自制一个由你掌控的
—— 超声波测距传感器（软件篇）

一、概述

本篇系“自制一个由你掌控的超声波测距传感器（硬件篇）”的续篇，着重介绍如何设计让前述硬件工作的软件，如何用人的智慧激活下面这个本不具“灵性”的东东，让读者感受一下“智能传感器”的魅力。

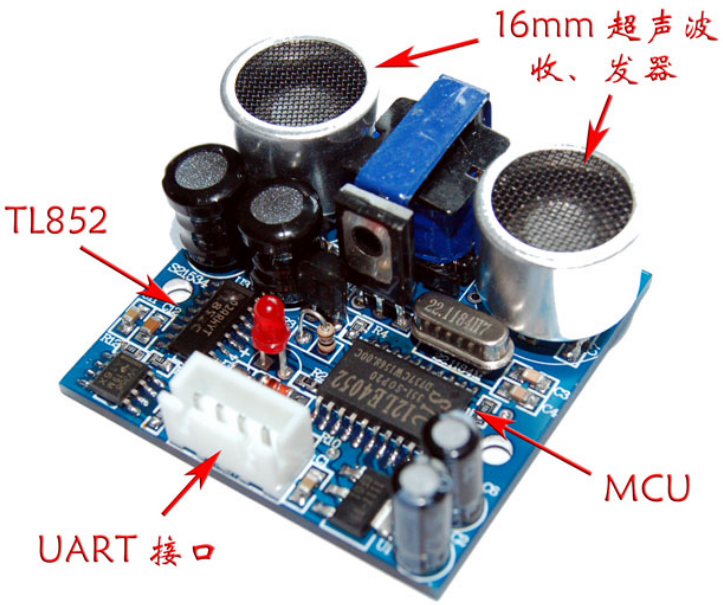


图 1 将被赋予“智慧”的东东

二、需求分析

测距传感器的核心功能是测量距离，但当其用于不同场合时，会有许多不同的需求。

如果是**传统的传感器**概念，只需将“**非电量转换为便于测量的电量**”即可，这是一个比较通俗也基本正确的定义，“便于测量的电量”通常为：直流小电流、小电压以及方波等。

这类传统传感器给系统带来了不少“麻烦”，因为其输出的所谓“便于测量的电量”只是物理上的，充其量达到“可测”而已，由于其输出的不统一、不灵活，甚至有些“粗糙”，使得系统不得不付出一些开销去弥补之。

就拿 GP2D12 来说，其输出是直流电压，可与距离的关系是非线性的，且是反比例，输出还是非连续的，就这三个特征就足以让系统耗费不少周折才能得到想要的距离值。

还有很多类似的例子，如热电偶温度测量传感器、光敏传感器等，在此就不一一枚举了。

从系统设计的角度考虑，最好是传感器将所测量的量转换成数字信息，系统不必再去理会这些“底层”的处理，专心于功能的实现。如同现在的 PC 操作系统，有统一的设备接口，系统级应用是“与硬件无关”的，设备的差异由各设备厂家通过驱动程序实现统一。

以往由于技术和成本的限制，为了节省开支，将很多功能都交给了主控系统完成，形成所谓“树形”架构，只有“主干”是有智能的，其余都是“末梢神经”，只具备最低级的信号采集能力，也就是传统传感器的角色。

随着单片机的功能提升、价格下降，新的构架方式逐渐显现：一个系统中，每个部分都自成体系，主控只是负责策略、协调，各个独立的功能模块“自行其事”。这就是“分布式”系统。

分布式系统概念的普及，催生了智能传感器的需求。

所谓“**智能传感器**”，至少有以下**特征**：

1) 能够将被测量转换为数字值，而非简单的模拟量；

2) 能够根据要求独立完成测量；

3) 能够通过数字通讯接口接受命令、输出数据。

具备此特征的传感器已有很多，有些已制成 IC，如常见的温度传感器 18B20。

智能传感器除了降低了系统的软硬件开销外，附带的一个好处就是便于传送，传统传感器的输出信号传输时的“干扰”和“衰减”是最令设计者头痛的！

因此，智能传感器是未来的方向。实际也是如此，读者可搜索一下新兴的 MESM（微机电系统）传感器，不论是两轴、三轴加速度，还是陀螺仪等，新产品几乎都是 I2C、SPI 等数字总线接口。

所以，我们这个超声波测距传感器也是按智能传感器理念设计的。

因本篇只是示范性软件设计，没有特定的应用场合，所以只好就测量本身来定义需求：

在**性能上**，测量关注两方面：一是得到数据的速度，二是数据的可靠度。

在**功能上**，测量有两类：一是不断的测量并输出结果，二是触发后开始测量。

所以至少应满足上述需求：

1) 可设置为快速测量模式，让系统最快得到测量数据；
2) 可设置为精确测量模式，返回给系统比较可靠的数据；
3) 可以设置为连续测量模式，不断提供给系统测量数据；
4) 可以按照系统请求开始测量，返回即时数据。

三、功能设计

按上述需求，传感器的功能设计如下：

传感器上电处于待命状态，等待系统命令做以下操作：

1) 可以支持连续测量，并存放最近 8 次数据，测量周期可以由系统设置。在此状态下，系统根据需要读取数据。
2) 可以支持连续测量，并且将每次的数据返回给系统，由系统进行需要的后处理。
3) 可以接受系统命令，返回待命状态。
4) 可以支持单轮测量，即系统发出命令通知传感器，采集几次数据，传感器可做基本的数据处理，如取平均、剔除最大最小值，完成后返回这组数据后，恢复到待命状态。

此外，为了便于调试，增加读、写单片机内存的功能。

四、详细设计

4.1 题外话

看懂别人软件是件相当困难的事，即使那些较正规的、有完善文档的项目，也不是十分轻

松，因为记录下来的只是结果，思维的过程无法再现，而读者有时更多关注的是如何“想到的”，特别是初学者!

