

单片机程序调试黑宝书

2012-1-25 全文完，如果您因故无法阅读完整内容，请给我发送电子邮件，我会将完整的版本发送给您！

欢迎广大网友批评讨论！

E-mail:soundman@sohu.com

一、前言：

1.1 你离高手有多远？

首先我必须放下架子，因为本文的读者中很大一部分在不久的将来都会超越我。而且我也 100%不能自诩为高手，我不过是比本文的部分读者碰的钉子多些罢了。再退一步讲，即使你请了一位“高手”帮忙，如果他不是对你的具体系统十分了解，也只能给你一些原则上的建议罢了。

 结论：没有绝对的高手，高手是积累出来的，程序调试靠自己。

1.2 谁应该读这篇文章？

我们经常在论坛看到类似这些主题的帖子“I2C 程序怎么调 为什么我的程序不对？”，然后贴一堆代码上来；“为什么我这样写对，那样写不对”。如果你提过类似问题，或者不知道程序该怎么 Debug，就请读本文了。

如果你刚开始学习单片机，可能觉得本文不着边际，那么请先死记这些结论，待到 3 年后再从头读一遍，一定会和我发生共鸣。

1.3 这篇文章针对哪种单片机或者哪种语言？

这篇文章不涉及任何具体单片机型号和任何具体语言，你可以把他理解为凌驾在具体嵌入式技术之上的技术，就像哲学那样。

1.4 这篇文章有版权吗？

有的！但是我不准备出版，也不准备收费。

因为我国 99.99%的高校毕业生（甚至读完研究生）都不曾看到这样专业化的程序调试教程，如果按大家为高等教育付出的几万元代价计算，这篇文章我起码会卖到 10 万/人，太天价了！这篇文章的目的是总结、提高，并在 21IC (bbs.21ic.com) 上提供免费下载，你转载的时候只要保证本文的完整性，并注明出处就可以了。

作为免费的等价条件，我也不承担读者因为本文造成的任何损失，请保持独立思考的习惯，并不要随意使用本文中的代码，这些代码有的是伪语句，这些代码只是为了配合文字说明问题用。

1.5 这篇文章所列举的事例和 BUG 真实吗？

孙子云：兵不厌诈。这些例子不一定是我所在公司所遇到的，也有我经过组装修饰的，也得给我点隐私权嘛，鉴于本文的非商业化目的，我不对文中任何所提及的产品和技术负责。

二、该如何写程序：

我们不怕得罪“Coder”，但是需要首先建立一个观点——程序是电子技术里面最简单的东西，因为程序的确定性比起硬件大得多。处理器的行为是认为设计的数字逻辑行为，不存在硬件上得容差问题。

话说硬件设计需要很多数据库型的知识支撑，高频还需要黑色艺术细胞，学写程序除了背点语句，掌握一些基本技巧外加做好规划之外，不需要其他东西，**会说话就会写程序！**

👉 结论：程序的确定性比硬件大，不要害怕程序问题。

2.1 什么是程序？

“程序就是为了让处理器做某件事情而编写的有序汇编代码集合”。

这里要注意两件事情，一是“做某件事情”，说明程序是为需求服务的，只有把需求搞清楚，程序才能写得出来；二是“汇编代码集合”，所有计算机只认识一种语言——机器码，也就是汇编所对应的机器语言，其他再华丽的高层语言（例如 C）最终都会成为汇编指令供机器执行，只是这个过程被编译器（例如 C 编译器）自动执行罢了。

从这个角度来说，无论你掌握了多少种语言，例如 C、C++、汇编，也无论你可以在计算机二级 C 语言考试得多高分，都不等于你会写程序。

👉 结论：写程序，最重要的不是学会某种语言，而是会分析问题并提出解决问题的方法。

2.2 顺序程序

如果当一个程序编写好后，所有语句被执行的先后顺序已经确定下来，这就是一个顺序程序。

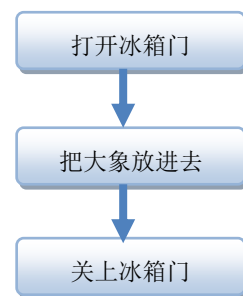
这种程序通常有如下特征：

- 1) 不使用中断系统（当然也就包括了不使用操作系统）
- 2) 不与操作者发生交互，或者在交互时，死等操作者指令

顺序程序可以用流程图非常明确地描述出来，例如非常经典的“如何把大象放进冰箱”问题，可以用右边的流程图【1】表达

虽然把大象放进冰箱只是一个笑话，但是说明了这个过程是由 3 个动作组成的，并且这 3 个动作之间的顺序是不可颠倒的。任何初学者，只要能够理解“如何把大象放进冰箱”的奥妙，就能开始写程序了！

作为一个特例，在程序中有等待用户操作环节的，只要在等待时不进行其他操作，同样也是顺序程序。



图【1】

2.3 含有中断的程序

如果一个程序使用了中断，无论这个中断是用硬件中断（例如外中断 INT，串口接收中断等）直接操作，还是通过定时器切换的操作系统，都统称为含有中断的程序。这种程序的特点是：

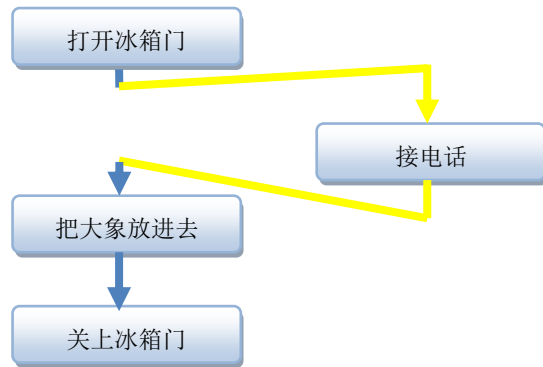
- 1) 含有多个并行运行的代码（例如主循环和中断服务程序）
- 2) 这些并行代码间运行的先后顺序错综复杂

我们继续用“如何把大象放进冰箱”问题，来描述。

金黄色部分流程线描述了在放大象的过程中接电话的“中断”。一旦程序加入了中断的环节，就会变得复杂起来，因为接电话这个事情可能发生在任何时候——打电话的人不可能知道你在放大象嘛。

