都江堰操作系统

用户手册

版权所有，转载必须注明出处，注明：来自 www.djyos.com 字样。

第1章 开发工具安装和配置

详见《DJYOS 开发工具手册 V2.0》。

第2章 **Hello led**

我们都是跟随“HelloWorld”进入编程的世界，当第一次见到屏幕上显示“Hello World”时，感觉那就是全世界最美的字符。
但在深度嵌入式环境中，想看到“HelloWorld”可不是件简单的事，你懂的，初始化输出输出终端环境，可不是件简单的事，至少，你需要有 LCD，或者驱动串口吧。
在嵌入式板件中，一般都会有 LED 灯，点个灯，比输出“Hello World”要简单得多，这里就用跑马灯来演示 djyos 下第一个应用程序。
先讲讲下面 main.c 文件中程序开头 include 的几个头文件。
os.h，几乎所有的 DJYOS 应用程序，都需要包含这个头文件，因为这个文件里面包含了所有 OS 内核数据结构和函数声明。
Cpu_peri.h，这个文件在“djysrc\bsp\cpudrv\stm32\include”目录下，其中 stm32 是 CPU 名。DJYOS 的 BSP 中，所有 CPU 片内外设，都在 bsp\cpudrv 目录下，不同的 CPU 都有各自的目录。该文件包含了 CPU 外设的数据结构定义、变量声明等。凡是需要使用 CPU 片内外设的程序，都应该包含这个头文件。
“gpio_api.h”中包含了 gpio 操作函数，stm32f10x.h 和 LED.h 就不说了,大家都懂。

mian.c 的文件内容

```
#include "os.h"
#include "cpu_peri.h"
#include "gpio_api.h"
#include "stm32f10x.h"
#include "LED.h"

#define LED1_GPIOC9          GPIO_Pin_9
#define LED2_GPIOC8          GPIO_Pin_8
#define LED3_GPIOC4          GPIO_Pin_4
#define LED4_GPIOC6          GPIO_Pin_6

void djy_main(void)
{
    while(1)
    {
        GPIO_SettoLow(CN_GPIO_C, (1<<4)|(1<<6)|(1<<8));
        GPIO_SettoHigh(CN_GPIO_C, (1<<9));
        Djy_EventDelay(500*mS);

        GPIO_SettoLow(CN_GPIO_C, (1<<4)|(1<<6)|(1<<9));
        GPIO_SettoHigh(CN_GPIO_C, (1<<8));
```

```
            Djy_EventDelay(500*mS);

            GPIO_SettoLow(CN_GPIO_C, (1<<6)|(1<<9)|(1<<8));
            GPIO_SettoHigh(CN_GPIO_C, (1<<4));
            Djy_EventDelay(500*mS);

            GPIO_SettoLow(CN_GPIO_C, (1<<9)|(1<<4)|(1<<8));
            GPIO_SettoHigh(CN_GPIO_C, (1<<6));
            Djy_EventDelay(500*mS);
        }
}
```

djy_main 函数，是应用程序的入口函数，所有 djyos 发布版本都会带有这个函数。

第3章 裁剪与配置

组件裁减和配置，超级简单的说。

涉及到的文件：“工程目录\src\user\critical\critical.c”，“工程目录\\qh_1\src\user\module-trim.c”。

3.1 启动时的看门狗

如果配置了硬件看门狗，且工程比较大，导致加载和启动时间超出硬件看门狗的狗叫周期，就需要修改“工程目录\src\user\critical\critical.c”的 critical 函数，使能启动过程喂狗功能。

```
void critical(void)
{
    //如果需要在启动过程中喂狗,则必须在初始化看门狗之前,初始化定时器硬件
//      Timer_ModuleInit();         //todo:函数名不合适
    //初始化硬件看门狗,需要启动过程中喂狗的话,必须在此初始化.
//      WWDG_STM32Init(1000);
    // 启动实施中断+定时器喂狗机制,调用了这个函数后,Sys_ModuleInit 函数中调用
    // WdtHal_BootEnd 是必须的,否则可以注释掉.
//      WdtHal_BootStart(20);
}
```

从 example 中 copy 该函数，把注释掉的三个函数加上，并实现就行。

3.2 组件配置

找个 example，在“工程目录\\qh_1\src\user\module-trim.c”文件的 sys_ModuleInit 函数中，把不要的组件初始化函数注释掉即可。

3.3 运行参数配置

同样，在“工程目录\\qh_1\src\user\module-trim.c”文件中，有一组 const 变量，修改这组变量的值即可。修改后，操作系统是不需要重编译的，只需要重编译应用程序就可以了，闪电般快速（如果你的工程不太大的话）。

第4章 shell 终端

DJYOS 的 shell 是一种命令行式的 shell。它可以接受用户命令，然后调用系统相应的程序，实现用户与操作系统之间的交互。

4.1 配置 shell 终端

shell 作为用户与操作系统之间的一种通讯方式。使用 shell 之前，必须安装和配置好输入输出终端设备。例如，要设置“uart2”为 shell 终端，配置方法如下：
1. 在工程目录的 module-trim.c 文件中，把 gc_pStddevName 变量改为“uart2”。
2. 在 Sys_ModuleInit 函数中确保相应串口的初始化函数被调用，并且设置了合适的 Baud。
3. 在电脑端，打开超级终端，按照“uart2”的参数进行设置，设置成“115200，N，8，1”，无硬件流控模式即可。

4.2 使用 DJYSH 注意事项

4.2.1 输入命令限制

DJYSH 的输入命令和命令所带的参数，总长度不能超过 255 字节。
DJYSH 虽然支持总长度超过 255 字符的路径名，但由于命令行长度限制，在命令行下，无法操作超过 255 字节的路径名。

4.2.2 文件和目录不能含有空格

DJYFS 支持在文件名和目录名中包含空格，但由于 DJYSH 以空格作为断字符，故在 shell 中输入的文件名或者目录名不能包含空格符。

4.2.3 当前工作路径

si 和 dlsp 版本是单进程的，当前工作目录是一个全局概念，在 shell 中使用 cd 命令改变当前目录时，需要特别注意，在 shell 中改变当前路径，应用程序是不知道的，如果应用程序继续使用原来的“当前路径”操作文件系统，将出现不可预料的错误。

4.2.4 限制

有些命令有赖于操作系统配置，例如：
在 Sys_ModuleInit 函数中没有调用 Debug_InfoInit 函数的话，意味着内核调试信息模块被裁减掉，heap、event、stack 等命令将无法使用。
没有配置文件系统的话，跟文件有关的命令无法使用。
配置并使用不支持文件目录的简易文件系统的话，跟目录相关的命令如“cd”将无法使用。

4.2.5 函数调用

这是在 dlsp 和 mp 版本中才提供的功能。
shell 允许你调用系统中函数，包括操作系统 API、你自己写的任何函数。这个功能有一定的危险性，建议不要在在线运行的设备中使用，只在调试时打开这个功能。它的危险性来自于以下两个方面：
1. 被调用的函数是在 shell 事件的上下文中执行，shell 并不知道被调用的函数所需的栈空间，也就存在栈溢出的可能，栈溢出的危险性，我不说，大家都清楚。你请客，我买单，我不知道你点什么菜，怎知道要带多少钱？带多少都可能不够用。

2. 你既然能用 shell 调用任何函数，也就可以干任何事，干好事坏事，全凭你的良心。

4.3 基本 shell 命令

4.3.1 help:帮助

用法：
help [cmd]
参数：
cmd：具体的命令，比如 cd、dir 等。
说明：
显示 cmd（shell 命令）的帮助文档，主要包括命令的解释和用法说明。如果 cmd 为空，则表示显示所有命令的帮助文档。

4.3.2 speed：测试空 for 循环运行时间

用法：
speed
参数：
无。
说明：
测试空 for 循环运行时间。

4.3.3 event：打印处理中的事件

用法：
event
参数：
无。
说明：
打印系统中正在处理的事件。

4.3.4 evtt：打印已登记的事件类型

用法：
evtt
参数：
无。
说明：
打印系统中已经登记的事件类型。

4.3.5 heap：打印堆内存总体使用情况

用法：
heap
参数：
无
说明：
显示堆内存使用情况。

4.3.6 heap-spy：打印堆内存块分配情况

用法：
Heap-spy
参数：
无
说明：
显示堆内存中各块的分配情况。

4.3.7 stack：打印事件运行栈信息

用法：
stack

参数：
无。
说明：
显示系统中所有事件处理函数的栈的信息，并报告是否有栈溢出风险。

4.3.8 lock：打印所有锁信息

用法：
lock
参数：
无。
说明：
显示系统中所有信号量和互斥量的信息。

4.3.9 d：打印内存数据

用法：
d [address] [unit_num] [unit_bytes]
参数：
address：显示的起始地址，强制按 unit_bytes 对齐，如果用户输入了一个不对齐的地址，将被强制对齐。对齐方法：
unit_bytes = 1，不对齐。
unit_bytes = 2，address 的最低位将被强制清零。
unit_bytes = 4，address 的最低两位将被强制清零。
unit_bytes = 8，address 的最低三位将被强制清零。
unit_num：显示的单元数量。
unit_bytes：unit_bytes 的可选值有：1，2，4，8。
说明：
从地址 address 开始显示 unit_num 组的内存数据，以 16 进制格式显示。

4.3.10 f：写内存数据

用法：
f [address] [unit_num] [unit_bytes] [unit_data]
参数：
address：写入的起始地址，强制按 unit_bytes 对齐，如果用户输入了一个不对齐的地址，写入内存的数据将被强制对齐。对齐方法：
unit_bytes = 1，不对齐，unit_data 的低 8 位被写入。
unit_bytes = 2，address 的最低位将被强制清零，unit_data 的低 16 位被写入，不足补 0。
unit_bytes = 4，address 的最低两位将被强制清零，unit_data 的 32 位被写入，不足补 0。
unit_num：写入的单元数量。
unit_bytes：unit_bytes 的可选值有：1、2、4。

unit_data：被写入内存的数据。
说明：
从地址 address 开始写入 unit_num 组数据到内存，每组包含 unit_bytes 字节，每组填写的数据为 unit_data，并强制按 unit_bytes 字节对齐。

4.3.11 ver：打印系统版本信息

用法：
ver
参数：
无。
说明：
显示系统版本等信息。

4.3.12 date：打印、修改系统日期

用法：
date
参数：
无。
说明：
显示及修改系统日期，输入日期的格式与显示的相同。

4.3.13 time：打印、修改系统时间

用法：
time
参数：
无。
说明：
显示及修改系统时间，输入事件的格式与显示的相同。

4.3.14 rsc-tree：打印资源树

用法：
rsc-tree [TreeName]
参数：
TreeName：资源树的名称。
说明：
列出系统中的资源名称和层次关系，要求显示名称为 TreeName 的资源队列中资源的名称，若查找不到该资源名称，则显示"没找到 TreeName 资源树"，若 TreeName 为空，则列出系

统所有资源名称。

4.3.15 rsc-son：打印资源分支

用法：
rsc-son [RscName]
参数：
RscName：父资源的名称。
说明：
按参数要求显示名称为 RscName 的父资源中子资源列表，若查找不到父该资源名称，则显示"没找到 RscName 资源"，若 RscName 为空，则列出系统所有子资源名称。

4.3.16 uninstall-cmd：卸载命令

用法：
uninstall-cmd[cmd-name]
参数：
cmd-name：被删除的 shell 命令名。
说明：
通过命令名删除 shell 命令。

4.4 扩展 shell 命令

4.4.1 增加 shell 命令

DJYOS 的 shell 提供开放的接口，用户可以自如添加 shell 命令。
用户通过调用 Sh_InstallCmd 将新 shell 命令添加到命令表中，新添加的命令不能出现重名，否则将报错。添加新 shell 命令过程如 代码 4-1 所示，该代码在 exp_shell.c 文件中。
代码 4-1 扩展 shell 命令

```
struct tagShellCmdTab    gtExpShellCmdTab[] =
{
    {
        "expi",
        Exp_ShellBrowseAssignedRecord,
        "查看指定异常条目",
        NULL
    },
    {
        "expn",
        Exp_ShellBrowseRecordNum,
        "查看异常条目数量",
        NULL
```

```
    },
    {
        "expc",
        Exp_ShellRecordClear,
        "清除所有异常条目",
        NULL
    }
};
#define CN_EXPSHELL_NUM    ((sizeof(gtExpShellCmdTab))/(sizeof(struct tagShellCmdTab)))
static struct tagShellCmdRsc sgExpShellRsc[CN_EXPSHELL_NUM];

bool_t Exp_ShellInit()
{
    bool_t result;
    if(Sh_InstallCmd(gtExpShellCmdTab,sgExpShellRsc,CN_EXPSHELL_NUM))
    {
        printk("成功添加异常组件的 shell 命令\n\r");
        result = true;
    }
    else
    {
        printk("添加异常组件的 shell 命令失败!S\n\r");
        result = false;
    }
    return result;
}
```

执行这段代码后，shell 中就增加了 expi、expn、expc 三个命令，这三个命令的执行函数分别是 Exp_ShellBrowseAssignedRecord 、 Exp_ShellBrowseRecordNum 和 Exp_ShellRecordClear，"查看指定异常条目"等三个字符串将出现在 help 命令中。

4.4.2 文件系统 shell 命令

4.4.2.1 cd/chdir：显示和更改当前工作目录

用法：
cd/chdir [分区:][路径名]
例如：cd C:\folder\fold
参数：
略。
说明：
显示或更改当前工作目录。

该命令使用方式比较多，如下所列：

cd/chdir [分区:] [路径名] 显示当前目录的名称或改变当前目录
举例 1：

```
sys:\>cd foldert
sys:\foldert\>
```

cd/chdir [](空字符) 显示当前目录名称
举例 2：

```
sys:\foldert\>cd
sys:\foldert\
```

cd/chdir [\] 更换到根目录
举例 3：

```
sys:\foldert\>cd \
sys:\>
```

cd/chdir [..] 更换到上一级目录
举例 4：

```
sys:\foldert\>cd ..
sys:\>
```

cd/chdir [分区:] 更换到指定分区的根目录（相当于在 dos 下直接输入驱动器）
举例 5：

```
sys:\>cd E:\
E:\>
```

或直接输入“分区：”，也可以更换到指定分区的根目录
举例 6：

```
E:\>C:\
C:\>
```

4.4.2.2 dir：显示目录内容

用法：
dir [分区:] [路径名]
例如：dir C:\folder\file
参数：
略。
说明：
显示指定目录下的文件和子目录列表

该命令使用方式比较多，如下所列：
dir {[分区:] [路径名 1]} {[分区:] [路径名 2]}
显示指定目录下的文件和子目录列表

dir {[分区：] [含有通配符路径 1]} {[分区：] [含有通配符路径 2]}
显示指定目录下匹配的文件和子目录列表

4.4.2.3 md 或 mkdir：创建目录

用法：
md [分区:] [路径名]
例如：md C:\folder\file
参数：
略。
说明：
创建一个或者多个目录。

md/mkdir　{[分区:] [路径名 1]} {[分区:] [路径名 2]}

4.4.2.4 rd 或 rmdir：删除目录

用法：
rd [分区:] [路径名]
参数：
略。
说明：
删除一个或多个目录。

rd/rmdir　{[分区：] [路径名 1]} {[分区：] [路径名 2]}
举例 1：

```
sys:\>rd foldert b2 b3
目录 foldert 不是空！
sys:\>dir
sys:\ 的目录

2012/07/12 12:10                282016B gb2312_1616
2012/07/12 12:10                8192B tree.bmp
2012/07/12 12:10                2048B pear.bmp
2012/07/12 12:10                105600B logo.bmp
2012/07/12 13:55                16B touch_init.dat
2012/07/12 14:00 <DIR>          0B foldert
5 个文件  397872 字节
1 个目录  15843328 可用字节
```

rd/rmdir [分区：] 含有通配符路径 [分区：] 含有通配符路径... ...
举例 2：

```
sys:\>rd a*
sys:\>dir
sys:\ 的目录

2012/07/12 12:10                282016B gb2312_1616
2012/07/12 12:10                8192B tree.bmp
2012/07/12 12:10                2048B pear.bmp
2012/07/12 12:10                105600B logo.bmp
2012/07/12 13:55                16B touch_init.dat
2012/07/12 14:00 <DIR>          0B foldert
2012/07/13 00:50 <DIR>          0B b3
2012/07/13 00:50 <DIR>          0B b2
5 个文件  397872 字节
3 个目录  15843328 可用字节
```

4.4.2.5 del：删除文件

用法：
del [分区：] [路径名]
参数：
略。
说明：
删除一个或多个文件。

del {[分区：] [路径名 1]} {[分区：] [路径名 2]}... ...
del {[分区：] [含有通配符路径 1]} {[分区：] [含有通配符路径 2]}... ...

4.4.2.6 ren 或 rename：重命名

用法：
ren/rename [分区:][路径][文件名 1] [文件名 2]
参数：
略。
说明：
修改指定文件名或者文件夹，文件名 1 为旧文件名，文件名 2 为新文件名。

4.4.2.7 copy：拷贝

用法：
copy [分区 1:] [路径名 1] [子目录名 1] [分区 2:] [路径名 2] [子目录名 2]
参数：
略。
说明：

将至少一个文件复制到另外一个位置。

4.4.2.8 move：移动文件

用法：
move [分区 1:] [路径名 1] [子目录名 1]　[分区 2:] [路径名 2] [子目录名 2]
参数：
略。
说明：
将至少一个文件移动到另外一个位置。

4.4.2.9 create-PTT：创建分区

用法：
create-dbx[PTT_size] [PTT_name]
参数：
dbx_size：分区大小，字节为单位。
dbx_name：分区名。
说明：
创建分区名为 PTT_name、大小为 PTT_size 的分区。Djyfs 的分区，有点类似 Windows 的磁盘，只是分区允许字符串，而磁盘只允许单字符。

4.4.2.10 format：格式化分区

用法：
format[PTT_name]
参数：
PTT_name：文件系统分区名。
说明：
格式化分区，执行 create-PTT 创建分区后，须格式化才能正常访问。

4.4.2.11 chkptt：打印分区信息

用法：
chkptt 分区
参数：
略。
说明：
显示分区空间的参数。

第5章 事件、事件调度和线程调度

5.1 事件

计算机处理的是现实世界中的具体任务，有因才有果，现实生活中的任务不会无缘无故地产生，人们做某一件事肯定是因为发生了某种事件使其需要去做这件事情，这就是事件。计算机中的事件与现实生活中的事件是一致的，CPU 不会无缘无故地执行某一段代码，就算是一段包含在一个 if 语句里的代码被执行，肯定是因为发生了使该条件成立的事件。人走到沙发前是一个事件，智能沙发上的计算机发现这个事件后然后处理这个事件，处理结果是执行调整坐垫到合适位置的操作；人转身面对电视机是一个事件，智能电视机里的计算机发现这个事件然后处理这个事件，处理结果是执行打开电视机的操作；人躺在床上并闭上眼睛，智能家居的计算机发现这个事件然后处理这个事件，处理的结果是执行关灯的操作。以上所述的事件，就是 DJYOS 操作系统中“事件”的原型。所有这些原型中，都有一个“发现”（或称“检测”）事件和执行一定操作以处理事件的过程，现实系统中，这两个过程可能非常复杂，甚至处于两个不同的学科，其软件实现模块可能会由两个不同专业方向的程序员编写。DJYOS 软件模型是：由一个软件模块专门用于监测人的行为，另外一些模块执行开关灯、开关电视机、调整沙发坐垫的操作。检测模块发现人靠近沙发的事件后，不是去调整沙发坐垫，而是把“事件”报告给操作系统了事。操作系统收到该事件后，先把该事件记录在调度队列中，再依据调度算法，当决定要处理该事件时，就分配或创建用于处理该类型事件的线程，并启动该线程，再由这个线程去执行调整沙发坐垫的操作。这样，就使“检测”和“执行”相互独立开来。进程、线程之类的东西只是操作系统内部的秘密，线程作为一个资源，是创建新资源还是使用现有资源来处理事件，完全由操作系统自动完成，应用程序的程序员不知道也不需要关心这些。

DJYOS 下程序运行的过程，就是新事件不断发生，然后被处理的过程。在此过程中，操作系统组织、创建、分配线程、进程以及其他资源去满足处理事件所需。每弹出一条事件，DJYOS 操作系统就为它分配一个事件控制块，事件处理完毕后收回事件控制块。未处理完毕的事件就会堆积在队列中，操作系统对队列的容量有一定的限制，当队列中事件数量达到上限时，操作系统将拒绝接受新事件。Module-trim.c 文件中的 gc_u32CfgEventLimit 常量用于确定队列容量，用户可以修改，但不能在程序运行过程中动态改变，最多允许 16384 个事件。

5.2 事件类型

程序运行期间会发生各种各样的事件，不同种类的事件由各自的线程处理，需要用事件类型去加以区分。DJYOS 操作系统为每一类事件分配一个唯一的事件类型 ID 号，并为每个事件类型分配一个事件类型控制块（struct event_type）。事件类型控制块是静态分配的，其数量须按照内存量和应用程序实际需求合理设定，由工程目录下的 Module-trim.c 文件中的 gc_u32CfgEvttLimit 常量确定，用户可以修改，但不能在程序运行过程中动态改变，最多允许 16384 个事件类型。

事件类型必须调用 Djy_EvttRegist 函数登记才能使用，登记时要为该事件类型指定一个事件处理函数，该函数将成为线程的入口函数，还要设定该类型的默认优先级，并且告诉操作系统该函数运行需要多少栈空间。在某些情况下，操作系统会为部分类型事件至少保留一个线

程（参见 0 节），当有该类型的事件发生(调用 Djy_EventPop)时，操作系统可能会自动创建线程处理该事件，也可能会指挥现有线程（或创建线程）去处理该事件。

5.2.1 独立型和关联型事件

如果某类型事件重叠发生（即事件未处理完成，又发生相同类型事件），产生的多条事件可以用多个线程并行处理的，称为独立型事件。典型的独立型事件是 web 服务，当 web 服务器收到多个客户端的服务请求后，这些请求一般是独立的，服务器会创建多个线程独立处理多个请求。独立型事件用 EN_INDEPENDENCE 表示，该类型的每条事件都需要单独处理，每次弹出独立型事件，都会在事件队列中添加一条事件，事件反复发生而又来不及处理时，事件队列中将积压多条同一类型的事件（受 vpus_limit 限制）。当然，即使是独立型事件类型，如果在调用 Djy_EventPop 函数时使用的参数是事件 ID，也不会添加新事件。
反之，如果重叠事件互相关联，必须在一个线程中顺序处理，称之为关联事件。该类型事件一般表示的是物理世界的一种状态，若此类型的事件重复发生，它也只代表系统处于某种状态，不是每次发生的事件都需要单独处理。关联型事件用 EN_CORRELATIVE 标记，这种事件无论重复发生多少次，事件队列中都只会保留一条事件。这是一种很重要的事件，因为现实世界中有太多的关联型事件，试举数例如下：
1. 快件投递中，当客户有快件需要投递，就会给快递公司电话，同一个地址，无论重复多少次电话，只需派收件员上门一次把积累的快件全部取走就可以了。
2. 在 LCD 面板显示软件中，在内存中设计了一个显存镜像，应用程序修改显示内容时，修改的是镜像显存，然后发出“显示刷新”类型事件，处理该事件的过程就是把镜像显存中的图像搬到物理显示器上。这是一个典型的关联型事件，无论应用程序修改了多少次镜像显存，都表示“显存被修改”这一物理状态，处理一次“显示刷新”事件将把历史上积累的事件完全清理。
3. 串口通信软件中，缓冲区接收到数据，会发出“缓冲区有数据需处理”类型的事件，无论该类型事件重复发出多少次，都表示这样一个状态，事件处理时只要把缓冲区的所有数据取走，就可以了。
独立型事件的例子也很多，例如：
1. 在百货商店，每进来一个顾客算发生了一个“顾客来了”类型的事件，由于每个顾客都是独特的，所以必须单独服务。
2. 同样是 LCD 面板显示软件中，当用户需要绘制时，就会发出“屏幕绘制”事件，因为每次绘制的内容都可能不同，所以每条事件都必须单独处理。
3. 通信软件中，应用程序需要发送数据，就把数据准备好，弹出“发送数据”类型的事件，并把数据缓冲区作为事件参数，由于每次事件的数据缓冲区都是独立的，不能把多条事件统一处理，而是每条事件都要单独处理。

5.3 线程

DJYOS 以事件为调度单元，理论上，操作系统可以用任何方法处理事件，只要能够调用事件处理入口函数就可以了，当前版本的 DJYOS 操作系统使用的方案是，创建一个线程执行事件处理入口函数来实现事件处理。注意，线程是操作系统自行创建的、用于执行用户提供的事件处理入口函数的手段，对程序设计者是不可见的，这就隐含了一个事实：操作系统还可以选择其他方案代替线程方案，遗憾的是，笔者至今也没有想到可以替代线程的可行方案。

要处理事件，操作系统就要为该事件创建线程，用户在登记事件类型时必须告诉操作系统执行该入口函数需要多少栈空间。在 DJYOS 系统中，线程是处理事件的执行者，也作为事件的资源而存在——完成该事件需要许多资源，线程是诸多资源之一。

事件、事件类型与线程的关系

套用面向对象的方法，事件类型相当于 C++的类，登记事件类型相当于声明一个类数据类型；事件相当于对象，弹出事件相当于定义对象，同时做一些构造函数的工作；事件处理完成相当于撤销一个对象，同时做一些类似析构函数的工作。

DJYOS 中应用程序的运行过程，就是不断地弹出新事件和处理事件的过程。每个事件都必须属于已经登记的事件类型，相同类型的事件使用相同的线程进行处理。

DJYOS 操作系统中，线程作为事件的资源而存在，而该事件就是线程的拥有者，因此，任何线程，都不能无缘无故地出生、存在和死亡，它必定与某一类型的某一条事件联系在一起。线程随事件的需要而生，随事件完成而消亡。线程无需登记，也无需有用户建立和启动，它的创建、启动和删除都是由操作系统自动完成的。这与传统操作系统不一样，传统操作系统可以由用户任意创建线程，创建一个毫无意义的线程是允许的。DJYOS 系统的调度依据是一个就绪事件队列和若干个同步事件队列，而不是线程队列（有的操作系统也称其为任务队列），DJYOS 中根本就没有线程队列。DJYOS 的调度针对事件而不是线程，创建一个线程是因为它的拥有者需要处理，线程被切入是因为该线程的拥有者需要被切入，线程被切离是因为它的拥有者被挂起。把线程作为事件的资源的积极意义在于，当某类型的事件连续发生，操作系统将调集更多的资源，为其创建多个线程来处理该事件，如果在多处理机（多核）系统中，把这些线程分配给不同的处理器，处理器本身也就成为一种资源，将极大地方便多处理器系统的管理。而传统的编程方法中，程序员创建若干线程待机，每个线程对应一种或数种事件，待相应的事件发生后，唤醒线程予以处理，这种方式在管理多处理器方面要复杂得多。因此，DJYOS 在多处理器系统中，有先天的优越性。

线程的属性必须与事件类型对应，相同类型的事件使用相同的线程处理，不同类型的事件使用不同的线程处理。用户登记一个事件类型时，必须传入操作系统创建用于处理该类型事件的线程所需要的两个关键参数：事件处理入口函数和该函数需要的栈空间。当某类型的事件发生后，操作系统就会在适当的时候创建线程（或分配存在的线程）执行该类型对应的事件处理函数。事件处理完成之后，操作系统会自动回收该线程所占用的资源，必要时还会删除线程。

每一条事件对应一个线程，如果有多条同一类型的事件需要处理，操作系统会创建多个相同的线程同时处理多个相同的事件。这可以更合理地使用计算机资源。虽然在单处理器的情况下，建立多个线程并不会比单个线程长期霸占处理器更充分利用处理器，但可以产生多个相同类型事件并行处理的效果，例如同时绘制多个窗口，特别是，在多处理机（或多核）系统中，可以把频繁发生的同一类型事件分配到不同的处理器上。而传统操作系统下，线程是由程序员创建的，如果程序员只为某项工作创建了一个线程，则该工作再繁忙也只有一个线程为它工作，该线程处理的多项任务只能串行执行。但是这样反复创建线程可能导致资源枯竭，比如处理某事件时需要使用串口，而串口又被其他线程占用，在串口被占用期间发生该事件，操作系统就会再次为其创建线程，该线程开始执行后会因串口资源繁忙而进入阻塞状态，如果事件反复发生，操作系统就会反复为其创建线程，直到消耗完所有内存，造成内存枯竭。为了防止发生资源枯竭事故，在事件类型控制块中提供了 vpus_limit 成员，表示该类型事件可以同时建立线程的个数。

5.4 线程调度和事件调度

DJYOS 系统中，调度是以事件为依据，线程是事件的资源，线程本身是没有优先级的，但上述理论依然适用，只要把其中的“线程”两字改为“事件”就可以了。
在实时系统中，事件的优先级是能否实现实时指标的关键，现实中，大多数事件会继承事件类型的默认优先级，因此确定每一类事件的默认优先级是系统设计的重中之重。
基于事件进行调度，这是 DJYOS 与传统操作系统最大的区别。图 5-1 是从传统操作系统抽象出来的，无论是简单的 OS 还是复杂 OS，其调度都是基于线程的。图 5-1 中，无论是初始化创建线程，还是线程执行过程中创建线程，都是在创建线程的时候分配线程所需要的资源，栈是其中的必备件。图 5-2 中，弹出事件时，除非新事件的优先级足够高，需要立即处理，否则弹出时不会分配或创建线程，直到该事件应该被处理的时候才创建或分配线程。

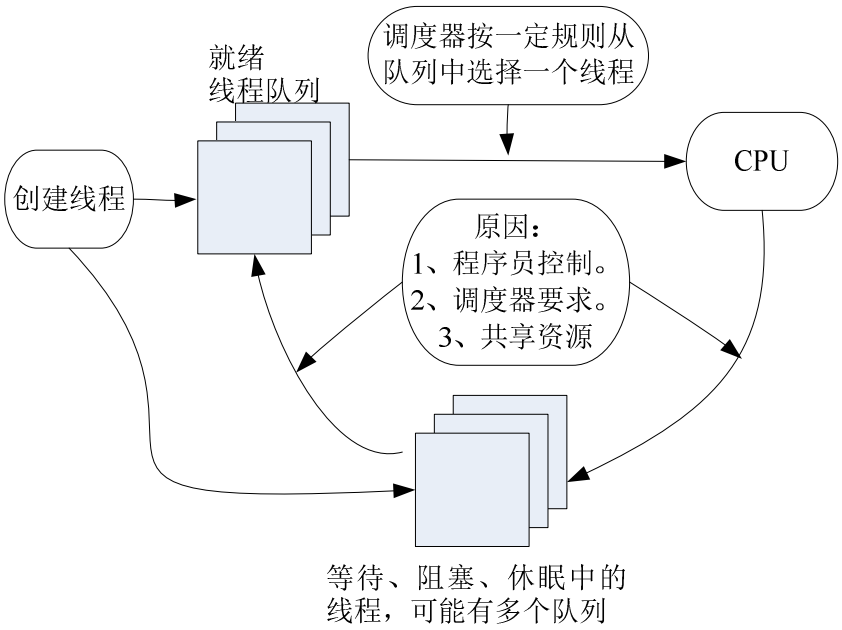


图 5-1 传统操作系统调度示意图

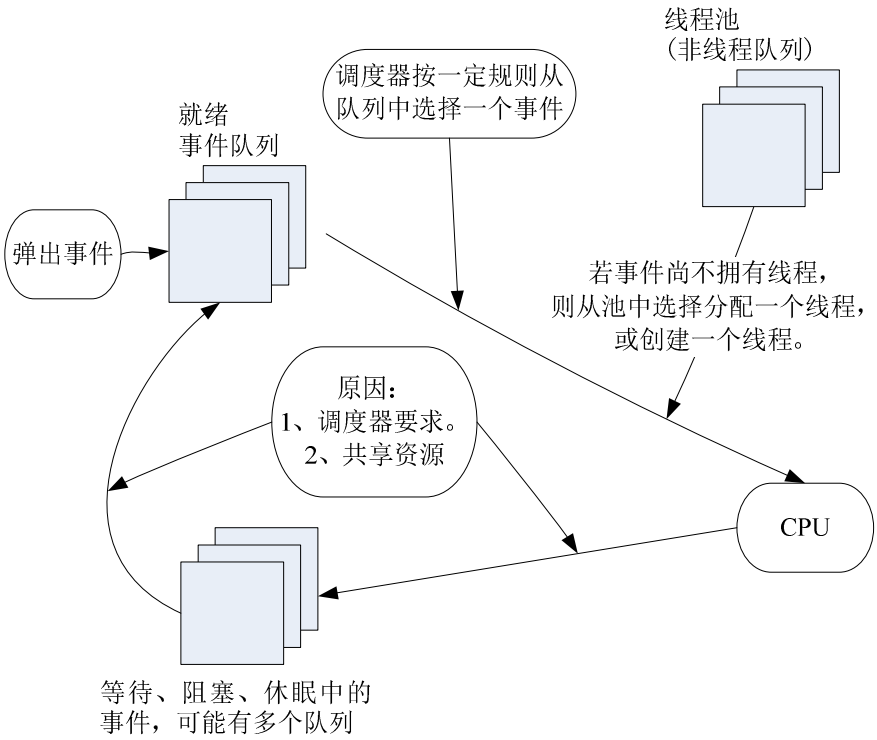


图 5-2 事件调度示意图

看起来，除了不能从中断处理函数中创建线程外，两图有很大的相似性，其实不然，传统 OS 下编程时，不应该频繁地在线程执行过程中创建新线程的，为什么呢？为了创建线程而导致阻塞，而被阻塞的却是当前线程。如果新线程的优先级低于当前线程，那当前线程就冤了，因为一个低优先级的线程，而导致高优先级线程阻塞，这在 RTOS 中是不允许的，从系统方案设计角度，也是不合理的；即使不被阻塞，高优先级线程执行过程中，花大量时间用于创建低优先级的线程，也是不合理的。所以，传统 OS 下编程时，所有线程总是在初始化阶段创建，极少在运行过程中创建。而 DJYOS 则没有这个问题，如果新弹出的事件优先级不够高，根本就不会为他创建线程。这使得 DJYOS 能够更加优化配置计算机的资源，假设有一个产品，有 uart 串口，在传统线程模式下， uart 通信线程是在初始化时创建的，即使 uart 通信线没有连接，uart 线程也必定占用内存资源。如果 uart 通信非常频繁地发生，也只能由初始化时创建的那个线程一点一点来处理，即使你有多个 cpu 核，其他 cpu 核也只能干着急。那不能创建多个线程吗？可以，线程池技术就是这样干的，究竟创建多少线程才合适，是一个非常令人头疼的问题。在传统 OS 下，线程池技术可是一门专门的学科哦。而 DJYOS 的事件调度呢？如果 uart 通信线上没有数据，则根本不会弹出该事件，也就不会占用任何资源。如果 uart 口频繁通信，就会频繁弹出事件，若计算机有多个 cpu 核，则自然而然地会把事件分散到不同的 cpu 核上，程序员根本不知道线程为何物，就能进行优化的多核编程。

大家都知道 vc、cbuilder 下编程吧，在桌面上放上一个按钮，然后为按钮点击事件编写处理函数，当用户点击该事件时，处理函数就被执行。这就是事件触发式编程，这是怎么实现的呢？原来，有一个线程一直在后台候着，等待 windows 发出的消息，一旦收到点击消息，线程便被唤醒执行。我们可以看到，无论该按钮是否被点击，甚至一辈子都不点击，该线程依然要占用系统资源。而 DJYOS 的方法则不同，只要该事件不发生，则不会占用任何系统资源。VC 为什么要实现事件触发式编程呢？是因为现实需要，与传统操作系统需要像 VC 这样的工具支持才能实现事件触发式编程相比，DJYOS 只需要文本编辑器就可以实现。VC 这样的工具只能用于较大的系统中，而 DJYOS 却可以用在单片机中！

传统的基于线程（进程）的调度模式下，操作系统只知道哪个线程（进程）正在占有 CPU，却不能知道该线程（进程）在干什么，调度器也只能针对线程（进程）分配 CPU，不能针对计算机所处理的具体事务分配 CPU。因此，传统操作系统下，程序员必须熟练掌握有关线程（进程）的知识，必须自己为程序需要处理的事务创建线程（进程），清楚在哪些状况下有哪些线程（进程）正在运行。而事件调度则不同，用户可能根本不知道线程（进程）的存在，以及谁正在运行，谁正在等待，实际上，程序员根本不需要关心这些。DJYOS 系统中，程序员只知道哪个事件正在被处理，哪些事件已经处理完成，哪些事件正在队列中等待处理。定义一个一个的事件类型并登记到系统中，为每类事件编写处理函数，便是编程的全部工作。当相同类型的多条事件，具有不同的优先级时，在传统操作系统下，要么为每一种可能的优先级建立一个线程（进程）来实现，要么在执行中动态改变线程（进程）的优先级，不管哪种方式，程序员都需要花费大量的时间和精力，以确保事件按正确的优先级得到处理。而 DJYOS 不同，它先天就是以事件优先级作为调度的基础，只要在产生事件时，直接在事件中做优先级标志就可以了。例如一个串口通信程序，中断函数负责底层接收，当接收到完整数据包后，就发给上层应用程序处理，上层应用程序处理接收到的所有数据包，而数据包中有一个命令字域，不同的命令要求的优先级不同。在传统操作系统下，要么创建一个 comm_app 线程，在中断函数中把数据传送给该线程的同时根据命令字改变 comm_app 线程的优先级，这种在中断函数中改变线程优先级的做法，在许多操作系统中是不允许的；另一种方法是，为命令字对应的每一种可能的优先级，均创建一个相同的线程，这些线程除优先级不同外，其它部分完全相同，当中断函数接收到完整数据包时，就根据数据包中的命令字，发消息给相应优先级的线程。这种方法虽然可行，但是会消耗很多 CPU 资源，且难于在编码阶段穷举所有可能的优先级，一旦命令字发生变化，很可能就需要修改软件源码。而 DJYOS 系统不一样，程序员只需要定义一个事件类型，并为之编写事件处理函数 proc_uart()，当中断函数接收到完整数据包，弹出事件时直接以命令字对应的优先级作为参数就可以，DJYOS 的调度系统自动会根据事件优先级域进行调度。

另外，嵌入式系统多是应激系统，应激系统的主要任务是对外界的事件做出正确且及时的反应，从这点看，程序员根本就不用知道进程和线程这些东西，为处理外界事件而建立线程（进程）实际上是不得已而为之，传统操作系统下，你必须做这些工作，你的系统才能正确地为你做些事情。基于事件的调度非常适合这种应激系统，被调度的目标就是反映外部刺激的事件，而不是处理这些外部刺激的线程，符合人们的习惯性思维。即使是非应激系统，或者是非实时系统，基于事件的调度仍然有其优势，“有事就做，没事就坐”是人们最为习惯的思维方式，以事件为调度单位的调度策略显然符合这种思维方式，而与人们习惯性思维相同的调度方式，又是避免人为错误，减少软件 bug 的有效方法。

现代计算机已经进入“ubiquitous/Pervasive Computing”时代，即普适计算。在普适计算时代，触手可及的计算产品里面也包含着触手可及的计算机程序，这些程序由大量的嵌入式程序员编写，是的，十年前的硬件工程师可以不懂软件，软件工程师可以不懂硬件，而今天的嵌入式技术，除了在一些很特殊的方向如射频设计，已经没有纯粹意义上的软件工程师和硬件工程师了。让这些队伍迅速壮大的、软硬兼顾的“普适计算工程师”们去掌握晦涩难懂的进程和线程技术并灵活应用，恐怕要花费不少的人才培养成本，而使用传统的操作系统去开发嵌入式产品，不理解这些复杂的概念根本就寸步难行，而人才的匮乏又将限制嵌入式产业的发展。DJYOS 操作系统不要求程序员操纵线程和进程，程序员只需把需要计算机处理的任务划分为一个个事件类型，并为各种不同类型的事件编写独立的事件处理函数，然后把它登记到系统中就可以了。当事件发生时，发现（检测到）该事件的程序员只要告诉操作系统“某类型事件发生了”，操作系统自动地创建或唤醒合适的线程去处理该事件，而无须程序员亲自创建或者唤醒相应的线程。当然，“普适计算工程师”即使是在 DJYOS 系统下编程，

深入理解线程和进程技术，对开发工作也是很有帮助的。

5.5 典型情景编程对比

DJYOS 与传统操作系统在调度方式上的差异，必然导致在其在编程模型上的差异。传统方式下，程序员编写函数以处理需要计算机完成的任务，然后建立线程来运行这些函数，创建线程、启动线程、线程同步、线程暂停、线程休眠、线程终止、线程删除等操作，均由程序员亲自完成。在 DJYOS 下，认为计算机执行某一段程序，必定是发生了使计算机执行这段程序的事件，计算机的所有操作，均由事件引发。程序员所需要做的工作是，以适当的粒度定义事件类型，为各类型事件编写处理函数，并且在需要的时候弹出事件，在必要的时候调用操作系统提供的事件同步功能。由此而导致了两种截然不同的编程模式，下面，我们通过不同场景的对照，来直观了解一下这两种编程方式。

5.5.1 场景 1

一个网络通信端口收发模块，监视网络端口状态以及收发数据流。
这种模块一般是系统启动后即开始运行，只要通信口不关闭，就需要持续接收数据，直至关机，在整个运行期需要不间断地监视网络端口，并做出相应的响应。

5.5.1.1 传统操作系统

```
Int   thread_comm(int para)
{
    无限循环，执行通信监视业务;
}
int   main(void)
{
    创建线程，指定 thread_comm 为线程入口函数；
    启动线程；
}
```

5.5.1.2 DJYOS

```
int   event_comm(int para)
{
    无限循环，执行通信监视业务;
}
int   main(void)
{
    登记 evtt_comm 事件类型，指定 event_comm 为事件处理函数；
```

```
    弹出 evtt_comm 类型事件；
}
```

5.5.1.3 分析

分析上面两个程序模型，可以看到，这种场景下，两种编程方式看似没有区别。但实际上是有区别的，因为事件是直观的，与人类思维模式比较接近；线程是抽象的，与计算机的执行过程接近。越接近人类自然思维，就越容易学习和掌握，编程就越不容易出错。这种区别，从这个简单的例子中不容易体现出来，但如果软件规模比较大、逻辑复杂的系统，这种差别就显示出来了。

现代计算机已经进入“ubiquitous/Pervasive Computing”时代，即普适计算。触手可及的计算产品里面也包含着触手可及的计算机程序，这些程序由大量的嵌入式程序员编写。设计这些产品需要大量的软件工程师，这使得嵌入式程序员的队伍迅速增大，新增的程序员，可能来自各行各业，他们原来在各自的行业中，可能都是非常了不起的专家，比如化工专家、医疗专家等。这些不同领域的专家，却未必是计算机领域的专家，让他们去掌握晦涩难懂的线程技术并灵活应用，恐怕要花费不少的人才培养成本，而使用传统的操作系统开发嵌入式产品，不理解这些复杂的概念根本就寸步难行。DJYOS 操作系统不要求程序员操纵线程和进程，程序员只需把需要计算机处理的任务划分为一个个事件类型，并为各种不同类型的事件编写独立的事件处理函数，并且把它登记到系统中就可以了。当事件发生时，发现（检测到）该事件的模块只要告诉操作系统“某类型事件发生了”，不需要管什么线程。从这个角度，DJYOS 降低了程序员培训要求，客观地为企业节约了人力资源费用。

5.5.2 场景 2

如场景 1 的通信口监视模块，通信口接收来自外部的请求，把接收到的请求交由服务器处理。为了确保通信口不丢数据包，通信口监视模块的优先级应该比服务模块高。

5.5.2.1 传统操作系统

5.5.2.1.1 方案 1

```
int   thread_comm(int para)
{
    while(1)
    {
        if（收到请求）
            唤醒 thread_server 线程  or  释放信号量；
        其他代码；
    }
}
int   thread_server(int para)
```

```
{
    while(1)
    {
        提供收到的请求对应的服务；
        线程暂停 or 请求信号量；
    }
}
int   main(void)
{
    创建 thread_comm 线程；
    启动 thread_comm 线程；
    创建 thread_server 线程；
    启动 thread_server 线程；
}
```

从上述代码中我们可以看到，thread_server 线程无论如何，都占据内存的，即使通信线没有插上，一辈子都收不到请求，也是这样。那么，我们有改进方法吗？能不能收到请求后再创建线程，像方案 2 的代码这样。

5.5.2.1.2 方案 2

```
    int  thread_comm(int para)
    {
        while(1)
        {
            if（收到请求）
            {
                if(thread_server 线程未创建）
                {
                    创建 thread_server 线程；
                    启动 thread_server 线程；
                }else
                    唤醒 thread_server 线程 or 释放信号量；
            }
            其他代码；
        }
    }
    int  thread_server(int para)
    {
        while(1)
        {
            提供收到的请求对应的服务；
            线程暂停 or 请求信号量；
```

```
        }
    }
    int main(void)
    {
        创建 thread_comm 线程;
        启动 thread_comm 线程;
    }
```

一般来说，这种方法是不合理的，因 thread_comm 线程是高优先级线程，thread_server 是低优先级线程。创建线程需要很长时间，并且分配栈需要使用动态分配内存，时间和结果都不确定。在高优先级的 thread_comm 线程中为低优先级的 thread_server 线程做执行这些操作，相当于在高优先级的线程中，花很多时间为低优先级线程做事，这是不合理的。
再者，在 thread_comm 中总是判断 thread_server 是否创建，也降低了运行效率。

5.5.2.2 DJYOS

```
int   event_comm (int para)
{
    while(1)
    {
        if（收到请求）
            弹出 evtt_server 类型事件;
        其他代码;
    }
}
int   event_server (int para)
{
    while(1)
    {
        提供收到的请求对应的服务;
        事件同步;
    }
}
int   main(void)
{
    登记 evtt_comm 事件类型;
    弹出 evtt_comm 类型事件;
    登记 evtt_server 事件类型;
}
```

表面上，DJYOS 下的代码和传统操作系统下方案 1 的代码类似，但在水面以下，是完全不同的。因为如果 event_comm 没有收到服务请求，就不会弹出 evtt_server 事件类型，其所对应的线程就不会被创建，也就不会占用计算机资源。

有读者可能会问了：这样不会出现“传统操作系统方案 2”的问题吗？很好的问题，但这个担心在 DJYOS 下是多余的，因为弹出事件仅仅是把事件控制块推进事件调度队列中，需

要的时间是非常少的。而创建线程的操作，只有等到该事件必须被处理时，才执行。因 event_comm 的优先级高于 event_server，故一定会在 event_comm 被阻塞后，才会执行创建线程的操作。

那又有读者可能要问，我要收到服务请求后，需要可靠响应怎么办？的确，创建线程需要分配内存，而分配内存有失败的可能。如果这样，只要把 evtt_server 的优先级定得高于 128，但低于 evtt_comm，这样，在登记事件类型的时候，系统会自动创建线程，等弹出事件之后，就无须创建线程，而是分配线程了。但线程投入运行，仍然是在收到端口发来的数据之后。

5.5.2.3 分析

传统操作系统下，无论任务是否产生，都要占用资源。
DJYOS 下，不实际产生任务，就不占用资源。
孰优孰劣，就不用我说了吧。
举个简单例子，某通信产品，依通信口数量不同，有两种型号，A8 型有 8 个口，A24 型有 24 个口，其他功能一模一样。每个口需要为之创建一个线程，需要分配缓冲区，因此两个型号需要的内存数量和 CPU 速度是不一样的。
传统操作系统实现该产品的话，因为线程在上电时统一创建，要使软件版本保持一致，就必须为 A8 和 A24 型配置相同的内存，付出更高的硬件成本；为了节省成本，A8 和 A24 型会按实际需要配置资源，此时，两个型号的软件版本是不一样的，至少差一个配置常量是不一样的。即使只有一个配置常量不一样，企业也要管理两个不同的版本。
在 DJYOS 下，未安装的通信口，自动就不占用资源，完全没有上述问题。

5.5.3 场景 3

在场景 2 的基础上，如果频繁收到通信口发来的服务请求，但各个请求之间没有内在的关联，各个请求是可以独立处理的，有许多网络服务器接收的请求，以及云计算中的服务请求，就有这个特点。

5.5.3.1 传统操作系统

如果使用“传统操作系统方案 2”的方法，CPU 就只能串行地处理一个个接收到的请求，无论 CPU 的主频多么高，无论计算机有几个 CPU 核，都只能如此。
改善的办法是，创建多个线程，把不同的请求分配到不同的线程，但这样做的问题是，你不知道需要创建多少线程，创建的线程要是少了，服务请求密集到达的时候，就不够用；创建多了，就要占用很多资源，即使没有接收到请求，或者请求很少，这些资源也必须占用着。况且，这些活，操作系统内核是不会帮你做的，一切都要应用程序自己张罗。尤其是，在多核环境下，尤其麻烦。于是，就发展出了复杂的线程池技术，用一个用户级的软件组件去管理线程，有兴趣的可以 google“线程池”，这里就不再赘述了。

5.5.3.2 DJYOS

在 DJYOS 下，天生就能支持优化这种应用。DJYOS 的事件类型控制块中，有三个成员是为此类应用服务的：
vpus_res：繁忙时系统为本类型事件保留的线程数量，在登记事件类型时指定，详见 djy_evtt_regist 函数说明。
vpus_limit：该类型事件同时拥有的线程数量上限，在登记事件类型时指定，详见 djy_evtt_regist 函数说明。
vpus：本类型事件已经拥有的线程数量。
在这种情况下，只要把 5.5.2.2 代码中的 event_server 函数改成如下:

```
int    event_server (int para)
{
    提供收到的请求对应的服务;//注意，事件入口函数不再是 while(1)循环。
}
```

并且在登记 evtt_server 事件类型时，指定该类型为 EN_INDEPENDENCE（独立型）就可以了。
当事件类型登记后，vpus = 0 或 1（如果事件类型的优先级高于 128），若事件频繁弹出，即频繁收到服务请求，操作系统将为此类事件创建多个线程，直到线程数量达到上限 vpus_limit，达到上限后，如果再有新事件弹出，新事件就将阻塞在调度队列中。线程执行完所需服务后，操作系统会查看事件队列中有没有 evtt_server 类型的事件，如果有，就直接把线程转交给它。如果没有，再查看 evtt_server 类型事件拥有的线程数量是否超过 vpus_res，若超过就销毁该线程，否则保留该线程。
这样，如果频繁收到服务请求，操作系统就为 evtt_server 类型事件保持较多线程，否则就维持在比较低的水平。

5.5.3.3 分析

本场景所涉及的，是大量相同或相似任务的线程级并行处理的问题。
传统操作系统面对这种应用时，需要在调度器之上，在应用程序这一级，用复杂的线程池技术才能实现线程级并行，整个程序的复杂度大大增加。而 DJYOS 却在调度器这一级别直接实现了需要用线程池技术才能解决的问题，仅用了非常少的代码量，系统复杂度几乎没有增加。我们都知道，系统越简单，可靠性越高。

5.5.4 场景 4

如何实现事件触发式编程，我们知道，微软的 GUI 是事件触发式编程的，VB、VC 等可视化编程工具下是事件触发式编程的，这说明，事件触发式编程是我们所需要的。在传统操作系统和 DJYOS 下，实现事件触发式编程有什么异同呢？

5.5.4.1 传统操作系统

让我们来看看 CBuilder 下的一个编程场景：

先优雅地拖一个按钮放到桌面上。
为鼠标点击该按钮的事件编写处理函数。
编码工作就这样完成了，很简单很强大吧。
接着编译、执行，用鼠标点击该按钮，就会执行处理函数。
这种优雅高效的编程方式是怎样实现的。原来，由开发工具创建的一个或多个线程一直在后台候着，等待操作系统弹出鼠标点击事件。操作系统则负责检测并弹出事件，潜伏的线程一旦收到鼠标点击事件，便被唤醒执行。我们可以看到，无论该按钮是否被点击，甚至一辈子都不点击，该线程依然要占用系统资源。
在事件触发是编程环境下，程序员只与程序需要处理的具体事件打交到，其编程过程完全与线程、进程等无关。我们也知道，在传统操作系统上，只有在 PC 上或者能运行 linux、wince 等的高端嵌入式平台上才能使用上述便利。为什么呢？究其原因，是因为操作系统是按照线程调度的，必须经过开发工具的包装后，才能转换成事件触发。而这种包装，需要耗费大量的计算机资源，所以只适合在高端平台上使用。再者，这种包装会大大增加软件的复杂性，其复杂性可能带来不可靠以及不可预测因素，也不适合在实时控制系统中应用。因此传统操作系统下，事件触发式只适合于做复杂的界面编程。

5.5.4.2 DJYOS

无须多说了，DJYOS 只提供事件触发式编程模式，无论在高端平台还是在单片机上，都只能用事件触发式编程，区别仅在于，单片机上可能不能提供可视化编程方式。

5.5.4.3 分析

传统操作系统要的面向事件编程，只能在高端平台上实现，是不折不扣的奢侈品，而 DJYOS 却可以在单片机上，使用简单的开发工具就可以实现，摇身一变成为日用品；
传统 OS，即使使用 VC、VB 之类的工具支持，事件触发式编程主要用于界面编程，不能覆盖所有需求，而 DJYOS 只需要单片机开发工具，就覆盖全部需求；
传统 OS 在 VB、VC 之类工具包装后，用事件触发式编程产生的目标程序尺寸庞大、效率低下，而 DJYOS 却跟线程编程有同样的效率。

5.6 事件操作接口

5.6.1 Djy_EvttRegist：注册事件类型

```
u16    Djy_EvttRegist(enum _EVENT_RELATION_ relation,
                      ufast_t default_prio,
                      u16 vpus_res,
                      u16 vpus_limit,
                      ptu32_t (*thread_routine)(void),
                      void *Stack,
                      u32 StackSize,
                      char *evtt_name)
```

头文件：
os_inc.h
参数：
relation：事件类型属性选项。EN_INDEPENDENCE 表示独立型事件类型，EN_CORRELATIVE 表示关联型事件类型。
default_prio：事件类型的默认优先级。
vpus_res：独立型事件类型参数。系统为该事件类型预备的空闲线程上限。
vpus_limit：独立型事件类型参数，系统对该事件类型事件的并行处理能力，即同时处理该类型的事件的数量上限。
thread_routine：线程入口函数(事件处理函数)。DJYOS 的事件，也是需要用线程来处理的。
Stack：静态的栈指针，如果用户已经分配了栈，由此传入指针，Stack==NULL 的话，系统将从堆中动态分配；
StackSize：执行 thread_routine 需要的栈尺寸，不包括 thread_routine 函数可能调用的系统服务。这里设置的栈尺寸，不包括系统补充分配的内核栈大小。
evtt_name：事件类型名。不同模块之间需要交叉弹出事件时，可用事件类型名的方式。如果不指定事件类型名，本参数则为 NULL；但如果指定，事件类型名不允许超过 31 个字符（超出部分忽略不计），且不能与其他事件类型名重名。
返回值：
注册成功返回该事件类型 ID；否则返回 CN_EVTT_ID_INVALID。
说明：
注册一个新的事件类型。下面就事件类型中涉及的一些知识作简要解释。

事件类型/事件 ID
系统为每个类型的事件分配一个唯一的事件类型 ID 号，事件类型 ID 是一个 u16 类型的数据，其中：
bit15 = 1 的 ID 号有特殊用途，例如内存管理模块中用来标识内存 page 的分配情况，具体见 struct mem_global 的 index_event_id 成员的注释。
bit14 = 1 表示该 ID 是事件类型 ID，bit14 = 0 表示该 ID 是事件 ID。事件 ID 的取值范围是 0~16383，事件类型 ID 取值的范围是 16384~32767。
因此，系统支持的事件类型 ID 和事件 ID 数量均为 16384 个，其中事件类型 ID16384 固定用于系统服务。
每次登记一个事件类型系统会分配一个事件类型控制块。事件类型控制块的总量是静态分配的，其数量需按照系统内存量和应用程序实际需求合理设定。工程目录下的 module-trim.c 文件中的常量 gc_u32CfgEvttLimit 的值确定，可以修改但不能在程序运行过程中动态改变。
事件类型 ID 一经注册分配，就不会改变，直到调用 Djy_EvttUnregist 注销之。注销后，该 ID 可以被重新利用。

优先级
DJYOS 允许的优先级是 0~255 ，但登记事件类型时只允许使用 5~249 ，0~4 、250~255 是操作系统保留，不允许用户使用。常用的优先级如下表表。

优先级宏定义	优先级	意义
cn_prio_wdt	1	这是DJYOS看门狗专用，由于djyos的看门狗使用闹钟同步，要每隔一定时间进行监测，优先级必须高于任何事件的优先级才能做到。

cn_prio_critical	100	实时性要求非常高的事件，使用该优先级。
cn_prio_real	120	实时性要求比较高的事件，使用该优先级。
cn_prio_RRS	200	对实时性不是很苛刻的事件。一般情况下，创建事件都使用该优先级。
cn_prio_sys_service	250	系统服务需要的优先级。
cn_prio_invalid	255	无效优先级。

优先级小于 128 的事件，属于紧急事件，在登记事件类型时，系统将为其创建一个线程，并且运行过程中也至少为它保留一个线程。

线程池功能

参数 vpus_res 和 vpus_limit 跟线程池功能相关。基于下面的考虑，系统设定了这两个参数。一方面如果某个独立型的事件频繁弹出，系统需要创建相应多的线程来处理它，但是，线程数量到达一定程度，会耗光系统资源，因此系统为独立型事件类型设置了创建线程数量的上限 vpus_limit。另一方面当事件处理完成后，线程留着也没用了，系统应该销毁它。且慢，由于创建线程会有一定的开销，故系统会保留一些线程，以备再次弹出该类型的事件时，立即把保留的线程分配给该事件。保留的线程数量由 vpus_res 确定。

注意，只有独立型事件类型才有线程池的机制，关联型事件系统最多只为其创建一个线程。

事件类型的先登记后使用原则

DJYOS 的事件类型采用先登记后使用的原则，事件类型登记后，应用程序可以弹出该类型的事件，操作系统通过调度器来决定何时调用事件处理函数。如果是首次弹出该类型事件，必须使用事件类型 ID 作为参数；如果事件队列中已经有该类型事件，弹出事件时也可以使用事件 ID 为参数，详见 Djy_EventPop 函数说明。在事件调度未开始前的初始化过程中，可以登记事件类型和弹出事件，但必须在调用操作系统初始化函数__Djy_InitSys 之后，不过这个限制对用户基本没有影响，因为系统在完成 CPU 核心硬件初始化后，就会立即调用__Djy_InitSys 函数。在__Djy_InitSys 函数中，登记了系统的第一个默认优先级为 250 的事件类型：系统服务事件类型。

参数

每次弹出事件，都可以携带两个 ptu32_t 类型的参数，该参数保存在事件控制块的 param1 和 param2 成员。Djy_GetEventPara 函数用于读取正在处理的事件的参数，连续多次调用 Djy_EventPop 函数的话，后调用的参数将覆盖前面的，而不管早先的参数是否被读取。

evtt_name 的用法

不需要跟模块外的其他模块交互的事件类型，注册事件类型时可以设 evtt_name 为 NULL；如果需与其他模块交互，设置一个字符串名称，可以降解模块间的耦合。假设事件类型不设置没有名称属性，两个分属不同模块，但互相有联系的类型事件，其代码是必须在一起编译，从而导致模块间耦合，而且不能独立开发；有了名称之后，模块间可以通过 Djy_GetEvttId 函数，即使用事件类型名称来联系，如此两个模块就可以互相独立开发和编译，项目经理或系统工程师只要管理好命名空间就可以了。

如何确定 StackSize

StackSize 是调用 thread_routine 所需要的内存尺寸，如果空间不够，将会发生非常严重且神秘莫测的问题。确定 StackSizee 的方法如下：找出线程入口函数 thread_routine 以及调用各

级子函数的可能路径，把该路径中每次调用函数时所需要的参数和局部变量（静态局部变量除外）所需要的内存相加得到该路径的栈需求，取栈需求最大的一条路径，再乘以一个安全系数就可以了。注意如果发生递归调用，则递归函数所需要的内存还应该乘以最大的迭代次数；有些系统中断处理函数使用被中断线程的栈，则每个线程的栈都要加上中断嵌套最深时的中断栈需求。

