


第四章 任务设计

在基于实时操作系统的应用程序设计中，任务设计是整个应用程序的基础，其他软件设计工作都是围绕任务设计来展开，任务设计就是设计“任务函数”和相关的数据结构。

4.1 任务函数的结构

在用户任务函数中，必须包含至少一次对操作系统服务函数的调用，否则比其优先级低的任务将无法得到运行机会，这是用户任务函数与普通函数的明显区别。任务函数的结构按任务的执行方式可以分为三类：单次执行类、周期执行类和事件触发类，下面分别介绍其结构特点。


4.1.1 单次执行的任务

此类任务在创建后只执行一次，执行结束后即自行删除，其任务函数的结构如下：

程序清单 L4-1 单次执行任务函数的结构

```
void MyTask (void *pdata)           //单次执行的任务函数
{
    进行准备工作的代码;
    任务实体代码;
    调用任务删除函数;              //调用 OSTaskDel(OS_PRIO_SELF)
}
```

单次执行的任务函数由三部分组成：第一部分是“进行准备工作的代码”，完成各项准备工作，如定义和初始化变量、初始化某些设备等等，这部分代码的多少根据实际需要来决定，也可能完全空缺。第二部分是“任务实体代码”，这部分代码完成该任务的具体功能，其中通常包含对若干系统函数的调用，除若干临界段代码（中断被关闭）外，任务的其它代码均可以被中断，以保证高优先级的就绪任务能够及时运行。第三部分是“调用任务删除函数”，该任务将自己删除，操作系统将不再管理它。

单次执行的任务采用“创建任务函数”来启动，当该任务被另外一个任务（或主函数）创建时，就进入就绪状态，等到比它优先级高的任务都被挂起来时便获得运行权，进入运行状态，任务完成后自行删除，启动任务”就是一个例子。

采用“启动任务”后，~~主函数~~就可以简化为三行，只负责与操作系统有关的事情，即初始化操作系统、创建“启动任务”、启动操作系统，使主函数的内容固定下来，与具体的应用系统无关。真正启动系统所需要的准备工作由“启动任务”来完成，它的内容与具体的系

统密切相关。主函数和“启动任务”的示意代码如下：

程序清单 L4-2 使用启动任务

```
void main (void) //主函数
{
    OSInit ();    //初始化操作系统
    OSTaskCreate(TaskStart,(void *)0,&TaskStartStk[TASK_STK_SIZE-1],1);//创建启动任务
    OSStart ();   //启动操作系统，开始对任务进行调度管理
}

void TaskStart(void *pdata)    //启动任务
{
    pdata = pdata;
    系统硬件初始化； //时钟系统、中断系统、外设等等
    创建各个任务； //如键盘任务、显示任务、采样任务、数据处理任务、打印任务等等
    创建各种通信工具； //如信号量、消息邮箱、消息队列等等
    OSTaskDel (OS_PRIO_SELF); //删除自己
}
```

在“启动任务”中完成与系统硬件有关的各种初始化工作，然后创建各个实质任务和所需要的各种通信工具，至此系统才真正完成准备工作，“启动任务”的使命也就结束了，最后将自己删除。为了保证“启动任务”能够连续运行，必须将“启动任务”的优先级选择为最高。否则，当“启动任务”创建一个优先级高于自己的任务时，刚刚创建的任务就会立即进入运行状态，而与此任务关联的其它任务可能还没有创建，它使用的通信工具也还没有创建，系统必然出错。“启动任务”调用的“自我删除”函数会进行任务调度操作，从而使系统开始运行各个实质任务。

由于“启动任务”不是用户系统的实质任务，又占用高优先级资源和任务资源，故不常用。更常用的方法是将“启动任务”所完成的操作交给一个用户系统的实质任务来完成。这时，主函数和有启动功能的任务函数的示意代码如下：



程序清单 L4-3 以用户任务代替启动任务

```
void main (void) //主函数
{
    OSInit ();    //初始化操作系统
    OSTaskCreate(TaskUser1,(void *)0,&TaskUser1Stk[TASK_STK_SIZE-1],1);//创建任务 1
    OSStart ();   //启动操作系统，开始对任务进行调度管理
}
```

```

void TaskUser1(void *pdata)           //用户任务 1
{
    pdata = pdata;
    系统硬件初始化; //时钟系统、中断系统、外设等等
    创建各个任务; //如键盘任务、显示任务、采样任务、数据处理任务、打印任务等等
    创建各种通信工具; //如信号量、消息邮箱、消息队列等等
    用户任务 1 本身的代码;
}

```

使用“单次执行”的任务函数结构的场合反而是可以多次执行的任务，每当需要执行该任务时就将该任务创建一次。由键盘操作来启动的任务常采用这种结构，如用一个“发送”按钮启动串口通信任务，其程序如下：

