# **谈谈单片机裸奔的程序框架**

初学单片机时，都会纠结于其各个模块功能的应用，如串口(232，485)对各种功能IC的控制,电机控制PWM，中断应用，定时器应用，人机界面应用，CAN总线等. 这是一个学习过程中必需的阶段，是基本功。很庆幸，在参加电子设计大赛赛前培训时，MCU周围的控制都训练的很扎实。经过这个阶段后，后来接触不同的MCU就会发现，都大同小异，各有各的优势而已，学任何一种新的MCU都很容易入手包括一些复杂的处理器。而且对MCU的编程控制会提升一个高度概况——就是对各种外围进行控制(如果是对复杂算法的运算就会

用DSP了)，而外围与MCU的通信方式一般也就几种时序：IIC,SPI,intel8080,M6800。这样看来MCU周围的编程就是一个很简单的东西了。

然而这只是嵌入式开发中的一点皮毛而已，在接触过多种MCU,接触过复杂设计要求，跑过操作系统等等后，我们在回到单片机的裸机开发时，就不知不觉的就会考虑到整个程序设计的架构问题;一个好的程序架构，是一个有经验的工程师和一个初学者的分水岭。

以下是我对单片机程序框架以及开发中一些常用部分的认识总结：

任何对时间要求苛刻的需求都是我们的敌人，在必要的时候我们只有增加硬件成本来消灭它;比如你要8个数码管来显示，我们在没有相关的硬件支持的时候必须用MCU以动态扫描的方式来使其工作良好;而动态扫描将或多或少的阻止了MCU处理其他的事情。在MCU负担很重的场合，我会选择选用一个类似max8279外围ic来解决这个困扰;

然而庆幸的是，有着许多不是对时间要求苛刻的事情：

例如键盘的扫描，人们敲击键盘的速率是有限的，我们无需实时扫描着键盘，甚至可以每隔几十ms才去扫描一下;然而这个几十ms的间隔，我们的MCU还可以完成许多的事情;

单片机虽然是裸机奔跑，但是往往现实的需要决定了我们必须跑出操作系统的姿态——多任务程序;

比如一个常用的情况有4个任务：

1 键盘扫描;

2 led数码管显示;

3 串口数据需要接受和处理;

4 串口需要发送数据;

如何来构架这个单片机的程序将是我们的重点;

读书时代的我会把键盘扫描用查询的方式放在主循环中，而串口接收数据用中断，在中断服务函数中组成相应的帧格式后置位相应的标志位，在主函数的循环中进行数据的处理，串口发送数据以及led的显示也放在主循环中;

这样整个程序就以标志变量的通信方式，相互配合的在主循环和后台中断中执行;

然而必须指出其不妥之处：

每个任务的时间片可能过长，这将导致程序的实时性能差。如果以这样的方式在多加几个任务，使得一个循环的时间过长，可能键盘扫描将很不灵敏。所以若要建立一个良好的通用编程模型，我们必须想办法，消去每个任务中费时间的部分以及把每个任务再次分解;下面来细谈每个任务的具体措施：

1 键盘扫描

键盘扫描是单片机的常用函数，以下指出常用的键盘扫描程序中，严重阻碍系统实时性能的地方;

众所周知，一个键按下之后的波形是这样的(假定低有效)：

在有键按下后，数据线上的信号出现一段时间的抖动，然后为低，然后当按键释放时，信号抖动一段时间后变高。当然，在数据线为低或者为高的过程中，都有可能出现一些很窄的干扰信号。

unsigned char kbscan(void)

{

unsigned char sccode,recode;

P2=0xf8;

if ((P2&0xf8)!=0xf8)

{

delay(100); //延时20ms去抖--------这里太费时了，很糟糕

if((P2&0xf8)!=0xf8)

{

sccode=0xfe;

while((sccode&0x08)!=0)

{

P2=sccode;

if ((P2&0xf8)!=0xf8)

break;

sccode=(sccode<<1)|0x01;

}

recode=(P2&0xf8)|0x0f;

return(sccode&recode);

}

}

return (KEY\_NONE);

}

键盘扫描是需要软件去抖的，这没有争议，然而该函数中用软件延时来去抖(ms级别的延时)，这是一个维持系统实时性能的一个大忌讳;

