SmartOS 用户手册

SmartOS 是无声物联独立研发完成的物联网操作系统，采用 C++11 语言编写，主要用户是高级软件工程师，它的目标是在各个行业实现智能化。

无声物联

万家灯火无声物联

www.wslink.cn

目录

版本	作者	时间	备注
v1.0	大石头	2016-10-26	创建
v1.1			
v1.2			

一、系统介绍

SmartOS 是无声物联独立研发完成的物联网操作系统，使用 C++11 开发完成，主要面向 C++/C#/Java 高级软件工程师。

SmartOS 系统，以下简称系统。

1. 功能特点

主要功能特点：

1) 系统以网络通信为核心，以**远程数据采集**、**数据按需处理**以及各种**执行器集中控制**为主要方向。
2) 主要向软件型物联网企业开放，系统可以有线无线对接各种传感器并支持实现大数据分析和人工智能。
3) 芯片内**支持 C++11 二次开发**，用户接口**完全符合 C#/Java 标准**，简单易用，可快速入门。
4) 内置 100 多种硬件设备驱动，可对接绝大部分常见传感器。
5) 系统在最低端的 Cortex-M0 微处理器上，完整**系统启动时间为 9.5ms**，系统**时间精度为 0.001ms**。
6) 具备自动组网技术，此系统技术已获得有多项发明专利。系统支持开放式无声云平台、支持手机 APP 对接、以及微信对接、支持远程内网穿透、支持十万级硬件设备集群采集与集中控制、支持内网扫描发现。

2．安全性

当下物联网产品的安全性很差，SmartOS 追求安全第一。

1) 我们把安全摆在首位，在安全上投入巨大研发，并申请了发明专利！

2) 使用随机密码加密设备所有通信数据，防止数据窃听

3) 系统采用 4096 位密钥验证设备和用户身份，超银行级别，防止非法入侵

4) 系统采用分布式部署，避免遭受单点 DDOS 攻击

5) 设计了专属路由机制，即使互联网 DNS 遭受攻击也能确保系统设备可用

3．名词解释

本文档以下内容可能用到的一些名词：

1) assert。断言宏定义，如果表达式为假，则系统停机，并输出后面的字符串提示。仅调试版可用，发行版被编译器忽略。

2) debug_printf。调试输出宏定义，向调试口输出信息。仅调试版可用，发行版被编译器忽略。

3) 引脚。芯片对外的连接点，每个引脚都有自己的名字，一般是 PA0、PA1~PA15，PB0~PB15，……等。分成多组，每一组 16 个引脚。注意，芯片外部引脚并不一定从 PA0 顺序往后排，很可能中间缺了某一些引脚。

二、快速入门

大分类的描述性文字

1. HelloWord

内容 1

2. 跑马灯

内容 2

三、开发环境准备

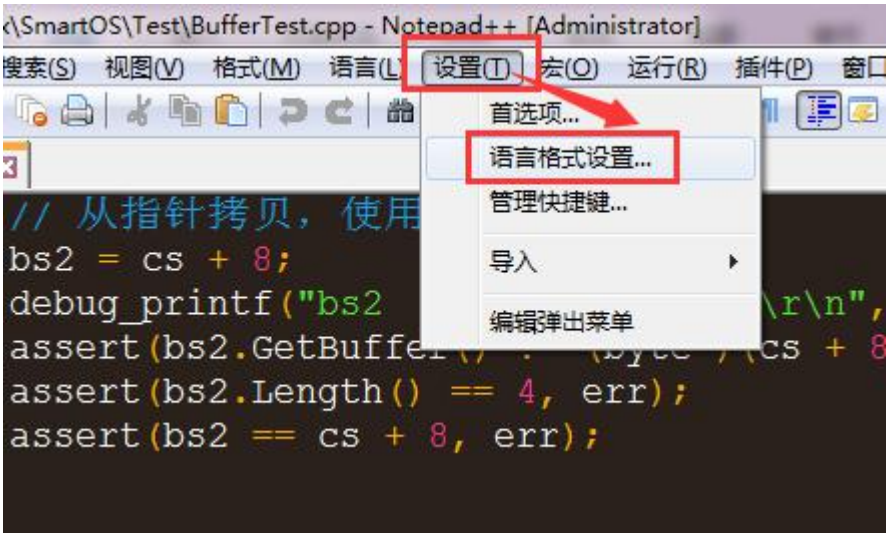
1. 安装 SDK

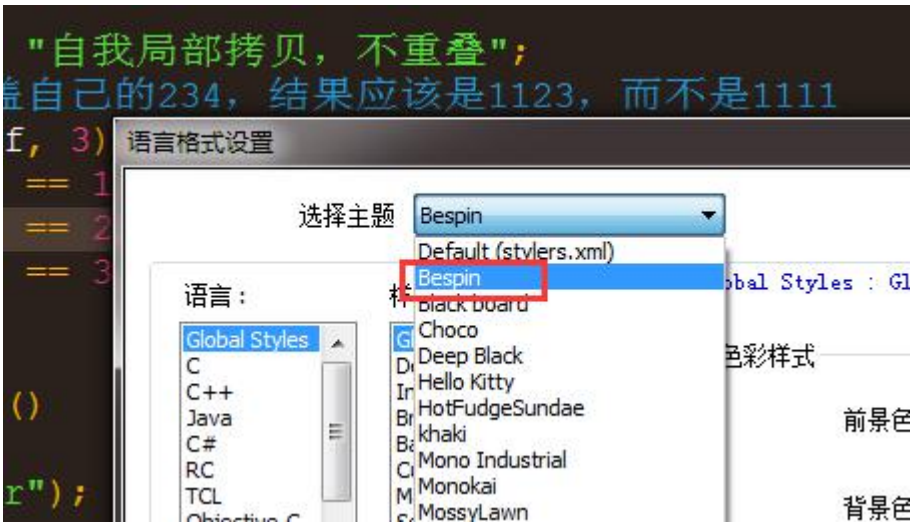
2. 安装编译工具

3. 编写工具

可以使用各种源代码编辑工具来编写程序，甚至记事本也可以。

这里强烈推荐 Notepad++，足够轻量级！设置一下样式更好看。菜单栏，选择“设置”，“语言格式设置”，弹出对话框中选择 Bespin 主题。也可以根据自己喜好选择其它主题。