另外，加入了中断环节的程序可能出现很多意想不到的事情，比如接电话期间，大象可能跑掉，或者冰箱门被加上了“不允许打开超过 1 分钟”的限制条件。

对于复杂的中断，还可能存在着“接到电话，要求把大象红烧吃了”的情况，这样接完电话以后就没有大象可放了，这就是中断和操作系统中经常遇到的“临界资源”问题。



图【2】

👉 结论：含有中断的程序较为复杂，需要编写者清楚同一时刻，我在做什么，其他人在做什么，用“并发”的方式思考问题，才能写好。

2.4 程序模块化

首先说明，程序模块化是为了提高编程效率，扩大编程者对程序的掌握能力，便于程序维护而产生的，对计算机本身而言，程序是没有工整和杂乱的区别的。

程序模块化的基本任务就是将复杂的设计任务划分为若干个功能明确，出入口简单的功能块。

👉 结论：程序模块化是为了编写而不是为了运行，模块化和函数是两个不同的概念，函数是为了将需要多次使用的代码统一编写，以便减少程序代码量，便于维护；模块化是指将复杂的程序功能化整为零而成的功能块，一个模块可能由多个函数组成，也可能就是一个函数，还有可能只是一段紧密相连的代码块。

我们继续用大象的例子来示范，这里的 3 个动作都可以看成模块。假设我们由一个机器人来做这件事，机器人不能理解“打开冰箱门”、“把大象放进去”、“关上冰箱门”这 3 个步骤，那么就要用更基础的语言来为机器人编程。

【模块——打开冰箱门】

1. 抬起右手，移动到 冰箱门把中心右侧 1cm 处
2. 弯曲右手手指，勾住冰箱门

3. 以 2kg 的力量向后拉
4. 完成

当然，站在不同的角度来看，模块的定义可能将发生变化。比如任务整体扩大到“红烧大象”的级别，“买大象”、“把大象放进冰箱”、“把大象取出冰箱”和“烹饪大象”将成为模块。在很多时候，模块的划分仁者见仁，但是总的原则要求是功能内聚、接口简洁，如果某种划分方式使得模块间到处都是接口，那么这个划分肯定是不好的。

👉 结论：好的模块化设计，模块间的接口简单明了。总的来说，好看好改的程序就是好程序。

2.5 程序编写

我们来看一个 211CBBS 上的实例：

```
#include<reg52.h>

sbit latch1=P2^1;
sbit latch2=P2^2;

void delay(unsigned int t);
unsigned char weima[8]= {0xfe,0xfd,0xfb,0xf7,0xef,0xdf,0xbf,0x7f};
unsigned char duanma[10]= {0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f};
unsigned char LED[]={0xcf,0xf3,0xfc,0xf3};
unsigned char tempdata[10];

void delay(unsigned int t);
void display(unsigned char firstbite,unsigned char num);
main(void)
{
    unsigned int m=30,n,i=0;
    P1=0xcf;
    TMOD=0x10;
    TH1=0x3c;
    TL1=0xb0;
    TR1=1;

    while(1)
    {
        if(TF1==1) //等待定时器溢出中断标志
        {
            TF1=0;
            n++; //软件定时器
        }
    }
}
```

```

    TH1=0x3c; //重载定时器
    TL1=0xb0;

    if(n==20)
    {
        n=0;

        if(m>0)
            m--;

        if(m==0)
        {
            i++;
            P1=LED[i];

            if(i==3)
                i=0;

            m=30;
        }
    } //End Of if(n==20)
} //End Of if(TF1==1)

tempdata[0]=duanma[m/10];
tempdata[1]=duanma[m%10];
display(3,2);
} //End Of While(1)
} //End Of main()

void delay(unsigned int t)
{
    while(t--);
}

void display(unsigned char firstbite,unsigned char num)
{
    unsigned int i;

    for(i=0;i<num;i++)
    {
        P0=0;
        latch2=1;
        latch2=0;
    }
}

```

```

        P0=weima[i+firstbite];
        latch1=1;
        latch1=0;

        P0=tempdata[i];
        latch2=1;
        latch2=0;

        delay(200);
    }
}

```

这个求助帖的原文是“这个程序的定时不准 我本来定的间隔是 1s 结果运行时是 1.04s 左右，通过改初值也无法精确到一秒，总会多或少 0.0 几秒，请高手给看看是怎么回事？晶振是 12MHz 的 谢谢”

很明显这是一个初学者写出来的程序。请注意，程序里的注释是我加上去的，因为程序实在太乱了，我用 UE 才排出现在大家看到的缩进。抛开程序里的其他设计问题不说，仅讨论楼主的问题——为何定时不准，老实说我也不知道！！

因为我根本就不会把程序写到这个程度再来调试！很明显这个程序由 3 个模块组成——基础定时模块（由 Timer 实现）、软件定时模块（由 m 和 n 组成的计数器实现）、显示模块（由函数 display 实现）。定时准确的核心是基础定时模块的溢出频率准确，所以在编写程序的时候先要对其进行测试。例如基础定时模块的设计中断周期是 1mS，我会在程序写到这个程度的时候停下来：

```

_isr_timer_ovf()
{
    timer_reload();
    P0^=0x01;          //测试代码
}

main()
{
    init_device();    //Init Timer, IO and Interrupt
    while(1);
}

```

假设设计的定时器溢出周期是 T，这段代码将在端口 P0.0 上产生一个以 2T 为周期(2mS)的方波，用示波器或者频率计对这个方波进行测量，就可以知道基础定时器模块工作是否正常。接下来再编写软件定时模块。

```

unsigned int timer;

_isr_timer_ovf()
{

```

```

timer_reload();

timer--;

if(timer == 0)
{
    timer = 1000;
    P0^=0x01;        //测试代码
}
}

main()
{
    init_device();    //Init Timer, IO and Interrupt
    timer = 1000;
    while(1);
}

```

在这一步，我们再一次用 P0.0 上的方波对软件定时器产生的 2S 周期定时进行检验，最后我们加入显示部分。

```

unsigned int timer;

_isr_timer_ovf()
{
    timer_reload();

    timer--;

    if(timer == 0)
    {
        timer = 1000;
        display(timer);
    }
}

main()
{
    init_device();    //Init Timer, IO and Interrupt
    timer = 1000;
    while(1);
}

```

如果这样写程序，即使有设计失误导致定时不准，在编写显示程序前已经被发现了。与

哲学的观点相同，事物是相互联系的，一个程序中包含的所有代码间都将相互关联，由于设计不当，显示程序可能侵占软件定时器所使用的内存空间，造成定时不准确。只有将这些代码从程序中去掉，才能彻底洗掉他们的嫌疑，使编写者集中精力调试要调试的模块。

👉 结论：编写程序的好习惯是分模块编写，边写边测试，在通过测试的模块基础上编写下一个模块，可以减少程序出现问题的可能性，快速排查与问题相关的模块并定位到程序语句。