一般还有一个判断按键释放的代码：

While( kbscan() != KEY\_NONE); //死循环等待

这样很糟糕，如果把键盘按下一直不放，这将导致整个系统其它的任务也不能执行，这将是个很严重的bug。

有人会这样进行处理：

While(kbsan() != KEY\_NONE )

{

Delay(10);

If(Num++ > 10)

Break;

}

即在一定得时间内，如果键盘一直按下，将作为有效键处理。这样虽然不导致整个系统其它任务不能运行，但也很大程度上，削弱了系统的实时性能，因为他用了延时函数;

我们用两种有效的方法来解决此问题：

1 在按键功能比较简单的情况下，我们仍然用上面的kbscan()函数进行扫描，只是把其中去抖用的软件延时去了，把去抖以及判断按键的释放用一个函数来处理，它不用软件延时，而是用定时器的计时(用一般的计时也行)来完成;代码如下

void ClearKeyFlag(void)

{

KeyDebounceFlg = 0;

KeyReleaseFlg = 0;

}

void ScanKey(void)

{

++KeyDebounceCnt;//去抖计时(这个计时也可以放在后台定时器计时函数中处理)

KeyCode = kbscan();

if (KeyCode != KEY\_NONE)

{

if (KeyDebounceFlg)//进入去抖状态的标志位

{

if (KeyDebounceCnt > DEBOUNCE\_TIME)//大于了去抖规定的时间

{

if (KeyCode == KeyOldCode)//按键依然存在，则返回键值

{

KeyDebounceFlg = 0;

KeyReleaseFlg = 1;//释放标志

return; //Here exit with keycode

}

ClearKeyFlag(); //KeyCode != KeyOldCode，只是抖动而已

}

}

else

{

if (KeyReleaseFlg == 0)

{

KeyOldCode = KeyCode;

KeyDebounceFlg = 1;

KeyDebounceCnt = 0;

}

else

{

if (KeyCode != KeyOldCode)

ClearKeyFlag();

}

}

}

else

{

ClearKeyFlag();//没有按键则清零标志

}

KeyCode = KEY\_NONE;

}

在按键情况较复杂的情况，如有长按键，组合键，连键等一些复杂功能的按键时候，我们跟倾向于用状态机来实现键盘的扫描;

//avr 单片机 中4\*3扫描状态机实现

char read\_keyboard\_FUN2()

{

static char key\_state = 0, key\_value, key\_line,key\_time;

char key\_return = No\_key,i;

switch (key\_state)

{

case 0: //最初的状态，进行3\*4的键盘扫描

key\_line = 0b00001000;

for (i=1; i<=4; i++) // 扫描键盘

{

PORTD = ~key\_line; // 输出行线电平

PORTD = ~key\_line; // 必须送2次!!!(注1)

key\_value = Key\_mask & PIND; // 读列电平

if (key\_value == Key\_mask)

key\_line <<= 1; // 没有按键，继续扫描

else

{

key\_state++; // 有按键，停止扫描

break; // 转消抖确认状态

}

}

break;

case 1: //此状态来判断按键是不是抖动引起的

if (key\_value == (Key\_mask & PIND)) // 再次读列电平，

{

key\_state++; // 转入等待按键释放状态

key\_time=0;

}

else

key\_state--; // 两次列电平不同返回状态0，(消抖处理)

break;

case 2: // 等待按键释放状态

PORTD = 0b00000111; // 行线全部输出低电平

PORTD = 0b00000111; // 重复送一次

if ( (Key\_mask & PIND) == Key\_mask)

{

key\_state=0; // 列线全部为高电平返回状态0

key\_return= (key\_line | key\_value);//获得了键值

}

else if(++key\_time>=100)//如果长时间没有释放

{

key\_time=0;

key\_state=3;//进入连键状态

key\_return= (key\_line | key\_value);

}

break;

case 3://对于连键，每隔50ms就得到一次键值，windows xp 系统就是这样做的

PORTD = 0b00000111; // 行线全部输出低电平

PORTD = 0b00000111; // 重复送一次

if ( (Key\_mask & PIND) == Key\_mask)

key\_state=0; // 列线全部为高电平返回状态0

else if(++key\_time>=5) //每隔50MS为一次连击的按键

{

key\_time=0;

key\_return= (key\_line | key\_value);