"自我局部拷贝，不重叠";
自己的234，结果应该是1123，而不是1111
语言格式设置
选择主题
Bespin
Default (stylers.xml)
Bespin
Choco
Deep Black
Hello Kitty
HotFudgeSundae
khaki
Mono Industrial
Monokai
MossyLawn
语言：
Global Styles
C
C++
Java
C#
RC
TCL
色彩样式
前景色
背景色

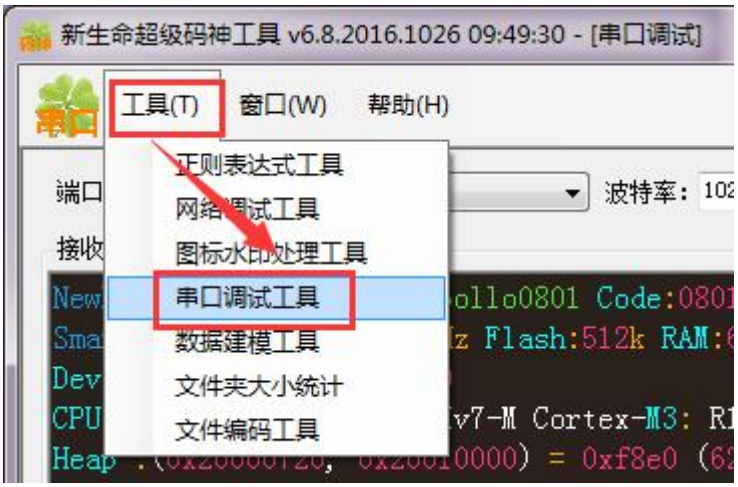
4. 串口调试工具

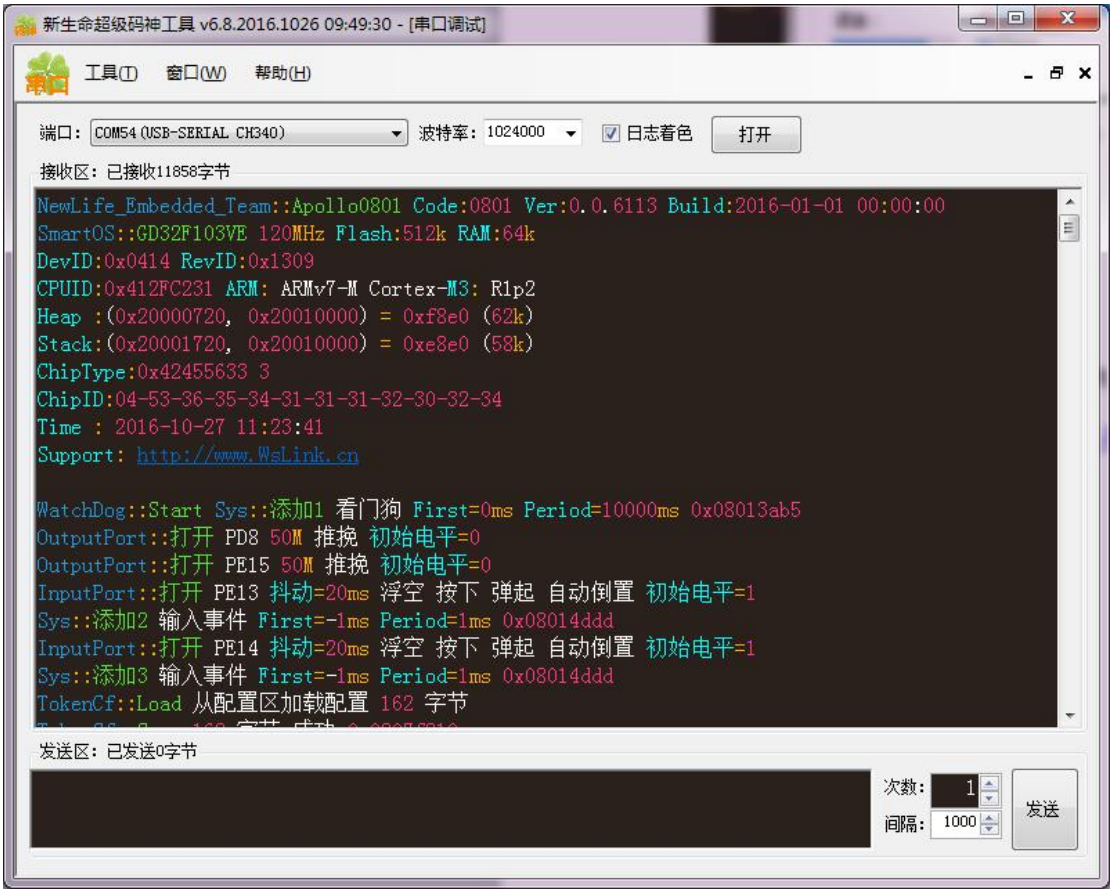
串口调试推荐使用新生命团队的码神工具，下载地址：

http://www.newlifex.com/showtopic-260.aspx

系统调试日志默认从板子的 COM1 输出，默认波特率 1024000。

*打开日志着色功能，效果更佳！



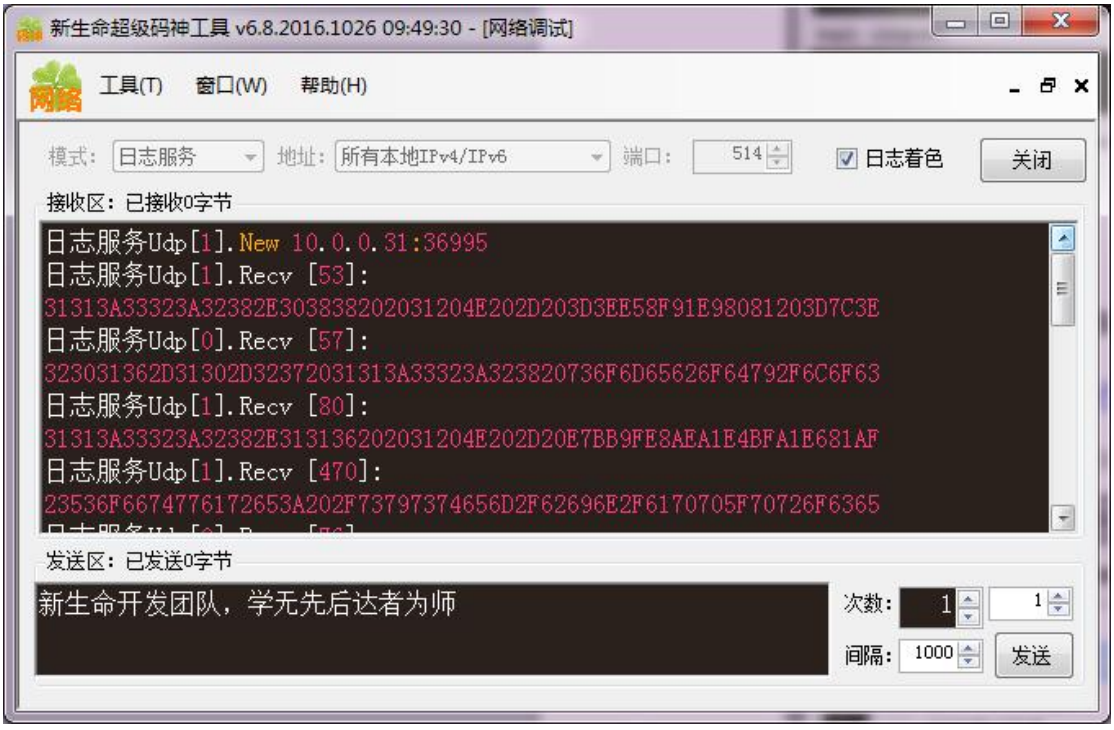


5. 网络调试工具

网络调试推荐使用新生命团队的码神工具，下载地址：

http://www.newlifex.com/showtopic-260.aspx

网络工具可以监听任意 Tcp/Udp/IPv4/IPv6 端口，也可以作为客户端连接任意远程端口，支持字符串和 HEX 格式显示数据。



四、基础类库

系统默认引用基础类库 Core 的大部分头文件，无需关注每个类型位于哪个头文件，直接使用即可。

1. 基本类型 Type

基本类型表，在 C++基础上尽可能参考 C#/Net。

类型	字节	范围	说明
byte	1	0~255	单字节
short	2		短整型
ushort	2	0~65535	无符号短整型
int	4		整型
uint	4		无符号整型
Int64	8		长整型
UInt64	8		无符号长整型
cstring	4		C 格式字符串

2. 根基类 Object

参考 C#的 Object 类，系统中大部分类都继承自 Object。

主要提供对象转字符串以及对象调试打印的抽象接口。

对象转字符串 ToStr/ToString 的默认实现时显示类名。

显示对象 Show 的默认实现是通过板子的调试串口输出。

```
// 根对象
class Object
{
public:
    // 输出对象的字符串表示方式
    virtual String& ToStr(String& str) const;
    // 输出对象的字符串表示方式。支持RVO优化
    virtual String ToString() const;
    // 显示对象。默认显示ToString
    virtual void Show(bool newLine = false) const;

    const Type GetType() const;
};
```

3．缓冲区 Buffer

Buffer 表示一个缓冲区，内部由一个指针和一个长度组成。

整个系统设计不允许传输单字节指针，以避免因处理不当而带来的内存泄漏或非法修改其它对象的数据。

常用操作：

1) 使用指针和长度构造一个缓冲区。GetBuffer 返回指针，Length 返回长度。

