USTC

Chapter 25
Embedded systems programming

王子磊 (Zilei Wang)
Email: zlwang@ustc.edu.cn
http://vim.ustc.edu.cn

Abstract

- 本章将介绍嵌入式系统程序设计及其与“一般编程”的区别之处，这会接触一些“接近硬件”时的突出问题，如自由存储使用、位操作、编码标准等
  - 不是所有的计算机都是办公室桌面下面的小灰盒子
- Contents
  - 嵌入式系统
  - 与普通系统不同的特别之处
    - 可预测性
  - 资源管理
    - 内存
  - 访问硬件
    - 绝对地址
    - 位
  - 编码标准

嵌入式系统

- 硬实时
  - 在时限之前必须有响应
- 软实时
  - 大部分时间下，时限之前能够响应
- 通常有诸多资源来处理一般情况
  - 但危机/故障还会出现，且必须要处理
- 可预测性是关键(实时性决定的)
- 正确性比普通程序更为重要
  - “正确性”不是一个抽象概念
  - “我假定硬件正常工作”不再是一个借口
    - 长时间在各种复杂的条件下，它不是简单的

嵌入式系统

- 计算机被用作大型系统的一部分
  - 它们看起来一般不像一个计算机
  - 通常是控制一个物理设备
- 通常，可靠性是至关重要的
  - “至关重要”表示“如果系统故障，会死人的”
- 通常，资源(内存、处理器能力)是有限的
- 通常，实时响应是重要的

嵌入式系统

- 我们正在讨论的有什么？
  - 装配线质量监视器
  - 条形码阅读器
  - 面包机
  - 照相机
  - 汽车组装机器人
  - 手机
  - 离心机控制器
  - CD 播放机
  - 磁盘驱动控制器
  - “智能卡”处理器
  - 燃料喷射控制器
  - 医用设备监视器
  - PDA
  - 打印机控制器
  - 音响系统
  - 电饭煲
  - 电话交换机
  - 水泡控制器
  - 焊接机
  - 风车
  - 腕表
  - …

你需要知道所有这些吗?

- 计算机工程师 – 你会建立并监督这些系统的构造
  - 所有“接近硬件”的代码都像这样
  - 对嵌入式系统代码来说，对正确性和可预测性的关注是构造所有代码思想中最重要的
- 电子工程师 – 你会建立并监督这些系统的构造
  - 你不得不与计算机工程师一起工作
  - 你不得不能够与他们交谈
  - 你不能不教他们
  - 你不得不陪着他们
- 计算机科学家 – 你要知道如何做这些，或者你仅仅工作在web应用上

可预测性

- C++ 操作具有良好的可预见性，即具有固定可测的执行时间
  - 比如，你能够简单地测量加法或虚函数调用所需的时间，且每个加法操作和虚函数调用都是一样的 (在现代处理器上，管道、缓存、隐式并发可能使它们稍微有些不同)
  - 除了：
    - 自由存储分配 **new**
    - 异常 **throw**
- 因此，典型情况下，**throw** 和 **new** 在硬实时应用中是禁止的
  - 目前，我们还不能启用使用这些特性的飞机 ☺
    - 五年后，我们将解决 **throw** 的这一问题
      - 每个独立的 **throw** 是可预测的
  - 不仅仅对C++程序如此
    - 在其它语言中，相似的操作也应该是避免的

理想/目标

- 给定约束条件
  - 保持高层抽象
    - 不用写完美的汇编代码
    - 直接用代码表达你的思想
  - 总是，尽量编写清晰、干净和最容易维护的代码
  - 必要时再进行优化
    - 人们通常在成形之前过早地进行优化
    - John Bentley 的优化法则
      - 第一条：不要做优化
      - 第二条(仅对专家里手)：还是不要做优化

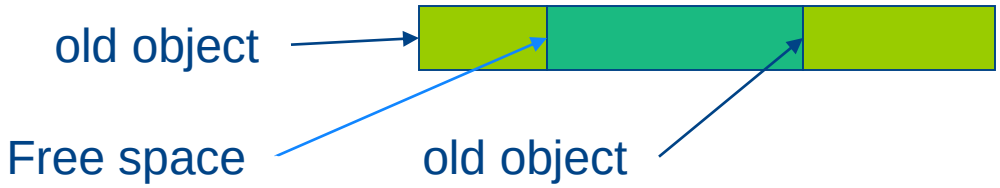
嵌入式系统程序设计

- 与普通程序相比，你在嵌入式系统编程中通常需要更多地注意资源消耗
  - 时间
  - 空间
  - 信道
  - 文件
  - ROM
  - Flash 存储
  - …
- 你不得不花费时间去学会你使用的语言特性在特定平台上是如何实现的
  - 硬件
  - 操作系统
  - 库