程序清单 L4-4 用创建任务的方式启动任务

```

void TaskKey (void *pdata)           //键盘任务函数（示意）
{
    INT8U key;
    for (;;)                          //无限循环，也可用 while (1)
    {
        key=keyin();                  //获取按键操作信息
        switch (key)
        {
            case KEY_SUART:           //“发送”按钮，创建串口发送任务
                OSTaskCreate(TaskUart,(void *)0,&TaskUartStk[TASK_STK_SIZE-1],3);
                break;
            case KEY_$$$:              //其它按钮的处理代码
                .
                .
                .
        }
        OSTimeDly(2);                 //延时
    }
}

void TaskUart(void *pdata)           //串口发送任务（示意）
{
    pdata = pdata;
    串口初始化;
}

```

```

组织发送帧;
数据指针初始化;
发送数据;
OSTaskDel (OS_PRIO_SELF); //删除自己
}

```

采用“任务创建”的方式来启动任务，可以省略用通信手段触发任务的麻烦，还可以通过*pdata 来传递原始参数，使得每次启动任务时可以有不同的工作状态。如下面的程序在创建串行口发送任务时同时指定波特率：

程序清单 L4-5 在创建任务时传送参数

```

void TaskKey (void *pdata) //键盘任务函数（示意）
{
    INT8U key;
    INT16U baud; //波特率，由用户通过键盘选定
    for (;;) //无限循环，也可用 while (1)
    {
        key=keyin(); //获取按键操作信息
        switch (key)
        {
            case KEY_SUART: //“发送”按钮，创建串行口发送任务，带参数
                OSTaskCreate(TaskUart,&baud,&TaskUartStk[TASK_STK_SIZE-1],3);
                break;
            case KEY_$$$: //其它按钮的处理代码
                .
                .
                .
        }
        OSTimeDly(2); //延时
    }
}

void TaskUart(void *pdata) //串行口发送任务（示意）
{
    baud = *pdata; //获取波特率
    串行口初始化; //用获取波特率的波特率初始化串行口
    组织发送帧;
    数据指针初始化;
    发送数据;
}

```

```

OSTaskDel(OS_PRIO_SELF); //删除自己
}

```

虽然用“创建任务”的方式来启动一个任务有以上方便之处，但每次启动任务都要调用“任务创建函数”，需要对“任务控制块”进行全面初始化，并对“任务控制块链表”和“任务就绪表”进行操作，比较耗时，故只适用于实时性要求不高的任务（如键盘操作启动的任务）。采用“创建任务”的方式来启动一个任务除了实时性差外，还可能在任务自我删除后出现后遗症：

- 占用的共享资源尚未释放，使其它需要使用该资源的任务不能运行。
- 通信关系的“上家”任务（或 ISR）发出的信号量或消息将被积压而得不到响应。
- 通信关系的“下家”任务因为得不到信号量或消息而被遗弃（被永远挂起）。
- 可能留下未删除干净的废弃变量。

因此：



- 如果该任务使用了共享资源，必须在自我删除之前释放（如释放内存块，发送互斥信号量）。
- 如果该任务有关联任务（或 ISR），必须在自我删除之前将这种关联关系解除，而解除关联关系需要删除关联任务和通信工具，这是得不偿失而又非常麻烦的事情。

适合采用“创建任务”的方式来启动的任务，通常是“孤立任务”，它们不和其它任务进行通信（ISR 除外），只使用共享资源来获取信息和输出信息。如果不满足这个条件，应该采用下面两种任务函数机构，并在系统启动时创建好。

4.1.2 周期性执行的任务

此类任务在创建后按一个固定的周期来执行，其任务函数的结构如下：

程序清单 L4-6 周期性任务函数的结构

```

void MyTask (void *pdata) //周期性执行的任务函数
{
    进行准备工作的代码;
    for (;;) //无限循环，也可用 while (1)
    {
        任务实体代码;
        调用系统延时函数; //调用 OSTimeDly()或 OSTimeDlyHMSM()
    }
}

```

周期性执行的任务函数也由三部分组成：第一部分“进行准备工作的代码”和第二部分“任务实体代码”的含义与单次执行任务的含义相同，第三部分是“调用系统延时函数”，把 CPU 的控制权主动交给操作系统，使自己挂起，再由操作系统来启动其它已经就绪的任务。

当延时时间到后，重新进入就绪状态，通常能够很快获得运行权。

通过合理设置调用 `OSTimeDly()` 或 `OSTimeDlyHMSM()` 时的参数值可以调整任务的执行周期。当任务执行周期远大于系统时钟节拍时，任务执行周期的相对误差比较小；当任务执行周期只有几个时钟节拍时，相邻两次执行的间隔时间抖动不能忽视，任务的执行周期的相对误差比较大，只适用于对周期稳定性要求不高的任务（如键盘任务）；当任务执行周期只有一个时钟节拍时，可将该任务的功能放到 `OSTimeTickHook()`（时钟节拍函数中的钩子函数）中去执行；当任务执行周期小于一个时钟节拍或者不是时钟节拍的整数倍时，将无法使用延时函数对其进行周期控制，只能采用独立于操作系统的定时中断来触发。采用独立定时器触发的任务具有很高的周期稳定性。

周期性执行的任务函数编程比较单纯，只要创建一次，就能周期运行。在实际应用中，很多任务都具有周期性，它们的任务函数都使用这种结构，如键盘扫描任务、显示刷新任务、模拟信号采样任务等等，键盘任务的示意代码参阅上一小节。