```
// 使用指针和长度构造一个内存区
char cs[] = "This is Buffer Test.";
Buffer bs(cs, sizeof(cs));
debug_printf("Buffer bs(cs, strlen(cs)) => %s\r\n", cs);
assert(bs.GetBuffer() == (byte*)cs, "GetBuffer()");
assert(bs == cs, "Buffer(void* p = nullptr, int len = 0)");
```

Buffer 可直接与指针进行相等比较。

2) 从源指针拷贝数据到缓冲区。赋值号是拷贝数据而不是替换内部指针。

以下例程，bs=bts 将会从 bts 拷贝 4 个字节到 bs，也就是 buf 所在内存区域。

如果源指针数据少于缓冲区长度，那么会在源指针数据后面接着拷贝其它数据。

```
byte buf[]  = { 1, 2, 3, 4 };
byte bts[]  = { 5, 6, 7, 8, 9, 10 };
Buffer bs(buf, sizeof(buf));

auto err    = "Buffer& operator = (const void* ptr)";

// 从指针拷贝，使用我的长度
bs  = bts;
assert(bs.GetBuffer() == buf, err);
assert(bs.GetBuffer() != bts, err);
assert(bs.Length() == sizeof(buf), err);
assert(buf[0] == bts[0] && buf[3] == bts[3], err);
```

3) 从缓冲区拷贝数据到另一个缓冲区。特别注意源缓冲区长度大于目标缓冲区的情况。

```
byte buf[]  = { 1, 2, 3, 4 };
byte bts[]  = { 5, 6, 7 };
Buffer bs(buf, sizeof(buf));
Buffer bs2(bts, sizeof(bts));

auto err    = "Buffer& operator = (const Buffer& rhs)";

// 从另一个对象拷贝数据和长度，长度不足且扩容失败时报错
// Buffer无法自动扩容，Arrar可以
//bs2   = bs;
bs  = bs2;
assert(bs.GetBuffer() == buf, err);
assert(bs.GetBuffer() != bts, err);
assert(bs.Length() == sizeof(bts), err);
assert(bs.Length() != sizeof(buf), err);
assert(buf[0] == bts[0] && buf[2] == bts[2], err);
assert(buf[3] == 4, err);
```

4) 改变缓冲区长度 SetLength。

SetLength 可以把缓冲区长度设置的更小。如果设置更大，则涉及自动扩容问题。

当源缓冲区长度大于目标缓冲区时，Buffer 本身会报错，但是 Buffer 的子类 Array/ByteArray/String 等会调用 SetLength 自动扩容，加大缓冲区。