StackSize 的计算过程如下：

1. 列出 thread_routine 直接和间接调用的所有函数，计算每个参数和局部变量（不含静态变量）所需的内存。
2. 列出线程入口函数 thread_routine 调用各级子函数的所有可能路径。
3. 把各调用路径所调用的函数所需的内存加起来，就是该路径所需要的内存。若有递归调用，该函数所需内存应该乘以最大可能的递归次数。
4. 挑出内存需求最大的一个路径，再乘以一个安全系数，得到 stack_size。该安全系数一般取 1.2 即可，要求特别高的系统，可取 1.5。

返回 CN_EVTT_ID_INVALID 的处理

如果函数返回 CN_EVTT_ID_INVALID，说明事件类型控制块已经耗尽，增大工程目录下的 module-trim.c 文件中的常量 gc_u32CfgEvttLimit 的值可以解决。Shell 命令 evtt 可以查看事件类型控制块的使用情况。

5.6.2 Djy_EventPop：弹出事件

```
u16    Djy_EventPop(u16 hybrid_id,
                    u32 *pop_result,
                    u32 timeout,
                    ptu32_t PopPrarm1,
                    ptu32_t PopPrarm2,
                    ufast_t prio)
```

头文件：

os_inc.h

参数：

hybrid _id：事件类型 ID 或者事件 ID。

pop_result：事件弹出或处理状态，如果函数返回了合法的事件 ID，且 timeout != 0，则 pop_result = CN_SYNC_SUCCESS，表示事件被处理完成后返回；如果 pop_result = CN_SYNC_TIMEOUT，表示事件未被处理完，超时返回；如果 pop_result = EN_KNL_EVENT_SYNC_EXIT，事件处理被异常终止；如果 timeout == 0，则 pop_result 无意义。如果函数返回了 CN_EVENT_ID_INVALID，则返回具体的出错信息。

timeout：超时时间。如果设置为零，则弹出事件后，函数立即返回到调用处继续运行；如果设置为非零，则当前事件（函数调用者）被阻塞，直至等待弹出事件处理完成或超时后才被唤醒，并且在 pop_result 中返回处理结果。无论何种情况，何时处理新弹出事件将由调度器根据优先级决定。timeout 的单位是微秒，其值系统自动向上取整为 CN_CN_CFG_TICK_US（系统心跳）的整数倍。

prio：事件优先级。事件优先级的设定分为两种情况：1. 设定 prio 为零，则使用注册事件类型时的默认优先级（详见函数 Djy_EvttRegist）。2. 设定 prio 非零，则该值为新弹出事件的

优先级。
返回值：
事件成功弹出返回事件 ID；否则返回 CN_INVALID_EVENT_ID。
说明：
向操作系统报告发生某事件类型的事件，操作系统接报后，将分配或建立合适的线程处理该事件。在 DJYOS 系统下，应用程序完成一个完整的事件处理周期是从弹出事件开始的。
事件类型在登记后首次弹出时，hybrid_id 必须使用事件类型 ID，随后，可以使用事件类型 ID，也可以使用事件 ID，前提是该事件 ID 还存在（事件处理完成后，ID 会被收回）。对于关联型事件，使用事件 ID 和事件类型 ID 是相同的。但对于独立型事件，hybrid_id 使用事件 ID 和事件类型 ID 的效果完全不同。如果使用事件类型 ID，则会分配新的事件控制块，新事件插入就绪队列中，并创建（分配）新的线程去处理这条事件；如果是事件 ID，则只会在相应的事件的参数队列中增加一个任务（假如有参数）。在“事件类型弹出同步”功能方面，也有所区别，
如果 timeout != 0，当前正在处理的事件将阻塞，直到调用 Djy_EventSessionComplete 函数，或者事件处理函数自然返回或异常退出，或者超时才解除阻塞。
警告：函数的 prio 参数只适用于弹出新事件。对于“旧（已存在）事件”，即参数 hybrid_id 为事件 ID 或者已弹出过的关联型事件类型 ID 时，该参数是无效的。

5.6.3 Djy_WaitEvttPop：同步事件类型

u32 Djy_WaitEvttPop(u16 evtt_id, u32 *base_times, u32 timeout)
头文件：
os_inc.h
参数：
evtt_id：目标事件类型 ID。
base_times：弹出次数起始值，目标事件累计弹出“*base_times+1”作为同步条件，同步条件达到时，返回实际弹出次数。如果给 NULL，则从调用时的弹出次数+1 做同步条件，不能得到实际弹出次数。
timeout：阻塞时间。表示当前事件被阻塞的时间。如果 timeout 设定为 CN_TIMEOUT_FOREVER，表示本事件被阻塞直至唤醒条件满足才被唤醒。否则，一旦设定时间到达，当前事件则超时返回（事件同样也会被唤醒）。单位为微秒，系统自动向上取整为 CN_CN_CFG_TICK_US（系统心跳）的整数倍。
返回值：
CN_SYNC_SUCCESS：表示等待事件类型的事件弹出次数满足，当前成功唤醒。
CN_SYNC_TIMEOUT：表示阻塞超时。
CN_SYNC_ERROR：表示当前事件阻塞失败（原因如事件类型未注册、事件类型需要被注销以及系统静止调度）。
EN_KNL_EVTTID_LIMIT：事件类型 ID 出错。
说明：
阻塞当前事件，等待 evtt_id 事件类型的事件弹出次数达到设定值（即当前事件被唤醒条件）后，系统调度器会唤醒当前事件。
若“*base_times-当前已弹出次数”>0x80000000，也认为同步条件已经达到，如此设计，是为防止出错条件下没玩没了地等。
同步条件只计有效弹出，不是目标事件类型每次弹出的事件都是有效的，有效弹出的判定比

较复杂，具体详见函数 Djy_EventPop。
1. 如果目标事件类型是关联型事件，每次弹出都有效。
2. 如果是独立型事件，则目标事件类型每次弹出对于弹出同步队列中的其他类型事件都是有效弹出。
3. 如果独立型事件且调用 Djy_WaitEvttPop 时 hybrid_id 是事件 ID，则对弹出同步队列中事件 ID == hybrid_id 的事件算有效弹出。
当 base_times 非 NULL 时，该参数返回值表示的是目标事件类型的事件已弹出的次数。注意，一般情况下，返回时 base_times 的值会加 1，但会有例外。例外情况：
如果调用 Djy_WaitEvttPop 时，传入参数*base_times = 100，正常情况下，pop_times = 101 时，函数将返回，*base_times 应该=101。但如果此时目标事件类型的 pop_times 已经大于 101 的话，函数将立即返回，*base_times 的值将被设为当前的 pop_times。
这个函数最常见的用法是等待自身再次弹出：
Djy_WaitEvttPop(Djy_MyEvttId ( ), NULL, CN_TIMEOUT_FOREVER);
源码 5-1 弹出同步典型用法

```
ptu32_t   net_service( void )
{
    while(1)
    {
        get_connect_request( );
        Djy_EventPop(protocol_evtt_id,...);
    }
}
ptu32_t   net_socket( void )
{
    while(1)
    {
        get_data_packet( );
        Djy_EventPop(protocol_event_id,...);
    }
}
ptu32_t   net_protocol( void )
{
    u32 ptimes = 1;
    while(1)
    {
        do_something;
        Djy_WaitEvttPop(Djy_MyEvttId ( ), &ptimes, CN_TIMEOUT_FOREVER);
    }
}
```

上述代码，网络服务器（net_service）收到连接请求后，弹出事件，使用的是事件类型 ID；socket 模块（net_socket）收到数据包后，弹出事件，用的是事件 ID；协议处理模块(net_protocal)的事件类型被设置为独立型事件，用于处理网络报文，每次处理完报文后，就等待自身再次被弹出。
net_service 弹出事件，由于使用的是事件类型 ID，每次调用系统都会创建新的事件，每条

事件都会有自己的事件 ID，并为每条事件创建新的线程来运行 net_protocol 函数。所有线程处理完当前数据包后，都会在 Djy_WaitEvttPop 阻塞，等待新的数据包。net_socket 弹出事件时，用的是事件 id，此时不会产生新的事件，而是仅仅把 protocol_event_id 相对应的那一条事件激活。

5.6.4 Djy_GetEvttId：查询事件类型 ID

u16 Djy_GetEvttId(char *evtt_name)
头文件：
os_inc.h
参数：
evtt_name：事件类型名称。
返回值：
查找成功返回事件类型 ID；否则则返回 CN_EVTT_ID_INVALID。
说明：
通过事件类型名查找对应的事件类型 ID。

5.6.5 Djy_SaveLastError：保存线程的最后错误

void Djy_SaveLastError (u32 ErrorCode)
头文件：
os_inc.h
参数：
ErrorCode：错误代码。
返回值：
无。
说明：
应用程序或者操作系统运行遇到错误，通过这个函数将错误信息告诉系统。事件控制块有一个变量，记录该事件处理过程中的最后一个错误代码，重复产生的错误码将互相覆盖。

5.6.6 Djy_GetLastError：获取事件最后一次错误码

u32 Djy_GetLastError(void)；
头文件：
os_inc.h
参数：
无。
返回值：
错误代码。
说明：
提取当前事件所记录的最近发生的错误代码。

5.6.7 Djy_SetRRS_Slice：设置轮转周期

void Djy_SetRRS_Slice (u32 slices)
头文件：
os_inc.h
参数：
slices：轮转调度时间片长度，单位为微秒。该值系统自动向上取整为 CN_CFG_TICK_US（系统心跳）的整数倍。
返回值：
无。
说明：
设置系统轮转调度时间片。时间片的系统缺省值为 1 个 CN_CFG_TICK_US，即与系统心跳同频率。如果就绪的最高优先级事件有多个，则轮流执行，每条事件处理时间达到 slices 后，将强制切换到下一条。事件处理线程调用 djy_event_delay(0)将主动出让 cpu，并把自己排到轮转调度的队列尾。如果 slices 设为 0，表示不允许时间片轮转。

5.6.8 Djy_GetRRS_Slice：获取轮转周期

u32 Djy_GetRRS_Slice (void)
头文件：
os_inc.h
参数：
无。
返回值：
当前系统轮转调度时间片长度，单位为 ticks 数（系统心跳）。
说明：
查询系统当前轮转调度时间片的长度。注意，返回值的单位是以系统心跳计的。原因见函数 Djy_SetRRS_Slice。

5.6.9 Djy_EvttUnregist：注销事件类型

bool_t Djy_EvttUnregist(u16 evtt_id)
头文件：
os_inc.h
参数：
evtt_id：待注销的事件类型 ID。
返回值：
注销成功返回 true；失败则返回 false。
说明：
注销一个事件类型。如果事件类型仍然在使用之中（尚有弹出的事件未被处理），事件类型则暂时无法完成注销，系统完成对该类型的所有弹出事件之后，自动注销该事件类型。在等待注销期间，系统不允许弹出该类型的新事件。

事件类型成功注销时，事件类型 ID 和事件类型控制块将被收回，属于该类型的事件处理线程也将被完全销毁。

5.6.10 Djy_SetEventPrio：设置事件优先级

bool_t　Djy_SetEventPrio (ufast_t new_prio)
头文件：
os_inc.h
参数：
new_prio：设置的新优先级。
返回值：
设置成功返回 true；失败则返回 false。
说明：
设置当前事件的优先级。当前事件处理过程中，可以调用本函数，改变自身的优先级。本函数可能会产生系统调度，如果优先级被改低了，且有更高优先级的事件处于就绪态，将立即调度，阻塞本事件运行。
这个函数只能改变调用者自身的优先级，不能修改其他事件优先级，包括优先级在内，DJYOS 没有任何直接改变其他事件的参数的函数。

5.6.11 Djy_EventDelay：事件延时

u32　Djy_EventDelay(u32 u32l_uS)
头文件：
os_inc.h
参数：
u32l_Us：延迟（阻塞）时间，单位是微秒。该值系统自动向上取 CN_CFG_TICK_US (系统心跳)的整数倍。
返回值：
实际延时（阻塞）时间，微秒数。
说明：
将当前（正在执行）事件延时（阻塞）u32l_uS 微秒后继续运行。
阻塞时间将向上调整为 tick 时间的整数倍。因此，不能获得精度高于 tick 间隔的延时。注意设置 u32l_uS 为零时，如果系统存在其他同优先级的事件，则调度器会让其他同优先级的事件将先执行，不管轮转调度是否允许，如果没有相同优先级的就绪事件，则直接返回。
由于参数值取整处理、优先级抢占或关中断等因素，会附加额外的延时时间，函数的实际返回的延时（阻塞）时间要比设定值长。

5.6.12 Djy_EventDelayTo：事件定时

u32 Djy_EventDelayTo(s64 s64l_uS);
头文件：
os_inc.h

参数：
s64l_uS：延时结束时刻。该时刻已开机时刻为基准，单位是微秒。参数值系统将自动向上取整 CN_CFG_TICK_US 的整数倍。
返回值：
实际延时时间，单位微秒。
说明：
将当前时间延时到 s64l_uS 时刻再继续执行。参数延时结束时刻是一个 64 位数，理论上需要 29 万年才会发生溢出，因此不考虑数据溢出问题。

5.6.13 Djy_WaitEventCompleted：等待事件完成

u32 Djy_WaitEventCompleted(u16 event_id, u32 timeout)
头文件：
os_inc.h
参数：
event_id：目标事件 ID。
timeout：阻塞时间。表示当前事件被阻塞的时间。如果 timeout 设定为 CN_TIMEOUT_FOREVER，表示本事件被阻塞直至指定事件完成处理才会唤醒。否则，一旦设定时间到达，当前事件则超时返回（事件同样也会被唤醒）。timeout 单位为微秒，系统自动向上取整为 CN_CFG_TICK_US（系统心跳）的整数倍。CN_CFG_TICK_US
返回值：
CN_SYNC_SUCCESS：指定事件完成处理。
CN_SYNC_TIMEOUT：当前事件阻塞超时。
EN_KNL_CANT_SCHED：当前事件阻塞失败。
EN_KNL_EVENT_FREE：参数 event_id 出错（不存在）。
说明：
等待事件完成。阻塞当前正在处理的事件，系统重新调度。当指定事件处理完成，或者异常退出，或者阻塞超时，系统唤醒当前事件。

5.6.14 Djy_WaitEvttCompleted：等待事件类型完成

u32 Djy_WaitEvttCompleted(u16 evtt_id, u16 done_times, u32 timeout)
头文件：
os_inc.h
参数：
evtt_id：目标事件类型 ID。
done_times：当前事件被唤醒的条件选项。当设定 done_times 为非零值，表示当前事件被阻塞后，系统完成 done_times 次 evtt_id 类型的事件处理后唤醒当前事件。当设定其为零值，则表示当前事件被阻塞后，系统完成所有 evtt_id 类型的事件，才唤醒当前事件。
timeout：阻塞时间。表示当前事件被阻塞的时间。如果 timeout 设定为 CN_TIMEOUT_FOREVER，表示当前事件被阻塞直至唤醒条件（见 done_times 参数）满足，才会唤醒。否则，一旦设定时间到达，当前事件则超时返回（事件同样也会被唤醒）。timeout 值，单位为微秒，系统自动向上取整为 CN_CFG_TICK_US（系统心跳）的整数倍。

返回值：
CN_SYNC_SUCCESS：指定事件类型的事件完成处理。
CN_SYNC_TIMEOUT：当前事件阻塞超时。
EN_KNL_CANT_SCHED：当前事件阻塞失败。
EN_KNL_EVTTID_LIMIT，事件类型 ID 出错（不存在）。
说明：
等待事件类型的事件完成处理。将当前事件阻塞，直至设定的唤醒条件满足，当前才事件被唤醒。本函数与函数 Djy_WaitEventCompleted 类似，但本函数的唤醒条件与事件类型相关。

5.6.15 Djy_GetEventResult：查询事件处理结果

ptu32_t　Djy_GetEventResult(void)
头文件：
os_inc.h
参数：
无
返回值：
当前事件的处理结果。
说明：
查询当前事件弹出的事件的处理结果。一个事件在处理过程中，如果弹出了新事件，并且等待事件处理结果(调用 Djy_EventPop 函数时 timeout !=0)，且调用 Djy_EventPop 时返回了合法的事件 id，又不是超时返回，则可以用本函数获取新事件的处理结果。只能取最后一次成功处理的事件结果。

5.6.16 Djy_GetEventPara：获取事件参数

void　*Djy_GetEventPara (ptu32_t *Param1, ptu32_t *Param2)
头文件：
os_inc.h
参数：
Param1：事件参数。
Param2：事件参数。
返回值：
无。
说明：
提取在弹出事件（Djy_EventPop）时，传递给当前事件的参数。

5.6.17 Djy_EventSessionComplete：“积累事件”完成

void　Djy_EventSessionComplete(ptu32_t result)
头文件：
os_inc.h

参数：
result：传递“阻塞事件”结果（数据）。
返回值：
无。
说明：

“积累事件”完成，对于关联型（EN_CORRELATIVE）事件，事件处理函数可以写成 while（1），是没有所谓的“完成”的，但系统运行中，其他模块是可以反复弹出该类型事件的。事件弹出后，根据系统运行状况，新弹出的事件不一定会立即被处理，等到被处理时，可能已经弹出了一次或多次事件，称作“积累事件。调用 Djy_EventSessionComplete 函数，就是报告操作系统，积累的事件所代表的任务已经处理完毕。同样，即使是独立型事件（EN_INDEPENDENCE），如果调用 Djy_EventPop 时使用了事件 ID，“积累事件”的概念同样适用。如果将激活阻塞在本事件的事件同步（即等待本事件完成）队列中的全部事件。并把当前事件的处理结果传递给所有被阻塞事件的事件控制块中的 event_result 成员，相应的事件可以调用 Djy_GetEventResult 函数获取。一般用于调用 Djy_WaitEvttPop 且返回 CN_SYNC_SUCCESS 后，用于通知调用 Djy_EventPop 的事件“该次弹出已经被处理”。 如果该次 Djy_EventPop 设定了同步(即 timeout !=0)，则同步条件达成。

5.6.18 Djy_MyEvttId：获取当前事件类型 ID

u16　Djy_MyEvttId(void)
头文件：
os_inc.h
参数：
无。
返回值：
查询成功返回事件类型 ID，否则返回 CN_EVTT_ID_ASYN。
说明：
由应用程序调用，取正在处理的事件的事件类型 ID。如果当前运行在异步信号中断中，则返回 CN_EVTT_ID_ASYN。

5.6.19 Djy_MyEventId：获取当前事件 ID

u16　Djy_MyEventId(void)
头文件：
os_inc.h
参数：
无。
返回值：
查询成功返回事件 ID，否则返回 CN_EVTT_ID_ASYN。
说明：
由应用程序调用，取当前（正在处理的）事件的 ID。如果当前运行在异步信号中断中，则返回 CN_EVTT_ID_ASYN。

5.6.20 Djy_WakeUpFrom：查询当前事件唤醒原因

u32　Djy_WakeUpFrom(void)
头文件：
os_inc.h
参数：
无。
返回值：
当前事件被唤醒原因。
说明：
查询当前事件被唤醒（切入）原因。在多线程环境下，事件处理过程可能被反复阻塞和唤醒，当前（正在执行中的）事件肯定是就绪态，调用本函数，可查询自己被唤醒的原因是什么。
唤醒原因如下：

```
#define CN_STS_EVENT_READY          (u32)0
#define CN_STS_EVENT_DELAY          (u32)(1<<0)        //闹钟同步
#define CN_STS_SYNC_TIMEOUT         (u32)(1<<1)        //超时
#define CN_STS_WAIT_EVENT_DONE    (u32)(1<<2)        //事件同步
#define CN_STS_EVENT_EXP_EXIT     (u32)(1<<3) //事件同步中被同步的目标事件异常退出
#define CN_STS_WAIT_EVTT_POP       (u32)(1<<4)        //事件类型弹出同步
#define CN_STS_WAIT_EVTT_DONE     (u32)(1<<5)        //事件类型完成同步
#define CN_STS_WAIT_MEMORY          (u32)(1<<6)        //从系统堆分配内存同步
#define CN_STS_WAIT_SEMP            (u32)(1<<7)        //信号量同步
#define CN_STS_WAIT_MUTEX           (u32)(1<<8)        //互斥量同步
#define CN_STS_WAIT_ASYN_SIGNAL (u32)(1<<9)        //异步信号同步
#define CN_STS_WAIT_CPIPE           (u32)(1<<10)       //定长 pipe
#define CN_STS_WAIT_VPIPE           (u32)(1<<11)       //动态长度 pipe
#define CN_STS_WAIT_MSG_SENT        (u32)(1<<12)       //等待消息发送
#define CN_STS_WAIT_PARA_USED       (u32)(1<<13)       //等待消息处理完成
#define  CN_WF_EVTT_DELETED          (u32)(1<<14)       //事件类型相关的同步，因目标类型
                                                        //被删除而解除同步。
#define CN_WF_EVENT_RESET           (u32)(1<<15)       //复位后首次切入运行
#define CN_WF_EVENT_NORUN           (u32)(1<<16)       //事件还未开始处理
```

如果事件是因为同步超时而返回，则有两个 bit 会置 1，一个是 CN_STS_SYNC_TIMEOUT，表示超时，另一个表示更具体的原因，例如 CN_STS_WAIT_SEMP = 1 表示在阻塞等待信号量时因超时而被唤醒。注意唤醒后并不一定立即投入运行，还得看优先级说话呢。

5.6.21 关闭调度

关闭调度和关闭异步信号是等同的操作，参见 6.4.1。
关闭实时中断也将禁止调度，参见节 6.4.1。
特别注意，如果是通过原子操作（int_low_atom_start 或者 int_high_atom_start）进入禁止中

断的状态，调度将处于允许状态，你在这种时候如果调用可能引起阻塞的操作，结果将不可预知。

5.6.22 Djy_QuerySch：查询调度器状态

bool_t Djy_QuerySch(void);
头文件：
os_inc.h
参数：
无。
返回值：
允许调度返回 true；否则返回 false。
说明：
查询当前是否允许调度

第6章 中断系统

使用 RTOS 的场合，大多都对实时性有要求，甚至有些要求极端高的实时性。一般来说，设计者会把实时性要求很高的部分功能，用中断来实现，从这个意义上，对于 RTOS 来说，最坏情况下的中断响应延迟，几乎就是该系统的实时性指标。如果操作系统会带来额外的、甚至不确定的中断延时，那么对实时性要求很高的应用，就不能使用操作系统，或者需要增加额外的硬件来处理实时任务。DJYOS 在最坏情况下，提供裸跑的中断延迟，实现无以伦比的实时性，使得一些原来只能裸跑的应用，也可以使用操作系统。

6.1 理解中断

中断如闲云野鹤，来去无踪，操作系统根本不知道什么时候会来中断；许多 CPU 响应中断后，会自动切换到特权级别，中断服务函数可以无法无天，根本不受控制。所有这些，都增加了中断系统的设计难度。很多操作系统，对中断，根本就是把中断视为虎狼，在执行临界代码时，一关了之；同时，为了降低操作系统对实时性的影响，又千方百计地缩短持续关中断时间，把代码搞得很复杂。
中断，从形式上看，是一个异步到达的（通知）事件，异步是什么意思呢？异步是相对于代码执行序列的，即 CPU 根本不知道什么时间会来中断，也不知道中断达到时，自己运行到什么地方、处于什么状态。中断从形式上，并没有告诉我们，它是否需要处理，更没有说是否需要紧急处理。因此，把中断都看作“需要紧急处理的事件”，是没有依据的。
由于 CPU 是串行执行指令的，如果靠 CPU 执行指令的方式查询获得事件，比如查询 IO 口获得上升沿事件。从事件发生到 CPU 查询到的延迟时间，必然跟 cpu 的查询周期相关，查询间隔短了，CPU 开销太大，间隔长了，又会错过中断，因此，紧急事件只能通过中断来获取。
所以，中断可以用三句话来概括：
1. 中断是异步事件。
2. 中断不一定是紧急事件。

3. 紧急事件必须用中断来通知。
了解了中断的三个特性，我们就知道，并不是所有中断，都需要极速响应的，事实上，绝大多数情况下，只有极少量的中断需要很高实时性的。对不同需求的中断，不能一视同仁，更不能把中断都看成是一个需要紧急处理的事件。
在操作系统支持下，需要接近裸跑的中断延迟吗？
答案显然是肯定的，但是，人们对此似乎很宽容。为什么会宽容呢？当没有汽车时，人们能够容忍马车的速度；没有火车时，人们容忍汽车的速度；没有飞机时，人们容忍火车的速度。当所有操作系统都做不到近乎裸奔的中断延迟时，人们便容忍 OS 给中断响应带来额外延迟。当 DJYOS 实现了裸跑实时性的时候，你还会犹豫吗？
为什么要为一些不紧急的事件（例如键盘）分配中断号呢？
设计者经常为键盘分配中断号，对于台式 PC 来说，键盘也许勉强算紧急事件，因为 PC 中有很多慢速操作；但对于使用 RTOS 的嵌入式控制系统来说，却并非紧急事件。为什么呢？因为键盘是人手通过机械按钮实现的，操作系统的反映，只要比人的动作快就可以了，响应太快了，反而可能导致误触发，失去按键防抖功能。那为什么不少人会把按键挂在中断线上，使用中断来响应按键操作呢？答案是，一可以简化软件设计，不需要单独启动一个键盘扫描线程；二可以减轻 cpu 负担，不用定期扫描键盘；三是低功耗系统特有的，可以用键盘中断来唤醒休眠的 cpu。

6.2 中断管理体系

理解了中断，我们就可以着手设计中断系统了。
1. 能够为异步事件提供服务，既然是事件，就应该提供与普通事件一样的操作系统服务，使其编码更加容易。
2. 为实时中断提供裸跑的中断延迟，即使在最坏情况下，也不能例外。
3. 用户用中断处理异步事件，是希望简化系统设计，因此，不能因为异步事件而使调度系统更加复杂。
既企图对所有中断极速响应，又企图给中断响应函数以尽可能多的服务，是不是太贪婪了点？鱼和熊掌，选一样吧。更多更便利的服务，必然需要更多的关中断，直接导致中断延迟加长。
最终的结果是：
1. 异步事件得不到操作系统的充分支持，对编写 ISR 程序有诸多限制。
2. 真正实时性要求很高的中断，却做不到很快响应。
3. 调度器保护临界资源时，分为关调度和关中断两级，使系统更加复杂，完全违背了用户的用中断响应异步事件以简化软件的目的。
DJYOS 系统设计中断管理模块时也体现了系统的“九九加一”原则：
1. 日常大量存在的、实时性并不是特别高的工作，系统提供最大的便利，让程序员能够简洁地实现。
2. 极少遇到的、高难度的、甚至挑战系统实时性和处理能力极限的工作，系统提供最大的灵活性，使问题的解决成为可能。
基于上述原则，在 DJYOS 系统中，中断被分为两大类，第一类是实时中断，对应现实世界中紧急程度非常高的中断信号，实时中断的响应与前后台系统无异，具有接近前后台系统的实时性，操作系统运行过程中，调度程序永远不会关闭实时中断，只是提供一个接口函数，使线程可以根据需要临时关闭实时中断。在实时中断里，程序员能够像在前后台系统中一样自由自在地编程，除了不能使用操作系统的系统调用外，没有太多的束缚，相应地，操作系统为实时中断提供的服务也最少。实时中断为用户的紧急突发事件提供绿色通道，它的实时

性能要远高于其他操作系统的中断体系。
第二类是异步信号，对应紧急程度不是很高的中断信号，异步信号和普通事件没有实质性的差别，内核把他们等同看待。但因硬件特性，异步信号与普通事件还是有区别的，他们的相同点表现在：
1. 禁止事件调度和禁止异步事件切换是等同的，也就是说，当禁止调度时，DJYOS 把异步信号当作事件一样也禁止了。
2. 异步信号 ISR 可以从堆中分配内存，可以释放和申请信号量，以及使用其他临界资源。
3. 异步信号 ISR 可以使用所有的系统服务。
异步事件和普通事件的不同点表现在：
1. 异步信号没有独立的上下文，故 ISR 不可以被阻塞。虽然允许调用所有的系统调用，但不会发生实际阻塞。例如 malloc 函数，如果在线程中调用，内存不足时，线程将被阻塞；如果在 ISR 中调用，线程不足时将直接返回 NULL。
2. 由于异步信号优先级高于所有普通事件，故其 ISR 不能像事件处理函数那样，做成死循环的形式。
错误！未找到引用源。显示了中断系统的整体架构。

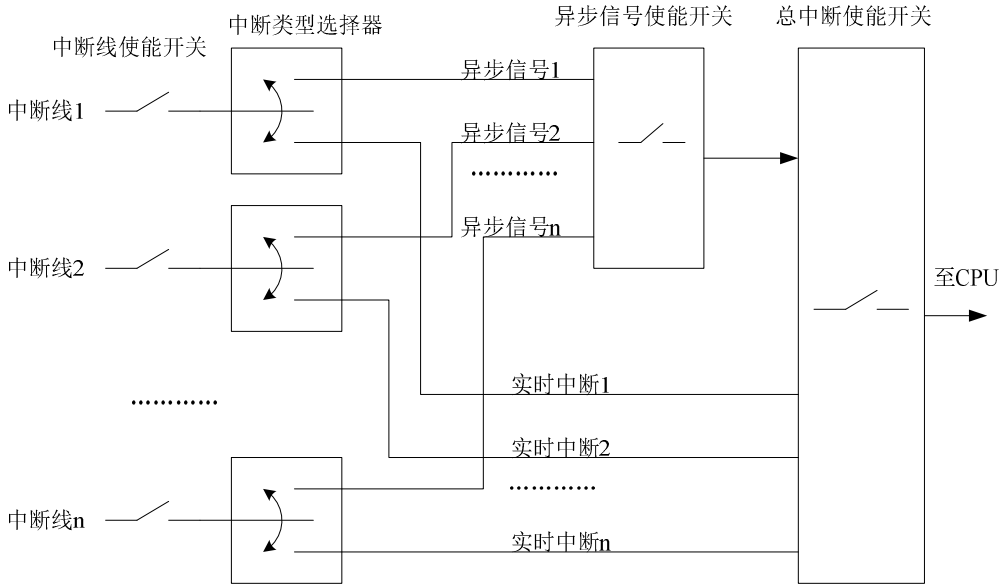


图 6-1 中断结构

一般 cpu 的中断控制器，都有许多中断输入线，每个中断线对应一个中断号，由一个独立的开关控制。
每个中断线，都允许独立设置为实时中断还是异步信号，该开关可能是硬件开关，也可能是软件开关，依赖于具体硬件以及移植者的选择。
异步信号有一个独立的总使能开关，也就意味着，异步信号是可以整体关断的。
实时中断没有独立总使能开关，异步信号和实时中断有一个公共的总使能开关，意味着，允许用户关闭所有中断。
DJYOS 中，“异步信号使能开关”同时又是关调度开关，DJYOS 没有独立的调度开关，使得 DJYOS 的临界区保护代码更加简洁，和用户希望通过“用中断实现异步信号”来简化软件设计的目的，是相适应的。简洁的调度器，也使 DJYOS 更加可靠。由于关异步信号和关中断等同，使得异步信号处理函数可以使用操作系统提供的全部服务，更加方便易用。
操作系统运行过程中，总中断使能开关时从来不会被关闭的，所以，DJYOS 的实时中断处理能力，达到裸跑的速度，这是架构决定的，跟中断响应的代码是否精简无关。

DJYOS 中断系统的特点

1. 实时中断响应具有裸跑的实时性，使得有些原本不能使用操作系统的应用，可以享受操作系统的服务。
2. 编写异步信号编程更加方便，像普通事件一样，允许使用所有操作系统服务。
3. 系统更加简洁可靠，合并关闭调度和关闭异步信号的操作后，使系统临界区保护相关的代码大大简化。

6.3 中断使用步骤

6.3.1 填写中断表

在工程目录\src\user\critical\critical.c 文件中，有一个 tg_IntUsed 表，这是与具体工程相关的配置代码，工程配置时，需要把该工程所使用到的所有中断源填入这个表格，如下：

```
const ufast_t    tg_IntUsed [] =
{
      CN_INT_LINE_TIM2,                 // (28) - TIM2 global Interrupt
      CN_INT_LINE_TIM3,                 // (29) - TIM3 global Interrupt
      CN_INT_LINE_TIM4,                 // (30) - TIM4 global Interrupt
      CN_INT_LINE_I2C1_EV,              // (31) - I2C1 Event Interrupt
};
```

在文件“djysrc\bsp\cpudrv\name_of_cpu\include\cpu_peri_int_line.h”中，定义了该 cpu 支持的所有中断源，但只有在这个表（数组）中配置了的中断，应用程序中才能使用。一般来说，内存较大的系统，建议把所有中断源都放进去。

特殊情况：有些 cpu，例如 ADI 的 adsp 系列 dsp，或者 TI 的 C6000 系列 dsp，cpu 支持的中断向量数少于外部中断源数，例如 adsp 的 cpu 只支持 32 个中断向量，而中断源却达到 44 个，这个表中就只能放 32 个。

6.3.2 实时中断编程模型

实时中断编程的 ISR 与落跑没有差异，其响应速度也与裸跑没有差异。

1. 编写中断服务程序 ISR，像普通 C 函数一样编写即可。

```
u32 __uart1_int(ufast_t uart_int_line)
{
     中断服务代码
}
```

2. 使用下列语句序列设置中断：

```
Int_IsrConnect(cn_int_line_uart1,__uart1_int);
Int_SettoReal(cn_int_line_uart1);
Int_ClearLine(cn_int_line_uart1);
Int_RestoreAsynLine(cn_int_line_uart1);
```

第一句，把 ISR 函数跟中断线联系起来。

第二句，把相应中断设置成实时中断。

第三句，清一下中断，以免受硬件的初始化状态影响。

第四句，使能中断
上述初始化过程，来自一个实际应用，该案例的 MCU 使用 lpc1225，主频 40Mhz，flash 速度 20Mhz，实际运行速度在 20Mhz~40Mhz 之间。但要实现 2.5Mbps baud 的串口通信，如果不能再接收中断发生 5uS 内响应中断的话，必然丢数据。因此，该应用使用了 DJYOS 的实时中断机制，实测下来，中断响应时间小于 1.5uS，效果很理想。

6.3.3 异步信号的事件模式编程

这种模型的优点是，不需要编写 ISR 函数，而且在异步信号的处理事件中，编程与普通线程编程一样，没有额外限制。下面以串口服务为例：
1. 编写事件处理函数。

```
void    uart1_event(void)
{
      While（1）
      {
            //do something
            Djy_WaitEvttPop(djy_my_evtt_id(),
                              NULL,
                              CN_TIMEOUT_FOREVER);
      }
}
```

2. 登记事件类型，用于异步信号处理的事件，一般设为关联型事件。

```
uart1_evtt = Djy_EvttRegist(EN_CORRELATIVE,
                                 100,
                                 0,
                                 uart1_event,
                                 0x1000,
                                 NULL);
```

3. 按照下列指令序列初始化中断号：

```
Int_EvttConnect(cn_int_line_USART1,uart1_evtt);
Int_SettoAsynSignal(cn_int_line_USART1);
      Int_SetClearType(cn_int_line_USART1, CN_INT_CLEAR_PRE);,
Int_ClearLine(cn_int_line_USART1);
Int_RestoreAsynLine(cn_int_line_USART1);
```

第一句，把事件类型与中断号联系起来。
第二句，把中断号设为异步信号。
第三句，设置清中断方式，不可以使用默认的 CN_INT_CLEAR_USER。详见 6.4.13 节。
第四句，清一下中断，以免硬件的初始化状态影响。
第五句，使能中断。
完成以上 3 步，只要发生 uart 中断，就会自动弹出 uart1_evtt 类型的事件。第一次弹出事件后，uart1_event 将被执行，处理完相应的事务后，将会在 *Djy_WaitEvttPop* ( )函数处阻塞，直到下一次中断的到来。应用程序完全不需要写 ISR 程序，由于 uart1_event 是普通的事件处

理函数，它跟普通事件处理函数一样，可以得到所有 OS 服务。
要实现超时处理（即串口长时间收不到数据）也很简单，只需要把：
Djy_WaitEvttPop(djy_my_evtt_id(), NULL, CN_TIMEOUT_FOREVER);
中的 CN_TIMEOUT_FOREVER 换成具体的时间（微秒数）就可以了。

6.3.4 异步信号的 ISR 模式编程

这种编程模型，是大家非常熟悉的方式，不同的是，DJYOS 的 ISR 可以调用全部系统服务，唯一的区别是，ISR 中调用可能导致阻塞的系统服务时，不会发生实际阻塞。比如，在 ISR 中调用 malloc，但如果内存不足，将返回 NULL，不会被阻塞，而在线程中调用的话，将会被阻塞。只要养成良好的编程习惯，检查系统调用的返回值，就不会出错。
异步信号的 ISR 模式编程步骤如下：
1.编写中断服务程序 ISR，像普通 C 函数一样编写即可。

```
u32   __uart1_int(ufast_t uart_int_line)
{
     中断服务代码
}
```

2.使用下列语句序列设置中断：

```
Int_IsrConnect(cn_int_line_uart1,__uart1_int);
Int_SettoAsynSignal(cn_int_line_uart1);
Int_ClearLine(cn_int_line_uart1);
Int_RestoreAsynLine(cn_int_line_uart1);
```

第一句，把 ISR 函数跟中断线联系起来。
第二句，把相应中断设置成异步信号中断。
第三句，清中断，以免硬件的初始化状态影响。
第四句，使能中断。
这个过程，跟实时中断非常相似，就是函数名不一样而已，但实现过程是有很大差别的。这种方式下，__uart1_int 函数可以调用全部系统服务，但中断响应延时可能比实时中断长。

6.3.5 异步信号的 ISR 和事件混合模式编程

这是一种类似把中断处理分为上半部和下半部编程的一种方法，在 DJYOS 中实现起来特别简单，步骤如下：
1. 编写事件处理函数。

```
void uart1_event(void)
{
      While（1）
     {
          //do something
          Djy_WaitEvttPop(djy_my_evtt_id(), NULL, CN_TIMEOUT_FOREVER);
     }
}
```

2. 登记事件类型，用于异步信号处理的事件，一般设为关联型事件。

```
uart1_evtt = Djy_EvttRegist(EN_CORRELATIVE,
                            100,
                            0,
                            uart1_event,
                            0x1000,
                            NULL);
```

3. 编写中断服务代码 ISR，像普通 C 函数一样编写即可。

```
u32    __uart1_int(ufast_t uart_int_line)
{
      中断服务代码
}
```

4. 按照下列指令序列初始化中断号：

```
Int_EvttConnect(cn_int_line_USART1,uart1_evtt);
Int_IsrConnect (cn_int_line_uart1,__uart1_int);
Int_SettoAsynSignal (cn_int_line_USART1);
Int_ClearLine (cn_int_line_USART1);
Int_RestoreAsynLine (cn_int_line_USART1);
```

第一句，把事件类型与中断号联系起来。
第二句，把 ISR 函数与中断号联系起来。
第三句，把中断号设为异步信号。
第四句，清中断，以免硬件的初始化状态影响。
第五句，使能中断。
此后，只要发生 uart 中断，就会先调用__uart1_int 函数，然后自动弹出 uart1_evtt 类型的事件。可以在__uart1_int( )函数中处理比较紧急的事务，比如 copy 硬件缓冲区，把其他事情留给 uart1_event( )做。第一次弹出事件后，uart1_event 将被执行，处理完相应的事务后，将会在 djy_wait_evtt_pop( )函数处阻塞，直到下一次事件的到来。

6.4 中断操作接口

6.4.1 Int_SaveAsynSignal：禁止异步信号

void Int_SaveAsynSignal(void)
头文件：
os_inc.h
参数：
无。
返回值：
无。
说明：
禁止异步信号。
这是函数与 Int_RestoreAsynSignal 配对使用。详见 Int_RestoreAsynSignal 函数。

6.4.2 Int_RestoreAsynSignal：使能异步信号

void Int_RestoreAsynSignal(void)
头文件：
os_inc.h
参数：
无。
返回值：
无。
说明：
使能异步信号。函数会直接操作底层硬件，关闭和开启异步信号类型的中断线。
另外注意，这对函数等同于“禁止/使能调度”。同时 DJYOS 的心跳时钟是通过异步信号实现的，禁止异步信号期间，系统心跳停止。
在禁止异步信号期间，如果调用可能会引起事件阻塞的函数，则阻塞作用是无效的，该函数会立即返回。例如某事件调用 malloc 函数时，内存不足且禁止异步信号，函数会立即返回 NULL。

6.4.3 Int_CheckAsynSignal：查询异步信号状态

bool_t Int_CheckAsynSignal(void)
头文件：
os_inc.h
参数：
无。
返回值：
允许异步信号返回 true；禁止异步信号返回 false。
说明：
查询异步信号是否允许。注意，这里所查询是软件概念上的异步信号，而非实际硬件中断源的状况。调用 Int_HighAtomStart 或 Int_LowAtomStart 关闭中断的话，不影响本函数。

6.4.4 Int_SaveAsynLine：禁止异步信号线

bool_t Int_SaveAsynLine(ufast_t ufl_line)
头文件：
os_inc.h
参数：
无。
返回值：
允许异步信号返回 true；禁止异步信号返回 false。
说明：
禁止异步信号线。
函数与 Int_RestoreAsynLine 成对使用。详情见函数 Int_RestoreAsynLine 说明。

6.4.5 Int_RestoreAsynLine：使能异步信号线

bool_t Int_RestoreAsynLine(ufast_t ufl_line)
头文件：
os_inc.h
参数：
ufl_line：中断线号，该编号必须出现在 critical.c 文件中定义的 tg_IntUsed 表中。
返回值：
true 表示成功操作，false 表示操作失败。
说明：
如果某中断线被设置为异步信号，使用这两个函数使能和禁止该中断线。
DJYOS 用计数器来控制中断线开关状态，允许嵌套调用，调用 Int_SaveAsynLine 使计数器加 1，调用 Int_RestoreAsynLine 使计数器减 1。计数器为 0 时使能中断，为 1 时禁止中断。

6.4.6 Int_EnableAsynLine：直接使能异步信号线

bool_t Int_DisableAsynLine(ufast_t ufl_line)
头文件：
os_inc.h
参数：
ufl_line：中断线号，该编号出现在 critical.c 文件中定义的 tg_IntUsed 表中。
返回值：
成功操作返回 true；失败则返回 false。
说明：
直接使能异步信号线。
函数与 Int_EnableAsynLine 成对使用。详情见函数 Int_EnableAsynLine。

6.4.7 Int_DisableAsynLine：直接禁止异步信号线

bool_t Int_EnableAsynLine(ufast_t ufl_line)
头文件：
os_inc.h
参数：
ufl_line：中断线号，该编号出现在 critical.c 文件中定义的 tg_IntUsed 表中。
返回值：
成功操作返回 true；失败则返回 false。
说明：
禁止和使能异步信号类型的中断线。
这对函数直接禁止和使能该中断线，类似于驱动中开启和关闭中断的操作。

6.4.8 Int_SaveRealLine：禁止实时中断线

bool_t Int_SaveRealLine (ufast_t ufl_line)
头文件：
os_inc.h
参数：
ufl_line：中断线号，该编号必须出现在 critical.c 文件中定义的 tg_IntUsed 表中。
返回值：
操作成功返回 true；失败则返回 false。
说明：
禁止实时信号。
本函数与 Int_RestoreRealLine 配对使用，详情见函数 Int_RestoreRealLine。

6.4.9 Int_RestoreRealLine：使能实时中断线

bool_t Int_RestoreRealLine(ufast_t ufl_line)
头文件：
os_inc.h
参数：
ufl_line：中断线号，该编号必须出现在 critical.c 文件中定义的 tg_IntUsed 表中。
返回值：
操作成功返回 true；失败则返回 false。
说明：
如果某中断线被设置为实时中断，使用这两个函数使能和禁止该中断线。
DJYOS 用计数器来控制中断线开关状态，允许嵌套调用，调用 Int_SaveRealLine 使计数器加 1，调用 Int_RestoreRealLine 使计数器减 1。计数器为 0 时使能中断，为 1 时禁止中断。

6.4.10 Int_EnableRealLine：直接禁止实时中断线

bool_t Int_EnableRealLine(ufast_t ufl_line)
头文件：
os_inc.h
参数：
ufl_line：中断线号，该编号必须出现在 critical.c 文件中定义的 tg_IntUsed 表中。
返回值：
操作成功返回 true；失败则返回 false。
说明：
直接实时实时中断线。
函数与 Int_DisableRealLine 配对使用，详情见函数 Int_DisableRealLine。

6.4.11 Int_DisableRealLine：直接使能实时中断线

bool_t　Int_DisableRealLine(ufast_t ufl_line)
头文件：
os_inc.h
参数：
ufl_line：中断线号，该编号必须出现在 critical.c 文件中定义的 tg_IntUsed 表中。
返回值：
true 表示成功操作，false 表示操作失败。
说明：
直接使能实时中断线。
如果某中断线被设置为实时中断，使用这两个函数使能和直接禁止该中断线。这对函数直接使能或者禁止相应中断线，执行效率较高，是不允许嵌套调用的。

6.4.12 Int_CheckLine：检查中断线状态

bool_t　Int_CheckLine (ufast_t ufl_line)
头文件：
os_inc.h
参数：
ufl_line，中断线号。
返回值：
true 表示中断线使能，false 表示禁止，或该中断线未使用，即未出现在 critical.c 文件中定义的 tg_IntUsed 中。
说明：
检查中断线 ufl_line 是否使能，不区分异步信号和实时信号。

6.4.13 Int_SetClearType：设置清中断方式

bool_t　Int_SetClearType (ufast_t ufl_line,ufast_t clear_type)；
头文件：
os_inc.h
参数：
ufl_line：中断线编号；
clear_type：设定清中断方式。
返回值：
设置成功返回 true；失败则返回 false。
说明：
中断响应后，绝大多数硬件都会暂时屏蔽本中断再次响应，直到调用 Int_ClearLine 清中断。在中断发生到清中断期间重复发生的同一个中断将被忽略，既不响应也不挂起。清中断后，再次发生的中断将会挂起，是立即响应还是等前一次中断返回后再响应，则取决于硬件以及软件设置。DJYOS 提供三种清中断方式，任意中断线可根据自身需要设置：

```
#define   CN_INT_CLEAR_PRE      0    //调用 ISR 之前由系统自动清
#define   CN_INT_CLEAR_USER     1    //系统不清，由用户在 ISR 中清，默认方式
#define   CN_INT_CLEAR_POST     2    //调用 ISR 返回之后、中断返回前由系统自动清
```

只有异步信号可以设置清中断方式，实时中断必须由用户的 ISR 清中断。
切记，事件模式实现的异步信号（参见 6.3.3 节），必须设置为 ISR 之前或之后清，不允许使用默认值。

6.4.14 Int_IsrConnect：注册中断响应函数

void Int_IsrConnect(ufast_t ufl_line, u32 (*isr)(ufast_t))
头文件：
os_inc.h
参数：
ufl_line：中断线号。
isr：被关联的中断处理函数。
返回值：
操作成功返回 true；失败则返回 false。
说明：
为中断线 ufl_line 指定（注册）中断响应函数。中断发生后，系统将调用该响应函数。本函数不区分该中断线是实时中断还是异步信号。

6.4.15 Int_IsrDisConnect：注销中断响应函数

void Int_IsrDisConnect(ufast_t ufl_line)
头文件：
os_inc.h
参数：
ufl_line：中断线号。
返回值：
无。
说明：
注销中断线 ufl_line 的中断响应函数。

6.4.16 Int_EvttConnect：注册异步事件

void Int_EvttConnect(ufast_t ufl_line,uint16_t my_evtt_id)
头文件：
os_inc.h
参数：
ufl_line：中断线号；
my_evtt_id：被关联的事件类型 id。
返回值：

无。
说明：
将中断线信号转化为系统的异步事件。即设置中断线触发的事件类型。当中断线信号发生后，系统首先跳往中断响应函数（如果没有设置，则没有）处理中断,然后弹出这个事件类型。同时，系统会将中断响应函数结果（u32 类型）和中断线号（u32 类型）作为输入参数传递给事件处理函数（事件线程）。
警告：只有异步信号才能关联事件类型。

6.4.17 Int_EvttDisConnect：注销异步事件

void　Int_EvttDisConnect (ufast_t ufl_line)
头文件：
os_inc.h
参数：
ufl_line：中断线编号。
返回值：
无。
说明：
取消中断线与事件类型的关联关系。

6.4.18 Int_GetRunLevel：查询嵌套深度

u32　Int_GetRunLevel(void)
头文件：
os_inc.h
参数：
无。
返回值：
异步信号嵌套深度。
说明：
查询当前异步信号的嵌套深度。

6.4.19 Int_AsynSignalSync：异步信号同步

bool_t　Int_AsynSignalSync (ufast_t ufl_line)
头文件：
os_inc.h
参数：
ufl_line：被同步的中断线号，必须是异步信号类型。
返回值：
设置成功返回 true；失败则返回 false。
说明：

调用后，将阻塞正在处理的事件的线程，直到指定的中断线的中断发生、响应并返回，然后才激活线程。不管中断线原来状态如何，调用本函数将导致中断线被使能(是直接使能，不是调用 Int_RestoreAsynLine)，并在返回后恢复禁止状态。
只有异步信号才允许被同步，实时中断是不允许的。
调用前，应该确保相应中断线处于禁止状态。但可以连接 ISR、evtt 等。
如果没有连接 ISR，应该把中断线的清中断类型设为 *CN_INT_CLEAR_PRE* 或者 *CN_INT_CLEAR_POST*。如果连接了 ISR 且清中断类型设为 *CN_INT_CLEAR_USER* 的话，ISR 中必须清中断。
如果连接了 ISR 并被调用，则被阻塞信号从阻塞恢复运行后，g_ptEventRunning-> event_result 将设置为 ISR 函数的返回值。
如果调用本函数时，被等待的中断已经挂起，则无论 ISR 是否被连接，都不会被调用，且函数理解作为同步成功返回，不会被阻塞。如果本函数调用过程中或之前，中断线信号已经发生了，则认为同步成功。

6.5 移植时需实现的接口

以下函数是在移植时实现的。具体实现的细节需依据不同的硬件平台特性。

6.5.1 Int_SetPrio：设置中断线优先级

bool_t Int_SetPrio(ufast_t ufl_line, u32 prio)
头文件：
os_inc.h
参数：
ufl_line：被操作的中断线编号；
prior：新的优先级。
返回值：
设置成功返回 true；失败则返回 false。
说明：
设定某中断线的嵌套优先级。
不同的硬件，优先级的含义也不同，有些中断控制器的优先级是抢占式的，即高优先级的中断能嵌套低优先级中断，像 cortex-m3；另一些中断控制器的优先级是查询式的，两个中断一起发生时，先服务高优先级的，例如 s3c2440。如果硬件不支持中断优先级，BSP 设计时直接返回 false 即可。
警告：此函数实现与硬件环境相关，在系统移植时确定。

6.5.2 Int_HighAtomStart/Int_HighAtomEnd 高级原子操作

进入高级原子操作：
atom_high_t Int_HighAtomStart(void)
离开原子操作：
void Int_HighAtomEnd(atom_high_t high)

头文件：
os_inc.h
参数：
high：总中断的开关状态，是调用 Int_HighAtomStart 函数的返回值。
返回值：
Int_HighAtomStart 函数的返回值表示总中断的开关状态。

说明：
高级原子操作是指期间不容许任何原因打断的操作，一般是因为特定的硬件要求，比如说用 gpio 产生一个 1uS 宽度的脉冲。从进入高级原子操作到离开的期间，所有中断被禁止。
Int_HighAtomStart 函数读当前总中断状态，然后禁止总中断。
Int_HighAtomStart 和 Int_HighAtomEnd 是姊妹函数，必须配套使用。
原子操作的用法：
源码 6-1 使用高级原子操作

```
void HighAtomExample(void)
{
    atom_high_t   high_atom;
    high_atom = Int_HighAtomStart();
    //硬件要求，必须在关闭总中断条件下操作 CTPR 寄存器
    *(u32*)(cn_core_ctpr_addr) = cn_prior_core_asyn_disable;
    __asm_disable_tick_int();
    Int_HighAtomEnd(high_atom);
    return;
}
```

上述代码是典型的受硬件约束而需要使用高级原子操作的。
原子操作函数还可以嵌套使用，方法如下：
源码 6-2 嵌套使用原子操作

```
atom_high_t   high_atom1, high_atom2;
high_atom1 = Int_HighAtomStart();
do something 1;
high_atom2 = Int_HighAtomStart();
do something 2;
Int_HighAtomEnd(high_atom2);
Int_HighAtomEnd(high_atom1);
```

在原子操作包包围的代码内，不允许调用“int_save_asyn_signal——int_restore_asyn_signal”，也不允许调用任何可能导致阻塞的 API。禁止总中断后，所有中断无法响应，调度也被禁止，因此禁止时间应该尽可能地短，最好不要在禁止期间调用任何 API。
警告：原子操作是系统提供的快速高效手段，但却很暴力，通过原子操作（Int_HighAtomStart 或者 Int_LowAtomStart）进入禁止中断的状态，是不可以发生调度的，然而调度器却不知道，调用 Djy_QuerySch 将返回“允许”状态，你在这种时候如果调用可能引起阻塞的操作，是自讨苦吃，结果将不可预知。
1. 此函数实现与硬件环境相关，在系统移植时确定。
2. 在原子操作过程中，不要调用会引发事件阻塞的操作（函数），一旦发生阻塞，其结果将会不可预知。

6.5.3 Int_LowAtomStart/Int_LowAtomEnd：低级原子操作

低级原子操作进入
atom_low_t　Int_LowAtomStart(void)
低级原子操作离开
void　Int_LowAtomEnd(atom_low_t low)
头文件：
os_inc.h
参数：
函数使用方法参见高级原子操作部分，不再赘述。
返回值：
函数使用方法参见高级原子操作部分，不再赘述。
说明：
低级原子操作跟高级原子操作类似，高级原子操作控制的是总中断开关，低级原子操作控制的是异步信号总开关，其他完全一样，使用方法也一样。
DJYOS 中 tick 是通过异步信号实现的，低级原子操作期间 tick 自然也不走了。
警告：此函数实现与硬件环境相关，在系统移植时确定。

6.5.4 Int_SettoAsynSignal：设置为异步信号

bool_t　Int_SettoAsynSignal(ufast_t ufl_line)
头文件：
os_inc.h
参数：
ufl_line：中断线号。
返回：
设置成功返回 true；失败则返回 false。
说明：
将中断线 ufl_line 设置为异步信号。如果此中断线正在响应，则设置在系统处理完此次中断线后才会生效。
警告：此函数实现与硬件环境相关，在系统移植时确定。

6.5.5 Int_SettoReal：设置为实时信号

bool_t　Int_SettoReal (ufast_t ufl_line)
头文件：
os_inc.h
参数：
ufl_line：中断线号。
返回：
设置成功返回 true；失败则返回 false。
说明：

将中断线 ufl_line 设置为实时信号。如果此中断线正在响应，则设置在系统处理完此次中断线后才会生效。
警告：此函数实现与硬件环境相关，在系统移植时确定。

6.5.6 Int_ContactAsynSignal：启动异步信号

void Int_ContactAsynSignal(void)
头文件：
int_hard.h
参数：
无。
返回：
无。
说明：
启动异步信号类型的中断线。
函数与 Int_CutAsynSignal 配对使用。
警告：此函数实现与硬件环境相关，在系统移植时确定。

6.5.7 Int_CutAsynSignal：禁止异步信号

void Int_CutAsynSignal(void)
头文件：
int_hard.h
参数：
无。
返回：
无。
说明：
禁止所有异步信号类型的中断线。
警告：此函数实现与硬件环境相关，在系统移植时确定。

6.5.8 Int_ContactTrunk：开启中断功能

void Int_ContactTrunk(void)
头文件：
int_hard.h
参数：
无。
返回：
无。
说明：
开启系统的中断功能。

本函数与 Int_CutTrunk 配对使用。

6.5.9 Int_CutTrunk：关闭中断功能

void　Int_CutTrunk(void)
头文件：
int_hard.h
参数：
无。
返回：
无。
说明：
禁止系统的中断功能。
函数与 Int_ContactTrunk 配对使用。
警告：此函数实现与硬件环境相关，在系统移植时确定。

6.5.10 Int_ContactLine：开启中断线

bool_t　Int_ContactLine(ufast_t ufl_line)
头文件：
int_hard.h
参数：
ufl_line：中断线号。
返回：
无。
说明：
开启中断线。
本函数与 Int_CutLine 配对使用。
警告：此函数实现与硬件环境相关，在系统移植时确定。

6.5.11 Int_CutLine：禁止中断线

bool_t　Int_CutLine(ufast_t ufl_line)
头文件：
int_hard.h
参数：
ufl_line：中断线号。
返回：
无。
说明：
禁止中断线。
函数与 Int_ContactLine 配对使用。

警告：此函数实现与硬件环境相关，在系统移植时确定。

6.5.12 Int_ClearLine：清中断线

bool_t Int_ClearLine (ufast_t ufl_line);
头文件：
os_inc.h
参数：
ufl_line，被清的中断线编号。
返回：
操作成功返回 true；失败则返回 false。
说明：
清中断线。
警告：此函数实现与硬件环境相关，在系统移植时确定。

6.5.13 Int_QueryLine：查询中断线状态

bool_t Int_QueryLine(ufast_t ufl_line)
头文件：
os_inc.h
参数：
ufl_line：中断线号。
返回值：
中断线信号已发生返回 true；否则返回 false。
说明：
查询中断线信号是否已经发生。
警告：此函数实现与硬件环境相关，在系统移植时确定。

6.5.14 Int_TapLine：软中断

bool_t Int_TapLine(ufast_t ufl_line)
头文件：
os_inc.h
参数：
ufl_line：中断线号。
返回值：
成功软件触发中断返回 true；否则返回 false。
说明：
用软件的方法触发一次中断，就像该中断真实发生了一样。这个功能依赖于 CPU，绝大多数 CPU 的硬件允许软件触发，但也有些 CPU 不允许，例如 freescale 的 p1020。
警告：此函数实现与硬件环境相关，在系统移植时确定。

6.5.15 Int_EnableNest：使能中断线嵌套

bool_t　Int_EnableNest(ufast_t ufl_line)
头文件：
os_inc.h
参数：
ufl_line：中断线号。
返回值：
使能成功返回 true；失败则返回 false。
说明：
使能中断线嵌套功能。与 Int_DisableNest 配对使用。
使能后，相应的中断服务期间，可以会被别的中断线抢占。但是否支持嵌套功能，由 BSP 设计者根据硬件的支持情况和实际工程需要决定。设计者可以设计成只允许实时中断嵌套，也可以只允许异步信号嵌套。
在使能嵌套的情况下，嵌套方式也可能有多种，例如三星的 S3C2440，只要嵌套被允许，则低优先级的中断也可以嵌套高优先级的中断，而 freescale 的 P1020，则只有更高优先级的才可以嵌套低优先级的。
这里所说的嵌套，可以是实时中断嵌套实时中断，也可以是异步信号嵌套异步信号，或者是实时中断嵌套异步信号，这些功能由 BSP 设计者保证。
警告：此函数实现与硬件环境相关，在系统移植时确定。

6.5.16 Int_DisableNest：禁止中断嵌套

bool_t　Int_DisableNest (ufast_t ufl_line)
头文件：
os_inc.h
参数：
ufl_line：中断线号。
返回值：
使能成功返回 true；失败则返回 false。
说明：
禁止中断线嵌套功能。
与 Int_EnableNest 配对使用。使能后，相应的中断服务期间，可以会被别的中断线抢占。但是否支持嵌套功能，由 BSP 设计者根据硬件的支持情况和实际工程需要决定。设计者可以设计成只允许实时中断嵌套，也可以只允许异步信号嵌套。
在使能嵌套的情况下，嵌套方式也可能有多种，例如三星的 S3C2440，只要嵌套被允许，则低优先级的中断也可以嵌套高优先级的中断，而 freescale 的 P1020，则只有更高优先级的才可以嵌套低优先级的。
这里所说的嵌套，可以是实时中断嵌套实时中断，也可以是异步信号嵌套异步信号，或者是实时中断嵌套异步信号，这些功能由 BSP 设计者保证。
警告：此函数实现与硬件环境相关，在系统移植时确定。