}

break;

}

return key\_return;

}

以上用了4个状态，一般的键盘扫描只用前面3个状态就可以了，后面一个状态是为增加“连键”功能设计的。连键——即如果按下某个键不放，则迅速的多次响应该键值，直到其释放。在主循环中每隔10ms让该键盘扫描函数执行一次即可;我们定其时限为10ms,当然要求并不严格。

2 数码管的显示

一般情况下我们用的八位一体的数码管，采用动态扫描的方法来完成显示;非常庆幸人眼在高于50hz以上的闪烁时发现不了的。所以我们在动态扫描数码管的间隔时间是充裕的。这里我们定其时限为4ms(250HZ) ，用定时器定时为2ms,在定时中断程序中进行扫描的显示，每次只显示其中的一位;当然时限也可以弄长一些，更推荐的方法是把显示函数放入主循环中，而定时中断中置位相应的标志位即可;

// Timer 0 比较匹配中断服务,4ms定时

interrupt [TIM0\_COMP] void timer0\_comp\_isr(void)

{

display(); // 调用LED扫描显示

}

void display(void) // 8位LED数码管动态扫描函数

{

PORTC = 0xff; // 这里把段选都关闭是很必要的，否则数码管会产生拖影

PORTA = led\_7[dis\_buff[posit]];

PORTC = position[posit];

if (++posit >=8 )

posit = 0;

}

3 串口接收数据帧

串口接收时用中断方式的，这无可厚非。但如果你试图在中断服务程序中完成一帧数据的接收就麻烦大了。永远记住，中断服务函数越短越好，否则影响这个程序的实时性能。一个数据帧一般包括若干个字节，我们需要判断一帧是否完成，校验是否正确。在这个过程中我们不能用软件延时，更不能用死循环等待等方式;

所以我们在串口接收中断函数中，只是把数据放置于一个缓冲队列中。

至于组成帧，以及检查帧的工作我们在主循环中解决，并且每次循环中我们只处理一个数据，每个字节数据的处理间隔的弹性比较大，因为我们已经缓存在了队列里面。

/\*==========================================

功能：串口发送接收的时间事件

说明：放在大循环中每10ms一次

输出：none

输入：none

==========================================\*/

void UARTimeEvent(void)

{

if (TxTimer != 0)//发送需要等待的时间递减

--TxTimer;

if (++RxTimer > RX\_FRAME\_RESET) //

RxCnt = 0; //如果接受超时(即不完整的帧或者接收一帧完成)，把接收的不完整帧覆盖

}

/\*==========================================

功能：串口接收中断

说明：接收一个数据，存入缓存

输出：none

输入：none

==========================================\*/

interrupt [USART\_RXC] void uart\_rx\_isr(void)

{

INT8U status,data;

status = UCSRA;

data = UDR;

if ((status & (FRAMING\_ERROR | PARITY\_ERROR | DATA\_OVERRUN))==0)

{

RxBuf[RxBufWrIdx] = data;

if (++RxBufWrIdx == RX\_BUFFER\_SIZE) //接收数据于缓冲中

RxBufWrIdx = 0;

if (++RxBufCnt == RX\_BUFFER\_SIZE)

{

RxBufCnt = 0;

//RxBufferOvf=1;

}

}

}

/\*==========================================

功能：串口接收数据帧

说明：当非0输出时，收到一帧数据

放在大循环中执行

输出：==0:没有数据帧

!=0:数据帧命令字

输入：none

==========================================\*/

INT8U ChkRxFrame(void)

{

INT8U dat;

INT8U cnt;

INT8U sum;

INT8U ret;

ret = RX\_NULL;

if (RxBufCnt != 0)

{

RxTimer = 0; //清接收计数时间，UARTimeEvent()中对于接收超时做了放弃整帧数据的处理

//Display();

cnt = RxCnt;

dat = RxBuf[RxBufRdIdx]; // Get Char

if (++RxBufRdIdx == RX\_BUFFER\_SIZE)