嵌入式系统程序设计

- 这类编程问题可能有：
  - 查看一个实时操作系统的某个特性
    - RTOS (Real Time Operating System)
  - 使用“非主机环境”(语言直接应用到硬件平台上，而没有操作系统)
  - 涉及一些复杂的设备驱动框架
  - 直接处理硬件设备接口
  - …
- 这里，我们不会深究这样的细节
  - 这些在特定的课程或手册中介绍

How to live without new



- 问题是什么？
  - C++ 代码直接引用内存
    - 对象一旦分配好，就不能再移动 (可以吗？)
  - 分配时延 — 可预测性
    - 获取给定大小空内存块所需的努力 (时间) 依赖于那些已经分配好的对象
  - 碎片
    - 如果你有一个大小为 N 的内存块，你可以分配一个大小为 M (M <N) 的对象，你将不得不再处理 N-M 大小的一个碎片
    - 一段时间后，那样的碎片将占据大部分内存空间
      - 整个内存空间由碎片构成

How to live without new

- 解决方案：预分配
  - 全局对象
    - 在启动时分配
      - 设置好一个固定大小的内存
  - 栈 Stacks
    - 只在头部增长和缩短
      - 没有碎片
      - 定长时间的操作
  - 固定大小对象的内存池 Pools
    - 我们能够分配和释放
      - 没有碎片
      - 定长时间的操作

How to live without new

- 不用 new
  - 也不用 malloc()
- 不用标准库容器
  - 它们间接使用了自由存储空间
- 代替方法
  - 定义一个固定大小的内存池
  - 定义一个固定大小的栈
- 不要倒退到使用数组和(多)指针
  - 会引起一些难以察觉的错误 25.4

内存池实例

```
// Note: element type known at compile time
// allocation times are completely predictable (and short)
// the user has to pre-calculate the maximum number of elements needed
template<class T, int N>class Pool {
public:
    Pool();                 // make pool of N Ts – construct pools only during startup
    T* get();               // get a T from the pool; return 0 if no free Ts
    void free(T*);          // return a T given out by get() to the pool
private:
    // keep track of T[N] array (e.g., a list of free objects)
};

Pool<Small_buffer,10> sb_pool;
Pool<Status_indicator,200> indicator_pool;
```

栈实例

```
// Note: allocation times completely predictable (and short)
// the user has to pre-calculate the maximum number of elements needed
template<int N>class Stack {
public:
    Stack();            // make an N byte stack – construct stacks only during startup
    void* get(int n);   // allocate n bytes from the stack;  return 0 if no free space
    void free(void* p);         // return the last block returned by get() to the stack
private:
    // keep track of an array of N bytes (e.g. a top of stack pointer)
};

Stack<50*1024> my_free_store;       // 50K worth of storage to be used as a stack

void* pv1 = my_free_store.get(1024);
int* pi = static_cast<int*>(pv1);       // you have to convert memory to objects

void* pv2 = my_free_store.get(50);
Pump_driver* pdriver = static_cast<Pump_driver*>(pv2);
```

需要进行强制类型转换以支持多种类型

释放时顺序必须对应

模板

- 对嵌入式系统能够完美的工作
  - 内联操作没有运行时负担
    - 有时候，性能很重要
  - 对没有使用的操作不消耗内存
    - 在嵌入式系统中，内存通常是非常有限的

硬件故障如何处理

- 什么故障？
  - 通常我们不知道
  - 在实际中，我们可以假定一些故障比其他的更常见
    - 不过，有时候一个内存位就会引起系统变化
- 原因？
  - 功率骤变/电源故障
  - 插头从插座上脱落
  - 落下的碎片集中系统
  - 系统坠落
  - X - 射线
  - …
- 瞬时故障是最麻烦的
  - *e.g.*, 只有在温度超过 100° F 且厨门关闭时才发生
- 在实验室之外发生的错误是难以修复的
  - *e.g.*, 在火星上