5) 高级拷贝 Copy/CopyTo。

支持缓冲区和缓冲区、缓冲区和数据指针之间指定偏移和长度的高级拷贝。

特别注意，目标缓冲区长度不足时，参考 SetLength 扩容原则。

```
// 拷贝数据，默认-1长度表示当前长度
virtual int Copy(int destIndex, const void* src, int len);
// 把数据复制到目标缓冲区，默认-1长度表示当前长度
virtual int CopyTo(int srcIndex, void* dest, int len) const;
// 拷贝数据，默认-1长度表示两者最小长度
virtual int Copy(int destIndex, const Buffer& src, int srcIndex, int len);
// 从另一个对象拷贝数据和长度，长度不足且扩容失败时报错
int Copy(const Buffer& src, int destIndex = 0);
```

6) 截取子缓冲区 Sub

7) 转为十六进制字符串 ToHex

8) 缓冲区与整型数据互换 ToUInt32/Write

9) 二进制数据直接作为字符串 AsString()

4. 数组 Array

变长数据 Array 继承自缓冲区 Buffer，主要提供自动扩容功能。执行各种操作，存储空间不足时，自动申请内存扩容，对象销毁时自动释放内存。

如果使用空指针构造 Array，那么第一次使用时就会申请内存。

5. 字节数组 ByteArray

字节数组 ByteArray 继承自数组 Array，除了自动扩容功能，还内置了一个 64 字节的小型缓冲区。

在性能底下内存紧张的嵌入式场合，申请内存堆分配是一项比较耗时且不确

定时间的工作。ByteArray 内置小型缓冲区，伴随着字节数组对象一起栈分配，在超出 64 字节之前，无需向堆申请内存。

6. 字符串 String

字符串 String 继承自 Array，内部存储指针和长度。特别注意，内部会为结尾的\0 保留存储空间，以确保取得的字符串指针是标准 C 格式字符串指针。

参考 C#版 String 类，所有基本类型都可以转为字符串，支持字符串赋值拷贝、连接、相等比较、转数字、截取等。

1) 基本类型转为字符

```
// 默认空字符串，使用内部数据区
String str;
assert(str.Length() == 0, err);
assert(str.Capacity() == 0x40 - 1, err);

String str1("456");
assert(str1 == "456", err);
assert(str1.GetBuffer() == "456", err);

err = "String(const String& str)";
String str2(str1);
assert(str2 == str1, err);
assert(str2.GetBuffer() != str1.GetBuffer(), err);

char cs[] = "Hello Buffer";
String str4(cs, sizeof(cs));
assert(str4 == cs, "String(char* str, int length)");
assert(str4.GetBuffer() == cs, "String(char* str, in

String str5((char)'1');
assert(str5 == "1", "String(char c)");
```

3) 数字转为十进制字符串

```
debug_printf("10进制构造函数:.....\r\n");
String str1((byte)123, 10);
assert(str1 == "123", "String(byte value, int radix = 10)

String str2((short)4567, 10);
assert(str2 == "4567", "String(short value, int radix = 1

String str3((int)-88996677, 10);
assert(str3 == "-88996677", "String(int value, int radix

String str4((uint)0xFFFFFFFF, 10);
assert(str4 == "4294967295", "String(uint value, int radi

String str5((Int64)-7744, 10);
assert(str5 == "-7744", "String(Int64 value, int radix =

String str6((UInt64)331144, 10);
assert(str6 == "331144", "String(UInt64 value, int radix

// 默认2位小数，所以字符串要补零
String str7((float)123.0);
assert(str7 == "123.00", "String(float value, int decimal

// 浮点数格式化的时候，如果超过要求小数位数，则会四舍五入
String str8((double)456.784);
assert(str8 == "456.78", "String(double value, int decima

String str9((double)456.789);
assert(str9 == "456.79", "String(double value, int decima
```

5) 数字转为十六进制字符串

```
debug_printf("16进制构造函数:.....\r\n");
String str1((byte)0xA3, 16);
assert(str1 == "a3", "String(byte value, int ra
assert(String((byte)0xA3, -16) == "A3", "String

String str2((short)0x4567, 16);
assert(str2 == "4567", "String(short value, int

String str3((int)-0x7799, 16);
assert(str3 == "ffff8867", "String(int value, i

String str4((uint)0xFFFFFFFF, 16);
assert(str4 == "ffffffff", "String(uint value,

String str5((Int64)0x331144997AC45566, 16);
assert(str5 == "331144997ac45566", "String(Int6

String str6((UInt64)0x331144997AC45566, -16);
assert(str6 == "331144997AC45566", "String(UInt
```

6) 字符串到字符串拷贝为数据拷贝，内容相等，指针不同

```
debug_printf("赋值构造测试\r\n");

String str = "万家灯火，无声物联！";
debug_printf("TestAssign: %s\r\n", str.GetBuffer());
str = "无声物联";
assert(str == "无声物联", "String& operator = (cstri

String str2 = "xxx";
str2    = str;
assert(str == "无声物联", "String& operator = (cstri
assert(str2.GetBuffer() != str.GetBuffer(), "String&
```

7) 字符串连接支持基本类型。数字默认以十进制方式连接。

***特别注意，为了避免产生过多碎片对象，字符串连接实质上是在源字符串上累加，会改变源字符串。这一点与 C#/Java/C++的字符串类机制不同。

```
debug_printf("字符串连接测试\r\n");

auto now    = DateTime::Now();
//char cs[32];
//debug_printf("now: %d %s\r\n", now.Second,

String str;

// 连接时间，继承自Object
str += now;
str.Show(true);
// yyyy-MM-dd HH:mm:ss
assert(str.Length() == 19, "String& operator

// 连接其它字符串
int len = str.Length();
String str2(" 中国时间");
str += str2;
assert(str.Length() == len + str2.Length(), "

// 连接C格式字符串
str += " ";

// 连接整数
len = str.Length();
str += 1234;
assert(str.Length() == len + 4, "String& oper

// 连接C格式字符串
str += " ";

// 连接浮点数
len = str.Length();
str += -1234.8856;  // 特别注意，默认2位小数，
str.Show(true);
assert(str.Length() == len + 8, "String& oper
```

