第四章 嵌入式 C 语言程序设计

作为嵌入式软件开发者，应该对嵌入式软件的编译和链接原理有一定的了解，这样才能解决程序在编译和链接过程中出现的错误。开发者需要对 C 语言程序的内存分配原理和方式形成一定的理解，从而避免 C 语言程序中常见的各种内存错误。本章对嵌入式裸机软件的几种常用流程进行了讲解，概述了嵌入式软件模块化的编程方法，有助于读者学习如何构建一个嵌入式应用软件的流程和框架。本章最后介绍了代码编写过程中应该遵守的一些基本规范，以减少编码错误的发生。

1 编译和链接

由于计算机只能识别二进制的机器码，用 C 语言或汇编语言编写的源程序，需要翻译成机器码才能在计算机上运行。把嵌入式软件的源代码转换为可执行的二进制映像的过程，分为三个步骤。

1) 首先，每一个源文件(包括汇编语言源文件和 C 语言源文件)都必须被编译或汇编到一个目标文件（ *.obj 或*.o），目标文件主要由二进制的机器码组成。
2) 然后．第一步产生的所有目标文件要被链接成一个目标文件，它叫做可重定位程序 （relocatable program）。
3) 最后，在一个称为重定位（relocation）的过程中，要把物理存储器地址指定给可重定位程序里的每个相对偏移处。第三步的结果就是产生一个包含可执行二进制映像的文件。

第 2 步往往和第 3 步合在一起，称为链接。

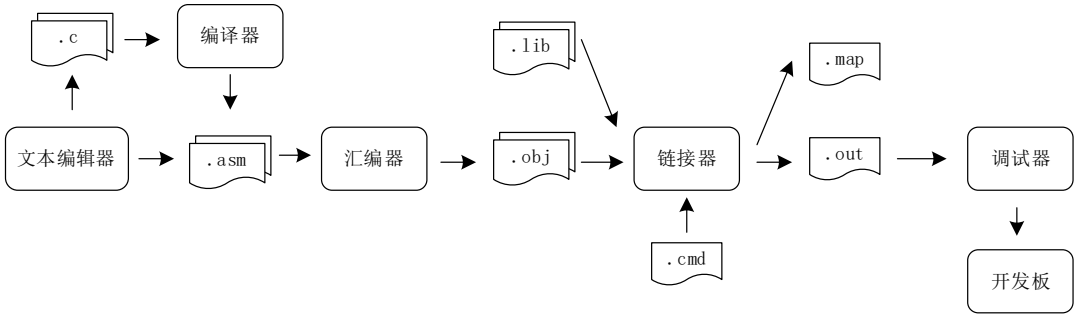

图 4-1 编译和链接过程

编译(Compile)

编译器的工作是把使用人容易读写的语言编写的程序，翻译为相应处理器上等效的一组机器码。源文件经过编译后生成目标文件 file.obj，目标文件就是经过编译产生的二进制文件。

在 C 语言程序中，指令和数据是交杂在一起的，例如可以在语句中间定义新的变量。

但在计算机中，指令和数据是分开存储的，指令在程序存储区，数据在数据存储区。因此在目标文件中，编译器会将程序中的指令和数据分开，汇集到不同的段中。比如，所有代码块都被放到 text 的段中，初始化的全局变量和静态变量放到 data 的段中，未初始化或初始化为零的全局变量和静态变量放到 bss 的段中。

程序一般会有多个源文件，每个源文件都会由编译器生成一个目标文件。目标文件中有 1 个符号表，记录源文件引用的所有变量和函数的名字和位置。其中，有些变量和函数是在其他的源文件中声明和定义的，因此目标文件中记录的符号可能是不完整的（在本目标文件中找不到这些函数和变量的定义）。

在编译过程中，编译器会找出源文件中的语法错误和词法错误。程序员可根据编译器输出的出错信息来修改源文件，直到编译器生成了正确的目标代码。

链接(Link)

链接器的工作是将程序中多个目标文件组合到一起，生成一个总的目标文件。链接器将各个目标文件中的 text、data 和 bss 段合并到一起，规划到统一的地址空间中。所有目标文件里的机器语言代码汇集到新文件的 text 段里，所有初始化的变量汇集到 data 段，所有未初始化变量汇集到 bss 段。链接器除了链接源文件生成的目标文件外，还要链接各种库文件(*.lib)，如操作系统库文件、标准 C 函数库文件、设备驱动库文件等。

在链接过程中，链接器还要解决不完整的符号问题。例如，一个目标文件中引用了一个未定义的函数 foo，而在另一个目标文件中发现了一个名字同为 foo 的函数定义，那么链接器将把它们匹配到一起，使用找到的 foo 函数定义来对应未解决的引用。

最后，需要为链接器提供关于目标电路板上的存储器信息。链接器将使用这个信息来为可重定址程序里的每一个代码和数据段指定存储器物理绝对地址。然后将产生一个包含二进制映像的输出文件（file.out）。这个文件就可以被下载或烧录到目标 ROM 或 FLASH 中执行。

以 workshop 的 lab2 工程为例，工程中包含了 main.c 和 tm4c123gh6pm_startup_ccs.c 两个源文件，编译后会在 debug 目录下生成这两个源文件的目标文件 main.obj 和 tm4c123gh6pm_startup_ccs.obj。

