C#上位机实战开发指南

第一章　C#和 Visual Stduio

1.1　.NET 时代

在.NET 之前，尤其是 20 世纪 90 年代，Windows 程序员几乎使用 VB，C 或者 C++。部分 C 和 C++开发者使用纯 Win32 Api，但是大多数人还是选择使用 MFC。这些语言开发难度较大，底层代码复杂。21 世纪初期越来越多的开发者迫切需要一个安全，集成度高，面向对象的开发框架。

2002 年，微软如期发布了.NET 框架的第一个版本，它具有如下几个特点：

●多平台　　可在任意计算机系统运行，包括服务器，台式机等。

●安全性　　提供更加安全的运行环境，即使有来源可疑的代码存在。

●行业标准　使用标准通信协议，比如 HTTP，SOAP，JSON 等。

在 2016 年最新一期的编程语言排行榜中 C#.NET 位列第四，而且呈上升趋势。排行榜如图 1-1 所示。

Jan 2016	Jan 2015	Change	Programming Language	Ratings	Change
1	2	↑	Java	21.465%	+5.94%
2	1	↓	C	16.036%	-0.67%
3	4	↑	C++	6.914%	+0.21%
4	5	↑	C#	4.707%	-0.34%
5	8	↑	Python	3.854%	+1.24%
6	6		PHP	2.706%	-1.08%
7	16	⇈	Visual Basic .NET	2.582%	+1.51%
8	7	↓	JavaScript	2.565%	-0.71%
9	14	⇈	Assembly language	2.095%	+0.92%
10	15	⇈	Ruby	2.047%	+0.92%
11	9	↓	Perl	1.841%	-0.42%
12	20	⇈	Delphi/Object Pascal	1.786%	+0.95%
13	17	⇈	Visual Basic	1.684%	+0.61%
14	25	⇈	Swift	1.363%	+0.62%
15	11	⇊	MATLAB	1.228%	-0.16%
16	30	⇈	Pascal	1.194%	+0.52%
17	82	⇈	Groovy	1.182%	+1.07%
18	3	⇊	Objective-C	1.074%	-5.88%
19	18	↓	R	1.054%	+0.01%
20	10	⇊	PL/SQL	1.016%	-1.00%

图 1-1

1.2 C#的前世今生

C#是微软发布的一种面向对象，运行于.NET 之上的高级语言。也是微软近几年主推的开发语言，可以说是微软.NET 框架的主角。只要具备一些 C 语言基础就可以非常迅速的入门 C#开发，这也是我极力推荐使用 C#开发上位机的一个重要原因。

1.3 难以置信的 Visual Studio 2015

Visual Studio 2015（以下简称 VS2015）是微软推出的开发环境，C#也是基于此开发。相比较之前的版本，VS2015 具有更强大的调试功能，甚至集成了安卓，IOS 等跨平台开发环境。作为一个强大的集成开发环境，VS2015 同时还能支持 STM32 单片机的编译。

具体教程请参看：http://www.openedv.com/thread-10273-1-1.html。

笔者认为 VS2015 是宇宙最强 IDE，完虐我们常用的单片机开发环境如 KEIL，IAR 等。更多使用技巧就留给读者自己去发现吧。

1.4 VS2015 的安装与使用

第二章　C#语法基础

2.1　C#编程概述

本章将为上位机开发打基础，当然具有 C 语言或者单片机开发经验的同学也可以跳过本章，直接进入第三章窗体程序的学习中。因为 C#和 C 语言在语法上大致相同。本章只讲解一些与单片机 C 语言相差较大的部分，其余不再过多讲解。代码分析也全部放在第三章以后。若想深入学习 C#，请参考专业入门书籍，推荐《C#图解教程》（第四版）。

2.2　命名空间

在 C#中，命名空间提供了一种组织相关类和其它类型的方式。我理解的命名空间就是一个集装箱，里面可以装下很多类和方法。其实我们也可以认为所谓的命名空间相当于 C 语言中的头文件，只不过 include 变为了 using namespace。具体的书写规范见代码清单 2-1。

代码清单 2-1：命名空间书写规范

```
1.  using System;                                   //命名空间类似于头文件
2.  using System.Collections.Generic;               //using ≈ include; Systerm ≈ xxxx.h
3.  using System.ComponentModel;
4.  using System.Data;
5.  using System.Drawing;
6.  using System.Text;
7.  using System.Windows.Forms;
8.
9.  //用户自定义命名空间，相当于新定义一个头文件
10. //一般情况下一个上位机工程对应一个新的命名空间
11. namespace Demo
12. {
13.
14.     public partial class Form1 : Form
15.     {
16.         //构造函数，新建窗体工程时自动创建这段代码，可先忽略
17.         public Form1()
18.         {
19.             InitializeComponent();
20.         }
21.     }
22. }
```

