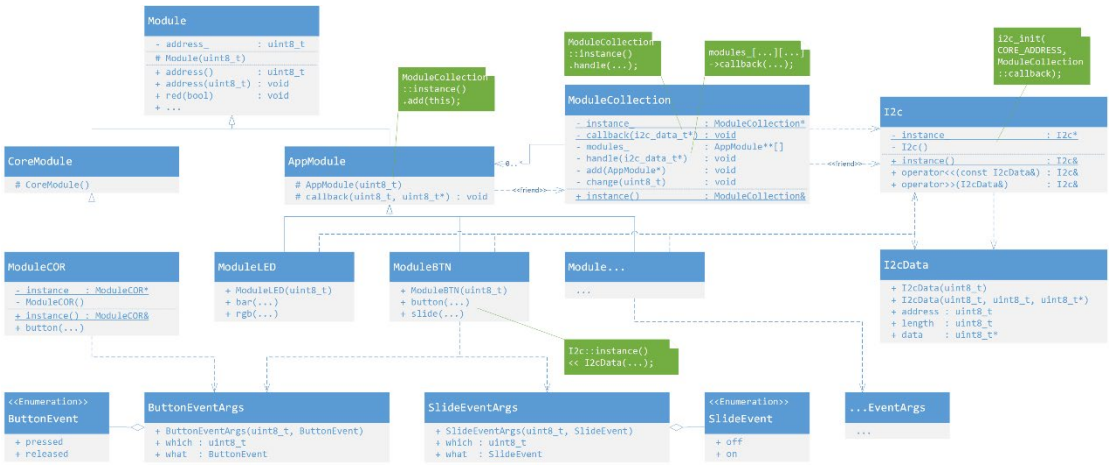


图 7 模块继承体系

Module 为所有模块的基类，提供操作指示灯与改变 I^2C 地址的虚函数。CoreModule 类继承自 Module 类，表示核心模块本身；AppModule 类也继承自 Module 类，表示核心模块程序中抽象出的应用模块，定义了虚函数 callback，表示回调。以上 3 个类都是抽象类。

ModuleCollection 类的实例 modules 维护 AppModule 实例的引用，定义了 handle 方法，根据数据选择模块调用 callback。相应地，AppModule 类的构造函数中将 this 指针注册给 modules。I2c 类的实例 i2c 管理总线操作，构造函数中用静态方法 callback 注册回调，其中通过 modules 调用 handle 方法。

由于 C++标准规定不同编译单元中非局部静态变量的初始化顺序是未定义的[11]，因此这些全局变量都使用单例模式创建，以保证被引用的变量在使用之前被初始化[16]。

每种应用模块的抽象都是继承自 AppModule 的类，命名格式为 ModuleXXX，其中 XXX 为模块的三位字母代号。子类覆写 AppModule 中的 callback 方法，带有输入属性的模块在方法中处理回调信息，其他模块将方法留空，或显式调用 AppModule 的该方法。

对于每一种输入组件，定义一个事件参数类，名称为 YyyEventArgs，其中 Yyy 为组件的名称。一些模块使用了相同的组件，如核心、按键、蓝牙模块上都有按键。为了避免重复定义，将按键操作抽离出来，独立于应用模块的抽象，定义在单独的头文件中，并在应用模块的头文件中引用。

回调函数使用 meds::event 作为接口，模板参数为 void(std::reference_wrapper<ModuleXXX>, YyyEventArgs)，接受 meds::function 委托，用 lambda 表达式可写作[](ModuleXXX&, YyyEventArgs) {...}，与 C#中的事

件参数 `object sender, EventArgs e` 类似。如果不关心参数，在 C++14 标准下可以写成`[](auto, auto) {...}`，更加简洁。

核心模块的抽象继承自 `CoreModule` 类，目前唯一的核心模块名称为 `ModuleCOR`，使用单例模式创建了一个名为 `core` 的实例。`ModuleCOR` 还包含了操作板载资源的方法，包括按键、GPIO、ADC、串口等，以及对应的回调接口。

5.4 小结

第三章至第五章分别介绍了 MEDS 的各个部分。`meds::function` 与 `meds::event` 类模板、可拼插的硬件设计、统一的硬件 API、面向对象的模块抽象，共同构建出一个模块化的、事件驱动的单片机开发系统。第六章将描述 MEDS 可能的应用场景，并用一个应用实例来验证这一设计的可行性。

第六章 应用场景与应用实例

目前，MEDS 仍处于原型阶段，这意味着它的用途非常有限。但是，设计中的诸多细节使它可以利用广大开发者的资源，成长为一个日益完善的系统。在系统发展成熟后，它可以应用在许多场合。

6.1 应用场景

MEDS 是为了解决现有开发板的问题而设计的，它的使用人群之一就是这些开发板的用户。模块化与事件驱动的设计将用户从杂乱的杜邦线与复杂的程序流程中解脱出来，使硬件与软件的结构都更加清晰，不仅减少了工作量，更能将用户的注意力从实现方法转移到功能上来。作为一个开源系统，用户除了使用它以外，还可以阅读源代码并从中学习。并且只要遵守开源协议，开发者也可以将 MEDS 的源码用于其他用途。

应用模块的程序是用 C 语言编写的，在跨平台驱动的支持下，模块的设计者只需关注总线上传输的指令，这使开发一个新的模块变得简单，开发者们可以轻松地加入到 MEDS 的扩展行动中。一个人设计的模块，只要有文档支持，就能给所有人使用。与提供平台相关的示例代码的市售模块不同的是，在 MEDS 架构中设计的模块，可以无需移植，直接接入另一处基于 MEDS 的平台。

低耦合、可扩展性等是软件工程中强调的，具有这些特点的 MEDS 也可以在工程上应用。由于每个模块上都有单片机，增加了模块的成本，MEDS 在工程上适用于对成本要求不严格或用量较少的场合。在产品更新迭代频繁的今天，MEDS 事件驱动的架构与丰富的模块内置功能可以减少开发者的工作量，加快原型设计的进度。同时，MEDS 也保持了适度的兼容，模块上有引出的 IO 端口可供设计者自由使用。工业控制系统也可以将 MEDS 的模块作为系统的外围设备，工程师可以选用自己熟悉的单片机来使用 MEDS。如果可靠性要求十分严格，工程师也可以在开源系统的基础上进行修改。

6.2 应用实例

接下来笔者将用 MEDS 完成一个简单的项目，以验证其设计的可行性。

这个项目使用了 4 个模块：核心模块、LED 模块、蜂鸣器模块与按键模块。核心模块上，一个绿色 LED 作电源指示；面包板上一个光敏电阻与普通电阻串联，分压点接在一个 ADC 端口上；LED 模块上的两个全彩 LED 随机亮起；按键模块上一个拨动

开关作为总开关；8 个按键对应 8 个音符，当按键按下时，LED 模块上一个灯亮起，同时蜂鸣器模块播放对应的音符；在同一时间最多有两个音符同时响起；当环境光较暗时，所有灯的亮度降低，蜂鸣器音量降低。下图为成品外观。