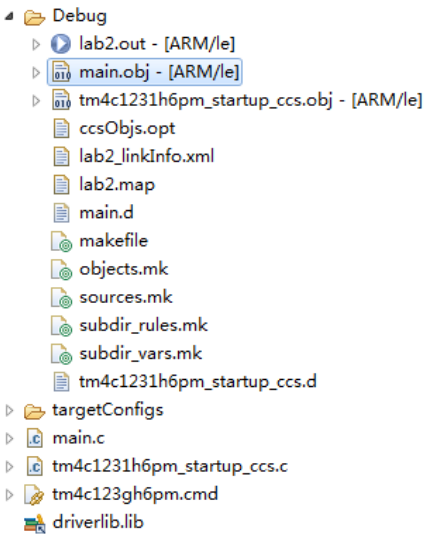

图 4-2 工程文件目录和生成的 obj 文件

```
MEMORY
{
    FLASH (RX) : origin = 0x00000000, length = 0x00040000
    SRAM (RWX) : origin = 0x20000000, length = 0x00008000
}

/* The following command line options are set as part of the CCS project.    */
/* If you are building using the command line, or for some reason want to    */
/* define them here, you can uncomment and modify these lines as needed.     */
/* If you are using CCS for building, it is probably better to make any such */
/* modifications in your CCS project and leave this file alone.              */
/*                                                                           */
/* --heap_size=0                                                             */
/* --stack_size=256                                                          */
/* --library=rtsv7M4_T_le_eabi.lib                                           */

/* Section allocation in memory */

SECTIONS
{
    .intvecs:   > 0x00000000
    .text    :  > FLASH
    .const   :  > FLASH
    .cinit   :  > FLASH
    .pinit   :  > FLASH
    .init_array : > FLASH

    .vtable :   > 0x20000000
    .data    :  > SRAM
    .bss     :  > SRAM
    .sysmem :   > SRAM
    .stack   :  > SRAM
}

__STACK_TOP = __stack + 512;
```

图 4-3 tm4c123gh6pm.cmd 文件

在 tm4c123gh6pm.cmd 文件中，MEMORY 定义了芯片中各存储器的起始地址和容量；SECTIONS 指定了各段的地址信息。在链接过程中，依据 tm4c123gh6pm.cmd 文件中的信息进行重定址。链接完成后，代码和数据段的绝对地址可以在 debug/lab2.map 中查到。

```
36 --------  ----  ----------  ----------   ----------------
37 .intvecs   0    00000000    0000026c
38                   00000000    0000026c     tm4c1231h6pm_startup_ccs.obj (.intvecs)
39
40 .text      0    0000026c    0000066c
41                   0000026c    00000132     driverlib.lib : gpio.obj (.text:GPIOPadConfigSet)
42                   0000039e    00000002     rtsv7M4_T_le_v4SPD16_eabi.lib : mpu_init.obj (.text)
43                   000003a0    000000f4     driverlib.lib : sysctl.obj (.text:SysCtlClockSet)
44                   00000494    0000009c     rtsv7M4_T_le_v4SPD16_eabi.lib : memcpy_t2.obj (.text)
45                   00000530    0000007a                                   : memset_t2.obj (.text)
46                   000005aa    00000002                                   : startup.obj (.text)
47                   000005ac    00000070                                   : autoinit.obj (.text)
48                   0000061c    0000006a                                   : copy_decompress_rle.obj (.text)
49                   00000686    00000002     --HOLE-- [fill = 0]
50                   00000688    00000054     main.obj (.text)
51                   000006dc    00000054     rtsv7M4_T_le_v4SPD16_eabi.lib : boot.obj (.text)
52                   00000730    00000054                                   : exit.obj (.text)
53                   00000784    0000004c                                   : cpy_tbl.obj (.text)
54                   000007d0    00000040     tm4c1231h6pm_startup_ccs.obj (.text)
55                   00000810    00000034     driverlib.lib : sysctl.obj (.text:SysCtlPeripheralEnable)
56                   00000844    0000002a                   : gpio.obj (.text:GPIODirModeSet)
57                   0000086e    00000018                   : gpio.obj (.text:GPIOPinTypeGPIOOutput)
58                   00000886    00000002     --HOLE-- [fill = 0]
59                   00000888    00000018     rtsv7M4_T_le_v4SPD16_eabi.lib : args_main.obj (.text)
60                   000008a0    00000014                                   : _lock.obj (.text)
61                   000008b4    0000000e                                   : copy_decompress_none.obj (.text:decompress:none)
62                   000008c2    00000006     driverlib.lib : gpio.obj (.text:GPIOPinWrite)
63                   000008c8    00000006                   : sysctl.obj (.text:SysCtlDelay)
64                   000008ce    00000006     rtsv7M4_T_le_v4SPD16_eabi.lib : copy_decompress_rle.obj (.text:decompress:rle24)
65                   000008d4    00000004                                   : pre_init.obj (.text)
66
67 .cinit     0    000008d8    00000038
68                   000008d8    00000016     (.cinit..data.load) [load image, compression = rle]
69                   000008ee    00000002     --HOLE-- [fill = 0]
70                   000008f0    00000008     (__TI_handler_table)
71                   000008f8    00000007     (.cinit..bss.load) [load image, compression = rle]
72                   000008ff    00000001     --HOLE-- [fill = 0]
73                   00000900    00000010     (__TI_cinit_table)
74
75 .init_array
76 *         0    00000000    00000000     UNINITIALIZED
77
78 .stack     0    20000000    00000200     UNINITIALIZED
79                   20000000    00000200     --HOLE--
80
81 .data      0    20000200    00000018     UNINITIALIZED
```

图 4-4 map 文件

启动代码（startup code）

启动代码（startup code）是系统上电或者复位后运行的第一段代码，是用来为高级语言写的软件做好运行准备的一小段汇编语言代码。C 语言程序的运行需要具备一定的条件，如初始化数据空间，堆栈空间和中断入口等。启动代码的作用是在用户程序运行之前对系统硬件及软件环境进行必要的初始化并在最后使程序跳转到用户程序（main 函数）。开发环境往往自动完整的提供了这个启动代码，不需要开发人员再编写。

例如 CCS 中的 tm4c123gh6pm_startup_ccs.c 文件中，定义了中断向量表和中断服务函数。中断向量表的第一条定义了堆栈栈顶的初始值。复位中断服务函数 ResetISR 中的_c_int00 定义了复位后的初始化操作。

2 堆栈和函数调用(内存的分配)

微控制器的内存资源比较少，在微控制器的 C 语言程序设计中更容易出现内存越界和内存溢出等错误。因此，开发者需要充分理解 C 语言中的内存分配机理，比如全局变量和局部变量是如何处理的，栈存储是如何工作的，才能规避这些错误。