对例子程序另外一个需要注意的地方是——作者完全没有写注释。注释是给人看的，编译器（IDE 环境，例如 Keil、ICC AVR 等）和目标 CPU（运行该程序的单片机）是完全不看注释行的，但是注释行能够理清编写者的思路，也方便日后修改维护的人（90%的可能是编写者自己）。

👉 结论：注释不是程序，但可以帮助编写者提高编写的正确性，也可以大大提高程序的可维护性。建议 C 语言程序注释到函数，一些重要的操作至少要注释；汇编语言程序，至少注释 70%的语句行，建议一行一注。

三、Debug:

3.1 BUG 无处不在

写程序不出 BUG 是每个设计者的追求，但往往事与愿违，就连软件大鳄微软也在天天写补丁。当然我们的程序规模远不如操作系统那么巨大，但即使是做了多年单片机程序的开发人员也几乎不可能做到程序一次通过无 BUG。

👉 结论：程序有 BUG 是很正常的，要学会找 BUG 的方法。

3.1.1 硬故障和软故障

从表现方式上，故障分为硬故障和软故障两大类。我们先来讲一下什么叫软故障。

比如某个顾客投诉说：我买的电视开不了机了。这就是一个“硬”故障，因为他是不可能自动恢复的，无论维修人员什么时候上门查看，故障的现象完全一致。

当另一个顾客投诉说：我买的电视有时候喇叭里有噪声。这就是一个“软”故障了，因为故障的现象不是随时都存在的，当维修人员上门查看时，有可能电视工作得好好的。

软故障是维修人员最头疼的，同时也是最容易引发消费者投诉和纠纷的地方。

从技术的层面来分析，硬故障可以称为“无条件故障”，他们的出现多是由于元器件不可逆转性的彻底损坏造成，比如保险管烧断了，芯片烧了。这种故障现象明确，容易查找，只要按照设备的工作条件逐个模块检查就可以发现。

软故障可以称为“条件故障”，他们的出现多是由于设计时考虑不周，工作条件临界（时

序临界、电压临界、信号电平临界、热设计临界等)，他们的出现多依赖于特定条件，比如打开电视 1 小时后出现死机，有可能是热设计不良导致；又比如在晚上 8 点左右出现显示扭曲甚至无法收看，有可能是电源适应性不足导致（晚 8 点为用电高峰，电网电压略低）。

👉 结论：故障分软硬，软故障最难找。软故障多源于设计临界所导致，在一定触发条件下发生，当条件被破坏时，故障可能消失得无影无踪。

软故障经常误导维修人员。我们来看一个例子。



案例 Example

一款使用 51 单片机的读卡器，操作逻辑加密 IC 卡时经常出现不识卡的故障。市场维修人员认为是 IC 卡插座接触不良，更换 IC 卡插座后，故障消失，但过段时间后顾客又反映其不识卡，更换卡座后又好了，几年下来换了一大堆卡座，故障就是从来没有消失过。后来产品升级换代，开发人员发现软件中对卡的复位时间短于卡片数据手册要求值，修正后不识卡的故障再也没有出现过。

软故障的误导性来源于对硬故障的经验积累。硬故障因为无需触发条件，人们往往用

尝试——判断——再尝试

（请注意，这行文字在后面蛮重要的，所以我加黑了）

的思维方式查找问题，比如维修完全没有声音和图像的电视机，会尝试更换保险管，如果可以开机了，那么就判断为保险管烧毁。但是这种思维方法换到处理软故障的时候就行不通了，因为软故障有出现的条件，更换了某个元件，故障不再出现，并不代表是这个元件引起的。

软件对 IC 卡的复位时间过短，是一个设计上的失误，但是通常 IC 卡的生产厂家在写数据手册时，也同样留有余量，例如当数据手册指明复位时间需要 5mS 时，当读卡器复位时间为 4mS 时，80% 的卡片可能能够成功复位。再者，设计例如读卡器这类低成本简单产品时，单片机的时钟源很可能是 RC 振荡器而非石英晶振，在温度发生变化时，其时钟频率将发生显著改变（例如 5%），这就导致了不识卡的故障在维修人员面前很难表现出来。

当一个维修人员来到现场，按照“卡座接触不良导致不识卡”的经验为用户更换了卡座，并抽出一张随身携带的测试卡插入读卡器后，凭其“可以识卡”的现象判断自己的判断和维修手段是正确的，然后到下一个顾客那里去继续加深自己对“卡座接触不良导致不识卡”的错误认识，并继续循环下去。

现代电子产品的基本模式是软件+硬件，任何人所做的事情都可能存在错误，所以软件和硬件都有出现问题的可能性，从上面的例子可以看出：在没有分清软硬件故障时盲目下手，可能将进一步误导自己。从另一个方面看，由于驱动层软件与硬件密切相关，当驱动软件存在问题，特别是边界性的软故障时，很容易造成误判。

3.1.2 软件和硬件

这是不团结的开发部最容易发生矛盾的地方——当产品遇到 BUG 时，硬件开发人员说是软件引发的，软件开发人员说是硬件造成的。

其实无论软件硬件，寻找 BUG 的方法大同小异。对一个开发人员来说，只有同时掌握了软件和硬件设计技术，才能合理分配软硬件资源，做出最优的方案。上面的案例已经说明，将软件问题误判为硬件问题，只会南辕北辙。

想区分硬件和软件 BUG，就要先知道硬件和软件各自的特点。

硬件最大的特点是离散性，跟达芬奇画鸡蛋的故事一样，简单说就是世界上没有两个完全一样的电阻。硬件设计的重要任务就是让图纸上的数值包容工业化生产可以容忍的误差范围。

软件最大的特点就是 consistency 好，由同一个 HEX 文件烧写出来的单片机，他们所包含的程序都是一致的。

硬件所引发的问题一般特征是：故障集中表现在某一台或某几台机器，而其他机器根本没有发生过这种故障；与温度、电压等工作环境密切相关；随着更换某个器件，故障消失。但是需要注意的是，芯片，特别是新取出的芯片，其损坏的概率是很低的（除非设计不当，上一个坏一个）。

软件所引发的问题一般特征是：故障随机表现在所有机器上，或至少绝大部分机器都出现这种故障；与工作环境关系不大；随着更换备份软件，故障消失。（在软件开发过程中，开发人员一般将备份很多个中间版本的备份软件）