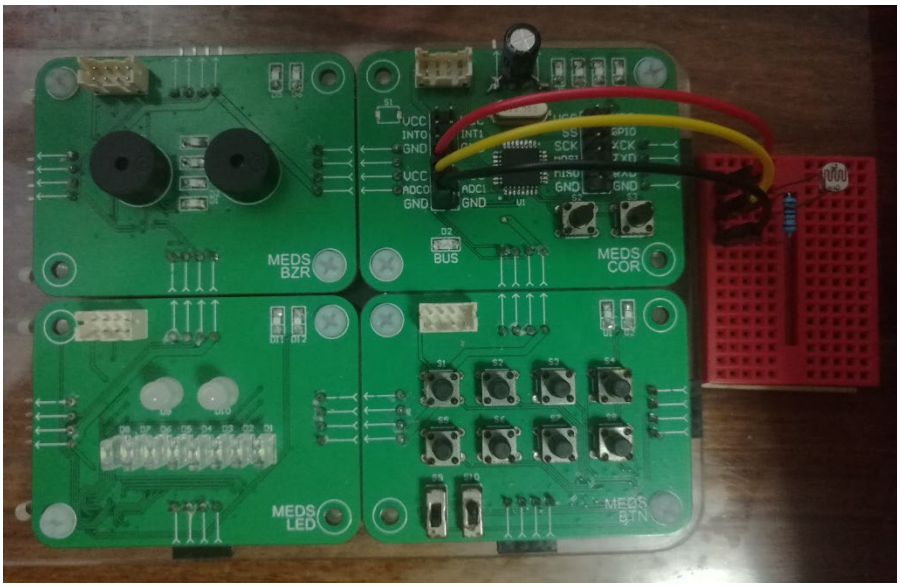


图 8 用 MEDS 搭建的项目

完整的代码见附录 4，这里摘取部分讲解。

应用模块在程序中抽象为全局变量，通过类静态方法创建：

```
auto& led = meds::ModuleLED::abstract(0x20);
auto& bzr = meds::ModuleBZR::abstract(0x30);
auto& btn = meds::ModuleBTN::abstract(0x40);
```

`main` 函数中用 lambda 注册了两个事件，分别根据事件参数执行不同操作，拨动开关的事件掌管其他事件的注册状态：

```
btn.Slide += [](meds::ModuleBTN&, meds::SlideEventArgs e) {
    switch (e.what)
    {
    case meds::SlideEvent::off:
        btn.button().enable();
        btn.Button -= onButtonChanged;
        led.bar().set(0);
        led.rgb().off();
        bzr.buzzer().off();
        break;
    case meds::SlideEvent::on:
        btn.Button += onButtonChanged;
        btn.button().disable();
        led.rgb().random();
```

```
15.          break;
16.      }
17.  };
```

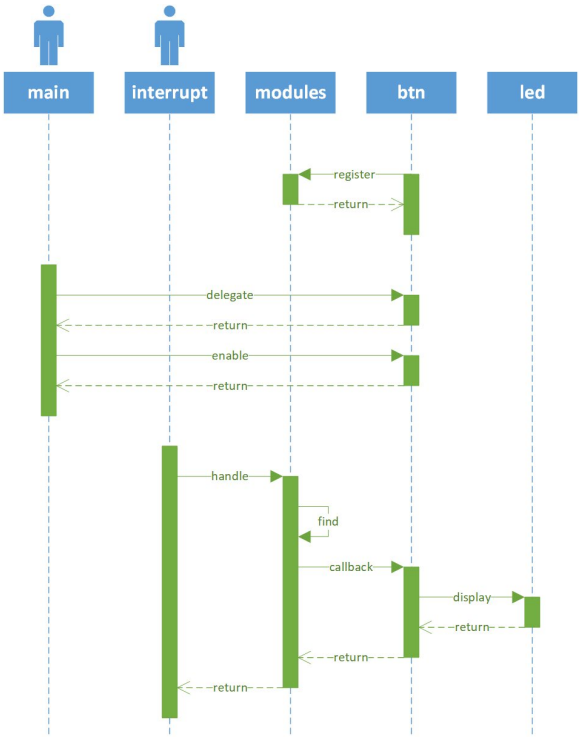
随后，根据当前的设备状态，模拟这两个事件的发生，以建立正确的初始状态：

```
1.   if (btn.slide(1).status())
2.       btn.Slide(btn, meds::SlideEventArgs(1, meds::SlideEvent::on));
3.   core.Adc(meds::AdcEventArgs(0, core.adc(0).status() ?
4.       meds::AdcEvent::higher : meds::AdcEvent::lower));
```

最后，程序启用了事件，并进入死循环：

```
1.   btn.slide(1).enable();
2.   core.adc(0).enable(meds::AdcEvent::window, 128);
3.   while (1)
4.       ;
```

在这个例子中，简化的程序流程如下图所示：



流程主要由 3 段组成：全局变量构造，此过程中模块都向集合注册好回调接口；`main` 函数中的初始化，设置好各种事件的处理方法；硬件中断，总线数据被一步步传到模块抽象中，并执行相应的回调函数。`main` 函数中做的事很少，只需在初始化完成后加一个死循环以保证程序持续运行。如果使用的函数都能保证中断安全，`main` 函数也可以做一些实际的工作。

程序是可以正常工作的，附件中有演示视频。这个应用实例说明，MEDS 框架的设计是可行的。不仅如此，从代码中不难发现，程序结构清晰，所有工作都在初始化阶段完成，后续都由框架自动处理，无需用户干预。

实际上，这个项目中的设备不算多，一个常规的单片机就有足够的 IO 口可以控制，甚至全部放在面包板上也不算复杂。对于更复杂的项目，可以换用更大封装的单片机，使用模块化的系统绝非必要。但是，调用底层操作直接控制设备的程序，其可移植性只能在同一系列单片机上得到保证，而使用 MEDS 不仅可以提供跨平台支持，硬件与软件的模块化还能降低耦合，使程序结构更清晰，扩展更加容易。总之，使用 MEDS 可以带来许多便利。

第七章 总结与展望

本文在单片机开发盛行的背景下，发现了现有产品的问题与市场上的空缺，从表面现象到问题根源、自顶向下地分析需求，又从底层到抽象、自底向上地设计接口并实现，完成了 MEDS 的原型，并分析了它的应用场景，给出一个应用实例供参考。

笔者做的主要工作与贡献有：

1. 用 C++编写了 `meds::function` 和 `meds::event` 两个类模板，前者扩展了标准库中 `std::function` 的功能，后者利用前者实现了多播的功能。两者在 C++编程中通用，可以增加消息传递的灵活性。
2. 提出一种完全模块化的单片机系统的实现方案，在前一条的基础上设计并制作出了原型，验证了方案的可行性。它开源、跨平台、使用方便、应用范围广，可以解决目前存在的问题，填补市场空缺。

然而，即使只是原型，MEDS 也还远不完美：

- 模块指令硬编码在核心模块库与应用模块程序中，不利于修改与扩展；
- 代码中存在许多漏洞，如函数可中断重入的问题，影响系统可靠性；
- 笔者精力有限，模块数量与功能太少，在此基础上难以实现一个完整的项目；
- 目前还没有文档支持，代码中注释也不够详细，开发与使用模块都比较麻烦。

未来，为了完善 MEDS，有以下工作需要完成：

- 修补漏洞，优化实现中的细节，增强对错误情况的处理，提高可靠性；
- 设计并制作更多模块，撰写文档与教程，使系统功能更丰富、完整；
- 把 MEDS 给开发者试用，邀请他们参与系统的扩展，并根据他们的建议完善系统。

参考文献

[1] (n.d.). Arduino Uno Rev3 Schematic. Retrieved from https://www.arduino.cc/en/uploads/Main/Arduino_Uno_Rev3-schematic.pdf

[2] (n.d.). Boost.Function Frequently Asked Questions. Retrieved from https://www.boost.org/doc/libs/1_71_0/doc/html/function/faq.html

[3] (n.d.). M5Flow. Retrieved from http://flow.m5stack.com

[4] (2011, April 8). Your messy breadboard pics for my Arduino cable organizer. Retrieved from https://forum.arduino.cc/index.php?topic=57951.0

[5] (2014, April 4). I^2C-bus specification and user manual. Retrieved from https://www.nxp.com/docs/en/user-guide/UM10204.pdf

[6] (2015, January 4). Why is Arduino digitalWrite So Slow? Retrieved from https://www.peterbeard.co/blog/post/why-is-arduino-digitalwrite-so-slow

[7] (2015, July 20). Delegates - C# Programming Guide. Retrieved from https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates

[8] (2017, November 13). Code gets complex and hard to understand, any C books to recommend? Retrieved from https://forum.arduino.cc/index.php?topic=511329.0

[9] (2018, March 22). Introducing the Top 10 Dev Boards of 2018. Retrieved from https://www.arrow.com/en/research-and-events/videos/the-top-10-development-platforms-dev-kits-2018

[10] (2019, May). Microcontroller Market Size & Share: Global Industry Report, 2019-2025. Retrieved from https://www.grandviewresearch.com/industry-analysis/microcontroller-market

[11] ISO/IEC. (2012, January 16). ISO International Standard ISO/IEC 14882:2014(E) – Programming Language C++. [Working draft]. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf

[12] Statista Research Department. (2016, April 19). Global microcontroller market size, by type 2019. Retrieved from https://www.statista.com/statistics/678610/worldwide-microcontroller-unit-market-size-by-segment

[13] Stroustrup, B. (2013). The C++ programming language, fourth edition. Boston: Addison-Wesley.

