第四章

嵌入式系统应用软件开发

主要内容

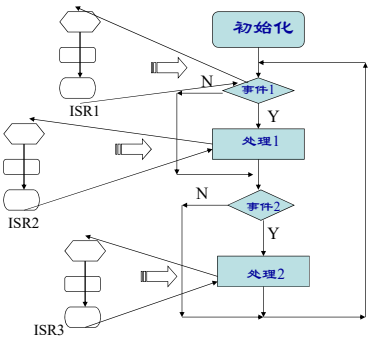
- 嵌入式应用软件运行模式
  - 无操作系统
  - 有操作系统（任务）
- 嵌入式系统开发流程和调试工具

嵌入式应用软件运行模式

- 无操作系统支撑
  - 前后台系统
  - 中断（事件）驱动系统
  - 循环服务系统
  - 基于定时器的循环服务系统

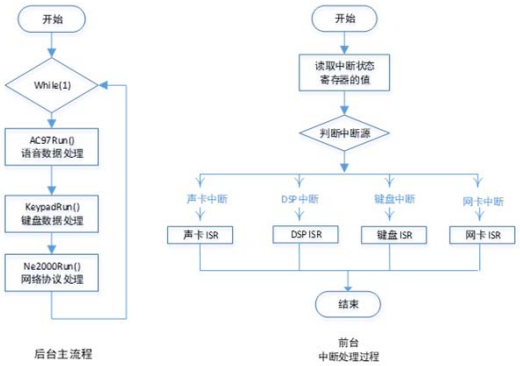
前后台系统

前台处理　　后台处理



- 后台(主程序)
  - 整个系统软、硬件资源的初始化分配
  - 无限循环执行多个事件，完成相应的操作(数据处理)。
- 前台(中断服务程序)
  - 强实时的关键操作

基于前后台系统的数字电话机



中断(事件)驱动系统

主程序

```
main()
{
    /* to do: 硬件初始化和数据结构的初始化*/
    /* to do: 系统的初始化*/
    /* 中断矢量表的适当处理*/
    while(1)
    {
    辅助性代码或
    Enter_low_power(); /* 进入低功耗状待机态*/
    }
}
```

中断服务程序

```
Isr_n() /* 其中的一个中断服务程序*/
{
   /* to do: 处理中断事件*/
   /* to do: 处理中断事件中采集的数据*/
}
```

循环服务系统

- 应用场景
  - 嵌入式处理器/控制器的中断源不多
  - 解决方案
    - 增加中断源-需要硬件，成本高
    - 软件方案-软件巡回服务

```
main()
{
   /* to do: 系统初始化*/
   while(1)
   {
      action_1(); /*巡回检测事件1并处理事件*/
      action_2(); /*巡回检测事件2并处理事件*/
      action_n(); /*巡回检测事件n并处理事件*/
   }
}
```

基于定时的循环服务系统

- 普通循环服务系统的缺点
  - 处理器全速运行，开销大-功耗高-电池供电系统
- 降低处理器工作时间——定时循环服务
- 构成
  - 主程序
  - 定时中断服务程序

主程序

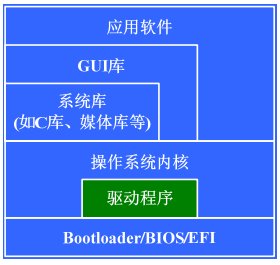
```
main()
{
    /* to do: 硬件初始化和数据结构的初始化*/
   /* to do: 系统的初始化*/
   /* 设置定时器中断服务*/
  while(1)
  {
    辅助性代码或
    Enter_low_power(); /* 进入低功耗状待机态*/
  }
}
```

定时器中断服务例程

```
Isr_timer() /* 定时器的中断服务程序*/
{
    action_1(); /*执行事件1的处理*/
    action_2(); /*执行事件2的处理*/
    …
    action_n(); /*执行事件n的处理*/
}
```