但描述软件的构思过程也并非易事!

前期我写过的“圆梦小车 StepbyStep”系列文章中，尝试通过一步步“搭建”的方式来引导读者理解思考过程，并在程序中特别注释了每一步所增加的内容，程序中排版顺序都放弃了逻辑关系而“屈从”于“搭建”的顺序，可似乎收效甚微！？猜测是没有交代最基本的思路所致，因为即使是每一步都很具体，读者仍会问：**怎么来的？为何?**

本文不是技术论文，其目的是帮助有意学习者实现自己的愿望，所以本篇尝试简述一下思考方式，看是否对学习者有帮助，但声明一点：此乃个人观点，并非“宝典”，不保证正确，仅供参考!

4.2 程序构建思考过程

我开始涉足单片机编程时，由于只有汇编语言可用，且编译环境较弱，变量名、标号限制较多，所以那时很讲究使用流程图来表达程序的构思，因为从汇编代码上看懂程序实在困难，毕竟那是为机器思维服务的逻辑顺序，与人理解所需的表述相差甚远。

当我转换到 C 语言编程时，开始还保持着画流程图的习惯，但逐渐觉得有些多余，因为 C 语言的自注释性（即语句和变量名的组合表达方式已接近人的理解需要）以及编译环境的提升，配合各类几乎无限制的定义手段，使程序本身就可以方便的为人所理解。如今编辑器也在优化，读者可以尝试一下 UltraEdit，其“折叠”、“展开”功能十分有助于理解程序的思路。所以渐渐的放弃了流程图。但还维持着按实现过程来构建程序的习惯。

自从我尝试编写 PC 环境下的 VC 程序后，逐渐构思习惯有了很大变化，读者如果没有尝试过，可以参照“圆梦小车 StepbyStep 之二”做一次，然后再用类似的方式构建几个自己想象的题目，一定会有所感受!

在 VC 中**构建**一个**程序**，其**过程**大致如下:

1) **设计功能** —— 这是机器所不能代替的，靠你的创造力实现之，需要用文本记录之;

2) **构思界面** —— 这就是 VC 为你提供的方便了，根据功能和工具可以实现你所要的界面
3) **变量定义** —— 构建界面时 VC 会自动生成变量，根据功能对这些变量进行类型定义；
4) **编写处理程序** —— 基于界面所产生的操作（按钮等）编写对上述变量进行处理的程序，这是你的智慧展示的空间。

在 PC 上编程（默认是 Windows 下），由于很多事情都由 Windows 操作系统帮助做了，所以在 VC 环境下编程确实比较轻松，只需关注和功能相关的事，无创意的琐碎事务都由系统和 VC 处理了。

单片机中虽没有这么“美”的事，但是这种构建过程倒是改变了我，我现在基本也是按此思路去构建一个程序，只不过一些 VC 帮助自动生成的过程由自己完成了。

首先，是**确定**所做的东西要完成哪些**功能**，这是基础。在需求分析和概要设计阶段基本搞定，在详细设计的开始处将其具体化，用技术**术语**表达之。

之后根据这些功能**定义**相应的**变量**。如需要记录 8 次测量数据，就需要有一个 8 元的数组，同时要有存放指针和取数指针（注意：此处所述“指针”，非 C 语言的指针，是指数组的下标，只是个人表达习惯而已，下同），以便于对数组操作。

根据硬件的性能和需求确定数组的类型，是用整型还是字节型等；因为我定义的测量范围为 5 米，即使用 cm 为单位字节型也不够，所以用整型。因为不可能有负数，所以用无符号整型。数值表达范围大了，将单位提高到 mm，虽然不一定需要，但不增加工作量，感觉却好多了 :P 何乐而不为?

在定义变量的同时，**定义**一些与变量处理相关的**常量**，一方面为了程序的可读性，同时也是为了日后便于修改。如现在设计是保留最近 8 次数据，但日后也许需要更多或较少，将数组的单元数声明为符号常量 —— DATA_SAVE_NUM，定义为 8，需要时只需修改此处定义即可，不用在程序中“遍地”去找，遗漏一个就形成一个 bug!