这段代码在新建工程之后由 VS2015 自动创建，第 1 到 7 行代码全都为系统自带的命名空间。第 11 行为开发人员自定义的命名空间，之后的每一个上位机项目都是一个自定义命名空间。

大概了解了 C#命名空间的书写格式规范后，我们再简单回忆一下 C 语言中头文件的书写规范并比较二者的异同点，C 头文件书写格式见代码清单 2-2。

代码清单 2-2：C 头文件书写规范

```
1.  #ifndef __USART_H
2.  #define __USART_H
3.
4.
5.  #include "stm32f10x.h"
6.  #include "stdio.h"
7.  #include "string.h"
8.
9.
10. #define  TxBuffSize    256
11.
12.
13. #define  Debug_ON       1
14.
15.
16. #define  DebugPutInfo(fmt,arg...) do{if(Debug_ON)printf(fmt,##arg);}while(0)
17.
18.
19. void USART_Config(void);
20. void USART1_SendByte(uint8_t DataToSend);
21. void USART1_SendString(const char* StringToSend);
22. void USART1_SendBuff(uint8_t* DataToSend, uint8_t DataNum);
23.
24.
25. #endif
```

通过代码清单 2-2 我们很容易发现，C#的命名空间和 C 的头文件遵循一样的规则，即要想使用某方法某函数则必须要包含方法所在的命名空间或者头文件。这是相同点。不同点则表现在 C 语言在声明了头文件和函数接口后必须要在对应的 C 文件中编写函数体后才可使用。C#则将省去了函数声明，直接即可编写对应的函数体。

2.3 类

2.3.1 什么是类

在 C#开发中，类(class)至关重要。可以认为类是 C#一个很大的主题。关于它的讨论将一直延续到本书结束。我们在单片机软件开发中设计数据结构时往往离不开先设计结构体，其实类就相当于结构体，这也是面向对象的一个前提条件。我们可以将类抽象成一个既能存储数据又能执行代码的数据结构。它包含数据成员和函数成员，因此类对 C#代码的封装起着举足轻重的作用。

2.3.2 如何声明一个类

类的声明和结构体类似，即定义了一个新类的成员和特征。但是它并不创建类的实例，相当于结构体声明后并不分配内存，只有在使用时声明后才会分配内存一样，类的声明和实例化不可混淆。类的声明方式如代码清单 2-3 所示。

代码清单 2-3：类的声明方式

```
1.  using System;                           //命名空间类似于头文件
2.  using System.Collections.Generic;       //using ≈ include; Systerm ≈ xxxx.h
3.  using System.ComponentModel;
4.  using System.Data;
5.  using System.Drawing;
6.  using System.Text;
7.  using System.Windows.Forms;
8.
9.  //用户自定义命名空间，相当于新定义一个头文件
10. //一般情况下一个上位机工程对应一个新的命名空间
11. namespace Demo
12. {
13.
14.     public partial class Form1 : Form
15.     {
16.         //构造函数，新建窗体工程时自动创建，可先忽略
17.         public Form1()
18.         {
19.             InitializeComponent();
20.         }
21.
22.         //类的声明方式
23.         class MyClass
24.         {
25.             //成员声明... ...
```

```
26.             //class 为类的关键字，MyClass 为类名
27.         }
28.     }
29. }
```

从代码清单 2-3 可以清晰看出，类的声明非常简单。其中类的成员可以是变量，也可以是函数方法。

2.4 Main：程序由你开始

每一个 C#程序都必须有一个类带有 Main 函数(方法)，它是程序的开始，它通常被声明在 Program 类中。这就好比我们在开发单片机时喜欢将 main 函数声明在 main.c 中一样。通常 Program.cs 文件随工程一起创建，详细代码见图 2-1。