硬件故障如何处理

- 备份
  - 紧急情况下，使用热备
- 自检测
  - 了解系统(或硬件)什么时候工作异常
- 一些原理：处理异常代码的快速方式
  - 让系统模块化
  - 由其它模块、计算机或系统的一部分负责严重错误
    - 最终，可能是一个人(手动处理)
- 监视子系统
  - 这种情况下，它们不能查看自身出现的问题

绝对地址

- 典型情况下，物理资源(如外部设备的控制寄存器)及其最基础的控制软件驻留在底层系统的特定地址处
- 我们可能需要访问那样的地址并显式指定一种类型
- 例如：

```
Device_driver* p = reinterpret_cast<Device_driver*>(0xffb8);
Serial_port_base *COM1 =
    reinterpret_cast<Serial_port_base*>(0x3f8);
```

位操作：无符号整数

- 在C++中如何表示一组二进制位？
  - **unsigned char uc;** *// 8 bits*
  - **unsigned short us;** *// typically 16 bits*
  - **unsigned int ui;** *// typically 16 bits or 32 bits*
    // (check before using)
    // many embedded systems have 16-bit ints
  - **unsigned long int ul;** *// typically 32 bits or 64 bits*
  - **std::vector<bool> vb(93);** *// 93 bits*
    - 只有需要超过32 bits 才使用
  - **std::bitset bs(314);** *// 314 bits*
    - 只有需要超过32 bits 才使用
    - 一般对 **sizeof(int)** 的倍数是高效的

位操作

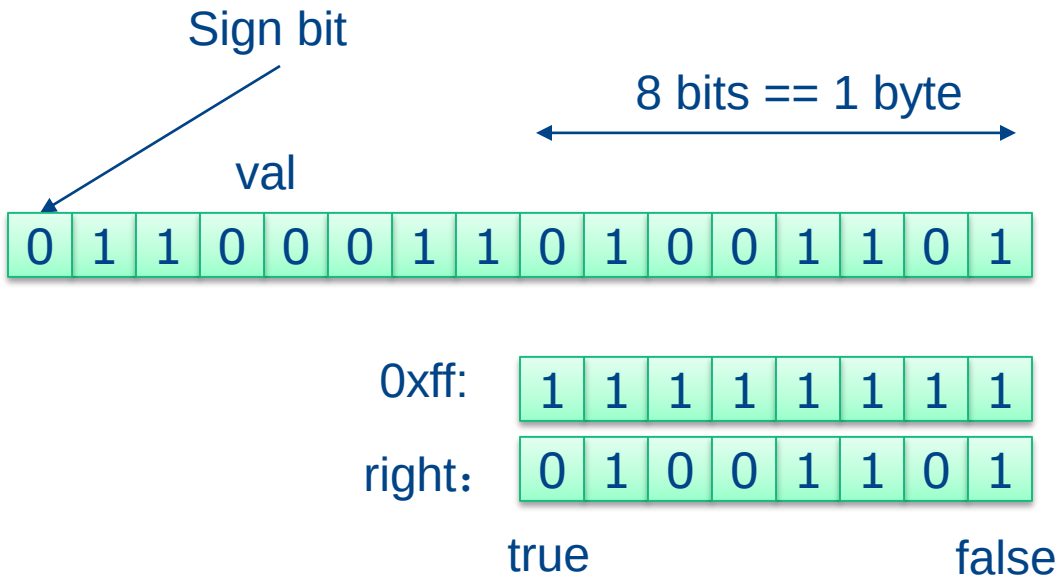
- & 与
- | 或
- ^ 异或
- << 左移
- >> 右移
- ~ 补(非)

a:	1	0	1	0	1	0	1	0	0xaa
b:	0	0	0	0	1	1	1	1	0x0f
a&b:	0	0	0	0	1	0	1	0	0x0a
a|b:	1	0	1	0	1	1	1	1	0xaf
a^b:	1	0	1	0	0	1	0	1	0xa5
a<<1:	0	1	0	1	0	1	0	0	0x54
b>>2:	0	0	0	0	0	0	1	1	0x03
~b:	1	1	1	1	0	0	0	0	0xf0