注意

由于单片机系统中，软件和硬件紧密结合，硬件的工作依赖于底层硬件的驱动，所以上面所说的判断方法仅供大致判断，更多时候应该将硬件和软件看做一个整体。



案例 Example

一个使用外部 E2 存储器的单片机系统，在硬件系统已经开发定型的情况下，分别由 2 人进行软件开发，第 1 人在完成整个程序后离职，第 2 人使用其所遗留的样机继续进行软件维护。维护后发现，将软件下载到新生产出的若干台机器中发现均无法工作，而将软件下载到样机中即可以运行。由于该软件规模较大，开发文档不完善，通读一次需花费很大精力，急需尽快确定是软件问题还是硬件问题，以便查找问题。

这个问题乍一看很像硬件问题，因为在样机上可以正常工作的软件在新硬件上出问题

了。首先初步查找这个问题，发现系统在向 E2 写入某个数据时流程无法完成，导致程序停止不前（由于原程序设计不当，并未给出出错提示）。可能导致这个故障的现象有：新机器 E2 故障、程序问题。

深入来想这个问题，首先硬件系统已经定型很久，且并没有进行修改；其次，新机器中每个 E2 存储器都损坏的概率几乎为 0，故从其他方向分析新机器和样机有什么不同。我们注意到新出厂的 E2 存储器内的所有数据一般均为 0xFF，而样机 E2 中有工作数据，遂怀疑软件会使用其中的工作数据，但没有做异常处理，所以出现了流程锁定。

验证这个问题，交换样机和新机器的 E2，样机变得无法工作，新机变得可以工作。用万用烧写器拷贝样机 E2 数据到空白 E2 后安装在新机器上，新机器可以工作。证实了上面推论的正确性。集中精力寻找该流程所需的 E2 中的工作数据，找到该数据，并修改相应流程，完整地处理了该 BUG。

👉 结论：单片机软件和硬件密不可分，查找问题时要两者并重，不可死认一方。

3.2 故障的查找

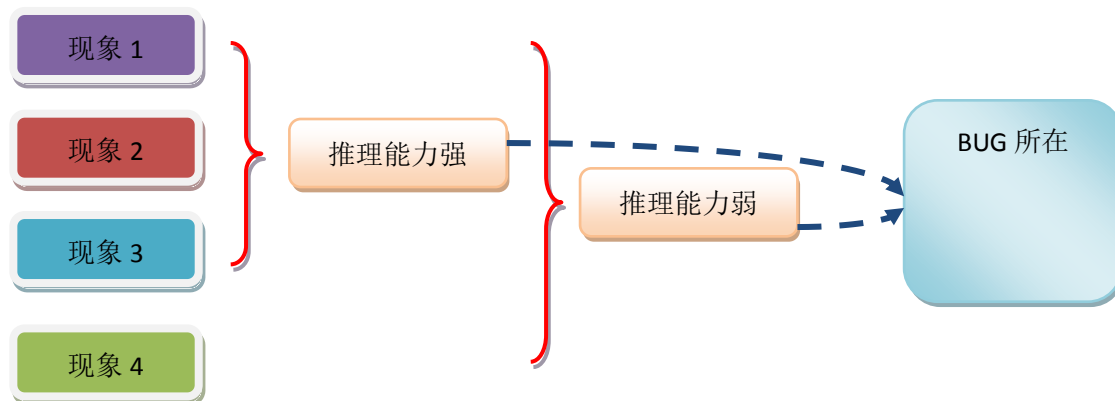
既然 BUG 无法避免，咱就必须练就一身找 BUG 的神功。找 BUG 的功力没有绝对，正如前面所说，一个“高手”在对你的具体系统没有任何经验的基础下，DeBug 能力可能远不如一个亲自编写这个程序的毕业生。

但是一般来说，我们可以总结出一些常用的技巧来加速推理过程。

👉 结论：Debug 高手和新手间的区别在于，高手知道该如何去找，而新手会迷茫。

3.2.1 故障查找的基本理论

故障查找的基本理论是通过分析推理出故障所在。



上面的图给出了故障查找的基本模型。我们首先要相信，当一个 BUG 发生时，除非是非常明确的故障（比如看到保险管爆裂烧毁），无论“新手”还是“高手”都不知道这个 BUG 是如何发生的，摆在他们面前的是平等的事实——这个 BUG 所带来的现象。



案例 Example

现象 1：一台 POS 机，在打印发票时总金额显示为乱码

现象 2：该现象在以管理员身份登录时不会发生

现象 3：该现象仅发生在每天 21 点以后

现象 4：该现象仅发生在当日自动日结报表运行后，该用户将自动日结报表时间设置为 21 点，以管理员身份登录时不运行该程序。

这个案例模拟了一个因为 POS 机“自动日结报表”程序模块漏洞导致的发票金额乱码问题，“现象 1”描述了最直白的故障现象，这个现象是最容易发现而且无需任何专业知识的，但是也是对 BUG 分析最无帮助的。现在你相信了，任何“高手”在“现象 1”面前都将束手无策。

“现象 2”提供了一个特例，这个点将严重区别出新手和高手，新手的脑袋里一团浆糊：“管理员登录和发票乱码咋可能有联系呢？”，高手将换一种思维方法——拿过一张草稿纸，开始罗列管理员登录和普通登录间的区别。

“现象 3”与“现象 2”相似，新手继续浆糊，高手继续找区别。

“现象 4”的层次属于真相大白，但是需要积累更多的经验，这种经验可能是通过一群焦头烂额的程序员围在收银员旁边观察一整晚，突然有人看到屏幕提示“自动日结报表运行中”而灵光闪现发觉的。

其实一个推理能力强的“高手”，用“现象 2”已经排除了一大部分程序模块，再利用“现象 3”联想到 21 点前后程序运行了什么操作，就很可能已经怀疑到“自动日结报表”功能上来了。相反，如果新手没有机会在收银员身边发现屏幕提示，这个 BUG 的查找可能会拖上几个月之久。

注意另一个事实：如果一个抽调过来的顶级 Debug 高手，连 POS 机是什么，“自动日结报表”是什么都不知道的情况下，最多得到下列结论——该 BUG 与管理员登录有关，与特定时间执行的操作有关。

3.2.2 柯南肯定是找 BUG 的高手

由于目前主要的单片机设计者都是 80 后，柯南是谁就不需要我多解释了。不过如果有一天，柯南愿意一改侦探职业来搞单片机开发的话，可以断定他肯定是个找 BUG 的高手。