2.1 内存的分配方式

C 语言的内存（RAM）分配方式主要有 3 种：

1) 静态存储区：对于全局变量、静态(static)变量，在程序编译时就会为其指定固定的内存空间，这块内存在程序的整个运行期间会一直被它们占用。CCS中可以在编译后生成的map文件中查到各全局变量和静态变量的物理地址。
2) 栈(Stack)存储：函数内的局部变量一般在栈上创建存储空间，函数执行完返回时将释放这些存储空间。由此可以看出，局部变量的存储期限是从变量声明到函数执行结束。
3) 动态存储分配（动态存储区也称为堆，Heap）。在程序执行时使用malloc或new函数申请内存单元，不需要时再使用free或delete函数释放掉这些内存单元。动态存储的使用非常灵活，例如可以按需求生成数据空间，以及改变数据空间的大小，但也最容易发生错误。

例如新建项目，将下面的例程添加到main.c文件。在cmd文件中增加`--heap_size=2048`语句，指定堆的大小。

```
/----------------------------------------内存分配----------------------------------------
#include <stdio.h>
#include <string.h>
char *g_pstr;                    //全局变量分配在全局未初始化区bss段
const char g_src[]="dsrc";       //常量分配在代码区的const段
 void main()
{
   int slen;                     //局部变量分配在栈区
   static char s_src[]="12345";  //静态局部变量分配在全局初始化区data段
```

```
    slen=strlen(g_src);
    g_pstr=(char*)malloc(slen);   //在堆区sysmem段分配空间
    memcpy(g_pstr,g_src,slen);
    free(g_pstr);                 //在堆区释放空间
    slen=strlen(s_src);
    g_pstr=(char*)malloc(slen);  //在堆区sysmem段分配空间
    memcpy(g_pstr,s_src,slen);
    free(g_pstr);                 //在堆区释放空间
    while(1)
    {
    }
}/-------------------------------------------------------------------------------------------
```

编译后，在 map 文件中可以查询到：

未初始化的全局变量 g_pstr 分配在全局未初始化段(bss)。

字符常量 g_src 分配在代码区的 const 段。

初始化的静态局部变量 s_src 分配在全局初始化段(data)。

运行程序后，可以在 Expressions 窗口观察到局部变量 slen 是在栈区分配存储空间，malloc 函数在堆区为 g_pstr 分配存储空间。

2.2 栈存储

栈存储(Stack)又称为堆栈，是计算机中常用的一种数据缓冲存储机制。堆栈只允许在一端进行操作，按照后进先出（LIFO--Last In First Out）的原理运作。Cortex-M 处理器使用 PUSH 指令向堆栈中存入数据（又称压栈或入栈），使用 POP 指令从堆栈内取出数据（又称出栈），后入栈的数据会先出栈，所以是后进先出。栈指针 SP(R13)记录栈顶的位置，以标记入栈的深度。

Cortex-M4 按照“满递减”的方式组织堆栈，堆栈的入栈顺序是从高地址向低地址方向进行(入栈后栈指针 SP 向低地址递减)，并且栈指针 SP 总是指向最后一个入栈的数据。SP 是 4 对齐的，最低两位始终为 0，也就是说每次入栈或者出栈的数据是以字（4 字节）为单位，每入栈 1 个字，SP 的值自动减 4。

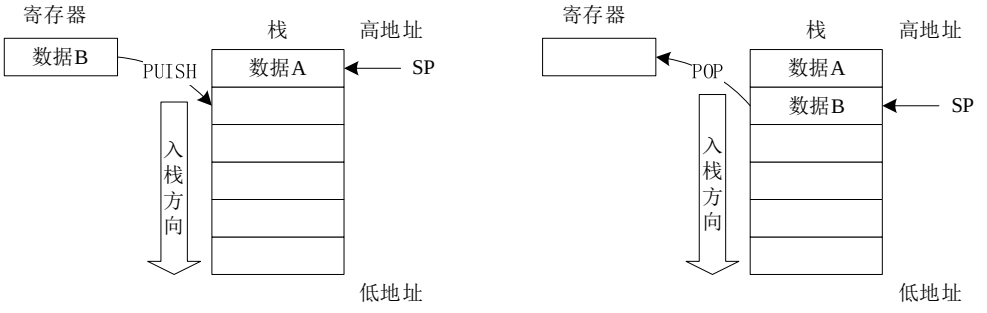

图 4-5 Cortex-M4 的“满递减”栈存储

栈主要被用来在函数调用或者异常产生时保存临时数据。具体来说，C 编译器主要通过栈完成以下 5 项工作。

1) 使用栈保存被调函数运行时要用到的寄存器的值。如果被调函数执行时用到了一些寄存器作为数据缓存，而且这些寄存器可能已经被使用，此时C编译器在函数调用时，需要增加压栈的代码，将这些寄存器中的原有值保存到栈中。在函数调用结束后，再从栈中恢复这些寄存器的值。
2) 在函数调用时，向函数传递参数。C编译器通过将函数的“实参”值压入栈来向函数传递信息。
3) 保存函数的返回地址。函数调用的返回地址会由硬件自动保存到链接寄存器LR中，但如果出现函数的嵌套调用时，上级函数的返回地址会由编译器插入的代码压入栈，待上级函数返回时，再将返回地址弹出到PC寄存器中。异常发生时，返回地址也会被硬件自动压入栈中。
4) 在栈中为局部变量分配内存空间。编译器一般把函数内声明的局部变量分配到栈中。值得注意的是，编译器有时会把局部变量分配到CPU内部的通用寄存器中，由于寄存器的访问速度远快于内存的访问速度，这样做可以优化程序的执行速度。
5) 在异常产生时，Cortex-M4自动将处理器的状态(xPSR)、一些寄存器(PC、LR、R0~R3、R12)的数据保存到堆栈内。