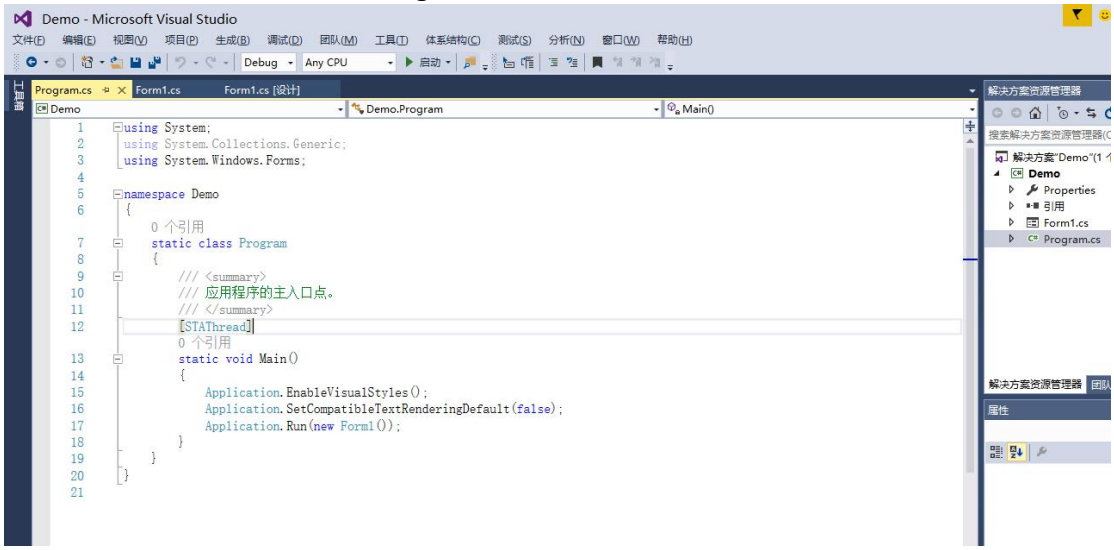


图 2-1

从图中我们清晰的看到了 Class 关键字，其实 Program.cs 本身就是一个类文件。

2.5 变量与常量

2.5.1 值类型与引用类型

值类型和我们单片机开发中的数据类型类似，需要一段独立内存存放它的实际数据。如果值类型变量定义在方法(函数)内部那么在调用结束后这片内存回收。相反如果定义为全局，那这片内存则不会被回收。这和 C 基本一样。char，int float，enum，struct 等都是值类型。

引用类型是一个特殊的类型，它的存储需要两片内存。实例数据存放在堆中，

引用存放在栈中，引用可以理解为指针。具体引用类型为什么需要两片内存不再做任何讨论，我们只需要知道引用类型的使用和常规的值类型有什么区别就行。C 语言中如果我们表示一段字符串可以定义一个指针，在 C#中直接使用 string 关键字即可定义。string 便是一个非常典型的引用类型，它不遵循值类型的规则。当我们定义一个 string 类型变量并且第一次赋值时假设它在地址 0x02000000 中，那么在第二次赋值再次查看内存时，它已经不在上一次地址中，即引用类型每次在使用后都会变更内存地址。引用类型在并行多线程的使用中尤为重要。

当然，在上位机开发中我们可以将引用类型当作一般类型来使用。

2.5.2 声明变量

C#声明变量和 C 语言相同，声明过程完成两件事。

- ●给变量命名，并且关联一种类型
- ●编译器为其分配一片内存

2.5.3 变量的作用域

类中的变量作用域就在类中，类被回收，变量即被回收。方法(函数)内部变量作用域为整个方法体。其中如果变量是某循环某判断中定义的，作用域就在循环或者判断体内。

2.5.4 访问修饰符

代码清单 2-3 中类的声明在 class 前未添加任何访问修饰符，C#规定无访问修饰符的情况下类成员即为隐式私有，外部不可访问。

C#常用的访问修饰符有以下 5 个。

◇私有的：private
◇公开的：public
◇受保护的：protected
◇内部的：internal
◇受保护内部的：protected internal

顾名思义，private 私有即外部不可访问，只能在类的内部使用，而 public 修饰的变量则可以在类的外部访问。关于 private 和 public 以及变量在类中的使用查看代码清单 2-4。

代码清单 2-4：访问修饰符及变量在类中的简单使用

```
1.  using System;                          //命名空间类似于头文件
2.  using System.Collections.Generic;      //using ≈ include; Systerm ≈ xxxx.h
3.  using System.ComponentModel;
4.  using System.Data;
```

```
using System.Drawing;
using System.Text;
using System.Windows.Forms;

//用户自定义命名空间，相当于新定义一个头文件
//一般情况下一个上位机工程对应一个新的命名空间
namespace Demo
{
    public partial class Form1 : Form
    {
        //构造函数，新建窗体工程时自动创建，可先忽略
        public Form1()
        {
            InitializeComponent();
        }

        int data0 = 0;              //全局变量声明，变量声明后即可在方法中使用
                                    //与 C 语言相同


        //类的声明方式
        class MyClass
        {
            //成员声明... ...
            //class 为类的关键字，MyClass 为类名

            int data1 = 0;          //无修饰符默认隐式私有
                                    //外部不可访问

            public int data2 = 0;//公有，外部可访问
        }
    }
}
```