9) 字符串以十六进制连接数字

```
String str = "十六进制连接测试 ";

// 连接单个字节的十六进制
str.Concat((byte)0x20, 16);

// 连接整数的十六进制，前面补零
str += " @ ";
str.Concat((ushort)0xE3F, 16);

// 连接整数的十六进制（大写字母），前面补零
str += " # ";
str.Concat(0x73F88, -16);

str.Show(true);
// 十六进制连接测试 20 @ 0e3f # 00073F88
assert(str == "十六进制连接测试 20 @ 0e3f #
```

10) 十六进制字符串转为二进制数组

```
// 十六进制字符串转为二进制数组
str = "36-1f-36-35-34-3F-31-31-32-30-32-34";
auto bs2    = str.ToHex();
bs2.Show(true);
assert(bs2.Length() == 12, "ByteArray ToHex()");
assert(bs2[1] == 0x1F, "ByteArray ToHex()");
assert(bs2[5] == 0x3F, "ByteArray ToHex()");
```

11) 字符串搜索

```
// 字符串搜索
assert(str.IndexOf("36") == 0, "int IndexOf
assert(str.IndexOf("36", 1) == 6, "int Index
assert(str.LastIndexOf("36", 6) == 6, "int I
assert(str.LastIndexOf("36", 7) == -1, "int
assert(str.Contains("34-3F-31"), "bool Conta
assert(str.StartsWith("36-"), "bool StartsWi
assert(str.EndsWith("-32-34"), "bool EndsWit
```

13) 字符串截取

```
// 字符串截取
str = " 36-1f-36-35-34\n";
len = str.Length();
str = str.Trim();
assert(str.Length() == len - 2, "String& Trim()");

str = str.Substring(3, 5).ToUpper();
str.Show(true);
assert(str == "1F-36", "String Substring(int start,
```

14) 字符串分割 Split

受制于 C++语言特性，无法仿造 C#版本 String.Split，而采用枚举结构。

```
String str  = "+IPD,3,96,10.0.0.21,3377:abcdef";

auto err    = "StringSplit Split(cstring sep) const";

int p   = -1;
auto sp = str.Split(",");
assert(sp.Position == p && sp.Length == 0, err);
assert(sp, err);

auto rs = sp.Next();

p   = 0;

assert(sp.Position == p && sp.Length == 4, err);
assert(rs == "+IPD", err);
p   += 4 + 1;

rs  = sp.Next();
assert(sp.Position == p && sp.Length == 1, err);
assert(rs.ToInt() == 3, err);
p   += 1 + 1;

rs  = sp.Next();
assert(sp.Position == p && sp.Length == 2, err);
assert(rs.ToInt() == 96, err);
p   += 2 + 1;

rs  = sp.Next();
assert(sp.Position == p && sp.Length == rs.Length(), err);
assert(rs == "10.0.0.21", err);
p   += rs.Length() + 1;

// 更换分隔符
sp.Sep  = ":";
rs  = sp.Next();
assert(sp.Position == p && sp.Length == 4, err);
assert(rs.ToInt() == 3377, err);
p   += 4 + 1;

// 最后一组
rs  = sp.Next();
assert(sp.Position == p && sp.Length == rs.Length(), err);
assert(rs == "abcdef", err);
```

7. 数据流 Stream

数据流 Stream 参考 C#版，主要提供长度、容量、位置以及读写操作。

针对嵌入式环境，特别提供压缩编码整数读写和字节数组读写功能。

```
// 数据流容量
uint Capacity() const;
void SetCapacity(uint len);
// 当前位置
uint Position() const;
// 设置位置
bool SetPosition(int p);
// 余下的有效数据流长度。0表示已经到达终点
uint Remain() const;
// 尝试前后移动一段距离，返回成功或者失败。如果失败，不移动游标
bool Seek(int offset);

// 数据流指针。注意：扩容后指针会改变！
byte* GetBuffer() const;
// 数据流当前位置指针。注意：扩容后指针会改变！
byte* Current() const;

// 读取7位压缩编码整数
uint ReadEncodeInt();
// 读取数据到字节数组，由字节数组指定大小。不包含长度前缀
uint Read(Buffer& bs);

// 写入7位压缩编码整数
uint WriteEncodeInt(uint value);
// 把字节数组的数据写入到数据流。不包含长度前缀
bool Write(const Buffer& bs);

// 从数据流读取变长数据到字节数组。以压缩整数开头表示长度
uint ReadArray(Buffer& bs);
ByteArray ReadArray(int count);
// 把字节数组作为变长数据写入到数据流。以压缩整数开头表示长度
bool WriteArray(const Buffer& bs);
```

8. 时间日期 DateTime

DateTime 提供 C#版类似功能，但用法有所不同。

年月日、时分秒、毫秒，分字段存储。构造函数传入秒数时，马上开始分割为不同字段。这里的总秒数和总毫秒数为 Unix 格式，1970-01-01 以来总数。

DateTime::Now() 表示系统当前时间。

```
// 时间日期
class DateTime : public Object
{
public:
    ushort  Year;
    byte    Month;
    byte    Day;
    byte    Hour;
    byte    Minute;
    byte    Second;
    ushort  Ms;

    DateTime();
    DateTime(ushort year, byte month, byte day);
    DateTime(uint seconds);
    DateTime(const DateTime& value);
    DateTime(DateTime&& value);

    // 重载等号运算符
    DateTime& operator=(uint seconds);
    DateTime& operator=(const DateTime& value);

    DateTime& Parse(uint seconds);
    DateTime& ParseMs(UInt64 ms);
    DateTime& ParseDays(uint days);

    uint TotalDays() const;
    uint TotalSeconds() const;
    UInt64 TotalMs() const;

    // 获取星期
    byte DayOfWeek() const;
    // 取时间日期的日期部分
    DateTime Date() const;
```