[14] Vandevoorde, D. M., Josuttis, N. M., & Gregor, D. M. (2018). C++ Templates: The Complete Guide (2nd Edition). Boston: Addison-Wesley.

[15] [美]Erich Gamma 等著.李英军等译.设计模式：可复用面向对象软件的基础[M].机械工业出版社,2000.

[16] [美]Scott Meyers 著.侯捷译.Effective C++：改善程序与设计的 55 个具体做法（第三版）[M].电子工业出版社,2011.

[17] 周富相,陈德毅,郑晓晶.基于事件驱动的单片机多任务程序设计[J].计算机系统应用,2012,21(07):196-199.

[18] 邹应全,刘建成.模块化单片机实验系统的设计与实现[J].南京信息工程大学学报(自然科学版),2010,2(02):185-189.

附录

可运行代码见附件。

1. meds::function

function.h

```
#include "pch.h"

#ifndef MEDS_FUNCTION_H
#define MEDS_FUNCTION_H

#include <exception>
#include <type_traits>
#include <utility>
#include <memory>

#include "../refwrap/refwrap.h"

namespace meds
{

template <typename>
class function;

template <typename R, typename... Args>
class function<R(Args...)>;

namespace detail
{

template <typename... Args>
struct function_argument_type { };

template <typename Arg>
struct function_argument_type<Arg>
{
    using argument_type = Arg;
};

template <typename Fst, typename Snd>
struct function_argument_type<Fst, Snd>
{
    using first_argument_type  = Fst;
```

```
38.      using second_argument_type = Snd;
39.  };
40.
41.  class function_data_field
42.  {
43.  public:
44.      using data_type = void*;
45.
46.      template <typename T>
47.      T& reinterpret_as()
48.      { return *(T*)this; }
49.      template <typename T>
50.      const T& reinterpret_as() const
51.      { return *(const T*)this; }
52.
53.  private:
54.      data_type data_field;
55.  };
56.
57.  class function_data_wrapper
58.  {
59.  public:
60.      using first_type  = function_data_field;
61.      using second_type = function_data_field;
62.
63.      template <typename T>
64.      T& reinterpret_as()
65.      { return *(T*)this; }
66.      template <typename T>
67.      const T& reinterpret_as() const
68.      { return *(const T*)this; }
69.
70.      first_type * get_first_ptr ()
71.      { return std::addressof(first_field); }
72.      second_type* get_second_ptr()
73.      { return std::addressof(second_field); }
74.      const first_type * get_first_ptr () const
75.      { return std::addressof(first_field); }
76.      const second_type* get_second_ptr() const
77.      { return std::addressof(second_field); }
78.
79.      template <typename T>
80.      T& reinterpret_first_as ()
81.      { return *(T*)std::addressof(first_field); }
```

```
82.      template <typename T>
83.      T& reinterpret_second_as()
84.      { return *(T*)std::addressof(second_field); }
85.      template <typename T>
86.      const T& reinterpret_first_as () const
87.      { return *(const T*)std::addressof(first_field); }
88.      template <typename T>
89.      const T& reinterpret_second_as() const
90.      { return *(const T*)std::addressof(second_field); }
91.
92.  private:
93.      first_type  first_field ;
94.      second_type second_field;
95.  };
96.
97.  enum class function_manager_operation
98.  {
99.      target, clone, destroy, compare
100. };
101.
102. template <typename T>
103. class function_equality_comparable
104. {
105. private:
106.     using one = int;
107.     struct two
108.     {
109.         one unused[2];
110.     };
111.
112.     template <typename U,
113.         typename = decltype(std::declval<U>() == std::declval<U>())>
114.     static one test(int);
115.     template <typename>
116.     static two test(...);
117.
118. public:
119.     static constexpr bool value = sizeof(decltype(test<T>(0))) == sizeof(one);
120. };
121.
122. template <typename T>
123. typename std::enable_if<is_reference_wrapper<T>::value,
124.     bool>::type
125.     function_object_compare(const T& _lhs, const T& _rhs)
```

```
126. {
127.     return std::addressof(_lhs.get()) == std::addressof(_rhs.get());
128. }
129. template <typename T>
130. typename std::enable_if<!is_reference_wrapper<T>::value
131.     &&  function_equality_comparable<const T&>::value,
132.     bool>::type
133.     function_object_compare(const T& _lhs, const T& _rhs)
134. {
135.     return _lhs == _rhs;
136. }
137. template <typename T>
138. typename std::enable_if<!is_reference_wrapper<T>::value
139.     && !function_equality_comparable<const T&>::value,
140.     bool>::type
141.     function_object_compare(const T& _lhs, const T& _rhs)
142. {
143.     return false;
144. }
145.
146. struct function_manager
147. {
148.     using manager_type = bool (*)(function_data_wrapper*,
149.         const function_data_wrapper*, function_manager_operation);
150.
151.     template <typename S, typename T>
152.     class object_manager
153.     {
154.     private:
155.         using local_storage = std::integral_constant<bool,
156.                 std::is_trivially_copy_constructible<T>::value
157.             && sizeof(T) <= sizeof(S)
158.             && alignof(S) % alignof(T) == 0
159.         >;
160.
161.     public:
162.         static void initialize(S* _tar, T&& _obj)
163.         { initialize(_tar, std::move(_obj), local_storage()); }
164.
165.         static const void* target(const S* _tar)
166.         { return target(_tar, local_storage()); }
167.
168.         static void clone(S* _dst, const S* _src)
169.         { clone(_dst, _src, local_storage()); }
```

```
170.
171.         static void destroy(S* _tar)
172.         { destroy(_tar, local_storage()); }
173.
174.     private:
175.         static void initialize(S* _tar, T&& _obj, std::true_type );
176.         static void initialize(S* _tar, T&& _obj, std::false_type);
177.
178.         static const void* target(const S* _tar, std::true_type );
179.         static const void* target(const S* _tar, std::false_type);
180.
181.         static void clone(S* _dst, const S* _src, std::true_type );
182.         static void clone(S* _dst, const S* _src, std::false_type);
183.
184.         static void destroy(S* _tar, std::true_type );
185.         static void destroy(S* _tar, std::false_type);
186.     };
187.
188.     template <typename S, typename T>
189.     class object_manager<S, reference_wrapper<T>>
190.         : public object_manager<S, T*>
191.     {
192.     public:
193.         static void initialize(S* _tar, T*&& _obj) = delete;
194.         static void initialize(S* _tar, reference_wrapper<T> _wrap);
195.
196.         static const void* target(const S* _tar);
197.     };
198.
199.     template <typename F>
200.     class unary_manager
201.     {
202.     public:
203.         static void initialize(function_data_wrapper* _tar, F&& _fun);
204.
205.         static bool manager(function_data_wrapper* _dst,
206.             const function_data_wrapper* _src, function_manager_operation _op);
207.     };
208.
209.     template <typename T, typename M>
210.     class binary_manager
211.     {
212.     public:
213.         static void initialize(function_data_wrapper* _tar,
```

```
214.             T&& _obj, M&& _mem);
215.
216.         static bool manager(function_data_wrapper* _dst,
217.             const function_data_wrapper* _src, function_manager_operation _op);
218.     };
219.
220.     template <typename T>
221.     static typename std::enable_if< std::is_convertible<T, bool>::value,
222.         bool>::type
223.         valid_object(const T& _obj)
224.     { return _obj; }
225.     template <typename T>
226.     static typename std::enable_if<!std::is_convertible<T, bool>::value,
227.         bool>::type
228.         valid_object(const T& _obj)
229.     { return true; }
230. };
231.
232. struct function_handler
233. {
234.     template <typename R, typename... Args>
235.     using handler_type = R (*)(const function_data_wrapper&, Args...);
236.
237.     template <typename F, typename R, typename... Args>
238.     class unary_handler
239.     {
240.     public:
241.         static R    invoke(const function_data_wrapper& _ptr, Args... _args);
242.     };
243.     template <typename F, typename... Args>
244.     class unary_handler<F, void, Args...>
245.     {
246.     public:
247.         static void invoke(const function_data_wrapper& _ptr, Args... _args);
248.     };
249.     template <typename F, typename R, typename... Args>
250.     class unary_handler<reference_wrapper<F>, R   , Args...>
251.     {
252.     public:
253.         static R    invoke(const function_data_wrapper& _ptr, Args... _args);
254.     };
255.     template <typename F, typename... Args>
256.     class unary_handler<reference_wrapper<F>, void, Args...>
257.     {
```

```
258.     public:
259.         static void invoke(const function_data_wrapper& _ptr, Args... _args);
260.     };
261.
262.     template <typename T, typename M, typename R, typename... Args>
263.     class binary_handler
264.     {
265.     public:
266.         static R    invoke(const function_data_wrapper& _ptrs, Args... _args);
267.     };
268.     template <typename T, typename M, typename... Args>
269.     class binary_handler<T, M, void, Args...>
270.     {
271.     public:
272.         static void invoke(const function_data_wrapper& _ptrs, Args... _args);
273.     };
274.     template <typename T, typename M, typename R, typename... Args>
275.     class binary_handler<reference_wrapper<T>, M, R   , Args...>
276.     {
277.     public:
278.         static R    invoke(const function_data_wrapper& _ptrs, Args... _args);
279.     };
280.     template <typename T, typename M, typename... Args>
281.     class binary_handler<reference_wrapper<T>, M, void, Args...>
282.     {
283.     public:
284.         static void invoke(const function_data_wrapper& _ptrs, Args... _args);
285.     };
286.
287.     template <typename R, typename... Args>
288.     class function_invoker
289.     {
290.     public:
291.         static inline R invoke(const function<R(Args...)>* _fun,
292.             Args... _args);
293.     };
294.     template <typename... Args>
295.     class function_invoker<void, Args...>
296.     {
297.     public:
298.         static inline void invoke(const function<void(Args...)>* _fun,
299.             Args... _args);
300.     };
301. };
```

```
302.
303. }
304.
305. class bad_function_call
306.     : public std::exception
307. {
308. public:
309.     bad_function_call() = default;
310.     virtual ~bad_function_call() = default;
311.     const char* what() const noexcept override
312.     { return "bad_function_call"; }
313. };
314.
315. template <typename R, typename... Args>
316. class function<R(Args...)>
317.     : public detail::function_argument_type<Args...>
318. {
319. public:
320.     using result_type = R;
321.
322.     function() noexcept = default;
323.     function(std::nullptr_t) noexcept
324.         : function() { };
325.     function(const function& _other);
326.     function(function&& _other) noexcept
327.     { swap(_other); }
328.     template <typename F>
329.     function(F _functor);
330.
331.     template <typename T, typename M>
332.     function(T _obj, M _mem);
333.
334.     ~function();
335.
336.     function& operator=(std::nullptr_t);
337.     function& operator=(function _other)
338.     { swap(_other); }
339.     template <typename F>
340.     function& operator=(F _functor);
341.
342.     template <typename T, typename M>
343.     void assign(T _obj, M _mem);
344.
345.     void swap(function&) noexcept;
```

```
346.
347.     explicit operator bool() const noexcept;
348.
349.     R operator()(Args...) const;
350.
351.     template <typename T>
352.     T* target() noexcept
353.     { return const_cast<T*>(const_cast<const function*>(this)->target<T>()); }
354.     template <typename F>
355.     const F* target() const noexcept;
356.
357.     template <typename T, typename M>
358.     std::pair<T*, M*> target() noexcept
359.     { return *reinterpret_cast<std::pair<T*, M*>*>(
360.         const_cast<const function*>(this)->target<T, M>()); }
361.     template <typename T, typename M>
362.     std::pair<const T*, const M*> target() const noexcept;
363.
364.     template <typename R1, typename... Args1>
365.     friend bool operator==(const function<R1(Args1...)>&,
366.                            const function<R1(Args1...)>&) noexcept;
367.
368. private:
369.     detail::function_data_wrapper data_;
370.     typename detail::function_manager::manager_type manager_ = nullptr;
371.     typename detail::function_handler::handler_type<R, Args...> handler_
372.         = nullptr;
373.
374.     template <typename R1, typename... Args1>
375.     friend class detail::function_handler::function_invoker;
376. };
377.
378. template <typename R, typename... Args>
379. bool operator==(const function<R(Args...)>& _lhs,
380.                 const function<R(Args...)>& _rhs) noexcept;
381. template <typename R, typename... Args>
382. bool operator!=(const function<R(Args...)>& _lhs,
383.                 const function<R(Args...)>& _rhs) noexcept
384. { return !(_lhs == _rhs); }
385.
386. template <typename R, typename... Args>
387. void swap(function<R(Args...)>& _lhs, function<R(Args...)>& _rhs) noexcept
388. { _lhs.swap(_rhs); }
389.
```

```
390. }
391.
392. #include "function_impl.cpp"
393.
394. #endif
```