第7章 紧急代码

想当年，MCU 普遍比较简单，内存容量不大，程序也比较短小，上电/复位后的初始化时间不长。这段时间内，即使系统对外界完全不响应，也关系不大，
待续……

第8章 内存管理

应用程序所占据或使用的内存，有两种来源。一种是静态分配，用户编程时定义全局或静态变量等，它们由编译器在编译时确定和分配。另一种是动态分配，即程序在运行过程中申请内存，由系统的内存管理机制来确定和分配。。前后两种的本质区别是应用程序获得内存的时刻前者是编译时，后者是运行时。
对于第二种来源，其核心内容是 DJYOS 的内存管理机制。DJYOS 有 3 种动态内存分配策略，分别是：
1. “准静态”分配，这种分配方式虽然是系统运行时分配，但执行效率跟静态分配类似，比编译器静态分配多了对齐损耗而已。
2. “块相联”分配，它把整个内存划分为一个一个的块，一次申请可以分配一块或多块。
3. “固定块”分配，它把大块内存划分由规格化的小块组成的“内存池“，每次申请分配一个规划化的小块。
“准静态”和“块相联”目前采用统一的用户接口，对应用程序或用户而言，这两种分配策略是透明的。那么，什么时候按照准静态方式，什么时候按照块相联方式分配内存呢？如果在 module-trim.c 中，有一行代码：
Heap_DynamicModuleInit(0); //自此 malloc 函数执行块相联算法。
如果没有注释掉这句，则在调用这句之前，执行准静态分配，调用之后，执行块相联分配。如果注释掉了，则一直执行准静态分配。
图 8-1 显示了静态分配和 DJYOS 的“准静态”和“块相联”分配策略的不同。采用这种不同内存分配策略，申请两块 101 字节大小的内存，在内存分配上的异同点。其中在“准静态”分配方式中，假设系统要求内存按 8 字节对齐。

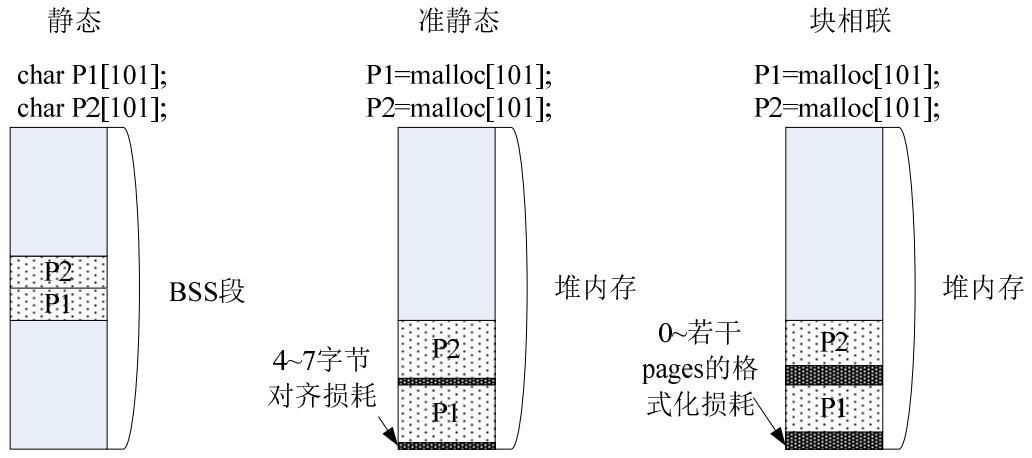


图 8-1 静态分配、准静态分配、块相联分配的效果

8.1 “准静态”分配

“准静态”分配是在系统初始化时，从堆中分配任意尺寸的内存。由于有些模块，在编译时无法确定要多少内存，需要模块运行初始化时才能确定。类似这种模块，虽然它自申请内存直至运行结束都不会释放内存，但让其使用静态定义的方式分配内存仍然是不合理的。相对而言，这种模块，采用准静态分配策略，既高效又快速。“准静态”分配法，从当前堆底开始切割，要多少内存就切多少，不浪费，但释放时，只能按照后申请先释放的原则，否则会照成内存泄露，否则会造成内存泄漏。

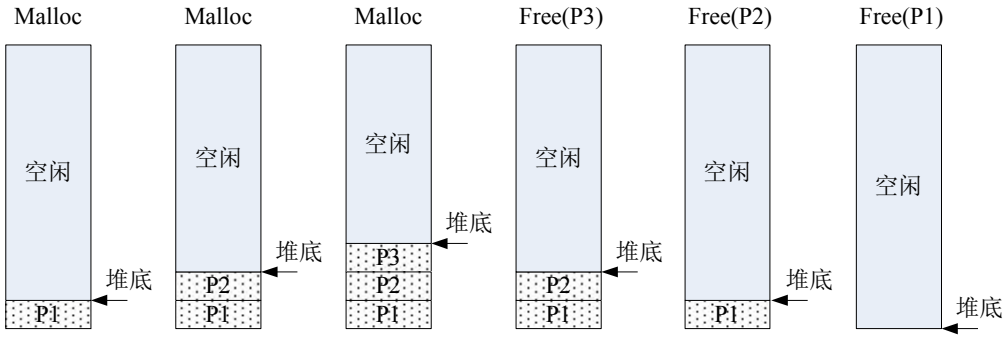


图 8-2 准静态分配和释放示意图

如图 8-2 所示，“准静态”分配虽然可以高效利用堆空间，无需查找空闲内存，执行速度很快，但释放必须严格按照分配的逆序进行。如果不按照逆序，先 free(P1)再 free(P2)的话，表面上 free 能“正常”返回，但实际上 P1 这块内存被泄漏掉了。

8.2 “块相联”分配

“块相联”分配是将整个内存划分为 2 个 n 次方个内存页，再通过独特地机制来管理内存。虽然这样做会使内存的使用率低一些。但是这个分配策略的执行速度很快，而且最重要的是，它的执行时间是确定的，比如一个总共 4GB，以 1KB 为一页的内存系统中，查找 1 个空闲页，在 32 位 cpu 中，最多需要 5 次搜索（共 5 次比较、5 次乘法和 5 次加法）就可以完成。“块相联”分配策略的对内存页的格式“损耗”虽然是不确定的，不过，悄悄告诉你一个秘密，这些所谓的“损耗”，其实就像你去银行取钱，银行多给了你钱，你拿去也是可以花的，而且，M_CheckSize 函数还能告诉你总共给了你多少钱。不要以为银行这么傻，多给的钱，也是会从你的账户中扣掉的。

8.3 “固定块”分配

“固定块”分配策略，是一种内存池分配技术。可以形象地描述为“批发”与“零售”的关系。不同于一般系统的固定块分配法，系统只允许“一次批发，卖完即止”。DJYOS 允许卖完后，再次批发，甚至允许开店时（内存池初始化时）不进货，等有人买时再批发。零风险的买卖，值得做吧。

版权所有，转载必须注明出处，注明：来自 www.djyos.com 字样。

8.4 “准静态”和“块相联”相关 API

如上所述“准静态”和“块相联”内存管理策略均使用统一的用户接口。至于系统运行时采用的哪个分配策略管理内存，请参见 Heap_StaticModuleInit 和 Heap_DynamicModuleInit。

8.4.1 Heap_StaticModuleInit：“准静态”初始化

ptu32_t Heap_StaticModuleInit(ptu32_t para)
头文件：
stdlib.h
参数：
para：参数保留。
返回值：
1。
说明：
“准静态”内存管理策略的初始化。扫描系统内存（堆）建立基于“准静态”策略的内存管理结构，注册相关实现函数。

8.4.2 Heap_DynamicModuleInit：“块相联”初始化

ptu32_t Heap_DynamicModuleInit(ptu32_t para)
头文件：
stdlib.h
参数：
para：参数保留。
返回值：
1。
说明：
“块相联”内存管理策略的初始化。扫描系统内存（堆）建立基于“块相联”策略的内存管理结构，注册相关实现函数。

8.4.3 M_Malloc：申请内存

void *M_Malloc (ptu32_t size, uint32_t timeout)
头文件：
stdlib.h
参数：
size：申请内存块的大小，以字节计。
timeout：等待时间。申请内存（调用函数）时，内存不足，事件（调用者）将被阻塞并等待其他事件释放内存。如果直至 timeout 时间之后，内存仍不足则申请失败。等待时间系统自动向上取整为整数个 ticks。

返回值：
申请成功返回内存首指针；失败则返回 NULL。
说明：
从堆中分配 size 大小的局部内存空间。由于 DJYOS 系统使用块相联分配算法，size 将被向上取整为一个规格化大小的块，规格化方法参见《都江堰操作系统原理与实现》中内存管理章节介绍。如果系统没有满足条件的内存块，且 timeout 不等于 0，则阻塞线程，直到 timeout 时间到或者有其他线程释放内存致使有合适的内存块。
由于规格化，所获得的内存很可能超过 size 参数，调用 M_CheckSize 函数可获取实际申请到的内存的大小。

8.4.4 M_Realloc：拓展内存

void * M_Realloc(void *p, ptu32_t NewSize,u32 Timeout)
头文件：
stdlib.h
参数：
p：原内存首地址。
NewSize：申请内存块的新尺寸，以字节计。
Timeout：等待时间。申请内存（调用函数）时，内存不足，事件（调用者）将被阻塞并等待其他事件释放内存。如果直至 Timeout 时间之后，内存仍不足则申请失败。等待时间系统自动向上取整为整数个 ticks。
返回值：
拓展成功返回内存首指针；失败则返回 NULL。
说明：
先判断当前内存 p 大小是否已经满足新设定的尺寸要求。如果满足，直接返回原内存地址。如果不满足，则先按照 NewSize 指定的大小分配空间，将原有内存中的数据拷贝到新分配的空间，然后返回新申请内存首地址。
警告：在“块相联”内存管理策略中，如果拓展内存成功，系统将释放原申请内存空间。而在“准静态”策略中，原内存不释放。

8.4.5 free：释放内存

void　free(void *pl_mem)
头文件：
stdlib.h
参数：
p：内存首地址。
返回值：
无。
说明：
释放申请的内存。
警告：在“准静态”内存管理策略中，释放的内存必须是最新（最近一次）申请的内存。

8.4.6 M_MallocHeap：申请指定堆内存

void *M_MallocHeap(ptu32_t size, struct tagHeapCB *Heap, u32 timeout)
头文件：
stdlib.h
参数：
size：申请内存大小。
Heap：堆句柄（控制块）。
timeout：等待时间。申请内存（调用函数）时，内存不足，事件（调用者）将被阻塞并等待其他事件释放内存。如果直至 Timeout 时间之后，内存仍不足则申请失败。等待时间系统自动向上取整为整数个 ticks。
返回值：
申请成功返回内存首指针；失败则返回 NULL。
说明：
从指定的 Heap 句柄（控制块）指定的堆空间中申请 size 大小的内存。

8.4.7 M_MallocLc：申请“事件内存”

void *M_MallocLc(ptu32_t size, u32 timeout)
头文件：
stdlib.h
参数：
size：申请内存大小。
timeout：等待时间。申请内存（调用函数）时，内存不足，事件（调用者）将被阻塞并等待其他事件释放内存。如果直至 Timeout 时间之后，内存仍不足则申请失败。等待时间系统自动向上取整为整数个 ticks。
返回值：
申请成功返回内存首指针；失败则返回 NULL。
说明：
申请“事件内存”。所谓“事件内存”是指所申请的这种内存的生命周期与事件的生命周期是同步的，当事件消亡的时候，仍然没有释放的内存，将被强制释放避免内存泄漏。然而，强制释放是很消耗时间的，实时软件的设计者在内存使用完毕后，应该及时调用 M_Free 函数释放内存，必须避免强制释放这种事情发生。欲在事件生存期外继续使用分配的内存，请调用 M_Malloc 函数分配全局内存块。

8.4.8 M_MallocLcHeap：申请指定堆“事件内存”

void *M_MallocLcHeap(ptu32_t size, struct tagHeapCB *Heap, u32 Timeout)
头文件：
stdlib.h
参数：

size：申请内存大小。
Heap：堆句柄（控制块）。
Timeout：等待时间。申请内存（调用函数）时，内存不足，事件（调用者）将被阻塞并等待其他事件释放内存。如果直至 Timeout 时间之后，内存仍不足则申请失败。等待时间系统自动向上取整为整数个 ticks。
返回值：
申请成功返回内存首指针；失败则返回 NULL。
说明：
从指定的 Heap 句柄（控制块）指定的堆空间中申请 size 大小的“事件内存”。“事件内存”说明参见 M_MallocLc 函数。

8.4.9 M_FreeHeap：释放指定堆内存

void *M_FreeHeap(void *pl_mem, struct tagHeapCB *Heap)
头文件：
stdlib.h
参数：
pl_mem：内存首地址。
Heap：堆句柄（控制块）。
返回值：
申请成功返回内存首指针；失败则返回 NULL。
说明：
释放从指定的 Heap 句柄（控制块）指定的堆空间中申请的内存空间。
警告：在“准静态”内存管理策略中，释放的内存必须是最新（最近一次）申请的内存。

8.4.10 M_FormatSize：换算“规格尺寸”

ptu32_t M_FormatSize(ptu32_t size)
头文件：
stdlib.h
参数：
size：内存空间尺寸。
返回值：
规格尺寸。
说明：
将内存空间尺寸换算系统的“规格尺寸”。所谓“规格尺寸”是指，如果你请求分配 size 尺寸的内存，系统实际分配给你的内存尺寸。

8.4.11 M_FormatSizeHeap：换算指定堆“规格尺寸”

ptu32_t M_FormatSizeHeap(ptu32_t size, struct tagHeapCB *Heap)
头文件：

stdlib.h
参数：
size：内存空间尺寸。
Heap：堆句柄（控制块）。
返回值：
规格尺寸。
说明：
将内存空间尺寸换算指定堆空间的“规格尺寸”。所谓“规格尺寸”是指，如果你请求分配 size 尺寸的内存，系统实际分配给你的内存尺寸。

8.4.12 M_GetMaxFreeBlock：查询最大块

ptu32_t *M_GetMaxFreeBlock(void)
头文件：
stdlib.h
参数：
无。
返回值：
最大块尺寸（字节数）。
说明：
查询系统最大内存块尺寸。

8.4.13 M_GetMaxFreeBlockHeap：查询指定堆最大块

ptu32_t M_GetMaxFreeBlockHeap(struct tagHeapCB *Heap)
头文件：
stdlib.h
参数：
Heap：堆句柄（控制块）。
返回值：
最大块尺寸（字节数）。
说明：
查询指定堆空间中最大内存块尺寸。

8.4.14 M_GetFreeMem：查询可用内存总量

ptu32_t *M_GetFreeMem(void)
头文件：
stdlib.h
参数：
无。
返回值：

可用内存总量（字节数）。
说明：
查询系统可用内存总量（字节数）。

8.4.15 M_GetFreeMemHeap：查询指定堆可用内存总量

ptu32_t　*M_GetFreeMemHeap(struct tagHeapCB *Heap)
头文件：
stdlib.h
参数：
Heap：堆句柄（控制块）。
返回值：
可用内存总量（字节数）。
说明：
查询指定堆空间的可用内存总量（字节数）。

8.4.16 M_CheckSize：查询“规划”空间

ptu32_t　M_CheckSize(void *mp)
头文件：
stdlib.h
参数：
mp：内存首地址。
返回值：
内存大小。
说明：
查询所申请的内存 mp 的实际大小。由于内存分配函数要对 size 参数做规格化调整，即将用户申请的内存换算成系统“规格化”大小，再分配给用户，malloc 函数实际获得的内存块可能比申请量大许多，本函数查询获得的内存块的实际可用尺寸。

8.5 “固定块”相关 API

8.5.1 Mb_CreatePool：创建内存池

```
struct mem_cell_pool    *Mb_CreatePool(void *pool_original,
                                       u32 init_capacital,
                                       u32 cell_size,
                                       u32 increment,
                                       u32 limit,
                                       char *name)
```

头文件：
os_inc.h

参数：
pool_original：最初由用户提供的内存空间，不提供设为 NULL。
init_capacital：原始内存池的尺寸，以块为单位，可以是 0。
cell_size：内存块尺寸，若系统有对齐要求，必须为指针长度的整数倍，且最小为两倍指针长度。
increment：当本参数不为零时，所创建的内存池被一但被耗光，会从堆中补充获取 increment 个的内存块。直至内存池再被消耗完，再次获取。注意，内存池扩大后，即使用户调用 mb_free 释放了内存，但除非释放了内存池中的全部内存，否则新增的内存不会被收回。
limit：如果 increment != 0，limit 限制内存池的最大块数，以防其无限制地增加，导致内存耗尽。
name：设置内存池的名称，表示名称的字符串不能是局部变量，无名称则设置为 NULL。
返回值：
指向所创建的内存池的指针。
说明：
初始化一个内存池，原始内存池的内存空间由用户提供。一般是定义一个大数组，这个数组的大小是 init_capacital 乘以 cell_size。如果系统有对齐要求，则起始地址至少要按照指针类型对齐，cell_size 也应该是指针长度的整数倍，不要用 char pool_original[n * cell_size]的方式定义内存，在要求对齐访问的系统中，创建内存池很可能失败，即使不失败，也存在内存访问对齐错误的可能。正确的方法是：struct tar pool_original[n];

8.5.2 Mb_CreatePool_r：创建可靠内存池

```
struct mem_cell_pool    *Mb_CreatePool_r(struct mem_cell_pool *pool,
                                         void *pool_original,
                                         u32 init_capacital,
                                         u32 cell_size,
                                         u32 increment,
                                         u32 limit,
                                         char *name)
```

头文件：
os_inc.h
参数：
pool：内存池指针（控制块）；
pool_original：最初由用户提供的内存空间，不提供设为 NULL。
init_capacital：原始内存池的尺寸，以块为单位，可以是 0.
cell_size：内存块尺寸，若系统有对齐要求，必须为指针长度的整数倍，最小为两倍指针长度.
increment：当本参数不为零时，所创建的内存池被一但被消耗，会从堆中格外获取 increment 个的内存块。直至内存池再被消耗完，再次获取。注意，内存池扩大后，即使用户调用 mb_free 释放了内存，但除非释放了内存池中的全部内存，否则新增的内存不会被收回。
limit：如果 increment != 0，limit 限制内存池的最大块数，以防无限制地增加，导致内存耗尽。
name：设置内存池的名称，表示名称的字符串不能是局部变量，无名称则设置为 NULL。
返回值：

指向所创建的内存池的指针。
说明：
初始化一个内存池，与 mb_create_pool 不同的是，内存池控制块 pool 由调用者提供，其他与 mb_create_pool 完全一致。高可靠性的应用中，不应该使用动态分配的方式，静态定义更可靠，然后把指针传递过来。内核中使用的内存池，都是使用静态定义的。应用程序配置 cfg_mem_pools 时，只需要考虑自己的需求就可以了。

8.5.3 Mb_DeletePool：删除内存池

bool_t Mb_DeletePool(struct mem_cell_pool *pool)
头文件：
os_inc.h
参数：
pool：内存池指针（空间块）。
返回值：
删除成功返回 true；失败则为 false。
说明：
删除一个内存池，当某内存池不再需要时，可调用本函数。本函数只清理了内存池的信号量和资源结点，内存池缓冲区是调用者提供的，理应由调用者清理。对于一个可扩容的内存池，如果发生了扩容行为，新增加的内存也将随之释放回堆中。

8.5.4 Mb_DeletePool_r：删除可靠内存池

bool_t Mb_DeletePool_r(struct mem_cell_pool *pool)
头文件：
os_inc.h
参数：
pool：内存池指针（控制块）。
返回值：
删除成功返回 true；失败则为 false。
说明：
本函数与 Mb_CreatePool_r 对应，功能与 Mb_DeletePool 一致。

8.5.5 Mb_Malloc：申请内存

void *Mb_Malloc(struct mem_cell_pool *pool,uint32_t timeout);
头文件：
os_inc.h
参数：
pool：内存池指针。
timeout：设置申请内存时的超时间，单位为微秒，同时设定值会自动向上取整为 CN_CFG_TICK_US 的整数倍。timeout = CN_TIMEOUT_FOREVER，代表无限等待。timeout

= 0，代表立即返回。
返回值：
申请成功返回内存地址；否则返回 NULL。
说明：
从指定的内存池中申请一块内存。注意，连续调用本函数，并不能保证连续申请的内存地址是连续的。如果指定的内存池允许扩容，则该内存池耗尽时，将从堆中获取一块内存添加到内存池中。但从堆中扩容内存到内存池中的操作时单向的，一经扩容永不释放，直到内存池被删除。

8.5.6 Mb_Free：释放内存

void Mb_Free(struct mem_cell_pool *pool,void *block);
头文件：
os_inc.h
参数：
block：待释放的内存块指针；
pool：目标内存池。
返回值：
无。
说明：
释放内存，把使用完毕的内存块放回指定的内存池，内存池和内存块必须匹配，否则会发生不可预料的错误，新释放的块链接到 free_list 队列中，而不是放回连续池，也不重新返回系统堆。

8.5.7 Mb_QueryFree：查询内存池空闲量

u32 Mb_QueryFree(struct mem_cell_pool *pool);
头文件：
os_inc.h
参数：
pool：目标内存池。
返回值：
空闲内存块数。
说明：
查询内存池还有多少空闲内存块，对于一个不可扩容的内存池，返回结果就是可供分配的块数；对一个可扩容的，返回的是当前池中的空闲块数。

8.5.8 Mb_QueryCapacital：查询内存池总量

u32 Mb_QueryCapacital(struct mem_cell_pool *pool)
头文件：
os_inc.h

参数：
pool：目标内存池。
返回值：
内存池总内存块数。
说明：
查询内存池总共有多少内存块，对于一个可扩容的内存池，其总容量也包括扩容部分，而不是允许扩容的上限。

第9章 锁

锁的用途是保证临界区数据的安全访问，或者线程与线程、线程与中断间的同步。临界区数据指的是不同线程或者中断可能同时访问的数据区。这里要注意并不是所有的共享数据区都是临界区，只有那些有可能被同时访问的区域，才称作临界区。DJYOS 只支持线程和异步信号间的同步，不支持线程和实时中断间的同步。另外，线程和异步信号间的同步还可以使用“异步信号同步”功能。DJYOS 支持 3 种锁：
1. 调度锁，锁住调度，等同于禁止中断。
2. 信号量，适用于允许一个或者有限多个用户同时访问的场合。信号量允许线程之间或中断与线程间的交叉请求和释放，即 A 线程请求信号量——B 线程释放信号量。
3. 互斥量，适用于单一资源且需要优先级继承的场合。互斥量的请求者和释放者必须是同源，即同一事件（线程）或者中断处理函数。
信号量（互斥量）控制块使用相同的内存池，称为锁控制块，采用固定块动态分配法。创建信号或者互斥量时，信号或互斥量总数受 module-trim.c 中 gc_u32CfgLockLimit 常量限制。另外，为了满足 OSEK、MISRA 等标准的要求，DJYOS 还提供另一对函数用于创建信号或互斥量：Lock_SempCreate_r、Lock_MutexCreate_r，这对函数是静态地创建信号或者互斥量，即要求调用者提供（指定）锁控制块空间，其数量不受 gc_u32CfgLockLimit 限制。
信号量和互斥量的共同点：

	信号量	**互斥量**
信号数	1~0xffffffff	1
初始信号数	任意	1
排队顺序	优先级或 FIFO	优先级
请求和释放①	可在不同的上下文中	必须在相同上下文中
优先级继承	不支持	支持多级优先级继承
优先级置顶	不支持	支持
嵌套请求②	请求数超过可用信号灯数时，会死锁（假设超时 timeout = cn_timeout_forever）	可嵌套请求
共同适用的场合	非计数型临界区访问保护	
适用场合	1. 线程间同步； 2. 线程间通信； 3. 计数型临界区访问。	1. 须防止优先级翻转的非计数型临界区访问保护； 2. 用于自动调整优先级。

注：
①相同上下文的意思是，在同一个线程中，或者在异步信号中。
②一个线程连续多次请求然后连续多次释放，称为嵌套请求。

信号量和互斥量有许多相似的特性，都能够提供临界区安全保护，有许多场合既可用信号量实现，又可以用互斥量实现，但他们在使用上，还是有区别的。
同步和线程间（线程与异步信号间）通信是信号量的拿手绝活，典型用途：
1. A 线程阻塞于某信号量（pend），B 线程（或异步信号）在合适的时机释放信号量（post），A 线程得以继续运行。
2. A 线程要访问某临界资源，先获取保护该资源的信号量，访问完毕后释放（post）。期间，如果 B 线程也来取该信号量，将会被阻塞，直到 A 释放该线程。
3. A 线程要读取数据，但该数据尚未被生产出来，A 线程将 pend 并阻塞在信号量上，B 线程（或异步信号中断）生产数据后，释放（post）信号量。A 线程解除阻塞并访问数据。
4. 某资源允许 n 个入口，可使用信号量的计数功能，例如使用信号量保护内存池。
信号量的用途要比互斥量广泛得多，除优先级继承功能外，互斥量的所有功能，信号量都可以涵盖。
在资源队列中，信号量和互斥量各有一颗资源树，其中信号量的树根名叫“semaphore”，互斥量的树根名叫“mutex”。
如果 shell 组件没有被裁掉，可以在 shell 中用“lock”命令查看所有的信号量和互斥量的状态。

9.1 锁调度和锁中断

参见 6.4.1 节和 6.4.2 节。
特别注意，原子操作并不锁调度，千万不能在原子操作期间调用可能发生调度的函数，最好不要调用任何系统 API。

9.2 信号量 API

9.2.1 Lock_SempCreate：创建信号量

```
struct tagSemaphoreLCB    *Lock_SempCreate(u32 lamps_limit,
                                           u32 init_lamp,
                                           u32 sync_order,
                                           char *name)
```

头文件：
os_inc.h
参数：
semp_limit：设定可用信号灯的上限，设定值不能为零，semp_limit = CN_LIMIT_UINT32 则表示无上限，无限多。
init_lamp：初始可用信号灯数量，取值范围 0 ~ semp_limit；
sync_order：被信号阻塞事件的排列顺序。sync_order = CN_SEMP_BLOCK_FIFO 表示按申请获取信号的先后顺序排列，sync_order = CN_SEMP_BLOCK_PRIO 表示按事件的优先级排列；
name：信号量的名字，所指向的字符串不能定义为局部变量，无名称即为 NULL。
返回值：
新创建的信号量。

说明：
建立一个信号量，信号量须先创建再使用。可以创建的信号量和互斥量的数量总和（两者之和），受 module-trim.c 中 gc_u32CfgLockLimit 常量限制。

9.2.2 Lock_SempCreate_r：可靠创建信号量

```
struct tagSemaphoreLCB    *Lock_SempCreate_r(struct tagSemaphoreLCB *semp,
                                             u32 lamps_limit,
                                             u32 init_lamp,
                                             u32 sync_order,
                                             char *name)
```

头文件：
os_inc.h
参数：
semp：信号量控制块指针。
semp_limit：设定可用信号灯的上限，设定值不能为零，semp_limit = CN_LIMIT_UINT32 则表示无上限，无限多。
init_lamp：初始可用信号灯数量。
sync_order：被信号阻塞事件的排列顺序。sync_order = CN_SEMP_BLOCK_FIFO 表示按申请获取信号的先后顺序排列，sync_order = CN_SEMP_BLOCK_PRIO 表示按事件的优先级排列。
name：信号量的名称，所指向的字符串不能定义为局部变量，无名称即为 NULL。
返回值：
无。
说明：
可靠地创建一个信号量，与 Lock_SempCreate 函数不同的是，调用者须提供信号量控制块，高可靠性的应用中，不应该使用动态分配的方式，静态定义更可靠，然后把指针传递过来。内核中使用的信号量，都是使用 Lock_SempCreate_r 创建的，应用程序配置 gc_u32CfgLockLimit 时，只需要考虑自己的需求就可以了。

9.2.3 Lock_SempDelete：删除信号量

```
bool_t    Lock_SempDelete(struct tagSemaphoreLCB *semp);
```

头文件：
os_inc.h
参数：
semp：被删除的信号量
返回值：
删除成功返回 true；失败返回 false。
说明：
删除一个信号量，与 Lock_SempCreate 函数对应。

9.2.4 Lock_SempDelete_r：可靠删除信号

bool_t Lock_SempDelete_r(struct tagSemaphoreLCB *semp)
头文件：
os_inc.h
参数：
semp：被删除的信号量
返回值：
删除成功返回 true；失败返回 false。
说明：
删除一个信号量，与 Lock_SempCreate_r 函数对应。

9.2.5 Lock_SempPend：等待信号量

bool_t Lock_SempPend(struct tagSemaphoreLCB *semp, u32 timeout)
头文件：
os_inc.h
参数：
semp：信号量指针
timeout：设置等待信号时间，单位微秒，该值会被自动向上取整为 CN_CFG_TICK_US 的整数倍。Timeout = CN_TIMEOUT_FOREVER，表示无限等待。timeout = 0,表示立即返回。
返回值：
获取（申请）信号量成功返回 true；获取信号量失败，返回 false，如超时，信号量检查失败等。具体的错误类型，通过错误码来确定。
说明：
申请获取信号量的信号灯，如果信号量有效且有信号灯可用，则将可用信号灯总量减一，并成功返回 true。如果同一个线程反复请求（pend）一个信号量，每次请求都会造成信号灯数量减一，减到零的时候，将被阻塞。这种情况下，由于被阻塞的原因，是本线程早先拿走了信号量，故只能等待自己释放，而自己又被阻塞不能运行，将造成死锁。假如 timeout = CN_TIMEOUT_FOREVER，线程将永久失去响应。

9.2.6 Lock_SempPost：发送信号量

void Lock_SempPost(struct tagSemaphoreLCB *semp)
头文件：
os_inc.h
参数：
semp：信号量指针。
返回值：
无。
说明：
释放一个信号灯，可用信号灯增一。注意，当可用信号灯数量大于 lamps_limit 后，就不会

再增加。

9.2.7 Lock_SempQueryCapacital：查询信号量容量

u32 Lock_SempQueryCapacital(struct tagSemaphoreLCB *semp);
头文件：
os_inc.h
参数：
semp：被查询的信号量。
返回值：
信号量包含的信号灯总数。
说明：
查询一个信号量包含信号灯的总数。

9.2.8 Lock_SempQueryFree：查询可用信号数

u32 Lock_SempQueryFree(struct tagSemaphoreLCB *semp)
头文件：
os_inc.h
参数：
semp：被查询的信号量。
返回值：
信号灯总数。
说明：
查询一个信号量可用信号灯的数量，如果大于 0，调用 Lock_SempPend 函数能取得信号量。

9.2.9 Lock_SempCheckBlock：查询是否存在被阻塞事件

bool_t Lock_SempCheckBlock(struct tagSemaphoreLCB *Semp)
头文件：
os_inc.h
参数：
Semp：信号量指针。
返回值：
存在等待同步的事件返回 true；否则 fasle。
说明：
查询信号量的同步队列中是否有事件在等待。

9.2.10 Lock_SempSetSyncSort：设置信号量阻塞方式

void Lock_SempSetSyncSort(struct tagSemaphoreLCB *semp, u32 order)

头文件：
os_inc.h。
参数：
semp：被设置的信号量；
order：被信号阻塞事件的排列顺序。sync_order = CN_SEMP_BLOCK_FIFO 表示按申请获取信号的先后顺序排列，sync_order = CN_SEMP_BLOCK_PRIO 表示按事件的优先级排列。
返回值：
无。
说明：
设置被信号量阻塞的事件，在该信号量同步队列中的排队方式。该设置只影响设置之后新加入的事件，已在队列中的事件不受影响。

9.3 互斥量 API

9.3.1 Lock_MutexCreate：创建互斥量

struct tagMutexLCB *Lock_MutexCreate(char *name)
头文件：
os_inc.h。
参数：
name：互斥量名称，所指向的字符串不能定义为局部变量。无名称则为 NULL。
返回值：
新创建互斥量的指针。
说明：
信号量所占内存由系统从锁内存池中分配。互斥量须先创建再使用。创建的信号量和互斥量加起来的总数，受 config-prj.h 中的常量 cfg_locks_limit 限制。

9.3.2 Lock_MutexCreate_r：可靠创建互斥量

struct tagMutexLCB *Lock_MutexCreate_r(struct tagMutexLCB *mutex,
                                        char *name)
头文件：
os_inc.h
参数：
mutex：互斥量控制块指针。
name：互斥量名称，所指向的字符串不能定义为局部变量。无名称则为 NULL。
返回值：
新创建互斥量的指针。
说明：
静态地创建一个互斥量，与 mutex_create 函数不同的是，调用者须提供互斥量控制块，高可靠性的应用中，不应该使用动态分配的方式，静态定义更可靠，然后把指针传递过来。内核中使用的互斥量，都是使用 Lock_MutexCreate_r 创建的。应用程序配置 gc_u32CfgLockLimit 时，只需要考虑自己的需求就可以了。

9.3.3 Lock_MutexDelete：删除互斥量

bool_t Lock_MutexDelete(struct tagMutexLCB *mutex)
头文件：
os_inc.h
参数：
mutex：被删除信号量的指针。
返回值：
删除成功返回 true；否则返回 false。
说明：
删除一个信号量，与 mutex_create 函数对应。

9.3.4 Lock_MutexDelete_r：可靠删除互斥量

bool_t Lock_MutexDelete_r(struct tagMutexLCB *mutex)
头文件：
os_inc.h
参数：
mutex：被删除信号量的指针。
返回值：
删除成功返回 true；否则返回 false。
说明：
删除一个信号量，与 mutex _create_r 函数对应。

9.3.5 Lock_MutexPend：等待互斥量

bool_t Lock_MutexPend(struct tagMutexLCB *mutex, u32 timeout)
头文件：
os_inc.h
参数：
mutex：互斥量指针。
timeout：设置等待信号时间，单位微秒，该值会被自动向上取整为 CN_CFG_TICK_US 的整数倍。timeout = CN_TIMEOUT_FOREVER，表示无限等待。timeout = 0,表示立即返回。
返回值：
获取（申请）互斥量成功返回 true；获取信号量失败返回 false，如超时，信号量检查失败等。具体的错误类型，通过错误码来确定。
说明：
由于 mutex 只能被一个事件拥有，owner 成员保存了该事件指针，因此嵌套请求是允许的。即如果请求一个处于忙状态的互斥量，如果请求者是拥有者，则不会被阻塞，否则将被阻塞。

9.3.6 Lock_MutexPost：发送互斥量

void　Lock_MutexPost(struct tagMutexLCB *mutex)
头文件：
os_inc.h
参数：
mutex：互斥量指针。
返回值：
无。
说明：
释放互斥量，再次强调，互斥量只能被拥有者释放，所以互斥量不适用于线程间或线程与异步信号中断间同步。

9.3.7 Lock_MutexCheckBlock：查询是否存在被阻塞事件

bool_t　Lock_MutexCheckBlock (struct tagMutexLCB *mutex)
头文件：
os_inc.h
参数：
mutex：互斥量指针。
返回值：
存在等待同步的事件返回 true；否则 fasle。
说明：
查询互斥量的同步队列中是否有事件在等待。

9.3.8 Lock_MutexGetOwner：查询占用者

u16　Lock_MutexGetOwner(struct tagMutexLCB *mutex)
头文件：
os_inc.h
参数：
mutex：互斥量指针。
返回值：
互斥量被事件占有，返回该事件 ID；否则返回 CN_EVENT_ID_INVALID。
说明：
获取占有互斥量 mutex 的事件的 ID。

9.3.9 Lock_MutexQuery：查询互斥量状态

bool_t　Lock_MutexQuery(struct tagMutexLCB *mutex)
头文件：

os_inc.h
参数：
mutex：被查询的互斥量。
返回值：
互斥量可用返回 true；不可用则返回 false。
说明：
查询互斥量是否可用。

第10章　GUI

见《DJYOS 图形编程（上）》和《DJYOS 图形编程（下）》。

第11章　资源

11.1 资源管理模型

软件设计中，如何组织、存储和检索数据，是一个核心的问题，甚至可以说，只有理顺了这个问题，才有可能设计出高质量的软件。广义上，应用程序需要的一切数据都是资源。DJYOS 提供了一个简单易用的资源管理模块，和丰富的资源管理 API 函数，用于管理软件中用到的数据。在 DJYOS 资源管理模块的协助下，应用程序管理数据将变得非常简单。更主要的是，如果用户坚持用资源管理器管理所有数据，将使不同软件模块的实现细节上获得惊人的一致性。一个具有一定规模的软件，必然要划分为若干模块才能顺利开发，如果不加约束，各模块管理数据或资源的方法可能不一致。这种不一致性将增加软件的复杂度，增加软件维护的成本，而且，这样不可避免地会增加软件规模，增加编程和测试的工作量。若要在开发团队内和团队间进行人员调配，新人将花较多的时间熟悉新模块。而如果各模块代码有一致性，就会降低人员调配时交接工作的难度。DJYOS 的资源管理体系是开放的，单进程版本（si 和 dlsp 版本）中，操作系统和应用程序共用一个资源体系；多进程版本（mp 版本）中，每个进程都有独立的资源体系。

我们也许可以这样理解 DJYOS 的资源管理模型之间的。操作系统就像一片万物繁茂的大地。在这大地之上，开垦了一片的果园。果园里种了各类果树。树枝上挂满了果实。如果将系统的资源管理模型比作上述的果园的话，那么里面一棵棵果树就是基于这个模型培育出的资源树了。一棵资源树代表着系统中同类资源的集合。而树枝的结构体现了同类资源之间的层次关系。至于说这个果园里应该种哪些果树，或者说系统中需要构建哪些资源树，果实（数据）由谁采摘、使用，由系统工程师根据实际软硬件需要才能确定。就像果园的园长需要根据气候和土壤条件才能确定安排园丁们种植什么样的果树一样。DJYOS 资源管理模型是一种链表管理方式，具体拓扑思想在下面介绍，这个模型的管理是开放的，系统提供了一组标准的 API，用户根据实际需求去建立和维护资源树。

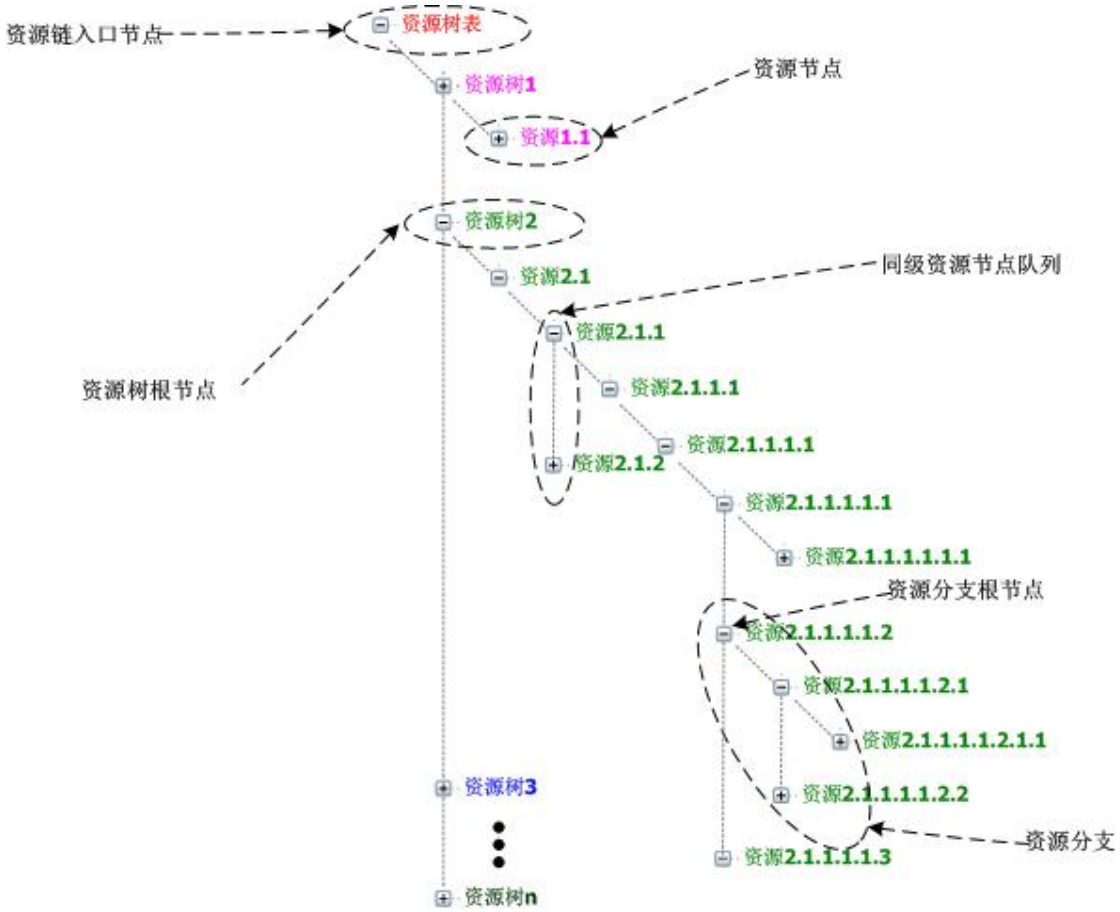


图 11-1 DJYOS 资源组织示意图

图 11-1 展示了一个 DJYOS 下资源组织结构。初始化时系统会为整个资源链（树）表建立一个指针 s_ptRscRoot 指向的入口节点“resouce queue”。所有的资源树都有建立在“resource queue”之下，两者之间是一种父子关系。而资源树之间属于同级关系，它们之间是一个双向环形链表结构，同时它也具有队列属性，被入口节点（child 指针）指向的资源树（根节点）即为队头。我们把每一棵资源树定义为同类资源的集合，它从资源树根节点开始，组织建立树的各级资源节点，节点的组织模型如下：

1.所有资源树都是从根节点开始，资源树根节点由入口节点“resource queue”管理。

2.临近（上下）层级节点之间是父子关系；例如图中节点资源 2.1 与节点资源 2.1.1 或者节点资源 2.1 与节点资源 2.1.2 之间都是父子关系。

3.同层级节点之间由双向环形链表联接，同时链表具有队列属性，被上级(父)节点所指向的节点为队头。例如图中节点资源 2.1.1.1.1.1、与节点资源 2.1.1.1.1.2 和节点资源 2.1.1.1.1.3 形成的同级节点队列，节点资源 2.1.1.1.1.1 为队头。

4.一个节点及其所有下级节点组成一个资源分支；例如图中资源 2.1.1.1.1.2 与资源 2.1.1.1.1.2.1 等就形成了一个资源分支，其中 2.1.1.1.1.2 为这个分支根节点；

对于应用者而言，实际上只要关心有哪些的资源树可用就足够了，最好不要访问入口节点“resource queue”，以免破坏整个资源体系，为此，系统设计者仅仅开放了一个调试用接口函数 Rsc_GetRoot 用于观测整个资源体系。在图 11-1 中的资源树根节点和资源节点的数据结构是相同的，定义如下：

```
struct rsc_node
{
    struct rsc_node *next,*previous,*parent,*child;
```

```
    uint32_t node_size;
    char *name;
};
```

资源节点结构体代表一个资源对象，其节点指针元素 next、previous、parent 和 child 用于构建资源体系；node_size 表示负载尺寸；name 是资源名称，根据资源类型不同而不同，可以是文件系统的文件名，也可以是设备的设备名，或者是 GUI 的窗口名称，当然也没有不定义名称。资源节点的命名规则如下：

1. 资源名可以是但不限于任何合法的 C 语言字符串，允许中文字符串和中英文混合字符串。但名称中不能包含回车符、换行符、'\'和\0 等。

2. 路径名是指从树根节点名开始沿着该支资源链表逐级检索到目标资源所经过各级节点名，节点名之间用字符 '\' 分隔，节点名之间先后关系代表父子关系。例如图 11-1 中资源 2.1.1.1 完整路径名是“资源树 2\资源 2.1\资源 2.1.1\资源 2.1.1.1”。

3. 资源路径的组成与 windows 系统的文件路径有点相似，但 DJYOS 中路径本身也可以是完整的资源，例如图中的”资源 22”是一个资源实体，而 windows 文件系统的路径名只能是文件夹。

为了便于分类管理，不同类型的资源应该各自形成一棵树，拥有自己的根节点，并把同类的资源节点作为根节点上的分支。例如下图所示，在设备管理中，所有设备组成一棵资源树，这棵树的每个节点都代表一个具体设备，* s_ptDeviceRscTree 指向这个设备树的根节点；在 GUI 设计中，同一个显示器下的所有窗口组成一棵树，树上的每个节点代表一个窗口，s_tWindowRootRsc 资源是窗口树的树根。

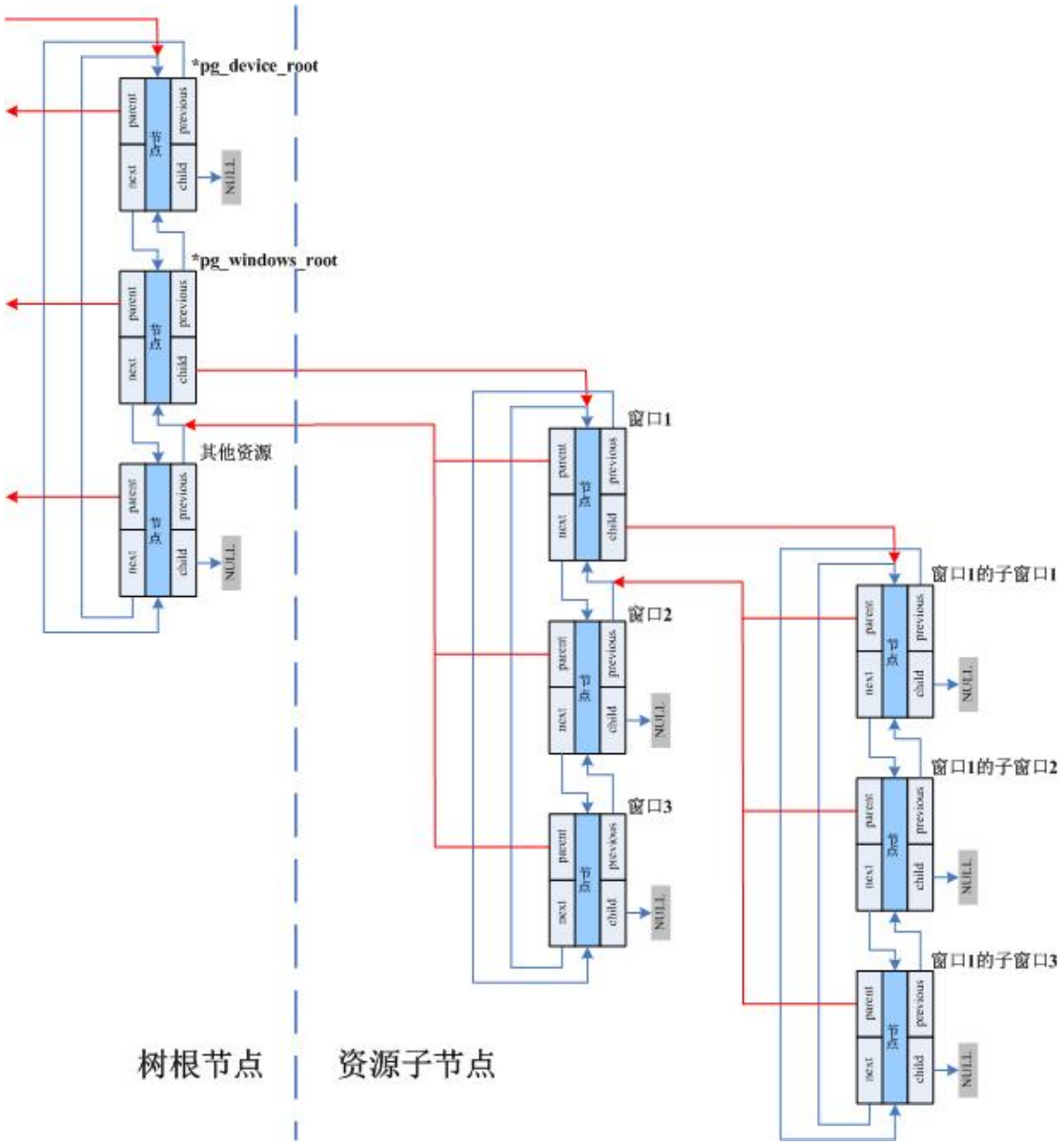


图 11-2 DJYOS 资源链表示意图

11.2 载荷

就像一个单纯运行操作系统的计算机是毫无意义的一样，一棵单纯的资源树也是毫无意义的，资源树是用来组织数据的，它所管理的数据，称作负载。通过资源链表，程序员可以实现对负载的有序化管理。例如操作系统的信号量管理就是依照资源链表进行管理的，信号量的定义如下：

```
struct tagSemaphoreLCB
{
    struct   tagRscNode node;
    u32 sync_order;
    u32 lamps_limit;     //信号灯数量上限，cn_limit_uint32 表示不限数量
    u32 lamp_counter;    //可用信号灯数量。
```

```
    struct    tagEventECB *semp_sync;       //等候信号的事件队列
};
```
在 DJYOS 中，所有信号量组成一棵信号树，每个信号量就是信号树上的一个结点，从而实现对信号量的集中管理。
在下一章中我们可以看到，作为都江堰操作系统最重要的模块之一——设备管理模块，就是通过资源链表进行管理的。

11.3 API 说明

11.3.1 Rsc_AddTree：添加树根节点

struct tagRscNode   *Rsc_AddTree(struct tagRscNode *node, u32 size, char *name)
头文件：
os_inc.h
参数：
node：指向新添加的树根节点指针。
size：新节点负载的尺寸，字节数，负载即资源节点所管理的数据。
name：资源名称，所指向的字符串不能定义为局部变量，无名称即为 NULL。
返回值：
指向新添加的树根节点指针。
说明：
在资源链中添加一个资源树根节点，基于根节点可以生长出一棵资源树。

11.3.2 Rsc_CleanNode：重置节点

void   Rsc_CleanNode(struct tagRscNode *node)
头文件：
os_inc.h
参数：
node：指向被清零的节点的指针。
返回：
无。
说明：
将资源节点的指针元素清零。注意，该函数只清除资源节点结构体中的指针元素。

11.3.3 Rsc_ AddToPrevious：前插同级节点

struct tagRscNode   * Rsc_AddToPrevious (struct tagRscNode *node,
                                          struct tagRscNode *new_node,
                                          u32 size,
                                          char *name)
头文件：

os_inc.h
参数：
node：基准节点，新节点是插入这个节点的 previous 位置。
new_node：待插入的新节点，数据结构由调用者提供，但调用者无需初始化它；
size：新节点的负载尺寸，字节数，负载即资源节点所管理的数据。
name：资源名称，所指向的字符串不能定义为局部变量，无名称即为 NULL。
返回值：
指向新插入节点的指针。
说明：
在同层级资源节点队列上，将新节点 new_node 插入到基准节点 node 之前（previous 位置）。

11.3.4 Rsc_AddToNext：后插同级节点

```
struct rsc_node    *Rsc_ AddToNext (struct rsc_node *node,
                                    struct rsc_node *new_node,
                                    uint32_t size,
                                    char *name)
```

头文件：
os_inc.h
参数：
node：基准节点，新节点是插入在这个节点的 next 位置。
new_node：待插入的节点，数据结构由调用者提供，但调用者无需初始化它。
size：新结点的负载的尺寸，字节数，资源结点所管理的数据即负载。
name：资源名称，所指向的字符串不能定义为局部变量，无名称即为 NULL。
返回值：
指向新插入结点的指针。
说明：
在同级资源节点队列上，将新节点 new_node 插入到基准节点 node 之后（next 位置）。

11.3.5 Rsc_AddSon：插入下级节点

```
struct tagRscNode    *Rsc_AddSon(struct tagRscNode *parent_node,
                                    struct tagRscNode *new_node,
                                    u32 size,
                                    char *name)
```

头文件：
os_inc.h
参数：
parent_node：基准节点，新节点是插入在这个节点的子层级的资源节点队列上。
new_node：待插入的新节点，数据结构由调用者提供，但调用者无需初始化它。
size：新节点的负载尺寸，字节数，负载即资源节点所管理的数据。
name：资源名称，所指向的字符串不能定义为局部变量，无名称即为 NULL。
返回值：

指向新添加节点的指针。
说明：
在 parent_node 的子层级资源节点队列上，将节点 new_node 插入到这个队列的队尾。

11.3.6 Rsc_AddHeadSon：插入下级首节点

```
struct tagRscNode    *Rsc_AddHeadSon(struct    tagRscNode *parent_node,
                                     struct    tagRscNode *new_node,
                                     u32 size,
                                     char *name)
```

头文件：
os_inc.h
参数：
parent_node：基准节点，新节点是插入在这个节点的子层级资源节点队列上。
new_node：待插入的节点，数据结构由调用者提供，但调用者无需初始化它。
size：新节点负载的尺寸，字节数，负载即资源节点所管理的数据。
name：资源名称，所指向的字符串不能定义为局部变量，无名称即为 NULL。
返回值：
指向新添加的节点的指针。
说明：
在 parent_node 的子层级资源节点队列上，将节点 new_node 插入到这个队列的队头。

11.3.7 Rsc_DisplaceNode：更换节点

```
bool_t    Rsc_DisplaceNode(struct tagRscNode *oldnode, struct tagRscNode *newnode)
```

头文件：
os_inc.h
参数：
oldnode：指向被替换节点的指针；
newnode：指向新节点的指针。
返回值：
替换成功返回 true；失败则返回 false。
说明：
用新节点替换掉原节点，注意，新节点必须不是资源链表中的节点。

11.3.8 Rsc_DelBranch：删除分支

```
struct tagRscNode    *Rsc_DelBranch(struct tagRscNode *branch)
```

头文件：
os_inc.h
参数：
branch：被删除的分支。

返回值：
成功删除返回值指向被删除分支节点的指针；返回值为 NULL 表示分支不存在。
说明：
将资源节点 branch 及其下层级资源节点队列，即资源分支，从资源树上剔除。注意，本函数只是将这个分支与资源树断开链接，并不释放该分支所占据的内存。

11.3.9 Rsc_DelNode：删除结点

struct tagRscNode *Rsc_DelNode(struct tagRscNode *node)
头文件：
os_inc.h
参数：
node：被删除的节点，注意该节点不含有子层级节点。
返回值：
删除成功返回被删除节点的指针；出错则返回 NULL；
说明：
断开节点与资源链表中之间的链接，注意该节点不能包含子层级节点，Rsc_DelBranch 函数删除。注意只是断开链表，并不释放内存，因为资源管理模块并不分配和释放结点内存。如果被删除的结点是同级资源的头节点，next 节点将成为新的头结点。

11.3.10 Rsc_MoveToTree：移动分支

bool_t Rsc_MoveToTree(struct tagRscNode *parent, struct tagRscNode *node)
警告：函数未实现。

11.3.11 Rsc_MoveToLast：移动节点至队尾

bool_t Rsc_MoveToLast(struct tagRscNode *node)
头文件：
os_inc.h
参数：
node：被移动的节点的指针
返回值：
执行成功返回 true；失败则返回 false。
说明：
将节点 node 移动到其所在的资源节点队列的队尾。

11.3.12 Rsc_MoveToHead：移动节点至队首

bool_t Rsc_MoveToHead(struct tagRscNode *node)
头文件：

os_inc.h
参数：
node：被移动的节点的指针。
返回值：
执行成功返回 true；失败则返回 false。
说明：
将节点 node 移动到其所在的资源节点队列的队头。

11.3.13 Rsc_MoveToNext：后移节点

bool_t Rsc_MoveToNext(struct tagRscNode *elder, struct tagRscNode *node)
头文件：
os_inc.h
参数：
node：被移动的节点的指针。
elder：基准节点，node 移动到本节点的 next 位置。
返回值：
执行成功返回 true；失败则返回 false。
说明：
将节点 node 移动到处于同层级资源节点队列上的节点 elder 之后（next 位置）。

11.3.14 Rsc_MoveToPrevious：前移节点

bool_t Rsc_MoveToPrevious(struct tagRscNode *lesser, struct tagRscNode *node)
头文件：
os_inc.h
参数：
lesser：基准节点的指针，node 移动到本节点 previous 位置。
node：被移动节点的指针。
返回值：
执行成功返回 true；失败返回 false。
说明：
将节点 node 移动到处于同层级资源节点队列上的节点 lesser 之前（previous 位置）。

11.3.15 Rsc_RoundPrevious：前移队首节点

bool_t Rsc_RoundPrevious(struct tagRscNode *parent)
头文件：
os_inc.h
参数：
parent：父节点的指针。
返回值：

执行成功返回 true；失败返回 false。
说明：
将节点 parent 的子层级资源节点队列的队头向前（previous 位置）移动一个节点位置。

11.3.16 Rsc_RoundNext：后移队首节点

bool_t Rsc_RoundNext(struct tagRscNode *parent)
头文件：
os_inc.h
参数：
parent：父节点的指针。
返回值：
执行成功返回 true；失败返回 false。
说明：
将节点 parent 的子层级资源节点队列的队头向后（next 位置）移动一个节点位置。

11.3.17 Rsc_RenameNode：节点重命名

bool_t Rsc_RenameNode(struct tagRscNode *node, char *new_name)
头文件：
os_inc.h
参数：
node：被修改的资源节点的指针。
new_name：节点新名称的指针。
返回值：
执行成功返回 true；失败返回 false。
说明：
修改资源节点名称（本函数执行过程不存在数据拷贝）。

11.3.18 Rsc_GetTree：追溯根节点

struct tagRscNode *Rsc_GetTree(struct tagRscNode *scion_node)
头文件：
os_inc.h
参数：
scion_node：目标节点。
返回值：
节点所在的资源树的根节点。
说明：
获取指向基准节点所在资源树根节点的指针。

11.3.19 Rsc_GetRoot：查询根节点

struct tagRscNode *Rsc_GetTree(struct tagRscNode *scion_node)
头文件：
os_inc.h
参数：
无。
返回值：
指向资源链表入口节点的指针。
说明：
获取资源链入口节点指针。

11.3.20 Rsc_GetName：查询节点名

char *Rsc_GetName(struct tagRscNode *node)
头文件：
os_inc.h
参数：
node：节点指针。
返回值：
节点名指针。
功能：
获取节点名指针。

11.3.21 Rsc_GetParent：获取父节点

struct tagRscNode *Rsc_GetParent(struct tagRscNode *son_node)
头文件：
os_inc.h
参数：
son_node：基准子节点。
返回值：
指向基准节点的父节点的指针。
说明：
获取基准节点的父节点指针。

11.3.22 Rsc_GetSon：获取子节点

struct tagRscNode *Rsc_GetSon(struct tagRscNode *parent_node)
头文件：
os_inc.h

参数：
parent_node：基准父节点。
返回值：
子节点的指针。
说明：
获取节点 parent_node 所指向的子节点的指针。注意：当节点 parent_node 存在多个子节点时，返回的是子节点同级队列的队列头。

11.3.23 Rsc_GetPrevious：获取前节点

struct tagRscNode *Rsc_GetPrevious(struct tagRscNode *next_node)
头文件：
os_inc.h
参数：
next_node：基准节点。
返回值：
基准节点前一个节点的指针（previous 位置）。
说明：
基准节点所在的同级资源节点队列上，获取基准节点的前一个节点（previous 位置）。

11.3.24 Rsc_GetNext：获取后节点

struct tagRscNode *Rsc_GetNext(struct tagRscNode *previous_node)
头文件：
os_inc.h
参数：
previous_node：基准节点。
返回值：
基准节点后一个节点的指针（next 位置）。.
说明：
基准节点所在的同级资源节点队列上，获取基准节点的后一个节点（next 位置）。

11.3.25 Rsc_GetHead：获取队首节点

struct tagRscNode *Rsc_GetHead(struct tagRscNode *rnode)
头文件：
os_inc.h
参数：
rnode：基准节点。
返回值：
基准节点所在同级节点队列的头节点指针。
说明：

获取基准节点所在同级节点队列的队列首节点。

11.3.26 Rsc_GetTwig：获取分支结点

struct tagRscNode *Rsc_GetTwig(struct tagRscNode *ancestor_node)
头文件：
os_inc.h
参数：
ancestor_node：基准节点。
返回值：
节点 ancestor_node 分支的末梢节点。所谓末梢节点，就是没有子节点的节点。当基准节点本身就是末梢节点，则返回 NULL。
说明：
获取节点 ancestor_node 分支的末梢节点（结点）。当需要删除一个资源树分支时，本函数结合函数 Rsc_DelNode，反复调用，直至本函数返回 NULL。例如需要删除一个文件夹或者存在子窗口的图形窗口时，就需要此类操作。

11.3.27 Rsc_GetClass：查询节点级别

u32 Rsc_GetClass(struct tagRscNode *node)
头文件：
os_inc.h
参数：
node：基准节点。
返回值：
基准节点在资源树中的层级。
说明：
查看基准节点在资源树上的层级。一棵资源树上，根节点的层级为 0，依次递增。