9. 时间间隔 TimeSpan

时间间隔 TimeSpan 表示一个时间段，主要来源于两个时间相减，精度为

毫秒。

10. 列表 List

列表 IList 是一个以 4 字节为单元动态存储数据的列表结构，常用于存储指针。功能参考 C#版 List，支持添加、删除、个数、查找、索引取值赋值。

模版 List<T>继承 IList，T 为元素类型，必须是 4 字节长度。T 常常是具体类型指针。

受限于 C++语言特性 ,无法完全实现 C#版 List<T> ,故实现了一个指针版。

```
//不同长度的原始数据
byte buf1[] = {1,2,3,4,5};
byte buf2[] = {6,7,8,9};
byte buf3[] = {10,11,12,13,14,15,16,17,18,19,20};

IList list;
list.Add(buf1);
list.Add(buf2);
list.Add(buf3);

assert(list.Count() == 3, "Count()");
assert(list[0] == buf1 && list[1] == buf2 && list[2] =

// 添加
list.Add(buf2);
list.Add(buf3);
assert(list.Count() == 5, "Count()");
assert(list[3] == buf2 && list[4] == buf3, "void Add(v

// 查找
int idx = list.FindIndex(buf2);
assert(idx == 1, "int FindIndex(const void* item)");

// 删除倒数第二个。后面前移
list.RemoveAt(list.Count());     // 无效
list.RemoveAt(list.Count() - 2);
assert(list.Count() == 4, "Count()");
assert(list[3] == buf3, "void RemoveAt(uint index)");

// 删除具体项。后面前移
list.Remove(buf2);
assert(list.Count() == 3, "Count()");
assert(list[1] == buf3 && list[2] == buf3, "void Remov

// 删除具体项。找不到的情况
list.Remove(buf2);
assert(list.Count() == 3, "Count()");

// 查找。找不到的情况
idx = list.FindIndex(buf2);
assert(idx == -1, "int FindIndex(const void* item)");
```

11. 字典 Dictionary

字典 IDictionary 是一个以 4 字节为单元动态存储数据的字典结构，常用于存储指针。功能参考 C#版 Dictionary，支持添加、删除、个数、查找、索引取值赋值。

模版 Dictionary<TKey, TValue>继承 IDictionary，TKey/TValue 为元素类型，必须是 4 字节长度。TKey/TValue 常常是具体类型指针。

受限于 C++语言特性，无法完全实现 C#版 Dictionary<TKey, TValue>，故实现了一个指针版。

```
//不同长度的原始数据
byte buf1[] = {1,2,3,4,5};
byte buf2[] = {6,7,8,9};
byte buf3[] = {10,11,12,13,14,15,16,17,18,19,20};

IDictionary dic;
dic.Add("buf1", buf1);
dic.Add("buf2", buf2);
dic.Add("buf3", buf3);

auto err    = "void* operator[](const void* key) c
assert(dic.Count() == 3, err);
assert(dic["buf1"] == buf1 && dic["buf2"] == buf2

// 赋值
dic["buf2"] = buf3;
dic["buf3"] = buf2;
err = "void*& operator[](const void* key)";
assert(dic.Count() == 3, err);
assert(dic["buf2"] == buf3 && dic["buf3"] == buf2,
```

```
// 同名覆盖
err = "void Add(const void* key, void* value)";
dic.Add("buf2", buf2);
assert(dic.Count() == 3, err);
assert(dic["buf2"] == buf2 && dic["buf3"] == buf2,
dic["buf2"] = buf3;

// 查找
bool rs = dic.ContainKey("buf2");
err = "bool ContainKey(const void* key)";
assert(rs, err);
rs  = dic.ContainKey("buf");
assert(!rs, err);

// 删除倒数第二个。后面前移
dic.Remove("buf2"); // 无效
err = "void RemoveKey(const void* key)";
assert(dic.Count() == 2, err);
assert(dic["buf2"] == nullptr, err);

// 尝试获取值
void* p = nullptr;
rs  = dic.TryGetValue("buf3", p);
err = "bool TryGetValue(const void* key, void*& va
assert(dic.Count() == 2, err);
assert(rs, err);
// 前面曾经赋值，所以buf3里面保存的是buf2
assert(p == buf2, err);

// 测试比较器
cstring str = "123456";
IDictionary dic2(String::Compare);
dic2.Add("test", (void*)str);

char cs[5];
cs[0]   = 't';
cs[1]   = 'e';
cs[2]   = 's';
cs[3]   = 't';
cs[4]   = '\0';
rs  = dic2.TryGetValue(cs, p);

err = "IDictionary(IComparer comparer = nullptr)";
assert(rs, err);
// 前面曾经赋值，所以buf3里面保存的是buf2
assert(p == str, err);
```

12. 委托 Delegate

受限于 C++语言特性，无法实现 C#中的委托和事件。

本系统提供了一组常用函数指针以及几个高级委托模版

```
// 没有参数和返回值的委托
typedef void (*Func)(void);
// 一个参数没有返回值的委托，一般param参数用作目标对象，调用者用静态函数包装成
typedef void (*Action)(void* param);
typedef void(*Action2)(void*, void*);
typedef void(*Action3)(void*, void*, void*);
// 事件处理委托，一般sender表示事件发出者，param用作目标对象，调用者用静态函数
typedef void (*EventHandler)(void* sender, void* param);
// 传入数据缓冲区地址和长度，如有反馈，仍使用该缓冲区，返回数据长度
typedef uint (*DataHandler)(void* sender, byte* buf, uint size, void* param);
```

Delegate 模板类根据参数不同共有 3 个版本，Delegate/Delegate2/Delegate3。

比如 Button 类增加一个 Press 事件，指明事件函数只有一个参数。

```
class Button : public Object, public ByteDataPort
{
public:
    cstring Name;    // 按钮名称
    int     Index;       // 索引号，方便在众多按钮中

    InputPort   Key;     // 输入按键
    OutputPort  Led;     // 指示灯
    OutputPort  Relay;   // 继电器

    Delegate<Button&>   Press;  // 按下事件
```

Button 类内部触发该事件时，使用 Press(*this)。

1) 外部绑定全局函数

```
void OnPress(Button& btn)
{
    debug_printf("按下 %c \r\n", btn.Name);
}

void Test()
{
    Button btn;
    btn.Name    = "客厅";
    btn.Press   = OnPress;
}
```