基于操作系统的应用开发

- 嵌入式系统软件结构
- 应用软件模型
  - 单处理器多任务
  - 多处理器多任务

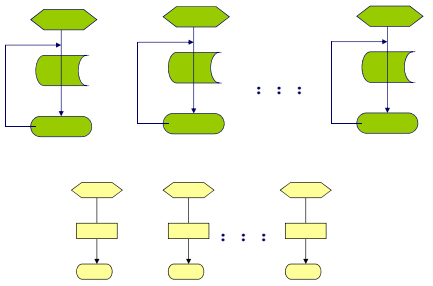


单处理器多任务系统

任务内部顺序执行，任务间的并行性通过操作系统来完成，任务间的相互通信和同步也需要操作系统的支持。

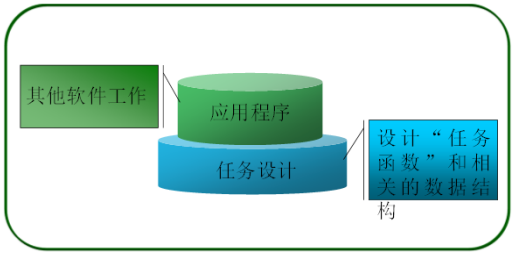
多处理器多任务系统

流程



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，其它软件设计工作都是围绕任务设计来展开。

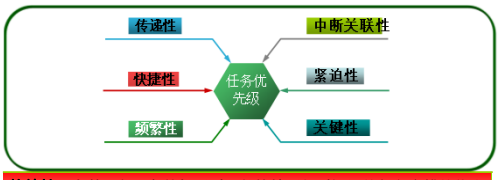


任务划分的一些原则

- 原则1：尽量将对同一个外设的访问放在一个任务中
  - 对每个独立的硬件(例如串行通信端口)进行操作的驱动程序段放在一个任务中。
- 原则2：避免长任务
  - 优先级低：执行中被频繁打断；现场保护占用资源
  - 优先级高：长时间占用CPU，其它任务得不到执行
  - 方法：任务分割；选择硬件指令执行
- 原则3：任务之间尽量不相关
  - 通过信号量、邮箱联系

任务设计 — 任务的优先级

任务的优先级安排原则如下：



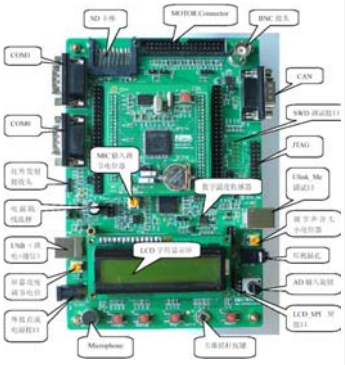
快捷性：在前面各项条件相近时，越快捷（耗时短）的任务安排的优先级越高，以使其它就绪任务的延时缩短；

排得比较低，CPU有可能被优先级比较高的任务长期占用，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；

基于μCOS-II的程序设计实例

实例环境介绍

- **硬件资源**
  - STM32F103 处理器
  - 高灵敏度数字温度传感器
  - 按键
  - 液晶显示
- **软件资源**
  - μCOS-II实时操作系统
- **目标应用**
  - 按键控制不同方式采样显示温度



实例任务划分

为了更合理的将整个系统划分为不同任务，要确定划分的原则和方法

- 任务划分的基本原则
- 任务划分的常用方法
- 本实例任务的划分

任务划分的基本原则

- 满足系统“实时性”
  - 若响应时间达不到要求，应用系统会出现错误甚至导致难以挽回的故障，故“实时性”是首要原则
- 较少资源需求
  - 尽量将使用同类资源的应用归入同一任务中，以减少操作系统调度时所消耗的资源
- 合理的任务数
  - 任务数目越多，加大了操作系统的调度负担，资源开销也随之加大；
  - 任务划分的数目太少，会增加每个任务的复杂性，使任务设计难度加大

任务划分的常用方法

- 以硬件模块为对象划分任务
  - 以硬件模块相关驱动为基础，根据硬件驱动在系统中的关键性设定优先级的任务划分方法
- 以实时性优先原则划分任务
  - 将对实时性要求较高的应用划分为单独任务，并赋予较高的优先级来保证整个系统实时性的要求
- 切分耗时任务
  - 将一些占用大量CPU处理时间的繁琐应用从系统中分离出来，作为一个优先级较低的任务在系统空闲时运行