4.1.3 事件触发执行的任务

此类任务在创建后，虽然很快可以获得运行权，但任务实体代码的执行需要等待某种事件的发生，在相关事件发生之前，则被操作系统挂起。相关事件发生一次，该任务实体代码就执行一次，故该类型任务称为事件触发执行的任务，其任务函数的结构如下：

程序清单 L4-7 事件触发的任务函数的结构

```
void MyTask (void *pdata)           //事件触发执行的任务函数
{
    进行准备工作的代码;
    for (;;)                          //无限循环，也可用 while (1)
    {
        调用获取事件的函数;         //如：等待信号量、等待邮箱中的消息等等。
        任务实体代码;
    }
}
```

事件触发执行的任务函数也由三部分组成：第一部分“进行准备工作的代码”和第三部分“任务实体代码”的含义与前面两种任务的含义相同，第二部分是“调用获取事件的函数”，使用了操作系统提供的某种通信机制，等待另外一个任务（或 `ISR`）发出的信息（如信号量或邮箱中的消息），在取得这个信息之前处于等待状态（挂起状态），当另外一个任务（或 `ISR`）发出相关信息时（调用了操作系统提供的通信函数），操作系统就使该任务进入就绪状态，通过任务调度，任务的实体代码获得运行权，完成该任务的实际功能。

如用一个“发送”按钮启动串行口通信任务，将数据发送到上位机。在键盘任务中，按下“发送”按钮后就发出信号量。在串行口任务中，只要得到信号量就将数据发给上位机，

示意代码如下：

程序清单 L4-8 用信号量触发任务

```
OS_EVENT *Sem;    //信号量指针
void TaskKey (void *pdata)    //键盘任务函数（示意）
{
    INT8U key;
    for (;;)    //无限循环，也可用 while (1)
    {
        key=keyin();    //读入按键操作信息
        switch (key)
        {
            case KEY_SUART:    //“发送”按钮
                OSSemPost(Sem);    //向串行口发送任务发出信号量
                break;
            case KEY_$$$:    //其它按钮的处理代码
                .
                .
                .
        }
        OSTimeDly(2);    //延时
    }
}

void TaskUart(void *pdata)    //串行口发送任务（示意）
{
    pdata = pdata;
    INT8U err;
    for (;;)    //无限循环
    {
        OSSemPend(Sem, 0, &err); //等待键盘任务发出的信号量
        串行口初始化;
        组织发送帧;
        数据指针初始化;
        发送数据;
    }
}
```

如果在触发任务时还需要传送参数，可以采用发送信息的方法，程序如下：

程序清单 L4-9 用消息触发任务

```
OS_EVENT *Mybox;           //消息邮箱
void TaskKey (void *pdata)  //键盘任务函数（示意）
{
    INT8U key;
    INT16U baud;           //波特率，由用户通过键盘选定
    for (;;)              //无限循环，也可用 while (1)
    {
        key=keyin();      //读入按键操作信息
        switch (key)
        {
            case KEY_SUART: //“发送”按钮
                OSMboxPost(Mybox,&baud) //发送消息（波特率）
                break;
            case KEY_$$$:  //其它按钮的处理代码
                .
                .
                .
        }
        OSTimeDly(2);     //延时
    }
}

void TaskUart(void *pdata)  //串口发送任务（示意）
{
    INT16U baud;          //波特率
    INT8U err;
    for (;;)              //无限循环
    {
        pdata=OSMboxPend(Mybox, 0, &err); //等待键盘任务发出的消息
        baud=(INT16U)*pdata; //获取波特率
        串口初始化; //用获取的波特率初始化串口
        组织发送帧;
        数据指针初始化;
        发送数据;
    }
}
```


☰ 当触发条件为“时间间隔”时（定时器中断触发），该任务就具有周期性，这种任务函数结构适用于执行周期小于一个时钟节拍或者不是时钟节拍的整数倍的周期性任务，即周期性任务也能用事件触发执行的任务函数来实现。定时中断负责按预定的时间间隔准确地发出信号量，被关联的任务总是处于等待信号量的状态下，每得一次信号量就执行一次，这种方式具有比较准确的周期。

有些触发任务的事件属于信号类（信号量），其作用仅仅是启动任务的运行，如第三章中介绍的“火灾监控系统”（图 3-5）中的“传感器检测任务”发出的事件就属于“信号类”，而“自动报警任务”、“喷淋灭火任务”、“保存警记录任务”和“打印记录任务”都是将该信号作为启动信号，这四个任务都是事件触发执行的任务。

还有一些触发任务的事件属于信息类（邮箱中的消息），其作用不仅仅是启动任务，而且为该任务提供原始数据和资料，如第三章中介绍的“能谱仪”（图 3-7）中的 ISR 发出的事件（A/D 转换数据）就属于“信息类”，它不但启动了“调整能谱数据任务”，而且是“调整能谱数据任务”所需要的原始数据。