在构思服务于功能的变量时顺便考虑如何处理之，还是拿测量数据存放为例。既然需要存

放最近 N 次的数据，可以这样处理：存满 8 个之后依次向前移，覆盖序号最小的单元，腾出序号最大的存放新数据。这样处理效率太低，常用的方式是环形缓冲区的概念，即将数据存放区看成是个首尾相接的环，存放数据时指针不断“加”，到尾时自动环到首，不用任何数据搬家操作，而取数也是同样，只是从存数指针向回“减”，到首时自动环到尾。

基于这个思路，自然 2 个指针变量的需求就产生了。这两个指针据需要能够“加”到尾环头，或“减”到头环尾。如凭直觉，就用比较的方式判断，每次运算都作一次检测，虽能完成，但似乎有些繁琐。考虑一下有无更好的方式？如果还记得二进制的基本运算，就可以理解我为何在程序中设计环形数据存放区的时候均要求单元数是 2 的幂，即 4、8、16……

按上述方式，可以依次定义出功能用的变量。之后就要结合运行定义一些处理用的变量，这就是 VC 和操作系统可以帮你完成的那部分。

嵌入式系统（或俗称“单片机系统”）有以下几个概念：

1）**死循环**

嵌入式系统是连续不断运行的，所以必然有一个“死循环”，一般用以下程序实现：

```
While（1）
{
……
}
```

2）**标志驱动**

对于嵌入式系统，小的应用一般没有操作系统，内存不够，开销也大，所以需要自己设置一些标志，以实现类似于 PC 上的“消息驱动”。

在上述循环中，程序顺序检测各个标志，根据标志做相应的处理。标志的建立一般由中断完成，或者是中断处理后产生。

3）**时基**

多数程序都有按时间处理的需求，如延时、定时查询、周期测量等等，所以一定有一个时

基，由定时器中断产生，建立标志。在 PC 中也有类似的机制，DOS 时代的 PC 我知道系统有一个 18.2ms 的时钟信号，现在的 PC 不太清楚了 :(

我设计系统通常使用 1ms 时基，因为多数处理不会小于这个间隔。由此，要设计一个 1ms 中断标志。所有与时间相关的处理都安排在 1ms 中断标志建立后的处理中。

4）**状态**

一个功能的实现，往往不能“一蹴而就”。

就拿超声波测距来说：首先得发出超声波，之后等待回波，等待过程中要根据时间控制增益，逐渐增大，以弥补声波随距离增加的衰减。收到后再计算，至此才能得到一个测量结果。

按照声波速度，2m 距离约需要 12ms（来回 4m）。如果程序设计成顺序依次处理模式，直到完成后再处理其它的任务，则大概通讯成功率只有一半了，因为 MCU 的处理给测距过程独占了。

也许有人说：可以用中断来处理。

在此，顺便说一下：在成熟的程序中，**中断处理中尽量少安排操作**。因为一是会由于中断内、外同时处理相同变量产生错误，除非你设计了严格的“闭锁”机制；二是降低了其它中断的响应速度，也许会导致程序性能下降；如需要精确捕捉脉冲，则会由于延时响应而降低精度。优先级机制虽可弥补，但那更容易导致数据错误，而且需要更多的堆栈空间。

为了避免上述问题，通常使用状态来标注一个功能做到了哪一步？设置一个状态变量，记录功能实现的进程，每次轮回时根据状态做相应的事，把能做的做完后立刻退出，把 MCU 的处理能力交给别的任务。

这就是“**分时**”处理的概念，只不过分时是由自己写的程序所调度，而非“操作系统”，随着所做的项目复杂度加大，你会感到“操作系统”的重要!

而利用状态控制大概就是所谓的“**有限状态机**”，这在嵌入式系统中是常见的。

具体实施时，采用螺旋式步骤完成程序。

先构建一个最基本的程序框架：（这一步在 VC 中，建立 MFC 工程时就自动完成了，可在

单片机上，得自己为之 ☹)

```
152
153  void main(void)
154  {
155    init_hardware();
156
157    init_var();
158
159    EA = TRUE;     // 启动中断，开始正常工作
160
161    while(1)
162    {
163      /*--------- 时基处理 ---------*/
164      if(g_b1msFlag)
165      {
166      }
213
214      /* --------- 通讯处理 ------ */
215      if(g_bNewData)
216      {
217      }
227
228      /* --------- 工作状态处理 -------- */
229      switch(g_ucWorkStat)
230      {
231      }
480    }
481  }
```

图 1 程序主框架

（使用图片表示并非想阻碍拷贝，只是想借用编辑器的彩色表示方式，更直观，下同）

从上图可以看出，程序基本上由初始化和死循环组成，死循环中设计了三个处理：时基、通讯、测量（工作），这三件事因为需要同时处理，所以设计在循环中依次得到 MCU 的处理时间。读者可回忆一下流程图格式，有了这个还需要吗?

然后逐步充实，每构思一个功能：

1） 定义相关变量

2） 在 init_var()中初始化；

3） 根据需要定义相关常数；

4） 涉及硬件，在 init_hardware()中初始化硬件；

5） 构建处理方法，也就是函数，完成功能；

6） 如完成需要等待外部条件，则定义状态变量，并定义状态，转换条件；

7） 处理时如涉及定时，则设置计时器，建立相应标志，在时基处理中添加处理；

不一定要一个螺旋就上到“顶”，可以逐步添加，首先是实现功能必备的处理，其次是防

护出错的保护性处理。程序将逐步变得完善、可靠。最终得到的程序自然比较复杂，而读程序者通常看到的是这个版本，所以自然费解！

读者如果接触过编程，一定知道“**面向对象**”，这个概念开始时我很不理解，汇编程序写多了，思维更“接近”机器。在编写几次 VC 程序后，觉得“**面向对象**”倒是一个不错的**思维方式**，即从你要做的事情开始思考，它要完成什么？它有什么特征？它怎么去做？可以类比一下，上述**功能定义**说明了它要完成什么？**变量定义**说明了特征，而**处理方法**指明它如何去做。也许不太贴切，但是自我感觉有不小的帮助。