本实例任务的划分

本实例被划分为7个任务：

- **采样任务（4个）**
  4个采样任务分别使用不同的采样条件：延时采样、使用系统时钟节拍采样、定时中断采样和使用高优先级中断的采样
- **负责和用户交互的键盘任务**
  键盘任务除负责接收用户输入并做出反馈外，还需要完成操作系统和系统资源的初始化，包括目标板的初始化，系统中用到的消息队列、邮箱和互斥信号量的创建等
- **显示任务**
- **向上位机传送数据的串口发送任务**

实例任务设计与优先级分配

- 优先级分配原则
- 实例任务优先级分配

优先级分配原则

- 外设相关任务安排高优先级
- 根据任务实现功能的重要性安排优先级
- 占用关键资源的任务优先级尽量高
- 周期性任务，执行周期越短的任务优先级高
- 以上条件相近时，耗时越短的任务优先级高

实例任务优先级分配

- 4个采样任务最高优先级，依次设为5、6、7、8
- 串口任务中高级别，设为13
- 键盘任务中等级别，设为15
- 显示任务最低优先级，设为17

```
void Task_FastSamp(void *pdata);    //使用高优先级中断的采样，优先级5
void Task_HookSamp(void *pdata);    //使用钩子函数的采样任务，优先级6
void Task_TimerSamp(void *pdata);    //使用定时中断的采用任务，优先级7
void Task_DelaySamp(void *pdata);    //使用延时函数的采用任务，优先级8
void Task_Send(void *pdata);    //串口发送任务，优先级13
void Task_Key (void *pdata);    //键盘任务，优先级15
void Task_Disply(void *pdata);    //显示任务，优先级17
```

基于启动任务的主程序

```
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
}
```

任务的数据结构设计

- 与操作系统有关的数据结构

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

任务参数表（用户定义）

任务堆栈（用户定义）

任务控制块（操作系统设置，包括优先级、信号量、消息邮箱、消息队列等，全局数据结构，可裁剪）

任务的数据结构设计

- 与操作系统有关的数据结构

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

- 与操作系统无关的数据结构
  - 每个任务需要处理的特定信息，包括变量、数组、结构体、字符串等；
  - 信息的生产者和消费者都是同一个任务，定义为私有变量；否则定义为全局变量。

基于用户任务的主程序

```
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;
}
```

单次执行任务

```
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，创建串行口发送任务
                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)      //串行口发送任务（示意）
{
    pdata = pdata;
    串行口初始化；
    组织发送帧；
    数据指针初始化；
    发送数据；
    OSTaskDel (OS_PRIO_SELF);    //删除自己
}
```

周期性执行任务

- 只要创建一次，就能周期运行
  - 键盘扫描任务、显示刷新任务、模拟信号采样任务等。

```
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( )
        }
}
```

事件触发执行任务

```
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;
        }
}
```