在实际应用系统中，同样存在各种“事件触发执行”的任务，那些非周期性的任务均可以归入这一类，如“打印任务”、“通讯任务”和“报警任务”等等。

4.2 优先级安排

为不同任务安排不同的优先级，其最终目标是使系统的实时性指标能够得到满足。本节分析任务的优先级资源和任务的优先级安排原则。

4.2.1 任务的优先级资源

任务的优先级资源由操作系统提供，以 $\mu\text{C}/\text{OS-II}$ 为例，共有 64 个优先级，优先级的高低按编号从 0（最高）到 63（最低）排序。由于用户实际使用到的优先级总个数通常远小于 64，为节约系统资源，可以通过定义系统常量 `OS_LOWEST_PRIO` 的值来限制优先级编号的范围，当最低优先级为定为 18（共 19 个不同的优先级）时，定义如下：

```
#define OS_LOWEST_PRIO 18
```

$\mu\text{C}/\text{OS-II}$ 实时操作系统总是将最低优先级 `OS_LOWEST_PRIO` 分配给“空闲任务”，将次低优先级 `OS_LOWEST_PRIO-1` 分配给“统计任务”。在此例中，最低优先级为定为 18，则“空闲任务”的优先级为 18，“统计任务”的优先级为 17，用户实际可使用的优先级资源为 0 到 16，共 17 个。

$\mu\text{C}/\text{OS-II}$ 实时操作系统还保留对最高的四个优先级（0、1、2、3）和 `OS_LOWEST_PRIO-3` 与 `OS_LOWEST_PRIO-2` 的使用权，以备将来操作系统升级时使用。如果用户的应用程序希望在将来升级后的操作系统下仍然可以不加修改地使用，则用户任务可以放心使用的优先级个数为 `OS_LOWEST_PRIO-7`。在本例中，软件优先级资源为 $18-7=11$ 个，即可使用的优先级为 4、5、6、7、8、9、10、11、12、13、14。

实际可使用的软件优先级资源数目应该留有余地，以便将来扩充应用软件的功能（增加新任务）时不必对优先级进行大范围的调整。

4.2.2 优先级安排原则

任务的优先级安排原则如下：

- 中断关联性：与中断服务程序（ISR）有关联的任务应该安排尽可能高的优先级，以便及时处理异步事件，提高系统的实时性。如果优先级安排得比较低，CPU 有可能被优先级高一些的任务长期占用，以致于在第二次中断发生时连第一次中断还没有处理，产生信号丢失现象。
- 紧迫性：因为紧迫任务对响应时间有严格要求，在所有紧迫任务中，按响应时间要求排序，越紧迫的任务安排的优先级越高。紧迫任务通常与 ISR 关联。
- 关键性：任务越关键安排的优先级越高，以保障其执行机会。
- 频繁性：对于周期性任务，执行越频繁，则周期越短，允许耽误的时间也越短，故应该安排的优先级也越高，以保障及时得到执行。
- 快捷性：在前面各项条件相近时，越快捷（耗时短）的任务安排的优先级越高，以使其他就绪任务的延时缩短。

例如一个应用系统中安排有键盘任务、显示任务、模拟信号采集任务、数据处理任务、串行口接收任务、串行口发送任务。在这些任务中，模拟信号采集任务、串行口接收任务和串行口发送任务均与 ISR 关联，实时性要求比较高。其中，串行口接收任务是关键任务和紧迫任务，遗漏接收内容是不允许的；模拟信号采集任务是紧迫任务，但不是关键任务，遗漏一个数据还不至于发生重大问题；在串行口发送任务中，CPU 是主动方，慢一些也可以，只要将数据发出去就可以。键盘任务和显示任务是人机接口任务，实时性要求很低。数据处理任务根据其运算量来决定，运算量很大时，优先级安排最低，运算量不大时，优先级可安排得比键盘任务高一些。



根据以上分析，最低优先级 OS_LOWEST_PRIO 定为 18，各个任务的优先级安排如下：串行口接收任务（优先级 2），模拟信号采集任务（优先级 4），串行口发送任务（优先级 6），数据处理任务（优先级 9），显示任务（优先级 12），键盘任务（优先级 13）。当优先级的安排比较宽松时，以后增加新任务就比较方便，在不改变现有任务优先级的情况下，很容易根据需要找到一个合适的空闲优先级。

4.3 任务的数据结构设计

对于一个任务，除了它的代码（任务函数）外，还有相关的信息。为保存这些信息，必须为任务设计对应的若干数据结构。任务需要配备的数据结构分为两类，一类是与操作系统有关的数据结构，另外一类是与操作系统无关的数据结构。

4.3.1 与操作系统有关的数据结构

一个任务要想在操作系统的管理下工作，必须首先被创建。在 $\mu\text{C}/\text{OS-II}$ 中，任务的创建函数原形如下：

```
INT8U OSTaskCreate (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio);
```

从任务的创建函数的形参表可以看出，除了任务函数代码外，还必须准备三样东西：任务参数指针、任务堆栈指针和任务优先级，这三样东西实际上与任务的三个数据结构有关：任务参数表、任务堆栈和任务控制块。