RxBufRdIdx = 0;

Cli();

--RxBufCnt;

Sei();

FrameBuf[cnt++] = dat;

if (cnt >= FRAME\_LEN)// 组成一帧

{

sum = 0;

for (cnt = 0;cnt < (FRAME\_LEN - 1);cnt++)

sum+= FrameBuf[cnt];

if (sum == dat)

ret = FrameBuf[0];

cnt = 0;

}

RxCnt = cnt;

}

return ret;

}

以上的代码ChkRxFrame()可以放于串口接收数据处理函数RxProcess() 中，然后放入主循环中执行即可。以上用一个计时变量RxTimer，很微妙的解决了接收帧超时的放弃帧处理，它没有用任何等待，而且主循环中每次只是接收一个字节数据，时间很短。

我们开始架构整个系统的框架：

我们选用一个系统不常用的TIMER来产生系统所需的系统基准节拍，这里我们选用4ms;

在meg8中我们代码如下：

// Timer 0 overflow interrupt service routine

interrupt [TIM0\_OVF] void timer0\_ovf\_isr(void)

{

// Reinitialize Timer 0 value

TCNT0=0x83;

// Place your code here

if ((++Time1ms & 0x03) == 0)

TimeIntFlg = 1;

}

然后我们设计一个TimeEvent()函数，来调用一些在以指定的频率需要循环调用的函数，

比如每个4ms我们就进行喂狗以及数码管动态扫描显示，每隔1s我们就调用led闪烁程序，每隔20ms我们进行键盘扫描程序;

void TimeEvent (void)

{

if (TimeIntFlg)

{

TimeIntFlg = 0;

ClearWatchDog();

display(); // 在4ms事件中，调用LED扫描显示，以及喂狗

if (++Time4ms > 5)

{

Time4ms = 0;

TimeEvent20ms();//在20ms事件中，我们处理键盘扫描read\_keyboard\_FUN2()

if (++Time100ms > 10)

{

Time100ms = 0;

TimeEvent1Hz();// 在1s事件中，我们使工作指示灯闪烁

}

}

UARTimeEvent();//串口的数据接收事件，在4ms事件中处理

}

}

显然整个思路已经很清晰了，cpu需要处理的循环事件都可以根据其对于时间的要求很方便的加入该函数中。但是我们对这事件有要求：

执行速度快，简短，不能有太长的延时等待，其所有事件一次执行时间和必须小于系统的基准时间片4ms(根据需要可以加大系统基准节拍)。所以我们的键盘扫描程序，数码管显示程序，串口接收程序都如我先前所示。如果逼不得已需要用到较长的延时(如模拟IIc时序中用到的延时)

我们设计了这样的延时函数：

void RunTime250Hz (INT8U delay)//此延时函数的单位为4ms(系统基准节拍)

{

while (delay)

{

if (TimeIntFlg)

{

--delay;

TimeEvent();

}

TxProcess();

RxProcess();

}

}

我们需要延时的时间=delay\*系统记住节拍4ms,此函数就确保了在延时的同时，我们其它事件(键盘扫描，led显示等)也并没有被耽误;

好了这样我们的主函数main()将很简短：

Void main (voie)

{

Init\_all();

while (1)

{

TimeEvent(); //对于循环事件的处理

RxProcess(); //串口对接收的数据处理

TxProcess();// 串口发送数据处理

}

}

整体看来我们的系统就成了将近一个万能的模版了，根据自己所选的cpu,选个定时器，在添加自己的事件函数即可，非常灵活方便实用，一般的单片机能胜任的场合，该模版都能搞定。

整个系统以全局标志作为主线，形散神不散;系统耗费比较小，只是牺牲了一个Timer而已，在资源缺乏的单片机中，非常适;曾经看过一个网友的模版“单片机实用系统”，其以51为例子写的，整体思路和这个差不多，不过他写得更为规范紧凑，非常欣赏;但个人觉得代码开销量要大些，用惯了都一样哦。但是由于本系统以全局标志为驱动事件，所以比较感觉比较凌乱，全局最好都做好注释，而其要注意一些隐形的函数递归情况，千万不要递归的太深哦(有的单片机不支持)。