我们先来看下柯南到底在哪些方面强于那些佩服他的人：

- a. 知识面丰富，例如 258 集中，知道英国人把警车称为“熊猫”，在 57 集中知道升高外界温度可以加速尸僵的软化，这些都是一般人不知道的知识。
- b. 善于注意细节，这个例子就太多了，柯南查看现场的时候从进门的那一刻起就在注意各种细节，甚至很多罪犯自己不留神说出来的疑点都不会被他放过。例如 52 集中仅凭几个庙里的师兄说有橡皮艇可以出去玩就联想到作案工具。
- c. 当有提示时迅速联想，做这个工作的多是步美或者小兰，通常是她们无意间说了什么或玩了什么，引起柯南的联想，找出破案的关键。例如 479 集从小兰玩的飞车推出榻榻米可以旋转摆放。
- d. 有推理后努力找证据证明，这个太多了。很多时候柯南自己推测到了罪犯的手段，但是没有找到关键性证据前，不敢说出来，这时候说“破案”只能说是直觉，知道找到证据，才把毛利放翻后公布答案。

如果把柯南的侦探经验翻译成找 BUG 经验的话，就是这样——

- a. 知识面丰富，包括硬知识（对所用芯片、开发环境、操作系统等本身的经验）和软知识（具体到系统本身的特殊点，例如前面 POS 机例子中日结报表的信息）
- b. 善于注意细节，当一个 BUG 放在大家面前时，谁先收集到足够多的细节信息，就意味着他所掌握的推理资源越多，推理越容易。
- c. 当有提示时迅速联想，找 BUG 经常会一筹莫展，这时你的大脑里存储了大量的信息，却找不到推理的头绪，这时候就需要最后一点提示来捅破窗户纸，当这种提示（可能是一个很重要，但是原来忽略了的现象）偶然到来时，你有准备的头脑才能灵光闪现。
- d. 有推理后努力找证据证明，80%的时候，靠着猜和直觉找到了 BUG 所在，但是要将所有的现象关联到这个 BUG 上，才算确认，如果其中有任何一个现象无法用 BUG 来解释，或者与之矛盾，这个推理都可能是错误的或者不完善的。

👉 结论：平时就要注意积累。学习软硬件知识，还要熟悉自己的实际应用，包括应用条件，否则会对放在面前，对推理极有价值的现象视而不见。找 BUG 可以用“猜”的方法，但是“猜”完了要推理到所有的现象上去“验证”。所以我们说“怀疑——否定——再怀疑——一再否定，知道找到真理所在”（前面所提到的“尝试——判断——再尝试”）是找 BUG 的基本方法。

3.2.3 以论据支持的方式寻找 BUG

请刚看完上一章节的读者们一定记住：在柯南刚踏入案发现场的时候，他跟其他人一样是晕的——对犯罪过程一无所知，他与别人的差距出现在整个侦查过程中。

同样，当一个 BUG 出现时，在场的新手和“高手”都是一头雾水。随着找 BUG 工作的展开，“高手”做一些有目的的测试，并且在有意和无意中发现一些现象。“高手”将这些现象加以推理，就可以找出 BUG 所在，这些现象就是我们说的“论据”。

👉 结论：在 BUG 面前，新手和高手的初始化状态是一样的，只是高手有意识去找现象用于分析，新手看着现象发呆罢了。

其实我们前面例举的定时器的例子就说明了新手和高手认识问题的区别——新手在已经发现 BUG 的程序上继续找问题，高手则告诉你，将无用的部分全部屏蔽掉，从调试最核心的定时器部分开始。



案例 Example

这是一个用 51 单片机从 DS18B20 读取温度值并在 LED 上显示的程序。遇到的 BUG 是，无论外界温度如何变化，LED 始终显示“00”。


```

char    temp;

main()
{
    init_device();          //Init Timer, IO and Interrupt
    temp = 0;
    while(1)
    {
        temp = get_temp();
        cal_temp(temp);    //修正温度值
        disp(temp);       //显示温度值
    }
}

```

新手在遇到问题的时候可能采用的方法就是发到 BBS 上寻求“高手”的帮助。在场的高手可能会要求先做一个这样的测试程序：

```

while(1)
{
    temp = get_temp();
    cal_temp(temp);    //修正温度值
    temp = 12;
    disp(temp);       //显示温度值
}

```

这样做所想找的论据是——显示函数 `disp()` 是否工作正常，因为 `temp = 12;` 已经对读取到的温度值做了硬赋值，即使前面的读取函数和修正函数存在问题，也不会影响显示。如果这个测试使得 LED 正确显示 12，那么可以证明显示函数基本正常，转而去寻找读取函数和修正函数的问题；如果仍然显示 00，那么应该首先解决 `disp()` 函数本身的问题，再继续调试。

 结论：论据有两种获得方式——寻找和傻等，高手以寻找为主，新手以傻等为主。

当然，即使是高手，也可能遇到束手无策的时候，这时候闲着也是闲着，他可能会做一些看似不相关的测试，从而发现一些新的论据，从而促使对 BUG 更深的分析。在后面我们将介绍一些常用的论据寻找方法，帮助大家快速掌握足够支持推理的论据。

3.2.4 以推理的方式确认 BUG

正如我们再 3.1.1 例子中看到的一样，BUG 常常被误判，可以从这个换卡座的例子可以看出误判所带来的不良后果。很多时候，我们和柯南一样，找到问题关键的时候是靠“灵光一闪”，带有“猜”的性质，没有把整个 BUG 的来龙去脉想清楚，这就需要我们再“灵光”过后，再以推理的方式确认 BUG 是否找准了。

一般来说，确认 BUG 找的是否准确可以用下面这些方法：

- a. BUG 和现象具有明确的对应关系。
- b. 能够清晰地理出从 BUG 所在位置出发，直到现象发生的完整线条。
- c. 所发现的现象中，没有用该 BUG 无法解释的现象，或至少没有与之矛盾的现象。

明确的对应关系，是指将改正了的 BUG 改回去，现象还能重现。例如我们发现一个 LED 数码管显示错误的程序是由于将循环体 `for(i=0;i<=4;i++)` 不慎写成了 `for(i=0;i<=3;i++)` 导致最后一个数码管没有显示，那么应该在找到 BUG 的根源后故意将 `i<=4` 再改回 `i<=3`，如果 `<=3` 时必然出错，而 `<=4` 时正确，则说明我们找到的 BUG 位置正确，与所观察到的现象是明确对应的。

在确认 BUG 时，不能像找论据那样天马行空，而需要理清条理，一板一眼地说明这个 BUG 是如何一步一步引起这个现象的。比如上面这个循环的问题，应该推断出循环少执行了一次，然后跟着程序流程来清理，说明这少循环的 1 次是如何影响显示效果的。