11.3.28 Rsc_TraveScion：遍历分支

struct tagRscNode *Rsc_TraveScion(struct tagRscNode *ancestor_node,
struct tagRscNode *current_node)
头文件：
os_inc.h
参数：
ancestor_node：源节点。
current_node：基准节点。
返回值：
当前搜索位置的下一个节点指针,如果已经搜索完成,则返回 NULL。
说明：
逐个返回源节点 ancestor_node 之下分支的所有节点成员，即遍历整个分支。注意本函数第

一次调用时，两个参数必须是源节点。
当需要对资源链表中某一个树枝或者整个链表中的结点逐一进行某种操作时（即遍历），可反复调用本函数，第一次调用 current_node = parent_node，其后 current_node = 上次返回值，直到返回空，可确保所有子孙结点都能够访问到。本函数不保证搜索顺序，如果对访问顺序有特殊要求，不能使用本函数。

11.3.29 Rsc_TraveSon：遍历下级节点队列

struct tagRscNode *Rsc_TraveSon(struct tagRscNode *parent_node,
struct tagRscNode *current_son)
头文件：
os_inc.h
参数：
parent_node：父节点。
current_node：基准节点。，
返回值：
父节点 parent_node 的子节点队列成员。
说明：
逐个返回（重复调用本函数，每调用一次，返回一个节点）父节点 parent_node 的子（下一级）节点队列成员，其中逐次返回的顺序是从队列头开始，直至队尾，当整个队列全部完成返回后，返回 NULL。注意本函数第一次调用时，两个参数必须是父节点。

11.3.30 Rsc_SearchSibling：检索同级资源

struct tagRscNode *Rsc_SearchSibling(struct tagRscNode *brother, char *name)
头文件：
os_inc.h
参数：
brother：基准节点。
name：需要检索的节点的节点名。
返回值：
名称检索成功，返回节点名为 name 的节点指针；失败则返回 NULL。
说明：
在与基准节点的同级资源节点队列之中，检索节点名为 name 的节点。

11.3.31 Rsc_SearchTree：检索资源树

struct tagRscNode *Rsc_SearchTree(char *tree_name)
头文件：
os_inc.h
参数：
tree_name：资源树名称

返回值：
名称检索成功，返回名称为 tree_name 的资源树的根节点；失败则返回 NULL。
说明：
检索名称为 tree_name 的资源树，返回该资源树的根节点。

11.3.32 Rsc_SearchSon：检索下级资源

struct tagRscNode *Rsc_SearchSon(struct tagRscNode *parent, char *name)
头文件：
os_inc.h
参数：
parent：基准节点。
name：节点名称。
返回值：
名称检索成功，则返回节点名为 name 的节点指针；否则返回 NULL。
说明：
在基准节点的子节点队列中，检索名称为 name 的子节点。

11.3.33 Rsc_SearchScion：检索分支中的资源

struct tagRscNode *Rsc_SearchScion(struct tagRscNode *ancestor_node, char *name)
头文件：
os_inc.h
参数：
ancestor_nod：基准节点。
name：节点名称。
返回值：
名称检索成功，则返回节点名为 name 的节点指针；否则返回 NULL。
说明：
在以节点 ancestor_node 为起点的分支上，检索名称为 name 的节点。

11.3.34 Rsc_Search：按路径检索资源

struct tagRscNode *Rsc_Search(struct tagRscNode *ancestor_node, char *path)
头文件：
os_inc.h
参数：
ancestor_node：基准节点，分支的源节点。
path：节点路径名称。
返回值：
检索成功返回目标节点；否则返回 NULL。
说明：

从节点 ancestor 为源头（起点）的分支开始，检查路径名与 path 一致的节点，注意这里的路径名是从节点 ancestor 开始的。

11.3.35 Rsc_IsHead：是否是队首节点

bool_t Rsc_IsHead(struct tagRscNode *node)
头文件：
os_inc.h
参数：
node：节点指针。
返回值：
节点为同级节点队列的队头返回 true；否则返回 false。
说明：
判断节点 node 是不是其所处的同级节点队列的头部。

11.3.36 Rsc_IsLast：是否是队尾节点

bool_t Rsc_IsLast(struct tagRscNode *node)
头文件：
os_inc.h
参数：
node：节点指针。
返回值：
节点为同级节点队列的列尾返回 true；否则返回 false。
说明：
判断节点 node 是不是其所处的同级节点队列的尾部。

11.3.37 Rsc_ Nodesequencing：获取节点序号

u32 Rsc_Nodesequencing (struct tagRscNode *node)
头文件：
os_inc.h
参数：
node：节点指针。
返回值：
节点位置号
说明：
获取节点 node 在其所属的同层级节点队列中所处的位置，如果该节点是队头，返回零；后续依次增一。

11.3.38 例程 1：显示资源树

本例程涉及到 Rsc_TraveScion、Rsc_GetClass 函数。

```
void __sh_show_branche(struct rsc_node *branche)
{
    struct rsc_node *current_node = branche;
    ucpu_t len;
    char neg[20];
    for(len = 0; len<20; len++)
        neg[len] = '-';
    while(1)
    {
        current_node = Rsc_TraveScion(branche,current_node);
        if(current_node == NULL)
        {
            printf_str("\r\n");
            break;
        }
        len = Rsc_GetClass(current_node);
        neg[len] = '\0';
        printf_str(neg);
        neg[len] = '-';
        if(current_node->name != NULL)
        {
            printf_str(current_node->name);
            printf_str("\r\n");
        }
        else
        {
            printf_str("无名资源\r\n");
        }
    }
}
```

该例程的功能是显示某资源结点起始的一个资源分支，不包含该资源节点自身。是通过 Rsc_TraveScion 函数遍历要显示的资源分支。然后在遍历的过程中，找到一个资源就打印一个，Rsc_GetClass 函数用来判断找到的资源节点是资源树里的哪级，然后打印的时候就通过“-”表示几级。该函数的显示效果，大家可以通过使用 shell 命令 “rsc_tree tree_name”在超级终端里进行体验。

第12章　设备驱动模型

12.1 设备驱动模型架构

Djyos 设备驱动模型，是从原来的泛设备模块修改而来。虽然从系统架构上讲，泛设备组件作为功能模块之间互相交互接口，有利于设计移植性强的模块，有利于企业处理历史包袱。但泛设备的概念与普通系统差异太大，许多用户感觉左右手接口概念比较拗口。DJYOS 崇尚功能全面而又洁易用，不希望拥有难于理解的组件，故而放弃泛设备模块，回归传统的设备驱动架构。

DJYOS 系统中，设备驱动模型赋予了广泛的含义，它是被设计成功能模块间互相访问的接口。功能模块可能是硬件，也可能是软件，还可能是软硬件结合的模块，驱动程序不再仅仅是访问硬件的接口。从软硬件联合设计的角度，DJYOS 系统并不区分软件模块还是硬件模块，如果完整产品由多个模块组成，任意一个模块在别的模块“眼里”都可以以设备的形式出现的，使用设备的模块并不知道该设备的实现细节，也不知道该设备是由硬件组成的还是由纯软件组成的。某一个功能由软件还是硬件实现并不重要，关键是，它具备需要的功能，并且为别的模块提供了相同的访问和操作接口。

DJYOS 不鼓励用户编写设备驱动程序时，不管自身特点，生搬硬套地往操作系统的设备驱动模型上套。操作系统的驱动架构，仅仅是给你提供了一个有一定通用性的模型，但不确保适合于所有的硬件模块，你觉得方便就用，不方便就不用。随 DJYOS 发行的驱动程序，也有相当部分没有按照 Djyos 的驱动模型设计，而是根据模块本身特性，为不同类型模块设计不同结构的驱动程序。

1. IIC 总线驱动、SPI 总线等驱动等，这些驱动都独立于设备驱动模型之外，仅直接描述硬件系统的实际连接。

2. djyfs 涉及的存储设备驱动，djygui 涉及的显卡驱动，协议栈涉及的网卡驱动，timer 模块涉及的定时器驱动，wdt 模块涉及的看门狗驱动，日历时钟模块涉及的 rtc 驱动，都没有套用驱动模型。

Djyos 的设备驱动模型，不再区分字符设备、块设备、网络设备。大容量存储设备有其专用的存储介质接口，网络驱动也有其专门的网卡接口，不再与设备驱动模型发生关联。

文件系统本是一种数据组织和读写的软件方案，所涉及的存储器驱动为这个特定方案服务，它规定了特定的接口协议，按这个协议编写具体的存储介质驱动就可以了，并不需要套用 Djyos 的设备驱动模型。Djyos 中没有块设备驱动的概念，文件系统也没有规定存储介质必须按块访问，例如 Djyos 提供的 DFFSD（Flash file system）模块，是按块组织 Flash 的，而 DEFSD（Easy norFlash File system）模块，就没有按块方式组织存储介质。文件和设备不再使用统一的接口函数，文件将不能用 open、write 和 read 访问，而是用 fopen、fwrite、fread 访问。

同样，网络系统的 socket 接口，也是专用接口，无须硬套设备架构的通用接口，socket 的句柄，也是独立的，跟设备驱动架构没有联系。网卡驱动也是按照网络协议栈提出的标准驱动接口进行编程。

除此以外，图形卡的驱动程序，也是 djygui 自行规定的接口标准。

DJYOS 中，设备是依托资源树实现的，资源树中有一棵名为“dev”的树，所有设备均是这棵树的子孙结点。

图中显示了设备管理对系统资源树的依赖关系，每个设备都是系统中的一个资源，“dev”资源结点是所有设备的祖先结点，静态全局指针 struct rsc_node *s_ptDeviceRscTree 指向该

结点。每个设备由 tagDjyDevice 结构体描述如下：

```
struct tagDjyDevice
{
    struct   tagRscNode Node;
    devWriteFunc   dWrite;
    devReadFunc    dRead;
    devCtrlFunc    dCtrl;
    devMultiplexAddFunc dMultiplexAdd;   //若设备 driver 不支持多路复用，请置空。
    struct tagMutexLCB *dMutex;          //互斥量,控制设备独占式访问
    u32 delete_lock;                     //删除锁，大于 0 表示该设备不能删除
    ptu32_t PrivateTag;                  //本设备特有的数据
};
```

1. Node，资源结点，用于把设备作为一个结点接入系统资源链表。

2. dWrite，设备的写接口，由设备驱动程序提供，用户调用 devWrite 函数后，将间接调用 dWrite 函数。

3. dRead，设备的读接口，由设备驱动程序提供，用户调用 devRead 函数后，将间接调用 dRead 函数。

4. dCtrl，设备的控制接口，由设备驱动程序提供，用户调用 devCtrl 函数后，将间接调用 dCtrl 函数。

5. dMultiplexAdd，设备多路复用功能接口，由设备驱动程序提供，用户调用 devMultiplexAdd 把设备 ID 加入一个 MultiplexSets，将间接调用 dMultiplexAdd 函数。

6. dMutex，用于控制设备独占式访问的互斥量，任何事件处理线程，只有取得设备的互斥量，才可以读写和控制设备。但是，跟 multiplex 相关的功能，无须取得互斥量。

7. delete_lock，删除锁定，用于防止设备被误删除。

8. private_tag 是具体设备的专用标签，它的数据类型 ptu32_t 是一个特别的数据类型，它定义成一个至少 32 位（4 字节）长的、并且长度大于或者等于指针类型长度的整数数据类型，它有 3 层含义：

a> 如果用户把它当整数使用，那么它至少可以得到一个 32 位字长的整数。

b> 用户可以安全地把它强制转化为指针使用；

c> 如果 DJYOS 被移植到一个指针长度小于 32 位的计算机中，那么 ptu32_t 就被定义成 u32 型，如果指针的长度是 64 位的，那么 ptu32_t 就定义成 u64 型数据。

用户为自己的设备编写 driver 时，可以灵活利用 private_tag 成员，根据具体设备的复杂程度，它既可以是一个指针，指向由具体设备定义的复杂的数据结构，又可以根本不用。

虽然设备是按名字访问的，但是 struct DjyDevice 结构中，并没有出现 name 成员，这是因为 DJYOS 系统借助资源管理模块组织设备的数据结构，与其他使用资源管理模块的模块一样，资源名（struct Rsc_Node 的 name 成员）就是设备名。

这个结构是应用程序无需也不能直接接触的，只有设备 driver 的设计者才需要关心这个结构的细节。struct tagDjyDevice 是实现设备管理核心数据类型，它包括操作系统管理设备所需数据成员。操作系统并不知道设备要实现的功能，这是设备的秘密，是驱动设计者所要关注的，private_tag 成员指向具体设备独有数据，它是私有的，隐藏了具体设备所有秘密。

12.2 使用设备步骤

12.2.1 初始化设备管理模块

在建立任何设备 driver 之前，先要调用系统函数 Driver_ModuleInit 初始化操作系统的设备管理器。该函数是在操作系统加载过程中的内核组件初始化步骤调用的。初始化工作包含 3 个步骤：

1. 在操作系统资源链表中增加了一个设备根结点，然后使根设备指针 s_ptDeviceRscTree 指向根结点。

2. 建立设备控制块内存池和设备句柄内存池，这是两个“固定块分配法”的内存池，设备控制块池的块尺寸是 sizeof(struct tagDjyDevice)，默认容量是 CN_DEVICE_LIMIT 块；设备句柄池与此类似。

3. 创建一个互斥量，用于保护资源树中设备子树的并发访问，请读者注意区分该互斥量与保护设备本身的信号量。

初始化完成后，就可以创建设备了。设备管理模块并不是操作系统运行所必须的，如果项目中没有使用设备驱动，在 module-trim.c 的 Sys_ModuleInit 函数中注释掉 Driver_ModuleInit 函数调用语句，就把设备管理模块裁减掉了。

12.2.2 创建和使用设备

如所示，使用 DJYOS 设备驱动程序非常简单，只需要少数几个步骤就可以完成，如图 12-1 所示，详细使用过程请参考 api 介绍章节。

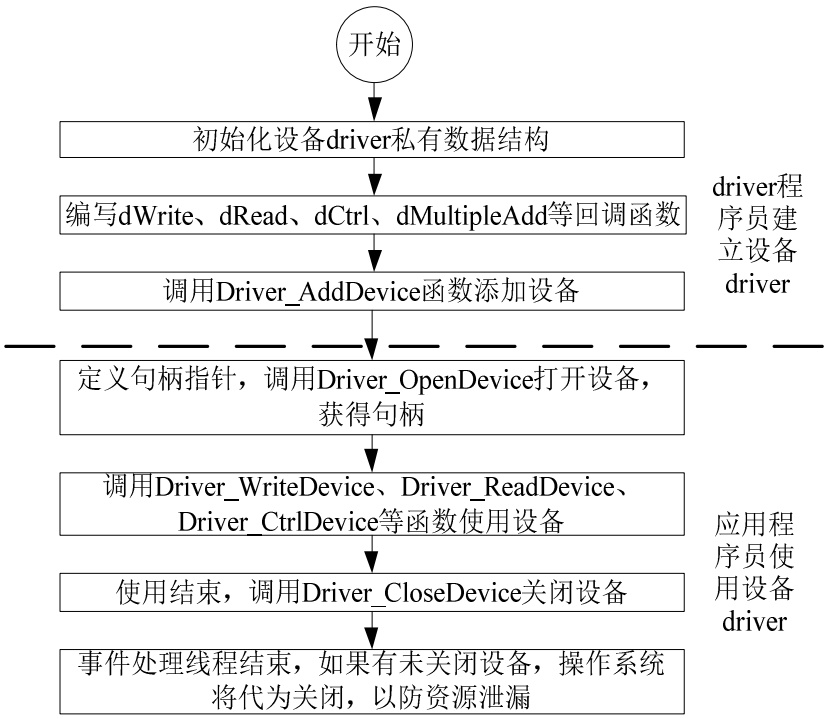


图 12-1 建立和使用设备流程示意图

如图所示在将设备挂接到设备树根节点 dev 下之前，设备驱动程序员需要编写 tagDjyDevice 里定义的四个回调函数，如果设备不需要支持多路复用功能，可不编写

版权所有，转载必须注明出处，注明：来自 www.djyos.com 字样。

dMultipleAdd。这四个函数函数原型如下：

```
typedef u32 (*devWriteFunc)(ptu32_t PrivateTag,u8 *buf,
                            u32 len,u32 offset,bool_t BlockOption,u32 timeout);
typedef u32 (*devReadFunc) (ptu32_t PrivateTag,u8 *buf,
                           u32 len,u32 offset,u32 timeout);
typedef u32 (*devCtrlFunc) (ptu32_t PrivateTag,u32 cmd,
                          ptu32_t data1,ptu32_t data2);
typedef bool_t (*devMultiplexAddFunc) (ptu32_t PrivateTag,
                                       struct tagMultiplexSetsCB *MultiplexSets,
                                       u32 DevAlias,
                                       u32 SensingBit);
```

除标准的打开和关闭设备方式外，driver 模块还支持另一种快速操作模式：别名操作。标准的打开过程需要用字符串比对来搜索设备，会比较慢，而别名操作则省去了搜索过程，直接定位到设备指针，速度非常快，特别适合那些需要反复打开和关闭设备的情况。系统提供了 Driver_FindDevice、Driver_FindScionDevice、Driver_OpenDeviceAlias 三个函数用于支持别名操作。

12.3 API 说明

12.3.1 Driver_AddDevice：添加设备

```
djy_handle_t    Driver_AddDevice(tagDevHandle ParentDevice,
                                 char *name,
                                 struct tagMutexLCB *dMutex,
                                 devWriteFunc WriteFunc,
                                 devReadFunc ReadFunc,
                                 devCtrlFunc Ctrl,
                                 devMultiplexAddFunc MultiplexAdd,
                                 ptu32_t tag);
```

头文件：
os_inc.h
参数：
ParentDevice：添加设备的上级设备句柄，缺省值是设备资源树根节点；
name：新添加设备名称，注意该指针指向的字符串不能是局部变量，不能为 NULL，不能包含“\”等字符；
dMutex：设备的互斥量，如果为 NULL，系统会自动为该设备创建一个；
WriteFunc：写设备的函数指针；
ReadFunc：读设备的函数指针；
Ctrl：配置设备的函数指针；
MultiplexAdd：支持多路复用的函数指针；
tag：设备驱动标记，代表设备驱动的数据结构。
返回值：
成功添加返回该设备句柄；否则返回 NULL。

说明：
将一个设备添加到设备树的某个设备（ParentDevice）之下，并对新添加设备进行初始化。这个设备句柄（ParentDevice）可以是设备树的根节点，也可以是设备树上其他设备资源，但缺省值是设备树的根节点。

12.3.2 Driver_DeleteDevice：删除设备

bool_t Driver_DeleteDevice(tagDevHandle handle);
头文件：
os_inc.h
参数：
handle：待释放设备的句柄。
返回值：
删除释放返回 true；否则返回 false。
说明：
将设备从设备树上删除，并释放该设备的控制块所占内存。设备被成功删除需具备以下条件：
1. 该设备必须是末端设备，即该设备没有从属设备。从资源管理概念来讲就是该设备资源节点没有子节点；
2. 该设备不在使用中。

12.3.3 Driver_LockDevice：锁定设备

bool_t Driver_LockDevice(u32 DevAlias);
头文件：
os_inc.h
参数：
DevAlias：设备别名，函数 Driver_FindDevice 的返回值。
返回值：
上锁成功，返回 true；失败则返回 false。
说明：
对设备进行一次上锁操作。

12.3.4 Driver_UnLockDevice：解除锁定设备

bool_t Driver_UnLockDevice(u32 DevAlias);
头文件：
os_inc.h
参数：
DevAlias：设备别名，函数 Driver_FindDevice 的返回值。
返回值：
解锁成功，返回 true；失败则返回 false。
说明:

对设备进行一次上锁操作。

12.3.5 Driver_FindDevice：查找设备

u32　Driver_FindDevice(char *name);
头文件：
os_inc.h
参数：
name，设备名字，包含路径名，但不必包含'dev\'这 4 个字符。
返回值：
设备句柄的别名。
说明：
通过设备在设备树上的完整路径名，检索出设备。注意这个路径名必须是从设备树根资源节点开始。

12.3.6 Driver_FindScionDevice：查找后代设备

u32　Driver_FindScionDevice(tagDevHandle ancestor,
　　char *scion_name);
头文件：
os_inc.h
参数：
ancestor：设备树分支根设备；
scion_name：设备路径名。
返回值：
检索成功，返回该设备句柄的别名；失败则返回 NULL;
说明：
从设备树的分支根设备开始，检索路径名与 scion_name 一致的设备。注意路径名 scion_name 是从分支根节点 ancestor 开始的。

12.3.7 Driver_OpenDevice：打开设备

tagDevHandle　Driver_OpenDevice(char *name, u32 timeout);
头文件：
os_inc.h
参数：
name：设备完整路径名
timeout：超时设置，单位是微秒，该值会被自动向上取整为 tick 的整数倍。timeout = CN_TIMEOUT_FOREVER 表示无限等待，timeout = 0 则立即返回。非 0 值将被向上调整为 tick 的整数倍。
返回值：
成功打开设备则返回该设备句柄；否则返回 NULL。

说明：
打开设备。在 timeout 时间内，通过设备的完整路径名检索设备并返回设备句柄。打开设备的同时，会开启设备的互斥锁。

12.3.8 Driver_OpenScionDevice：打开后代设备

```
tagDevHandle    Driver_OpenScionDevice(tagDevHandle ancestor,
                                       char *scion_name,
                                       u32 timeout);
```

头文件：
os_inc.h
参数：
ancestor：设备树分支根设备；
scion_name：设备路径名；
timeout：超时设置,单位是微秒，该值会被自动向上取整为 tick 的整数倍。timeout = CN_TIMEOUT_FOREVER 表示无限等待，timeout = 0 则立即返回。非 0 值将被向上调整为 tick 的整数倍。
返回值：
成功打开设备则返回该设备句柄；否则返回 NULL。
说明：
打开设备。在 timeout 时间内，从设备树的分支根设备开始，检索路径名与 scion_name 一致的设备并返回设备句柄。注意路径名 scion_name 是从分支根节点 ancestor 开始的。打开设备的同时，会开启设备的互斥锁。

12.3.9 Driver_OpenDeviceAlias：快速打开设备

```
tagDevHandle    Driver_OpenDeviceAlias(u32 DevAlias, u32 timeout);
```

头文件：
os_inc.h
参数：
DevAlias：设备别名；
timeout：超时设置，单位是微秒，该值会被自动向上取整为 tick 的整数倍。timeout = CN_TIMEOUT_FOREVER 表示无限等待，timeout = 0 则立即返回。非 0 值将被向上调整为 tick 的整数倍。
返回值：
成功返回设备句柄；否则返回 NULL。
说明：
在 timeout 时间内，根据设备别名，返回设备句柄。

12.3.10 Driver_CloseDevice：关闭设备

```
bool_t    Driver_CloseDevice(tagDevHandle handle);
```

头文件：
os_inc.h
参数：
handle：设备句柄。
返回值：
成功关闭设备返回 true；否则返回 false。
功能：
关闭设备。

12.3.11 Driver_ReadDevice：读设备

```
u32    Driver_ReadDevice(tagDevHandle handle,
                         u8 *buf,
                         u32 len,
                         u32 offset,
                         u32 timeout);
```

头文件：
os_inc.h
参数：
handle：设备句柄；
buf：接受数据的缓冲区；
len：读取数据的长度；
offset：读取内容在设备数据区的位置，对于流设备，例如串口，其位置为零；
timeout：超时设置，单位是微秒，该值会被自动向上取整为 tick 的整数倍。timeout = CN_TIMEOUT_FOREVER 表示无限等待，timeout = 0 则立即返回。非 0 值将被向上调整为 tick 的整数倍。
返回值：
成功调用设备驱动程序提供的读设备函数，则返回该函数的返回值，含义请参考相应设备的使用手册，通常是实际读取的数据量。否则返回-1。
说明：
从设备读取数据，间接调用设备驱动程序提供的 devReadFunc 函数，返回值比较特别，正常情况下，是 devReadFunc 函数的返回值。如果调用 Driver_ReadDevice 时参数检查出错，返回-1，如果 devReadFunc 函数返回值也是-1 的话，可能会混淆，建议 devReadFunc 函数在出错时才返回-1。

12.3.12 Driver_WriteDevice：写设备

```
u32    Driver_WriteDevice(tagDevHandle handle,
                          u8 *buf,
                          u32 len,
                          u32 offset,
                          bool_t BlockOption,
```

u32 timeout);
头文件：
os_inc.h
参数：
handle：设备句柄；
buf：写数据的缓冲区；
len：写数据的长度；
offset：写入的内容在设备数据区的位置对于流设备，例如串口，其位置为零；
BlockOption：阻塞选项，取值为 CN_BLOCK_BUFFER 或 CN_BLOCK_COMPLETE；
timeout：超时设置,单位是微秒，该值会被自动向上取整为 tick 的整数倍。timeout = CN_TIMEOUT_FOREVER 表示无限等待，timeout = 0 则立即返回。非 0 值将被向上调整为 tick 的整数倍。
返回值：
成功调用设备驱动程序提供的写设备函数，则返回该函数的返回值，含义请参考相应设备的使用手册，通常是实际写入的数据量。否则返回-1。
说明：
写入数据到设备，间接调用设备驱动程序提供的 devWriteFunc 函数，返回值比较特别，正常情况下，是 devWriteFunc 函数的返回值。如果调用 Driver_WriteDevice 时参数检查出错，返回-1，如果 devWriteFunc 函数返回值也是-1 的话，可能会混淆，建议 devWriteFunc 函数在出错时才返回-1。

12.3.13 Driver_CtrlDevice：设备控制

u32 Driver_CtrlDevice(tagDevHandle handle,
u32 cmd,
ptu32_t data1,
ptu32_t data2);
头文件：
os_inc.h
参数：
handle：设备句柄；
cmd：控制命令号；
data1：控制命令参数；
data2：控制命令参数。
返回值：
成功调用设备驱动程序提供的控制设备函数，则返回该函数的返回值，含义请参考相应设备的使用手册。否则返回-1。
说明：
控制设备。通过调用用户自定义的控制设备函数（添加设备时注册），实现对设备的控制，具体的控制命令参见 cmd 的宏定义。本函数的返回值即用户自定义的控制设备函数的返回值。

12.3.14 Driver_MultiplexCtrl：设备多路复用控制

```
u32    Driver_MultiplexCtrl(u32 DevAlias,
                            u32 *ReadLevel,
                            u32 *WriteLevel);
```

头文件：
os_inc.h
参数：
DevAlias：设备别名；
ReadLevel：读触发阈值。ReadLevel = NULL 表示无设备；*ReadLevel = 0 表示查询；
WriteLevel：写触发阈值。WriteLevel = NULL 表示无设备；*WriteLevel = 0 表示查询；
返回值：
CN_MULTIPLEX_SUCCESS，设置成功；
CN_MULTIPLEX_INVALID，设备不支持多路复用功能；
CN_MULTIPLEX_ERROR，设置支持多路复用功能，但执行失败。
说明：
设置设备的多路复用（Multiplex）参数和读写触发阈值。调用 devMultiplexAdd 前建议先调用本函数，否则读写触发阈值将是该设备的默认值，而默认值具体要参考此设备的驱动说明。若设备不支持设置阈值，函数将返回 CN_MULTIPLEX_ERROR。而设备不支持 Multiplex 功能，函数则返回 CN_MULTIPLEX_INVALID。

12.3.15 Driver_MultiplexAdd：设备多路复用添加

```
u32    Driver_MultiplexAdd(struct tagMultiplexSetsCB *MultiplexSets,
                           u32 *DevAliases,
                           u32 num,
                           u32 SensingBit);
```

头文件：
os_inc.h
参数：
MultiplexSets：目标多路复用集指针；
DevAliases：一组被操作的设备别名数组,设备别名可由 Driver_FindDevice 提供；
num：设备别名数组成员数；
SensingBit：操作位，见 Multiplex.h 中 CN_MULTIPLEX_SENSINGBIT_READ 常量及相关常量定义。特别注意，如果是 0，则从 MultiplexSets 中删除对象(driver 将调用 Multiplex_Del 函数)。共 31 个感兴趣的 bit，bit31 用于定义对象触发类型，取值 CN_MULTIPLEX_SENSINGBIT_AND 或 CN_MULTIPLEX_SENSINGBIT_OR
返回值：
CN_MULTIPLEX_SUCCESS，设置成功；
CN_MULTIPLEX_INVALID，设备不支持多路复用功能；
CN_MULTIPLEX_ERROR，设置支持多路复用功能，但执行失败。
说明：
将一组设备添加到多路复用集。

12.3.16 Driver_MultiplexDel：设备多路复用删除

```
void    Driver_MultiplexDel(struct tagMultiplexSetsCB *MultiplexSets,
                            u32 *DevAliases,
                            u32 num);
```

头文件：
os_inc.h
参数：
MultiplexSets：目标多路复用集指针；
DevAliases：一组被操作的设备别名数组,设备别名可由 Driver_FindDevice 提供；
num：设备别名数组成员数；
返回值：
无。
说明：
将一组设备从多路复用集中删除。

12.4 Driver 使用范例-UART 驱动

12.4.1 串口驱动模型

DJYOS 提供典型的串口驱动模型，在此模型基础上，移植到不同硬件平台时，只需按规定的接口提供若干硬件操作函数，即可完成串口驱动开发，使开发工作变得简单而快速。

如图 12-2 所示。该模型包括由系统提供的通用设备驱动 driver.c 及 UART 接口 uart.c 及底层驱动部分 cpu_peri_uart.c。该驱动模型具有明显分层结构，顶层是应用程序，它通过访问设备驱动接口访问串口通用接口；中间一层是通用 uart 驱动模型，其是底层物理驱动与设备驱动之间桥梁，其提供了若干钩子函数接口，只需将底层驱动提供的钩子函数挂接到上述对应接口，设备驱动层即可通过钩子函数间接访问 UART。最下面一层是由 BSP 设计者提供的硬件驱动，BSP 设计者只需要按照标准的接口实现硬件相关函数即可。

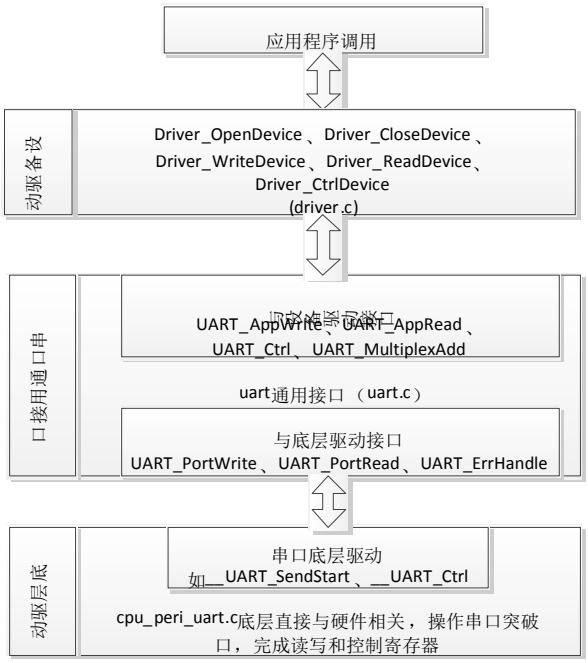


图 12-2 串口驱动架构示意图

使用 DJYOS 串口驱动模型操作串口时，用户需要将串口添加到设备队列中成为一个设备，应用程序通过调用设备驱动层接口进而操作串口。

12.4.2 使用 UART

串口使用流程图如图 12-3 所示，应用层使用该串口模型的步骤大体如下：

1. 初始化串口模型，通过调用 UART_ModuleInit (x)，x 为串口号。该函数中，默认波特率配置为 115200，停止位为 1，无奇偶校验，数据长度为 8 个比特,同时将该 UART 添加到设备队列中；
2. 打开串口，获得串口句柄，调用 Driver_OpenDevice()；
3. 配置串口参数（若使用默认设置，此步可省略），调用 Driver_CtrlDevice()；
4. 开启串口接收发送，调用 Driver_CtrlDevice ()，命令为 CN_UART_START；
5. 向串口写数据或从串口读数据，调 Driver_WriteDevice()/Driver_ReadDevice()；
6. 使用完串口设备需要将串口设备及时关闭，调用 Driver_CloseDevice()。

版权所有，转载必须注明出处，注明：来自 www.djyos.com 字样。

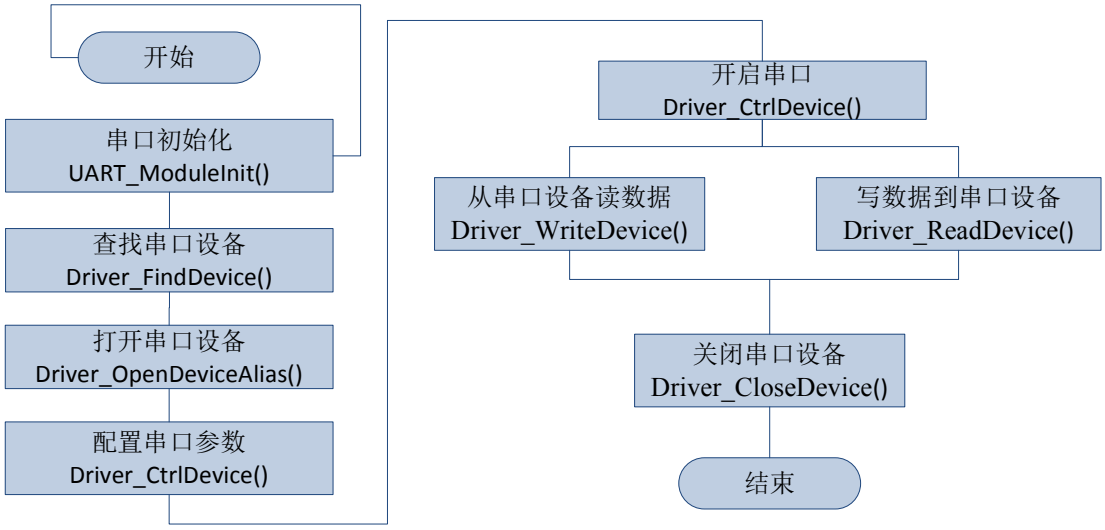


图 12-3 串口使用流程示意图

12.4.3 串口收发数据

DJYOS 标准串口模型将串口作为设备加载到设备列表，在串口初始化后，应用程序将获取串口设备的句柄，然后应用层所有的串口操作都是基于句柄通过通用 UART 层提供的 API 函数访问 UART。

通用 UART 层为设备驱动层提供 API 函数 UART_AppWrite、UART_AppRead、UART_Ctrl、UART_MultiplexAdd，这四个函数具体功能详见下一小节。通用 UART 层访问底层驱动是通过以下 API 函数 UART_PortRead、UART_PortWrite。

应用层读写串口其基本过程如下：发送数据，调用 Driver_WriteDevice，该函数内部调用 UART_AppWrite，UART_AppWrite 将要写的数据写进 UART 发送环形缓冲区 TxRingBuf 中，然后启动串口发送，在串口发送中断中，调用 UART_PortRead 读取环形缓冲区 TxRingBuf 数据并将数据依次发送出去；读串口过程类似，只是过程反过来了，在串口中断中接收数据并将数据写进串口接收缓冲区 RxRingBuf 中，应用层调用 Driver_ReadDevice，该函数内部调用 UART_AppRead，UART_AppRead 读取串口接收缓冲区 RxRingBuf，应用层即可获得串口接收到的数据。

12.4.3.1 应用程序调用

因为串口模型初始化时，将串口设备登记为系统的设备，所以应用层收发数据时，对串口的操作，就转化为对设备的操作，读写串口转化为读写串口设备函数 Driver_ReadDevice 和 Driver_WriteDevice，而本质是调用了串口读写函数 UART_AppRead 和 UART_AppWrite。Driver_CtrlDevice 本函数将间接调用 UART_Ctrl，用法详见 12.4.5.5 节。Driver_WriteDevice 本函数将间接调用 UART_AppWrite，用法详见 12.4.5.1 节。Driver_ReadDevice 本函数将间接调用 UART_AppRead，用法详见 12.4.5.2 节。

版权所有，转载必须注明出处，注明：来自 www.djyos.com 字样。

12.4.3.2 硬件层调用

串口被登记为设备后，底层驱动通过串口通用接口提供的 API 函数读写数据，往串口设备写数据(UART_PortWrite)是指将底层接收到的数据写入串口控制块的环形缓冲区，然后应用层接收到了数据；从串口设备读数据(UART_PortRead)是指从串口控制块的环形缓冲区读数据，然后将读到的数据从底层发送出去。UART_PortWrite 本函数将由底层驱动在发送数据时调用，用法详见 12.4.5.3 节。UART_PortRead 本函数将由底层驱动在发送数据时调用，用法详见 12.4.5.4 节。UART_InstallPort 本函数装载串口到系统设备队列中，用法详见 12.4.5.8 节。

12.4.4 串口通信例程

DJYOS 串口驱动模型下，应用层使用串口更加简单快速，下面以应用层使用串口编程作简单的用例说明，本例程通过初始化串口 1，将“hello world! ”字符串打印到终端设备。

代码 12-1 串口初始化

```
void djy_main(void)
{
    tagDevHandle                                                    uart_lhdl;
//1
    u32                                                             DevAlias;
//2
    char *str = “hello world\r\n”;
    UART_ModuleInit(CN_UART0);
//3
    DevAlias=Driver_FindDevice(“UART0”);
//4
    uart_lhdl = Driver_OpenDeviceAlias(DevAlias,0);
//5
    if(uart_lhdl != NULL)
    {
        Driver_CtrlDevice(uart_lhdl,                CN_UART_SET_BAUD,57600,0，0);
//6
        Driver_CtrlDevice(uart_lhdl,                        CN_UART_START,0,0);
//7

Driver_WriteDevice(uart_lhdl,(ptu32_t)str,strlen(str),0,CN_BLOCK_BUFFER\ ,10*ms)        //8
        Driver_CloseDevice(uart_lhdl);
//9
    }
}
```

示例代码 12-1 说明如下：

1.定义设备的句柄变量；

2.定义设备的别名；

3.初始化串口模块，参数为串口号，如代码所示，此处初始化 uart0 模块。该函数会会默认配置串口参数，如配置串口波特率 115200，停止位 1，数据位 8 位，无校验方式，同时将串口作为设备加载到设备队列中；
4.根据串口设备名从设备队列中查找串口设备获得串口设备别名；
5.打开设备获取设备句柄，设备句柄是应用程序访问泛设备的对象。通过串口设备别名可以打开已经初始化的串口设备，并返回设备句柄。若此时，串口被占用或者无法找到该输入设备，返回为 NULL，打开设备失败；
6.调用串口控制函数，配置串口的波特率，参数 1 为打开串口后所获得的句柄，CN_UART_SET_BAUD 为命令码，参数 3 为波特率，参数 4 和参数 5 为控制命令所需要的参数，这里不需要直接设为 0。控制函数提供修改的串口硬件参数只有波特率，其他硬件参数如停止位、数据位等由底层驱动确定。当使用默认的波特率时此行命令不需要；
7.启动串口，使能发送和接收数据；
8.写串口设备，发送“hello world!”字符串到串口，超时参数为 10ms，超时未发送则返回实际发送的字符数据量；
9.关闭串口设备。

12.4.5 UART 驱动 API 说明

12.4.5.1 UART_AppWrite：串口设备 APP 写

```
ptu32_t   UART_AppWrite(struct tagUartCB *UCB,
                        ptu32_t src_buf,
                        ptu32_t len,
                        u32 offset,
                        bool_t block_option,
                        u32 timeout);
```

头文件：

uart.h

参数：

UCB：被操作的串口 tagUartCB 结构体指针；

src_buf：应用程序写数据时，发送的数据存放于该缓冲区，该类型为 ptu32_t，表示可能会转化为指针的数据类型，此时应用程序应该提供缓冲区的指针；

len：将要写的数据长度，单位字节；

offset：偏移量，在串口模块中，此变量无效；

block_option：阻塞选项，非零为阻塞方式.当block_option为true，函数会等待串口将所有数据发送完成后返回，属于阻塞发送方式；当block_option为false时，函数会等待所有数据被填入串口模块的缓冲区中后，立刻返回，非阻塞发送方式在函数返回时，串口可能仍在发送数据；

timeout：超时时间，单位 us。

返回值：

实际写入环形缓冲区的的字符数。

说明：

该函数实现的功能说明如下：

1. 若未开调度，则采用直接发送方式，发送前先将积压在缓冲区的数据发送完成；
2. 若开调度，把数据写入串口 UCB 的环形发送缓冲区中，然后启动发送数据；
3. 写入数据后，启动发送数据，启动发送的方式可为中断或轮询，由驱动决定；
4. 若缓冲区满，则阻塞等待缓冲区中数据降低到触发水平后，再继续写缓冲区，直到全部数据写入环形缓冲区；
5. timeout，如果在 timeout 时间内，不能把数据全部写入缓冲区，则函数将超时返回，返回实际写入的数据量；
6. 当所有数据写入缓冲区后，判断是否阻塞发送，若为阻塞，则等待阻塞信号量再返回；
7. 返回前，将该串口所在的多路利用集的写感应位清除。

12.4.5.2 UART_AppRead：串口设备 APP 读

```
ptu32_t   UART_AppRead(struct tagUartCB *UCB,
                       ptu32_t dst_buf,
                       ptu32_t len,
                       u32 offset,
                       u32 timeout)
```

头文件：

uart.h

参数：

UCB：被操作的串口 tagUartCB 结构体指针；

dst_buf：应用程序读数据时，提供的存放读到数据的缓冲区，该类型为 ptu32_t，其表示可能会转化为指针的数据类型，此时应用程序应该提供缓冲区的指针；

len：将要读取的数据长度，单位字节；

offset：偏移量，在串口模块中，此变量无效；

timeout：超时时间，单位 us。

返回值：

实际读到的字符数。

说明：

该函数实现的功能说明如下：

1. 读出缓冲区中数据，若缓冲区中数据足够，则直接返回；
2. 若数据不够，则以缓冲容量的一半为单位，阻塞等待分多包接收数据；
3. 阻塞等待数据包时，会更改接收触发水平，并将信号量清除；
4. timeout，如果在 timeout 时间内，不能读出所需数据量，则函数将超时返回，返回实际读到的数据量。

12.4.5.3 UART_PortWrite：串口设备端口写

ptu32_t UART_PortWrite(struct tagUartCB *UCB,u8* buf,u32 len,u32 res)

头文件：

uart.h

参数：

UCB：被操作的串口 tagUartCB 结构体指针；

buf： 底层硬件要写的数据存放于该缓冲区中，该缓冲区中的数据会被写入环形缓冲区中。类型为 ptu32_t，表示可能会转化为指针的数据类型，此时底层硬件应该提供缓冲区的指针；

len：将要写的数据长度，单位字节；

res：保留参数。

返回值：

发送的数据量字符数。

说明：

该函数实现的功能说明如下：

1. 把数据写入串口 uart_UCB 的环形接收缓冲区中；
2. 如果写入环形缓冲区的数据未达到触发水平，则继续收集数据；
3. 环形缓冲区中数据量达到触发水平时，发送信号量，通知上层取取数；
4. 若缓冲区溢出，则 pop 错误处理线程，错误处理事件由应用程序注册。

12.4.5.4 UART_PortRead：串口设备端口读

ptu32_t UART_PortRead(struct tagUartCB *UCB,u8* dst_buf,u32 len,u32 res)

头文件：

uart.h

参数：

UCB：被操作的串口 tagUartCB 结构体指针；

dst_buf：底层硬件从环形缓冲区中读数据存放于该 buf 中。类型为 ptu32_t，表示可能会转化为指针数据类型，此时应用程序应该提供缓冲区的指针；

len：将要读取的数据长度，单位字节；

res：保留参数。

返回值：

实际读到的字节数。

说明：

该函数实现的功能说明如下：

1. 从 uart_UCB 的环形缓冲区中读取数据到底层驱动提供的 dst_buf；
2. 检查本次从缓冲区中读出数据前后是否跨过发送触发水平，若跨过了发送触发水平，则释放信号量，否则继续发送数据。

12.4.5.5 UART_Ctrl：串口设备控制

```
ptu32_t   UART_Ctrl(struct tagUartCB *UCB,
                    u32 cmd,
                    ptu32_t data1,
                    ptu32_t data2)
```

头文件：

uart.h

参数：

UCB：被操作的串口 tagUartCB 结构体指针；

cmd：操作类型，用串口设备控制命令常数表示各种控制类型，各常数定义于 uart.h 中；

data1：命令所需的参数，如不需要则可置 0；
data2：命令所需的参数，如不需要则可置 0。

返回值：

失败返回 0；成功返回 1。

说明：

串口设备的控制函数,由应用程序调用，UART 被注册为设备，调用设备操作函数 Driver_CtrlDevice，实质就是调用了该函数，函数调用过程：Driver_CtrlDevice() ---> Dev->dCtrl() ---> UART_Ctrl()。

该函数实现的功能根据命令字符决定，说明如下：

1. 启动停止串口，由底层驱动实现；
2. 设置半双工发送或接收状态，由底层驱动实现；
3. 暂停和恢复接收数据，由底层驱动实现；
4. 设置波特率和硬件触发水平，由底层驱动实现；
5. 设置错误弹出事件类型。

12.4.5.6 UART_ErrHandle：串口错误处理

```
ptu32_t  UART_ErrHandle(struct tagUartCB *UCB, u32 ErrNo)
```

头文件：

uart.h

参数：

UCB：被操作的串口 tagUartCB 结构体指针；

ErrNo：错误标识的比特位，该比特位必须是多路复用模块未用到感觉位（3-30 比特）。

返回值：

0，错误；1，正确。

说明：

错误处理函数，发生错误时调用该函数，配置多路复用集相应的错误标识位。

12.4.5.7 UART_MultiplexAdd：串口设备添加多路复用

```
bool_t  UART_MultiplexAdd(struct tagUartCB *UCB,
                          struct tagMultiplexSetsCB *MultiplexSets,
                          u32 DevAlias,
                          u32 SensingBit)
```

头文件：

uart.h

参数：

PrivateTag：串口模块的私有标签，此处为 UCB 控制块；

MultiplexSets：多路复用集；

DevAlias：被 Multiplex 的串口的设备别名；

SensingBit：对象敏感位标志，31 个 bit，设为 1 表示本对象对这个 bit 标志敏感，bit31 表示敏感类型，CN_SENSINGBIT_AND，或者 CN_SENSINGBIT_OR。

返回值：

true,添加成功；false,添加失败。

说明：

将 UART 添加到多路复用集，tagUartCB 控制块有成员 pMultiplexCB，调用该函数时，未指向任何 tagMultiplexSetsCB，因此需赋值为 0。

12.4.5.8 UART_InstallPort：添加 UART 到设备队列

tagDevHandle UART_InstallPort(struct tagUartCB *UCB,
struct tagUartParam *Param)

头文件：

uart.h

参数：

UCB：串口控制块指针；

name：串口名称，如串口 1 可命名为“uart1”，添加串口设备时，将串口设备命名为 name，而在打开串口左手或右手时，可通过该设备名找到泛设备；

send_buf：发送缓冲区，发送环形缓冲区的实体，由驱动提供；

send_buf_len：发送缓冲区大小，字节为单位；

recv_buf：接收缓冲区，接收环形缓冲区的实体，由驱动提供；

recv_buf_len：接收缓冲区长度，字节为单位。

返回值：

设备端口指针，NULL 为失败。

说明：

该函数实现的功能说明如下：

1. 该函数由底层驱动调用，并提供缓冲区、名称等相关参数；
2. 初始化环形缓冲区，使缓冲区处于未使用的状态；
3. 添加串口设备到系统设备资源列表，只有添加到资源列表的设备，打开设备时，系统才能根据提供的设备名找到相应的设备。添加设备时，将泛设备的左右手接收和发送的函数指针初始化为串口的左右手接收和发送函数。

第13章 辅助模块

13.1 环形缓冲区

环形缓冲区，是一种类 FIFO 缓存机制，用于模块间的数据通信。下图显示了环形缓冲区的结构，有如下特点：

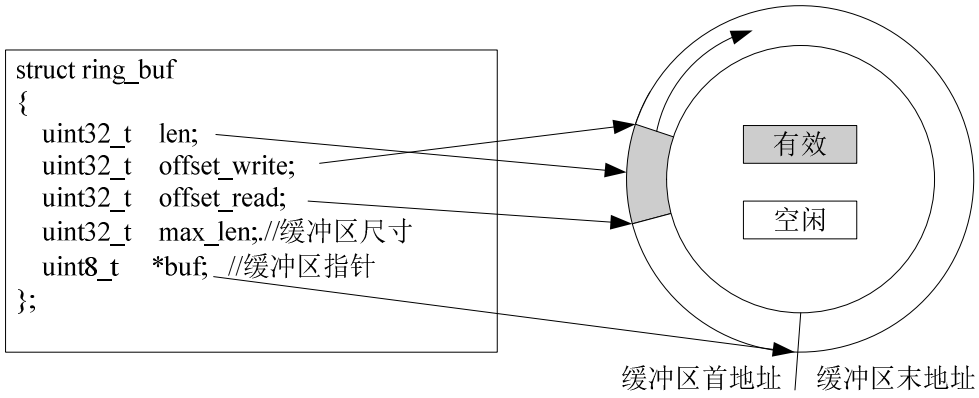


图 13-1 环形缓冲区示意图

1. 缓冲区数据结构只定义了一个字节池指针，即缓冲区空间是静态创建的（由用户提供和维护），初始化以后，这个指针指向实际的缓冲空间首地址。

2. offset_write 和 offset_read 记录的是下一个读(写)位置，是从缓冲区首地址开始的偏移量。

3. len 记录了缓冲区中的数据量。从 offset_write 和 offset_read 也可以获得 len 参数，但是这样的话，当 offset_write 和 offset_read 相等时，就无法判断是缓冲区满还是缓冲区空。使用 len 参数也可以加速缓冲区操作。

DJYOS 提供了一系列函数用于管理环形缓冲区，缓冲区的操作做了原子保护，因此执行以下并发操作是安全的：

1. A 线程读——B 线程写；
2. A 线程读——异步信号中断 ISR 写；
3. A 线程写——异步信号中断 ISR 读。

而以下的操作的结果是不可预料的：

1. A 线程读——B 线程读；
2. A 线程写——B 线程写；
3. A 线程读——异步信号中断 ISR 读；
4. A 线程写——异步信号中断 ISR 写。

13.1.1 API 说明

13.1.1.1 Ring_Init：创建环形缓冲

void Ring_Init(struct ring_buf *ring, u8 *buf, u32 len)

头文件：

os_inc.h

参数：

ring：环形缓冲区控制块指针。

buf：由用户提供的缓冲区空间。

len：缓冲区长度，以字节计。

返回值：

无。

说明：

建立环形缓冲区并初始化，使用这个函数之前，用户应该定义缓冲区内存块（buf）和缓冲

区数据结构(ring)。

13.1.1.2 Ring_ Capacity：获取缓冲容量

u32　Ring_Capacity(struct ring_buf *ring)
头文件：
os_inc.h
参数：
ring：环形缓冲区控制块指针。
返回值：
缓冲区尺寸，就是调用 Ring_Init 时使用的 len 参数。
说明：
获取环形缓冲区的总容量。

13.1.1.3 Ring_GetBuf：获取缓冲地址

u8　*Ring_GetBuf(struct ring_buf *ring)
头文件：
os_inc.h
参数：
ring：环形缓冲区控制块指针。
返回值：
环形缓冲区的存储区域首地址，就是调用 Ring_Init 的 buf 参数。
说明：
获取环形缓冲区的存储区域首地址，这个地址是用户调用 Ring_Init 时用户设定的地址。

13.1.1.4 Ring_Write：写缓冲区

u32　ring_write(struct ring_buf *ring, u8 *buffer, u32 len)
头文件：
os_inc.h
参数：
ring：环形缓冲区控制块指针。
buffer：待写入的数据指针。
len：待写入的数据长度.单位是字节数。
返回：
实际写入的字节数。
说明：
将 buffer 中的 len 大小的数据写入到环形缓冲区 ring。注意，写入到环形缓冲区的数据大小，受到缓冲区剩余尺寸的限制，实际写入的数据大小参见函数返回值。

13.1.1.5 Ring_Read：读缓冲区

u32 Ring_Read(struct ring_buf *ring, u8 *buffer, u32 len)
头文件：
os_inc.h
参数：
ring：环形缓冲区控制块指针。
buffer：接收数据的缓冲区指针。
len：待读出的数据长度.单位是字节数。
返回值：
实际读出的字节数。
说明：
从环形缓冲区 ring 获取 len 长度的数据到 buffer。注意，获取到数据的大小受到环形缓冲区实际数据大小的限制，实际获取到的数据参见函数返回值。

13.1.1.6 Ring_Check：检查缓冲使用量

u32 Ring_Check(struct ring_buf *ring)
头文件：
os_inc.h
参数：
ring：环形缓冲区控制块指针。
返回值：
环形缓冲区已缓存的数据量。
说明:
获取环形缓冲区中已缓存数据量，以字节为单位。

13.1.1.7 Ring_IsEmpty：缓冲是否空

bool_t Ring_IsEmpty(struct ring_buf *ring)
头文件：
os_inc.h
参数：
ring：环形缓冲区控制块指针。
返回值：
环形缓冲区为空返回 true；反之则为 false。
说明：
检查指定的环形缓冲区是否为空。

13.1.1.8 Ring_IsFull：缓冲是否满

bool_t Ring_IsFull(struct ring_buf *ring)

头文件：
os_inc.h
参数：
ring：环形缓冲区控制块指针。
返回值：
环形缓冲区已满返回 true；反之则为 false。
说明：
检查指定的环形缓冲区是否已满。

13.1.1.9 Ring_Flush：缓冲区重置

void Ring_Flush(struct ring_buf *ring)
头文件：
os_inc.h
参数：
ring：环形缓冲区控制块指针。
返回：
无。
说明：
重置环形缓冲区，即将缓冲区中的数据清零。

13.1.1.10 Ring_DumbRead：擦除“旧”数据

u32 Ring_DumbRead(struct ring_buf *ring,u32 len)
头文件：
os_inc.h
参数：
ring：环形缓冲区控制块指针。
len：需擦除旧数据的大小。
返回值：
实际擦除掉的旧数据的大小。
说明：
擦除环形缓冲区 len 长度的旧数据。所谓“旧”数据，是指在环形缓冲区缓存的数据中，相对于最近一次写操作的数据来说，早前写入且尚未被读取的数据，即以 offset_read 所指为起点的数据区域，大家知道，offset_read 指向的永远是环形缓冲区中最旧的数据。另外要注意，如果要删除的数据大于环形缓冲区中现存数据的长度，则以该现存数据长度为准。

13.1.1.11 Ring_RecedeRead：恢复数据

u32 Ring_RecedeRead(struct ring_buf *ring,u32 len)
头文件：
os_inc.h
参数：

ring：环形缓冲区控制块指针。
len：需恢复数据的大小。
返回值：
实际可恢复数据的大小。
说明：
将环形缓冲区中最近读取的 len 长度的数据，恢复为未读取，用户通过再次调用 Ring_Read 函数可以再次读取该段数据。但要注意，可重读数据的大小，受环形缓冲区写行为的影响。因为缓冲区是一个环形，之前被读取的数据区可能会被最近写入的数据覆盖，导致部分乃至全部数据无法恢复。所以实际可恢复数据的大小需参照函数的返回值。

13.1.1.12 Ring_SkipTail：擦除“新”数据

u32 Ring_SkipTail(struct ring_buf *ring, u32 size)
头文件：
os_inc.h
参数：
ring：环形缓冲区控制块指针。
size：需擦除的新数据大小。
返回值：
实际可擦除的新数据大小。
说明：
擦除环形缓冲区 len 长度的新数据。所谓“新”数据，是指环形缓冲区缓存的数据中，最近被写入的数据，即 offset_write 所指向的区域中的数据，而大家知道，offset_wrtie 指向的永远是环形缓冲区中最新的数据。但要注意，可删除的数据大小，受环形缓冲区读行为的影响。因为缓冲区是一个环形，最新写入的数据可能会被最近读操作给读走，导致部分乃至全部新数据已无法删除（因为已被读走）。所以实际可删除的新数据的大小需参照函数的返回值。

13.1.1.13 Ring_SearchCh：检索字符

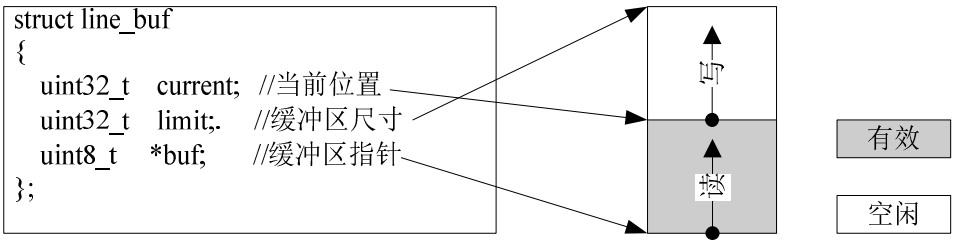
u32 Ring_SearchCh(struct ring_buf *ring, char c)
头文件：
os_inc.h
参数：
ring：环形缓冲区控制块指针。
c：需查找的字符。
返回值：
查找成功，返回字符 C 首次出现在环形缓冲区中的相对于最旧数据的偏移量；查找失败则返回 CN_LIMIT_UINT32。
说明：
从环形缓冲区最旧的数据（offset_read 所指的数据）开始查找字符 C，查找比对成功，则返回该字符相对于最旧数据且首次出现的位置。

13.1.1.14 Ring_SearchStr：检索字符串

u32　Ring_SearchStr(struct ring_buf *ring, char *string,u32 str_len)
头文件：
os_inc.h
参数：
ring：环形缓冲区控制块指针。
string：需查找的字符序列，并非'\0'结束的字符串。
str_len：字符序列长度。
返回值：
查找成功，返回字符序列 string 首次出现相对最旧数据的偏移量；失败，则返回 CN_LIMIT_UINT32。
说明：
从环形缓冲区最旧的数据（offset_read 所指的数据）开始查找字符序列 string，查找比对成功，则返回该字符相对于最旧数据且首次出现的位置。

13.2 线性缓冲区

线性缓冲区也是一种类似于 FIFO 的缓冲区，它实质上是环形缓冲区的简化版。它与环形缓冲区的区别在于数据不发生环绕，它的读指针总是在缓冲区的起始地址。在实际应用中，经常会有这样的需求，用一个缓冲区收集数据，数据量积累到一定数量或者超时时间到后，再一次处理完所有数据。对于这种应用，使用环形缓冲区虽然也可以满足要求，但环形缓冲区有数据环绕的问题，环绕就是指针到达缓冲区末地址后又从缓冲区首地址的过程。读写环形缓冲区都需要判断数据是否发生环绕，这是很消耗时间的。而使用线性缓冲区则没有这个问题，速度比环形缓冲区快很多。
线性缓冲区每次读都要求完全拷贝缓冲区中所有数据，然后把读指针清零，不支持读任意长度数据，读和写不能异步进行。如果允许不完全读取，读函数把剩余的数据拷贝到缓冲区头部，是一种更安全更通用的做法，但是，这就违背了建立线性缓冲区的初衷。线性缓冲区就是作为一个快速缓冲区来设计的，它省略了一切会影响效率的繁文缛节，拷贝剩余数据的过程将使线性缓冲区的速度比缓存缓冲区还慢，如果需要完全功能的缓冲区，使用环形缓冲区更加理想。



表格 13-1 环形缓冲区与线性缓冲区特性比较

项目	环形缓冲区	线性缓冲区
读写方式	先进先出	先进后出
读/写速度	慢	快

同步读写	支持	不支持
任意长度读	支持	必须一次读完所有数据
任意长度写	支持	支持

13.2.1 API 说明

13.2.1.1 Line_Init：创建线性缓冲

void Line_Init(struct line_buf *line, u8 *buf, u32 len)
头文件：
os_inc.h
参数：
line：线性缓冲区控制块指针，线性缓冲区控制块由用户提供。
buf：缓冲区指针，缓冲区由用户提供。
len：,缓冲区容量，单位是字节数。
返回值：
无。
说明：
建立线性缓冲区并初始化。注意，线性缓冲区控制块和缓存区都由用户设定。

13.2.1.2 Line_Capacity：获取缓冲容量

u32 Line_Capacity(struct line_buf *line)
头文件：
os_inc.h
参数：
line：线性缓冲区控制块指针。
返回值：
缓冲区容量。
说明：
获取线性缓存区的容量。

13.2.1.3 Line_SkipTail：擦除“新”数据

u32 line_skip_tail(struct line_buf *line, u32 len)
头文件：
os_inc.h
参数：
line：线性缓冲区控制块指针。
len：需删除的新数据大小。
返回值：
实际删除的新数据大小。