以下面的程序为例。

```
/---------------------------------------函数调用栈-------------------------------------------
uint32_t func(uint32_t parm) //被调函数
{
    uint32_t temp;
    ……
    Return temp;
}

main()
{
   uint32_t arg;
    ……
    arg = func(arg);          //调用函数
    ……
}
/-------------------------------------------------------------------------------------------
```

main 函数在执行到 arg＝func (arg)时，C 编译器在栈中的操作如下：

1) C 编译器增加汇编代码，将 func 函数中可能用到的寄存器的值压入栈中保存起来。func 函数执行时，会使用一些寄存器来保存临时数据。

2) 将上一级函数的返回地址压到栈中；更新 LR 寄存器，保存 func 函数执行完时的返回地址。
3) 在栈中为函数的形参 parm 分配存储空间，并减小 SP 的值。
4) 在栈中为函数的局部变量 temp 分配存储空间，并减小 SP 的值。
5) 在函数 func 执行完时，会增加 SP 的值，同时将函数的返回值 temp 存放在 R0 中，由 R0 来传递给调用函数。

和函数调用栈类似，系统在异常发生时也会使用堆栈保存返回地址等过程信息。但和函数调用不一样的是：Cortex-M4 异常调用栈主要由硬件自动执行，将处理器的状态(xPSR)和其他一些寄存器的值压入栈中保存；异常服务函数是没有输入参数和返回值的，因此也不用传递参数。但如果异常服务函数在执行过程中使用了过多的通用寄存器(R4~R11)，则这些寄存器的值必须由 C 编译器插入的代码保存和恢复，以确保这些寄存器的值不会被异常服务函数修改。

2.3 堆栈溢出

堆栈溢出是指堆栈中压栈的数据过多，以致超过了堆栈的最大空间，压栈的数据进入了其他的存储空间。堆栈溢出会造成压栈的数据丢失或者其他内存空间的数据被修改。堆栈溢出往往会产生一些出乎意外地错误，而且不容易找出错误原因。

造成堆栈溢出的主要原因有：

1) 栈地空间太小。如果程序员在一开始分配了太小的堆栈空间，则很容易造成堆栈溢出。
2) 函数中使用了大容量的局部变量。函数的局部变量是在栈中分配内存空间，如果在函数中声明大的数组，将占用大量的栈空间。
3) 程序使用了过深的函数递归。每级函数调用都会产生函数调用栈，如果程序中函数的嵌套调用太深，则需要多级函数调用栈，消耗大量栈空间。

在 MCU 系统中，其 RAM 空间一般较小，对栈空间的分配和使用要非常小心，特别要注意以下几点：

1) 在函数和中断服务函数中的局部变量，特别是大型数组和数据结构；
2) 函数的嵌套与递归的深度（级数）；
3) 中断嵌套的深度；
4) 库函数需要的栈空间。

3 裸机嵌入式软件流程

裸机嵌入式软件是指没有操作系统的嵌入式软件。裸机系统往往不能支持多任务并发操作，而是宏观串行地执行每一个任务。而多任务系统则可以宏观并行（微观上可能串行）地“同时”执行多个任务。

3.1 基本流程

最简单的程序只有一个任务，则其典型流程为：

1) 从 CPU 复位时的指定地址开始执行 startup 代码，然后跳转至用户主程序 main()；

2) 初试化各硬件设备和各软件模块；

3) 进入无限循环，处理任务。

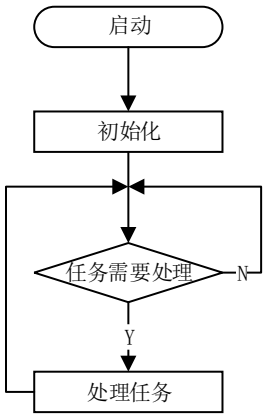

图 4-6 简单应用程序流程

所有的嵌入式主程序最后都会以无限循环结束，其首选方案是 while 循环：

```
/-------------------------------------- ------------------------------------------------
while(1)
{
    任务处理;
}
/-------------------------------------- ------------------------------------------------
```

对于只需执行一次的任务，则将任务处理放到 while 循环前：

```
/-------------------------------------- ------------------------------------------------
任务处理;
while(1)
{
}
/-------------------------------------- ------------------------------------------------
```

3.2 循环轮询

对于大多数嵌入式软件，都会有多个任务需要处理，例如对于一个仪表程序，在执行过程中需要执行 AD 采样、计算测量结果、监测按键、串行通信，显示测量结果等等多个任务。这些事件都会占用处理器时间和内存空间，此时需要采用合适的程序流程，以保证任务能及时有效的得到处理，同时还要考虑程序的稳定性，易开发和维护性。