- 任务参数表：由用户定义的参数表，用来向任务传输原始参数（即任务函数代码中的参数 `void *pdata`）。通常设为空表，即 `(void *)0`。
- 任务堆栈：其容量由用户设置，必须保证足够大。
- 任务控制块：由操作系统设置。

操作系统还控制其它数据结构，这些数据结构与一个以上的任务有关，如信号量、消息邮箱、消息队列、内存块、事件控制块等等。

操作系统控制的数据结构均为全局数据结构，用户可以对这些与操作系统有关的数据结构进行剪裁，详情参阅第十一章。

4.3.2 与操作系统无关的数据结构

每个任务都有其特定的功能，需要处理某些特定的信息，为此需要定义对应的数据结构来保存这些信息，常用的数据结构有变量、数组、结构体、字符串等。

每个信息都有其生产者（对数据结构进行写操作）和消费者（对数据结构进行读操作），一个信息至少有一个生产者和一个消费者，且都可以不止一个。

当某个信息的生产者和消费者都是同一个任务（与其它任务无关）时，保存这个信息的数据结构应该在该任务函数内部定义，成为它的私有信息，如局部变量。

当某个信息的生产者和消费者不是同一个任务（包括 `ISR`）时，保存这个信息的数据结构应该在任务函数的外部定义，使它成为共享资源，如全局变量。对这部分数据结构的访问需要特别小心，必须保证访问的互斥性，详情参阅第七章的资源同步内容。

4.4 任务设计中的问题

每个任务都有其规定的功能，这些功能必须在任务函数的设计中得到实现。任务的功能设计过程即任务函数的编写过程，与传统的（没有操作系统的）功能模块设计类似，同样需要注意运行效率、可靠性和容错性等常规问题。

- 运行效率：针对具体场合，采用最合适的处理方法（算法），提高处理效率。

- 可靠性：采用合适的算法与措施，提高系统的抗干扰能力。
- 容错性：采用合适的算法与措施，提高系统的容错能力。

以上问题包含非常丰富的内容，并非一言两语可以说清楚，有关内容可参阅相关专题书籍。本节只讨论在操作系统管理下任务函数编写中出现的新问题。

4.4.1 公共函数的调用

当若干个任务均需要使用某些基本处理功能时，为简化设计，通常将这种基本处理功能单独编写为一个公共函数，供不同任务调用。因为大多数任务都有数据处理过程，所以各种数据处理的基本函数常常被编写为公共函数。

如果一个任务正在调用（运行）某个公共函数时被另一个高优先级的任务抢占，当这个高优先级任务也调用同一个公共函数时，非常可能会破坏原来任务的数据。为了防止这种情况发生，常采用两种措施：互斥调用和可重入设计。

- 互斥调用：将公共函数作为一种共享资源看待，以互斥方式调用公共函数。如果公共函数比较简单，运行时间很短，可以采用先关中断（或关调度）再调用公共函数，调用结束后再开中断（或开调度），从而避免其它任务打扰。如果公共函数比较复杂，运行时间较长，以上方法将严重影响系统的实时性，这时最好为这个公共函数配备一个互斥信号量，任何任务在调用这个公共函数前必须首先取得对应的互斥信号量，否则就会被挂起。
- 可重入设计：“可重入函数”允许多个任务嵌套调用，各个任务的数据相互独立，互不干扰。对于比较简单的公共函数，尽可能设计成可重入函数，免除采用互斥调用方法的麻烦。将公共函数设计成为“可重入函数”的关键是不使用全局资源（如全局变量），可重入函数中所有的变量（包括指针）均为局部变量（其中也包括形式参数）。由于函数的局部变量是在调用是临时分配到储存空间，不同的任务由于在不同的时刻调用该函数，它们的同一个局部变量分配的存储空间并不相同，互不干扰。另外，如果“可重入函数”调用了其它函数，则这些被调用的函数也必须是“可重入函数”。

以一个简单的排序函数为例，该函数对两个变量 a 和 b 的大小进行检查，如果 a 小于 b，则交换它们的数值，以保证 a 不小于 b。该函数的处理过程中需要一个临时变量，如果临时变量为全局变量，则该函数不可重入（程序清单 L4-10）：当一个任务调用该函数，将变量 a 的数值保存到临时变量中时，正好被另一个高优先级任务抢占，如果高优先级任务也调用该函数，就会将临时变量的数值改变。当高优先级任务挂起后，原来的任务继续执行，由于临时变量的数值已经改变，最后的结果自然出错。如果在函数内部定义临时变量，使其成为局部变量，则该函数就成为可重入函数（程序清单 L4-11）：每个调用该函数的任务均具有一套完整的私有变量，相互完全独立。

程序清单 L4-10 不可重入函数

```
INT16U temp;
```

```
//全局变量
```

```

void Fun (INT16U *a, INT16U *b)      //不可重入函数
{
    if (*a < *b) { //如果 a 小于 b, 则交换它们的数值。
        temp=*a;
        *a=*b;
        *b=temp;
    }
}

```

程序清单 L4-11 可重入函数

```

void Fun (INT16U *a, INT16U *b)      //可重入函数
{
    INT16U temp; //局部变量
    if (*a < *b) { //如果 a 小于 b, 则交换它们的数值。
        temp=*a;
        *a=*b;
        *b=temp;
    }
}