从代码清单 2-4 可看出类 MyClass 中 data2 添加了修饰符 public，因此它可以在类的外部被调用。

2.6 多线程的使用

2.6.1 线程概述

相信大家在嵌入式 RTOS 中就已经接触过多线程(多任务)的处理机制。同样在

多线程的使用下 C#便可以并行执行代码。注意，这里的并行并不是真正意义上的同时执行，只是任务上下文切换速度极快，给人的感觉好像是在并行。

一个 C#程序开始于一个单线程(Main 方法入口)，这个线程是由操作系统自动创建的，我们也称之为主线程或者 UI 线程。同时主线程下可以创建多个子线程。

2.6.2 何时使用多线程

多线程一般情况下用在后台处理耗时任务，主线程保持执行。对于 Winform 来讲，如果所有耗时任务都放在主线程执行，那就会带来鼠标键盘等响应迟钝现象。为了避免这个现象，我们可以在主线程中再创建一个子线程，这样就避免了阻塞主线程，导致 UI 响应迟钝的现象。一个优秀的交互软件必定会有多线程的使用。

2.6.3 多线程的优缺点

在多线程的帮助下，我们可以快速的实现异步操作，这使得软件的 UI 可以迅速响应，给客户一个极佳的 UI 体验。无论我们是否使用过 RTOS，但 STM32 中的 DMA 我相信大家使用的非常之多，DMA 在进行内存拷贝传输时完全不需要 CPU 干预，由此我们完全可以理解为 DMA 是一个全硬件实现的子线程。

当然多线程并非全无缺点，最大的问题便是加大了代码的复杂性。当然多线程本身非常简单，但线程间的交互却非常复杂，使用不当甚至会带来间歇性或重复性的 BUG。同时多线程无意间又增加了 CPU 资源的消耗。

2.6.4 多线程的简单使用

一般情况下上位机多线程都使用局部线程，它和局部变量类似，用时创建，用完销毁。全局线程在上位机开发当中使用的相对比较少。当然全局也可以使用但必须要自己实现挂起和恢复函数，系统自带的接口函数已经过时，容易造成阻塞，实际开发中我们也几乎很少用到全局线程。因此我将只介绍局部线程的使用方法。局部线程存在于方法中，像局部变量一样使用，具体介绍请看代码清单 2-5。

代码清单 2-5：局部线程的使用

```
1.  using System;                           //命名空间类似于头文件
2.  using System.Collections.Generic;       //using ≈ include; Systerm ≈ xxxx.h
3.  using System.ComponentModel;
4.  using System.Data;
5.  using System.Drawing;
6.  using System.Text;
7.  using System.Threading;
8.  using System.Windows.Forms;
9.
```

```
//用户自定义命名空间，相当于新定义一个头文件
//一般情况下一个上位机工程对应一个新的命名空间
namespace Demo
{
    public partial class Form1 : Form
    {
        //新建窗体工程时自动创建这段代码，可先忽略
        public Form1()
        {
            InitializeComponent();
        }

        int data0 = 0;              //全局变量声明，变量声明后即可在方法中使用
                                    //与 C 语言相同


        //类的声明方式
        class MyClass
        {
            //成员声明... ...
            //class 为类的关键字，MyClass 为类名

            int data1 = 0;          //无修饰符默认隐式私有
                                    //外部不可访问

            public int data2 = 0;//公有，外部可访问
        }

        /// <summary>
        /// 多线程的使用
        /// </summary>
        public void ThreadTest()
        {
            new Thread(() =>
            {
                ///
                ///添加自定义代码
                ///例如线程内部运行函数方法：xxxxx
                /// xxxxx();
            })
            { IsBackground = true, Name = "多线程的使用" }.Start();
        }
    }
}
```

从代码清单 2-5 中我们看到函数 ThreadTest，内部就是一个局部子线程的写法。其中 IsBackgroud = true 是将该线程设置为了后台线程。后台线程的作用主要在于当上位机程序关闭时线程也自动随窗体一起销毁了，如果不设置为后台线程窗体关闭时线程依然在消耗着 CPU。因此使用时切记一定要将后台属性设置为 true。Name 即线程的名字，注意区分的作用，没有什么好讲，写与不写没有区别。Start 即启动线程的接口函数，运行后就将执行线程内部的方法。内部函数执行完线程销毁。