function_impl.cpp

```
1.   #include "function.h"
2.
3.   #include <new>
4.
5.   namespace meds
6.   {
7.
8.   namespace detail
9.   {
10.
11.  template <typename S, typename T>
12.  void
13.  function_manager::object_manager<S, T>::
14.  initialize(S* _tar, T&& _obj, std::true_type)
15.  {
16.      new (reinterpret_cast<T*>(_tar)) T(std::move(_obj));
17.  }
18.  template <typename S, typename T>
19.  void
20.  function_manager::object_manager<S, T>::
21.  initialize(S* _tar, T&& _obj, std::false_type)
22.  {
23.      _tar->template reinterpret_as<T*>() = new T(std::move(_obj));
24.  }
25.
26.  template <typename S, typename T>
27.  const void*
28.  function_manager::object_manager<S, T>::
29.  target(const S* _tar, std::true_type)
30.  {
31.      return (const T*)_tar;
32.  }
33.  template <typename S, typename T>
34.  const void*
35.  function_manager::object_manager<S, T>::
36.  target(const S* _tar, std::false_type)
37.  {
```

```
38.      return _tar->template reinterpret_as<const T*>();
39.  }
40.
41.  template <typename S, typename T>
42.  void
43.  function_manager::object_manager<S, T>::
44.  clone(S* _dst, const S* _src, std::true_type)
45.  {
46.      new (reinterpret_cast<T*>(_dst)) T(_src->template reinterpret_as<T>());
47.  }
48.  template <typename S, typename T>
49.  void
50.  function_manager::object_manager<S, T>::
51.  clone(S* _dst, const S* _src, std::false_type)
52.  {
53.      _dst->template reinterpret_as<T*>() =
54.          new T(*_src->template reinterpret_as<T*>());
55.  }
56.
57.  template <typename S, typename T>
58.  void
59.  function_manager::object_manager<S, T>::
60.  destroy(S* _tar, std::true_type)
61.  {
62.      _tar->template reinterpret_as<T>().~T();
63.  }
64.  template <typename S, typename T>
65.  void
66.  function_manager::object_manager<S, T>::
67.  destroy(S* _tar, std::false_type)
68.  {
69.      delete _tar->template reinterpret_as<T*>();
70.  }
71.
72.  template <typename S, typename T>
73.  void
74.  function_manager::object_manager<S, reference_wrapper<T>>::
75.      initialize(S* _tar, reference_wrapper<T> _wrap)
76.  {
77.      _tar->template reinterpret_as<T*>() = std::addressof(_wrap.get());
78.  }
79.
80.  template <typename S, typename T>
81.  const void*
```

```
82.  function_manager::object_manager<S, reference_wrapper<T>>::
83.      target(const S* _tar)
84.  {
85.      return _tar->template reinterpret_as<const T*>();
86.  }
87.
88.  template <typename F>
89.  void
90.  function_manager::unary_manager<F>::
91.      initialize(function_data_wrapper* _tar, F&& _fun)
92.  {
93.      object_manager<function_data_wrapper, F>::initialize(_tar, std::move(_fun));
94.  }
95.
96.  template <typename F>
97.  bool
98.  function_manager::unary_manager<F>::
99.  manager(function_data_wrapper* _dst, const function_data_wrapper* _src,
100.     function_manager_operation _op)
101. {
102.     switch (_op)
103.     {
104.     case function_manager_operation::target:
105.     {
106.         _dst->reinterpret_as<const F*>() = reinterpret_cast<const F*>(
107.             object_manager<function_data_wrapper, F>::target(_src));
108.         break;
109.     }
110.     case function_manager_operation::clone:
111.         object_manager<function_data_wrapper, F>::clone(_dst, _src);
112.         break;
113.     case function_manager_operation::destroy:
114.         object_manager<function_data_wrapper, F>::destroy(_dst);
115.         break;
116.     case function_manager_operation::compare:
117.         return function_object_compare(*reinterpret_cast<const F*>
118.             (object_manager<function_data_wrapper, F>::target(_dst)),
119.                                        *reinterpret_cast<const F*>
120.             (object_manager<function_data_wrapper, F>::target(_src)));
121.     }
122.     return false;
123. }
124.
125. template <typename T, typename M>
```

```
126. void
127. function_manager::binary_manager<T, M>::
128. initialize(function_data_wrapper* _tar, T&& _obj, M&& _mem)
129. {
130.     object_manager<function_data_wrapper::first_type, T>::initialize
131.         (_tar->get_first_ptr(), std::move(_obj));
132.     object_manager<function_data_wrapper::second_type, M>::initialize
133.         (_tar->get_second_ptr(), std::move(_mem));
134. }
135.
136. template <typename T, typename M>
137. bool
138. function_manager::binary_manager<T, M>::
139. manager(function_data_wrapper* _dst, const function_data_wrapper* _src,
140.         function_manager_operation _op)
141. {
142.     switch (_op)
143.     {
144.     case function_manager_operation::target:
145.         _dst->reinterpret_first_as <const T*>() = reinterpret_cast<const T*>
146.             (object_manager<function_data_wrapper::first_type , T>::
147.                 target(_src->get_first_ptr()));
148.         _dst->reinterpret_second_as<const M*>() = reinterpret_cast<const M*>
149.             (object_manager<function_data_wrapper::second_type, M>::
150.                 target(_src->get_second_ptr()));
151.         break;
152.     case function_manager_operation::clone:
153.         object_manager<function_data_wrapper::first_type , T>::clone
154.             (_dst->get_first_ptr (), _src->get_first_ptr ());
155.         object_manager<function_data_wrapper::second_type, M>::clone
156.             (_dst->get_second_ptr(), _src->get_second_ptr());
157.         break;
158.     case function_manager_operation::destroy:
159.         object_manager<function_data_wrapper::first_type , T>::destroy
160.             (_dst->get_first_ptr());
161.         object_manager<function_data_wrapper::second_type, M>::destroy
162.             (_dst->get_second_ptr());
163.         break;
164.     case function_manager_operation::compare:
165.         return function_object_compare(*reinterpret_cast<const T*>
166.             (object_manager<function_data_wrapper::first_type , T>::
167.                 target(_dst->get_first_ptr ())),
168.                                        *reinterpret_cast<const T*>
169.             (object_manager<function_data_wrapper::first_type , T>::
```

```
                target(_src->get_first_ptr ())))
            && function_object_compare(*reinterpret_cast<const M*>
            (object_manager<function_data_wrapper::second_type, T>::
                target(_dst->get_second_ptr())),
                                    *reinterpret_cast<const M*>
            (object_manager<function_data_wrapper::second_type, T>::
                target(_src->get_second_ptr())));
    }
    return false;
}

template <typename F, typename R, typename... Args>
R
function_handler::unary_handler<F                   , R   , Args...>::
invoke(const function_data_wrapper& _ptr, Args... _args)
{
    return (*_ptr.reinterpret_as<const F*>())(std::forward<Args>(_args)...);
}
template <typename F, typename... Args>
void
function_handler::unary_handler<F                   , void, Args...>::
invoke(const function_data_wrapper& _ptr, Args... _args)
{
    (*_ptr.reinterpret_as<const F*>())(std::forward<Args>(_args)...);
}
template <typename F, typename R, typename... Args>
R
function_handler::unary_handler<reference_wrapper<F>, R   , Args...>::
invoke(const function_data_wrapper& _ptr, Args... _args)
{
    return (*_ptr.reinterpret_as<F*>())(std::forward<Args>(_args)...);
}
template <typename F, typename... Args>
void