2) 外部绑定成员函数

```
void TouchSwitch::OnPress(Button& btn)
{
    debug_printf("按下 %c \r\n", btn.Name);
}

void TouchSwitch::Test()
{
    Button btn;
    btn.Name    = "客厅";
    btn.Press.Bind(&TouchSwitch::OnPress, this);
}
```

绑定成员函数的用法相对复杂，需要特别小心。

13. 队列 Queue

队列 Queue 为先进先出 FIFO 结构，单字节存储，主要提供 Push/Pop 功能。

14. 随机数 Random

随机数默认以当前时间为种子，主要提供 Next/Next(max)等产生随机数的功能。

五、系统内核

SmartOS 操作系统内核主要提供处理器管理、内存管理、时间管理、中断管理以及任务调度等。

1. 系统核心 Sys

Sys 是系统类 TSys 的全局对象，上层可以直接使用。

（1）系统启动

因为是全局对象，Sys 对象构造会在 main 函数之前完成。此时获取芯片信息填充 Sys 对象的各个字段，并完成内存堆栈设置以及中断向量表设置。此过程用户不可干涉。

（2）系统初始化

系统初始化 Sys.Init 主要配置系统主时钟以及初始化系统时间。

STM32/GD32 系列芯片已经按照默认主频配置系统主时钟，如果晶振不是 8M，或者需要超频降频，可以在初始化之前修改 Sys.Clock。

默认调试口为 COM1，也可以在初始化之前修改 Sys.MessagePort 为其它端口。

```
int main(void)
{
    // 初始化系统
    //Sys.Clock = 72000000;
    //Sys.MessagePort = COM1;
    Sys.Init();
    Sys.ShowInfo();
```

（3）启动日志

Sys.ShowInfo 输出系统启动信息，调试版自动忽略。

系统能自动识别常见芯片类型、主频、Flash 大小、Ram 大小。

Sys.ID 是 12 字节芯片唯一标识，也就是上图的 ChipID，同一批芯片仅前面几个字节不同。

（4）系统时间

Sys.Ms() 系统启动以来的毫秒数，无符号长整型 8 字节。

Sys.Seconds() 系统绝对 UTC 时间，整型 4 字节，Unix 格式，1970 年以来的总秒数。

（5）延迟睡眠

Sys.Delay(us) 微秒级延迟，常用于高精度外设信号控制。

Sys.Sleep(ms) 毫秒级睡眠，常用于业务层暂停等待一定时间。

毫秒级睡眠期间，系统将会安排执行其它耗时较短的任务。如果没有可用任务，系统将会进入低功耗模式，以节省能耗！

（6）重启

Sys.Reboot(msDelay) 异步热重启系统。延迟一定毫秒数执行。

（7）系统主调度

入口 main 函数最后，一定是 Sys.Start，启动系统任务调度，该函数内部为死循环。

*在此之间，添加的所有任务函数将得不到调度，所有睡眠方法无效！

2．系统时间 Time

Time 是系统时间 TTime 类的全局对象，不建议用户层直接使用。

Time.Current 当前毫秒数。同 Sys.Ms。

Time.CurrentTicks 嘀嗒数。比微秒还短的超高精度时间。

Time.SetTime(UInt64 seconds) 设置系统当前时间，Unix 格式秒数。

3．时间轮 TimeWheel

时间轮用于在指定超时时间内，循环执行指定代码逻辑。

```
TimeWheel tw(1000);
tw.Sleep    = 50;
while(!tw.Expired())
{
    // 检查是否有数据到来
    CheckReceive();
}
```

如上图，在 1000ms 内，多次调用 CheckReceive，间隔睡眠 50ms，通过 Sys.Sleep(50)实现，允许调度其它较小耗时的任务。

4. 时间开销 TimeCost

TimeCost 用于统计某一段代码逻辑的执行时间，精度为微秒。

Show 方法可以直接调试输出结果。

```
TimeCost tc;

// 初始化为输出
OutputPort leds[] = {PD0, PD1};
for(int i=0; i<ArrayLength(leds); i++)
    leds[i].Invert = true;

// 耗时 tc.Elapsed() 微秒
tc.Show();  // 显示: 执行 xxx us
```

5. 任务调度 Task

系统默认使用协作式调度，支持添加无限多个任务。设计上类似于 C#上的定时器 Timer。

Sys.AddTask 添加任务，参数分别是：任务函数、参数、首次时间、间隔时间、名称。返回值是一个 uint 的任务唯一编号。

Sys.Remove(taskid) 删除任务。

如下图，添加一个“守护 2860”的任务，500ms 后执行，每隔 500ms 执行一次。

```
_ResetTask = Sys.AddTask(ResetTask, this, 500, 500, "守护2860");
```

如下图，添加一个单次任务，30ms 后执行一次，然后自动销毁

```
Sys.AddTask(LoopOpenTask, this, 30, -1, "Open8266");
```

如下图，添加一个事件型任务，该任务不会被系统调度，默认处于禁用状态，

常用于就绪队列，通过 Sys.SetTask(taskid, true)唤醒它马上执行一次，系统在执行完当前任务后将会尽快调度。或者 Sys.SetTask(taskid, true, 20)要求它延迟 20ms 执行，该设计常用于连续多次数据到来时避免频繁调用数据处理。

```
Sys.AddTask(InputTask, this, -1, 1, "输入事件");
```

6. 中断管理 Interrupt

中断管理仅用于内部硬件驱动移植，对用户不公开。

SmartIRQ 类用于关闭打开全局中断。

```
{
    // 进入区域，关闭全局硬件中断
    SmartIRQ irq;

    queue.Push(bt);
    // 离开区域，自动打开全局硬件中断
}
```

7. 等待句柄 WaitHandle

等待句柄类似于 C#版本，主要用于多线程同步（多任务）。

如下图，当前任务最大等待 3000ms，除非其它任务把它的 Result 设置为 true，它才会提前结束 WaitOne。

该用法比时间轮 TimeWheel 效果更好。

```
WaitHandle handle;
handle.WaitOne(3000);
// handle.Result = true;
```