4.3 软件设计

言归正传，下面开始“务实”。

为便于读者理解，将工作原理框图再次列出：

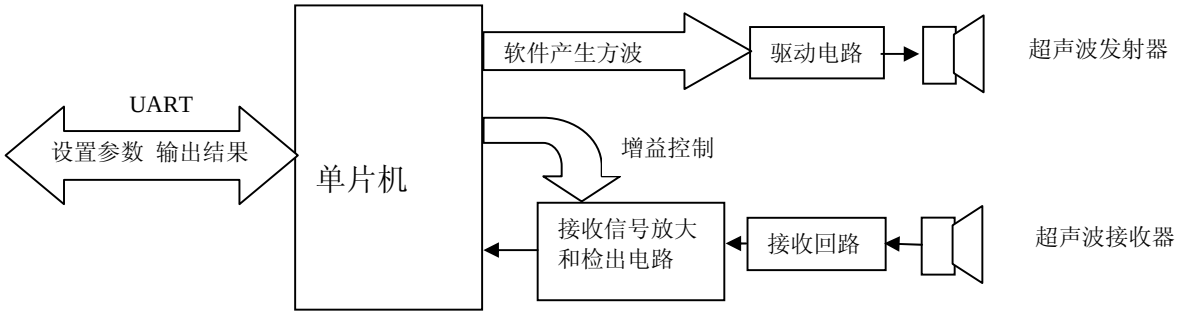


图 2 工作原理框图

具体**功能描述**如下：

传感器的工作由通讯命令控制，**上电状态为待命状态。**

工作分为“**自动测量**”和“**单轮测量**”两种模式。

“自动测量”时，传感器按一定周期自动完成测量过程，并保存测量数据。

“自动测量”又分为“被动数据返回”和“主动数据返回”两种方式。

“被动数据返回”方式下，传感器只将测量结果保存下来，等待系统读取。

“主动数据返回”方式下，传感器每完成一次测量均立即将数据发送给系统。

“自动测量”可以设置测量周期。

“单轮测量”为接收到命令后开始测量，并返回数据，测量命令可设置测量次数、数据处理方式，传感器按要求返回，增加测量的可靠性。

在硬件设计中，已将 MCU 的 I/O 口分配，软件设计则需要将 MCU 的内部硬件资源分配，以完成相应的功能。

实际上，有些 I/O 口定义后，其相应的内部资源就确定了，如中断输入。

MCU 内部资源分配：

INT0 —— 保留；

INT1 —— 作为收到超声波回波中断；

Timer0 —— 产生 1ms 时基；

Timer1 —— UART 波特率发生器；

PCA0 —— 保留；

PCA1 —— 发射时作 40 KHz 方波计时，接收时作增益变换计时，采用同样的工作模式；

UART —— 串行通讯，命令输入和数据输出；

按上述功能，程序由两个部分构成，

通讯 —— 负责接收命令，返回数据

测量 —— 负责超声波发射、回波检测、计算和数据处理

为了让程序可靠运转，各功能所需的定时要求得到满足，增加时基处理。程序的基本框架如图 1 所示。

4.3.1 通讯功能

凡是设计通讯方面的软件，首先是定义协议，以及协议所承载的内容！

通讯协议及命令定义：

基本通讯格式：（和圆梦小车及无线接口兼容）

标准 UART 格式 —— 19200 8 N 1

帧格式：

帧头（2 字节） 接收方地址（1 字节） 发送方地址（1 字节） 帧长（1 字节）

命令（1 字节） 数据域（N 字节） 校验和（1 字节）

其中：

帧头 —— 由 2 个特殊的字节 0x55 0xAA 构成；

接收方地址 —— 通讯对象的“名字”，在有线通讯时也许多余，但无线时就需要了。

发送方地址 —— 告诉接收方，便于接收方回答。

帧长 —— 命令和数据域字节之和，

命令 —— 说明操作内容，详见下面的定义

数据域 —— 与命令配合，表达一个完整的含义。

校验和 —— 从命令开始到数据域结束所有字节的算术和，取最低字节的反码。

命令定义：

为了便于调试，保留小车中设计的读写内存命令。

✧ **命令一 ： 读内存**，实现读指定地址开始的 N 个字节，地址用两字节表示。

命令字 —— 0x01

数据域 —— 低地址（1 字节） 高地址（1 字节） 读字节数（1 字节）

地址与硬件的对应关系：

0x0000 — 0x00FF —— 对应 STC12LE4052 的 256 字节内部 RAM（idata）；

0x0100 — 0x7FFF —— 保留，为大 RAM 的单片机预留；

0x7F80 — 0x7FFF —— 对应 STC12LE2052 的 128 字节 SFR；

0x8000 — 0x87FF —— 对应 STC12LE2052 的 2K FlashROM（Code）；

0x8700 — 0xFFFF —— 保留，为大 ROM 的单片机预留；

例：要读地址 0x56 起始的 3 字节内部 RAM 数据，命令帧如下：

0x55 0xAA XX XX 0x04 0x01 0x56 0x00 0x03 0xA5

返回数据帧为:

帧头 发送方地址 自己的地址 帧长 命令 低地址 高地址 读字节数

N 字节数据 校验和

返回帧中将命令及附属信息（地址、读字节数 ）包含在内，虽然有些冗余，但保证了信息的完备性，不需要接收时还要查找原来读的是什么？为通讯需求日渐复杂提供方便。

✧ **命令二：写内存**，实现写指定地址开始的 N 个字节，地址用两字节表示。

命令字 —— 0x02

数据域 —— 低地址（1 字节） 高地址（1 字节） 写字节数（1 字节）数据（N 字节）

其地址与硬件的关系与读命令相同。

返回数据帧为：

帧头 发送方地址 自己的地址 帧长 命令 低地址 高地址 写成功字节数 校验和

通过“**写成功字节数**”来告之发送方是否写成功，如果为“0”，表示写操作失败。

✧ **命令三：设置工作模式**，设置测量模式，并启动测量。

命令字 —— 0x03

数据域 —— 工作模式（1 字节） 工作参数（1 字节）

其中：

工作模式 —— 高 4 位为主模式，低 4 位为子模式；

工作参数 —— 与工作模式对应，自动测量时为测量周期，单位 10ms；单轮测量时为测量的次数，暂定最多为 8 次。

返回数据帧为：

帧头 发送方地址 自己的地址 帧长 命令 测量数据 校验和

对于自动模式，测量数据返回一个字节“0”，说明 OK。

对于单轮测量模式，测量数据为工作模式中所要求的数据。

自动模式对应的子模式：

a) 自动数据返回 —— 每测完一个数据都返回，命令位置返回工作模式；

b) 被动数据返回 —— 内部只是测量、保存数据，等待读数据命令读回，读数据命令实际上只对自动模式的被动数据返回有效。

单轮测量对应的子模式：

a) 无数据处理返回 —— 只是将命令中要求测量的数据全部返回

b) 剔除最大最小值返回 —— 将测量数据中最大、最小值剔除，返回数据比指定数少 2 个，当指定的测量次数小于 3 时，强制切换到“无数据处理模式”；

c) 平均值返回 —— 将指定的测量数据平均后返回，只有一个平均值；

d) 剔除最大最小后平均值返回 —— 先剔除最大最小值，剩下的再取平均值，当数据数小于 4 时，强制切换到“剔除最大最小值” 模式。

上述返回帧中命令位置返回数据处理方式。

✧ **命令四：读测量数据**，对于自动测量方式的被动数据返回模式，需要此命令。

命令字 —— 0x04

数据域 —— 读最近几次的数据（1 字节）

返回数据帧为：

帧头 发送方地址 自己的地址 帧长 命令 测量数据 校验和

通讯是个过程，包含等待处理，所以需要定义状态变量，以控制处理的内容和方式。

通讯分为两个状态：一是什么也没有收到；二是收到了帧头，等待帧结束。因此可以用位标志 g_bStartRcv 来表示，为“假”表示没有收到帧头，为“真”表示收到帧头，正在收剩余的内容。

通讯部分实现以下功能：

1） 接收数据

2） 判断帧格式

3） 解析命令，执行相应操作

4） 返回数据

根据上述功能可以看出，首先要为接收**设置变量**：

```
unsigned  char  idata ga_ucRcvBuf[MaxRcvByte_C];    // 接收缓冲区
unsigned  char  data  gi_ucSavePtr;                 // 存数指针，每收到一个字节保存到缓冲区后加“1”。
```

为了判断帧格式，解析命令，必须有相应的取数变量：

```
unsigned  char  data  gi_ucGetPtr;                  // 从缓冲区中取数的指针，每取出一个字节后加“1”。
unsigned  char  idata gi_ucStartPtr;                // 帧起始位置，指向数据区开始。
unsigned  char  idata gi_ucEndPtr;                  // 帧结束位置。
unsigned  char  idata gc_ucDataLen;                 // 帧长，即数据区字节数。
bit                   g_bNewData;                   // 串口收到一个字节标志,为减少变量交互。
bit                   g_bStartRcv;                  // 开始接收数据帧标志。
```

为了返回数据，需要设置发送用变量：

```
unsigned  char  idata ga_ucTxdBuf[MaxTxdByte_C];    // 发送缓冲区，用于返回数据。
unsigned  char  data  gi_ucTxdPtr;                  // 发送指针
unsigned  char  data  gc_ucTxdCnt;                  // 发送字节计数

unsigned  char        g_ucSenderAddr;               // 保存命令发送者的地址，为了自动返回数据
bit                   g_bTxdFinish;                 // 数据发送结束标志
```

上述变量所需的**常数定义**：（注意缓冲区大小）

```
#define       MaxRcvByte_C    32          // 接收缓冲区大小
#define       MaxTxdByte_C    32          // 发送缓冲区大小，设置单独的发送缓冲区是为了能全双工通讯
```