在循环轮询系统(polling loop)中，程序循环检查每一个任务的执行条件，满足条件则执行，一直循环下去。

循环轮询系统容易编程和理解，但是轮询难以区分任务的优先级。当系统规模大、任务多时，每一个任务都要等到其它所有任务处理完之后才能得到处理，很多时候紧急的任务不

能得到及时响应，因此循环轮询系统只适合慢速系统和简单系统。

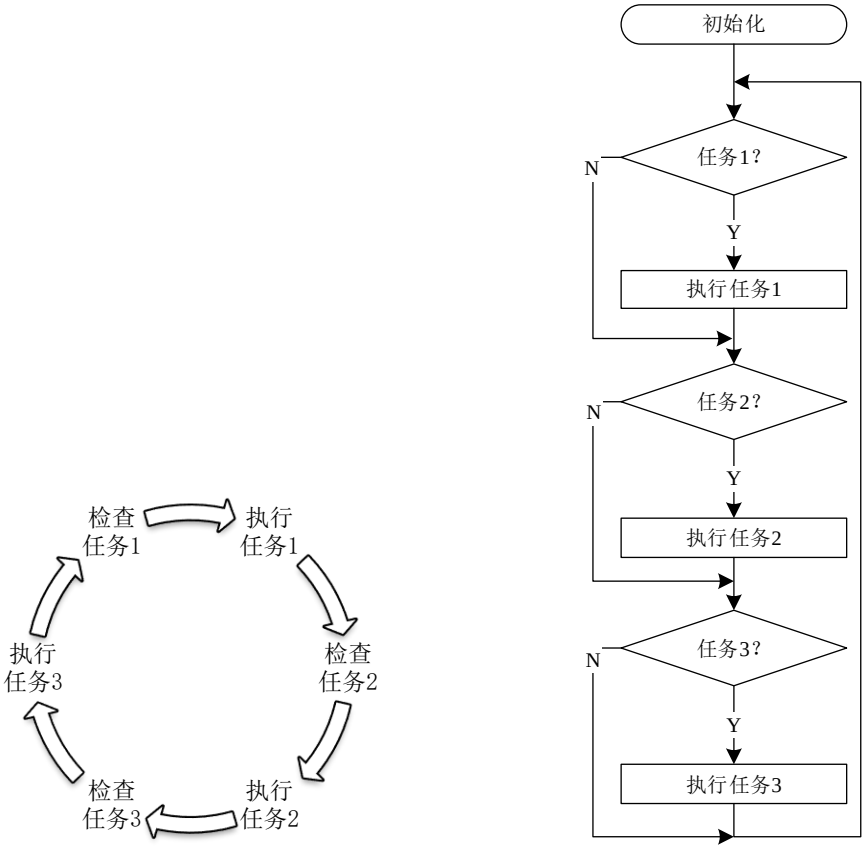

图 4-7 循环轮询流程

3.3 前后台系统

循环轮询在没有任务需要执行的时候，也会一直循环查询，处理器持续运行，因此能量消耗较大。如今很多微控制器采用了中断驱动的模式，在没有外部事件需要处理的时候，处理器保持休眠，有事件到来时，使用中断唤醒处理器，处理完事件后再次进入休眠。

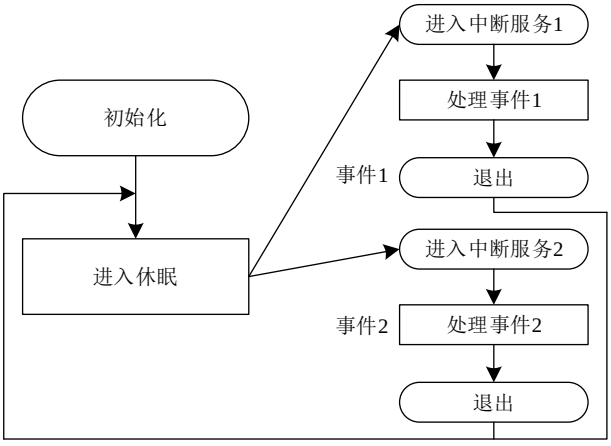

图 4-8 中断驱动模式

将循环轮询和中断驱动结合起来，就产生前后台系统（Foreground/Background System），这是裸机最常采用的模式。前后台系统把程序分成前台程序和后台程序。主程序是一个无限的循环，在循环中检查任务队列，并根据响应速度的要求，处理大部分任务，这部分可以看成后台程序。中断服务程序处理中断事件，这部分可以看成前台程序。

前后台系统将任务或者事件划分为了实时性要求高和实时性要求不高的两类。实时性高的事件在请求发生（中断产生）时立即得到处理（在中断服务函数中处理事件）。对于实时性要求不高的事件则只在中断服务程序中标记事件的发生，不再做任何工作就退出中断。在后台程序（主循环）中循环检查是否有这些事件的标记（是否到来），如果有，则处理事件。如果一段时间没有事件到来，微控制器还可以进入休眠。

前后台系统在前台处理高优先级的事件，使得可以及时响应紧急事件。低优先级事件在前台只做标记，而在后台处理事件，避免了因为在中断服务程序占用过多时间，而造成其它中断无法及时响应，或者出现多重中断嵌套而导致堆栈溢出。