```

4.4.2 与其它任务的协调

一个任务的功能往往需要其它任务配合才能完成。~~在没有操作系统的传统的编程模式下, 只要直接调用这些模块就可以了。在操作系统的管理下, 不允许任务之间相互调用, 必须采用操作系统提供的同步和通信机制来进行任务之间的协调运行, 这个问题在第六章的“行为同步”内容中将有详细的讨论。~~

4.4.3 共享资源的访问

任务在运行过程中, 需要访问共享资源。如果不采取措施, 共享资源的完整性和安全性将很难保障, 这个问题在第七章的“资源同步”内容中将有详细的讨论。

4.5 任务的代码设计过程

任务函数的代码中包含若干处对操作系统服务函数的调用, 通过对系统服务函数的调用完成各种系统管理功能, 如任务管理、通信管理、时间管理等等。凡是操作系统已经提供了的服务功能, 必须调用相应的服务函数, 用户使用自己编写的代码来实现相同功能是非常冒

险的。

由于系统的实际运行效果是各个任务配合运行的结果，这种配合过程又是通过操作系统的管理来实现的，即通过调用操作系统服务函数来实现的。“何时调用系统服务”和“调用什么系统服务”是任务设计中的关键问题，这个问题与任务之间的相互关联程度有关，需要通过分析这种关联关系才能确定。

在一般的程序模块设计中，模块之间的接口只有数据接口，编写代码相对容易。由于任务的独立性和并发性，任务代码的编写与程序模块差异较大，任务之间不仅有数据流动，还有行为互动，而且相互之间的作用是通过操作系统的服务来实现的。

一个任务的代码设计过程是从上到下的过程，先分析系统总体任务关联图，明确每个任务在系统整体中的位置和角色，再一个一个任务进行详细关联分析，然后画出任务的程序流程图，最后按流程图编写程序代码。

4.5.1 系统总体任务关联图

在第三章中，我们讨论了任务划分的原则，当我们完成了任务划分的工作后，就确定了系统总的任务数目和关联的 ISR 数目。为了进行任务设计，必须把这些任务（包括 ISR）之间的相互关系搞清楚。为此，可以使用系统总体任务关联图来表示各个任务（包括 ISR）之间的相互关系。

我们来讨论一个简单的系统，这个系统完成能谱数据的采集、显示和向上位机发送三项功能。系统应用软件包含两个 ISR（峰值数据采集 ISR 和串行口发送 ISR）和四个任务：键盘任务、能谱数据采集和调整任务、能谱显示任务和能谱数据发送任务。系统有三个按键：“采集”、“显示”和“发送”，分别启动三个任务。键盘任务由主函数创建，其它三个任务由键盘任务创建。峰值数据采集 ISR 和能谱数据采集和调整任务之间用消息队列进行通信。串行口发送 ISR 和能谱数据发送任务之间用信号量进行通信。能谱数据为一个全局数组，由能谱数据采集和调整任务生成，供能谱显示任务和能谱数据发送任务任务使用，是三个任务的共享资源，配备了一个互斥信号量。能谱数据采集任务按“定数方式”工作，完成预定采样次数后即结束。系统总体任务关联图如图 4-1 所示。

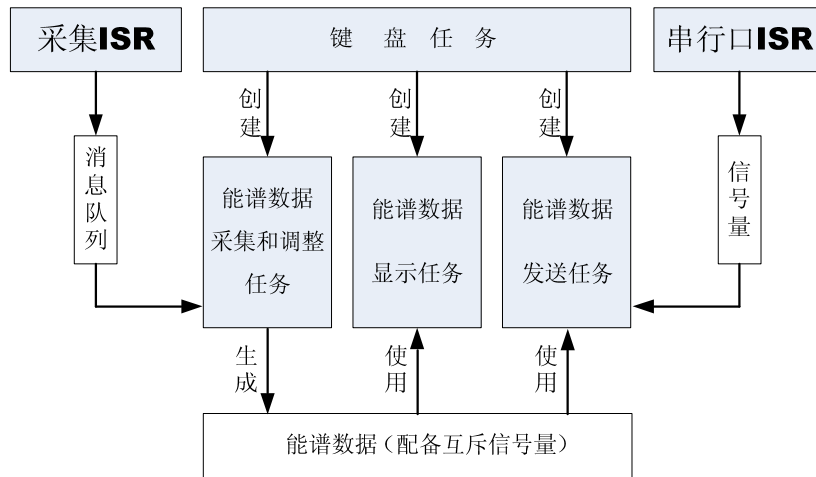


图 4-1 总体任务关联图

第十章介绍了一个复杂一些的设计实例，其中图 10-3 就是该实例的系统总体任务关联图。

4.5.2 任务的关联分析

任务函数要一个一个来编写，在编写某个具体的任务函数（或 ISR）时，必须非常清楚地掌握它与其它任务（或 ISR）的关系。这里包含两种类型的关系，第一种关系是行为同步关系，体现为时序上的触发关系，第二种关系是资源同步关系，体现为信息的流动和共享关系。关于这两种的关系的具体处理方法，将在第六章进行详细讨论。