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;
        }
}
```

消息触发任务

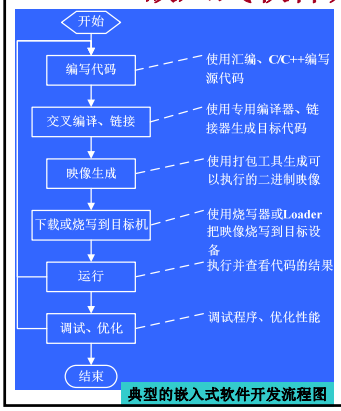
```
OS_EVENT *Mybox;          //消息邮箱
void   TaskKey (void *pdata)             //键盘任务函数（示意）
{
    INT8U key;
    INT16U baud;                         //波特率，由用户通过键盘选定
    for (;;)                             //无限循环，也可用 while (1)
        {
        key=keyin();                     //读入按键操作信息
        switch (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)        //串行口发送任务（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
        串行口初始化;                    //用获取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;
        }
}
```

嵌入式软件开发流程

嵌入式软件开发流程

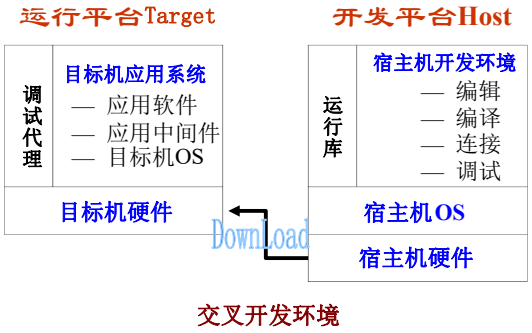
嵌入式软件开发流程



典型的嵌入式软件开发流程图

- 一般地来说，嵌入式软件开发依次要经历编辑（代码准备）、编译、重定位（定址和打包）、烧写、下载、调试、优化等步骤，在某些嵌入式系统中可能还需要测试和验证等步骤。
- 从流程上说，大致可分为编码阶段、构建阶段、部署阶段、调优和其他阶段。

嵌入式软件的交叉开发环境



交叉编译

- 交叉编译：在一个架构下编译另一个架构的目标文件。
- 采用何种交叉编译器产生何种格式的目标文件还要取决于目标机的操作系统。
- 交叉编译器和交叉链接器是能够在宿主机上运行，并且能够生成在目标机上直接运行的二进制代码的编译器和链接器。

部署阶段

- Flash芯片的烧写，类似可编程器件的烧录，可分为
  - 脱线方式（运行阶段）
    - 需要将Flash ROM芯片从目标板上取下
      - 一般目标板将Flash ROM做成插座式安装，而非焊死
    - 使用专门的编程器和配套软件
  - 在线方式（开发阶段）

嵌入式软件调试

- **调试技术对嵌入式软件开发至关重要，如果将错误带到最终的嵌入式系统产品中，其后果可能是致命的。**
  - 嵌入式软件**调试的效率比较低**（即使改动一行代码，都可能需要把编译、链接、重定位、烧写、下载等步骤完整走一遍）
  - **快速地定位到问题**可以节省大量的时间。

嵌入式系统调试的特色

- 嵌入式软件调试的最大问题：缺少监控程序实时运行的手段