说明：
删除线性缓冲区中 len 大小的新数据。所谓“新”数据是以最近一次的写入的数据为基准，离基准越近的数据，则越新。当需删除的数据大小大于缓冲区中已缓存的数据大小时，则以已缓存的数据大小为准。实际删除了的数据量参照函数返回值。

13.2.1.4 Line_Write：写缓冲区

u32 Line_Write(struct line_buf *line, u8 *buffer, u32 len)
头文件：
os_inc.h
参数：
line：线性缓冲区控制块指针。
buffer：待写入数据的指针。
len：待写入数据的长度，单位是字节数。
返回：
实际写入到线性缓冲区的数据长度。
说明：
将 buffer 中 len 长度的数据写入到线性缓冲区。如果线性缓冲区的剩余空间不足，则参照函数返回值来获取实际写入的数据长度。

13.2.1.5 Line_Read：读缓冲区

u32 Line_Read(struct line_buf *line, u8 *buffer)
头文件：
os_inc.h
参数：
line：线性缓冲区控制块指针。
buffer：接收数据空间。
返回值：
从线性缓冲区中读出的数据量。
说明：
将线性缓冲区中的数据全部读出。

13.2.1.6 Line_GetBuf：获取缓冲地址

u8 *Line_GetBuf(struct line_buf *line)
头文件：
os_inc.h
参数：
line：线性缓冲区控制块指针。
返回值：
线性缓冲区的缓存空间指针。
说明：

获取线性缓冲区的缓存数据空间的指针。

13.2.1.7 Line_Check：检查缓冲使用量

u32 Line_Check(struct line_buf *line)
头文件：
os_inc.h
参数：
line：线性缓冲区控制块指针。
返回值：
线性缓冲区中已缓存的数据量，单位是字节。
说明:
获取线性缓冲区中的已缓存的数据量。

13.2.1.8 Line_IsEmpty：缓冲是否空

bool_t Line_IsEmpty(struct line_buf *line)
头文件：
os_inc.h
参数：
line：线性缓冲区控制块指针。
返回：
线性缓存区为空，返回 true；否则返回 false。
说明：
检查线性缓冲区是否为空。

13.2.1.9 Line_IsFull：缓冲是否满

bool_t Line_IsFull(struct line_buf *line)
头文件：
os_inc.h
参数：
line：线性缓冲区控制块指针。
返回值：
线性缓存区已满，返回 true；否则返回 false。
说明：
检查线性缓冲区是否已满。

13.2.1.10 Line_Flush：缓冲区重置

void Line_Flush(struct line_buf *line)
头文件：

os_inc.h
参数：
line：线性缓冲区控制块指针。
返回值：
无。
说明：
重置线性缓冲区。

13.2.1.11 Line_SearchCh：检索字符

u32　Line_SearchCh(struct line_buf *ring, char c)
头文件：
os_inc.h
参数：
line：线性缓冲区控制块指针；
c：需查找的字符。
返回值：
查找成功，返回 C 相对于缓存数据底部的偏置；否则返回 CN_LIMIT_UINT32。
说明：
从线性缓冲区中从缓存数据底部开始查找字符,成功检索到该字符后返回该字符相对于缓存数据底部且首次出现的偏置。

13.2.1.12 Line_SearchStr：检索字符串

u32　Line_SearchStr(struct line_buf *line, char *string, u32 str_len)
头文件：
os_inc.h
参数：
line：线性缓冲区控制块指针；
string：需查找的字符序列，并非’\0’结束的字符串；
str_len：字符序列长度。
返回值：
查找成功，返回字符序列 string 相对于缓存数据底部的偏置；否则返回 CN_LIMIT_UINT32。
说明：
从线性缓冲区中从缓存数据底部开始查找字符序列 string,成功比对到该字符序列后，返回该字符序列相对于缓存数据底部且首次出现的偏置。

第14章　标准输入

DJYOS 的标准输入管理模块，可以实现：
1. 键盘、二维鼠标、三维鼠标、串行终端、Telnet 终端等输入。
2. 支持多鼠标、多键盘。

3. 允许多键盘、鼠标控制同一个光标，也可以每个键盘、鼠标控制各自的光标。多个键盘、鼠标与多个光标之间，可以任意组合。

DJYOS 的输入管理模块是系统一个独立的功能组件，适合于数据量少、允许一定延迟的人机交互数据的传输，是人机接口的一部分，用于连接外部人机接口硬件的输入，例如鼠标、触摸屏、键盘等。它可以连接多个鼠标、键盘等设备，典型结构图 14-1 所示。

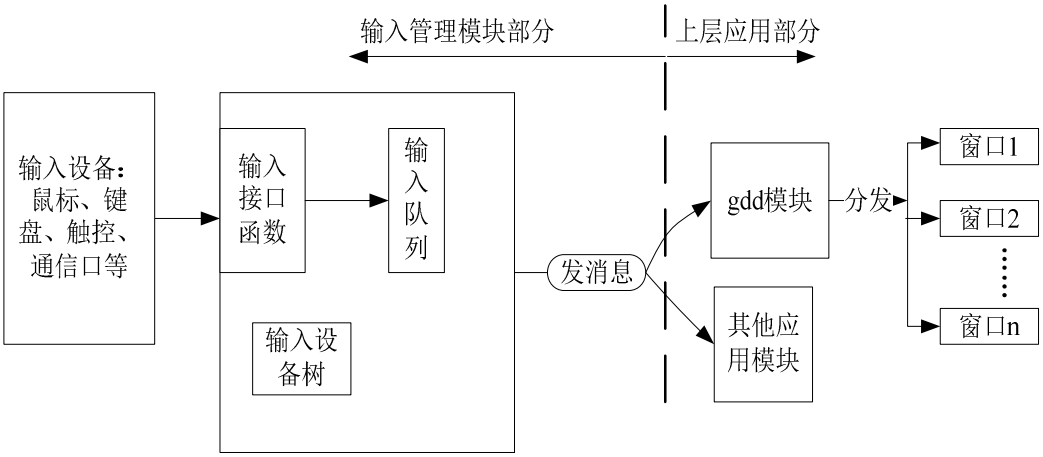


图 14-1 输入设备模型

以键盘输入为例，按键动作从硬件检测到通过标准输入模块到达应用程序的过程如下：

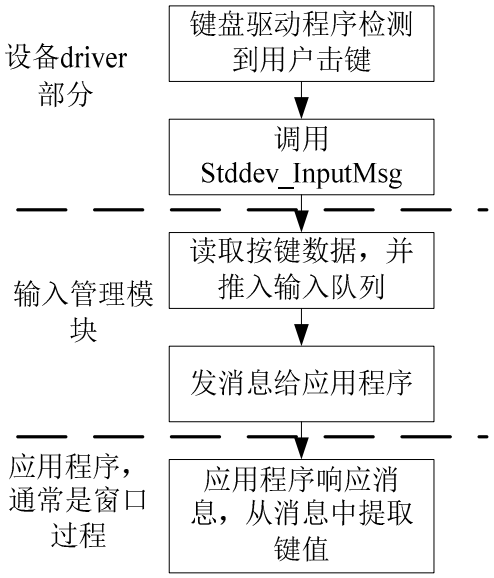


图 14-2 输入设备的数据流

输入消息（例如鼠标点击）产生后，将通过“消息队列”传送给应用程序。消息队列可以有多个，可以为每个输入设备单独指定消息队列，也可以多个设备对应一个消息队列。总之，消息队列和输入设备之间的对应关系，是任意的。标准输入模块初始化时，创建了一个默认消息队列，新创建的设备，都指向这个默认消息队列。应用程序可以从默认消息队列中获取消息，也可以指定具体的消息队列。

虽然鼠标、键盘等名叫“标准输入设备”，但并没有按照泛设备来设计。

版权所有，转载必须注明出处，注明：来自 www.djyos.com 字样。

14.1 API 说明

14.1.1 Stddev_ModuleInit：初始化标准设备模块

ptu32_t　Stddev_ModuleInit(ptu32_t para)
头文件：
stddev.h
参数：
para：未使用（保留）。
返回值：
1。
说明：
初始化标准设备功能模块，即创建标准设备挂载点。

14.1.2 Stddev_InstallDevice：安装标准输入设备

```
s32　Stddev_InstallDevice (char *device_name,
                           enum_STDIN_INPUT_TYPE_ stdin_type,
                           void *myprivate);
```

头文件：
stddev.h
参数：
device_name：输入设备名，例如“char_term”字符终端。
stdin_type：输入设备类型。
myprivate：输入设备私有数据结构。
返回值：
安装成功，返回标准输入设备 ID；否则返回-1。
说明：
将设备 device_name 安装为一个标准输入设备。

14.1.3 Stddev_SetFocus：设置输入通道

bool_t　Stddev_SetFocus(char *device_name, tagInputMsgQ FocusMsgQ)
头文件：
stddev.h
参数：
device_name：输入设备名。
FocusMsgQ：输入通道。
返回值：
设置成功返回 true；失败则返回 false。
说明：
设置输入设备（device_name）传递数据的输入通道。

14.1.4 Stddev_SetFocusDefault：设置缺省输入通道

void　Stddev_SetFocusDefault(tagInputMsgQ FocusMsgQ)
头文件：
stddev.h
参数：
FocusMsgQ：缺省输入通道。
返回值：
无。
说明：
设置标准输入设备的缺省输入通道。

14.1.5 Stddev_GetFocusDefault：查询缺省输入通道

tagInputMsgQ　Stddev_GetFocusDefault(void)
头文件：
stddev.h
参数：
无。
返回值：
标准输入模块的缺省输入通道。
说明：
查询标准输入设备的缺省输入通道。

14.1.6 Stddev_InputMsg：输入消息

bool_t　Stddev_InputMsg (s32 stdin_id, u8 *msg_data, u32 msg_size)
头文件：
stddev.h
参数：
stdin_id：输入设备 ID。
msg_data：输入消息。
msg_size：输入消息的长度，字节为单位。
返回值：
消息发送成功返回 true；否则返回 false。
说明：
将一个消息通过输入设备发送给系统。

14.1.7 Stddev_ReadMsg：获取输入消息

bool_t Stddev_ReadMsg(tagInputMsgQ InputMsgQ,

struct tagInputDeviceMsg *MsgBuf,
u32 TimeOut)
头文件：
stddev.h
参数：
InputMsgQ：输入设备 ID。
MsgBuf：输入消息。
TimeOut：等待输入消息时间。如果输入设备未发送消息，则当前事件阻塞等待，直至获取消息或者超时返回。
返回值：
消息发送成功返回 true；否则返回 false。
说明：
通过输入通道获取输入设备的信息。

14.1.8 Stddev_UnInstallDevice：卸载输入设备

bool_t Stddev_UnInstallDevice(char *device_name)
头文件：
stddev.h
参数：
device_name：输入设备名。
返回值：
卸载成功返回 true；失败则返回 false。
说明：
卸载一个标准输入设备 device_name，同时释放该设备所占据的内存。

14.1.9 Stddev_CreatInputMsgQ：创建输入通道

tagInputMsgQ Stddev_CreatInputMsgQ(u32 MsgNum, const char *Name)
头文件：
stddev.h
参数：
MsgNum：输入通道容量。
Name：参数保留，未使用。
返回值：
创建成功，返回输入通道句柄；失败则返回 NULL。
说明：
为输入设备创建一个输入通道。

14.1.10 Stddev_DeleteInputMsgQ：删除输入通道

bool_t Stddev_DeleteInputMsgQ(tagInputMsgQ InputMsgQ)
头文件：
stddev.h
参数：
InputMsgQ：输入通道句柄。
返回值：
删除成功返回 true；失败则返回 false。
说明：
删除一个输入设备的输入通道。

第15章 时间管理模块

DJYOS 维护两个时间量，分别是本地日历时间和系统运行时间。本地日历时间是用“从一个标准时间点（1970.1.1.0:0:0:0）以来的时间经过的秒数或微秒数”来表示的时间，其所在的时区是本地。而系统运行时间是从开机算起，系统的运行时间，其精度是微秒。日历时钟是可以手动调的，即使不主动地调，地球也会调，因为日历时钟跟地球自转是相关的，地球自转不是匀速的哦。所以，日历时间不是始终匀速向前的，可向前也可能向后“跳跃”，需要准确计算“经过”多少时间的场合，请不要使用日历时钟，而应该使用系统时钟。

15.1 本地日历时间

DJYOS 采用一个 64 位有符号数来统计本地日历时间，其最小计数单位是微秒。用户可以通过不同的 API 来获取以秒计或微秒计的日历时间。如果系统硬件提供了 RTC 功能，DJYOS 可以直接从硬件 RTC 获取日历时间，系统设计了相应的注册函数，方便用户移植。否则，系统的日历时间则以系统心跳计时，而每次系统上电时，都需要手动设置当时的本地日历时间。
下面的结构体 tagDjyTm 是日历时间的结构化表示，称作“分解时间”。

```
struct tagDjyTm
{
    s32 tm_us;/*微秒-取值区间[0,999999]*/
    s32 tm_sec;/* 秒－取值区间为[0,59]*/
    s32 tm_min;/* 分 - 取值区间为[0,59]*/
    s32 tm_hour;/* 时 - 取值区间为[0,23]*/
    s32 tm_mday;/* 一个月中的日期 - 取值区间为[1,31]*/
    s32 tm_mon;/* 月份（从一月开始，0 代表一月） - 取值区间为[0,11]*/
    s32 tm_year;/* 年份，其值从 1900 开始。*/
    s32 tm_wday;/* 星期－取值区间为[0,6]，其中 0 代表星期天，1 代表星期一，以此类推。*/
    s32 tm_yday;/* 从当年的 1 月 1 日开始的天数－取值区间为[0,365]，其中 0 代表 1 月 1
```

```
日，1 代表 1 月 2 日，以此类推。*/
    s32 tm_isdst;/* 夏令时标识符，实行夏令时的时候，tm_isdst 为正。不实行夏令时的进
候，tm_isdst 为 0；不了解情况时，tm_isdst()为负。*/
};
```

如上所示，结构体的大部分成员与常见“分解时间”结构体一样，但增加了一个 tm_us 成员，用于增加日历时间的精度。不过，这个精度能否实现依赖于实际系统配置情况，要看相应的 BSP 设计手册。

15.1.1 硬件 RTC 注册

系统硬件提供了 RTC 功能，需要用户将硬件 RTC 注册到时间管理模块，并对日历时间进行调整。具体步骤如下：
1.编写 RTC 驱动，__rtcdev_gettime 和__rtcdev_settime，实现对硬件 RTC 的读写。
2.RTC 设备注册，将 RTC 驱动程序链接到进时间管理模块。
3.更新 RTC 时间。

15.1.2 日历时间操作接口

15.1.2.1 Tm_GmTimeUs：换算格林威治分解时间

struct tagDjyTm *Tm_GmTimeUs(s64 *time)
头文件：
os_inc.h
参数：
time：64 位数表示的日历时间，缺省值为系统的当前日历时间，单位是微秒。
返回值：
格林威治时区分解时间，精度微秒级。
说明：
将用户输入日历时间转换成格林威治时区的分解时间。如果 time == NULL，函数默认使用本地日历时间，整个时间换算精度是微秒。
注意，由于函数返回值共用一块静态缓冲区，多事件或线程调用本函数，会造成新数据覆盖旧数据的复杂情况，因此不建议多事件或线程情况下，调用此函数。如果要调用，配合互斥锁一起使用。

15.1.2.2 Tm_GmTimeUs_r：可重入换算格林威治分解时间

struct tagDjyTm *Tm_GmTimeUs_r(s64 *time, struct tagDjyTm *result)
头文件：
os_inc.h
参数：

time：64 位数表示的日历时间，缺省值为系统的本地日历时间，精度微秒级。
result：格林威治时区的分解时间，精度微秒级；注意，指针不能为空。
返回值：
指向时间转换结果，格林威治时区的分解时间，精度微秒级。注意指针不能为空。
说明：
将用户输入的日历时间转换成格林威治时区的分解时间，如果 time == NULL，函数默认使用本地日历时间，整个转换精度微秒级。与上面的 Tm_GmTimeUs 不同的是，由于返回值空间由用户传入，是线程安全的。

15.1.2.3 Tm_GmTime：换算格林威治分解时间

struct tagDjyTm *Tm_GmTime(s64 *time)
头文件：
os_inc.h
参数：
time：64 位数表示的日历时间，缺省值为系统的本地日历时间，精度秒级。
返回值：
分解时间，精度秒级。
说明：
将用户输入日历时间转换成格林威治时区的分解时间，如果 time == NULL，函数默认使用本地日历时间，整个换算精度是秒。另外，由于函数实现时采用了静态方式（返回值共用一块静态缓冲区），多事件或线程调用本函数，会造成新数据覆盖旧数据的复杂情况，因此不建议多事件或线程情况下，调用此函数。如果要调用，配合互斥锁一起使用。

15.1.2.4 Tm_GmTime_r：可重入换算格林威治分解时间

struct tagDjyTm *Tm_GmTime_r(s64 *time, struct tagDjyTm *result);
头文件：
os_inc.h
参数：
time：64 位数表示的日历时间，缺省值为系统的本地日历时间，精度秒级。
result：指向时间转换结果，即格林威治时区的分解时间，精度秒级；注意，指针不能为空。
返回值：
格林威治时区的分解时间，精度秒级。
说明：
将用户输入的日历时间转换成格林威治时区的分解时间，如果 time==NULL，函数默认使用本地日历时间，整个换算精度秒级。与上面的 Tm_GmTime 不同的是，本函数是线程安全的。

15.1.2.5 Tm_LocalTime：换算同时区分解时间

struct tagDjyTm *Tm_LocalTime(s64 *time)
头文件：
os_inc.h

参数：
time：64 位数表示的日历时间。精度是秒级。
返回值：
在相同时区下，输入日历时间的分解时间。
说明：
将用户输入的日历时间转换成相同时区下的分解时间，如果输入的时间是以本地时区为基准，则返回值就是本地时区的分解时间。时间换算精度为秒。另外，由于函数返回值共用一块静态缓冲区，多事件或线程调用本函数，会造成新数据覆盖旧数据的复杂情况，因此不建议多事件或线程情况下，调用此函数。如果要调用，配合互斥锁一起使用。

15.1.2.6 Tm_LocalTime_r 可重入换算同时区分解时间

struct tagDjyTm *Tm_LocalTime_r(s64 *time, struct tagDjyTm *result)
头文件：
os_inc.h
参数：
time：64 位数表示的日历时间，缺省值为系统的本地日历时间，精度秒级；
result：指向在相同时区下，输入日历时间的分解时间。指针不能为空。
返回值：
输入日历时间的分解时间
说明：
将用户输入的日历时间转换成相同时区下的分解时间，如果输入的时间是以本地时区为基准，则返回值就是本地时区的分解时间。时间换算精度为秒。与函数 Tm_LocalTime，本函数是线程安全的。

15.1.2.7 Tm_LocalTimeUs：换算同时区分解时间

struct tagDjyTm *Tm_LocalTimeUs(s64 *time_us)
头文件：
os_inc.h
参数：
time：64 位数表示的日历时间。精度是微秒。
返回值：
在相同时区下，输入日历时间的分解时间，精度是微秒。
说明：
将用户输入的日历时间转换成相同时区下的分解时间，如果输入的日历时间是以本地时区为基准，则返回值就是本地时区的分解时间。时间转换精度为微秒。另外，由于函数返回值共用一块静态缓冲区，多事件或线程调用本函数，会造成新数据覆盖旧数据的复杂情况，因此不建议多事件或线程情况下，调用此函数。如果要调用，配合互斥锁一起使用。

15.1.2.8 Tm_LocalTimeUs_r：可重入换算同时区分解时间

struct djy_tm *Tm_LocalTimeUs_r(s64 *time, struct djy_tm *result);

头文件：
os_inc.h
参数：
time：指向 64 位表示的日历时间。
result：指向在相同时区下，输入日历时间的分解时间。指针不能为空。
返回值：
在相同时区下，输入日历时间的分解时间。
说明：
将用户输入的日历时间转换成相同时区下的分解时间，如果输入的时间是以本地时区为基准，则返回值就是本地时区的分解时间。时间转换精度为微秒。与函数 Tm_LocalTimeUs，本函数是线程安全的。

15.1.2.9 Tm_SetDateTimeStr：设置本地日历时间

s32 Tm_SetDateTimeStr(char *buf)
头文件：
os_inc.h
参数：
buf：指向格式例如“2011/10/28,22:37:50”字符串所表示的时间。
返回值：
设置成功返回 1；否则返回错误码。
说明：
用格式为"2011/10/28,22:37:50"的字符串所表示的时间设置系统本地日历时间。注意设置的时间精度是秒。如果有硬件 RTC，新时间也会更新硬件。

15.1.2.10 Tm_SetDateTime：设置本地日历时间

void Tm_SetDateTime(struct tagDjyTm *tm)
头文件：
os_inc.h
参数：
tm，分解时间。
返回值：
无
说明：
用分解时间格式更新（设置）系统本地日历时间，在具备硬件 RTC 的系统中，如果硬件直接输出分解时间的，硬件驱动调用本函数。因为日历时间只精确到秒，因此 tm 中的 us 成员，由硬件驱动自行维护，本函数不读取。如果硬件不维护 us 成员(调用 tm_connect_rtc 时参数 get_rtc_hard_us=NULL)，rtc driver 须使用中断方式，以确保秒跳变时立即调用 get_rtc_hard_us，系统将以此时刻为起点计算 us 数。

15.1.2.11 tm_inc_second

void tm_inc_second(u32 inc)
头文件：
os_inc.h
参数：
inc：增加的秒数
返回值：
无
说明：
日历时间+若干秒，有 rtc 硬件的系统中，如果只是计秒，而不输出日历的，硬件驱动调用本函数。因为日历时间只精确到秒，因此 tm 中的 us 成员，由硬件驱动自行维护，本函数不读取。如果硬件不维护 us 成员(调用 tm_connect_rtc 时参数 get_rtc_hard_us=NULL)，rtc driver 须使用中断方式，以确保秒跳变时立即调用 get_rtc_hard_us，系统将以此时刻为起点计算 us 数。

15.1.2.12 Tm_AscTime：转换时间为“字符串”

void Tm_AscTime(struct tagDjyTm *tm, char buf[])
头文件：
os_inc.h
参数：
tm：分解时间；
buf：用来返回结果的数组地址，至少 21 字节。
返回值：
以格式“年/月/日,时:分:秒”显示的字符串。
说明：
把一个分解时间换算成“字符串”时间，时间单位为秒。“字符串”时间格式为“年/月/日,时:分:秒”。

15.1.2.13 Tm_AscTimeMs：转换时间为“字符串”

void Tm_AscTimeMs(struct tagDjyTm *tm, char buf[]);
头文件：
os_inc.h
参数：
tm：分解时间；
buf：用来返回结果的数组地址，至少 24 字节。
返回值：
以格式“年/月/日,时:分:秒：毫秒”显示的字符串。
说明：
把一个分解时间换算成“字符串”时间，时间单位是毫秒，“字符串”时间格式“年/月/日,

时:分:秒:毫秒”。

15.1.2.14 Tm_AscTimeUs：转换时间为“字符串”

void　Tm_AscTimeUs(struct tagDjyTm *tm, char buf[]);
头文件：
os_inc.h
参数：
tm：分解时间
buf：用来返回结果的数组地址，至少 27 字节。
返回值：
以格式“年/月/日,时:分:秒:毫秒:微秒”显示的字符串。
说明：
把一个分解时间换算成“字符串”时间，时间单位是微秒，“字符串”时间格式“年/月/日,时:分:秒:毫秒:微秒”。

15.1.2.15 TM_Time：获取本地日历时间

s64　TM_Time(s64 *ret)
头文件：
os_inc.h
参数：
ret：指向用于存放系统的本地日历时间的内存空间，指针为空，则无效。
返回值：
本地日历时间。
说明：
获取系统的本地日历时间，时间单位为秒。

15.1.2.16 TM_TimeUs：获取本地日历时间

s64　TM_TimeUs(s64 *ret);
头文件：
os_inc.h
参数：
ret：指向用于存放系统的本地日历时间的内存空间，指针为空，则无效。
返回值：
本地日历时间。
说明：
获取系统的本地日历时间，时间单位为秒。

15.2 系统运行时间

15.2.1 系统运行时间接口

15.2.1.1 Rtc_RegisterDev：注册设备

bool_t Rtc_RegisterDev(__rtcdev_gettime gettimefunc,
__rtcdev_settime settimefunc)
头文件：
os_inc.h
参数：
gettimefunc：函数指针，用于实现读取硬件 RTC 模块时间,不能为空；
settimefunc：函数指针，用于实现设置硬件 RTC 模块时间，不能为空。
返回值：
注册成功返回 true；否则返回 false
说明：
将硬件 RTC 驱动函数__rtcdev_gettime 和__rtcdev_settime 注册进系统的时间管理模块。

15.2.1.2 DjyGetTime：查询当前系统运行时间

s64 DjyGetTime(void)
头文件：
os_inc.h
参数：
无。
返回值：
当前系统已运行时间。
说明：
获取自系统开机以来的运行时间，单位是微秒。

15.2.1.3 DjyGetTimeTick：查询当前系统心跳计时

s64 DjyGetTimeTick(void)
头文件：
os_inc.h
参数：
无。
返回值：
当前系统已运行时间，以系统滴答（tick）计。
说明：
获取自系统开机以来的运行时间，以系统滴答（tick）计，单位是微秒

第16章　看门狗

16.1 看门狗原理

16.1.1 看门狗的硬件

为了提高系统可靠性，嵌入式系统一般都会设计看门狗电路，在软件严重错误、或者硬件受干扰而使系统运行出错的时候，看门狗电路将输出复位信号重置系统。看门狗有 CPU 内嵌和外置两种，无论哪一种看门狗电路，其原理不外乎如图 16-1 所示。系统正常运行时，会定期地执行“喂狗”动作，即将看门狗时钟清零。一旦系统出现异常，无法“喂狗”，看门狗会因其时钟超时而产生超时信号，而这个信号一般作为系统的复位信号。

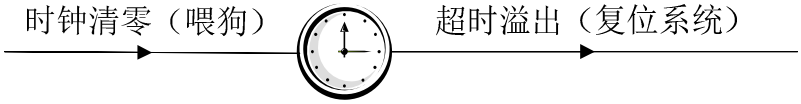


图 16-1 看门狗电路设计原理

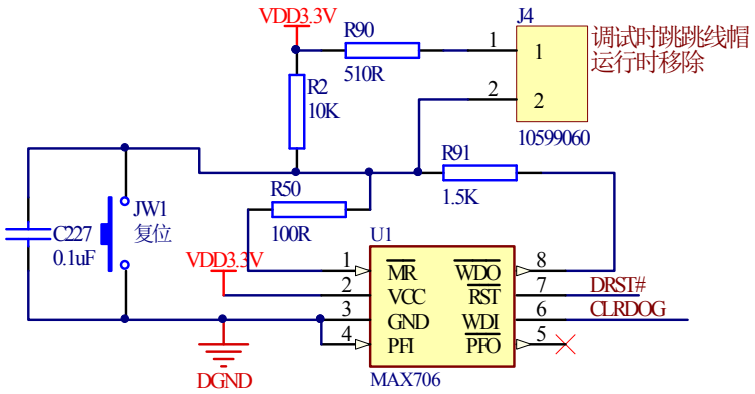


图 16-2 典型外置看门狗电路

图 16-2 是典型的外置看门狗电路图，MAX705 内置两个定时器，一个用于控制看门狗溢出时间，时间常数固定为 1.6 秒，WDI 信号的每次改变都会清零定时器，如果 wdt 信号连续 1.6 秒没有变化时，MAX706 的 7 脚就输出一个低电平脉冲表示看门狗溢出，该信号用于复位计算机。另一个定时器控制看门狗溢出脉冲的宽度，时间常数固定为 200mS。与大多数看门狗芯片一样，MAX705 兼具电源监测功能，图中没有画出。也有一些看门狗芯片的定时时间是可以调整的，例如 MAX691，绝大多数 MCU 内置的看门狗，也是可调整的。插上跳线 J4 可以禁止看门狗复位，用于调试。顺带提一下，有些人设计看门狗电路时，把 J4 设计成调试时拔掉，运行时插上，这是不对的。跳线帽是机械连接，可能会接触不良，且生产时要多一道插跳线帽的工序，这道工序是人工操作，易遗漏，且遗漏后难于检测。

16.1.2 看门狗的软件原理

计算机系统有大有小，有复杂有简单，但是对于使用看门狗的软件程序，当我们抽其脉络，发现无异于如图 16-3 所示。不同的是分支的多少、条件判断的复杂度、各种循环执行的次数和频度，以及中断发生的频度等。软件运行过程中，会有一些关键点，只要这些关键路径

版权所有，转载必须注明出处，注明：来自 www.djyos.com 字样。

正常执行到，基本上可以保证软件是正常的，而看住这些关键点往往需要看门狗的协助，看门狗的数量以及看门狗资源的竞争往往成了软件设计的一个关键点。

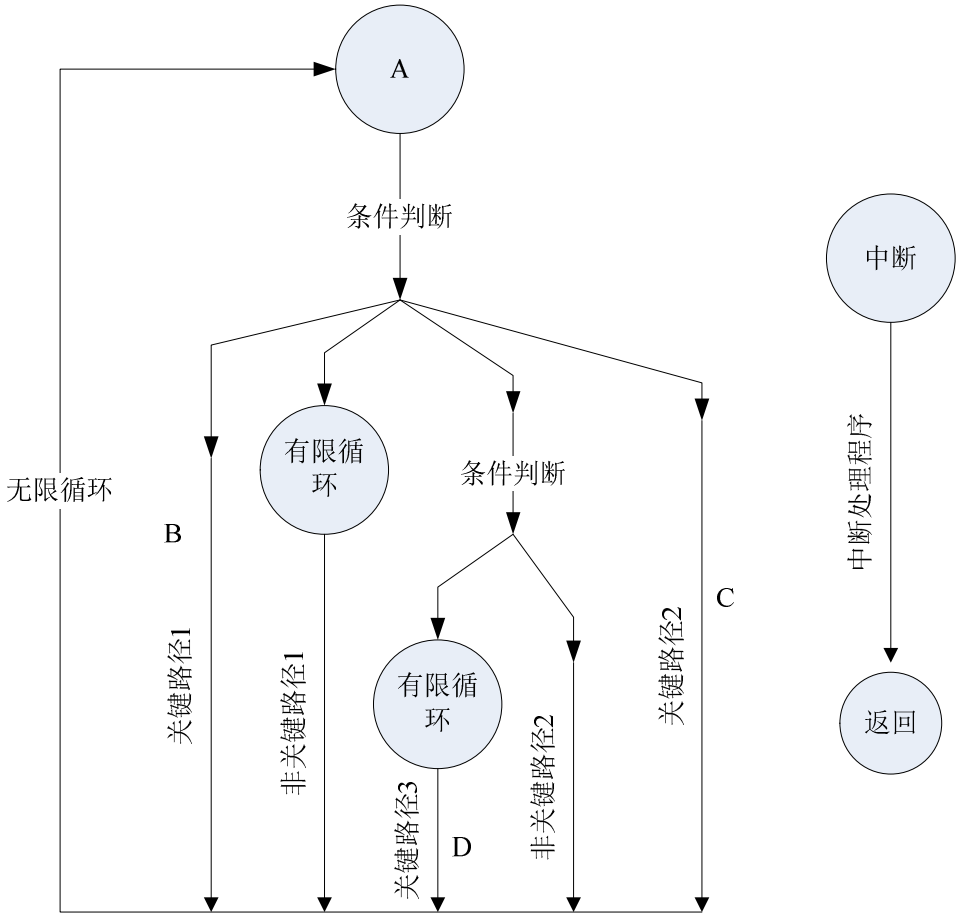


图 16-3 使用看门狗程序脉络

16.2 启动过程的看门狗

16.2.1 尴尬的看门狗

启动过程，是硬件看门狗系统长久以来的难言之隐，直到 DJYOS 出现，才得以一洗了之。硬件看门狗定时器计满即会输出复位脉冲，硬件启动计时的时机有两种，一是上电即启动计时，外置硬件看门狗一般属于这种；二是上电时不启动，需要软件触发一下才启动，启动后则不能关闭，MCU 内置的看门狗一般属于这种形式。
很多时候，软件都有个加载和启动过程，在操作系统支持下，尤其如此。这期间正常的喂狗程序往往难于执行，这段时间超过了看门狗的溢出周期怎么办？系统初始化未完成，狗就忍不住要叫，让你的系统无休止地复位，怎么办？

16.2.2 启动代码中喂狗

显然，如果在启动过程中就按时喂狗，问题便烟消云散了。但这往往是一厢情愿，个中苦衷，

荣我慢慢道来。
一般来说，系统启动前，首先要执行一个加载器，该加载器会 copy 代码到内存中（如果在 flash 中运行就免了），然后初始化 data 段（包含已初始化变量），最后把 bss 段（包含未初始化变量）清零，这个过程中，由于 wdt 代码未加载，喂狗代码并不能调用。如果是高度定制的系统，你可以先单独加载 wdt 模块，也可以在 startup 代码中也放置一份 wdt 代码。这里面，有两个问题：
1. 加载器本来是与硬件无关的，但如果你要在其中“夹塞”喂狗代码的话，就变成硬件的附庸了。
2. 裸跑系统还好说，如果使用了操作系统，你要找出它的加载器并“夹塞”喂狗代码进去，难度可想而知，特别是，不开源的系统你怎么办？
事情还没完呢，加载器虽然难搞，毕竟是固定的一部分代码，花点功夫还是能搞定。driver 以及中间件的初始化过程，如果超过了狗叫间隔，也会让你走投无路。如果你自己写的 driver，当然可以改；不是你自己写的，有源码，也可以改；但如果是外购的中间件，没源码，你怎么办？即使你可以改，也不代表没问题，如果所有 driver 都跟 wdt 相关，你的整个代码，就会是一张蜘蛛网，各个模块耦合在一起。“低耦合、高内聚”的教诲，你忘了么？

16.2.3 推迟启动看门狗

既然不能在启动过程喂狗，那让看门狗在启动过程闭嘴行不？
推迟启动看门狗计时器，直到初始化完成，这是最容易、最直观的方式了，有一部分看门狗硬件有这种功能，它允许用软件往特定的寄存器写特定的数值后，才能启动，当然，启动以后是不能再关闭的，除非复位，硬件工程师不约而同地把看门狗设置成这样。
这种方案可靠吗？
只要不启动看门狗计时器，狗是永远不会叫的，加载和初始化可以正常完成。但问题是，加载和初始化过程可能出错，导致系统将无法启动怎么办？而此时看门狗尚未开启，狗是永远不会叫的，系统将永远死寂一片。不要以为这是危言耸听，这种事情确实发生过。有些硬件设计考虑不周，或者受到现场骚扰信号干扰，上电复位不彻底，导致在上电过程使系统进入死锁。
另一个问题是，看门狗计时器启动后不能关闭，虽然复位可以关闭它，但需要软件重启怎么办？此时需要执行“漫长”的初始化过程，既不喂狗又没有复位，忠诚的看门狗一定会在你初始化到一半的时候，狂吠起来，你又万劫不复。

16.2.4 启动过程硬件喂狗

不推迟启动看门狗计时器，也不在启动代码中喂狗，而是安排专门的硬件，在启动过程喂狗，看上去很完美。
喂狗的硬件，可以是 CPU 自带的 PWM，许多嵌入式 CPU 提供这样的功能，但这样一来，跟“推迟启动看门狗计时器”有什么区别呢？毫无区别！
改进的方法，是使用 CPLD 或 FPGA 这类可编程器件，实现智能化喂狗，即上电（或复位）后自动喂狗，但必须设计一个最后时限，如果最后时限到达时，初始化还没有完成，狗就必须叫了。系统初始化完成后，把 CPLD 的看门狗切换到正常模式。
看起来很美了，但事情并不是那么简单。
CPLD 增加的成本由谁买单？

系统软重启怎么办？
一个嵌入式应用系统，重启可能有三种情况，一是软重启，即软件自己跳到起始位置，重新加载和初始化；二是 CPU 复位而其他硬件不复位；三是整个硬件系统复位。
对于前两种情况，外部的 CPLD 是不知道的，看门狗会一如既往地计时，到点复位 CPU。那只能在重启前，用软件“命令”CPLD 重新进入自动喂狗模式。这样行吗？降低了可靠性啊，这等于程序运行中可以关闭看门狗啊。为什么所有 MCU 内置的看门狗都设计成启动后不能关闭，就是怕软件误操作把看门狗关闭了。

16.2.5 柳暗花明

前面所述的种种尴尬，在 DJYOS 面前，均不存在。回忆一下在 16.2.2 节，如果把“在启动代码中喂狗”改为“在启动过程中喂狗”，所有问题是否迎刃而解？那怎么才能在启动过程中喂狗呢？答案是中断。遗憾的是，许多操作系统启动过程都是关中断的，而且没有提供在启动前加载 wdt 模块到内存的功能，以致即使开中断，也无法使用。
DJYOS 提供两个机制，可以实现在启动过程中喂狗：
1. 只要在 lds 文件中把 wdt 代码放到预加载区域，就会在加载系统前被加载到内存。
2. 在系统加载和初始化过程中，实时中断是允许的。
那么，我们就可以这样来实现加载过程中“喂狗”，即在系统上电后，预加载一段代码专门用于实现“启动过程中喂狗”的代码。该代码启动看门狗的同时，额外启动再一个定时器用于看门狗的主动“喂狗”。系统开始正常运行后，用来喂狗的定时器就功成身退了。但启动定时器喂狗的那段代码还在内存中，如果你实在不放心，担心被误触发，可把那段代码删掉。
系统启动中喂狗整个工作原理如图 16-4 所示。

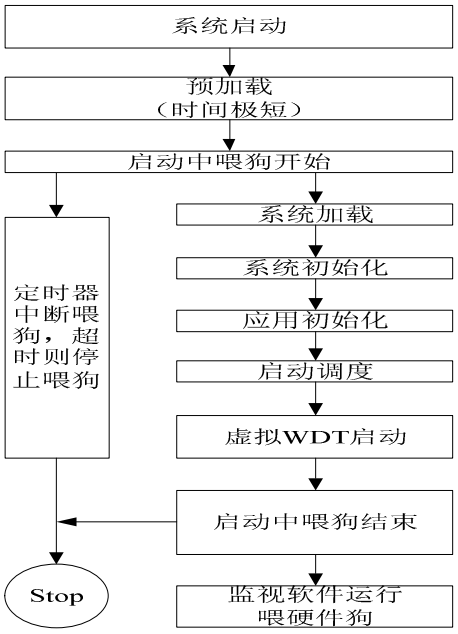


图 16-4 启动中喂狗原理

16.3 看门狗模块设计

16.3.1 功能

DJYOS 看门狗模块具有如下功能：
1. 监察应用程序是否定期执行到某关键位置。
2. 监察关键任务是否在规定的时间内完成。
3. 如有硬件看门狗，提供系统启动（加载和初始化）过程中安全喂狗的机制；
4. 允许用户独立设置各独立看门狗狗叫后的处理措施；
5. 与异常处理模块结合，狗叫事件将作为软件异常予以记录。
6. 如有硬件看门狗，可管理硬件看门狗。

可见，DJYOS 的看门狗模块，是一个独立的软件模块，它能管理硬件看门狗，但不依赖于硬件看门狗工作。即使用户硬件中没有配置看门狗，也能起到看家护院的作用，大大提高用户软件的可靠性。

16.3.2 基本架构

在层次上，本模块在软件上分为三个层次：芯片驱动层、看门狗硬件抽象层、看门狗虚拟层。芯片驱动层仅仅实现了本芯片的基本驱动，按照硬件抽象层的格式将本芯片注册进系统；硬件抽象层向下提供芯片注册的接口，在本层实现启动中的喂狗功能，对上提供所驱动的硬件芯片的基本信息；用户可根据需要看护的软件路径，获取无数的“看门狗”来满足用户的需求。整个设计层次如图 16-5 所示。

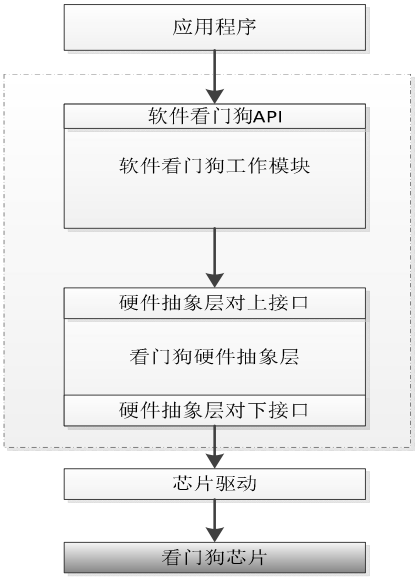


图 16-5 看门狗软件层次

16.3.3 看门狗硬件接口层

16.3.3.1 硬件抽象层概述

设计此层的目的是为了让看门狗模块能够实现硬件上的隔离，同时使我们设计的模块很好地满足跨平台移植的需求。本层有两个主要功能：1，作为硬件的转接口，向下提供芯片注册接口，向上提供获取硬件参数接口；2，实现启动过程中的喂狗。

WdtHal_RegisterWdtChip 负责将我们的硬件看门狗注册进系统的内核中，在此之前，我们应该保证看门狗已经初始化完毕。该 API 的参数包括三项：芯片名字（可以没名字，NULL）、看门狗周期、喂狗方法。注册之后，看门狗组件就知道如何操作看门狗硬件了。

硬件看门狗的操作分为两个阶段：OS 内核启动前、OS 内核启动后。OS 内核启动前，即我们所谓的 OS 启动过程中的喂狗操作；OS 启动后，由系统的软件看门狗进行管理。在此我们谈谈启动过程中的喂狗，以及如何交给软件看门狗模块进行管理的。

在 Critical 的代码中，我们先调用 WdtHal_BootStart 该功能函数，该函数有一个唯一的参数：喂狗时间，如果为 0，表示在启动的过程程中不喂狗，否则代表在启动过程中的喂狗时间，在该 API 中，会启动一个定时器来触发实时中断来完成喂狗。当系统启动后，会启动软件看门狗模块，在该模块中会调用 WdtHal_BootEnd 来结束定时器的喂狗，后续硬件看门狗就交给软件看门狗模块来操作。

在此你可能有疑问：如果系统启动不成功怎么办？这个依然在我们的掌控之中。还记得调用 WdtHal_BootStart 时指定的时间么，该时间就是实时中断喂狗的时间，超过该时间，定时器触发的实时中断将不再喂狗，硬件看门狗自然会复位整个系统。

16.3.3.2 硬件抽象层 API

16.3.3.2.1 WdtHal_RegisterWdtChip:注册看门狗

```
bool_t   WdtHal_RegisterWdtChip(char *chipname,
                                u32 yipcycle,
                                bool_t (*wdtchip_feed)(void))
```

头文件：
os_inc.h
参数：
chipname：芯片名称，无名称则为 NULL；
yipcycle：看门狗“喂狗”周期，
wdtchip_feed：实现“喂狗”服务的函数指针。
返回值：
注册成功返回 true；失败则为 false。
说明：
注册看门狗，包括看门狗名称、“喂狗”周期和“喂狗”服务函数。看门狗注册成功就意味着看门狗已经开始工作，因此在注册看门狗（调用本函数）之前，务必要完成对看门狗的初始化工作。

16.3.3.2.2 WdtHal_GetChipPara：查询芯片信息

bool_t　WdtHal_GetChipPara(struct tagWdtHalChipInfo *hardpara)
头文件：
os_inc.h
参数：
hardpara：获取硬件芯片参数。
返回：
成功返回 true；失败则为 false。
说明：
获取看门狗芯片的参数，即注册的信息内容。

16.3.3.2.3 WdtHal_BootStart：开启“启动喂狗”服务

bool_t　WdtHal_BootStart(u32 bootfeedtime);
头文件：
os_inc.h
参数：
bootfeedtime：在启动加载过程中主动喂狗时长，单位为微秒。
返回值：
开启成功返回 true；失败或者无需开启则返回 false。
说明：
开启启动加载过程的“喂狗”服务。参数 bootfeedtime 表示需“喂狗”的时长，实际就是启动加载过程所经历时间长度。系统会根据实际的看门狗周期对这个时间长度向上取整为看门狗周期的倍数。如果用户不想在启动加载过程中启动“喂狗”服务，则将输入参数置为零。
注意，本函数依赖系统的时钟模块，即 CN_CFG_SYSTIMER == 1。

16.3.3.2.4 WdtHal_BootEnd：关闭“启动喂狗”服务

bool_t　WdtHal_BootEnd(void);
头文件：
os_inc.h
参数：
无。
返回值：
开启关闭返回 true；失败则返回 false。
说明：
关闭启动加载过程中的“喂狗”服务，同时释放相关系统定时器。
注意，本函数依赖系统的时钟模块，即 CN_CFG_SYSTIMER == 1。

16.3.4 软件看门狗

16.3.4.1 软件看门狗概述

为什么要做软件看门狗？
在传统嵌入式领域，一般的都是直接使用硬件看门狗，但是这样无疑会有更大的负面作用：
1. 当无硬件看门狗时，使用看门狗的策略将无法实施，或者额外增加硬件导致成本增加；
2. 一般的硬件看门狗只有一只，多个关键任务共同使用这一只或者数量有限的硬件看门狗，导致交叉使用，造成任务同步困难；
3. 多任务交叉使用看门狗时,当某些关键任务运行不正常时，有可能不会产生用户所需要的效果，因为其他使用看门狗的任务有可能还在继续喂狗，导致看门狗功能性失效；
4. 传统意义上的看门狗，当狗叫时产生的效果一般只有复位系统，对于那些有能力通过一定手段修复所属任务出现的问题的软件需求而言，爱莫能助；
5. 传统意义上的硬件看门狗的周期一般是固定的，多个任务使用时其留有很大的余量（一般取最大的一个），导致其保护的任务失败时其响应不及时；
6. 传统意义上的看门狗，当看门狗叫时，只能简单的表示自己的任务超时，无法分析出该任务是因为自己的逻辑错误不能喂狗还是因为系统调度原因导致该任务无法及时喂狗。
以上这些问题，在使用 DJYOS 的看门狗时都会烟消云散。

16.3.4.2 DJYOS 的软件看门狗

在 DJYOS 中，所有的看门狗都由看门狗模块负责管理，不论是“硬件狗”还是“软件狗”，都是平等的，拥有相同的属性，工作机理一致，即规定时间内必须去喂狗，不然的话对应的狗就会叫。大家可能感觉晕晕乎乎，举个例子，大家可能就明白了：闹钟大家都知道，闹铃时间一到，就会响。怎么让闹铃不响呢，要么不开启，要么就是提前重置它。但现在的闹钟更强大了，可以设定多个闹铃时间，且某个闹铃时间到达之前，可以独立地重置掉它，即该时间点不闹铃。这么多闹铃时间在很多时候反映的是你的行动计划。比如闹铃时刻 6：00 前起床，闹铃时刻 7:00 前叫醒小孩，闹铃时刻 7:45 前是出门等等。我们设置多个闹铃时刻却并不需要多个闹钟，举例的从多闹铃时刻，我们就可以理解为“软件看门狗”了。狗叫后，允许你做一些补救措施（执行狗叫善后回调函数）。需要注意的是，那一只唯一的硬件狗和众多软件模拟的狗操作方式完全相同，硬件狗巧妙地利用了狗叫善后实现喂硬件狗！应用程序永远不要企图直接操作硬件狗，而是喂养软件狗，否则，硬件狗撑死了，就看不了门了。不论你用看门狗来控制关键代码的运行时间，还是看住关键代码会不会跑飞，都可以简单的认为：看门狗就是为了监控规定的任务有没有在规定的时间内完成！因此从这个本质的问题上而言，用看门狗的地方无疑都可以抽象成如图 16-6 所示的简单模型。启动一个定时器，在闹钟超时前必须喂狗，否则狗叫！狗叫后，你有机会补救，补救措施自己写（善后函数）。你可以用善后函数返回值告诉告诉看门狗模块，狗叫以后执行什么操作，也可以不提供善后函数，用看门狗参数直接告诉看门狗模块。
图 16-6 中，A 是用看门狗检查关键代码有没有定期执行到；B 除 A 的功能外，还检查执行结果是否正确；C 比 B 更彻底，检测到致命问题后，不但不喂狗，还要执行相应的错误处理程序；D 用于看住限期完成的任务。E 是一个善后处理函数，应用程序提供，由看门狗模块在检测到喂狗异常时调用。应用程序可以在此做各种善后操作，例如发出告警信息等，这个

函数在看门狗上下文中调用，使用看门狗模块的栈，最好不要在这里做复杂的工作，局部变量中不要定义缓冲区。

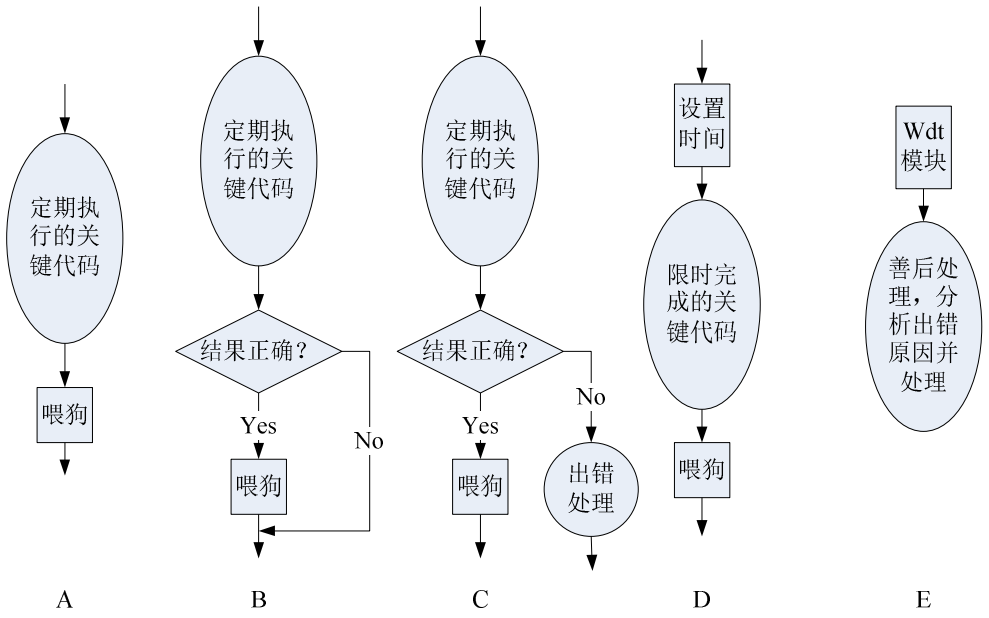


图 16-6 应用程序看门狗操作典型模型

16.3.4.3 软件看门狗 API

16.3.4.3.1 Wdt_Create：创建“软看门狗”

```
struct tagWdtRsc    *Wdt_Create(char *dogname,
                                u32 yip_cycle,
                                u32 (*yip_remedy)(struct tagWdtRsc *wdt),
                                ptu32_t yip_action,
                                struct tagWdtYipJudgeLevelPara *levelpara)
```

头文件：
wdt.h
参数：
dogname：“软看门狗”名称，不能为空；
yip_cycle：“喂狗”周期，单位是微秒，该值会自动向上取整为系统周期的整数倍；
yip_remedy：狗叫善后函数，即狗叫的钩子函数，该函数在看门狗发生超时后被执行。函数可以为空，意味着不需要善后函数。DJYOS 在调用完本函数之后，还将调用看门狗异常动作函数，本函数的返回值将作为该动作函数的参数。
yip_action：由本“软看门狗”引发的看门狗异常动作函数输入参数，注意，只有狗叫善后函数为空才有效，否则会造成参数冲突。
levelpara：该成员具体负责传递看门狗狗叫时候的各种判断标准，目前有以下三种（取值）：
1. consumed_time_level，以用户任务上次操作看门狗（写类操作）到狗叫发生之间的时间为界限，在此范围内认定为任务调度原因，超过此标准认定为逻辑错误导致狗叫；
2. yip_for_shedule_times_level，以任务调度原因导致的连读狗叫次数为标准，此范围内执行

记录，超出此范围执行 yip_action,仅仅对无狗叫钩子函数有效；
3. yip_for_logic_times_level，逻辑错误导致的连读狗叫次数标准，此范围内执行执行记录，超出此范围执行 yip_action,仅仅对无狗叫钩子函数有效；
返回值：
创建成功则返回“软看门狗”指针；否则返回 NULL。
说明：
该函数用于创建一个“软看门狗”，该看门狗的控制块空间由系统自行分配，因此不允许用户做擅自做任何的更改。如果没有在规定的周期内（yip_cycle）喂狗，就会认定该看门狗狗叫，执行相应动作。因此，请务必慎重设定周期，留给自己余量。
看门狗叫的钩子函数的返回值类型 enum exp_action 在 sys_exp_action.h 文件定义。具体行为可参看该文件注释。
注意：该 API 为对看门狗的写类操作！执行后自动喂狗一次！

16.3.4.3.2 Wdt_Create_r：可靠创建“软看门狗”

```
struct tagWdtRsc    *Wdt_Create_r(struct tagWdtRsc *wdt,
                                  char *dogname,
                                  u32 yip_cycle,
                                  u32 (*yip_remedy)(struct tagWdtRsc *wdt),
                                  ptu32_t yip_action,
                                  struct tagWdtYipJudgeLevelPara *levelpara)
```

头文件：
wdt_soft.h
参数：
wdt：“软看门狗”控制块指针，不能为空。
其余参数函数 Wdt_Create 与一致。
返回值：
创建的看门狗指针否则 NULL。
说明：
除了用户须自定义看门狗控制块外，功能与 Wdt _Create 同。注意，看门狗控制块不要使用局部变量定义。

16.3.4.3.3 Wdt_Delete：删除“软看门狗”

```
bool_t Wdt_Delete(struct tagWdtRsc *wdt)
```

头文件：
wdt_soft.h
参数：
wdt：被删除的“软看门狗”。
返回值：
删除成功返回 true；否则返回 false。
说明：
删除一只“软看门狗”，并释放“软看门狗”控制块占用的内存块。注意，如果在关调度期

间、或者在异步信号 ISR 中调用本函数，将不会成功。

16.3.4.3.4 Wdt_Delete_r：可靠删除“软看门狗”

bool_t Wdt_Delete_r(struct tagWdtRsc *wdt)
头文件：
wdt_soft.h
参数：
wdt：被删除的“软看门狗”。
返回值：
删除成功返回 true；否则返回 false。
说明：
删除一个由 Wdt_Create_r 创建的“软看门狗”。

16.3.4.3.5 Wdt_Pause：暂停“软看门狗”

struct tagWdtRsc *Wdt_Pause(struct tagWdtRsc *wdt)
头文件：
wdt_soft.h
参数：
wdt：“软看门狗”指针
返回值：
操作成功返回被操作的“软看门狗”，否则 NULL。
说明：
暂停一只“软看门狗”，实际上是通过把这个狗的狗叫间隔临时设为近似无穷大来实现的。

16.3.4.3.6 Wdt_Resume：重启“软看门狗”

struct tagWdtRsc *Wdt_Resume(struct tagWdtRsc *wdt)
头文件：
wdt_soft.h
参数：
wdt：“软看门狗”指针。
返回值：
操作成功返回被操作的“软看门狗”，否则 NULL。
说明：
恢复“软看门狗”运行。注意：下次狗叫时间是“当前时间+timeout”，而不是调用 wdt_pause 的剩余时间。

16.3.4.3.7 Wdt_Ctrl：设置“软看门狗”

bool_t　Wdt_Ctrl(struct tagWdtRsc *wdt, u32 type, ptu32_t para)
头文件：
wdt.h
参数：
wdt：“软看门狗”指针。
type：设置命令；
para：设置参数，参数含义依据设置命令。
返回值：
设置成功返回 true；否则返回 false。
说明：
设置“软看门狗”参数。设置命令如下表：

设置命令	描述
EN_WDT_CTRCMD_SETCYCLE	设置“软看门狗”狗叫周期
EN_WDT_CTRCMD_SETYIPACTION	设置“软看门狗”的狗叫后动作
EN_WDT_CTRCMD_SETYIPHOOK	设置狗叫钩子函数
EN_WDT_CTRCMD_SETCONSUMEDTIMELEVEL	设置狗叫时原因判断标准
EN_WDT_CTRCMD_SETSCHLEVEL	设置调度原因导致的连续狗叫次数忍耐限度
EN_WDT_CTRCMD_SETLOGICLEVEL	设置逻辑错误原因导致的连续狗叫次数忍耐限度

注意：该 API 为对看门狗的写类操作，如果在其他任务中执行该 API 会导致喂看门狗一次！允许看门狗狗叫善后钩子函数调用此类 API。注意：连续狗叫次数在用户任意的对看门狗的执行写操作都会清 0。

16.3.4.3.8 Wdt_GetInfo：查询“软看门狗”信息

bool_t　Wdt_GetInfo(struct tagWdtRsc *wdt, u32 type, void *storage)
头文件：
wdt.h
参数：
wdt：“软看门狗”指针；
type：操作命令；
storage：操作命令的返回值，即“软看门狗”信息和状态。
返回值：
获取成功返回 true；失败则返回 false。
说明：
获取“软看门狗”的状态和信息，是看门狗的读操作，可以用在任何地方。操作命令 type 如下表。

命令	描述
EN_WDT_INFOGETCMD_YIPREASON	获取“软看门狗”最后一次狗叫原因
EN_WDT_INFOGETCMD_YIPCYCLE	获取“软看门狗”狗叫周期

EN_WDT_INFOGETCMD_YIPACTION	获取“软看门狗”狗叫动作
EN_WDT_INFOGETCMD_EVENTID	获取“软看门狗”所属事件 ID
EN_WDT_INFOGETCMD_SHETIMES	获取调度原因导致连续狗叫次数
EN_WDT_INFOGETCMD_LOGICTIMES	获取逻辑错误原因导致连续狗叫次数
EN_WDT_INFOGETCMD_SHELEVEL	获取调度原因连续狗叫忍耐限度
EN_WDT_INFOGETCMD_LOGICLEVEL	获取逻辑错误原因连续狗叫忍耐限度
EN_WDT_INFOGETCMD_JUDGETIME	获取狗叫原因判断标准

第17章　文件系统

17.1 DJYFS 与 ANSI C 标准的差异

与 ANSI C89 一样， DJYFS（都江堰文件系统）支持流式文件访问，但并不支持 ANSI C89 标准要求的与文本文件相关的操作。除此之外，DJYFS 与 ANSI C89 完全兼容。与 UNIX 文件系统一样，DJYFS 只支持二进制数据存储，也就是说，DJYFS 把所有读写操作都看做二进制码流，不对其内容做任何假设和解析。我们知道，Windows 和 DOS 都支持文本流的操作，并且得到 ANSI C 标准的支持，如果用户用“r”、“w”、“a”、“r+”、“w+”或“a+”模式打开文件，文件系统将把用户对文件的读写数据流自动按文本文件格式解析，可以用文本文件的方式打开、读取和保存文件，必要时会自动插入和删除回车和换行符等文本文件特有的操作。注意，ANSI C 标准也只支持普通文本，不支持 unicode 文本。如果用“rb”、“wb”、“ab”、“rb+”、“wb+”或“ab+”这些模式打开文件，文件系统就不对读写数据做任何解析。在 DJYFS 中，虽然不支持文本文件操作，但是仍然可以识别 ANSI C 格式的所有模式字符串，“r”与“rb”等价，“w”与“wb”等价，……。

fp = djyfs_fopen(“myfile”, “r”);与 fp = djyfs_fopen(“myfile”, “rb”);将执行完全一致的操作。

上述差异可能会给一些习惯在 windows 下编程的用户带来些许不习惯，并且造成 DJYFS 与 ANSI C 不完全兼容。为什么要这样做呢？笔者认为，软件模块应该保持独立性和完整性，避免模块间在功能上互相交叉，这样会导致模块间的功能性耦合，又是“高内聚低耦合”。文件系统是一个有组织地存储数据的模块，它应该恪守数据存储的职责，而不应该过问所保存的数据的含义。应用程序从文件中读取数据后，将按其需要对数据进行解析，文本是其中一种组织方式。支持文本流实际上就要求文件系统要把存储在文件中的数据按照文本格式解析，而解析文件内容应该是文件系统使用者的事，这样做将造成文件系统模块和使用者程序模块间功能性的交叉。现实中，ANSI C 支持文本文件操作是有明显的副作用的：