变量初始化：

```
/* 初始化UART 部分变量 */
gi_ucSavePtr = 0;
gi_ucGetPtr = 0;

g_bNewData = FALSE;
g_bStartRcv = FALSE;
g_bTxdFinish = FALSE;

ga_ucTxdBuf[0] = 0x55;                  // 因为发送缓冲区暂不作他用，所以先存好，以减少代码
ga_ucTxdBuf[1] = 0xAA;                  // 帧头，
ga_ucTxdBuf[3] = MY_ADDR;               // 自己的地址作为发送方地址送出
```

因为涉及硬件，所以还要**初始化硬件**：

```
/* 初始化UART  */
init_SIO(B_19200);
IE = IE|EnUART_C;                   // 允许 UART 中断
```

```
/*****************************************/
/* 名称: init_SIO                        */
/* 用途: 初始化串口,                     */
/* 参数: 波特率 , 模式固定为:1           */
/*    1 START 8 DATA 1 STOP              */
/*****************************************/

void init_SIO(unsigned char baud)
{
  // 波特率表
  unsigned char code  TH_Baud[5]={B4800_C,B9600_C,B19200_C,B38400_C,B57600_C};

  AUXR = AUXR|SET_T1X12_C;
  TH1 = TH_Baud[baud];
  TL1 = TH_Baud[baud];
  TR1 = TRUE;

  SCON  = UART_MODE1_C|EN_RCV_C;  // 8 位模式( MODE 1)
}
```

注意：上面所有涉及寄存器初始化的参数全部用的是符号常量，这样看似麻烦，但大大改善了程序的可读性，同时也增加了可靠性；因为定义这些符号常量时不需要考虑逻辑，只需对照手册认真键入即可；而编程时只需引用那些直观的名字，不必翻阅手册。

往往是一边思考编程逻辑一边查手册时出错，因为要一心二用！

程序中的常数定义：（关于 MCU 寄存器的太多，就省略了，请看源代码）

```
// 波特率定义
#define       B_57600         4
#define       B_38400         3
#define       B_19200         2
#define       B_9600          1
#define       B_4800          0

// 在 22.1184Hz 下用 T1 作波特率发生器, 1 分频, T1 自动重加载值
#define       B57600_C        244
#define       B38400_C        238
#define       B19200_C        220
#define       B9600_C         184
#define       B4800_C         144
```

上面算是把通讯的准备工作完成了，接下来构建通讯的处理程序。

通讯采用中断方式接收、发送数据，在中断中只做必须的事，如：将收到的数据保存到接收缓冲区、建立标志、将发送缓冲区的数据送出、送完后建立标志。

主循环中处理如下：

```
222     /* -------- 通讯处理 ------ */
223     if(g_bNewData)
224     {
225       g_bNewData = FALSE;
226       if(dataFrame_OK())          // 串口收到数据后的处理, 与PC程序中的函数 DataFrame_OK() 相当
227       {
228         g_ucWorkDispStat = UART_OK;                              // 指示收到命令
229         gc_ucWorkDispNumCnt = ga_ucWorkDispNum[g_ucWorkDispStat];
230         gc_uiWorkDispTimeCnt = ga_uiWorkDispTime[g_ucWorkDispStat];
231
232         do_Command();             // 执行数据帧中的命令
233       }
234     }
```

中断处理如下：

```
/*******************************************/
/*          串口中断服务                   */
/* 说明:  将收到的数据保存到接收缓冲区     */
/*******************************************/

void  SioInt(void)  interrupt 4  using 1
{
  if(RI==TRUE)
  {
    RI=FALSE;
    ga_ucRcvBuf[gi_ucSavePtr]=SBUF;                    // 将数据填入缓冲区
    gi_ucSavePtr=(gi_ucSavePtr+1)&(MaxRcvByte_C-1);    // 利用屏蔽高位的方式实现指针的环形处理
    g_bNewData = TRUE;
  }

  if(TI == TRUE)
  {
    TI = FALSE;                                        // 处理发送
    gc_ucTxdCnt--;                                     // 发送计数
    if(gc_ucTxdCnt>0)
    {
      gi_ucTxdPtr++;
      SBUF = ga_ucTxdBuf[gi_ucTxdPtr];                 // 取下一字节
    }
    else
    {
      g_bTxdFinish = TRUE;
    }
  }
}
```

注意程序中的“环形”处理方式！

接着是编写接收帧判断程序，在处理时借用存、取数指针来判断收到了几个字节，所以收到数据的标志可简化为位标志 g_bNewData。

```
634  /*******************************************/
635  /*名称: dataFrame_OK                       */
636  /*用途: 检测接收缓冲区数据,                */
637  /*说明: 如果收到正确的数据帧则返回真       */
638  /*******************************************/
639
640  bit dataFrame_OK(void)
641  {
642    unsigned char i,j,k;
643    bit flag;
644
645    flag = FALSE;
646
647    while(gi_ucGetPtr != gi_ucSavePtr)
648    {
649      if(g_bStartRcv == FALSE)
650      {
681      }
682      else
683      {
706      }
707      gi_ucGetPtr=(gi_ucGetPtr+1)&(MaxRcvByte_C-1);
708    }
709    return (flag);
710  }
```