至于无法解释的现象，例如 for 循环是这样编写的：

`for(i=0;i<j;i++)` 这里 j 定义为本次有几个数码管参与显示，如果经过查找发现，是 `i<=j` 被错误写成了 `i<j`，但是已知的现象里，当 `j=3` 和 `j=5` 时，最后一个数码管不亮，`j=4` 时却要亮，那么说明这个“少写等号”的推断可能是错误的，或者至少是不完整的。

在某些时候，引起一个现象的 BUG 还可能不止一处，当发生无法解释的现象时，一定不要放松对“不止一个 BUG”的怀疑。

3.2.5 随机对随机的方式确认 BUG

这是一种很极端的情况，由于程序引发的软故障现象，随机地出现在产品中，且概率不高。我们用“猜”的方式似乎找到了问题所在，但是又暂时无法理出从 BUG 到现象的完整路线，这时你可能就需要用随机对随机的方式来处理了。

随机对随机是基于小概率事件难得一见的原则来思考的，例如：

你敲开一个鸡蛋——一个双黄蛋，恭喜你中奖了！

你又敲开了一个鸡蛋，还是双黄蛋，你手气这么好？

你再敲开了一个鸡蛋，还是双黄蛋，昨天买彩票的时候咋没这么好手气？

当你连续敲开了 6 个双黄蛋——赶紧给电视台打电话吧，你要红了！

我确实见过新闻报道连敲 6 个双黄蛋的，但那个已经成为国际新闻，一般来说你不会有这么好的手气。



案例 Example

某个批量生产中的使用干电池供电的设备，由于进行了一次技术升级而造成偶发性故障，表现为上电后某个芯片偶然性初始化失败。通过技术人员反复阅读该芯片的数据手册，发现该芯片的使能引脚是没有内部上下拉的，在单片机复位期间，由于 I/O 处于高阻输入状态，可能使该使能端悬浮，但是无法将使能端悬浮和初始化失败联系起来，只能知道“输入端悬浮”是违反设计规则的。

故障现象中，上电初始化失败的概率大约在 1% 左右（平均每做 100 次上电，有 1 次失

败，但每 100 次内不一定有 1 次) 当加入使该芯片处于非使能状态的下拉电阻后，设计人员在 1 小时内都没有做出上电初始化失败的现象。

因为故障现象出现的概率很低，这个问题到底解决了没有呢？开发人员也不是很肯定。

为了在短时间内确认 BUG 寻找是否准确，开发人员从生产线找来了 10 台曾经被测出过上电复位失败现象的“样机”，选取了其中 5 台加装了下拉电阻，记录了产品编号，装回原样交给生产线测试，要求他们将测出故障现象的样机单独放在一边。

半小时后，5 台没有装下拉电阻的样机被“准确”地筛选了出来。由于生产线并不知道 10 台样机中只有 5 台进行了改装，所以对所有样机的测试强度和认真程度完全一致，排除了人为因数。在这个前提条件下，5 台改装的样机齐刷刷地“不出问题”是非常小概率的事件，几乎不可能发生，倒过来说就是——下拉电阻真的解决了这个问题。

在随后的数天里，开发人员找到了上电期间由于该使能端悬浮所引起问题的确实线索。

👉 结论：随机对随机的方法是一种在查找小概率软故障时用的极端确认方法，需要使用者对故障的基本特征熟练掌握，明确“小概率事件很少发生”的原理。

其实前面的例子里，我们已经贯穿了“小概率事件很少发生”的思想，例如说新取出的芯片很少损坏，说找到了 BUG 以后再改回去看故障现象是不是随之发生。

3.3 查找故障的常用方法

现在我们已经知道，找 BUG 需要现象支持，高手用一些总结出来的方法去尝试寻找现象，这些方法在 90% 的情况下都是奏效的，所以我们来总结一下这些方法。

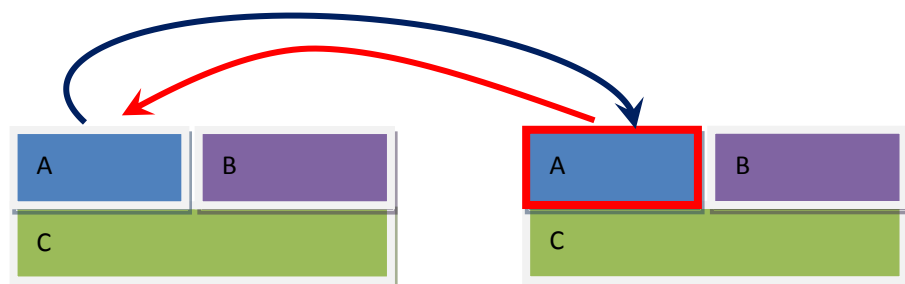
3.3.1 替换法

替换法是最基本的故障查找方法，也是最不需要技术水平的方法。

替换法使用的前提是：

有 2 台以上的整机，且有的整机有故障，但有的整机能正常工作。

比如发现电脑显示器没有显示时，马上换一个显示器看看有没有显示，如果有，就是刚才的显示器故障；如果也




没有，就可能是电脑主机故障；如果换下来的显示器在其他电脑上有显示，而新换来的显示器变得没有图像了，那么更加可能是主机坏了。

例如在上面的图中，每个系统由 ABC 三个部分组成，任何一个部分故障都将造成系统功能异常，带红框的 A 部分是故障系统，当左右两个系统交换 A 部分后，原本存在功能异常的右侧系统恢复正常，原本正常工作的左系统变得异常，且现象和原来的右系统一样，这就可以怀疑，是否是 A 部分出现了异常。

进一步确认这个问题，将刚才所做的交换动作撤销，左系统变得正常，右系统变回原来的样子，这就符合了我们“故障现象随着该部分迁移”的判据，故障确实源于 A 部分。

替换法即可用于查找**硬故障**，也可以用于查找**软故障**，但在使用的时候必须观察足够久，以便确保引发故障的条件出现（例如对开机 1 小时后显示器没有显示的电脑，更换一个显示器后应该观察至少 1 个小时）。

 **结论：** 当一个系统由若干个部分组成时，寻找故障的时候可以将怀疑引起故障的部分更换到其他系统上，通过观察故障现象是否随该部分迁移来判断 BUG 所在。

对替换法的理解不应僵化在对已有系统的硬件问题查找上，例如当我们改动了若干个软件模块，但没有改动这些模块的输入输出接口，程序运行却不正常了。此时，可以从备份程序中，将这些改动过的程序模块一一替换回“老样子”，从现象的改变来帮助我们快速判断是什么改动带来了异常。