1. 造成了理解上的歧义，从使用者的角度去思考，word 文档应该是典型的文本文件，ANSI 标准中却不能当做文本文件操作，unicode 文本文件也是这样的遭遇。反过来，以 ASCII 码格式保存的 protel 的电路原理图，典型的图形表达，却有千真万确是文本文件。以至于“什么是文本文件？”会成为一道无解计算机专业考试题。

2. 好像有点不公平，文本文件是一种文件格式，同样，还有许多文件类型如 pdf、doc、html 等同样也得到广泛应用，ANSI C 却只对文本文件提供编程语言级的支持，对其他文件格式却视如不见，奈何厚此薄彼？

3. 从软件维护和管理角度上，我们希望用相同的方法实现相同的功能，但是，为了支持文本文件却使这成为一种奢望。以 seek 函数为例，当用户执行 fseek(fp,100000, SEEK_CUR)函数，文本方式和非文本文件的执行方式是完全不同的。如果是二进制文件，则只要把文件

指针在存储器中的偏移量移动 100000 字节就可以了，而文本文件却不行，因为文本文件的回车和换行是按照一个字符解析的，因此，seek 文本文件就需要从当前文件指针开始连续读取超过 100000 字节数据，连续的回车字符和换行字符计为一个字符，才能确定文件指针在存储器中究竟要移动多少。同样是 fseek 操作，两种文件的实现方式完全不一样。
4. 本来，按文件方式存储数据有利于在不同的计算机系统之间交换和共享数据，但由于 ANSI C 文件系统对文本文件的内容进行解析，给这种数据共享造成了不必要的障碍，所有处理文本文件的程序，都要对 DOS 文件系统和 UNIX 文件系统区别处理。
因此，为了维护软件模块的独立性，不造成模块间耦合，DJYOS 甘冒不兼容 ANSI C 的风险，将只支持二进制流，不迎合 ANSI C 标准。

17.2 约定规范

使用 DJYOS 必须遵循以下几个规范：
1. 分区（相当于 win/dos 的磁盘）名的长度不得超过 255 字符，汉字按两个字符计算，具体的存储介质驱动模块所支持的分区名的长度可能小于 255 字符，比如 flash 存储介质驱动模块 DFFSD 的分区名就只支持 31 字符。
2. 文件名长度也不得超过 255 字符，与分区名的规定一样，比如 flash 存储介质驱动模块 DFFSD 的文件名长度只支持 215 字符。
3. 文件夹名、文件名和分区名均不能包含‘*’、‘\’、‘?’、‘/’、‘:’、‘|’、‘<’、‘>’和‘“’这 9 个字符，我们定义为“非法字符”。
4. 字符‘：’是分区名与目录之间的分隔符。一个完整的路径中字符‘：’之前的部分是分区名。
5. ‘\’是目录分隔符，包含多级目录的字符串（路径）被‘\’分割为一个个独立的分区名、目录名或文件名。
6. 不含分区名的路径（目录或文件名）中，如果以‘\’开头则认为从当前分区的根目录为该路径的根目录，如果不是以‘\’开头则认为当前路径 pg_work_path 为该路径的根目录；如果路径以‘\’作为结尾则认为该路径是一个目录的路径，否则认为是文件。
7. 默认目录嵌套最深不超过 20 级。这个可以通过修改 file.h 文件中的常量，但不能用配置进行修改。
8. 单个文件的长度允许达到 1G*blocksize 字节，一次读或者写的长度不得超过 232-1 字节。
9. 完整路径格式例如 D:\CCC\BBB\AAA，或者 D:\CCC\BBB\AAA\。其中“D”实际上即是文件系统的根目录也是文件所在的分区名称。
10. 绝对路径格式除了包含完整路径格式外，还包括冒号开头的路径：\XXX\XXX。
11. 相对路径格式例如 CCC\BBB\AAA。
都江堰文件系统限定了文件名长度和目录嵌套深度，作者并不愿意限制用户的自由，但作为与实时操作系统 DJYOS 配套的文件系统，却又不得不这样做。我们知道，文件和目录是按照名字操作的，访问文件首先要提供文件名和它的存放位置，文件系统通过目录路径逐级比较字符串定位文件的物理存储位置。如果文件名长度和目录嵌套深度不做限制，就有可能出现不可预估长度的“文件名+路径名”字符串，使得处理时间变得不确定，而不确定的执行时间是实时系统的大忌。另外，高可靠系统必须分析用户提供的名字串是否合法，其中就包含判断名字的长度是否合法，允许任意长的名字串长度使合法性判断无从下手。因此必须限定名字串的总长度，限定的方法有两种，第一种方法是每个文件或者目录的名字长度均可以达到 255 字节，但限制目录深度不能超过某一数值。第二种方法是限制包含路径名在内的整个名字串的总长度不得超过某一数值，单个文件或目录的名字长度以及允许的目录深度则不

计较。DJYFS 选择了前者，不选择后者的原因是：
1. 如果用户修改一个目录名，那么文件系统就必须扫描这个目录的上级目录和所有下级目录以及文件，其名字加起来不越界，这是一种执行时间开销极大而且不确定的操作，实时系统要尽量避免这种不确定执行时间的操作。
2. DJYOS 的信条之一，软件中的不同元素应该互相独立，尽量减少模块（元素）之间的耦合。文件系统中的一个个的文件和目录可能分属不同的模块，应该作为独立元素，应该有独立的名字空间，限制名字的总长度将导致不同的目录和文件的名字长度互相牵制，任何一个目录或者文件名的允许长度，都与其上下级目录（文件）名字的长度相关，使其命名空间不独立。
存储介质驱动模块所支持的文件名和分区名长度可能小于 DJYFS 模块，那么，因两者不一致而发生冲突的时候如何处理呢？DJYFS 规定如下：
1. 存储介质驱动模块支持的名字长度不得超过 DJYFS 模块，即分区名和文件名（目录名）长度均不得超过 255 字符。
2. 如果名字的实际长度不超过存储介质驱动模块所支持的长度，直接使用原字符串。
3. 如果名字长度超过存储介质驱动模块的名字长度限值 n，则只有前 n 个字符有效，在同一命名空间内，前 n 个字符如果相同，则构成名字冲突的条件。比如在 flash 芯片内创建两个分区，如果文件名的前 31 个字符相同而第 32 字符以后不同，则 DJYFS 模块认为是合法的字符串，而 flash 驱动模块则认为发生了字符串重名错误。

17.3 API 说明

17.3.1 Djyfs_SetWorkPath：设定“工作工作”目录

u32 Djyfs_SetWorkPath(char *path)
头文件：
os_inc.h
参数：
path：新的工作路径。
返回值：
设置成功返回 1；失败则返回 0。
说明：
设置工作路径（目录）。
如果工作路径是 NULL 或者空字符串（已经分配内存，但没有存储字符），则将文件系统第一个分区的根目录设为工作路径。

17.3.2 Djyfs_Fopen：打开文件（目录）

djyfs_file *Djyfs_Fopen(const char *fullname, char *mode)
头文件：
os_inc.h
参数：
fullname：路径名，目录名或者文件名不能包含“非法字符”。
mode：模式字符串，含义如下：

枚举变量	对应的模式字符串	功能
EN_R_RB	“r”，“rb”	打开一个文件用于读。
EN_W_WB	“w”，“wb”	打开或创建一个文件用于写，并删除文件已经存在的内容（如果有）。
EN_A_AB	“a”，“ab”	按追加数据方式打开一个文件用于写，若目标文件不存在则创建新文件并以追加数据方式打开。
EN_R_RB_PLUS	“r+”，“rb+”，“r+b”	打开一个文件用于读写，若目标文件不存在则失败。
EN_W_WB_PLUS	“w+”，“wb+”，“w+b”	打开或创建一个文件用于读写，并删除文件已经存在的内容（如果有）。
EN_A_AB_PLUS	“a+”，“ab+”，“a+b”，“cd”	按追加数据且可读的方式打开一个文件，若目标文件不存在则创建新文件并以追加数据且可读的方式打开。其中“cd”附带关闭即删除文件的属性。

返回值：
文件描述符指针。
说明：
依据操作模式 mode，打开或者创建并打开文件（目录）。如目标文件（目录）存在则打开，若文件（目录）不存在，且 mode 允许创建新文件（目录），则创建新文件（目录）。假设路径名 fullname 中包含的多级目录是不存在的，系统也会依据操作模式依次创建（如果模式是允许创建的话）。
如果路径名 fullname 不是一个完整路径（不含分区名），则所有的操作都是在系统的当前路径（假设为 D:\XX\YY\）所在的分区下（D:）完成。进一步细分，如果该路径名是一个绝对路径（例如\AA\BB\CC），则该路径在系统中的完整路径为 D:\AA\BB\CC；如果该路径是一个相对路径,例如 AA\BB\CC，则该路径在系统中的完整路径为 D:\XX\YY\AA\BB\CC。
警告：本函数通过参数 fullname 来判断打开或创建对象是文件还是目录。例如 fullname = “D:\XX\YY\”则打开或创建的是目录 YY，但如果 fullname = “D:\XX\YY”则表示打开或者创建文件 YY。

17.3.3 stat 按名字查询文件信息

s32 stat(const char *fullname, struct tagFileInfo * buf)
头文件：
os_inc.h
参数：
fullname：绝对路径（含分区名）或者相对路径。比如绝对路径“sys:\\first folder\\file.txt"，相对路径"\\first folder\\fitst.txt"，单纯文件名"file.txt"，纯路径名"\\first folder\\"。
buf：非空则返回文件信息
返回值：
返回: -1 表示失败，0 表示成功
说明：
查找一个文件（目录）是否存在，存在则返回文件信息。

17.3.4 Djyfs_Fstat：按句柄查询文件信息

bool_t Djyfs_Fstat(struct tagFileRsc *fp,struct tagFileInfo *fp_info)
头文件：
os_inc.h
参数：
fp：待操作的文件句柄。
fp_info：传递文件信息。
返回：
查询成功返回 true；失败则返回 false。
说明：
查询文件信息。具体信息内容，请参见 tagFileInfo 结构体成员。

17.3.5 Djyfs_Remove：删除文件（目录）

u32 Djyfs_Remove(const char *fullname)
头文件：
os_inc.h
参数：
fullname：绝对路径（含分区名）或者相对路径。比如绝对路径“sys:\\first folder\\file.txt"，相对路径"\\first folder\\fitst.txt"，单纯文件名"file.txt"，纯路径名"\\first folder\\"。
返回值：
删除成功返回零。否则返回错误码，详情看文件系统出错代码表。
说明：
删除一个未打开文件（目录），如果是目录则必须是空目录。

17.3.6 Djyfs_Rename：文件（目录）重命名

u32 Djyfs_Rename(const char *old_fullname, const char *new_filename)
头文件：
os_inc.h
参数：
fullname：绝对路径（含分区名）或者相对路径。比如绝对路径“sys:\\first folder\\file.txt"，相对路径"\\first folder\\fitst.txt"，单纯文件名"file.txt"，纯路径名"\\first folder\\"。
new_filename：把 old_fullname 中以字符'\'分割的最后一个文件（目录）的名字改为 new_filename。
返回值：
修改成功返回 true；失败返回 false。
说明：
修改文件（目录）的名字，只能修改一个对象，如果 old_fullname 包含多级路径，只能改最后一个，也不能改已经打开的对象。

17.3.7 Djyfs_Fclose：关闭文件（目录）

u32　Djyfs_Fclose(djyfs_file *fp)
头文件：
os_inc.h
参数：
fp：被关闭的文件（目录）。
返回值：
关闭成功返回 0；失败则返回 CN_LIMIT_UINT32。
说明：
关闭一个文件（或目录），如果是目录，则该目录不能有未关闭下级目录或文件。当文件使用完毕后，应当及时关闭，一方面可以回收有限的文件句柄资源，防止发生资源泄漏；另一方面可以使数据及时写入到存储介质中，DJYFS 是带缓冲的流式文件，写入数据时，文件系统将尽量利用缓冲区暂存，在必要的时候（缓冲区满、调用 flush 或关闭文件）才写入到存储介质中，如果文件操作已经完成，就应该及时关闭以使数据写入到存储设备，以防断电丢失。

17.3.8 Djyfs_Fread：读文件

size_t　Djyfs_Fread(const void *buf,size_t size, size_t nmemb, djyfs_file *fp)
头文件：
os_inc.h
参数：
buf：被读出数据的缓冲区。
size：待读出的单元数。
nmemb：每个单元的字节数。
fp：被读的文件。
返回值：
读出的数据数量（单元数）。
说明：
从文件中读出数据，功能与 ANSI C 相同。

17.3.9 Djyfs_Fread_r：可重入读文件

```
size_t　Djyfs_Fread_r(djyfs_file *fp,
                      const void *buf,
                      size_t size,
                      size_t offset)
```

头文件：
os_inc.h
参数：
fp：文件（句柄）指针。

buf：读出数据的缓冲区。
size：待读出的字节数。
offset：文件偏移量。
返回值：
读出的数据数量（字节）。
说明：
多线程安全读文件，功能与 Djyfs_Fread 类似，参数含义不一致。C 标准的 Djyfs_Fread 函数默认从当前文件指针处读，如果该指针被别的线程改变，读出的数据将是错误位置的，故不是多线程安全函数。实际上，本函数就是一个在信号量保护下的 fseek 和 fread 组合。

17.3.10 Djyfs_Fwrite：写文件

size_t Djyfs_Fwrite(const void *buf, size_t size, size_t nmemb, djyfs_file *fp)
头文件：
os_inc.h
参数：
buf：写入数据的缓冲区。
size：欲写入的单元数。
nmemb：每个单元的字节数。
fp：被写的文件。
返回值：
实际写入的数据数量（单元数）。
说明：
写数据到文件中，功能与 ansi c 相同。

17.3.11 Djyfs_Fwrite_r：可重入写文件

size_t Djyfs_Fwrite_r(djyfs_file *fp,
const void *buf,
size_t size,
size_t offset)
头文件：
os_inc.h
参数：
fp：文件指针。
buf：保存读出的数据的缓冲区。
size：待读出的字节数。
offset：文件偏移量。
返回值：
读出的数据数量（字节）。
说明：
多线程安全读文件，功能与 Djyfs_Fwrite 类似，参数含义不一致。C 标准的 Djyfs_Fwrite_r 函数默认在当前文件指针处写，如果该指针被别的线程改变，将从错误位置读出数据，故不

是多线程安全函数。实际上，本函数就是一个在信号量保护下的 fseek 和 fwrite 组合。

17.3.12 Djyfs_Fflush：刷新文件

u32 Djyfs_Fflush(djyfs_file *fp)
头文件：
os_inc.h
参数：
fp：文件句柄。
返回值：
刷新操作写入到存储介质的数据量。
说明：
DJYFS 中，每打开一个文件，系统会默认为其分配一个读写缓存区。此函数是将文件缓存区的内容写入到其所在的物理介质，例如 Flash 上。

17.3.13 Djyfs_Fseek：定位文件指针

u32 Djyfs_Fseek(djyfs_file *fp, sint64_t offset, int whence)
头文件：
os_inc.h
参数：
fp：文件句柄。
offset：移动长度。
whence：移动的参考位置。

whence 值	描述
SEEK_SET	从 0 开始计算
SEEK_CUR	从当前位置开始计算
SEEK_END	从文件结束位置计算

返回值：
移动成功返回 1；失败则返回 0。
说明：
把文件指针从 whence 指定的位置开始移动 offset 长度。

17.3.14 Djyfs_Format：格式化分区

bool_t Djyfs_Format(u32 format_para1, u32 format_para2, char *PTT_name)
头文件：
file.h
参数：
format_para1：格式化参数。
format_para2：格式化参数。

PTT _name：被格式化的分区名。
返回值：
格式化成功返回 true；失败则返回 false。
说明：
格式化分区。为分区上的文件系统分配目录存储空间、建立根目录和初始化文件系统的分区管理数据结构等。

17.3.15 Djyfs_GetRootFolder：获取根目录

struct tagFileRsc *Djyfs_GetRootFolder(struct tagFileRsc *fp)
头文件：
os_inc.h
参数：
fp：文件句柄。
返回值：
根目录指针。
说明：
查询文件的根目录。

17.3.16 Djyfs_PTT_Stat：查询分区信息

bool_t Djyfs_PTT_Stat(const char *fullname, s64 *sum, s64 *valid, s64 *free)
头文件：
os_inc.h
参数：
fullname：路径名。如 fullname 中包含了分区名，则取该分区的信息；如果没包含，则取当前路径所在分区信息，若当前路径又没有设置，返回 false。若只想获取当前路径所在分区信息，可给 NULL
sum、valid 和 free：传递查询结果的参数。分别为总空间尺寸、有效空间尺寸和空闲（剩余）空间尺寸。
返回值：
查询成功返回 true；失败则失败 false。
说明：
总空间指这个分区在 flash 芯片中占用的空间尺寸，有效空间指可以用来保存文件的部分，也就是“总空间-坏块-系统占用部分”，空闲空间是只当前剩余空间。接收数据的参数，如果是 NULL，则该参数不返回。

17.3.17 Djyfs_PTT_StatFp：查询分区信息

bool_t Djyfs_PTT_StatFp(struct tagFileRsc * fp, s64 *sum, s64 *valid, s64 *free)
头文件：
os_inc.h

参数：
fp：文件句柄。
sum、valid 和 free：传递查询结果的参数。分别为总空间尺寸、有效空间尺寸和空闲（剩余）空间尺寸。
返回值：
查询成功返回 true；失败则失败 false。
说明：
参见 Djyfs_PTT_Stat 函数。

17.3.18 Djyfs_GetPathLen：获取“目录分支”字长

u32 Djyfs_GetPathLen(struct tagFileRsc *fp)
头文件：
os_inc.h
参数：
fp：文件句柄。
返回：
文件的“目录分支”的长度，以字节为单位。'
说明：
获取文件所在“目录分支”的字符串长度。这个“目录分支”是指从文件的根目录直至该文件的完整路径。例如：A 文件在分区中的绝对路径为“E:\CCC\DDD\A”，那么这个绝对路径就是所谓的“目录分支”，“目录分支”字符串长度（含’\0’）即是函数的返回值。

17.3.19 Djyfs_GetCwdLen：获取“工作路径”字长

u32 Djyfs_GetCwdLen(void)
头文件：
os_inc.h
参数：
无。
返回值：
系统“当前工作路径”的字符串长度。
说明：
查询系统“当前工作路径”的字符串长度。查询的方式与 Djyfs_GetPathLen 一致，就是将当前工作路径（目录）作为该函数的输入参数。

17.3.20 Djyfs_GetPath：获取“绝对路径”

bool_t Djyfs_GetPath(struct tagFileRsc *fp, char *buf, u32 maxlen)
头文件：
os_inc.h
参数：

fp：文件句柄。
buf：传递文件的“目录分支”（返回值）。
maxlen：“目录分支”字符串长度上限。
返回值：
获取成功返回 true；失败则返回 false。
说明：
取 fp 的绝对路径到 buf 中，buf 的最大尺寸为 maxlen。buf 为 NULL 则动态分配 buf 所需内存，调用方须在使用完毕后释放内存。djyos 强烈不推荐这种方式，仅为了与传统函数保持兼容才保留这种方式。良好的编程习惯，应该是“malloc-free”这对函数在同一个函数内调用，malloc 在一个函数中调用而 free 在另一个函数调用，是容易滋生 bugs 的编码习惯，因此，应该避免使用 buf=NULL 的方式调用本函数。djyfs 提供替代方案是：先调用 Djyfs_GetPathLen 获取目录长度，用户自己根据长度分配内存，再调用 Djyfs_GetPath，就可以让"malloc--free"这对函数出现在同一个函数内。

17.3.21 Djyfs_GetCwd：获取“工作路径”

bool_t Djyfs_GetCwd(char *buf, u32 maxlen)
头文件：
os_inc.h
参数：
buf：传递系统“当前工作路径”（返回值）。
maxlen：“当前工作路径”字符串长度上限。
返回值：
获取成功返回 true；失败则返回 false。
说明：
取当前路径到 buf 中，功能与 Djyfs_GetPath 非常相似，参考之。

17.3.22 Djyfs_GetPTT_Name：获取分区名

bool_t djyfs_get_PTT_name(char *PTT_name, u32 namebuf_len)
头文件：
os_inc.h
参数：
PTT_name：传递分区名（返回值）。
namebuf_len：分区名称长度上限。
返回值：
获取成功返回 true；失败则返回 flase。
说明：
获取“当前工作路径（目录）”所在分区的名称。注意，以下几种情况会造成获取失败
1. 如果分区名称大于 namebuf_len；
2. 返回参数 PTT_name 指针为空；
3. 当前工作路径（目录）未设置。

17.3.23 Djyfs_FolderTraversalSon：遍历文件夹

djyfs_file *Djyfs_FolderTraversalSon(struct tagFileRsc *parent, struct tagFileRsc *current)
头文件：
os_inc.h
参数：
parent：被遍历的目录。
current：当前遍历点。
返回：
查询成功返回 true；失败则返回 false。
说明：
遍历一个文件夹（目录）的下一级（仅一层）的所有项，即该目录下所有的文件和目录，但不包括目录的目录下的文件。首次调用时，parent 设置为希望被遍历的目录句柄，current 设为 NULL，函数将返回目录 parent 的第一个子项（目录或者文件）。再次调用，把上一次返回的子项赋值给 current，将返回目录的下一个子项，直到返回 NULL，即完成了遍历目录的下一层级。如下代码将完成遍历 fp 目录

```
struct file_rsc *current=NULL,*parent=fp;
while(1)
{
    current =(struct file_rsc *)Djyfs_FolderTraversalSon(parent, current);
    if(current == NULL)
        break;
}
```

需要特别注意的是，current 作为函数间传递文件句柄的参数，其所占用的内存，是在首次调用 Djyfs_FolderTraversalSon 函数时动态分配的，并在最后一次调用（返回 NULL）时释放，如果没执行至函数返回 NULL，中途就退出循环的话，将存在发生内存泄漏的风险，切记切记！

17.3.24 Djyfs_ModuleInit：初始化文件系统

ptu32_t Djyfs_ModuleInit(ptu32_t para)
头文件：
file.h
参数：
未使用。
返回值：
初始化成功返回 1，失败则返回 0.
说明：
文件系统初始化。建立文件系统根节点（挂载点）、分配目录内存等。用户需注意，文件系统初始化后（即调用本函数之后），需要将具体格式的文件系统加载到该挂载点，系统才具备文件系统功能。

第18章　文件系统实现

DJYOS 文件系统大体由文件系统公共部分、具体文件系统实现部分和存储介质驱动部分组成，如第 17 章文件系统讲述的是文件系统的公共部分，它提供文件系统应用层标准接口，而这章重点阐述一个具体文件系统的实现。

18.1 flash 文件系统

DFFS 文件系统是一个功能相对完整的采用 Flash 作为存储器实现的文件系统。根据 Flash 文件系统设计要求，DFFS 文件系统已经基本具备了物理地址到逻辑地址的逻辑管理、坏块管理、均衡擦写次数、FDT 表和可靠性等基本功能来克服许多 Flash 设备的问题。下面，对该文件系统的实现原理作一些简介。

18.1.1 实现原理

18.1.1.1　几个重要的数据结构

DFFS 文件系统实现主要涉及两个数据结构，一个面向硬件的 tagFlashChip，另一个面向更上层文件系统接口的 tagPTTDevice。其中 tagPTTDevice 大部分成员用于实现了文件系统基本接口的注册，理解难度不大，但成员 tagStPTT_Flash 较为复杂，其用于 DFFS 文件系统的存储管理和实现，下面对这个结构体中的成员作些讲解。

18.1.1.2　MAT

MAT 表反映了分区所占用（物理）存储块的在软件概念上的组织结构，极为重要。它就是物理存储空间的一张属性表，通过它，我们可以方便地检索存储空间内所有块之间的关系。MAT 表由 item 组成，每个 item 都代表分区上的一个个（物理）块。item 的数据结构如下

```
struct tagMAT_Table
{
    uint32_t previous;
    uint32_t next;
    uint8_t attr;
};
```

在软件概念上，当分区上几个（物理）存储块具有同一属性（用途）时，例如几个块都是用作 FAT，那么它们所对应的 item 除了属性上具有相同的标志外，还会组成一个双向链表，如下图。这样，尽管这几个存储块在物理上不是连续的，但在软件概念上，将通过这个双向链表组成一个连续的空间。

图 18-1 MAT 示意图

18.1.1.3 ART

ART 是 Abrasion Record Table 的简称，用于实现 Flash 设备的磨损平衡策略。所谓磨损平衡是指通过一定的统计和调整，使 Flash 设备中的所有块的擦除次数处于大致同一个水平。ART 主要由一个索引表和一个磨损平衡表组成。DFFSD 建立分区时会对分区所占用的物理存储块进行分类，建立出磨损平衡表（ART_block_no、ART_times 和 balance_region），并将它们分类成忙区、闲区和死区。同时为实际物理存储块建立一张索引表 ART_position，用于物理块属性的快速查找，如下图所示。

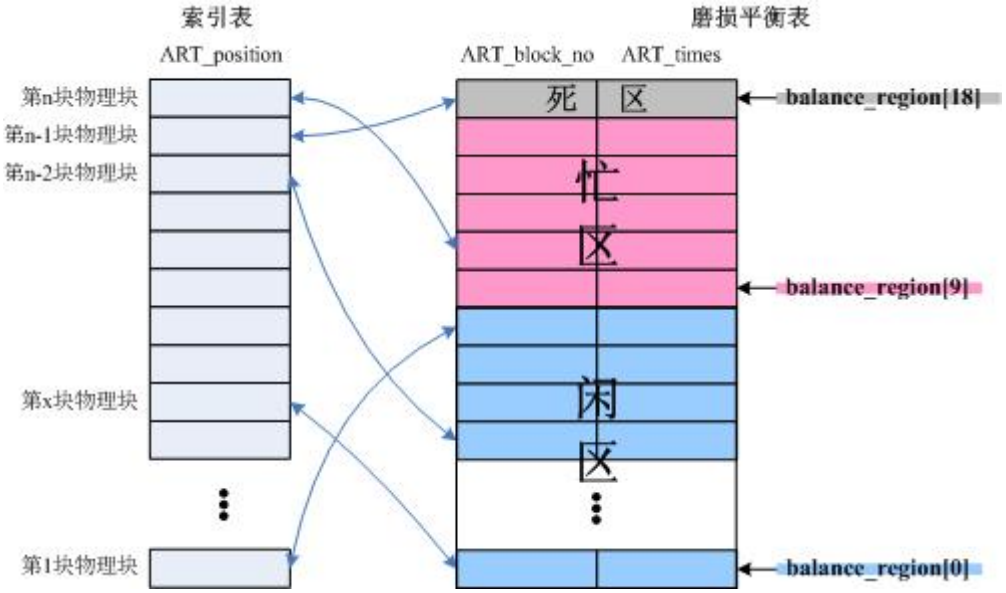


图 18-2 ART 概念示意图

磨损平衡策略的实现过程中涉及到几个重要的操作函数（技术方法）。下面作一些简要介绍。

18.1.1.3.1 坏块管理

在文件系统使用过程中，难免会出现坏块的情况。基于 ART 策略，一旦出现坏块，需要将坏块移动至磨损平衡表的死区。这个功能的实现方法如下图所示，具体参见函数__DFFSD_RegisterInvalid。

版权所有，转载必须注明出处，注明：来自 www.djyos.com 字样。

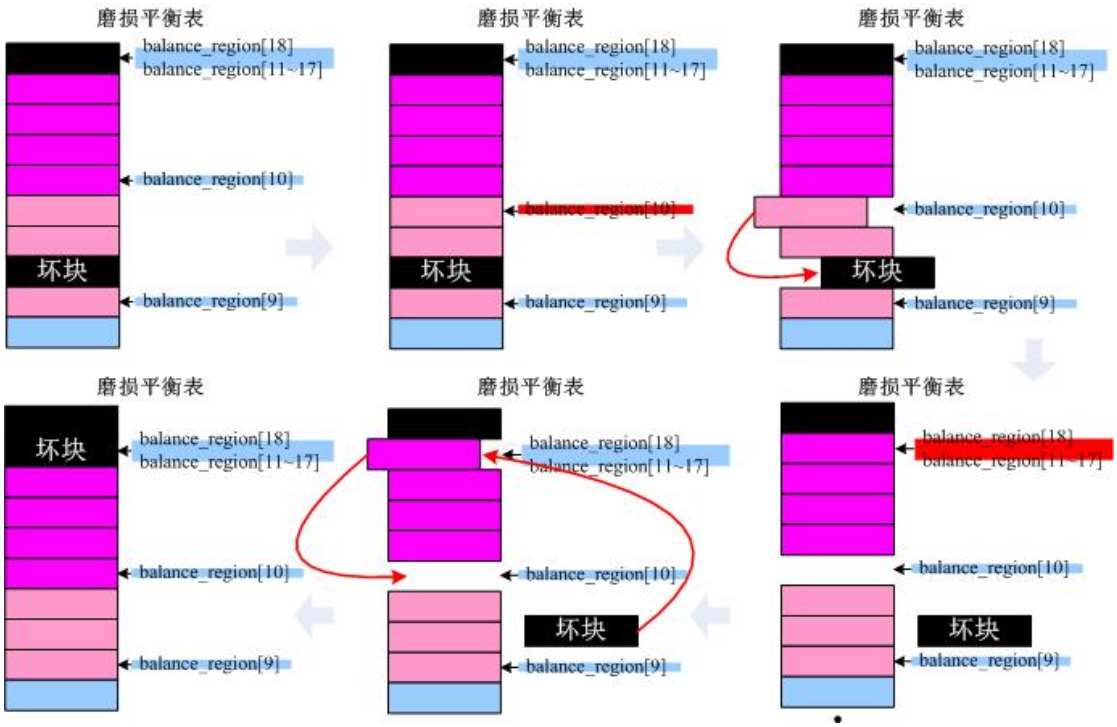


图 18-3 坏块管理示意图

18.1.1.3.2 储存块换区

当需要将一个存储块从一个区域跳转到另一个区域时，DFFSD 采用如下图的方法。具体实现函数参见__DFFSD_ART_Flip。

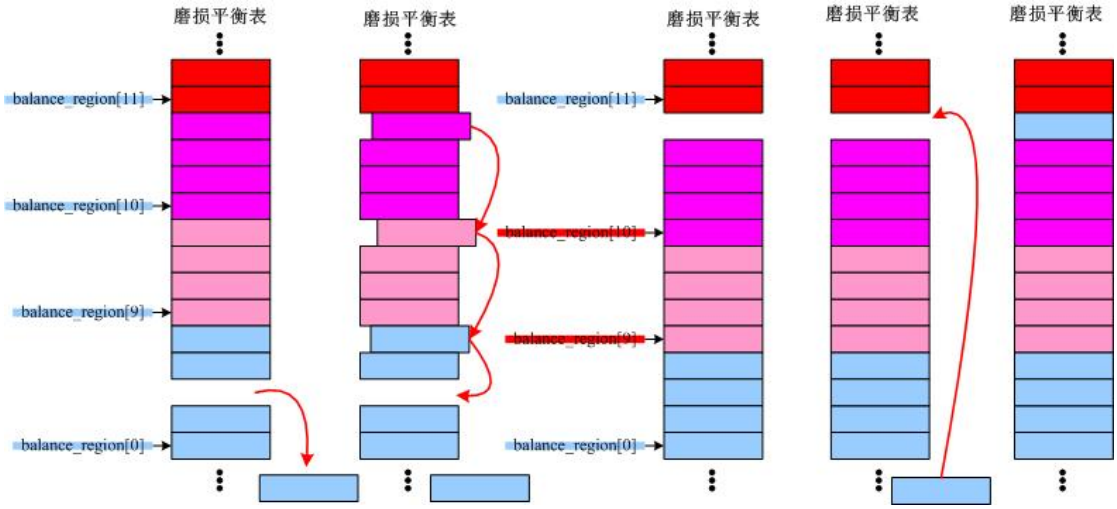


图 18-4 存储块换区示意图

18.1.1.4 FDT

FDT 是 File Description Table 的简称。由一个个结构体 tagFdtInfo 构成，其描述了文件的类型、名称和创建时间等信息。除此以外，结构体中的 parent、fstart_dson、previous 和 next

构成了如图 18-5 所示的 FDT，即 DFFSD 文件和目录的管理体系。

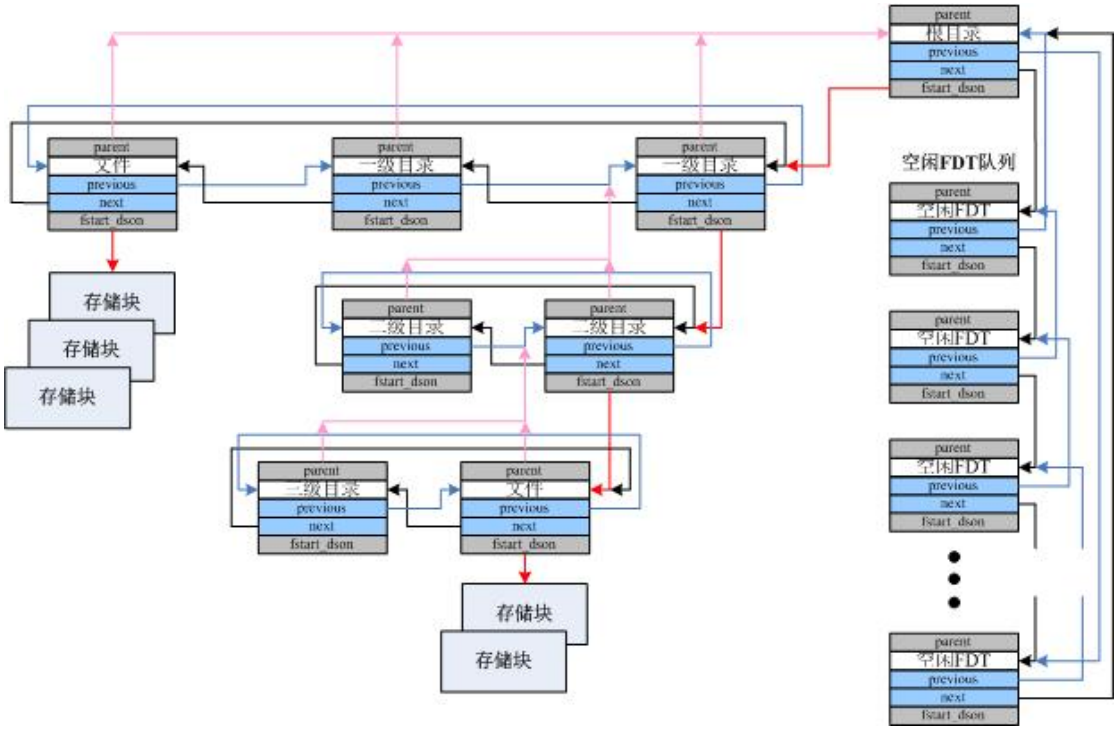


图 18-5 FDT 结构示意图

18.1.1.5 底层驱动

如果说文件系统是建房子的话，那么文件系统的驱动就好比是打地基。需要建造什么样的房子，就相应的需要按照房子的设计图纸建地基。

要写好 DFFS 的底层驱动程序，理解结构体 tagFlashChip 是精髓，该结构体中规定了建议 DFFS 操作 Flash 的方式(接口)和文件系统总体基本配置（MDR）。针对 DFFS 基本配置，作一些简单说明：

1. DFFS 是将文件系统建立在 Flash 芯片某段连续的存储空间上，该段空间以块为单位。
2. DFFS 内的单个文件至少占用一个存储块，因此驱动程序的接口函数必须包含块号参数。

底层驱动实现过程大体可分为以下三个步骤：

1. 实现 flash 驱动函数，即具体操作芯片的 API 函数；
2. 定义并初始化 tagFlashChip 结构类型的全局变量，变量的函数指针赋值为 flash 芯片操作函数；
3. 调用 DFFSD_InstallChip 函数，参数是初始化后的全局变量的地址，函数返回为 0，则装载成功。

18.1.2 DFFSD 建立流程

DFFSD 文件系统的建立过程大致分为如下几个步骤。

版权所有，转载必须注明出处，注明：来自 www.djyos.com 字样。

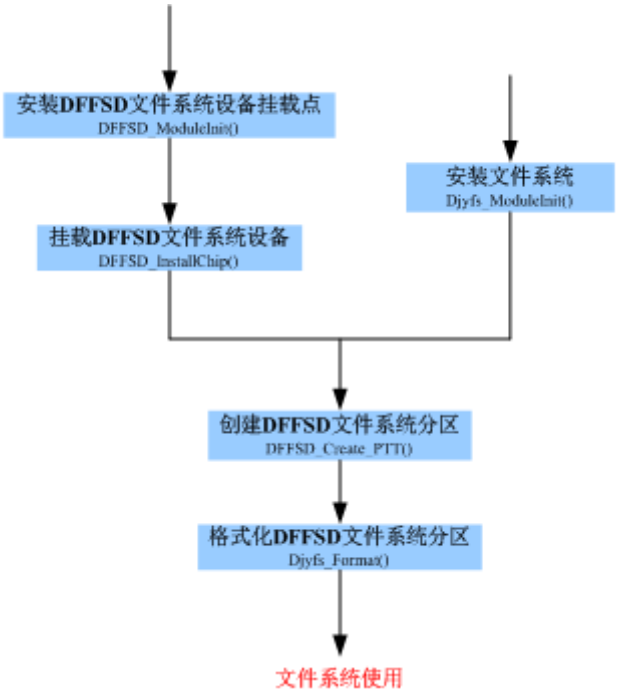


图 18-6 建立流程

系统在挂载 DFFSD 文件系统设备时，会首先将该存储设备格式化为 DFFSD 设备并注册设备驱动。再经过后续的分区创建和格式化操作，就在设备上建立了一个基本的 DFFSD 文件系统的目录结构和存储管理体系。

18.1.3 相关接口

18.1.3.1 DFFSD_ModuleInit：初始化 flash 文件系统

ptu32_t DFFSD_ModuleInit(ptu32_t para)
头文件：
flashfile.h
参数：
para：未使用。
返回值：
创建成功返回 true；失败则返回 false。
说明：
初始化 flash 文件系统，之后就可以挂在 flash 芯片了。

18.1.3.2 DFFSD_InstallChip：挂载 Flash 芯片

bool_t DFFSD_InstallChip(struct tagFlashChip *chip, char *name, uint32_t rev_blocks)
头文件：
flashfile.h

参数：
chip：flash 设备句柄（控制块）。
name：flash 设备名。
rev_blocks：flash 设备块的保留数量（该区域解释见书《都江堰操作系统与嵌入式系统设计》）。
返回值：
安装成功返回 true；失败则返回 false。
说明：
往 Flash 设备挂载点上装载一个 Flash 芯片。
该设备上如果不存在 MDR 表（作用于类似于硬盘的 MBR 表，位于设备的首块内存区），系统将自动为设备建立一个 MBR 表。如果 MDR 表已经存在，则系统会对该表进行解析，将设备的文件系统信息挂载到 DJYFS 之中。

18.1.3.3 DFFSD_FormatPTT：格式化分区

bool_t DFFSD_FormatPTT(u32 fmt_para1,
u32 fmt_para2,
struct tagPTTDevice *PTT_device_tag)
头文件：
flashfile.h
参数：
fmt_para1：ECC 功能开关；
fmt_para2：未使用；
PTT_device_tag：DFFS 分区句柄（控制块）。
返回值：
格式成功返回 true；失败则返回 false。
说明：
格式化一个 DFFS 文件系统的分区。

18.1.3.4 DFFSD_Create_PTT：创建分区

bool_t DFFSD_Create_PTT(struct tagFlashChip *chip, uint32_t size, char *name)
头文件：
mdr.h
参数：
chip：flash 设备句柄（控制块）。
size：分区尺寸（以块计）。
name：分区名。
返回值：
创建成功返回 true；失败则返回 false。
说明：
创建一个分区，文件系统格式为 DFFS。

18.1.3.5 DFFSD_RemoveFile：删除文件（目录）

bool_t DFFSD_RemoveFile(struct tagFileRsc *fp)
头文件：
flashfile.h
参数
fp：文件或者目录句柄。
返回值：
移除成功返回 ture；失败则返回 false。
说明：
移除一个文件或者目录。注意，以下情况会造成移除失败：
1. 目录非空；
2. 文件或者目录不存在；
3. 文件正在使用。

18.1.3.6 DFFSD_WriteFile：写文件

uint32_t DFFSD_WriteFile(struct tagFileRsc *fp, uint8_t *buf, uint32_t len)
头文件：
flashfile.h
参数
fp：文件句柄。
buf：写入数据。
len：写入数据长度（字节数）。
返回值：
写入到文件的新增数据长度。
说明：
将 len 长度的数据写入到文件中。
如果文件缓冲区空间足够，则写入的数据将暂存在该缓冲区。否则数据将连同缓冲区内的已有数据一并写入到文件系统的存储设备中。

18.1.3.7 DFFSD_ReadFile：读文件

uint32_t DFFSD_ReadFile(struct tagFileRsc *fp, uint8_t *buf, uint32_t len)
头文件：
flashfile.h
参数
fp：文件句柄。
buf：读出数据。
len：读出数据长度（字节数）。
返回值：
从文件中实际读出的数据长度。

说明：
从文件中读出 len 长度的数据。
注意，对文件的读操作会造成一次文件缓存区的刷入动作（缓存区数据刷新到文件系统的存储设备）。

18.1.3.8 DFFSD_FlushFile：刷新文件

uint32_t DFFSD_FlushFile(struct tagFileRsc *fp)
头文件：
flashfile.h
参数
fp：文件句柄
返回值：
实际刷入存储设备中的数据数量（字节为单位）。
说明：
将文件写缓存中的数据刷新到文件系统的存储设备。

18.1.3.9 DFFSD_QueryFileStocks：查询文件未读数据量

sint64_t DFFSD_QueryFileStocks(struct tagFileRsc *fp)
头文件：
flashfile.h
参数
fp：文件句柄。
返回值：
文件未读数据大小（字节为单位）。
说明：
查询文件还有多少未读数据（在文件系统设备中）。

18.1.3.10 DFFSD_QueryFileCubage：查询文件剩余空间

sint64_t DFFSD_QueryFileCubage(struct tagFileRsc *fp)
头文件：
flashfile.h
参数
fp：文件句柄。
返回值：
文件可写空间（已字节为单位）。
说明：
查询文件（在文件系统设备中）剩余（可写）空间。

18.1.3.11 DFFSD_check_PTT：查询分区信息

void DFFSD_check_PTT(struct tagPTTDevice *PTT_device_tag,
sint64_t *sum_size,
sint64_t *valid_size,
sint64_t *free_size)
头文件：
flashfile.h
参数
PTT_device_tag：分区句柄。
sum_size：分区总空间大小。
valid_size：分区有效空间（除坏块）大小。
free_size：分区空闲空间大小。
返回值：
无。
说明：
查询分区的总空间、有效空间和空闲空间的大小，单位已字节计。

18.1.3.12 DFFSD_SetFileSize：设置文件大小

sint64_t DFFSD_SetFileSize(struct tagFileRsc *fp, s64 new_size)
头文件：
flashfile.h
参数
fp：文件句柄。
new_size：文件尺寸。
返回值：
文件的实际尺寸。
说明：
（重新）设置文件尺寸。

18.1.3.13 DFFSD_SeekFile：定位文件指针

uint32_t DFFSD_SeekFile(struct tagFileRsc *fp, struct tagSeekPara *position)
头文件：
flashfile.h
参数
fp：文件句柄。
Position：
返回值：
设置成功返回 0；失败则返回 1。
说明：

定位文件指针。

18.1.3.14 DFFSD_CreateItem：创建文件/目录对象

```
bool_t    DFFSD_CreateItem (char *name,
                            struct tagFileRsc *parent,
                            struct tagFileRsc *result,
                            union file_attrs attr,
                            enum _FILE_OPEN_MODE_ mode)
```

头文件：
flashfile.h
参数
name：文件或目录名。
parent：上级目录句柄。
result：文件或者目录句柄。
attr：文件或目录属性。
mode：文件或者目录模式。
返回值：
成功创建返回 true；失败则返回 false。
说明：
创建一个目录或者文件。如果是文件，将同时为文件分配一块存储块和相应的缓存区。

18.1.3.15 DFFSD_OpenItem：打开（创建）文件对象

```
uint32_t    DFFSD_OpenItem(char *name,
                           struct tagFileRsc *parent,
                           struct tagFileRsc *result,
                           enum _FILE_OPEN_MODE_ mode)
```

头文件：
flashfile.h
参数
name：文件或者目录名。
parent：上级目录。
result：文件或者目录句柄。
mode：文件模式。
返回值：
打开成功返回 CN_FS_OPEN_SUCCESS；失败则返回错误码：所查找文件或目录（包括上级目录）不存在（CN_FS_ITEM_INEXIST）、输入参数错误（CN_LIMIT_UINT32）、文件缓存创建失败（CN_FS_ITEM_EXIST）。
说明：
查找 parent 目录下名称为 name 的文件或者目录。如果查找到的是文件，文件系统会同时为该文件建立相应的缓存区。

18.1.3.16 DFFSD_CloseItem：关闭文件对象

bool_t DFFSD_CloseItem(struct tagFileRsc *fp)
头文件：
flashfile.h
参数
fp：文件或者目录句柄。
返回值：
关闭成功返回 true；失败则返回 false。
说明：
关闭一个文件系统对象，文件或者目录。

18.1.3.17 DFFSD_LookForItem：查找文件对象

bool_t DFFSD_LookForItem(char *name,
                          struct tagFileRsc *parent,
                          struct tagFileRsc *result)
头文件：
flashfile.h
参数
name：文件或者目录名。
parent：上级目录。
result：文件或者目录句柄。
返回值：
查找成功返回 true；失败则返回 false。
说明：
查找 parent 目录下一级是否存在名称为 name 的目录或者文件。

18.1.3.18 DFFSD_RenameItem：重命名文件对象

bool_t DFFSD_RenameItem(struct tagFileRsc *fp, char *newname)
头文件：
flashfile.h
参数
fp：文件或者目录句柄。
newname：新名称。
返回值：
重新命名成功返回 true；失败则返回 false。
说明：
重新命名一个文件或者目录。

18.1.3.19 DFFSD_ItemTraversalSon：遍历文件对象

struct tagFileRsc　*DFFSD_ ItemTraversalSon (struct tagFileRsc *parent,
　　struct tagFileRsc *current)
头文件：
flashfile.h
参数
parent：上级目录句柄。
current：下级目录句柄。
返回值：
下级目录的一个文件或者目录。
说明：
遍历一个文件对象（目录）下的内容。此函数使用方式有些特殊，详细内容参见资源管理函数 Rsc_TraveScion。

18.1.3.20 DFFSD_CheckFDT_Size：查询 FDT 表容量

uint32_t　DFFSD_CheckFDT_Size(struct tagPTTDevice *PTT_device_tag)
头文件：
flashfile.h
参数
PTT_device_tag：分区句柄。
返回值：
目录（文件）表总容量，字节为单位。
说明：
获取一个分区当前目录（文件）表尺寸，即 FDT 的总容量，包括使用和未使用的。

18.1.3.21 DFFSD_ReadFDT：读取 FDT 表

void　DFFSD_ReadFDT(struct tagPTTDevice *PTT_device_tag, uint8_t *buf)
头文件：
flashfile.h
参数
PTT_device_tag：分区句柄。
buf：传递 FDT 数据（返回值）。
返回值：
无。
说明：
获取一个分区表当前目录（文件）表，即 FDT 的内容。

18.1.3.22 DFFSD_CheckFolder：统计目录下级对象总数

uint32_t DFFSD_CheckFolder(struct tagFileRsc *parent)
头文件：
flashfile.h
参数
parent：目录句柄。
返回值：
文件和目录数量。
说明：
统计 parent 目录下所包含的文件和目录数量。
本函数只统计 parent 目录的下一级，而不是所有。例如 parent 目录下有 son1、son2 这两个目录，而 son1 目录下又包含了 grandson1 文件。那么调用本函数的统计结果是 2，而不是 3。

18.2 简易 nor flash 文件系统

DEFFS 文件系统是相对于完整 Flash 文件系统而言的，与完整 Flash 文件系统相比，DEFFS 文件系统做了精简，对于使用者的影响如下：
1. fopen 函数在操作简易文件系统内的文件时，功能有所残缺，只能操作存在的文件，不能创建文件。
2. 创建文件后必须随即设置文件的尺寸上限。
3. 文件在系统中顺序存储，不能随意删除文件，只能删最后一个文件。
4. 无 ECC 校验，不能在 nand flash 上使用。
5. 写入文件的数据不能超过创建文件时指定的尺寸上限。使用 fwrite 写入文件时，超出的部分将丢失。
6. 不支持擦写平衡和坏块替换，频繁写入影响器件寿命。
7. 不支持创建目录，DEFFS 文件系统只存在根目录，所有的文件都放置在根目录下。。
8. 文件名的长度不大于 31 字节。
完整的 Flash 文件系统虽然功能强大，但是代码相当复杂，操作也慢，我们知道，相对于简单系统，保证一个复杂系统的高可靠性，要难得多。在高可靠性的嵌入式系统应用，对于这种高复杂性的组件，能不用就不用吧。但是如果不使用文件系统，则应用程序就要直接操作 Flash 存储芯片，这不利于代码归一化。
使用 DEFFS 文件系统，既能保证应用程序使用标准的文件系统接口，确保代码归一化，又不降低系统可靠性。

18.2.1 实现原理

18.2.1.1 存储空间管理

DEFFS 文件系统空间结构图如图 18-7 所示。一般情况下，文件分配表至少占用两个块（主表和备份表分别占用一个块）。表后的块用来存放文件内容，每个文件至少占用一个块大小。

因此 DEFFS 文件系统的文件数量受 Flash 芯片容量（块数量）的限制，适用于小型的嵌入式系统。

DEFFS 文件系统的物理存储管理主要分为两大部分。一部分是文件分配表，其同时又分为主区和备份区。另一部分则是各文件的存储区。

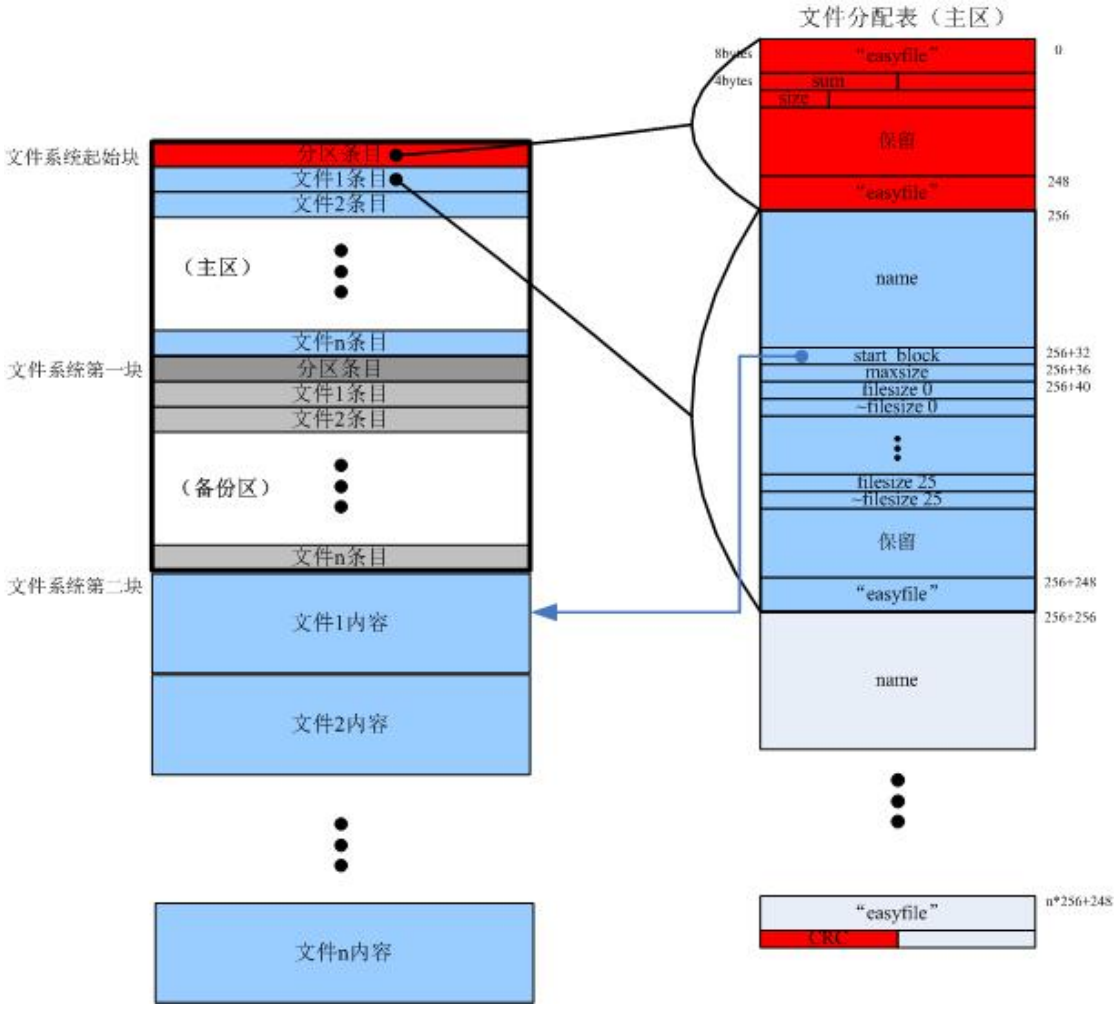


图 18-7 文件系统结构图

18.2.1.2 文件分配表

文件分配表用于记录文件系统分区和各文件的信息。分配表有主区和备份区两部分备份区的内容与主区的始终保持一致，用于文件系统被破坏等情况下的文件修复。。

DEFFS 文件系统的文件分配表具有以下特点：

1.分配表中主区和备份区的大小相等，缺省值为 16384 个字节，用户可通过修改头文件 Easynorflash.h 中的宏定义 CN_EASYNOR_CFG_LIMITDEFFS 来修改文件分配表大小。

2. 文件分配表由若干个条目构成，每个条目大小都为 256 个字节；

3. 文件分配表首个条目是分区条目，用于存放文件系统分区信息，包括标记符、分区块数、块大小和校验符等；

4. 除首个条目外，文件分配表中其他条目都是文件条目，保存了文件的相关属性，如文件名、文件起始块、文件最大长度、文件长度和校验符等；同时，在每个文件条目中有一块 filesize 记录区（26 条），循环依次更新文件的尺寸，防止每次更改文件大小需要擦除文件分配表。

5. 文件分配表在两个区（主区和备份区）的最后 2 字节用于记录 CRC 校验码。

18.2.1.3 检查分区

DEFFS 文件系统分区检查校验过程相对比较复杂，其流程图如下图所示。检查分区的流程大体可分为主分配表 CRC 校验错误、备份分配表 CRC 校验错误和主备分配表 CRC 校验都正确三种情况，从流程图可清晰的看到各种情况的处理过程，此处不再赘述，实现函数参照__Easynor_CheckNorChip。

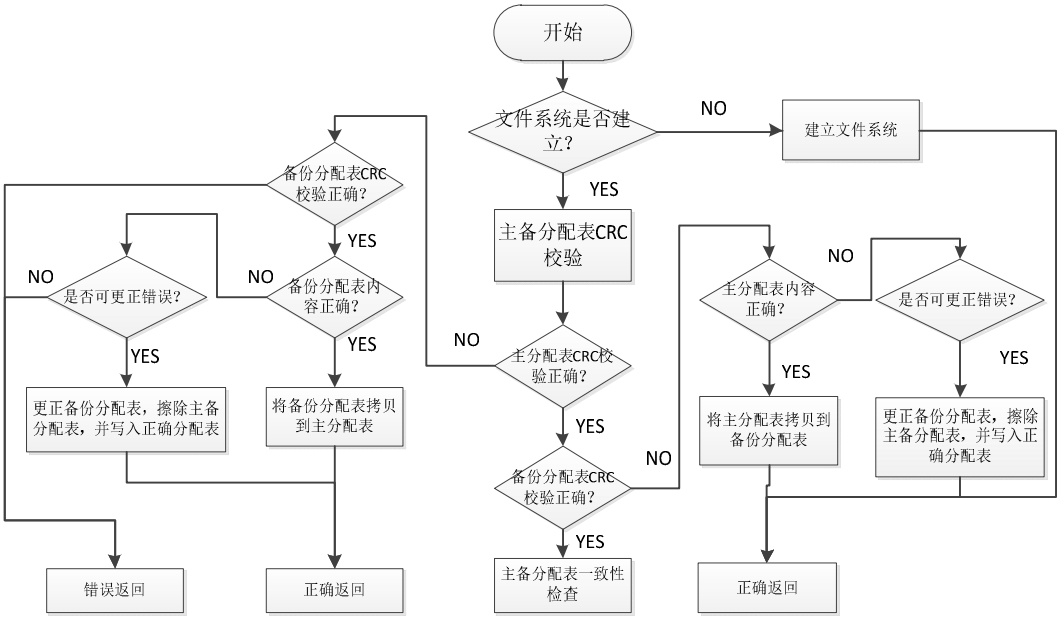


图 18-8 文件系统检查流程图

当主备分配表 CRC 校验都正确的情况下，还需对主备分配表一致性检查，若发现错误，则需对错误分配表进行纠正。主备分配表核对流程图如所示。

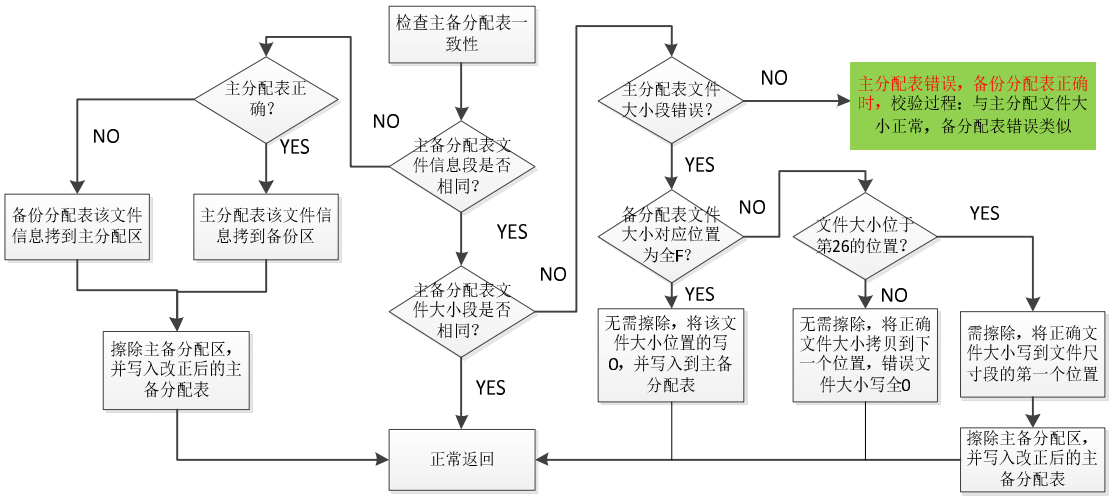


图 18-9 主备分配表校对图

版权所有，转载必须注明出处，注明：来自 www.djyos.com 字样。

18.2.1.4 驱动实现

DEFFS 文件系统的通过结构体 tagEasynorFlashChip 实现对驱动层的管理。结构体中规定了需用户实现和设置的驱动接口以及文件系统参数。结构体的具体内容请参加文件 easynorflash.h。
结合前文所述 DEFFS 对文件系统的驱动实现作一些补充：
1. DEFFS 文件系统是将文件系统建立在 Flash 芯片的某段连续的存储空间，该存储空间以块为单位；
2. 每个文件内容至少占用一个完整块，因此驱动程序的接口函数必须包含块参数；
3. 驱动函数的实现需按照结构体 tagEasynorFlashChip 中的成员（即 Flash 操作函数指针）功能定义以及根据实际 Flash 芯片特性，若无须实现则为 NULL。
安装 DEFFS 文件系统最终实际很简单，用户可参考目录 bsp\chips\s29gl128 下的源文件 flash_s29glxxx.c 的函数 module_init_fs_s29glxxx。
驱动实现逻辑大体可分为以下三个步骤：
1. 实现 flash 驱动函数，即操作芯片的 API 函数；
2. 定义并初始化 tagEasynorFlashChip 结构类型的全局变量，变量的函数指针赋值为 flash 芯片操作函数；
3. 调用 Easynor_InstallChip 函数，参数是初始化后的全局变量的地址，函数返回为 0，则装载成功。