具体的帧头检测、帧尾处理看程序。

收到正确帧后，要解析命令，并做出相应的操作。通常采用 switch() case() 方式处理（此处只是显示框架，详细的处理请看源代码。注意左侧的行号，使用编辑器的折叠功能是不是感觉不错？）：

```
712  /*****************************************/
713  /*名称: do_Command                       */
714  /*用途: 根据收到的数据帧命令做相应处理      */
715  /*****************************************/
716
717  void do_Command(void)
718  {
719
720    ucCommand = ga_ucRcvBuf[gi_ucStartPtr];       // 取出数据帧中的命令
721
722    switch (ucCommand)
723    {
724      case READ_MEMORY:
725      {
798      }
799
800      case WRITE_MEMORY:
801      {
855      }
856
857      case SET_WORKMODE:
858      {
886      }
887
888      case READ_DATA:
889      {
926      }
927
928      default: break;
929    }
930  }
```

通讯部分软件描述基本结束，读者可对照源代码消化。

如此“啰嗦”的描述不知能否将前述抽象的“思考过程”具体化，便于初学者理解。

4.3.2 测量功能

测量由通讯激活，激活后的每次测量经历以下过程：

1） 发射超声波；

2） 计时，等待回波，定时控制增益；

3） 收到回波后，获取计时数据，计算距离值；

4） 根据模式设置确定是否返回数据；

5） 判断是否自动重新启动新一轮测量。

这里就不再像通讯那样详述了，只说几个关键点：

A） PCA 的使用，由于两个定时器均已占用，所以需要使用 PCA 实现定时功能，此处用的是 PCA 的比较器功能，当计数器和设定值相等时产生中断，从而实现了定时。

B） 发送使用高低电平定时控制是为了灵活处理，原来担心由于三极管导通、截止速度不同，需要用占空比来弥补呢，实际上没有用到。

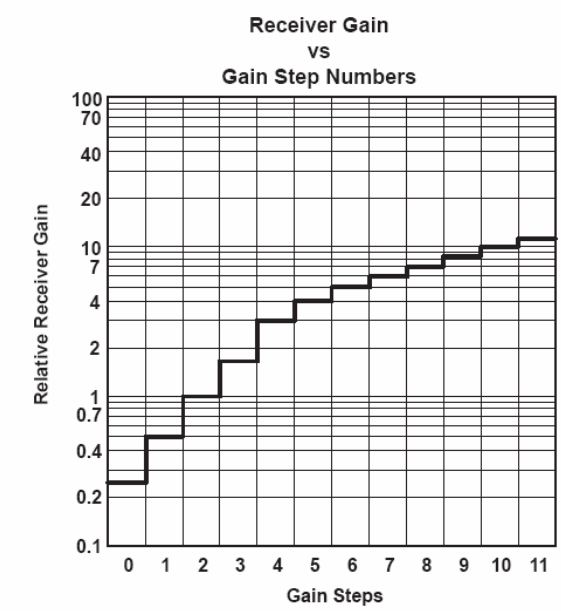
```
unsigned char code ga_uiUS_T_PulsWidth[2]={136,136};   // 超声波脉冲的高低电平时间，根据具体的波形调整！
```

C） 增益控制时间用数组来定义转换的时间是为了灵活，因为看 TL851 资料上的控制时间似乎不是等间隔的，程序中的控制完全是仿照 TL851 做的：

GAIN CONTROL OUTPUT TABLE

STEP NUMBER	GCD	GCC	GCB	GCA	TIME (ms) FROM INITIATE*
0	L	L	L	L	2.38 ms
1	L	L	L	H	5.12 ms
2	L	L	H	L	7.87 ms
3	L	L	H	H	10.61 ms
4	L	H	L	L	13.35 ms
5	L	H	L	H	16.09 ms
6	L	H	H	L	18.84 ms
7	L	H	H	H	21.58 ms
8	H	L	L	L	27.07 ms
9	H	L	L	H	32.55 ms
10	H	L	H	L	38.04 ms
11	H	L	H	H	INIT↓

*This is the time to the end of the indicated step and assumes a nominal 420 kHz ceramic resonator.




```
unsigned int code ga_uiDeltaTime[11] = {26320 - CUT_OFF_DELAY,30358,30358,30358,30358,30358,30358,30358,60715,60715,60715};
```

上面数组是 PCA 的计数比较值的增量，PCA 的计数频率是 11.0592MHz，读者可以计算、比较一下，看是否和上面的那个表相同。

D） 余波抑制功能就属于前面所说的“保护性”处理，是为了改善性能、提高可靠性所为，与测量原理无直接关联。从这部分的处理读者可以感受一下软件和硬件结合的灵活性，如果以前搞过单纯的硬件电路，体会更深！

E） 因为测量是个过程，所以设计为以下几个状态：

```
/* —— 工作状态 ——*/
#define       WAIT_START      0x00
#define       ULTRASONIC_T    0x01
#define       ULTRASONIC_R    0x02
#define       CAL_RESULT      0x03
#define       DATA_BACK       0x04
#define       WAIT_FINISH     0x05
#define       WAIT_NEXT       0x06    // 等待单次测量模式下多次数据采集的间隔时间
```