注意局部线程和全局线程只是个人的称呼，在 C#中并没有严格的定义来区分这两种线程，我们会用即可。

2.7 异常处理

2.7.1 异常概述

见名知意，异常就是软件在运行中所发生的错误。比如上位机串口未打开就调用了发送方法，此时系统就会捕获到这个错误，并抛出一个异常。如果软件设计时没有提供一个异常处理的方法，则系统自动将软件挂起。通常我们在调用串口发送方法前都会判断是否开启串口或者嵌套 try…catch 语句来捕获异常防止软件被系统挂起。

2.7.2 try...catch 语句

try 语句用来指明为避免异常而被保护的代码段，并在发生异常时提供处理代码。一般情况下我们使用 try…catch 组合语句来保护关键性代码。具体使用实例在上位机实战章节具体介绍。

2.8 属性和方法

2.8.1 什么是属性

在本章第二小节中我们简要的接触了类的概念，类相当于一个结构体但不能等价于一个结构体，因为类是具有属性的，而结构体没有。在结构体内部定义一个缓冲区，这个缓冲区的大小必须在程序编译前确定下来，运行中不可改变。但类通过属性却可以修改这个缓冲区的大小。那么什么是属性呢？属性就好比一个人的发色，生来黑色，但不会永远是黑色，我们可以随意染成红蓝紫色。也就是说属性是一个类的动态特性，比如上位机在运行过程中我们可以随时修改波特率。

2.8.2 属性的优点

上一节中我们提到上位机的波特率可以在运行过程中任意修改，这就是属性的一个优点。

当然属性也对类内部的私有变量提供了一种保护机制。要想修改类内部私有变量值就必须通过属性来操作。这就好比去银行存钱我们无法进入金库，只能通过 ATM 机是一样的。

2.8.3 属性的使用介绍

属性一般情况下可直接在控件属性栏中设置，也可通过代码设置。上位机开发中一般都是使用系统自带的类库，很少会自己编写类库以及属性。因此本节就不再介绍如何写声明属性，在具体讲到控件使用时再介绍属性的妙用。

2.8.4 什么是方法

C#中的方法类似于 C 语言函数，只是 C#赋予了方法属性，方法既可以是私有的，又可以是共用的。谈及私有方法，其实我们完全可以将它理解为 C 语言函数套了一个 static 关键字，代表这是单文件中使用的。

2.9 委托和事件

2.9.1 什么是委托

委托可以说是 C#第一个要跨过去的坎儿，理解难度比较大。但我会在接下来的上位机实战章节中具体介绍学习的每一步，在本章就做一个简要的介绍。

我非常喜欢将委托比喻成 C 语言中的函数指针数组，我们知道函数指针的存在极大的方便了我们设计单片机软件架构，事件回调机制等封装技术都基于函数指针实现。无独有偶，在 C#中事件回调机制也是通过委托实现，所以我一直认为软件思想都是相通的，只是表现形式上换了一个说法而已。对我们单片机出身的软件开发人员来讲，理解起委托易如反掌，因为我们已经在底层深耕多年。

那么 C#如何定义委托呢？可以认为委托是持有一个或者多个方法的对象。委托和类一样，是一种用户自定义类型，不同在类是数据和方法的集合。执行委托即执行了委托中所有的方法。

2.9.2 什么是事件

学习 STM32 之时，我们已经接触过事件的概念。事件是由硬件实现，可触发中断以及关联性操作，如 ADC，DMA 等。它和中断最大的区别在于事件无需返回，而中断需要返回。事件不仅在 MCU 硬件中大量使用，同时又与单片机软件

架构设计息息相关。

所有的 PC 端程序都需要在某个特定的时刻响应某个操作，处理某件事情，比如响应鼠标单击事件，键盘事件等，因此 C#也引入了事件触发机制。在上一节的内容中我们简要介绍了委托，事件本质上源于委托，是一种特殊的委托，它为委托提供了封装性，一方面允许从类的外部增删绑定方法，另一方面又严禁从类的外部触发委托所绑定的方法。

我们的目的是快速开发上位机，因此在使用过程中完全可以将事件理解为中断，事件回调函数就是我们常说的中断服务函数。同时一般情况下我们也不需要自己封装事件，调用控件已经封装好的事件函数即可。因此本章就不再做过多的代码实例讲解，事件的使用以及注意点将在上位机实战章节中做具体的介绍。