- 调试的基本原则：需要找出一种监控程序运行的手段，让自己知道代码到底是如何运行的
  - 通过一两行汇编代码让开发板上的LED灯闪烁
  - 初始化一个开发板上的串口，然后在PC端通过终端软件接受开发板上串口发送的信息

嵌入式系统调试

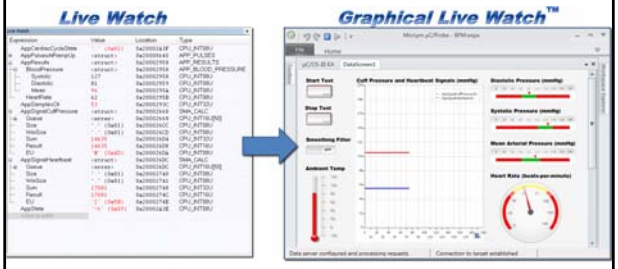
- 基于软件工具的调试
  - 类似于Windows上的软件调试的机制
  - 优点：可以观察到操作系统层次的一些机制，如虚拟内存、线程等
  - 缺点：目标硬件平台必须已经相当稳定，而且具备了运行软件debugger的必要环境（如已能运行操作系统）

嵌入式软件调试工具

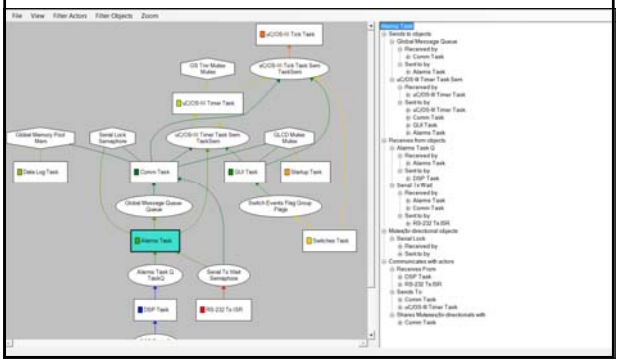
嵌入式软件的功能越来越复杂，无法再依靠经验开发。越来越多的RTOS厂商提供各类辅助工具。

- 嵌入式软件调试
- 嵌入式软件优化
- 嵌入式软件验证与测试

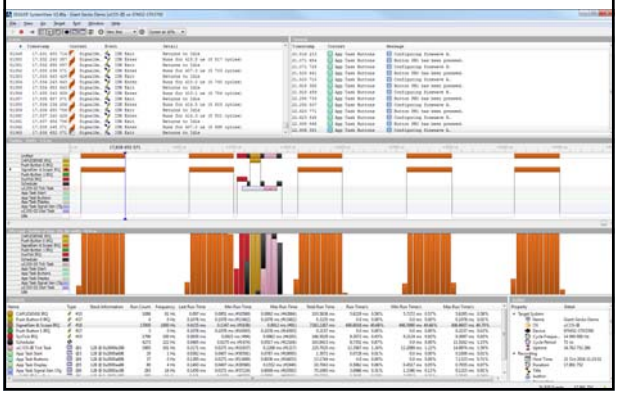
µC/Probe Graphical Live Watch



Tracealyzer for µC/OS-III



SystemView for µC/OS

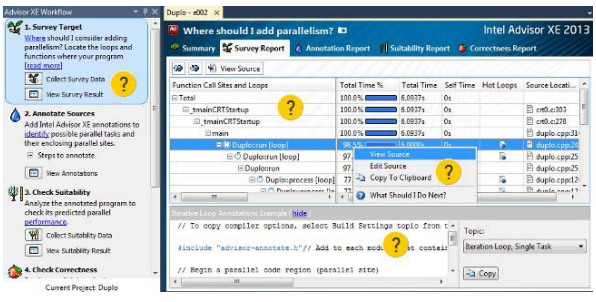


软件优化手段

- 代码剖析（code profiling）：借助工具分析程序代码，精确分析性能瓶颈，据此引导和建议开发者进行改进。
  - 对应的工具常常称为剖析器（profiler）或性能分析器（performance analyzer）
  - Intel Parallel Studio

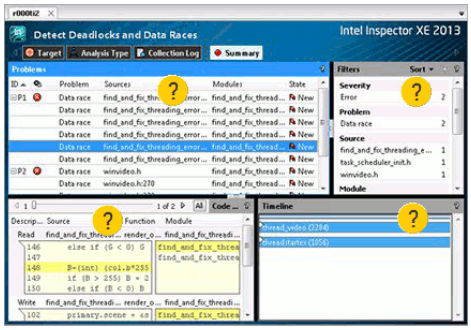
Intel® Advisor XE

Find where to add parallelism to your code



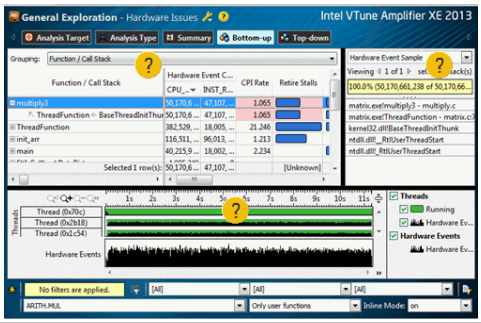
Intel® Inspector XE

improving code reliability and quality



Intel® VTune™ Amplifier XE

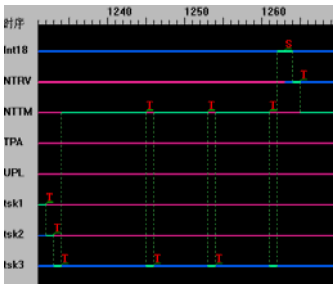
improving your code performance



软件验证与测试

- 在某些特殊的嵌入式系统上，软件开发还需进行一些特殊的步骤。
  - 软件验证是其中之一，软件验证是验证程序逻辑上的正确性和常见的错误。这对于某些难以测试和调试的环境，例如航空，逻辑验证显得至关重要。
  - 软件测试中借助专门的工具来帮助测试人员找出程序中存在的错误。在一定程度上人力难以实现此工作，例如压力测试和自动测试

测试工具实例：逻辑分析仪



嵌入式应用软件运行的逻辑流程

嵌入式软件的测试

- 嵌入式软件测试中经常用到的测试工具有：
  - 内存分析工具
  - 性能分析工具
  - 覆盖分析工具
  - 缺陷跟踪工具等

覆盖分析工具实例