3.3.2 模块隔离法

模块隔离法的理论依据是程序的模块化编写，每个模块有明确的功能分工，有明晰的输入输出接口。模块隔离法就是出现问题的模块单独出来调试，可以让调试的精力集中，目标明确，提高调试的效率。




案例 Example

某网友提问，自己做的 ARM 系统通过 U 盘进行软件升级，但是最后写到 FLASH 中的程序代码有错，他已经确认过从 U 盘读出的文件是正确的，实在找不到原因，所以来论坛求助。

分析这个问题，完成软件升级有 2 个步骤——从 U 盘读出和写入到 FLASH。FLASH 是不会“挑食”的，不管写入的数据是什么，都将照抄照考。既然作者已经确定了从 U 盘读出的文件是正确的，那么问题很可能就在 FLASH 写上了。

再者，既然写什么数据进去都可以，那么跳过 U 盘读出部分，自己用测试程序生成一个自加的序列写入 FLASH，用开发工具直接读出 FLASH 中的数据，就很容易检查写入部分是否正确了，这比起面对扑朔迷离的程序代码检查写入问题要简单得多了。

 **结论：** 将精力集中在需要查找的部分，排除其他模块的影响，用最简单的方法去判断，这就是模块隔离法的价值所在。



使用模块隔离法时不能忘记模块间是相互联系的，当被隔离出的模块单独测试无法发现问题时，就要考虑是否是模块间的相互影响导致了故障现象了。极端地，当模块内和模块间都存在问题时，这种怀疑精神更为重要！

3.3.3 出入口检查和变量跟踪

出入口检查的基础也是程序模块化，该方法的基本理论是——模块设计是正确的，每个模块又是正常工作的，那么系统就应该正常工作，反之，肯定能找到不符合设计要求的位置。这句话倒过来说就是——只要检查每个模块的出入口上是否正常就 OK 了。

3.3.2 中的例子本身就是一个出入口检查的例子，这个作者还是有检查的意识来判断“从 U 盘读出的文件是正确的”，这也是我们在论坛上回答问题时初学者常犯的错误。比如在上面的例子中，如果作者问的是“写入 FLASH 的数据就是不对，请问怎么回事”，那么回帖必然是“能确认读出的文件是正确的么？”

出入口检查，使用的方法多种多样——

对纯的计算类模块，最简单的方法是用 IDE 提供的软件仿真功能，直接输入入口数据，在出口处设断点检查。

当系统有显示、按键时，可以通过这些资源写测试程序，控制程序流向和执行条件，输出变量数据，还可以在程序中设置测试代码输出来跟踪程序的运行流程。

3.3.4 区分时间和空间相关性

我们前面已经说过，程序分为顺序程序和含有中断的程序两大类。顺序程序在 BUG 查找上相对简单，因为各模块的前后衔接顺序清楚明了；含有中断的程序在 BUG 查找的时候需要多动一个脑筋——BUG 发生在时间上还是空间上？

有很多时候，在含有中断的程序中，我们把各个模块独立开来测试，他们的运行都正确，当组装到一起后，就是要出问题，这时候就应该考虑是否是时间上存在问题了。我们来举一个简单的例子说明什么时程序的时间相关性



案例 *Example*

有这样一个串口接收程序，作者的原意是建立单片机系统和上位机之间的交互通道，在单片机向上位机发送数据包的同时，如果收到上位机新发来的数据包，通过发送 0xEE 向上位机标明单片机正在忙碌，请求上位机暂停发送。

```
unsigned char flag_uart_tx_ing=0; //串口发送中标志

_isr_uart_rec() //串口接收数据包中断程序
{
    if(flag_uart_tx_ing)
```

```

        uart_byte_send(0xEE);
    else
        ..... //数据包接收和处理
}

void uart_send(unsigned char *p) //串口数据包发送程序
{
    .....
    uart_tx_buffer = *p;
    p++;
    while(!uart_tx_i); //等待串口发送完成中断标志
    uart_tx_i = 0; //清除串口发送完成中断标志
    .....
}

void uart_byte_send(unsigned char a) //串口发送单个字节程序
{
    uart_tx_buffer = a;
    while(!uart_tx_i); //等待串口发送完成中断标志
    uart_tx_i = 0; //清除串口发送完成中断标志
}

```

这个代码将导致当机。

当单片机正在执行“串口数据包发送程序”过程中收到上位机来的数据包，且不说这将导致正在发送的数据包中断，而且可能导致当机的严重后果。很明显这个程序的发送基于查询方式进行，如果 `uart_tx_i` 标志不为 1，程序将始终在 `while` 循环处死等。对顺序程序而言，没有其他程序流程干扰的情况下，在填充了串口发送缓冲区 `uart_tx_buffer` 后，数据最终会被发送完毕，该标志置 1，但在有接收中断内发送 `0xEE` 的条件存在时，`uart_tx_i` 标志可能被发送 `0xEE` 程序“抢先”清 0，从而导致中断外的发送程序永远等不到该标志置 1，造成死机。

如果不从时间上并行的方式思考问题，而是单独测试每个程序模块，很难联想到标志被意外清除这个问题。这就是程序的时间相关性。

至于空间相关性就好理解了：顺序运行的程序间，程序的运行条件被改变的情况就是程序的空间相关性。比如有一个全局变量 `a`，初始化值为 0，本来是留给第二段程序使用的，但是第一段程序误将其赋值为 1，那么第二段程序只要使用了初始化值 0，就必将出错。

👉 结论：发生时间相关性 BUG，其分开测试正确，合并测试出错的特性往往让初学者抓狂，在调试含有中断的程序时，一定要多条思路，怀疑程序的时间相关性。

严格来说，**一切出现的 BUG 都是由于没有遵守一些基本规则而造成**（虽然有时候是粗心和笔误），常见的例如：

- 【a】 中断现场保护有遗漏，常见于汇编编写的程序，例如 PIC16 单片机的 DPTR 间址指针，用 C 语言编程时，中断内外使用了相同的全局变量等。
- 【b】 使用了临界资源而没有采取任何保护措施，例如前面串口的例子中，发送完成中断标志就是一个临界资源。
- 【c】 硬件上有严格时序要求，软件虽实现了算法，却没有配上时序的情况，例如打印头控制。