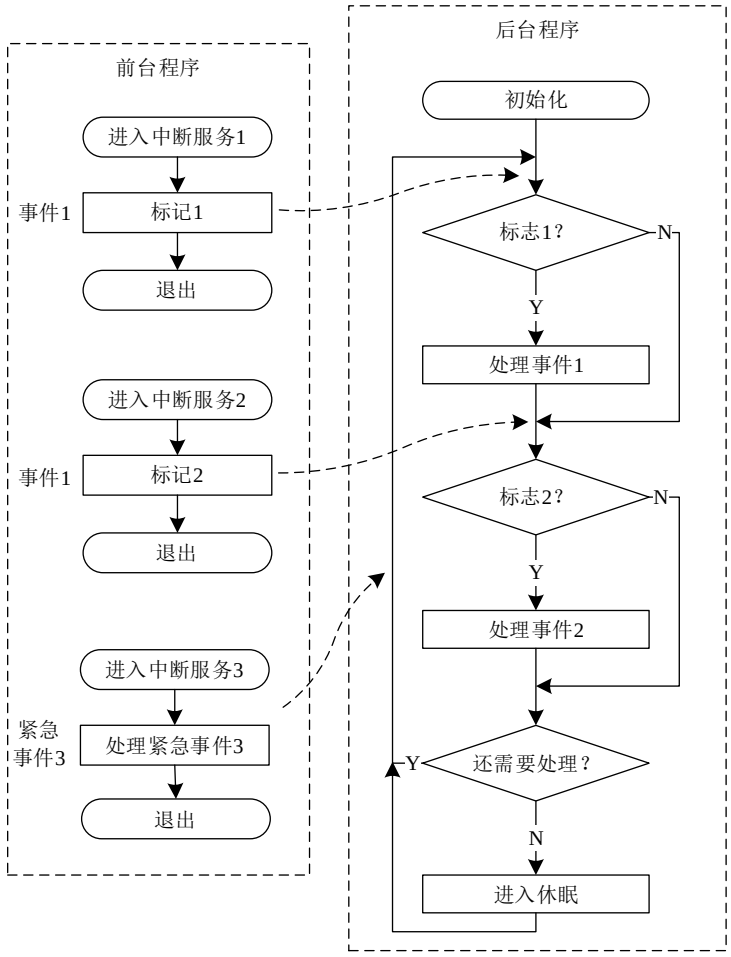

图 4-9 前后台系统

例如：在设计一个三相电能质量分析仪时，软件系统需要完成以下任务：

1) 每 0.5 秒对 LCD 显示屏进行一次刷新。
2) 每 0.2 秒对按键进行一次检测。
3) 每 0.5 秒对测量数据进行一次计算。

4) 通过 UART 串行接口与上位机通信，波特率为 115200bps。

5) 对采集到的 10 个周期的信号数据进行 FFT 变换。

在上面这些任务中，任务 4 对实时性要求比较高，对接收到缓冲器中的数据如果不能及时处理，可能会造成数据丢失，应放到前台（中断服务函数）中执行，可以随时打断其它的任务，保证其能得到及时处理。其它任务对实时性要求不高，稍微延迟一点处理也不会造成严重后果。任务 3 和任务 5 花费的时间较长，应避免在前台（中断服务函数）中处理，否则可能造成紧急任务无法及时响应（不使用抢占优先级），或者中断嵌套和堆栈溢出的概率增大，系统稳定性下降（使用抢占优先级）。所以任务 1、2、3、5 都可以在中断服务函数中只做标记，快速退出，而在主程序中排队执行。

下面是一个前后台程序实例（省略 main 函数）：要求 Led0 每 1 秒改变一次状态，Led1 每 2 秒改变一次状态。这个程序中有两个事件：Led0 和 Led1，由定时器中断驱动。在前台程序（中断服务函数 TimerISR）中改变两个事件的标记；在后台程序（while 循环）中检查标志，满足条件则处理事件（改变 Led 的状态）。

```
/---------------------------------------前后台程序实例---------------------------------------
led0_counter = 1;      // led0 的标志
led1_counter = 2;      // led1的标志
while (1)             // 后台程序
{
     if (led0_counter == 0)         //检查led0的标志，如果为0，则事件led0已经到来，需要处理
     {
          led0_counter = 1;         //恢复标志
          Led0Change();             //改变led0的状态
     }
     if (led1_counter == 0)         //检查led1的标志，如果为0，则事件led1已经到来，需要处理
     {
          led1_counter = 2;         //恢复标志
          Led1Change();             //改变led1的状态
     }
}

void TimerISR(void)                 //定时器中断服务函数，每秒钟执行 1 次
{
     if (led0_counter)
     {
          led0_counter--;           //将 led0 的标志减 1
     }
     if (led1_counter)
     {
          led1_counter--;           //将 led1 的标志减 1
     }
}
```

/----------

前后台适合于资源高度受限的微控制器系统，但由于该模式没有提供多任务的协调和通信机制，所以在面临复杂的大型系统时，复杂度提高，稳定性下降，特别是对于系统中多个任务需要同时处理时就更力不从心了，此时就需要嵌入式操作系统的支持。

4 模块化设计

4.1 模块化设计的概念

随着微控制器性能的提升、存储器容量增大和外设类型增多，微控制器能实现更强大的功能。相应的，微控制器软件的规模也变得更大，开发的复杂度和工作量大大增加，往往需要进行模块化(modularity)设计，由多人组成的团队完成。

将软件划分为多个相对独立的模块，每个模块实现软件的一部分功能，由这些模块组合起来满足问题的需求。模块化使得完成软件所需的工作量和时间更少。其原理是：如果将一个复杂问题分解为若干个的小问题，把这些小问题“分而治之”，问题的复杂性将会下降，更容易解决。同时，借鉴机械、建筑等传统行业的生产模式：先生产标准的零件，再将这些零件组装成产品。将软件模块设计为相对独立、接口规范和容易复用的软件构件(Software Component)，在软件开发中大量复用构件，大大提高了开发的效率和质量。

模块化使得软件结构清晰，易于理解，便于开发的组织和管理。模块化有助于团队开发，可以将模块分派给不同成员开发，在使用其他成员开发的模块时，只需知道其功能，而不需要知道实现细节。模块化有助于软件的维护，独立模块的测试更容易，修改造成的副作用更可能限定在小范围内，更易查找和修订错误，软件的可靠性更高。

4.2 模块的划分

将系统分解成子系统有两种方式：分层(layer)和分区(partition)。分层是将软件结构按抽象级别划分为不同的层次，分区是划分层内部的模块。人类在解决问题的时候，往往采用“自上而下”的方法：先从全局的角度，概括的描述问题的解决方案，再逐步细化每一个部分，直至可以直接执行的操作。类似的，软件结构也可以采用自顶而下的层级结构。上层的抽象程度较高，负责系统功能的实现和全局控制；下层的抽象程度较低，负责局部功能的具体处理。

典型的三层结构系统也可用于裸机系统。

1) 应用层：调用低层的功能模块实现系统功能，提供与用户的交互。
2) 应用逻辑层：针对项目具体需求的功能模块，它们的协作能实现业务功能。
3) 硬件驱动层：提供通用的驱动服务，与具体的应用没有直接关系，例如：片内硬件设备驱动函数和片外硬件设备驱动函数。

层次架构的上层面向具体的项目问题，下层面向通用的基础设施。当项目需求发生变化时，只需修改上层的业务逻辑；当需要移植到新的硬件平台时，只需替换掉下层的驱动程序。

模块划分时需要遵循一些基本原则：

1) 低耦合：模块间应尽量独立，模块的接口较为简单，和其他模块的联系较少。
2) 高内聚：模块的功能应该较为专一，模块内的元素相互联系紧密，都是围绕着这个功能。
3) 信息隐藏：模块对其他模块隐蔽部分信息（模块的具体实现细节），外部不能访问这些非必要的信息，以保护模块的内部数据和逻辑不被篡改。

4.3 模块在 C 语言中的具体实现

在C语言中，可以将每个功能独立的模块编写为一个源文件，这样各个模块的代码划分较为清晰，适于分布开发，有利于复用。一个模块由一个.c源文件和一个.h头文件组合而成。模块对外提供的服务由.c文件中的函数实现，外部通过调用这些函数来使用模块的服务。.h头文件中描述了接口信息，其中包含那些可以被外部访问的函数和变量的声明。模块的内部数据和内部函数应对外隐藏，不能被外部访问。