位操作

位操作

& (and)
| (or)
^ (exclusive or – xor)
<< (left shift)
>> (right shift)
~ (one's complement)
基本上，它们是硬件提供的

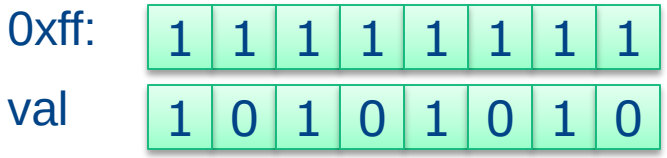


例如：

```
void f(unsigned short val)      // assume 16-bit, 2-byte short integer
{
    unsigned char right = val & 0xff ;       // rightmost (least significant) byte
    unsigned char left = (val>>8) & 0xff ; // leftmost (most significant) byte
    bool negative = val & 0x8000 ;           // sign bit (if 2's complement)
    // …
}
```

位操作

- 或 |
  - 设置一位
- 与 &
  - 该位是否设置了？选择一些位（掩码）



- 例如：

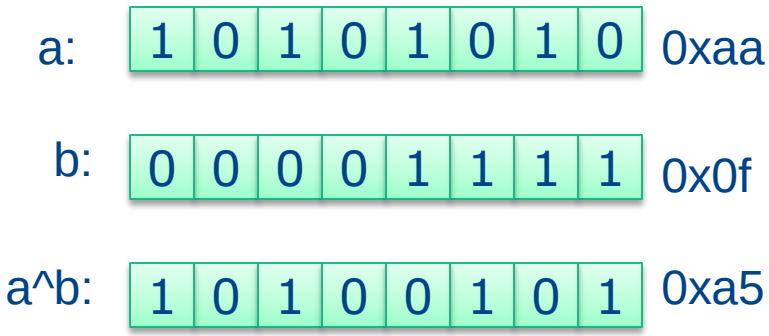
```
enum Flags { bit4=1<<4, bit3=1<<3, bit2=1<<2, bit1=1<<1, bit0=1 };
unsigned char x = bit3 | bit1; // x becomes 8+2
x |= bit2;                     // x becomes 8+4+2
if (x&bit3) {                  // is bit3 set? (yes, it is)
    // …
}
unsigned char y = x &(bit4|bit2);       // y becomes 4
Flags z = Flags(bit2|bit0); // the cast is necessary because the compiler
                            // doesn't know that 5 is in the Flags range
```

位操作

- 异或 (xor) ^
  - a^b 表示 (a|b) & !(a&b)
    - “要么a，要么b，但不能两者都是”

```
unsigned char a = 0xaa;
unsigned char b = 0x0f;
unsigned char c = a^b;
```

- 在图形学和加密学中非常重要



无符号整数

- 你能在无符号整数上做普通算术运算
  - 你应该尽量避免
    - 不要为了多出一个二进制精度而使用无符号数
    - 如果你需要一个额外的二进制位，你还会再需要一个
    - 不要在表达式中混合使用有符号和无符号数
  - 你无法完全避免无符号运算
    - 标准库容器的索引使用的是无符号数
      (BS 认为，这是一个设计错误，呵呵)

```
vector<int> v;
// …
for (int i = 0; i<v.size(); ++i) …
for (vector<int>::size_type i = 0; i<v.size(); ++i) …
for (vector<int>::iterator p = v.begin(); p!=v.end(); ++p) …
```

signed

unsigned

correct, but pedantic

Yet another way

复杂性

- 出错原因之一是问题的复杂性
  - 天生(必然)复杂性
- 另一个原因是编写的差劲代码
  - 偶然复杂性
- 不必要复杂性的原因：
  - 过于聪明的程序员
    - 使用不容易理解的特性
  - 未经良好培训的程序员
    - 没有使用最合适的特性
  - 大量混用的编码风格

编码标准

- 编码标准是一个规则集合，指明代码应该是什么样的
  - 典型地会指定命名和缩进规则
    - *e.g.*, 使用“Stroustrup”的布局
  - 典型地会指定使用的语言特性集合
    - *e.g.*, 不要使用 **new** 或 **throw**，以避免可预测性问题
  - 典型地会指定注释规则
    - 每个函数必须有一个注释以解释它的功能
  - 通常，需要使用某种库
    - e.g., 使用**<iostream>** 而不是**<stdio.h>** 以避免类型安全问题
- 各单位组织通常通过编码标准来尽量管理复杂性
  - 通常，他们会失败，并创造出更大的复杂性