找时间相关性 BUG 可以从两个方向入手——正向清理自己在上面 3 个方面有没有失误，反面通过“时间整垫”的方式来寻找更多的现象支持推理。

所谓“时间整垫”是指用没有（或尽量没有）空间相关性的代码插入到被测代码中，以测试所发生的故障现象是否发生改变的方法。这里又可以将时间相关性程序区分为两类：

- 【a】 自相关类时间相关性，与外部事件无关的时间相关性，程序不等待某个外部中断或按键等外部指令，例如由内部相关的定时器之类触发中断。
- 【b】 外部相关类时间相关性，与外中断或按键等人机交互环节相关，例如打印头同步信号控制的打印针动作。

对于自相关类时间相关性，由于程序执行和定时器运行实质上都是由主时钟（通常是晶体）驱动的，所以定时器发生中断的时间点实际上是确定的，通过“时间整垫”可以方便地改变中断发生的位置。




案例 Example

```
func_a();  
  
delay_ms(1);    //时间整垫  
  
func_b());
```

通过在出现问题的 a、b 两个函数之间加入延时函数，“错开”发生 BUG 的位置，如果 BUG 消失或者现象发生变化，那么就要注意是否是时间相关性问题，并且可以进一步确定 BUG 的位置。

对于外部相关类时间相关性，由于外部事件和程序运行不可能精确同步，故障现象也往往是“偶发性”的。这种程序可以通过时间整垫的方式进行测试，但效果不佳，这时可以考虑尽量排除相关因素来缩小查找范围。

 结论：因为只有在实际运行时才能表现出来，时间相关性 BUG 比较难于查找，这也就是我们建议程序在实际环境中调试，而不是仿真环境下调试的原因之一。遵守基本规则编写程序是避免时间相关性 BUG 甚至所有 BUG 的根源。多观察现象，积累判断依据是找时间相关性

BUG 的法宝。

3.3.5 故障拦截

其实我们前面所讨论的东西都属于故障拦截。故障拦截包括两方面的意思——

- 【a】寻找更多的现象以支持推理
- 【b】对软故障提供一套保护现场的机制

特别是软故障，因为其出现是不固定的，找推理支撑就变得更困难，一般来说有这些经验可以遵守：

- 【a】通过软件的方法监控怀疑的变量，在发现异常时，通过特殊提示（例如蜂鸣器叫、屏幕显示特殊的内容）等提示。
- 【b】发生当机时，不要简单臆断下电复位，要想方设法保持系统当机状态，全面记录发生当机前系统所作所为（即发生了什么，输入了什么，输出了什么），全面记录当机后单片机各相关（甚至有时需要记录自己认为不相关）的端口电平状态。
- 【c】在没有理顺整个推理过程前，坚持怀疑的原则，以免被误导。

3.4 其他一些容易出现 BUG 的地方

我们首先来看一个简单的原则——

世界上只有神不犯错，人做的事情都是有可能出错的。

既然单片机是人做的，编译器是人做的，操作系统是人做的，那么就不能排除他们犯错的可能性。既然使用了别人所提供的产品，就等于承认了别人可能犯下的 BUG，换句话说，有时候 BUG 的根源可能真的不在你身上！



案例 *Example*

使用 SST 公司（目前已被 MICROCHIP 收购）52 单片机，P4 端口状态无法读入到 A 寄存器中，我们用只有几行的测试程序进行测试，已经完全排除了其他程序代码影响的可能。在这种情况下，我们不得不向厂家技术支持求助，得到的回答是——这是一个 BUG，该机型的 P4 端口状态只能读入到 B 寄存器中。

在这个年代做技术，最好的莫过于互联网，在使用芯片前到厂家网站下载最新的勘误表，多使用 Google 和厂家的电子邮件支持是避免在芯片 BUG 上浪费时间的有效方法。



案例 *Example*

这是一个容易混淆视听的例子，在 KEIL 环境下用汇编开发的 51 单片机程序，居然出现了在上电初始化内多加一行语句就造成程序当机的情况！这个情况首先让人感觉不可思议——因为所加入的语句仅是向 A 寄存器赋立即数而已，没有任何理由影响到程序的运行。

在不知道方向的情况下，我们将这行语句换为“与世无争”的 NOP 语句，以测试该 BUG 与加入的语句是否存在关联，奇怪的是，NOP 语句同样会造成当机！现在这个 BUG 很像是一个时间相关性的了，但是我们仍然想不到任何可以关联的地方。

我们再继续加入若干个 NOP，发现加入 2 个、3 个和 4 个 NOP 的时候，程序正常运行，加入 5 个 NOP 的时候，程序出现当机。虽然这个判据已经很像时间相关特性了，但我们反复确认了，初始化阶段，总中断使能是没有打开的！

在没有任何头绪的情况下，我们坐下来翻指令表玩，发现我们每天都在使用的 CALL 指令，在指令表里并没有出现！指令表中的调用指令是 ACALL 和 LCALL，一个念头一闪而过——会不会是调用出现了越界，而编译器没有给出警告呢？

为了验证这个问题，我们先将所有的 CALL 改为 ACALL，在编译器的帮助下将越界的指令改为 LCALL，保存编译得到的 HEX 文件，和原来用 CALL 指令时生成的 HEX 文件进行比对，发现他们是不一样的，这就验证了 KEIL 编译器的确使用了名称为 CALL 的伪指令，且不能正确将其匹配为 ACALL 和 LCALL，新加入的语句改变了调用的距离，程序在调用时出现了越界从而导致当机。

好了，现在我们已经看过了各种 BUG，最后一句话是——金钟罩强过还魂丹，与其等到 BUG 出现再费劲寻找，不如在编写前认真规划，在编写中遵守基本规则，分模块认真测试，确保程序不出 BUG 来得省心！

全文完

作者简介：

江海波，82 年的狗狗，因税控机（GB18240）项目（悼念）而有幸使用多种单片机和外设，用过的单片机包括 TOSHIBA 900L 系列 16 位机、SST51、AVR（Mega48、8、16）、PIC16、ARM7 等，用过的大外设包括 EPSON M-G120、M-U110II，VFD，多种 LCM，多种串、并行存储器，软件模拟 ISO7816 卡接口等，编程语言包括汇编、C、VB 等，因同时进行软硬件设计，故擅长生产和消灭各种 BUG。

作者目前研究方向为 ISM 波段无线数传技术，已完成了 CC1020、SI4432 无线模块，我公司主要产品为 IC 卡智能气表、无线远传气表、IC 卡智能水表等三表产品和物联网产品。我们的网站是：

成都前锋集团-应用产品事业部

www.chiffo.com

Soundman@sohu.com（私人邮箱）