1) 声明和定义

声明是将变量名或函数名的标识符相关信息告诉编译器，使编译器“认识”该标识符。

例如：int i;　　//是告诉编译器，名字i表示数据类型为int的变量。

float func(float a);　//告诉编译器，func是一个返回值为float型的函数，并且此函数有一个float型的参数。

定义是变量或函数的实现，例如：

```
/----------------------------------------函数定义举例----------------------------------------
int sum(int a，int b)  //这是函数的定义
{
    return a+b;
}
/-------------------------------------------------------------------------------------------
```

对于变量来说，定义要为其分配内存空间。很多时候定义就是声明。例如: int a；我们可以说它是定义，也可以说它是声明。

编译器在调用函数和使用变量前，必须先获得函数和变量的声明，因为编译器需要知道函数的返回类型、形参类型和变量的类型。

2) 模块实例

例如一个液晶显示屏(LCD)驱动模块，以实现字符的显示，命名为: lcd_display.c，该.c文件大体可以写成：

```
/-----------------------------------------lcd_display.c-------------------------------------
#include lcd_display.h                    //包含头文件

static void DelayMs (long ms);     //内部（局部）函数的声明
uint32_t value;                    //全局变量定义，在模块中应尽可能的不使用全局变量
```

```
static void DelayMs (long ms)      //只供本模块调用的内部函数，所以用 static 关键字修饰
{
     ……
}

LcdCharDispaly (unsigned char , unsigned char , char *)        //供外部调用的字符显示函数
{
     ……
}
/-------------------------------------------------------------------------------------------
```

这里只列出了模块中的DelayMs ()和LcdCharDispaly ()两个函数。DelayMs () 只在本模块内使用，LcdCharDispaly ()是为外部提供的服务。源文件中包含了模块的头文件、内部函数的声明和所有函数的定义。

将供外部调用的函数和变量的声明放到模块的头文件中，这样其他所有源文件要调用这些函数，只需包含该头文件，如： #include “lcd_display.h”，就能获得声明。供外部调用的函数及变量需在声明中冠以extern关键字，extern关键字告诉编译器，函数和变量是在其他文件中定义的，对于变量则不用分配空间。

LcdCharDispaly ()会被外部调用，那么头文件中就必须将这个函数声明为外部函数（使用extern关键字修饰），外部变量value的声明类似。该.h文件可以写成：

```
/----------------------------------------lcd_display.h-------------------------------------
#ifndef __LCD_DISPLAY_H__
#define __LCD_DISPLAY_H__
#include ……                    //包含必要的引用文件

extern unsigned char value;                        //全局变量定义，在模块中应尽可能的不使用全局变量
extern LcdCharDispaly (unsigned char , unsigned char , char *);   //供外部调用的字符显示函数

#endif   //__LCD_DISPLAY_H__
/-------------------------------------------------------------------------------------------
```

对于模块内部使用的函数和变量，其声明不需要也不应该出现在模块头文件中，否则可能造成误解。这些声明应放到源文件中，而且冠以static关键字，以防止外部对这些函数的调用，达成信息隐藏。如DelayMs ()函数只在本模块内使用，用static关键字修饰。

模块源文件也需要包含模块头文件，编译器需要验证头文件中的函数（变量）声明和源文件中的函数（变量）定义相匹配。

如果源文件多次包含同一个头文件，则可能导致编译错误。在模块头文件中的开头使用#ifndef和#endif将头文件的内容封闭起来，并定义一个宏__LCD_DISPLAY_H__。预处理器如果没有该宏定义，则包含头文件的内容；如果已经有了该宏定义，则证明已经包含了该头文件，不需要再次包含。

4.4 模块化设计示例

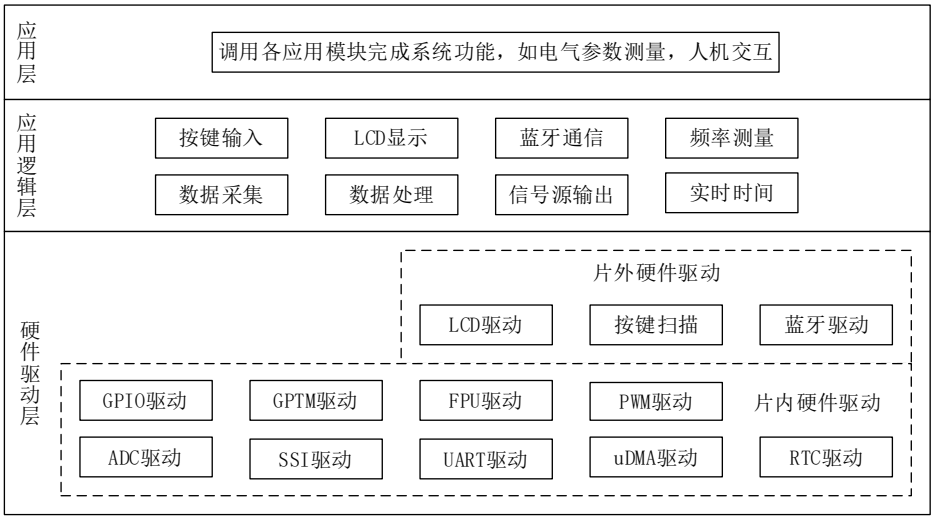

图 4-10 模块化设计

如图 4-10 所示，这是一个电气测量仪表的模块化设计示意图。主程序 man.c 完成系统的初始化，以及通过调用功能模块的服务来实现系统功能。应用逻辑层的功能模块针对系统的某个具体需求，负责实现专一功能。底层的硬件驱动为上层各模块提供支持。架构中的每一个模块都可以由一个单独的 C 语言源文件来实现。