18.2.2 操作接口

18.2.2.1 Easynor_ModuleInit：建立挂载点

ptu32_t Easynor _ModuleInit(ptu32_t para)
头文件：
easynorflash.h
参数：
para：未使用。
返回值：
创建成功返回 true；失败则返回 false。
说明：
建立 DFFSD 文件系统物理设备（Flash）挂载点。在概念上，只有建立 Flash 挂载点后，才能安装 Flash 设备。

18.2.2.2 EASYNOR_InstallChip：挂载 DEFFS 文件系统

u32 EASYNOR_InstallChip(struct tagDEFFSFlashChip *chip)
头文件：
Easynorflash.h
参数

chip：设备句柄。
返回值：
挂载成功返回 true；失败则返回 false。
说明：
挂载 DEFFS 文件系统。本函数实际包含了两个动作，一个是挂载文件的物理载体（设备），另一个是挂载 DEFFS 文件系统。

18.2.2.3 EASYNOR_Format_PTT：格式化 DEFFS 文件系统

```
bool_t   EASYNOR_Format_PTT(u32 fmt_para1,
                            u32 fmt_para2,
                            struct tagPTTDevice *PTT_device_tag)
```

头文件：
Easynorflash.h
参数
fmt_para1：保留（未使用）。
fmt_para2：保留（未使用）。
PTT_device_tag：分区句柄。
返回值：
格式化成功返回 true；失败则返回 false。
说明：
将一个分区格式化为 DEFFS 文件系统。

18.2.2.4 EASYNOR_WriteFile：写文件

```
u32   EASYNOR_WriteFile(struct tagFileRsc *fp, uint8_t *buf, u32 len)
```

头文件：
Easynorflash.h
参数
fp：文件句柄。
buf：写数据。
len：写数据长度，字节计。
返回值：
实际写入的数据大小，字节计。
说明：
将一定长度（len）的数据写入到 DEFFS 文件系统（物理设备）的文件之中。

18.2.2.5 EASYNOR_ReadFile：读文件

```
u32   EASYNOR_ReadFile(struct tagFileRsc *fp, uint8_t *buf, u32 len)
```

头文件：
Easynorflash.h
参数

fp：文件句柄。
buf：读数据。
len：读数据长度（字节）。
返回值：
实际读出的数据大小（字节）。
说明：
从 DEFFS 文件系统（物理设备）的文件中读出一定长度（len）的数据。

18.2.2.6 EASYNOR_FlushFile：冲刷文件

u32 EASYNOR_FlushFile(struct tagFileRsc *fp)
警告：省略，未实现。

18.2.2.7 EASYNOR_SeekFile：定位文件指针

u32 EASYNOR_SeekFile(struct tagFileRsc *fp, struct tagSeekPara *position)
头文件：
Easynorflash.h
参数
fp：文件句柄。
position：文件指针新位置。
返回值：
定位成功返回 1；失败则返回 0。
说明：
重新定位文件的读写指针。
警告：DEFFS 文件系统不允许本函数扩展文件尺寸。

18.2.2.8 EASYNOR_MoveFile：移动文件

bool_t EASYNOR_MoveFile(struct tagFileRsc *src_fp, struct tagFileRsc *dest_fp)
警告：省略，函数未实现。

18.2.2.9 EASYNOR_QueryFileStocks：查询文件未读数据量

sint64_t EASYNOR_QueryFileStocks(struct tagFileRsc *fp)
头文件：
Easynorflash.h
参数
fp：文件句柄。
返回值：
文件未读数据大小。
说明：

查询文件还有多少未读数据（在文件系统设备中）。

18.2.2.10 EASYNOR_QueryFileCubage：查询文件剩余空间

sint64_t　EASYNOR_QueryFileCubage(struct tagFileRsc *fp)
头文件：
Easynorflash.h
参数
fp：文件句柄。
返回值：
文件剩余（可写）空间。
说明：
查询文件（在文件系统设备中）还有多少剩余（可写）空间。

18.2.2.11 EASYNOR_CreateItem：创建文件对象

```
bool_t　EASYNOR_CreateItem(char *name,
                           struct tagFileRsc *parent,
                           struct tagFileRsc *result,
                           union file_attrs attr,
                           enum _FILE_OPEN_MODE_ mode)
```

头文件：
Easynorflash.h
参数
name：文件对象名。
parent：根目录。
result：传递新创建的文件句柄（返回值）。
attr：保留（未使用）。
mode：保留（未使用）。
返回值：
创建成功返回 true；失败则返回 false。
说明：
创建一个文件对象。
警告：DEFFS 文件系统的文件对象只能是文件。

18.2.2.12 EASYNOR_ItemTraversalSon：遍历所有文件

```
struct tagFileRsc　*EASYNOR_ItemTraversalSon(struct tagFileRsc *parent,
                                             struct tagFileRsc *current)
```

头文件：
Easynorflash.h
参数
parent：根目录句柄。

current：文件句柄。
返回值：
文件句柄。
说明：
遍历根目录下的文件。
警告：本函数在 DEFFS 文件系统下的实现比较特殊。首先，参数 parent 必须是 DEFFS 文件系统的根目录（分区名）。其次，遍历实现过程是通过函数的递归调用实现的。而且首次调用时参数 current 必须设置 NULL，直至返回值为 NULL。举例如下：

```
struct tagFileRsc    *file_handle = NULL;
do{
      file_handle = EASYNOR_ItemTraversalSon(root_handle,file_handle);
      //添加关于文件的操作
}while(file_handle != NULL);
```

18.2.2.13 EASYNOR_OpenItem：打开文件对象

```
u32    EASYNOR_OpenItem(char *name,
                        struct tagFileRsc *parent,
                        struct tagFileRsc *result,
                        enum _FILE_OPEN_MODE_ mode)
```

头文件：
Easynorflash.h
参数
name：文件名。
parent：根目录。
result：传递文件句柄（返回值）。
mode：保留（未使用）。
返回值：
CN_LIMIT_UINT32：输入参数错误。
CN_FS_ITEM_INEXIST：未发现文件。
CN_FS_OPEN_SUCCESS：成功打开文件。
说明：
打开 DEFFS 文件的一个文件对象。
警告：DEFFS 文件系统的文件对象只能是文件。

18.2.2.14 EASYNOR_LookforItem：检索同名文件对象

```
bool_t    EASYNOR_LookforItem(char *name,
                              struct tagFileRsc *parent,
                              struct tagFileRsc *result)
```

头文件：
Easynorflash.h
参数

name：文件对象名。
parent：根目录。
result：传递同名文件对象（返回值）。
返回值：
检索成功返回 true；失败则返回 false。
说明：
检索同名文件对象。
警告：DEFFS 文件系统的文件对象只能是文件。

18.2.2.15 EASYNOR_CloseItem:关闭文件对象

bool_t EASYNOR_CloseItem(struct tagFileRsc *fp)
警告：省略，函数未实现。

18.2.2.16 EASYNOR_RenameItem：文件对象重命名

bool_t　EASYNOR_RenameItem(struct tagFileRsc *fp, char *newname)
警告：省略，函数未实现。

18.2.2.17 EASYNOR_CheckPTT：检查分区

void　EASYNOR_CheckPTT(struct tagPTTDevice *PTT_device_tag,
　　sint64_t *sum_size,
　　sint64_t *valid_size,
　　sint64_t *free_size)
警告：省略，函数未实现。

18.2.2.18 EASYNOR_SetFileSize：设置文件尺寸

sint64_t　EASYNOR_SetFileSize(struct tagFileRsc *fp, s64 new_size)
头文件：
Easynorflash.h
参数
fp：文件句柄。
new_size：文件新尺寸或上限。
返回值：
设置成功返回 0；失败则返回-1。
说明：
设置文件的尺寸或者上限尺寸。对于一个文件而言，创建完成后第一次调用本函数即为设置文件的尺寸上限。而后续调用则是动态设置文件尺寸，并且动态设置的尺寸不能超出第一次设置的尺寸上限。
警告：在 DEFFS 文件系统中，文件创建后需要立即设置文件的尺寸上限。否则文件系统无

法获知文件大小，会造成其他新文件的创建失败。

18.2.2.19 EASYNOR_CheckFDT_Size：检查 FDT

u32　EASYNOR_CheckFDT_Size(struct tagPTTDevice *PTT_device_tag)
警告：省略，DEFFS 文件系统没有 FDT 表（目录管理），函数未实现。

18.2.2.20 EASYNOR_CheckFolder：检查目录

u32　EASYNOR_CheckFolder(struct tagFileRsc *parent)
警告：省略，函数未实现。

第19章　网络

DJYIP 协议栈提供基本兼容 BSD 的 socket 接口。

第20章　消息队列

消息队列是事件之间的一种通信机制，可以实现一个事件向另一个事件发送变量等数据。为了使用 DJYOS 的消息队列功能，在初始化系统时，必须调用 MsgQ_ModuleInit 来初始化消息队列模块。此后，在各事件处理函数（进程）中调用 MsgQ_Create 或者 MsgQ_Create_r 来创建所需的消息队列。
DJYOS 的消息队列有两种管理模式，在事件创建消息队列时设定。一种是 CN_MSGQ_TYPE_PRIO，即优先级管理模式。这个模式下，当许多事件从同一消息队列中读取消息时被阻塞时，这些事件被新消息唤醒的次序是根据事件的优先级来排列的。另一种是 CN_MSGQ_TYPE_FIFO，即先进先出管理模式。这种模式下，当多事件被同一消息队列阻塞后，它们被唤醒的次序是谁先申请读取消息，谁先被唤醒。

20.1 相关 API 说明

20.1.1 MsgQ_ModuleInit：初始化消息队列模块

ptu32_t　MsgQ_ModuleInit(ptu32_t para)
头文件：
msgqueue.h
参数：
para：未使用参数。
返回值：
零.
说明：

初始化消息队列模块。本函数用于系统的功能的裁剪，当系统具有消息队列功能时，在系统初始化是需引用本函数。

20.1.2 MsgQ_Create：创建消息队列

struct tagMsgQueue *MsgQ_Create(u32 MaxMsgs, u32 MsgLength, u32 Options)
头文件：
msgqueue.h
参数：
MaxMsgs：消息队列所能容纳消息的数量上限。
MsgLength：单个消息长度（字节数）。
Options：消息队列属性。CN_MSGQ_TYPE_PRIO 表示优先级类型的消息队列；CN_MSGQ_TYPE_FIFO 表示先进先出类型的消息队列。
返回值：
创建成功返回消息队列句柄（控制块）；失败则返回 NULL。
说明：
动态地创建一个消息队列。系统从堆中为消息队列分配空间。

20.1.3 MsgQ_Create_r：创建静态消息队列

```
struct tagMsgQueue  *MsgQ_Create_r(struct tagMsgQueue *pMsgQ,
                                   u32 MaxMsgs,
                                   u32 MsgLength,
                                   u32 Options,
                                   void *buf)
```

头文件：
msgqueue.h
参数：
pMsgQ：消息队列句柄（控制块）。
MaxMsgs：消息队列所包含消息数的上限。
MsgLength：单个消息长度（字节数）。
Options：消息队列属性。CN_MSGQ_TYPE_PRIO 表示优先级类型的消息队列；CN_MSGQ_TYPE_FIFO 表示先进先出类型的消息队列。
buf：消息空间。
返回值：
创建成功返回消息队列指针；失败则返回 NULL。
说明：
静态地创建一个消息队列。这里所谓的静态，是指消息队列所占用的内存空间由用户指定和保证。

20.1.4 MsgQ_Delete：删除消息队列

bool_t　MsgQ_Delete(struct tagMsgQueue *pMsgQ)
头文件：
msgqueue.h
参数：
pMsgQ：消息队列句柄（控制块）。
返回值：
删除成功返回 true；失败则返回 false。
说明：
动态地删除消息队列。其与 MsgQ_Create 对应。

20.1.5 MsgQ_Delete_r：删除静态消息队列

bool_t　MsgQ_Delete_r(struct tagMsgQueue *pMsgQ)
头文件：
msgqueue.h
参数：
pMsgQ：消息队列句柄（控制块）。
返回值：
删除成功返回 true；失败则返回 false。
说明：
静态地删除消息队列。其与 MsgQ_Create_r 对应。

20.1.6 MsgQ_Send：发送消息

```
bool_t　MsgQ_Send(struct tagMsgQueue *pMsgQ,
                  u8 *buffer,
                  u32 nBytes,
                  u32 timeout,
                  bool_t priority)
```

头文件：
msgqueue.h
参数：
pMsgQ：消息队列句柄（控制块）。
buffer：消息缓冲区指针。
nBytes：消息大小（字节数），不能超过创建消息队列时指定的 MsgLength。
timeout：等待消息队列时间。
priority：消息紧急属性。CN_MSGQ_PRIO_URGENT 表示紧急消息，消息将被放于消息队列头部；CN_MSGQ_PRIO_NORMAL 表示普通消息，消息则将消息队列尾部。
返回值：
发送成功返回 true；失败则返回 false。

说明：
通过消息队列发送一个消息，如果消息队列已满，则阻塞，直到有其他线程从消息队列中接收消息，或者超时时间到。

20.1.7 MsgQ_Receive：接受消息

bool_t MsgQ_Receive(struct tagMsgQueue *pMsgQ, u8 *buffer, u32 nBytes, u32 Timeout)
头文件：
msgqueue.h
参数：
pMsgQ：消息队列句柄（控制块）。
buffer：消息数据空间。
nBytes：消息大小（字节数），不能超过创建消息队列时指定的 MsgLength。
Timeout：等待消息的时间。
返回值：
读取成功返回 true；失败则返回 false。
说明：
从消息队列读取一个消息。
当消息队列中无消息，且设置了等待消息时间，则会引起事件阻塞。

20.1.8 MsgQ_NumMsgs：查询消息数

u32 MsgQ_NumMsgs(struct tagMsgQueue *pMsgQ)
头文件：
msgqueue.h
参数：
pMsgQ：消息队列句柄（控制块）。
返回值：
消息队列中的消息数。
说明：
查询消息队列中的消息数。

第21章 DjyBus 总线

嵌入式系统中，总线通信是很常见的通信方式。因此，在 DJYOS 中，将总线通信驱动模块标准化和模块化，使其成为操作系统的重要模块，能有效降低程序复杂度，使程序设计、调试和维护等操作简单化。DJYOS 的总线驱动模块是一个独立的组件，虽然许多硬件设备需要使用这个组件来与系统连接，但总线驱动模块本身和设备驱动架构并无必然联系。DJYOS 小心避免出现大而复杂的组件，努力使系统由小而完备、易于理解、互相没有耦合的组件组件构成。

21.1 DjyBus 架构模型

DjyBus 将总线以资源形式连接到系统资源链表，如图 21-1 所示，“DjyBus”是总线资源树，不同类型总线(如 IIC 总线、SPI 总线、PCI 总线、USB 总线)挂接到“DjyBus”树下面，成为“DjyBus”子树。每种类型总线又允许多条总线(有很多 cpu 可能会有不止一个 IIC 总线控制器，每条 IIC 总线控制在“IIC”总线类型资源结点下建立一颗资源树，资源树的名字建议为“IICn”，其中 n 是总线编号。)，所有符合这种协议器件都将挂到相应的总线上。如同是从同一个树干分出的树枝，每增加或删除一个器件，相应的树上便增加或减少一个资源结点，它们继承了树干所具有的特性。

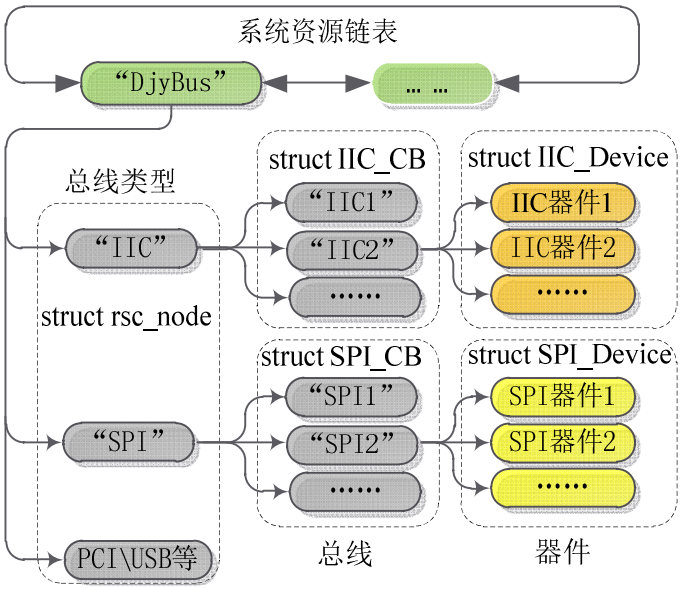


图 21-1 DjyBus 框架示意图

21.2 DjyBus API 说明

系统在创建、删除总线类型等方面提供以下 API 函数：

21.2.1 ModuleInit_DjyBus：初始化 DjyBus

struct rsc_node * ModuleInit_DjyBus(ptu32_t para);

头文件：

djybus.h

参数：无。

返回值：

true=成功，false=失败。

说明：

初始化DjyBus。

版权所有，转载必须注明出处，注明：来自 www.djyos.com 字样。

21.2.2 DjyBus_BusTypeAdd：添加总线类型

struct rsc_node * DjyBus_BusTypeAdd (char* NewBusTypeName);

头文件：

djybus.h

参数：

NewBusTypeName，总线类型结点的名称。

返回值：

新增的总线类型结点指针，增加失败时返回NULL。

说明：

在总线根结点“DjyBus”上增加总线类型结点，使用内存池分配的方式分配总线类型资源节点所需的内存。

21.2.3 DjyBus_BusTypeAdd_r：添加静态总线类型

struct rsc_node *DjyBus_BusTypeAdd_r(char* NewBusTypeName, struct rsc_node * NewBusType);

头文件：

djybus.h

参数：

NewBusTypeName,创建总线类型名称；

NewBusType,创建总线类型结点，用记定义。

返回值：

总线类型结点指针，NULL表示失败。

说明：

在总线根结点“DjyBus”上增加总线类型结点，调用者提供总线资源节点内存。

举例如下：

在spibus模块中，添加bus总线类型。

```
struct tagRscNode *ModuleInit_SPIBus(ptu32_t Para)
{
    return DjyBus_BusTypeAdd_r("DJY_SPIBUS",&s_SPIBusType);
}
```

21.2.4 DjyBus_BusTypeDelete：删除总线类型

bool_t DjyBus_BusTypeDelete(struct rsc_node * DelBusType);

头文件：

djybus.h

参数：

DelBusType，删除的总线类型结点指针。

返回值：

true,删除成功；false,删除失败。

说明：

删除树根结点“DjyBus”上的总线类型子结点。

21.2.5 DjyBus_BusTypeDelet_r：删除静态总线类型

bool_t DjyBus_BusTypeDelete(struct rsc_node * DelBusType);

头文件：

djybus.h

参数：

DelBusType，删除的总线类型结点指针。

返回值：

true,删除成功；false,删除失败。

说明：

删除树根结点“DjyBus”上的总线类型子结点。

21.2.6 DjyBus_BusTypeFind：查找总线类型

struct rsc_node *DjyBus_BusFind(char *BusTypeName);

头文件：

djybus.h

参数：

BusTypeName，查找的总线类型结点名称。

返回值：

总线类型结点指针，未查找到返回NULL。

说明：

在树根结点“DjyBus”上查找总线类型结点 。

21.3 IIC 驱动概述

DjyBus V1.1.0 版本的 IIC 驱动设计适用性说明如下：

1. IIC 工作于主模式，与从设备进行通信；
2. 不支持自动重发机制，发生错误时，采用弹出事件的方式告知应用层；
3. 不支持仲裁丢失重新竞争机制，采用弹出事件方式告知应用层。

21.3.1 IIC 驱动框架

DjyBus 将 IIC 程序分成 IIC 通用接口和底层驱动两个部分，如图 21-2 所示。IIC 通用接口是不以具体硬件平台而变化的公共代码部分，它为上层应用（通常是挂接在 IIC 总线上的器件驱动，如 PCF8563）提供标准化接口函数，并对 IIC 总线操作提供各种信号量互斥访问和缓冲区保护。

底层驱动部分与具体硬件平台息息相关，IIC 驱动移植到不同的平台时，需要在该层进行硬件适配。DjyBus 已经将整个底层驱动的框架搭建完整，移植到不同平台只需要填鸭式的完成寄存器级的工作，使重复性工作尽量降少。

底层驱动程序被通用接口层的程序调用是通过注册回调函数的方式被初始化，将通用接口与底层隔离，通用接口层可进行独立编译，与具体硬件耦合度降低。

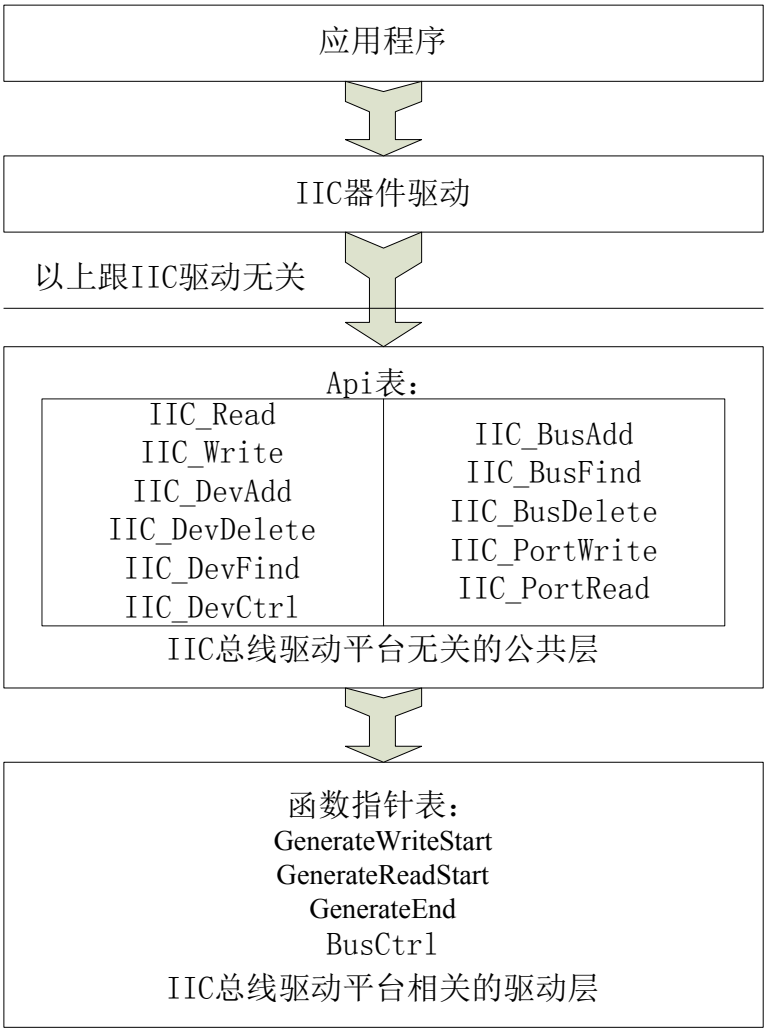


图 21-2 IIC 驱动框架示意图

21.3.2 IIC 总线主要数据结构

IIC 总线控制块结构体 tagIIC_CB

```
struct IIC_CB
{
    struct rsc_node    iic_bus_node;              //总线资源节点//1
    struct IIC_Buf     iic _buf;                  //IIC 缓冲区//2
    struct semaphore_LCB    *iic_bus_semp;     //IIC 总线保护信号量//3
    struct semaphore_LCB    *iic_buf_semp;     //缓冲区同步信号量     //4
    u16    recv_error_evtt;                    //出错处理事件的类型//5
    u32    counter;                            //发送或接收的计数器//6
```

版权所有，转载必须注明出处，注明：来自 www.djyos.com 字样。

```
    u8 *pbuf;                                   //发送或接收缓冲区指针
    ptu32_t specific_flag;                      //个性标记//7
    WriteStartFunc    pGenerateWriteStart;    //开始写函数指针//8
    ReadStartFunc     pGenerateReadStart;     //开始读函数指针//9
    GenerateEndFunc   pGenerateEnd;           //停止传输函数指针//10
    IICBusCtrlFunc    pBusCtrl;               //总线控制函数指针//11
```

下面着重说明 IIC 控制块的结构体：

1. 总线资源节点，用于资源树操作。

2. IIC 总线的发送缓冲区，用于发送时将数据缓存，它是实现异步发送的关键部分，若非阻塞发送数据量大于缓冲区大小，则驱动程序发送数据直到剩余发送的数据量达到缓冲区大小，将剩余数据全部写入缓冲区，若非阻塞发送数据量小于缓冲区大小，则数据被写入缓冲区后立刻返回。

3. 总线保护信号量，该信号量初始化为 1，用于确保每次只有一个设备操作 IIC 总线。操作总线前必须获得总线控制权，它相当于一把 IIC 总线保护锁的钥匙，每次读或写总线时都需获得这把钥匙；

4. 缓冲区信号量，用于非阻塞发送，发送数据量大于缓冲区大小时，剩余发送数据量减小到缓冲区大小，并将剩余数据填充到缓冲区后，释放该信号量；

5. 出错处理事件类型，上层用户可通过注册该事件，当发生错误时，弹出该事件，事件处理内容完成由上层用户决定；

6. Counter 发送或接收数据计数器，在 IIC_Write 和 IIC_Read 函数内部初始化，pbuf 用于指示发送或接收时数据存储地址；

7、8、9、10 和 12 是注册接口的回调函数，用于产生读写起始时序，停止时序等，该函数由 IIC 底层驱动层实现；

上述四个函数指针原型如下：

```
typedef bool_t (*WriteStartFunc)(ptu32_t SpecificFlag,u8 DevAddr,
                                 u32 MemAddr,u8 MenAddrLen, u32 Length,
                                 struct tagSemaphoreLCB *IIC_BusSemp);
typedef bool_t (*ReadStartFunc)(ptu32_t SpecificFlag,u8 DevAddr,
                                u32 MemAddr,u8 MemAddrLen, u32 Length,
                                struct tagSemaphoreLCB *IIC_BusSemp);
typedef void (*GenerateEndFunc)(ptu32_t    SpecificFlag);
typedef s32 (*IICBusCtrlFunc)(ptu32_t SpecificFlag,u32 cmd,
                              ptu32_t data1,ptu32_t data2);
```

(二). IIC初始化参数结构体tagIIC_Param

该结构体中元素同 IIC 控制块结构体，该结构体在 IIC 初始化时将各元素值赋值给 IIC 总线控制块中对应的元素，该结构体详细阐述见 21.3.3.1 节。

(三).IIC 器件数据结构 tagIIC_Device

结构 struct tagIIC_Device 用于描述挂在 IIC 总线上的器件的特征，根据 IIC 总线的物理层协议，仅 IIC 总线控制块并不能确定物理层数据传输，还需要结合具体器件的特征。IIC 器件中包含了三个跟总线物理层传输相关的信息：

1. 器件地址；
2. 器件的存储器地址长度；
3. 器件地址中包含多少 bit（0~3）存储器地址。

```
struct IIC_Device
```

```
{
    struct rsc_node dev_node;
    u8 dev_addr;   //七位的器件地址
        u8 bit_of_maddr_in_daddr;   //器件地址中内部地址所占比特位数
    u8 bit_of_maddr;   //器件内部地址寻址总 bit 数
};
```

对 tagIIC_Device 结构体作如下详细说明：

1. 器件地址 dev_addr 是七个比特地址，如 0x50，在总线上体现的地址为 0x80/0x81；
2. bit_of_maddr_in_daddr 是指 dev_addr 的低三个比特中，有 3 个位用于内部存储地址的高位，例如该位为 1 时，表示 dev_addr 的最低位表示器件内部地址的最高位；
3. bit_of_maddr 表示被操作的器件内部地址的总位数，包含器件地址上的比特位。

举例说明：存储大小为 128K 的铁电，器件地址为 0x50，页大小为 64K，则地址范围为 0x00000 ~ 0x1FFFF，若存储地址占用器件地址的 1 个比特，则 dev_addr 为 0x50，bit_of_maddr_in_daddr 为 1，bit_of_maddr 为 17，构成了 128K 的寻址空间。

21.3.3 IIC 底层驱动

DJYOS 提供了标准的 IIC 驱动模型，但是具体的 IIC 驱动芯片底层驱动程序不同，因此硬件驱动需要软件开发人员自己开发。

DJYOS 源代码都有特定的存放位置，如果 IIC 是 CPU 的片内外设，则建议将底层驱动命名为 cpu_peri_iic.c/cpu_peri_iic.h，将其存放在目录路径…\bsp\cpu_peripheral\xxx，xxx 对应于具体的芯片。

Djyos IIC 驱动中与硬件息息相关的部分是底层驱动部分，在该部分，实现了具体了对 IIC 寄存器级别的控制，包括读写数据寄存器、状态寄存器、控制寄存器等。

21.3.3.1 初始化

IIC 初始化函数 IICn_Init()(其中 n 是 IIC 总线编号)，初始化函数是提供给使用 IIC 总线的器件驱动程序员调用的 API 函数，该函数实现了以下所列功能：

1、 给 tagIIC_Param 赋值，为了将 IICn 添加到资源树做准备；
2、 初始化硬件平台，例如 IIC 默认时钟和寄存器的配置，中断和 GPIO 配置等；
3、 调用 IICBusAdd_r 函数添加 IICn 到资源树。

IIC 初始化时，底层驱动是 IIC 控制块的缓冲区的提供者，缓冲区的实体可以为全局数组变量，也可动态分配的缓冲区，而缓冲区大小以及器件的命名等都在硬件初始化时决定，因此装入 IIC 器件是非常重要的一个环节，是底层驱动者的主要工作之一。

tagIIC_Param 结构体是添加 IICBusAdd_r 函数的参数，是初始化 IIC 控制块所需参数的集合，其定义如下：

```
struct tagIIC_Param
{
    char   *BusName;   //总线名称，如 IIC1
    u8   *IICBuf;   //总线缓冲区指针
    u32   IICBufLen;   //总线缓冲区大小，字节
    ptu32_t   SpecificFlag;
```

```
    WriteStartFunc    pGenerateWriteStart;   //启动写，回调函数
    ReadStartFunc    pGenerateReadStart;   //启动读，回调函数
    GenerateEndFunc    pGenerateEnd;   //结束时序，回调函数
    IICBusCtrlFunc    pBusCtrl;   //控制操作，回调函数
};
```

IICBuf 和 IICBufLen 分别是该 IIC 总线缓冲区指针和大小，缓冲区由 IIC 驱动程序提供，必须为全局或静态缓冲区。SpecificFlag 为 IICn 寄存器基址地址，4 个回调函数则是需要 IIC 总线驱动程序员重点完成的工作。

硬件初始化主要是指与 IIC 相关的硬件模块，比如默认配置的 IIC 控制器初始化，GPIO 的初始化（某些 CPU 不需要配置），中断的初始化（如果使用中断发送和接收）等。

有多少条 IIC 总线是由具体的平台决定，因此增加 IIC 总线到 DjyBus 上是由总线驱动程序员完成。

增加 IIC 总线的 API 函数可以调用 IIC_BusAdd 函数或 IIC_BusAdd_r 函数，两者的区别在于，IIC_BusAdd 只需调用者提供已初始化好的参数结构体 IIC_Param，而后者更需要提供 IIC_CB 结构体控制块，该控制块必须为静态或全局变量（建议定义为本 C 文件内部静态变量）。

21.3.3.2 回调函数

总线驱动模块的回调函数是标准接口模块实现 IIC 总线时序的关键，是与具体的硬件寄存器相关的函数，使用回调函数的方式有效的降低了标准驱动模块对底层驱动的耦合，使编译独立。IIC 总线驱动提供的回调函数包括启动写、启动读、停止和控制函数。

1.启动发送

启动发送的回调函数__IIC_GenerateWriteStart 完成了发送数据时 IIC 的启动时序，它完成的功能主要包括如下几项：

1. 根据 specific_flag 的值，将发送数据量和信号量保存在静态变量中（建议使用结构体）；
2. 关中断，产生 start 时序，配置硬件相关寄存器（如模式设置）；
3. 写器件地址，将 dev_addr 左移 1 比特，最低位写 0 后作为器件地址发送，并等待回复 ACK，出错则立即返回 false；
4. 写器件内部地址，将 mem_addr 地址的低 maddr_len 字节发送到总线上，并等待回复 ACK，出错则立即返回 false；
5. 开中断，返回 true。

启动发送执行的流程为 start---->发送器件地址---->发送内部地址，并开启中断，然后返回。对应在时序上，如图 21-3 所示。

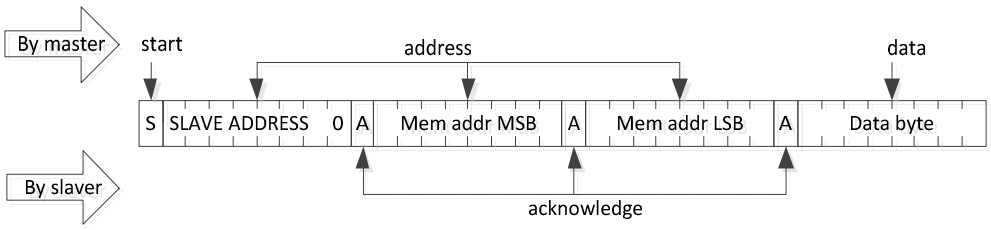


图 21-3IIC 启动发送示意图

如图中所示，写时序是以主控制器发送 start 信号为起始条件，紧接最低位为 0 从器件地址表示写操作，当收到 ACK 信号后会将从器件的存储地址（图中地址为 2 字节，具体多少字节视情况而定）发送到总线，最后发送正式的正文。

回调函数说明如下：

```
bool_t __IIC_GenerateWriteStart (( ptu32_t    specific_flag,
                                   u8 dev_addr, u32 mem_addr,
                                   u8 maddr_len,
                                   u32 length,
                                   struct semaphore_LCB *iic_ semp))
```

参数：

reg：寄存器基址

dev_addr：器件地址的前 7 比特，已将内部地址所占的 bit 位更新，该函数需将该地址左移一位增加最后一位读/写比特；

mem_addr：存储器内部地址，即发送到总线上的地址，该地址未包含放在设备地址上的比特位；

maddr_len：存储器内部地址的长度，字节单位，未包含在设备地址里面的比特位；

length：发送的数据总量，最后一个字节发送完时，需产生停止时序，并释放信号量；

iic_ semp：总线控制信号量，发送完数据后需底层驱动释放。

返回值：

TRUE，启动发送过程正确，FALSE，发生错误。

功能：

产生 IIC 写数据时序，并发送内部地址。

2.启动接收

启动接收__IIC_GenerateReadStart 是在应用程序读数据时被调用的回调函数，该函数需完成的功能主要包含如下：

1. 根据 specific_flag 的值，将发送数据量和信号量保存在静态变量中（建议使用结构体）；
2. 关中断，产生 start 时序，配置硬件相关寄存器（如模式设置）；
3. 写器件地址，将 dev_addr 左移 1 比特，最低位写 1 后发送，并等待回复 ACK，出错则立即返回 false；
4. 写器件内部地址，将 mem_addr 地址的低 maddr_len 字节发送到总线上，并等待回复 ACK，出错则立即返回 false；
5. 重新启动时序，并将 dev_addr 左移 1 比特，最低位写 1 后发送，并等待 ACK，出错则立即返回 false；
6. 开中断，返回 true。

读时序的时序控制过程如图 21-4 所示。该函数依次实现了写 start---->器件地址（写）---->写存储地址---->start（或者 restart）---->器件地址（读）的时序过程。在启动接收时序正确完成后，需使能中断（若不使用中断，则需配置接收到数据 pop 的事件），并配置回复 ACK，在中断中接收从器件发送的数据。

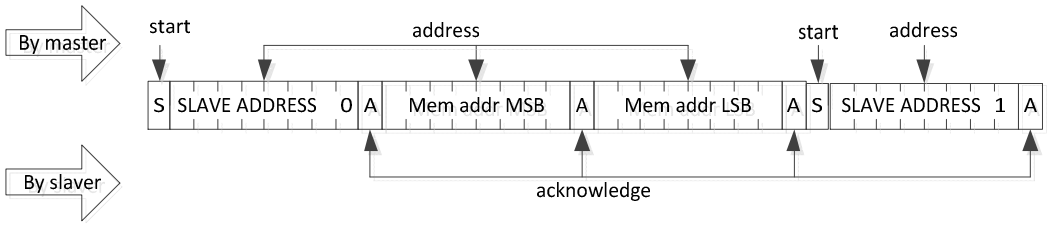


图 21-4 IIC 启动接收示意图

如图所示的时序图中，有两个 start 时序，可能通过配置 repeated 来重新启动一次新的时序，而不产生停止位。

回调函数说明如下：

```
bool_t __IIC_GenerateReadStart ( ptu32_t    reg,u8 dev_addr,
                                 u32 mem_addr,
                                 u8 maddr_len,
                                 u32 length,
                                 struct semaphore_LCB *iic_buf_semp)
```

参数：

reg：寄存器基址

dev_addr：从器件地址的高七比特（同__IIC_GenerateWriteStart 参数说明）

mem_addr：存储器件的内部地址（同__IIC_GenerateWriteStart 参数说明）

maddr_len：存储器件地址长度，字节单位（同__IIC_GenerateWriteStart 参数说明）

length，接收的数据总量，接收数据的倒数第一字节，即 count-1，停止产生 ACK 信号，当接收的字节数为 count 时，产生停止时序，并释放信号量 iic_buf_semp;

iic_buf_semp：发送完成的缓冲区信号量，告知上层，本次发送已经完成。

返回值：

true：启动读时序成功，false 失败。

功能：

完成读时序的启动，并使能中断

3.结束传输

结束传输的回调函数__IIC_GenerateEnd 主要用于停止当前正在进行的传输，特别是在发生超时传输时，用于停止本次发送或接收，实际上，该函数调用了产生停止时序的函数，使 IIC 主器件停止本帧数据的传输。启动和停止时序如图 21-5 所示。

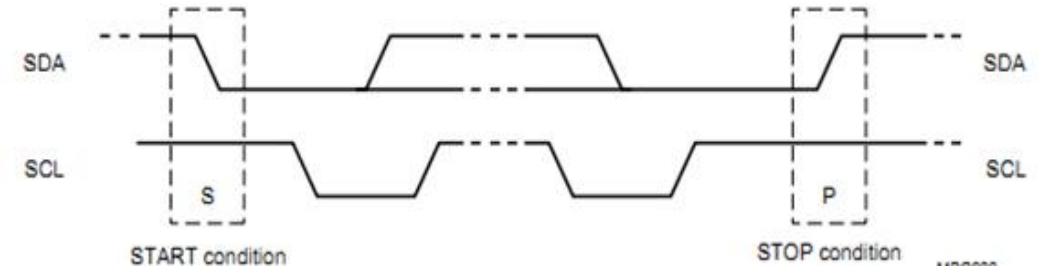


图 21-5 IIC 启动和停止时序示意图

该函数相当于强制停止正在传送的时序，如果是发送过程，因由主机主导整个传送过程，因此可直接产生 STOP，若为接收数据过程，则首先禁止发送 ACK 信号，再产生 STOP。

4.控制函数

目前，控制 IIC 的底层驱动只需要实现对 IIC 总线传输时钟的控制即可，相对较为简单，此处不作详细说明，请参看源码 cpu_peri_iic.c。

21.3.4 中断方式发送及接收

相比轮询的发送和接收方式，中断方式执行效率更高，对 CPU 的消耗更少。由于各种控制器五花八门，因此，中断的具体实现方式也不同。但是基于 DjyBus 设计的 IIC 中断方式接收与发送数据大体的框架和流程基本相似。

IIC 模块对 IIC 总线驱动程序在中断中需要完成的功能作如下要求：

1. 根据中断线或中断标志判断使用的 IIC 控制块和静态变量参数；
2. 发送数据中断中，调用 API 函数 IIC_ReadPort 读取需要发送的参数，并将静态变量计数

器 IntParam->TransCount 递增；
3. 若发送结束，即 IIC_ReadPort 读不到数据，IntParam->TransCount = IntParam->TransTotalLen，则需要产生停止时序和释放信号量；
4. 若为接收数据中断，则需调用 IIC_WritePort 将接收到的数据写入缓冲区，并将计数器 IntParam->TransCount 递增，接收到倒数第二个数据时，还需配置寄存器不发送 ACK 信号；
5. 若接收到所有数据，则需产生停止时序和释放信号量。

下面以 p1020 的 IIC 控制器连接铁电为例，简要讲解一下中断服务函数中的流程。

在中断服务函数内部，通过寄存器判断是发送中断还是接收中断，如图 21-6 所示。发送中断时，需要判断是否收到 IIC 从器件 ACK 信号，然后读简易缓冲区中的数据，并发送之；若缓冲区中为空，判断发送的总量 count 是否为零，若是，则表示该帧数据已经全部发送完毕，需产生停止时序，释放信号量 IntParam->pDrvPostSemp，该信号量是__IIC_GenerateWriteStart 的参数。

在接收中断中，需要判断是否为倒数第二个接收的字节，若是，需要配置控制寄存器不发送 ACK 信号，使控制器接收到倒数第二个字节时不发送 ACK 信号，用于通知从设备接收的数据足够。接收到数据后调用 IIC_PortWrite，该函数将接收到的数据保存到用户缓冲区。若接收到最后一个字节数据，则产生停止时序，并释放信号量 IntParam->pDrvPostSemp，本次接收完成。

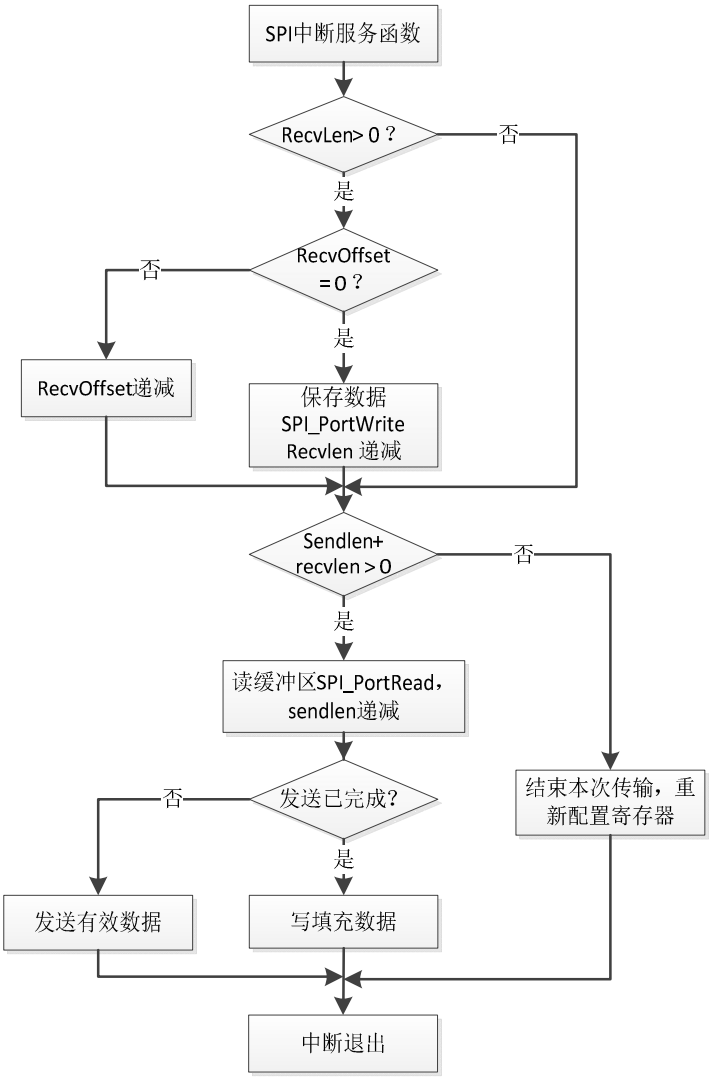


图 21-6 中断方式接收及发送流程示意图

21.4 IIC 驱动 API 函数

21.4.1 ModuleInit_IICBus：IIC 总线初始化

struct rsc_node *ModuleInit_IICBus(ptu32_t Para);

头文件：

iicbus.h

参数：

para：参数保留未使用。

返回值：

返回建立的资源结点指针，失败时返回 NULL。

说明：

将IIC总线类型结点挂接到“djybus”根结点。

21.4.2 IIC_BusAdd：添加 IIC 总线(内核)

struct IIC_CB *IIC_BusAdd(struct IIC_Param *NewIICParam);

头文件：

iicbus.h

参数：

NewIICParam：新增总线参数，参数说明详细请参照 IIC_Param 结构体。

返回值：

返回建立的资源结点指针，失败时返回 NULL。

说明：

新增 IIC 总线结点到 IIC 总线类型结点,并初始化 IIC_CB 控制块结构体。

21.4.3 IIC_BusAdd_r：添加 IIC 总线

struct IIC_CB *IICBusAdd_r(struct IIC_Param *NewIICParam，struct IIC_CB *NewIIC,);

头文件：

iicbus.h

参数：

NewIICParam：新增总线参数，参数说明详细请参照 IIC_Param 结构体；

NewIIC：新增 IIC 控制块指针。

返回值：

返回建立的资源结点指针，失败时返回 NULL。

说明：

新增 IIC 总线结点到 IIC 总线类型结点，并初始化 IIC_CB 控制块结构体。

21.4.4 IIC_BusDelete：删除 IIC 总线(内核)

bool_t IIC_BusDelete(struct IIC_CB *DelIIC);

头文件：

iicbus.h

参数：

DelIIC：删除 IIC 控制块指针。

返回值：

true,删除成功;false,删除失败。

说明：

删除在 IIC 总线类型上的 IIC 结点，若被删除的总线上有器件结点，则删除会不成功,需删除新增结点时 malloc 的控制块内存。

21.4.5 IIC_BusDelete_r：删除 IIC 总线

bool_t IIC_BusDelete_r(struct IIC_CB *DelIIC);

头文件：

iicbus.h

参数：

DelIIC：删除 IIC 控制块指针。

返回值：

true，删除成功；false，删除失败。

说明：

删除在 IIC 总线类型上的 IIC 结点，若被删除的总线上有器件结点，则删除会不成功。

21.4.6 IIC_BusFind：查找 IIC 总线

struct IIC_CB *IIC_BusFind(char *BusName);

头文件：

iicbus.h

参数：

BusName：查找的总线名称。

返回值：

查找的控制块结点指针，未找到时返回 NULL。

说明：

查找 IIC 总线类型结点的子结点中是否存被查找名称的总线。

21.4.7 IIC_DevAdd：IIC 总线添加器件(内核)

struct IIC_Device *IIC_DevAdd(char *BusName ,
char *DevName,

u8 DevAddr,
u8 BitOfMaddrInDaddr,
u8 BitOfMaddr);

头文件：

iicbus.h

参数：

BusName：新增子器件挂接的总线名称。

DevName：器件名称。

DevAddr：器件地址，低七个比特有效，未包含读写比特，如 0x50。

BitOfMaddrInDaddr：存储地址在器件地址的比特位数。

BitOfMaddr：器件存储地址的位数，未包含在器件地址的比特位。

返回值：

设备控制块结点指针，添加失败时返回 NULL。

说明：

在 IIC 总线结点上增加器件，即总线上挂接的器件，该器件属于总线结点的子结点。

举例：如器件存储地址寻址空间为 17 个比特，即 0x1FFFF,BitOfMaddrInDaddr = 1,则 BitOfMaddr = 16。

21.4.8 IIC_DevAdd_r：IIC 总线添加器件

struct IIC_Device *IIC_DevAdd_r(char *BusName,
char *DevName,
struct IIC_Device *NewDev)

头文件：

iicbus.h

参数：

BusName：新增子器件挂接的总线名称。

DevName：器件名称。

NewDev：新增器件指针。

返回值：

设备控制块结点指针，添加失败时返回 NULL。

说明：

将调用者提供的新增器件结点挂接到 IIC 总线结点上。

21.4.9 IIC_DevDelete：删除 IIC 总线上器件(内核)

bool_t IIC_DevDelete(struct IIC_Device *DelDev)

头文件：

iicbus.h

参数：

DelDev,删除的器件指针。

返回值：

true，删除成功；false，删除失败。

说明：

删除 IIC 总线上的器件，从总线子结点中删除。

21.4.10 IIC_DevDelete_r：删除 IIC 总线上器件

bool_t IIC_DevDelete_r(struct IIC_Device *DelDev)

头文件：

iicbus.h

参数：

DelDev,删除的器件指针。

返回值：

true，删除成功；false，删除失败。

说明：

删除 IIC 总线上的器件，从总线子结点中删除。

21.4.11 IIC_DevFind：IIC 总线上查找器件

struct IIC_Device *IIC_DevFind(char *DevName)

头文件：

iicbus.h

参数：

DevName,器件名称。

返回值：

器件控制块构体指针。

说明：

查找器件结点，该结点必然是挂接在相应的 IIC 总线结点上。

21.4.12 IIC_Write：IIC 写

```
s32   IIC_Write(struct IIC_Device *Dev,
                u32 addr,u8 *buf,
                u32 len,
                bool_t block_option,u32 timeout)
```

头文件：

iicbus.h

参数：

DelDev，器件结构体指针；
addr，发送地址，即指存储地址；
buf，发送数据缓冲区指针；
len，发送数据长度，字节单位；
block_option，阻塞选项；
timeout，超时选项。

返回值：

实际发送字节数，或超时。

说明：

上层调用写 API 函数，该函数根据 Dev 结点获得其所在总线结点，调用总线回调函数完成发送启动时序，并且根据阻塞选项，决定是阻塞等待发送完成还是直接返回。timeout 参数用于指定函数在确定的时间内完成，超时则强制返回，并停止时序，释放信号量。

21.4.13 IIC_Read：IIC 读

s32 IIC_Read(struct IIC_Device *Dev,u32 addr, u8 *buf,u32 len,u32 timeout);

头文件：

iicbus.h

参数：

Dev，器件结构体指针；

addr，发送地址，即指存储地址；

buf，发送数据缓冲区指针；

len，发送数据长度，字节单位；

timeout，超时选项。

返回值：

实际读取字节数，或超时。

说明：

读启动函数，该函数首先根据 Dev 查出其所属总线，然后调用总线回调函数，启动读取产生读数据时序，并阻塞等待接收完成，当获得接收完成的信号量时，释放总线控制信号量。超时参数 timeout 用于用户指定在确定时间内完成总线操作，超时则强制返回。

21.4.14 IIC_PortRead：IIC 端口端读

s32 IIC_PortRead(struct IIC_CB *IIC,u8 *buf,u32 len);

头文件：

iicbus.h

参数：

IIC：总线控制块结构体指针；

buf：读数据缓冲区指针；

len：读数据长度，字节单位。

返回值：

实际读到字节数。

说明：

读缓冲区数据，由底层驱动调用，读取的数据会被写入寄存器发送。

21.4.15 IIC_PortWrite：IIC 端口端写

Void IIC_PortRead(struct IIC_CB *IIC,u8 *buf,u32 len);

头文件：

iicbus.h

参数：

IIC：总线控制块结构体指针；

buf：读数据缓冲区指针；

len：读数据长度，字节单位。

返回值：

实际读字节数。

说明：

写数据到缓冲区，由底层驱动调用。

21.4.16 IIC_BusCtrl：IIC 总线控制

s32 IIC_BusCtrl(struct IIC_Device *Dev,u32 cmd,ptu32_t data1,ptu32_t data2);

头文件：

iicbus.h

参数：

Dev,器件结构体指针；

cmd：命令；

data1,data2：参数数据，根据具体命令定义不同。

返回值：

状态，-1=参数检查出错，否则由 cmd 决定，若需要调用用户的 pBusCtrl，则是该函数返回值。

说明：

根据 cmd 命令，解析数据 data1 和 Data2，控制 IIC，硬件相关的命令需调用回调函数由底层驱动完成。

21.4.17 IIC_ErrPop：IIC 弹出错误

s32 IIC_ErrPop(struct IIC_CB *IIC, u32 ErrNo);

头文件：

iicbus.h

参数：

IIC, 总线控制块结构体指针。

ErrNo,错误序号，由 IIC 模块或用户自己确定。

返回值：

状态，无错误时返回 CN_EXIT_NO_ERR

说明：

发生错误时弹出错误事件类型，并将 ErrNo 作为参数传递给事件 。

21.5 SPI 驱动概述

DjyBus V1.1.0 版本 SPII 驱动设计适用性说明如下：
1. SPI 工作于主模式，与从设备进行通信；
2. 不支持自动重发机制，发生错误时，采用弹出事件的方式告知应用层；
3. 读数据必须只能使用阻塞方式，即接收到足够数据后，函数返回。

21.5.1 SPI 驱动框架

类似于 IIC 驱动框架，DjyBus 将 SPI 程序分成 SPI 通用接口和底层驱动两个部分，如图 21-7 所示。

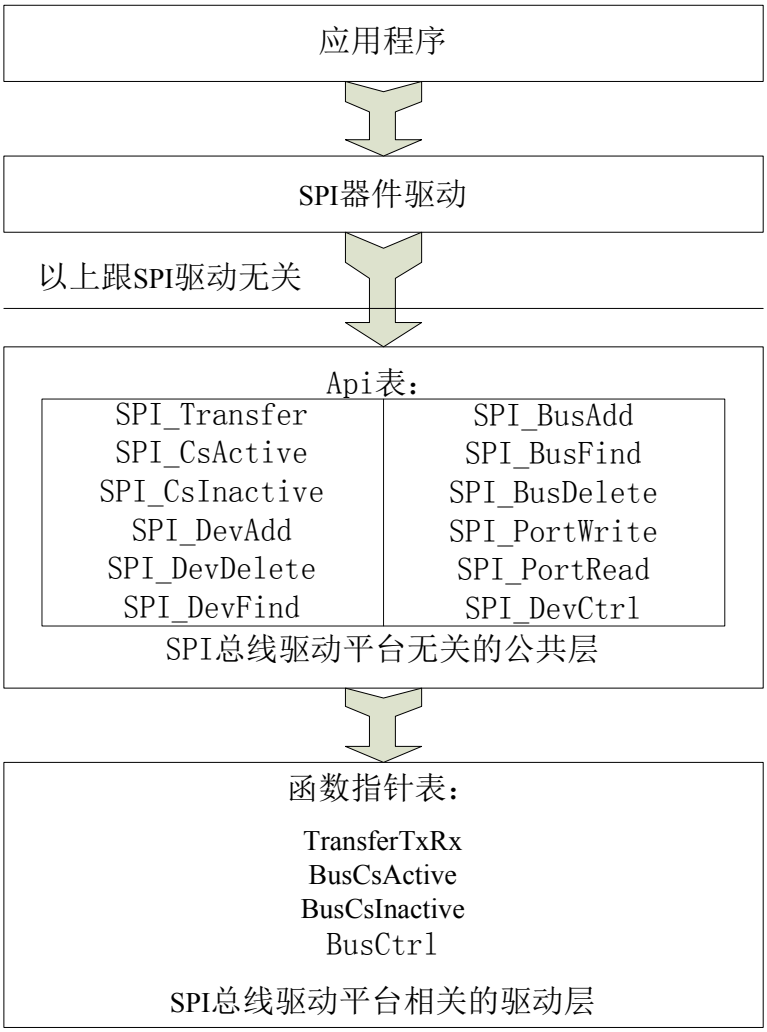


图 21-7 SPI 驱动模块框图

为使 SPI 通用接口层能更好的在多平台移植，同时可有效分离接口层和硬件驱动层，DjyBus 采用将硬件驱动函数注册到 SPI 控制块结构体 SPI_CB 中的方式，接口层通过回调函数方式实现对硬件寄存器控制。

SPI 控制块结构体 SPI_CB 定义如下：

```
struct SPI_CB
```

```
{
    struct rsc_node    spi_bus_node;    //1
    struct SPI_Buf    spi_buf;    //2
    struct semaphore_LCB    * spi_bus_semp;    //3
    struct semaphore_LCB    * spi_block_semp;
    struct SPI_DataFrame    *Frame;    //4
    struct SPI_Device    *CurDev;    //5
    u16    error_pop_evtt;    //6
    bool_t    multi_cs_reg;    //7
    u8    block_option;
    ptu32_t    specific_flag;
    bool_t (*TransferTxRx)(ptu32_t    specific_flag, u32 sendlen,
    u32 recvlen, u32 recvoff);    //8
    bool_t (*BusCsActive)    (ptu32_t specific_flag, u8 cs);    //8
    bool_t (*BusCsInactive)(ptu32_t specific_flag, u8 cs);    //8
    s32 (*BusCtrl)(ptu32_t specific_flag, u32 cmd, ptu32_t data1, ptu32_t data2);    //8
};
```

1. 总线资源结点，通过该资源结点将 SPIn 挂接到 DjyBus 资源树上面；
2. SPI 内部缓冲区，使用非阻塞发送方式时，将发送数据暂存于此缓冲区；
3. spi_bus_semp：总线访问保护信号量，实现同一条 SPI 总线上多个器件对总线的互斥访问，同一时刻只能有一个器件拥有总线。spi_block_semp：阻塞信号量，用于阻塞当前线程直到数据发送完成或全部数据填入缓冲区；
4. 传输数据通信结构体，包含发送接收数据长度和指针，接收偏移量（从接收到的第几个数据开始存储）；
5. 当前占有 SPI 总线的设备指针；
6. 出错事件类型，当发生错误时，弹出该事件类型，事件注册和实现由用户完成；
7. multi_cs_reg ：SPI 控制器每个片选是否具有独立的一套控制寄存器；block_option ：阻塞标记，非零表示阻塞方式，阻塞方式时用户数据被传问发送函数才返回，而非阻塞方式是指，用户数据被全部填写到缓冲区立即返回，此时，数据未必全部发送完成；specific_flag：个性标记，由底层驱动定义和解析，可为 SPI 寄存器基址；
8. SPI 总线控制回调函数，用于操作具体的 SPI 控制器寄存器，实现总线时序的控制。

数据结构 SPI_DataFrame 的成员表示了接收和发送的地址及字节数。

```
struct SPI_DataFrame
{
    u8* sendbuf;      //发送数据指针
    u32 sendlen;      //发送数据长度，字节
    u8* recvbuf;      //接收数据指针
    u32 recvlen;      //接收数据长度，字节
    u32 recvoff;      //接收数据偏移字节，即从接收到的 recvoff 个字节开始存储数据
};
```

结构 struct SPI_Device 用于描述挂在 SPI 总线上的器件的特征，根据 SPI 总线的物理层协议，仅 SPI 总线控制块并不能确定物理层数据传输，还需要结合具体器件的特征。SPI 器件中包含了三个跟总线物理层传输相关的信息：

1. 片选线；

2. auto_cs，从设备是否自动片选；
3. charlen 数据比特位，4-16；
4. mode，总线模式配置，mode 的低两比特表示 CPHA 和 CPOL；
5. freq，传输速度，单位为 Hz。

```
struct SPI_Device
{
    struct rsc_node dev_node; //资源结点
    u8 cs; //片选信号
    bool_t auto_cs; //是否自动片选
    u8 charlen; //数据长度
    u8 mode; //模式选择
    u32 freq; //速度,Hz
} ;
```

21.5.2 SPI 底层驱动实现

SPI 底层驱动与具体硬件平台息息相关，基于 DjyBus 总线架构的 SPI 驱动移植到不同平台时，需在底层驱动层进行硬件适配。在底层驱动层需实现 SPI 初始化及 SPI 总线层调用的回调函数。这些回调函数是 SPI 总线实现对 SPI 底层操作的关键。使用回调函数的方式有效降低通用 SPI 接口模块和硬件驱动之间耦合，使编译独立化。

21.5.2.1 初始化

bool_t SPI_Init(void)
头文件：
无。
参数：
无。
返回值：
初始化成功返回 1，失败返回 0。
功能：
SPI 初始化与具体平台相关，因此由硬件驱动程序员完成，SPI 初始化函数需完成以下功能：
1. 初始化硬件，默认时钟频率配置，GPIO 配置（如果需要）；
2. 初始化 SPI 中断，注册中断号和中断服务函数；
3. 初始化 SPI 总线参数结构体 SPI_Param 并调用 SPI_BusAdd_r，新增一个 SPI 总线结点到 SPI 总线结点下。

SPI 初始化时，底层驱动是 SPI 控制块缓冲区的提供者，缓冲区实体可为全局数组变量，也可为动态分配缓冲区，而缓冲区大小及器件命名等都在硬件初始化时决定，因此初始化 SPI 器件参数是重要的一个环节，是底层驱动者的主要工作之一。