function_handler::unary_handler<reference_wrapper<F>, void, Args...>::
invoke(const function_data_wrapper& _ptr, Args... _args)
{
    (*_ptr.reinterpret_as<F*>())(std::forward<Args>(_args)...);
}

template <typename T, typename M, typename R, typename... Args>
R
function_handler::binary_handler<T                   , M, R   , Args...>::
invoke(const function_data_wrapper& _ptrs, Args... _args)
```

```
214. {
215.     return (_ptrs.reinterpret_first_as<const T*>()->*
216.         *_ptrs.reinterpret_second_as<M*>())(std::forward<Args>(_args)...);
217. }
218. template <typename T, typename M, typename... Args>
219. void
220. function_handler::binary_handler<T                  , M, void, Args...>::
221. invoke(const function_data_wrapper& _ptrs, Args... _args)
222. {
223.            (_ptrs.reinterpret_first_as<const T*>()->*
224.         *_ptrs.reinterpret_second_as<M*>())(std::forward<Args>(_args)...);
225. }
226. template <typename T, typename M, typename R, typename... Args>
227. R
228. function_handler::binary_handler<reference_wrapper<T>, M, R   , Args...>::
229. invoke(const function_data_wrapper& _ptrs, Args... _args)
230. {
231.     return (_ptrs.reinterpret_first_as<T*>()->*
232.         *_ptrs.reinterpret_second_as<M*>())(std::forward<Args>(_args)...);
233. }
234. template <typename T, typename M, typename... Args>
235. void
236. function_handler::binary_handler<reference_wrapper<T>, M, void, Args...>::
237. invoke(const function_data_wrapper& _ptrs, Args... _args)
238. {
239.            (_ptrs.reinterpret_first_as<T*>()->*
240.         *_ptrs.reinterpret_second_as<M*>())(std::forward<Args>(_args)...);
241. }
242.
243. template <typename R, typename... Args>
244. inline R
245. function_handler::function_invoker<R, Args...>::
246. invoke(const function<R(Args...)>* _fun, Args... _args)
247. {
248.     if (!_fun->manager_)
249.         throw bad_function_call();
250.     detail::function_data_wrapper ptr;
251.     _fun->manager_(std::addressof(ptr), std::addressof(_fun->data_),
252.         detail::function_manager_operation::target);
253.     return _fun->handler_(ptr, std::forward<Args>(_args)...);
254. }
255. template <typename... Args>
256. inline void
257. function_handler::function_invoker<void, Args...>::
```

```
258. invoke(const function<void(Args...)>* _fun, Args... _args)
259. {
260.     if (!_fun->manager_)
261.         return;
262.     detail::function_data_wrapper ptr;
263.     _fun->manager_(std::addressof(ptr), std::addressof(_fun->data_),
264.         detail::function_manager_operation::target);
265.     return _fun->handler_(ptr, std::forward<Args>(_args)...);
266. }
267.
268. }
269.
270. template <typename R, typename... Args>
271. function<R(Args...)>::
272. function(const function& _other)
273.     : manager_(_other.manager_), handler_(_other.handler_)
274. {
275.     if (manager_)
276.         manager_(&data_, &_other.data_,
277.             detail::function_manager_operation::clone);
278. }
279.
280. template <typename R, typename... Args>
281. template <typename F>
282. function<R(Args...)>::
283. function(F _functor)
284. {
285.     if (detail::function_manager::valid_object(_functor))
286.     {
287.         detail::function_manager::unary_manager<F>::initialize
288.             (&data_, std::move(_functor));
289.         manager_ = &detail::function_manager::
290.             unary_manager<F>::manager;
291.         handler_ = &detail::function_handler::
292.             unary_handler<F, R, Args...>::invoke;
293.     }
294. }
295.
296. template <typename R, typename... Args>
297. template <typename T, typename M>
298. function<R(Args...)>::
299. function(T _obj, M _mem)
300. {
301.     if (detail::function_manager::valid_object(_obj) &&
```

```
302.         detail::function_manager::valid_object(_mem))
303.     {
304.         detail::function_manager::binary_manager<T, M>::initialize
305.             (&data_, std::forward<T>(_obj), std::forward<M>(_mem));
306.         manager_ = &detail::function_manager::
307.             binary_manager<T, M>::manager;
308.         handler_ = &detail::function_handler::
309.             binary_handler<T, M, R, Args...>::invoke;
310.     }
311. }
312.
313. template <typename R, typename... Args>
314. function<R(Args...)>::
315. ~function()
316. {
317.     if (manager_)
318.         manager_(&data_, nullptr, detail::function_manager_operation::destroy);
319. }
320.
321. template <typename R, typename... Args>
322. function<R(Args...)>&
323. function<R(Args...)>::
324. operator=(std::nullptr_t)
325. {
326.     this->~function();
327.     manager_ = nullptr;
328. }
329.
330. template <typename R, typename... Args>
331. template <typename F>
332. function<R(Args...)>&
333. function<R(Args...)>::
334. operator=(F _functor)
335. {
336.     this->~function();
337.     new (this) function(std::forward<F>(_functor));
338. }
339.
340. template <typename R, typename... Args>
341. template <typename T, typename M>
342. void
343. function<R(Args...)>::
344. assign(T _obj, M _mem)
345. {
```

```
346.     this->~function();
347.     new (this) function(std::forward<T>(_obj), std::forward<M>(_mem));
348. }
349.
350. template <typename R, typename... Args>
351. R
352. function<R(Args...)>::
353. operator()(Args... _args) const
354. {
355.     return detail::function_handler::function_invoker<R, Args...>::invoke
356.         (this, std::forward<Args>(_args)...);
357. }
358.
359. template <typename R, typename... Args>
360. void
361. function<R(Args...)>::
362. swap(function& _other) noexcept
363. {
364.     using std::swap;
365.     swap(data_, _other.data_);
366.     swap(manager_, _other.manager_);
367.     swap(handler_, _other.handler_);
368. }
369.
370. template <typename R, typename... Args>
371. function<R(Args...)>::
372. operator bool() const noexcept
373. {
374.     return manager_;
375. }
376.
377. template <typename R, typename... Args>
378. template <typename F>
379. const F*
380. function<R(Args...)>::
381. target() const noexcept
382. {
383.     if (manager_ == &detail::function_manager::unary_manager<F>::manager)
384.     {
385.         detail::function_data_wrapper ptr;
386.         manager_(&ptr, &data_, detail::function_manager_operation::target);
387.         return ptr.reinterpret_as<const F*>();
388.     }
389.     return nullptr;
```

```
390. }
391.
392. template <typename R, typename... Args>
393. template <typename T, typename M>
394. std::pair<const T*, const M*>
395. function<R(Args...)>::
396. target() const noexcept
397. {
398.     if (manager_ == &detail::function_manager::binary_manager<T, M>::manager)
399.     {
400.         detail::function_data_wrapper ptr;
401.         manager_(&ptr, &data_, detail::function_manager_operation::target);
402.         return { ptr.reinterpret_first_as<const T*>(),
403.             ptr.reinterpret_second_as<const M*>() };
404.     }
405.     return { nullptr, nullptr };
406. }
407.
408. template <typename R, typename... Args>
409. bool
410. operator==(const function<R(Args...)>& _lhs,
411.            const function<R(Args...)>& _rhs) noexcept
412. {
413.     if (_lhs.manager_ != _rhs.manager_)
414.         return false;
415.     return _lhs.manager_(
416.         const_cast<detail::function_data_wrapper*>(&_lhs.data_), &_rhs.data_,
417.         detail::function_manager_operation::compare);
418. }
419.
420. }
```