5 编码规范

编程规范是众多程序员从长期的编程实践中凝练出来的经验和教训，能有效的减少编码错误，预防代码中潜在问题的出现。规范的编码能在程序设计的早期就杜绝许多错误，从而大大减少后期程序纠错的工作量。编码规范可以增强代码的可读性，让代码易于被团队其他成员理解，将提高合作开发的效率，促进代码复用，让代码的维护变得更容易。因此，编码规范对于程序员而言尤为重要。本文对编程规范中的排版、命名和注释做了简要介绍。

5.1 排版

对代码进行排版能增强可读性，使程序结构清晰，更容易理解。

1) 在程序的各个相对独立的程序块间加入空行，以便分辨程序的结构。
2) 合适的缩进能让程序的嵌套一目了然。程序块统一采用四个空格进行缩进，且注释应与所描述内容进行同样的缩进。以防止使用不同的编辑器阅读程序时，因 TAB 键所设置的空格数目不同而造成程序布局不整齐。
3) 函数代码的参数过长，分多行来书写。

```
/--------------------------------------------------------------------------------------------
if ((taskno < max_act_task_number)
       && (n7stat_stat_item_valid (stat_item)))  /*if 条件的注释*/
{
       ... // program code
```

```
}
/--------------------------------------------------------------------------------------------
```

4) 一行只写一条语句。

```
/--------------------------------------------------------------------------------------------
rect.length = 0;
rect.width = 0;
/--------------------------------------------------------------------------------------------
```

5) if、do、while、switch、for、case、default 等关键字独占一行，必须加上大括号{}，大括号独占一行。

```
/--------------------------------------------------------------------------------------------
if (NULL == pUserCR) //if 语句的注释
{
       return;
}
/--------------------------------------------------------------------------------------------
```

5.2 命名规则

命名应意义明确，能让人快速理解代码的含义，起到“见名知意”的效果。应使用英文单词或单词的常用缩写来命名，命名风格要始终一致。

1) 变量的命名

变量、参数的命名必须使用“名词”或者“形容词_名词”的语义结构，形容词、名词必须使用小写字母；

全局变量的名字必须使用“g_模块名_名词”或者“g_模块名_形容词_名词”的语义结构，其中“g”为全局变量的作用域标识；

静态变量的名字必须使用“s_名词”或者“s_形容词_名词”的语义结构，其中“s” 为静态变量的作用域标识；

在变量定义的语义结构中，每个形容词或名词都应使用有意义的单词，且字符个数应不少于2个；

禁止使用纯数字来替代形容词或名词；禁止出现除小写字母、下划线和数字以外的符号；

除循环变量以外，禁止使用诸如 ijkmnxyz 等单字符的名字对变量进行命名。

```
/--------------------------------------------------------------------------------------------
int count; //局部变量，有意义的名词
int g_switchmodule_count; //全局变量，g+模块名+名词
short new_value;  //局部变量，形容词+名词
short g_new_value; //全局变量，g+模块名+形容词+名词
static short s_new_value; //静态变量，s+形容词+名词
int *direct _shelf_id; //局部变量，形容词+形容词+名词
int  *g_switchmodule_direct_shelf_id; //全局变量，g+模块名+形容词+形容词+名词
```

```
/------------------------------------------------------------------------------------------
```

2) 大写宏定义、常量和枚举值

统一使用大写字母加下划线的方式来定义宏、常量和枚举值。

```
/------------------------------------------------------------------------------------------
#define _EXAMPLE_0_TEST_
#define _EXAMPLE_1_TEST_
/------------------------------------------------------------------------------------------
```

5.3 注释

注释对于程序的可读性和可维护性非常重要，有效注释量应在代码总行数的30％以上。

1) 程序体中注释的位置

注释应与其描述的代码相近，对代码行的注释应放在其上方相邻位置或右方（对单条语句的注释）相邻位置，不可放在下面，如放于上方则需与其上面的代码用空行隔开。

```
/------------------------------------------------------------------------------------------
void example_fun( void )
{
        /* 第一行代码的注释*/
        CodeBlock One
        /* 第二行代码的注释*/
        CodeBlock Two
}
/------------------------------------------------------------------------------------------
```

更多的编码规范见《微控制器原理及实践 C 语言编程规范.pdf》。

6 课堂实验

1) 阅读《微控制器原理及实践 C 语言编程规范.pdf》，找出下面代码中所有不符合编程规范之处。

```
/------------------------------------------------------------------------------------------
float b,c[10];
void abc(void)
{float zongfen = 0; int d;
for(d=0; d<10; d ++){
if([d]>0)
zongfen+ =c[b];
b=zongfen/10；
}}
/------------------------------------------------------------------------------------------
```

2) 参考 lab2，要求使用 systick 定时器中断编写一个前后台程序，实现 tiva 板上 3 色灯每 1s 点亮 0.2s 红灯，每 3s 点亮 0.2s 紫灯，每 6s 点亮 0.2s 白灯，注意编程规范。同时点亮红灯和蓝灯为紫灯，同时点亮红灯、蓝灯和黄灯为白灯。

```
/-------------------------------------------------------------------------------------------
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1,0x02);//点亮红灯
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1,0x00);//熄灭红灯
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2,0x04);//点亮蓝灯
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2,0x00);//熄灭蓝灯
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3,0x08);//点亮黄灯
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3,0x00);//熄灭黄灯
/-------------------------------------------------------------------------------------------
```

7 课后作业

1) Tiva_C 的默认堆栈空间有多大？下面的代码有什么错误？

```
/-------------------------------------------------------------------------------------------
char *dosomething(void)
{
     char i[1024];
     memset(i, 0, 1024);
     return i;
}
/-------------------------------------------------------------------------------------------
```

2) 请简述堆栈溢出的一般原理、造成的原因以及可能的后果。

8 预习习题

1) 阅读 SW-TM4C-DRL-UG-2.1.4.178.pdf 第 13 章 Floating-Point Unit (FPU)、TM4C LaunchPad Workshop 实验练习步骤指南.pdf 第 9 章 Floating-Point Unit，完成 workshop 实验 lab9。