编码标准

- 一个好的编码标准要比没有好
  - 如果没有就不要启动一个大项目(需要多人多年完成)
- 一个差的编码标准要比没有坏
  - 约束编程要像C风格的C++ 编码标准是有害的
  - 它们是不一般的
- 所有的编码标准都会有程序员不喜欢
  - 即使好的也如此
  - 所有的程序员都想用自己的方式来写代码
- 一个好的编码标准是规范的，并提供限制
  - “这是做这件事的好方式”，以及
  - “从来不要这样”
- 一个好的编码标准会给出自身规则的合理性
  - 以及示例

编码标准

- 共同目标
  - 可靠性 Reliability
  - 可移植性 Portability
  - 可维护性 Maintainability
  - 可测试性 Testability
  - 可重用性 Reusability
  - 可扩展性 Extensibility
  - 可读性 Readability

一些样本规则

- 任何函数不要超过200行 (30或许更好)
  - 是指不含注释的200行源码
- 每条语句都从新的一行开始
  - *e.g.*, **int a = 7; x = a+7; f(x,9);** *// violation!*
- 除了源码控制，不要使用宏
  - 使用 **#ifdef** 和 **#ifndef**
- 标识符应该采用描述性的名字
  - 可以包括常用的缩写或缩略语
  - 如果 **x, y, i, j,** etc. 是按传统习惯使用的，认为具有描述性
  - 使用下划线风格 **number_of_elements** 而不是字头风格 **numberOfElements**
  - 类型和模板以大写字母开头
    - *e.g.*, **Device_driver** 和 **Buffer_pool**
  - 标识符不应该仅仅采用大小写字母进行区分
    - *e.g.*, **Head** 和 **head***// violation!*

一些样本规则

- 作用域内外的标识符不用相同的名字
  - *e.g.*, **int var = 9; { int var = 7; ++var; }** *// violation: **var** hides **var***
- 应该在尽可能小的作用域中进行声明
- 变量应该初始化
  - *e.g.*, **int var;** // violation: **var** is not initialized
- 只有在必要时才进行cast转换
- 代码不应该依赖于算术表达式优先级之下的优先级规则

  e.g., **x = a*b+c;** *// ok*

  if(a<b || c<=d) *// violation: parenthesize **(a<b)** and **(c<=d)***
- 递增和递减操作不应该用在自表达式中
  - *e.g.*, **int x = v[++i];** *// violation (that increment might be overlooked)*

位操作实例

- The Tiny Encryption Algorithm (TEA)
  - 由 David Wheeler 最早设计的
  - http://143.53.36.235:8080/tea.htm
- 不必过于仔细地阅读加密程序，只是让你体验如何编写实用的位处理代码
- 该算法输入一个字 (一次 4 bytes)
  - *e.g.*, string中的4个字符或图形文件
- 它假定 4字节长的整数
- 算法具体解释参见连接

TEA

```
void encipher(
    const unsigned long *const v,
    unsigned long *const w,
    const unsigned long * const k)
{
    unsigned long y = v[0];
    unsigned long z = v[1];
    unsigned long sum = 0;
    unsigned long delta = 0x9E3779B9;
    unsigned long n = 32;
    while(n-->0) {
        y += (z << 4 ^ z >> 5) + z ^ sum + k[sum&3];
        sum += delta;
        z += (y << 4 ^ y >> 5) + y ^ sum + k[sum>>11 & 3];
    }
    w[0]=y;
    w[1]=z;
}
```

TEA

```
void decipher(
    const unsigned long *const v,
    unsigned long *const w,
    const unsigned long * const k)
{
    unsigned long y = v[0];
    unsigned long z = v[1];
    unsigned long sum = 0xC6EF3720;
    unsigned long delta = 0x9E3779B9;
    unsigned long n = 32;
    // sum = delta<<5; in general, sum = delta * n
    while(n-->0) {
        z -= (y << 4 ^ y >> 5) + y ^ sum + k[sum>>11 & 3];
        sum -= delta;
        y -= (z << 4 ^ z >> 5) + z ^ sum + k[sum&3];
    }
    w[0]=y;
    w[1]=z;
}
```