2. meds::event

event.h

```
1.  #include "pch.h"
2.
3.  #ifndef MEDS_EVENT_H
4.  #define MEDS_EVENT_H
5.
6.  //#include <forward_list>
7.
8.  #include "../function/function.h"
9.
10. namespace meds
```

```
11. {
12.
13. template <typename>
14. class event;
15.
16. template <typename R, typename... Args>
17. class event<R(Args...)>;
18.
19. enum class event_modifier_operation
20. {
21.     add, remove, clear,
22.     _destroy, _clone
23. };
24.
25. template <typename>
26. struct event_modifier_args;
27.
28. template <typename R, typename... Args>
29. struct event_modifier_args<R(Args...)>
30. {
31.     using operation_type = event_modifier_operation;
32.     operation_type operation;
33.     using function_type = function<R(Args...)>;
34.     const function_type& function;
35. };
36.
37. namespace detail
38. {
39.
40. template <typename... Args>
41. struct event_argument_type { };
42.
43. template <typename Arg>
44. struct event_argument_type<Arg>
45. {
46.     using argument_type = Arg;
47. };
48.
49. template <typename Fst, typename Snd>
50. struct event_argument_type<Fst, Snd>
51. {
52.     using first_argument_type  = Fst;
53.     using second_argument_type = Snd;
54.
```

```
55.      using object_type =
56.          typename remove_reference_wrapper<first_argument_type>::type;
57.      using pointer_type = object_type*;
58.      using event_type  = second_argument_type;
59.  };
60.
61.  struct event_manager
62.  {
63.      template <typename F, typename R, typename... Args>
64.      class modifier_manager
65.      {
66.      public:
67.          static void* manager(const event<R(Args...)>* _event,
68.              event_modifier_operation _op, const function<R(Args...)>* _func);
69.      };
70.
71.
72.      template <typename T>
73.      static typename std::enable_if< std::is_convertible<T, bool>::value,
74.          bool>::type
75.          valid_object(const T& _obj)
76.      { return _obj; }
77.      template <typename T>
78.      static typename std::enable_if<!std::is_convertible<T, bool>::value,
79.          bool>::type
80.          valid_object(const T& _obj)
81.      { return true; }
82.  };
83.
84.  }
85.
86.  template <typename R, typename... Args>
87.  class event<R(Args...)>
88.      : public detail::event_argument_type<Args...>
89.  {
90.  public:
91.      using function_type = function<R(Args...)>;
92.
93.      event() = default;
94.      event(const event& _other);
95.      event(event&& _other)
96.      { swap(_other); }
97.      template <typename F>
98.      event(F _functor);
```

```
99.
100.      ~event();
101.
102.      event& operator=(event _other)
103.      { swap(_other); }
104.
105.      void swap(event& _other);
106.
107.      bool add(const function_type& _func)
108.      { return emplace(_func); }
109.      bool add(function_type&& _func)
110.      { return emplace(std::move(_func)); }
111.      event& operator+=(const function_type& _func)
112.      { add(_func); return *this; }
113.      event& operator+=(function_type&& _func)
114.      { add(std::move(_func)); return *this; }
115.      template <typename... A>
116.      bool emplace(A&&... _args);
117.
118.      template <typename... A>
119.      bool remove(A&&... _args);
120.      event& operator-=(const function_type& _func)
121.      { remove(_func); return *this; }
122.
123.      bool empty() const
124.      //{ return delegate_.empty(); }
125.      { return !delegate_; }
126.      bool empty_removing(const function_type& _func) const;
127.
128.      void clear();
129.      //{ delegate_.clear(); }
130.
131.      void operator()(Args... _args) const;
132.
133.      template <typename F>
134.      void for_each(F&& _functor) const;
135.
136. private:
137.      void* modifier_ = nullptr;
138.      void* (*manager_)(const event* _event, event_modifier_operation, const
function_type*) = nullptr;
139.      //std::forward_list<function_type> delegate_;
140.      //typename std::forward_list<function_type>::iterator back_ =
delegate_.before_begin();
```

```
141.     struct node
142.     {
143.         template <typename... A>
144.         node(A&&... _args)
145.             : function_(std::forward<A>(_args)...) { }
146.         function_type function_;
147.         node* next_ = nullptr;
148.     };
149.     node* delegate_ = nullptr;
150.     node* back_ = nullptr;
151.
152.     template <typename F, typename R, typename... Args>
153.     friend class detail::event_manager::modifier_manager;
154. };
155.
156. template <typename R, typename... Args>
157. void swap(event<R(Args...)>& _lhs, event<R(Args...)>& _rhs) noexcept
158. { _lhs.swap(_rhs); }
159.
160. }
161.
162. #include "event_impl.cpp"
163.
164. #endif
```