第一种关系涉及到以下方面：

- 本任务的运行受到哪些任务（或 ISR）的制约？即本任务在运行过程中需要等待哪些任务（或 ISR）发出的信号量或消息。
- 本任务（或 ISR）可以控制哪些任务的运行？即本任务（或 ISR）在运行过程中会向哪些任务发出的信号量或消息，以达到触发这些任务运行的目的。

第二种关系涉及到以下方面：

- 本任务在运行过程中需要得到哪些任务（或 ISR）提供的数据？提供数据的形式是什么？通常提供数据的形式有全局变量、全局数组或数据块、消息和消息队列。
- 本任务（或 ISR）在运行过程中会向哪些任务提供数据？提供数据的形式是什么？

以图 4-1 中的系统为例，各个任务的关联分析如下：

- 键盘任务：键盘任务由主函数创建后，周期性运行，不受其它任务制约，也不需要其它任务提供数据。而键盘任务通过创建任务的形式控制其它三个任务的运行。
- 能谱数据采集和调整任务：本任务由“键盘任务”创建，控制数据采集 ISR 的启动和停止，接收 ISR 通过消息队列提供的原始数据，生成能谱数据（配备了互斥信号量），以全局数组的形式供显示任务和数据发送任务使用。

- 能谱显示任务：本任务由“键盘任务”创建，使用能谱数据（配备了互斥信号量），完成能谱图形显示。
- 数据发送任务：本任务由“键盘任务”创建，使用能谱数据（配备了互斥信号量），控制数据采集串口发送 ISR 的启动和停止，完成能谱数据发送功能。

4.5.3 任务的程序流程图

在掌握了本任务与其它任务（或 ISR）的各种关系后，合理安排本任务的工作流程，使得本任务和其它任务（或 ISR）协调工作，完成预定的功能。

任务的程序流程设计就是画出该任务的程序流程图。任务的流程图与普通程序模块的流程图不同，普通程序模块的流程图是一直运行的，而任务的程序流程图中包含至少一处系统服务函数调用，有可能被挂起，故任务的程序流程图实际上是断续运行的。

如果某任务与其它任务的关联比较简单，任务本身完成的功能也很简单，可以不画程序流程图，直接开始编写程序代码。如果某任务与其它任务的关联比较复杂，任务本身完成的功能也较复杂，最好先画出任务的程序流程图，用来指导任务程序代码的编写，可以减少差错。以图 4-1 中的系统为例，各个任务的程序流程图如图 4-2、图 4-3 和图 4-4 所示。