状态之间的转换多数为依次完成后切换，读者可在源代码中看到；在此只提一下：自动测量和单轮多次测量需要和时间配合，所以借用了时基处理实现：

自动测量时，设置周期计数，减到“0”开始下一次测量

```
        // 10ms 为计时单位的处理
        if(gc_ucMeaPeriodCnt>0)
        {
          gc_ucMeaPeriodCnt--;
        }
```

单轮多次测量时，经测试一次完成后不可以立即启动下一次，需要有个空隙，所以补了空隙计数：

```
      if(gc_ucGapAtSingleMea >0)
      {
        gc_ucGapAtSingleMea--;
        if(gc_ucGapAtSingleMea == 0)
        {
          startUltraSonicSend();                // 启动超声波发射，开始下一次数据采集
        }
      }
```

F) 至于那些数据处理功能，纯属“花哨”，只是为了表示一下：可以在程序中增加一些功能，从而使传感器更具“个性”。这部分应该是读者自己发挥的空间！

传感器中所涉及的程序大概介绍到此。这里所有程序的目的都是为学习而写，所以不是最优化的，更够不上实用级别，仅供参考。

4.4 PC 侧程序

关于 PC 程序的设计，读者有兴趣可阅读“圆梦小车 StepbyStep 之二”，此处就不再赘述了。

所有的设计都是根据前面通讯协议而来的，所以如果读者自己增加了功能，也可以在此基础上添加，我觉得 PC 是一个不错的“助手”。

五、调试

软件调试过程实际上和硬件是交织在一起的，有时硬件的不足需要软件来弥补，如单轮测量中的多次数据采集，调试时发现返回的数据只有第一次是对的，后面的都不对，只好在软件上增加空隙控制。

反之软件上的“无能”又要硬件配合，余波抑制部分就是最好例子。为了改善近距离测量特性，尝试了多种方式都不行，如减少发射脉冲个数。最后只好增加了余波抑制的电路。

5.1 电感问题弥补

读者还记得硬件篇中的“调试花絮”吗？一个硬件设计 Bug，由于不是每个爱好者都能有示波器辅助调试，所以在此做个弥补，用程序来实现这个功能。

在通讯协议的模式设置命令中，增加一个**电感检测模式**，基本原理是在待命状态下启动中断检测，如果有信号且多于 100 次，则表示有振荡，发送消息，如果没有则不发送。

在 INT1 中断处理中增加：

```
if((g_ucWorkMode&GET_MAIN_MODE) == INDUCTOR_TEST)   // 为弥补PCB排版问题增加的电感检测功能 080412
{
  g_b852Inhibit = DIS852OUT;        // 禁止852输出
  gc_ucINT1_Cnt++;
}
```

主循环中增加：

```
case INDUCTOR_TEST:
{
  // 为弥补PCB排版而添加 080412
  if(g_bUltraSonic_In == 0)
  {
    g_b852Inhibit = DIS852OUT;
  }
  g_b852Inhibit = EN852OUT;

  if(gc_ucINT1_Cnt > 100)
  {
    gc_ucINT1_Cnt = 0;

    if(g_bTxdFinish)
    {
      g_bTxdFinish = FALSE;

      ga_ucTxdBuf[2] = g_ucSenderAddr;          // 作为接收方地址发送
      ga_ucTxdBuf[4] = 2;                       // 帧长
      ga_ucTxdBuf[5] = INDUCTOR_TEST;           // 返回测试状态
      ga_ucTxdBuf[6] = 100;                     // 问题信息

      sum = ga_ucTxdBuf[5]+ga_ucTxdBuf[6];

      ga_ucTxdBuf[7] = ~sum;                    // 校验和

      gc_ucTxdCnt = 8;                          // 发送字节计数
      gi_ucTxdPtr = 0;                          // 发送指针
      SBUF = ga_ucTxdBuf[0];                    // 启动发送
    }
  }
  break;
}
```

上述程序的功能是：如在没有信号下振荡，则不断有超声波输入中断，在中断中计数，如达到 100，则发送一帧信息给 PC。

PC 侧程序增加一个工作模式设置，使传感器能进入电感测试状态，而显示则不作修改，借用原来自动测量返回数据的计数显示，如振荡则此计数值会连续增加，反之则不动，停留在数字“1”。读者可借此判断电感的焊接方向，再次抱歉！

六、结语

至此，我们的“智能传感器”算是大功告成了，测试后自我感觉不错。希望读者能做出更有创意的东东。

本文与其说是传感器的“制作说明”，不如说是以传感器为例的“单片机应用介绍”更为贴切。对于传感器的性能文中并未详尽探讨，包括计时到底应该从发送第一个脉冲开始？还是发完所有脉冲再开始等问题都未做落实，留给有兴趣的读者自行研究吧！

还是那句话：抛砖引玉，希望本文对有意入行者有所帮助！

文中所述乃个人观点，所示程序也是未经雕琢之产物，希望读者本着消化之、再造之的心态对待，简单引用已属不妥，更有甚者，不如意处还要“问责”，岂不悲哉！

收笔于 2008 年 4 月 12 日星期日

参考资料:

1、 TI 公司 TL852 数据手册

2、 TI 公司 TI851 数据手册

3、 SensComp 公司 6500 测距模块数据手册

4、 TI 公司 LMV358 数据手册

5、 TI 公司 LM358 数据手册

6、 STC12C5410AD 单片机数据手册（STC12 系列全部包含在内）