event_impl.cpp

```
1.  #include "event.h"
2.
3.  namespace meds
4.  {
5.
6.  namespace detail
7.  {
8.
9.  template <typename F, typename R, typename... Args>
10. void* event_manager::modifier_manager<F, R, Args...>::
11. manager(const event<R(Args...)>* _event, event_modifier_operation _op,
12.     const function<R(Args...)>* _func)
13. {
14.     switch (_op)
15.     {
16.     case event_modifier_operation::_destroy:
17.         delete reinterpret_cast<F*>(_event->modifier_);
18.         break;
```

```
19.      case event_modifier_operation::_clone:
20.          return new F(*reinterpret_cast<const F*>(_event->modifier_));
21.      case event_modifier_operation::add:
22.      case event_modifier_operation::remove:
23.      case event_modifier_operation::clear:
24.          return (*reinterpret_cast<const F*>(_event->modifier_))
25.              (*const_cast<event<R(Args...)>*>(_event),
26.                  event_modifier_args<R(Args...)>{_op, *_func}) ?
27.              const_cast<void*>(reinterpret_cast<const void*>(_event)) : nullptr;
28.      }
29.      return nullptr;
30.  }
31.
32.  }
33.
34.  template <typename R, typename... Args>
35.  event<R(Args...)>::event(const event& _other)
36.      : manager_(_other.manager_), delegate_(_other.delegate_)
37.  {
38.      if (manager_)
39.          modifier_ = manager(&_other, event_modifier_operation::_clone,
40.              nullptr);
41.      //auto prev = delegate_.before_begin();
42.      //auto cur = delegate_.begin();
43.      //auto end = delegate_.end();
44.      //for (; cur != end; ++cur)
45.      //    prev = cur;
46.      //back_ = prev;
47.
48.      // TODO
49.  }
50.
51.  template <typename R, typename... Args>
52.  template <typename F>
53.  event<R(Args...)>::event(F _functor)
54.  {
55.      if (detail::event_manager::valid_object(_functor))
56.      {
57.          modifier_ = new F(std::move(_functor));
58.          manager_ = &detail::event_manager::modifier_manager<F, R, Args...>::
59.              manager;
60.      }
61.  }
62.
```

```
63.  template <typename R, typename... Args>
64.  event<R(Args...)>::~event()
65.  {
66.      if (manager_)
67.          manager_(this, event_modifier_operation::_destroy, nullptr);
68.      while (delegate_)
69.      {
70.          auto next = delegate_->next_;
71.          delete delegate_;
72.          delegate_ = next;
73.      }
74.  }
75.
76.  template <typename R, typename... Args>
77.  void event<R(Args...)>::swap(event& _other)
78.  {
79.      using std::swap;
80.      swap(modifier_, _other.modifier_);
81.      swap(manager_, _other.manager_);
82.      swap(delegate_, _other.delegate_);
83.      swap(back_, _other.back_);
84.  }
85.
86.  //template <typename R, typename... Args>
87.  //bool event<R(Args...)>::add(const function_type& _func)
88.  //{
89.  //    for (const auto& f : delegate_)
90.  //        if (_func == f)
91.  //            return false;
92.  //    if (!manager_ || manager_(this, event_modifier_operation::add, &_func))
93.  //    {
94.  //        back_ = delegate_.emplace_after(back_, _func);
95.  //        return true;
96.  //    }
97.  //    return false;
98.  //}
99.
100. template <typename R, typename... Args>
101. template <typename... A>
102. bool event<R(Args...)>::emplace(A&&... _args)
103. {
104.     function_type func(std::forward<A>(_args)...);
105.     for (auto cur = delegate_; cur; cur = cur->next_)
106.         if (cur->function_ == func)
```

```
107.             return false;
108.     if (!manager_ || manager_(this, event_modifier_operation::add, &func))
109.     {
110.         //back_ = delegate_.emplace_after(back_, func);
111.         auto next = new node(std::move(func));
112.         if (back_)
113.         {
114.             back_->next_ = next;
115.             back_ = next;
116.         }
117.         else
118.         {
119.             delegate_ = next;
120.             back_ = next;
121.         }
122.
123.         return true;
124.     }
125.     return false;
126. }
127.
128. template <typename R, typename... Args>
129. template <typename... A>
130. bool event<R(Args...)>::remove(A&&... _args)
131. {
132.     auto func = function_type(std::forward<A>(_args)...);
133.     //auto prev = delegate_.before_begin();
134.     //auto cur = delegate_.begin();
135.     //auto end = delegate_.end();
136.     //for (; cur != end; prev = cur, ++cur)
137.     //    if (*cur == func)
138.     //        break;
139.     //if (cur == end)
140.     //    return false;
141.     auto cur = delegate_;
142.     node* prev = nullptr;
143.     for (; cur; prev = cur, cur = cur->next_)
144.         if (cur->function_ == func)
145.             break;
146.     if (!cur)
147.         return false;
148.     if (!manager_ || manager_(this, event_modifier_operation::remove, &func))
149.     {
150.         //if (++cur == end)
```

```
151.         //    back_ = prev;
152.         //delegate_.erase_after(prev);
153.         if (!cur->next_)
154.             back_ = prev;
155.         auto next = cur->next_;
156.         delete cur;
157.         if (cur == delegate_)
158.             delegate_ = next;
159.         else
160.             prev->next_ = next;
161.         return true;
162.     }
163.     return false;
164. }
165.
166. template <typename R, typename... Args>
167. bool event<R(Args...)>::empty_removing(const function_type& _func) const
168. {
169.     //auto iter = delegate_.begin();
170.     //return !delegate_.empty() && ++iter == delegate_.end() && *delegate_.begin() ==
_func;
171.     auto cur = delegate_;
172.     return delegate_ && !delegate_->next_ && delegate_->function_ == _func;
173. }
174.
175. template <typename R, typename... Args>
176. void event<R(Args...)>::clear()
177. {
178.     while (delegate_)
179.     {
180.         auto next = delegate_->next_;
181.         delete delegate_;
182.         delegate_ = next;
183.     }
184. }
185.
186. template <typename R, typename... Args>
187. void event<R(Args...)>::operator()(Args... _args) const
188. {
189.     //for (const auto& f : delegate_)
190.     //    f(_args...);
191.     auto cur = delegate_;
192.     for (auto cur = delegate_; cur; cur = cur->next_)
193.         cur->function_(_args...);
```

```
194. }
195.
196. template <typename R, typename... Args>
197. template <typename F>
198. void event<R(Args...)>::for_each(F&& _functor) const
199. {
200.     //for (const auto& f : delegate_)
201.     //    _functor(f);
202.     auto cur = delegate_;
203.     for (auto cur = delegate_; cur; cur = cur->next_)
204.         _functor(cur->function_);
205. }
206.
207. }
```