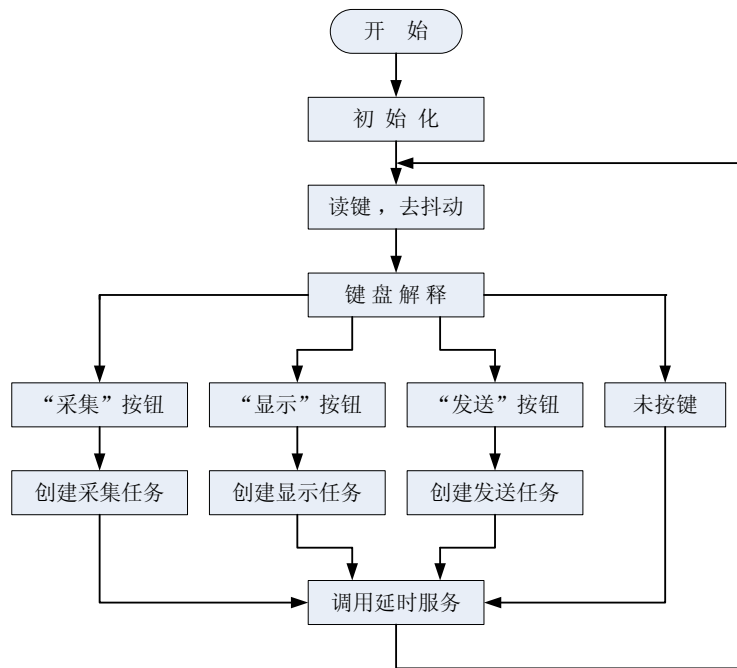


图 4-2 键盘任务程序流程图

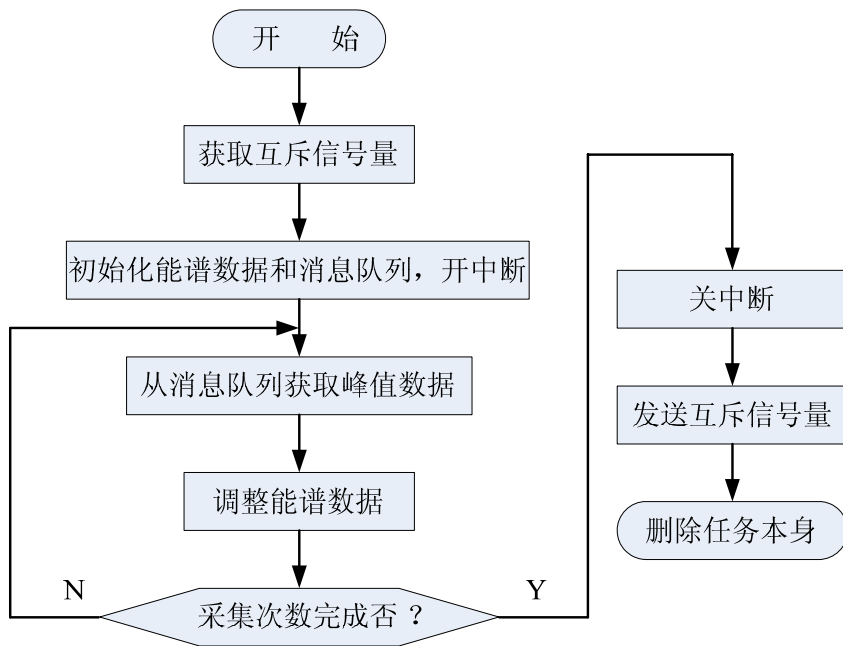


图 4-3 数据采集和调整任务程序流程图

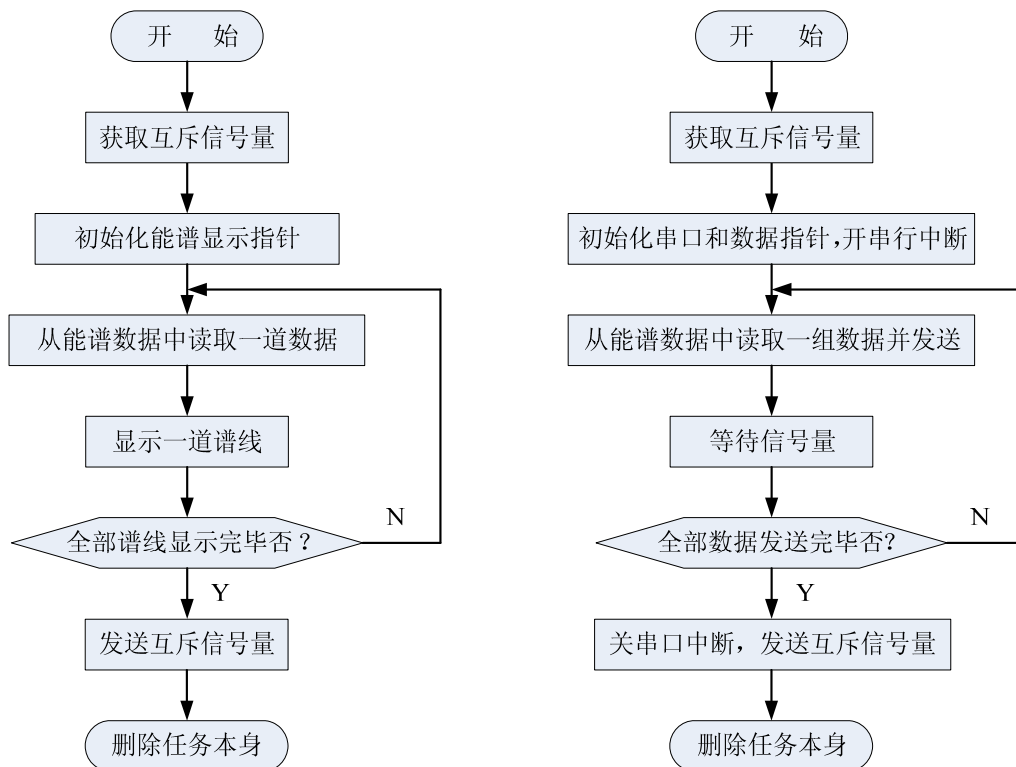


图 4-4 显示任务和数据发送任务程序流程图

4.5.4 编写任务的程序代码

有了任务的程序流程图后，编写任务函数的程序代码就比较顺利了。当然，真正要编写好任务函数的程序代码，还必要掌握更多的知识和技能，至少必须掌握后续各章节的内容。

这里以键盘任务为例（该任务涉及知识面较窄），任务函数的代码如下（实际代码比这要复杂一些）：

程序清单 L4-12 键盘任务函数

```
void TaskKey (void *pdata)          //键盘任务函数。
{
    INT8U key;
    while (1)                       //无限循环。
    {
        key=keyin();                //获取按键操作信息。
        switch (key)
        {
            case KEY_SAMP:          //“采集”按钮，创建数据采集任务。
                OSTaskCreate(TaskSamp,(void *)0,&TaskSampStk[TASK_STK_SIZE-1],2);
                break;
            case KEY_DISP:          //“显示”按钮，创建能谱显示任务。
                OSTaskCreate(TaskDisp,(void *)0,&TaskDispStk[TASK_STK_SIZE-1],7);
                break;
            case KEY_SUART:         //“发送”按钮，创建串行口发送任务。
                OSTaskCreate(TaskUart,(void *)0,&TaskUartStk[TASK_STK_SIZE-1],4);
                break;
            default : break;        //未按键或无效按键，不处理。
        }
        OSTimeDly(2);              //延时。
    }
}
```