六、外设驱动

引脚 PA0~PA15、PB0~PB15 等，每一个引脚用两个字节存储，P0 为 0xFFFF，表示非法引脚，PA0 为 0。

1. 输出口 OutputPort

数字端口输出控制高低电平。表现在 LED 小灯上就是亮灭，表现在继电器上就是接通和断开。

```
OutputPort led(PD0);
led.Open();
led = true;
led.Write(false);
```

上图使用引脚 PD0 定义输出口对象 led，向 led 赋值 true 表示让 PD0 引脚输出高电平，同理 false 为低电平。也可以使用 Write 方法控制，效果相同。

因电路设计需要，有时候输出低电平为有效，为了逻辑一致性，我们需要让输出口逻辑倒过来，初始化时需要指定第二参数 invert 为 true。

```
OutputPort led(PD0, true);

OutputPort led;
led.Init(PD0, true);
```

如上图两种方法都可以定义倒置的输出口对象。此时 led=true 表示让 PD0 输出低电平。

默认情况下，第二参数 invert 为 2，表示自动识别倒置。初始化输出口对象时，如果该输出口处于高电平状态，则需要倒置，否则不需要倒置。按照标准设计的电路，加有上拉下拉，都会有明确的初始高低电平，因此输出口都能自动识别倒置情况。所以，大多数情况下，并不需要明确指定是否需要倒置，而是让其

自动识别。

2. 输入口 InputPort

输入口基本功能是 Read 读取外部高低电平状态，同样具有自动倒置判断功能。

输入口最常用功能是设置一个事件，当输入口收到高低电平跳变（例如按钮按下和弹起）时，除法该事件。

```
// 按键事件
void OnPress(InputPort& port, bool down)
{
    debug_printf("Press P%c%d down=%d\r\n", _PIN_NAME(port._Pin), down);
}

InputPort key(PE13);
key.Press   = OnPress
key.UsePress();
key.Open();
```

上图是一个标准的输入口事件例程，启动 down 参数指示当前动作是按下还是弹起。

触发弹起事件时，还可以读取输入口的 PressTime 得到这一次按下弹起的时长（ms），用于设计单独按钮长按多少秒就执行专门动作。

3. 复用输出 AlternatePort

复用输出继承自输出口 OutputPort，该端口不仅可以控制输出高低电平，也可以通过 ReadInput 读取输入电平。

4. 模拟量 AnalogInPort

模拟量输入口，由 ADC 调用，用户不直接使用，详见 ADConverter

5. 串口 SerialPort

通过指定 COM1/COM2/COMn 和波特率来构造串口对象，Register 注册一个异步接收委托，当串口收到数据时调用该委托函数。

Write 直接发送 Buffer 内存区，String 继承自 Buffer，所以可以直接使用。

```
uint OnUsartRead(ITransport* transport, Buffer& bs)
{
    debug_printf("收到: ");
    bs.Show(true);

    // 原路发回去
    transport->Write(bs);

    return 0;
}

SerialPort* sp1;
void TestSerial()
{
    debug_printf("\r\n\r\n");
    debug_printf("TestSerial Start.....\r\n");

    // 串口输入
    sp1 = new SerialPort(COM1, 1024000);
    sp1->Register(OnUsartRead);
    sp1->Open();

    String str = "http://www.NewLifeX.com \r\n";
    sp1->Write(str);
```

6. 串行外设接口 Spi

Spi 常用于操作外部 Flash 芯片、无线芯片、以太网芯片以及各种传感器芯片等。

常用 Spi 口有 Spi1/Spi2/Spi3 等，第二参数指定目标芯片最大速度，Spi

类内部将根据自己能力计算一个最接近的实际可用频率。

实际使用 Spi 时，需要先 Start 开启事务，然后 Read/Write 操作，最后 Stop。

系统内封装了 SpiScope 类，可自动打开关闭事务，如下：

```
    Spi _spi(Spi2, 10000000, true);

// 擦除扇区
bool AT45DB::Erase(uint sector)
{
    SpiScope sc(_spi);
    // 扇区擦除命令
    _spi->Write(0x7C);
    SetAddr(sector);

    return WaitForEnd();
}
```

7. 串行总线 I2C

I2C 类有 HardI2C/SoftI2C 两个子类实现。实际使用时可以把使用 I2C 指针，然后外部装配使用硬件 I2C 还是软件 I2C。

比如以下是环境探测器的光照传感器 BH1750 例程。

```
SoftI2C iic;
iic.SetPin(PB0, PB1);
```

```
BH1750::BH1750()
{
    IIC     = nullptr;

    // 7位地址，到了I2C那里，需要左移1位
    Address = 0x5C;
}

void BH1750::Init()
{
    debug_printf("\r\nBH1750::Init Addr=0x%02X \r\n", Address);

    IIC->Address = Address << 1;

    Write(CMD_PWN_ON);  // 打开电源
    Write(CMD_RESET);   // 软重启
    Write(CMD_HRES);    // 设置为高精度模式
}

ushort BH1750::Read()
{
    if(!IIC) return 0;

    ByteArray bs(2);
    if(IIC->Read(0, bs) == 0) return 0;

    ushort n = (bs[0] << 8) | bs[1];

    return n;
}

void BH1750::Write(byte cmd)
{
    if(!IIC) return;

    if(!IIC->Write(0, cmd)) debug_printf("BH1750::Write 0x%02X 失
}
```

七、网络开发

1. 网络主机 ISocketHost

2. 网络接口 ISocket

3. Tcp 通信

4. Udp 通信

5. DHCP

6. DNS

八、数据存储

1. 配置管理 Config

九、附录

1. 关于我们

无声物联是一个拥有多项核心专利技术的物联网品牌。研发团队拥有 10 多年软硬件开发经验，自主研发了物联网操作系统 SmartOS 和云平台 SmartCloud，并在有线组网和无线组网上申请了多项发明专利。

产品主要方向：灯光控制、智能传感、入侵检测、消防安全、工业控制、智能农业

电话：+86-0769-23107897

邮箱：Smart@wslink.cn

网站：http://www.wslink.cn

地址：广东省东莞市高埗镇广场北路宝源工业园