3. 核心模块抽象层类定义

core_i2c.h

```
1.  #ifndef MEDS_CORE_I2C_H
2.  #define MEDS_CORE_I2C_H
3.
4.  #include "../stdcpp/cstdint.h"
5.
6.  namespace meds
7.  {
8.
9.  struct I2cData
10. {
11.     explicit I2cData(uint8_t _address);
12.     I2cData(uint8_t _address, uint8_t _length, uint8_t* _data);
13.     uint8_t address;
14.     uint8_t length;
15.     uint8_t* data;
16. };
17.
18. class I2c final
19. {
20. public:
21.     I2c(const I2c&) = delete;
22.     I2c(I2c&&) = delete;
23.     I2c& operator=(const I2c&) = delete;
24.     I2c& operator=(I2c&&) = delete;
25.
26.     static I2c& instance();
```

```
27.
28.     I2c& operator<<(const I2cData& _packet);
29.     I2c& operator>>(I2cData& _packet);
30.
31. private:
32.     static I2c* instance_;
33.
34.     I2c();
35. };
36.
37. extern I2c& i2c;
38.
39. }
40.
41. #endif
```

module.h

```
1.  #ifndef MEDS_CORE_MODULE_H
2.  #define MEDS_CORE_MODULE_H
3.
4.  #include "../stdcpp/cstdint.h"
5.
6.  namespace meds
7.  {
8.
9.  class Module
10. {
11. public:
12.     Module(const Module&) = delete;
13.     Module(Module&&) = delete;
14.     Module& operator=(const Module&) = delete;
15.     Module& operator=(Module&&) = delete;
16.
17.     virtual ~Module() = 0;
18.
19.     uint8_t address();
20.     virtual void address(uint8_t _address) = 0;
21.
22.     void red   (bool _on);
23.     void yellow(bool _on);
24.     void green (bool _on);
25.     void blue  (bool _on);
26.
27. protected:
```

```
28.      Module(uint8_t _address);
29.
30.      virtual void set_led(uint8_t _which, bool _on) = 0;
31.
32.  private:
33.      uint8_t address_;
34.  };
35.
36.  }
37.
38.  #endif
```

core_module.h

```
1.   #ifndef MEDS_CORE_CORE_H
2.   #define MEDS_CORE_CORE_H
3.
4.   #include "module.h"
5.   #include "collection.h"
6.
7.   namespace meds
8.   {
9.
10.  class CoreModule
11.      : public Module
12.  {
13.  public:
14.      virtual ~CoreModule() override = 0;
15.
16.      using Module::address;
17.      virtual void address(uint8_t _address) override final;
18.
19.  protected:
20.      CoreModule();
21.  };
22.
23.  }
24.
25.  #endif
```

app_module.h

```
1.   #ifndef MEDS_CORE_APP_H
2.   #define MEDS_CORE_APP_H
3.
4.   #include "module.h"
```

```
5.
6.  namespace meds
7.  {
8.
9.  class AppModule
10.     : public Module
11. {
12. public:
13.     virtual ~AppModule() override;
14.
15.     using Module::address;
16.     virtual void address(uint8_t _address) override final;
17.
18. protected:
19.     friend class ModuleCollection;
20.
21.     AppModule(uint8_t _address);
22.
23.     virtual void callback(uint8_t _size, uint8_t* _data) = 0;
24.
25.     virtual void set_led(uint8_t _which, bool _on) override final;
26. };
27.
28. }
29.
30. #endif
```

collection.h

```
1.  #ifndef MEDS_CORE_COLLECTION_H
2.  #define MEDS_CORE_COLLECTION_H
3.
4.  #include "../i2c_avr/i2c.h"
5.  #include "app_module.h"
6.
7.  namespace meds
8.  {
9.
10. class ModuleCollection final
11. {
12. public:
13.     ModuleCollection(const ModuleCollection&) = delete;
14.     ModuleCollection(ModuleCollection&&) = delete;
15.     ModuleCollection& operator=(const ModuleCollection&) = delete;
16.     ModuleCollection& operator=(ModuleCollection&&) = delete;
```

```
17.
18.      ~ModuleCollection();
19.
20.      static ModuleCollection& instance();
21.
22.      template <typename F>
23.      void for_each(F&& _functor);
24.
25.  private:
26.      friend class I2c;
27.      friend class AppModule;
28.
29.      static constexpr uint8_t high_max = 14;
30.      static constexpr uint8_t low_max = 8;
31.
32.      static ModuleCollection* instance_;
33.
34.      AppModule** modules_[low_max];
35.
36.      ModuleCollection();
37.
38.      static void callback(i2c_data_t* _packet);
39.
40.      static uint8_t address_low(uint8_t _address);
41.      static uint8_t address_high(uint8_t _address);
42.
43.      void handle(i2c_data_t* _packet);
44.
45.      void add(AppModule* _module);
46.      void change(uint8_t _original);
47.  };
48.
49.  extern ModuleCollection& modules;
50.
51.  }
52.
53.  #endif
```

4. 应用实例

main.cpp

```
1.   #include "../meds_cor/cor.h"
2.   #include "../meds_led/led.h"
3.   #include "../meds_bzr/bzr.h"
```

```
4.  #include "../meds_btn/btn.h"
5.
6.  using meds::core;
7.  auto& led = meds::ModuleLED::abstract(0x20);
8.  auto& bzr = meds::ModuleBZR::abstract(0x30);
9.  auto& btn = meds::ModuleBTN::abstract(0x40);
10.
11. uint16_t freq[] = {
12.     262, 294, 330, 349, 392, 440, 494, 523
13. };
14.
15. void onButtonChanged(meds::ModuleBTN&, meds::ButtonEventArgs e)
16. {
17.     switch (e.what)
18.     {
19.     case meds::ButtonEvent::released:
20.         led.bar(e.which).turn(0);
21.         bzr.buzzer().add(freq[e.which]);
22.         break;
23.     case meds::ButtonEvent::pressed:
24.         led.bar(e.which).turn(1);
25.         bzr.buzzer().remove(freq[e.which]);
26.         break;
27.     }
28. }
29.
30. int main()
31. {
32.     core.green(1);
33.     bzr.led().mode(meds::ModuleBZR::LedMode::level);
34.
35.     btn.Slide += [](meds::ModuleBTN&, meds::SlideEventArgs e) {
36.         switch (e.what)
37.         {
38.         case meds::SlideEvent::off:
39.             btn.button().enable();
40.             btn.Button -= onButtonChanged;
41.             led.bar().set(0);
42.             led.rgb().off();
43.             bzr.buzzer().off();
44.             break;
45.         case meds::SlideEvent::on:
46.             btn.Button += onButtonChanged;
47.             btn.button().disable();
```

```
48.             led.rgb().random();
49.             break;
50.         }
51.     };
52.     core.Adc += [](meds::AdcEventArgs e) {
53.         switch (e.what)
54.         {
55.         case meds::AdcEvent::lower:
56.             led.brightness(1);
57.             bzr.brightness(1);
58.             bzr.volume(1);
59.             break;
60.         case meds::AdcEvent::higher:
61.             led.brightness(3);
62.             bzr.brightness(3);
63.             bzr.volume(2);
64.             break;
65.         }
66.     };
67.
68.     if (btn.slide(1).status())
69.         btn.Slide(btn, meds::SlideEventArgs(1, meds::SlideEvent::on));
70.     core.Adc(meds::AdcEventArgs(0, core.adc(0).status() ?
71.         meds::AdcEvent::higher : meds::AdcEvent::lower));
72.
73.     btn.slide(1).enable();
74.     core.adc(0).enable(meds::AdcEvent::window, 128);
75.
76.     while (1)
77.         ;
78. }
```

致谢

感谢上海中学王亚娟老师在论文写作过程中给予我指导。

感谢上海交通大学张士文教授与我交流观点，以及提供实验室资源。

感谢上海交通大学姚瑞文教授倾听我的想法，并给了我许多建议。

感谢我的班主任柳怡汀与刘丽霞、赵奇玮等任课老师对我的宽容，让我有充足的时间来完成研究。