21.5.2.2 发送接收

static bool_t __SPI_TransferTxRx(ptu32_t specific_flag,
u32 sendlen,
u32 recvlen,
u32 recvoff)
头文件：
无
参数：。
specific_flag：寄存器基址。
sendlen：发送数据长度，字节单位。
recvlen：接收数据长度，字节单位。
recvoff：接收偏移，接收到多少个字节后开始保护数据，即有用数据。
返回值：
成功返回 true；失败在返回 false。
功能：
SPI 总线采用四线收发方式，收、发、时钟和片选。__SPI_TranferTxRx 实质上是实现了相关的寄存器的配置，并使能中断，交由中断完成 SPI 接收或发送。
1. 保存静态变量，如发送接收数据长度，接收偏移（从接收到的第几个数据开始保存数据）；
2. 配置 SPI 寄存器，使其处于发送和接收的状态；
3. 配置中断使能，并触发中断，在中断中将数据发送接收完成；
4. 若接收或发送的数据全部完成，则重新配置寄存器，使其处于初始状态。

21.5.2.3 使能片选

static void__SPI_BusCsActive(ptu32_t specific_flag, u8 cs)
头文件：
无。
参数：
specific_flag：寄存器基址。
cs：片选号。
返回值：
空。
功能：
使能指定片选信号。
虽然对于具体的芯片，该函数的实现过程不相同，但是功能是相同的，选择 CS 对应的 SPI 从器件通信。若控制器具有硬件自动片选，硬件驱动可加以利用，提高效率。

21.5.2.4 禁能片选

static void __SPI_BusCsInActive(ptu32_t specific_flag, u8 cs)
头文件：

无。
参数：
specific_flag：寄存器基址。
cs：片选号。
返回：
空。
功能：
禁能片选。

21.5.2.5 控制函数

static s32 __SPI_BusCtrl(tagSpiReg *Reg,u32 cmd,ptu32_t data1,ptu32_t data2)
头文件：
无。
参数：
specific_flag：个性标记，本模块内即IIC寄存器基址。
cmd：命令标识符。
data1,data2：数据，与具体命令相关。
返回值：
目前，控制函数主要实现对总线的配置，如自动片选、传输速度、SPI 时序配置等。应用层将通过调用 SPI_Ctrl 接口函数，传递不同的命令和参数，实现对总线的控制。

如果 SPI 控制器对每个片选信号都有独立的控制参数配置寄存器，则在添加设备时（SPI_DevAddr），配置好每个片选控制参数寄存器；若多个片选共用一套控制参数配置寄存器，则每次传输都必须重新配置。在结构体 SPI_CB 中，成员 multi_cs_reg 是用来标记 SPI 控制器是否具有多套控制参数寄存器，在调用 SPI_ModuleInit 时，硬件驱动会作相应的标记。
功能：
SPI 总线控制回调函数，被上层调用。

21.5.2.6 SPI 中断发送接收

SPI 接收和发送使用中断方式的好处在于，将发送任务由 SPI 控制器完成，节省 CPU 的处理负荷，因此提高了程序的运行效率。现在绝大多数的主流 CPU 的中断系统都支持 SPI 中断，包括发送接收中断等。
SPI 模块要求在中断服务函数内部完成的功能有如下：
1. 清中断标志，处理好接收与发送数据同时进行的硬件机制；
2. 接收数据从接收到 recvoff 字符数据后开始存储，即调用 SPI_PortWrite；
3. 发送数据从 SPI_PortRead 读取，若没有读到数据，则代表数据已经发送完成；若此时接收的数据还未完成，应该继续往寄存器中写数据，直到接收完成；
4. 数据传输完成时，配置相应的寄存器，使其处于初始状态（视控制器而定）；
下面以 Atmel 芯片为例，通过流程图的方式简要说明 SPI 中断服务函数的数据处理流程，如图 21-8 所示。值得注意的是，中断服务函数中有些变量是通过__SPI_TransferTxRx 传递参数到底层硬件驱动，底层驱动通过静态变量存储，并在中断服务函数中使用。如发送接收数据大小，信号量等。

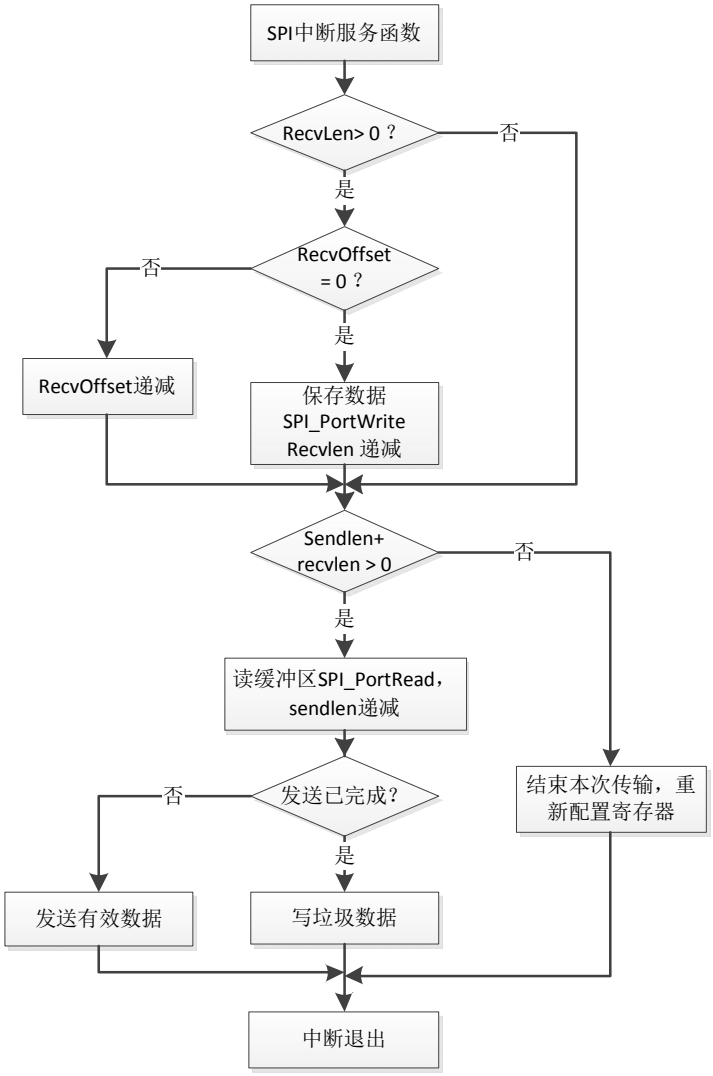


图 21-8 SPI 中断发送接收流程示意图

21.6 SPI 驱动 API

21.6.1 ModuleInit_SPIBus：SPI 总线初始化

struct rsc_node *ModuleInit_SPIBus(ptu32_t Para)

头文件：

spibus.h

参数：

para：无实际意义。

返回值：

返回建立的资源结点指针，失败时返回 NULL。

说明：

将SPI总线类型结点挂接到“djybus”根结点上。

21.6.2 SPI_BusAdd：添加 SPI 总线(内核)

struct SPI_CB *SPI_BusAdd(struct SPI_Param *NewSPIParam);

头文件：

spibus.h

参数：

NewSPIParam：新增SPI总线参数，参数说明详细请参照SPI_Param结构体。

返回值：

返回建立的SPI总线控制块指针，失败时返回NULL。

说明：

新增SPI总线结点到SPI总线类型结点上,由SPI_BusAdd函数分配SPI_CB控制块结构体内存，同时根据SPI总线参数NewSPIParam初始化SPI_CB。

21.6.3 SPI_BusAdd_r：添加 SPI 总线

struct SPI_CB *SPIBusAdd_r(struct SPI_Param *NewSPIParam,struct SPI_CB *NewSPI)

头文件：

spibus.h

参数：

NewSPIParam：新增总线参数，参数说明详细请参照 SPI_Param 结构体；

NewSPI：调用方提供的 SPI 控制块。

返回值：

返回建立的 SPI 总线控制块指针，失败时返回 NULL。

说明：

新增 SPI 总线结点到 SPI 总线类型结点,并初始化 SPI_CB 控制块结构体，调用方提供 SPI_CB 控制块。

21.6.4 SPI_BusDelete：删除 SPI 总线(内核)

bool_t SPI_BusDelete(struct SPI_CB *DelSPI);

头文件：

spibus.h

参数：

DelSPI：删除SPI控制块指针。

返回值：

true,删除成功;false,删除失败。

说明：

删除 SPI 总线类型上指定 SPI 总线结点，若被删除总线上有器件结点，则删除会失败，直接返回；若被删除的总线上没有器件结点，则删除该 SPI 总线控制块内存。

21.6.5 SPI_BusDelete_r：删除 SPI 总线

bool_t SPI_BusDelete_r(struct SPI_CB *DelSPI)

头文件：

spibus.h

参数：

DelSPI：删除SPI控制块指针。

返回值：

true,删除成功;false,删除失败。

说明：

删除 SPI 总线类型上指定 SPI 总线结点，若被删除的总线上有器件结点，则删除会不成功。

21.6.6 SPI_BusFind：查找 SPI 总线

struct SPI_CB *SPI_BusFind(char *BusName);

头文件：

spibus.h

参数：

BusName：查找的总线名称。

返回值：

若成功找到则返回找到的总线控制块指针，未找到时返回 NULL。

说明：

查找 SPI 总线类型结点的子结点中是否存在被查找名称的总线。

21.6.7 SPI_DevAdd：SPI 总线上添加器件(内核)

```
struct tagSPI_Device *SPI_DevAdd(char *BusName,
                                 char *DevName,
                                 u8 cs,
                                 u8 charlen,u8 mode,
                                 u8 shiftdir,
                                 u32 freq,
                                 u8 autocs)
```

头文件：

spibus.h

参数：

BusName：新增子器件挂接总线名称。

DevName：器件名称。

cs：该器件片选号。

charlen：传输的字符宽度，可能的值为 4-16 比特。

shiftdir：MSB or LSB。

mode：SPI 传输模式，根据相伴和电平共有四种模式。
freq：总线传输的时钟频率，单位 Hz。
autocs：新增加设备是否自动片选。
返回值：
成功时返回新增 SPI 器件参数结构体指针，失败时返回 NULL。
说明：
在 SPI 总线结点上增加器件，即总线上挂接器件，该器件属于总线结点的子结点。由 SPI_DevAdd 从内存池中分配器件结构体内存。

21.6.8 SPI_DevAdd_r：SPI 总线上添加器件

```
struct tagSPI_Device *SPI_DevAdd_r(char *BusName,
                                   char *DevName,
                                   struct tagSPI_Device *NewDev)
```

头文件：
spibus.h
参数：
BusName：新增子器件挂接的总线名称。
DevName：器件名称。
NewDev：调用方提供的 SPI 总线器件参数结构体指针。
返回值：
成功时返回 SPI 总线器件参数结构体指针，失败时返回 NULL。
说明：
在 SPI 总线结点上增加器件，即总线上挂接器件，该器件属于总线结点的子结点，由调用方提供 NewDev 结构体指针。

21.6.9 SPI_DevDelete：SPI 总线上删除器件(内核)

```
bool_t    SPI_DevDelete(struct SPI_Device *DelDev);
```

头文件：
spibus.h
参数：
DelDev：删除的器件参数结构体指针。
返回值：
true,删除成功;false,删除失败。
说明：
删除 SPI 总线上指定器件，从总线子结点中删除，同时释放 SPI 总线器件参数结构体所占内存。

21.6.10 SPI_DevDelete_r：SPI 总线上删除器件

```
bool_t    SPI_DevDelete_r(struct SPI_Device *DelDev)
```

头文件：

spibus.h

参数：

DelDev：删除的器件参数结构体指针。

返回值：

true,删除成功;false,删除失败。

说明：

删除 SPI 总线上指定器件，从总线子结点中删除。

21.6.11 SPI_DevFind：SPI 总线上查找器件

struct SPI_Device *SPI_DevFind(char *DevName)

头文件：

spibus.h

参数：

DevName：查找的器件名称。

返回值：

查找的 SPI 器件参数结构体指针，未找到时返回 NULL。

说明：

查找 SPI 总线类型结点的子结点中是否存在被查找名称器件。

21.6.12 SPI_Transfer：SPI 数据传输

```
s32 SPI_Transfer(struct SPI_Device *Dev,
                 struct SPI_DataFrame *spidata,
                 u8 block_option,u32 timeout)
```

头文件：

spibus.h

参数：

Dev：器件指针；

spidata：SPI 数据结构体；

block_option：阻塞选项，true 时，表明最后一次传输为阻塞方式，否则为非阻塞；

timeout：超时参数，单位 us。

返回值：

返回发送状态，超时或错误或无错误。

说明：

数据传送函数，完成数据的发送和接收。该函数完成的功能如下：

1. 若器件驱动为了避免组包的麻烦，可先发命令再发送数据，分多次调用，多次调用前后被 CSActive 和 CsInactive 函数包裹；
2. 根据 Dev 查找所属 SPI 总线；
3. 若缓冲区大于发送字节数，则直接将数据填入缓冲区；
4. 若为阻塞发送，则等待总线信号量，若为非阻塞，则等待 buf 信号量；
5. 发生超时或错误时，拉高 CS 并释放信号量。

21.6.13 SPI_CsActive：使能片选信号

bool_t SPI_CsActive(struct SPI_Device *Dev,u32 timeout);

头文件：

spibus.h

参数：

Dev：器件参数结构体指针。

timeout：超时参数，单位us。

返回值：

true,操作成功;false，操作失败。

说明：

使能片选，该函数运行必须获得总线信号量，即spi_bus_semp，然后根据CS是否具有自己的配置寄存器标志，判断是否需要配置寄存器。若multi_cs_reg标志为false,则每次CS使能时，都需要配置寄存器。

21.6.14 SPI_CsInactive：禁能片选信号

bool_t SPI_CsInactive(struct SPI_Device *Dev);

头文件：

spibus.h

参数：

Dev：器件参数结构体指针。

timeout：超时参数，单位us。

返回值：

true,操作成功;false，操作失败。

说明：

禁能片选,同时释放SPI总线信号量。

21.6.15 SPI_PortRead：SPI 端口端读

s32 SPI_PortRead(struct SPI_CB *SPI,u8 *buf,u32 len);

头文件：

spibus.h

参数：

SPI：SPI 控制块结构体指针。

buf：存放读到的数据缓冲区指针。

len：需要读的数据长度，单位字节。

返回值：

读到的字节数。

说明：

读发送缓冲区数据，由底层驱动调用，总线驱动读取到数据后将数据写入寄存器发送，若阻塞发送，则直接读发送缓冲区指向的数据，若非阻塞发送，则需判断是否剩余数据已达

到缓冲区边界，若已到达缓冲区边界，则填写缓冲区，并释放阻塞信号量。

21.6.16 SPI_Port_Write：SPI 端口端写

s32 SPI_PortWrite(struct SPI_CB *SPI,u8 *buf,u32 len);

头文件：

spibus.h

参数：

SPI：SPI 控制块结构体指针。

buf：数据缓冲区指针。

len：数据长度，字节单位。

返回值：

成功写的字节数。

说明：

将接收到的数据写入用户提供的缓冲区中，接收是阻塞方式，因此使用用户的缓冲区。

21.6.17 SPI_BusCtrl：SPI 总线控制

s32 SPI_BusCtrl(struct SPI_Device *Dev,u32 cmd,ptu32_t data1,ptu32_t data2);

头文件：

spibus.h

参数：

Dev：器件参数结构体指针。

cmd：命令。

data1,data2：参数数据，根据具体命令定义不同。

返回值：

状态，-1=参数检查出错，否则由 cmd 决定，若需要调用用户的 pBusCtrl，则是该函数返回值。

说明：

根据cmd命令，解析数据data1和data2，控制SPI，硬件相关的命令需调用回调函数，这些回调函数由底层驱动完成。

21.6.18 SPI_ErrPop：SPI 错误弹出

s32 SPI_ErrPop(struct SPI_CB *SPI, u32 ErrNo);

头文件：

spibus.h

参数：

SPI：总线控制块结构体指针；

ErrNo：错误序号，由 SPI 模块或用户自己确定。

返回值：

状态，无错误时返回CN_SPI_EXIT_NOERR。

说明：

发生错误时弹出错误事件类型，并将ErrNo作为参数传递给事件 。

21.7 SPI 驱动使用实例

下面以时钟芯片 DS1390 为例阐述 SPI 通用驱动架构的使用方法，步骤如下：

Step1：组件初始化阶段，调用系统函数 DjyBus_ModuleInit(0)创建 DjyBus 根节点；

Step2：组件初始化阶段，调用系统函数 ModuleInit_SPIBus(0)在 DjyBus 根节点下添加 DJY_SPIBUS 总线类型结点；

Step3：组件初始化阶段，调用底层驱动提供的函数 SPI_Init()，初始化 SPI 并将 SPI 总线结点添加到 DJY_SPIBUS 总线类型结点下。

Step4：编写 SPI 器件驱动。

SPI 器件驱动中要实现以下三个函数：RTC_ModuleInit、__rtc_read、__rtc_write。RTC_ModuleInit 函数中需配置 SPI 器件参数并将 SPI 器件时钟芯片添加到 SPI 总线结点下。实例代码如下：

```
ptu32_t RTC_ModuleInit(ptu32_t para)
{
    static struct tagSPI_Device s_DS1390_Dev;
    s_DS1390_Dev.AutoCs     = true;
    s_DS1390_Dev.CharLen    = 8;
    s_DS1390_Dev.Cs         = CN_DS1390_SPI_CS;
    s_DS1390_Dev.Freq       = CN_DS1390_SPI_FRE;
    s_DS1390_Dev.Mode       = SPI_MODE_0;
    s_DS1390_Dev.ShiftDir   = SPI_SHIFT_MSB;
    if(NULL != SPI_DevAdd_r("SPI","SPI_Dev_DS1390",&s_DS1390_Dev))
    {
        s_DS1390_InitFlag = true;
        ps_DS1390_Dev = &s_DS1390_Dev;
        Rtc_RegisterDev(RTC_TimeGet,RTC_TimeUpdate);
    }
    return true;
}
```

__rtc_read 及__rtc_write 也即通过 SPI 读/写 SPI 器件，器件驱动需要做的是对 SPI 传输数据参数结构体赋值，SPI 读之前需调用系统函数 SPI_CsActive 使能片选信号，SPI 读调用系统函数 SPI_Transfer 进行 SPI 读/写，在完成 SPI 传输后需要禁能 SPI 片选信号。实例代码如下：

```
static unsigned char __rtc_read (unsigned char reg)
{
    unsigned char ret;
    struct tagSPI_DataFrame data;
    data.RecvBuf = &ret;
    data.RecvLen = 1;
    data.RecvOff = 1;
    data.SendBuf = &reg;
```

```
    data.SendLen = 1;
    SPI_CsActive(ps_DS1390_Dev,CN_DS1390_TIMEOUT);
    SPI_Transfer(ps_DS1390_Dev,&data,true,CN_DS1390_TIMEOUT);
    SPI_CsInactive(ps_DS1390_Dev);
    return ret;
}
```

第22章　多路复用

22.1 Multiplex 概述

所谓多路复用，就是一个处理程序处理多个传输对象的输入输出数据，例如网络服务器，需要面对成千上万个 socket，如果每个 socket 都创建一个线程去处理的话，占用大量资源，且频繁线程切换也会消耗太多 cpu 资源，显然是不划算的，Multiplex 可以解决这个问题。

Djyos 中，Multiplex 是一个独立的功能模块，并不与文件系统发生关联。文件系统、设备模型、网络模块也是互相独立的，没有关联性。自然，文件描述符、设备描述符、socket 描述符，也是没有直接的关联的对象。所以，在 Djyos 中，不像 Linux，可以找到一个统一的万能对象：文件描述符，来承载 select。Djyos 中，提供多路复用功能的模块，其实现方法和使用方法都和标准的 select 不一样，为避免误读，直接命名为 Multiplex。脱离文件系统独立实现的 Multiplex，提供了一些简单的方法和接口，任何模块，只要按照这些标准接口编码，就能成为 Multiplex 的目标，并不要求其一定要加入设备队列中。例如多个 CAN 口，任何一个 CAN 口收到数据时，都激活 CAN 口服务器接收数据处理，只要把各 CAN 口加入到 Multiplex 的对象中就可以。但是因为 CAN 口是该进程私有的的硬件，无论从执行效率开发效率来说，程序员都不希望套用复杂的标准驱动模型来编写该 CAN 驱动程序，只想直接在进程空间编写一个简单的驱动程序。这种情况下，用 Linux 是无法实现 Multiplex 的，而 Djyos 的 Multiplex 模块是独立的组件，在这方面没有任何限制。

事件处理过程中（线程运行中），创建 MultiplexSets 并可以把多个对象(Object)加入 MultiplexSets 中，并设定感兴趣的 bit，以及触发条件（“与”或者“或”），然后阻塞等待 MultiplexSets 被触发。MultiplexSets 和 Multiplex Object 可以分别设定触发条件，MultiplexSets 的触发条件含义：“或”方式是指，MultiplexSets 中的对象，只要有一个被触发，即 MultiplexSets 被触发。“与”方式是指，MultiplexSets 中的对象，全部被触发后，MultiplexSets 才被触发。Object 的触发条件含义为：“或”方式是指，Object 中，只要有一个该 Object 感兴趣的 bit 被触发，Object 就被触发。“与”方式是指，Object 中，只有全部该 Object 感兴趣的 bit 被触发，Object 才被触发。每个 Object 可以单独设定感兴趣的 bit。

一个对象，也可以同时被多个 MultiplexSets 包含，对象被触发时，所有这些 MultiplexSets 将同时受影响。

22.2 Multiplex 层次框架

从用户角度看，Multiplex 模块与其他模块的层次框架大致如图 22-1 所示，Multiplex 是一个相对底层的模块，应用程序一般不会单独使用它，而是需要被 Multiplex 的对象相关的模块协助。MultiplexSets 中包含多种对象，如果 MultiplexSets 包含设备，那么就需要设备

驱动相关模块支持；如果 MultiplexSets 中包含了 Socket，就需要协议栈模块协助，等等。

Multiplex 模块提供的主要 API 有 5 个，其中绿色框内的两个通常是由应用程序直接调用的；蓝色框内的三个，虽然应用程序也可以直接调用，但通常由中间模块过渡间接调用。例如，devMultiplexAdd 函数将把设备 ID 加入到 MultiplexSets 中，devMultiplexAdd 函数内部将辗转调用 MultiplexAdd 函数完成把设备 ID 加入 MultiplexSets 的操作。

这里有个细节，在 Linux 中，能加入 MultiplexSets 的对象，只有文件句柄，djyos 并没有这个限制，它可以是任何模块。在 Multiplex 模块看来，图中的“设备驱动、socket、文件系统”和“其他”是等同的，并无特殊之处。

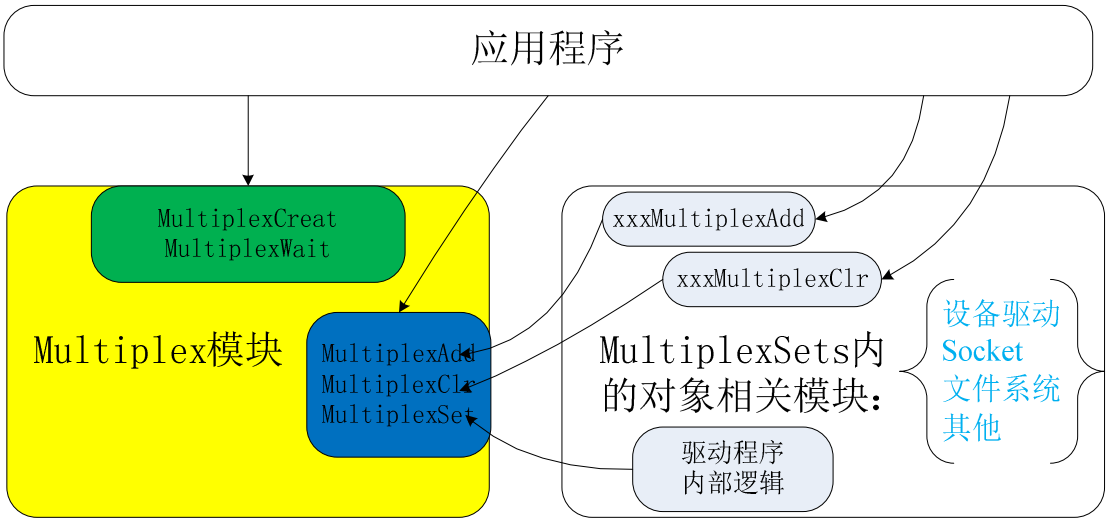


图 22-1Multiplex 框架示意图

应用程序使用 Multiplex 模块过程，大体如图 22-2 所示。

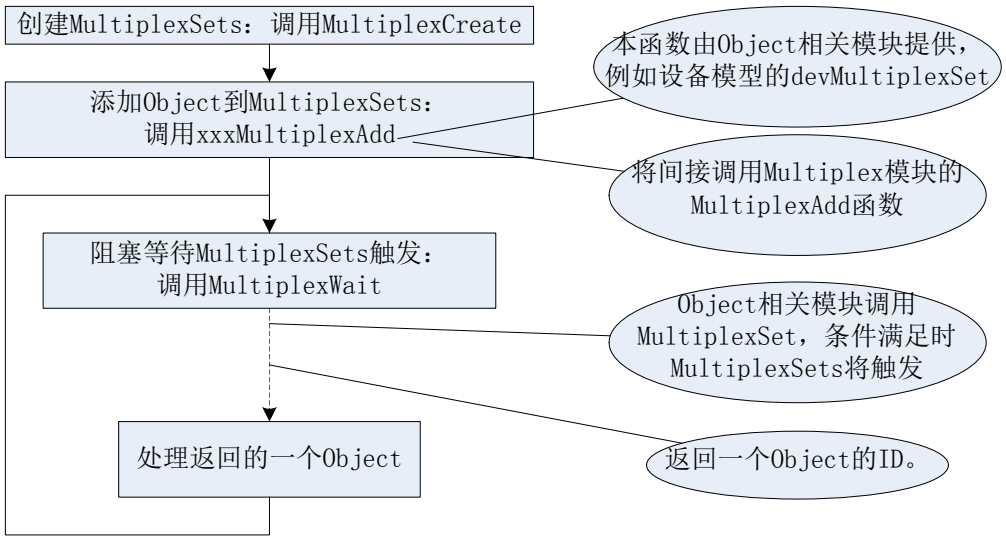


图 22-2 Multiplex 运行过程

1、 应用程序调用 MultiplexCreate 创建 MultiplexSets，获得其指针 pSets。
2、 调用 MultiplexAdd（可多次调用）函数把被 Multiplex 的 Object 加入 pSets 中，应用程序一般不会直接调用，而是通过 Object 相关模块间接调用。
3、 应用程序调用 MultiplexWait 函数，阻塞等待 MultiplexSets 被触发。
4、 MultiplexSets 被触发后，MultiplexWait 函数将返回触发的对象 ID，应用程序完成数据处理。
5、 反复执行 3、4 两步骤，直到程序结束，实现多路复用。

版权所有，转载必须注明出处，注明：来自 www.djyos.com 字样。

Object 相关模块配合过程，以设备驱动框架为例如下：

1、 提供 devMultiplexAdd 函数供应用程序调用，用户调用 devMultiplexAdd 函数时，将间接调用到 Multiplex 模块的 MultiplexAdd 函数，把一个或一组设备 ID 添加到 MultiplexSets 中。

2、 当被 Multiplex 的设备状态发生变化时，驱动程序调用 MultiplexSet 把状态告诉 Multiplex 模块。

22.3 API 说明

22.3.1 MultiplexModuleInit：组件初始化

ptu32_t MultiplexModuleInit(ptu32_t para);

头文件：

multiplex.h

参数：

para：无意义。

返回值：

-1 表示出错，一般是因为内存池容量不足。0 表示成功。

说明：

MultiplexModuleInit 函数完成模块初始化工作，模块初始化的工作很简单：

1、创建互斥量，该互斥量用来保护多路复用集链表访问。

2、创建 MultiplexSets 控制块内存池和 MultiplexObject 控制块内存池，这两个内存池的初始尺寸都是 0，软件运行中实际需要时，才从堆中动态分配。

22.3.2 Multiplex_Creat：创建 MultiplexSets

struct tagMultiplexSetsCB *Multiplex_Creat(u32 ActiveLevel);

头文件：

multiplex.h

参数：

ActiveLevel：触发水平。

返回值：

新创建的 MultiplexSets 指针。

说明：

创建一个 MultiplexSets，并且设定属性。

22.3.3 Multiplex_AddObject：添加 Object 到 MultiplexSets

```
bool_t Multiplex_AddObject(struct tagMultiplexSetsCB    *Sets,
                           struct tagMultiplexObjectCB **ObjectHead,
                           u32 ObjectStatus,ptu32_t ObjectID, u32 SensingBit);
```

头文件：
multiplex.h
参数：
Sets，被操作的 MultiplexSets 指针；
ObjectHead，被操作的 Object 队列头指针的指针，一个对象第一次调用本函数前，*ObjectHead 应该被初始化为 NULL，第一次调用 MultiplexSet 以后，*ObjectHead 值再也不允许在外部修改，否则结果不可预料；
ObjectStatus，加入时的状态，31bit 的位元组,bit31 无效，与 MultiplexSets 的 InterestingFlag 成员对应，Object 加入 MultiplexSets 时的状态，该状态决定 Object 的初始状态是已触发态还是未触发态，还可能是部分触发态；
ObjectID，被 Multiplex 的对象的 ID，新加入的 Object 的 ID，由使用者约定，比如设备的 ObjectID 是设备 ID，socket 的 ID 是 socket 号；
SensingBit，对象敏感位标志，31 个 bit，设为 1 表示本对象对这个 bit 标志敏感 bit31 表示敏感类型，CN_SENSINGBIT_AND，或者 CN_SENSINGBIT_OR。
返回值：
true=成功，false=失败。
说明：
MultiplexSets 中添加一个对象。如果该 Object 的初始状态是已经触发，则加入到 ActiveQ 队列，否则加入 ObjectQ 队列。

22.3.4 Multiplex_DelObject：MultiplexSets 中删除 Object

函数原型：

```
bool_t Multiplex_DelObject(struct tagMultiplexSetsCB    *Sets,
                           struct tagMultiplexObjectCB **ObjectHead);
```

头文件：
multiplex.h
参数：
参数含义也跟 MultiplexAdd 一致，不再赘述。
返回值：
true=成功，false=失败。
说明：
Multiplex_DelObject 是 MultiplexAdd 的逆函数，从多路复用集中删除某个对象。

22.3.5 Multiplex_Set：设置 Object 状态

```
bool_t Multiplex_Set(struct tagMultiplexObjectCB *ObjectHead,u32 Status);
```

头文件：
multiplex.h
参数：
ObjectHead，被操作的 Object 队列的头指针。
Status，Object 的当前状态。
返回值：

true=成功，false=失败。

说明：

当 MultiplexSets 中的对象状态发生变化，由相关模块调用本函数告知 Multiplex 模块。事件处理线程阻塞在 MultiplexSets 上之后，什么时候被唤醒，就看 MultiplexSet 函数了，如果没有人调用 MultiplexSet 函数，线程将一直“睡”下去，或者超时唤醒。当然，也不是调用 MultiplexSet 就必然导致线程唤醒，还得看 Status 参数和 MultiplexSets 的属性设置。

Djyos 是实时操作系统，Multiplex 模块要求被 Multiplex 对象相关模块状态发生变化时，主动调用 MultiplexSet 函数报告状态变化。例如你 Multiplex 了一个串口，就要求串口收到数据、或者发送缓冲区空、或者出错时，主动调用 MultiplexSet 函数报告状态，而不是等待 Multiplex 模块查询。这样既提高了实时性，又降低了 cpu 负担，无须单独准备一个线程来定时查询所有被 Multiplex 的对象状态。

22.3.6 Multiplex_Wait：等待 MultiplexSets 触发

ptu32_t Multiplex_Wait(struct tagMultiplexSetsCB *Sets,u32 *Status,u32 Timeout);

头文件：

multiplex.h

参数：

Sets，被操作的 MultiplexSets 指针；

Status，返回对象当前状态的指针，如果应用程序不关心其状态，可以给 NULL；

Timeout，阻塞等待的最长时间，单位 uS。

返回值：

如果 MultiplexSets 被触发，则返回 MultiplexSets 中一个被触发对象的 ObjectID，否则返回-1。

说明：

设置好 MultiplexSets 后，应用程序调用本函数等待 MultiplexSets 被触发，根据 Type 值，等待方式分轮询和异步触发两种。

第23章　异常管理

23.1 异常管理概述

这里的异常指：系统的逻辑问题、各种硬件问题及各种软件 bug 等，异常的千差万别，意味着我们没有统一的标准去处理，但是我们可以有统一的标准去管理。本文在此谈论的是如何更好的管理我们的异常信息，而不是具体异常的处理，具体异常要随着 OS 或者平台的不同而有具体差异。

DJYOS 的异常管理组件具有以下功能：

1. 为各个需要管理异常信息的组件提供轻松加 EASY 的接口，跳出异常时随便一个 Exp_Throw 就可以搞定了；同时根据制定的异常指示进行下一步动作，如记录、热重启、冷重启等；

2. 搜集的异常信息不仅仅是用户搜集的异常信息，异常组件内部会搜集当时 OS 的状态信息，如果用户注册有 HOOK，还可以在异常时执行该 HOOK 去搜集特定的异常信息。

3. 良好的再现能力，异常信息被搜集存储后，当某年某月某日你想起来要修复这个问题时，

可以把当年搜集的信息完整的重现给你！

4. 易于扩展的存储接口，搜集的异常信息的存储，可以按照用户所想随意存储，这依赖于灵活的异常存储接口，脱离于具体的存储介质。

异常组件的工作流程如图 23-1 所示。

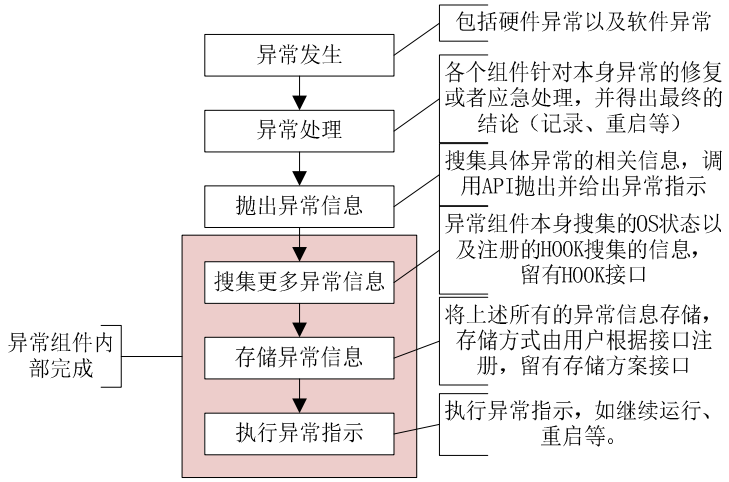


图 23-1 异常组件工作流程示意图

异常信息的再现，则是利用异常模块留出的调试接口从存储介质中读取指定条目的异常信息，当然各种条件查询只是后话了，因为异常信息条目既然可以读取的话，没有理由做不到条件搜索查询。从上述流程看，异常组件主要有四部分构成：1，留给用户使用的异常信息抛出接口；2，留给用户使用的异常信息搜集 HOOK（可选项）；3，异常存储方案注册接口；4，异常指示执行单元。下面逐一分析各个模块。

23.1.1 异常抛出接口

所谓的异常抛出接口，就是不论何时何地，只要你想，只要你愿意，都可以将你需要记录的异常信息输出到异常记录当中，都可以坚定不移的执行你的异常指示，而不在乎你到底是硬件异常还是软件异常，还是信口开河的异常，只要你调用这个接口即可，至于后续的工作，都由这个接口帮你搞定。但是前提是你得按照这个接口的标准扔出你要存储的记录，这个接口有两个参数：throwpara 和 dealresult。throwpara 即代表我们要抛出的异常信息标准。其结构体如下：

```
struct tagExpThrowPara
{
    bool_t   validflag;                 //true 表示该异常参数有效，否则无效
    char    *name;                      //异常信息解析器名字
    u8    *para;                        //异常信息地址
    u32    para_len;                    //信息长度
    u32    dealresult;                  //异常处理类型,参考_SYSEXP_DEAL_TYPE
};
```

各个成员在此就不赘述了，注释上写的明白。在此有个问题：为什么有名字这么个说法？很简单，当你抛出这个异常信息的时候，我们希望保存这个异常信息的解析器的名字，以便若干年后我们知道用那种解析器来解析这条记录，毕竟我们不能靠辨识你存储的字节流异常信息去猜你到底是什么类型的异常，这也就是为什么后来会有注册解析器这种说法；在该结构

体重有个成员 dealresult，该参数表示我们希望执行单元执行的操作：如仅仅记录、热重启、冷重启等指示。细心的朋友可能发现我们在该 API 的输出参数中还有个叫 dealresult，这个是怎么回事？不要惊慌，这个是我们的异常指示的返回结果，为什么还有这种说法？举个简单的例子，你拿了一大叠单子去找领导报销，试问，领导会照单全收么？总有那么几个不批，当然如果心情好，都会批。同样的，异常模块也是这个道理，不是你说重启就重启，不是你想记录就记录的，这要看 BOSS 的心情了，谁是 BOSS?那就是后文要谈论的注册 HOOK。OK，可以看看这个 ThrowExp 到底是怎么来处理扔出异常，处理流程如图 23-2 所示。

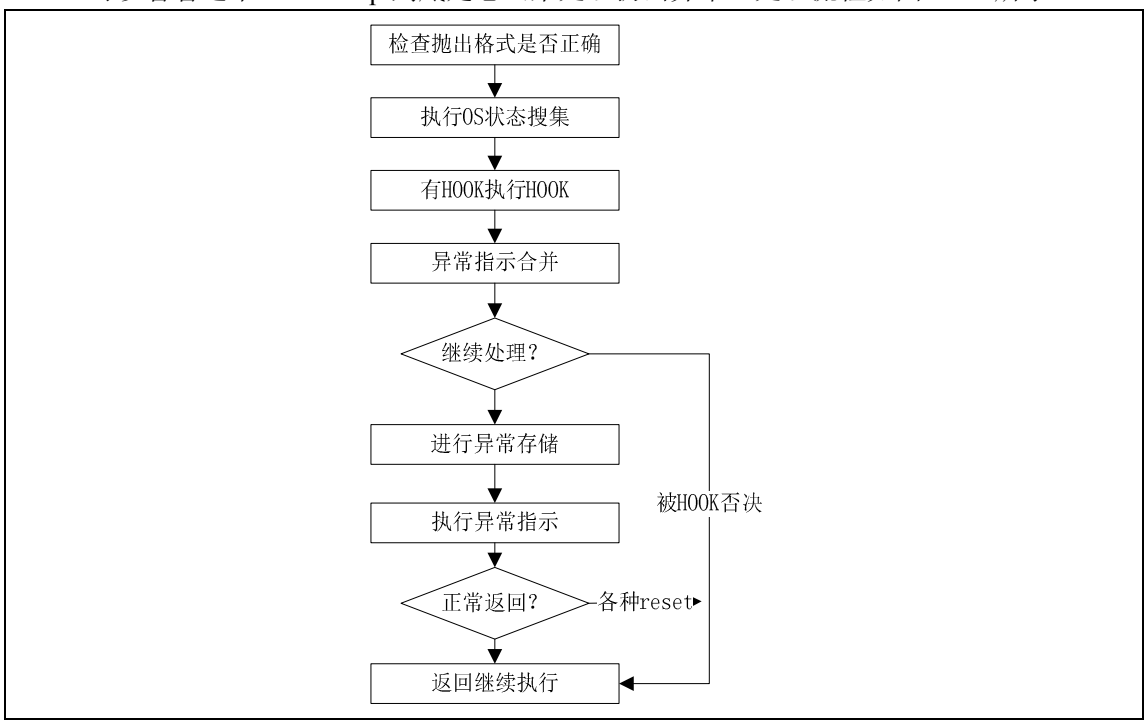


图 23-2 抛出异常执行流程示意图

23.1.2 注册 HOOK

既然前面提到了 HOOK 这个 BOSS，那么我们就来谈谈 HOOK 吧。HOOK 主要是针对 OS 的具体应用而设计的。对于某个项目，不是你随随便便的就允许你重启的，也不是你想记录就记录的，毕竟重启或者记录会影响设备的运行甚至功能。因此，当抛出异常时，需要知道当前能不能按照异常抛出者的意思进行，比方如果 HOOK 发现自己的各个功能都正常只是 OS 的某个组件出了点小小毛病，而设备正在执行某个重要任务并且设备的主要功能不受影响，那么 HOOK 能就会否决该异常。当然，如果 HOOK 肯定了发起者抛出的异常，那么此时 HOOK 会搜集当前设备的状态，然后一并交给存储单元进行存储，至于 HOOK 到底搜集的什么信息，完全取决于你们的项目经理或者系统工程师了。

23.1.3 异常信息存储

存储介质千差万别，存储手段千变万化，异常组件无法做到统一，但是可以提供统一的接口给异常组件使用。定义的存储接口如下所示，各个存储方案只要提供相应的功能接口即可，其他话无需多言。

```
struct tagExpRecordOperate
```

```
{
    fnExp_RecordScanModule   fnexprecordscan;   //开机存储区扫描，
    fnExp_RecordModule   fnexprecord;   //记录一条异常信息
    fnExp_RecordCleanModule   fnexprecordclear;   //清除所有异常信息
    fnExp_RecordCheckNumModule   fnexprecordchecknum;//获取存储异常信息的条目数
    fnExp_RecordCheckLenModule   fnexprecordchecklen;//获取指定条目的长度
    fnExp_RecordGetModule   fnexprecordget;   //获取指定条目的异常信息
}
```

只要填充好该数据结构，然后注册进去就 OK 了，至于到底是不是云存储以及异常信息是如何链接起来的，那就看你的了。以在 P1020 做的 NORFLASH 存储方案为例，其在 FLASH 中的存储方式如图 23-3 所示。在条目头中会有成员指示异常信息的长度等（非等长异常信息存储）。

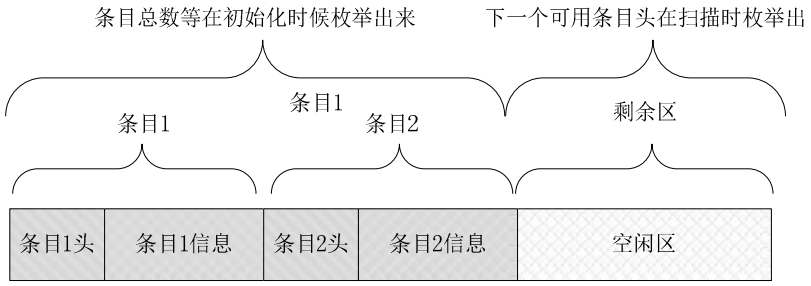


图 23-3P1020 异常存储示意图

23.1.4 异常指示执行单元

何为异常指示执行单元？简单的说就是执行异常指示：重启、正常返回。当异常信息搜集完毕之后，指示进一步动作。_EN_EXP_DEAL_TYPE_定义了异常指示类型如下：

```
enum _EN_EXP_DEAL_TYPE_
{
    EN_EXP_DEAL_IGNORE = 0,          //忽略该异常信息
    EN_EXP_DEAL_DEFAULT,             //按默认方式处理，抛出异常的时候不能使用该选项
    EN_EXP_DEAL_RECORD,              //需要记录该异常
    EN_EXP_DEAL_RESET,               //硬件重启，相当于上电启动
    EN_EXP_DEAL_REBOOT,              //跳转到运行模式选择
    EN_EXP_DEAL_RESTART,             //跳转到本次运行方式的最开始部分
};
```

其中，抛出异常时指定的异常类型只能大于等于 EN_EXP_DEAL_RECORD。该类型值会在异常抛出或者 HOOK 返回时指定。

23.1.5 移植

异常组件的接口都采用注册的方式，完全与硬件无关，可以放心大胆的使用，只需要按

照接口类型说明注册相关的接口即可。有以下需要注册的钩子函数。

23.1.6 HOOK 处理钩子函数

```
typedef    enum _EN_EXP_DEAL_TYPE_ (*fnExp_HookDealermodule)(
                    struct tagExpThrowPara *throwpara, ptu32_t *infoaddr,u32 *infolen)
```

头文件：

exp_api.h

参数：

throwpara, 异常抛出者抛出的异常参数；

infoaddr,存储异常信息的地址；

infolen,存储搜集信息长度。

返回值：

_SYSEXP_RESULT_TYPE，该处理结果会覆盖掉 BSP 的处理异常结果。

说明：

异常时调用的 APP 提供的异常处理函数，用来检测异常时系统应用程序的运行状态应用程序提供的钩子函数，应用程序可在此做一些善后工作，并可返回附加的异常信息，这些信息将与 BSP 收集的硬件和系统异常信息一起保存该钩子的返回结果将会和 BSP 的处理结果一起作为异常结构的判断依据(一般而言采用木桶原则，由严重程度大的来决定)。

23.1.7 HOOK 信息解析钩子函数

```
typedef    bool_t (*fnExp_HookInfoDecodermodule)(struct tagExpThrowPara
                                                 *throwpara,
                                                 ptu32_t infoaddr,
                                                 u32 infolen,u32 endian)
```

头文件：

exp_api.h

参数：

throwpara：常抛出者抛出的异常参数；

infoaddr：HOOK 信息存储地址；

infolen：HOOK 信息有效长度；

endian：HOOK 信息存储的大小端。

返回值：

true 成功　false 失败（没有注册等因素）

说明：

解析 HOOK 信息。

23.1.7.1　异常抛出信息解析器模型

```
typedef bool_t (*fnExp_ThrowinfoDecoderModule)(struct tagExpThrowPara *para,
                                               u32 endian);
```

头文件：
exp_api.h
参数：
para：异常抛出者抛出的异常参数；
endian：抛出异常信息存储的大小端。
返回值：
true 成功　false 失败（没有注册等因素）
说明：
解析 HOOK 信息。

23.1.7.2　记录一条异常信息

typedef　enum _EN_EXP_RECORDRESULT_ (*fnExp_RecordModule)(
　　struct tagExpRecordPara *recordpara);
头文件：
exp_api.h
参数：
recordpara：异常存储参数。
返回值：
参见_EN_EXP_RECORDRESULT_　。
说明：
记录一条异常信息。tagExpRecordPara 及_EN_EXP_RECORDRESULT_定义如下：

```
struct tagExpRecordPara
{
    u32   headinfolen;       //头信息长度
    ptu32_t   headinfoaddr;      //头信息地址
    u32   osstateinfolen;   //OS 状态信息长度
    ptu32_t   osstateinfoaddr; //OS 状态信息地址
    u32   hookinfolen;       //HOOK 信息长度
    ptu32_t   hookinfoaddr;      //HOOK 信息地址
    u32   throwinfolen;      //throw 信息长度
    ptu32_t   throwinfoaddr;     //throw 信息地址
};
//存储返回结果
enum _EN_EXP_RECORDRESULT_
{
    EN_EXP_DEAL_RECORD_SUCCESS,          //记录成功
    EN_EXP_DEAL_RECORD_NOSPACE,          //存储空间不足
    EN_EXP_DEAL_RECORD_HARDERR,          //存储硬件出错
    EN_EXP_DEAL_RECORD_PARAERR,          //存储参数出错
    EN_EXP_DEAL_RECORD_NOMETHOD,         //存储参数出错,无存储方案
};
```

23.1.7.3 清除所有异常信息

typedef bool_t (*fnExp_RecordCleanModule)(void);

头文件：

exp_api.h

参数：无。

返回值：

TRUE 成功， FALSE 失败。

说明：

清除所有的异常信息，清除异常信息存储区域。

23.1.7.4 查看存储异常信息条目

typedef bool_t (*fnExp_RecordCheckNumModule)(u32 *recordnum);

头文件：

exp_api.h

参数：

recordnum,存储的异常信息条目数。

返回值：

TRUE 成功， FALSE 失败。

说明：

查看一共存储了多少条异常信息。

23.1.7.5 查看指定条目异常信息长度

typedef bool_t (*fnExp_RecordCheckLenModule)(u32 assignedno, u32 *recordlen)

头文件：

exp_api.h

参数：

assignedno：指定的条目；

recordlen：该条目的长度。

返回值：

TRUE 成功， FALSE 失败。

说明：

查看指定条目异常信息的长度。

23.1.7.6 获取指定条目异常帧信息

typedef bool_t (*fnExp_RecordGetModule)(u32 assignedno,u32 buflen,u8 *buf,
struct tagExpRecordPara *recordpara);

头文件：

exp_api.h

参数：
assignedno：指定的条目；
buflen：缓冲区长度；
buf：用于存储获取指定条目的异常信息；
recordpara：异常信息存储时的参数，在此是对 buf 的各个部分的定义。
返回值：
TRUE 成功， FALSE 失败。
说明：
从存储介质中获取指定条目的异常帧信息。

23.1.7.7 扫描异常存储记录

```
typedef    void (*fnExp_RecordScanModule)(void);
```

头文件：
exp_api.h
参数：
无。
返回值：
无。
说明：
开机的时候扫描异常存储记录，获取关键信息方便以后存储。

23.2 API 说明

23.2.1 Exp_ModuleInit：异常组件初始化

```
ptu32_t    Exp_ModuleInit(ptu32_t para);
```

头文件：
exp_api.h
参数：
para：无意义。
返回值：
ptu32_t 暂时无定义。
说明：
该函数只是将异常相关的 shell 命令初始化。

23.2.2 Exp_Throw：抛出异常信息

```
bool_t    Exp_Throw(struct tagExpThrowPara *throwpara, u32 *dealresult);
```

头文件：
exp_api.h
参数：

throwpara：抛出的异常信息参数；
dealresult：该异常的最终处理结果。
返回值：
true，成功， false，失败(参数或者存储等未知原因)。
说明：
处理所有异常的入口，本函数可以在 bsp 中调用，也可以在系统中调用，此设计是为了异常模块处理的统一，版本之间差异缩减为最低，同时方便移植各个异常处理者之间相互独立，不相互干扰。

23.2.3 Exp_RegisterRecordOpt：注册异常信息处理方法

```
bool_t    Exp_RegisterRecordOpt(struct tagExpRecordOperate *opt);
```

头文件：
exp_api.h
参数：
opt：需要注册的异常信息处理方法。
返回值：
true，成功， false，失败(失败的话会使用 BSP 默认的处理方法)。
说明：
注册异常信息处理方法，理论上参数结构里面指定的处理方法都应该提供，否则的话会注册不成功。

23.2.4 Exp_UnRegisterRecordOpt：注销异常信息处理方法

```
bool_t Exp_UnRegisterRecordOpt(void);
```

头文件：
exp_api.h
参数：无。
返回值：
true 表示成功；false 表示失败(失败的话会使用 BSP 默认的存储方案)。
说明：
注销异常信息处理方法。

23.2.5 Exp_RegisterHook：注册 HOOK

```
bool_t Exp_RegisterHook(fnExp_HookDealermodule    fnappdealer,\
                        fnExp_HookInfoDecodermodule fnappdecoder);
```

头文件：
exp_api.h
参数：
fnappdealer：提供的异常处理器；
fnappdecoder：提供的异常信息解析器。

返回值：
true，成功， false，失败。
说明：
注册 APP 提供的异常处理 HOOK。fnExp_HookDealermodule 及 fnExp_HookInfoDecodermodule 均为函数指针，其原型定义详见 22.1.5。

23.2.6 Exp_UnRegisterHook：注销 HOOK

```
bool_t Exp_UnRegisterHook(void);
```
头文件：
exp_api.h
参数：无。
返回值：
true，成功， false，失败。
说明：注销 APP 提供的异常处理 HOOK 以及异常信息解析器。

23.2.7 Exp_RegisterThrowinfoDecoder：注册 EXP 解析器

```
bool_t Exp_RegisterThrowinfoDecoder(fnExp_ThrowinfoDecoderModule decoder,
                                    const char *name);
```
头文件：
exp_api.h
参数：
decoder：异常解析器；
name：异常解析器名字,至少保证是全局的且不会变的。
返回值：
true，成功注册， false，注册失败。
说明：
注册软件异常信息解析器。当对应的异常号已经被注册了的时候，会查找未被注册的异常号进行注册；无名或者已经存在对应命名的异常会导致注册失败澹(只是‘\0’也是无效的)；有注册失败的可能，因此注意检查返回结果。fnExp_ThrowinfoDecoderModule 为函数指针。其原型详见 22.1.5。

23.2.8 Exp_UnRegisterThrowinfoDecoder：注销 EXP 解析器

```
bool_t Exp_UnRegisterThrowinfoDecoder(char *name);
```
头文件：
exp_api.h
参数：
name：已经被注册的异常名字。
返回值：
true，成功注销， false，注销失败。

说明：
注销软件异常信息解析器。

第24章　定时器组件

24.1 定时器概述

DJYOS 的定时器模块提供如下功能：
1. 利用一个硬件定时器为应用程序提供高精度多个定时服务；
2. 管理硬件定时器，并为用户提供通用统一接口。

整个定时器模块框架分为三层，如图 24-1 所示。其中，硬件层负责具体芯片的驱动，提供通用而又基本的定时器芯片驱动功能；硬件接口层对上提供物理定时器的分配和操作，对下提供具体定时器芯片的注册接口，硬件接口层使应用不再依赖于具体定时器芯片；软件定时器层负责软件定时器的管理，让应用突破了物理定时器个数的限制。

需要注意的是这里提到的定时器芯片，可以是片上定时器也可以是外设定时器芯片。

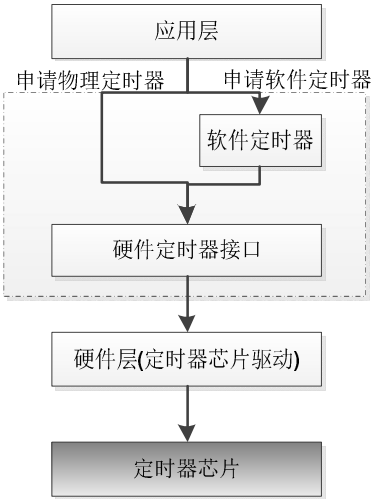


图 24-1 定时器模块架构示意图

24.1.1　硬件接口层

硬件接口层的职责是对上层隐藏具体硬件属性，其本身是应用和硬件之间的桥梁，其功能的实现依赖于硬件驱动。从该层申请的定时器都是物理定时器，因此操作的都是对应的物理定时器。

该层向上提供对硬件定时器的操作接口：申请定时器、释放定时器、设置周期、设置中断、暂停和使能定时器等功能。向下提供硬件驱动注册定时器芯片的接口：注册定时器芯片、注销定时器芯片。硬件接口层定义了时钟芯片参数结构体 tagTimerChip 如下：

```
struct tagTimerChip
{
        char    *chipname;      //chip 名字，必须为静态型
        fnTimerHardAlloc    timerhardalloc;     //分配定时器
```

```
        fnTimerHardFree    timerhardfree;       //释放定时器
        fnTimerHardCtrl    timerhardctrl ;      //看定时器定时时间是否到
};
```

通过 tagTimerChip 中三个钩子函数 fnTimerHardAlloc、fnTimerHardFree、fnTimerHardCtrl 实现对具体时钟芯片的间接操作，这三个钩子函数是由硬件层时钟芯片驱动中实现。

24.1.2 软件定时器

如何做软件定时器？共享定时器。既然连我们的手机信号也是分频的，为什么我们的定时器不能呢？一个定时器如果仅仅指定了一个定时周期，是不是有点太浪费了？毕竟在这个定时期限内它什么也不做，仅仅是计数，为什么我们不试着让她总是在工作状态呢？依次触发，对，这就是我们的工作模式，多个软件定时器依次触发同一个硬件定时器。当我们把应用程序的所有定时需求做一定的整合，然后设定我们的定时器按需定时触发，岂不是 perfect 的感觉

如何达到共享定时器的目的？其实很简单，只要将我们需要定时的时间从近到远（近，表示离当前时间很近）依次排列，那么取最近的先定时依次，当第一次定时到的时候再取第二近的时间再次定时。定时时间到的定时器（软件定时器）重新计算下次定时器时间然后继续排队。这样理论上实现了无限多的定时器需求。整个架构如图 24-2 所示。

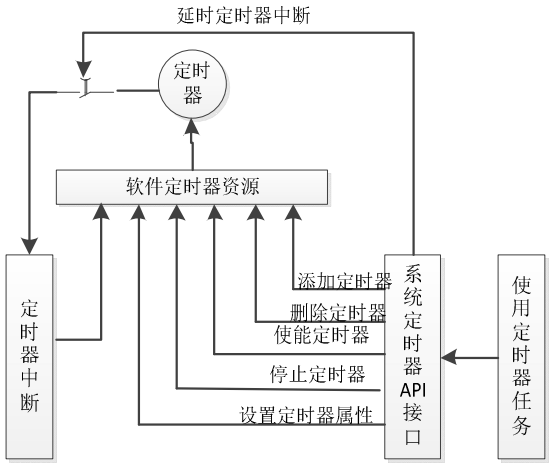


图 24-2 软件定时器框架图

首先声明：为了让整个定时模块更加准确，特别使用一个 64 位定时器。该定时器不会触发中断，也不会被暂停，永远不停的滴答下去，它只需要提供给我们一个接口即可，即已经走了多长时间。因此，该 64 位定时器就像一个挂钟，当然，它是一个周期几乎无限大的挂钟（64 位应该可以定时很多年）。

所有软件定时器的定时时限都是 64 位的，当我们用作定时的 32 位定时器定时时间到了之后，就会在其 ISR 中依次检查软件定时器队列，看其定时时限是否比当前 64 位定时器时间小，如果小，则证明超时，依次处理所有软件定时器。

当处理完所有超时软件定时器之后，那么排在队列头的软件定时器一定是未超时的，如果该软件定时器不是暂停状态，那么就用其定时时限减去 64 位定时器的走时时间作为 32 位定时器的定时周期，然后启动 32 位定时器；否则证明所有的软件定时器都出于暂停状态，那么只需要维持 32 位定时器为暂停状态即可。具体流程如图 24-3 所示。

版权所有，转载必须注明出处，注明：来自 www.djyos.com 字样。

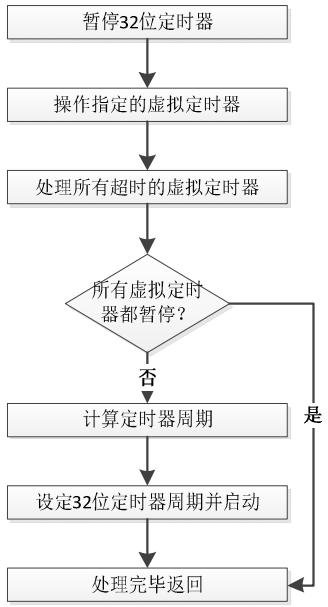


图 24-3 软件定时器工作流程示意图

24.1.3 移植

要想使用 Djyos 的定时器组件各功能，用户必须要针对各自使用平台的定时芯片编写底层驱动，这也是移植唯一要做的事情。在硬件层需要完成以下 4 个函数：Timer_ModuleInit(Timer 的初始化)以及 23.1.1 节中提及的三个钩子函数。

24.1.3.1 定时器初始化

void Timer_ModuleInit(void);
头文件：
cpu_peri_timer.h
参数：
无。
返回值：
无。
说明：
该函数功能为 Timer 初始化，需要使能 Timer 时钟，配置时钟频率等，同时将时钟芯片结构体成员赋值，最后调用 TimerHard_RegisterChip 将该 Timer 注册到系统时钟芯片中。

24.1.3.2 分配定时器

typedef ptu32_t (*fnTimerHardAlloc)(u32 cycle,fnTimerIsr timerisr)
头文件：
timer_hard.h
参数：

cycle：指定分配定时器的定时周期，该属性可使用API函数进行更改设定（单位：微秒）。
timerisr：分配的定时器的中断服务函数，中断中调用。

返回值：

NULL 分配不成功，否则返回定时器句柄，该句柄的结构由定时器芯片自己定义。

说明：

硬件定时器分配，一般而言刚分配的定时器，其计时停止，中断禁止）依赖芯片驱动实现刚开始分配的定时器应该是各种属性都关闭的，因此属性必须自己重新设置。默认：停止计数，异步中断，reload,中断禁止。该钩子函数会被 TimerHard_Alloc 调用。
fnTimerIsr 为函数指针，其原型为：
typedef u32 (*fnTimerIsr)(ufast_t irq_no);
其中 irq_no 为定时器对应的中断号。

24.1.3.3 释放定时器

typedef bool_t (*fnTimerHardFree)(ptu32_t timerhandle)

头文件：

timer_hard.h

参数：

timerhandle，待释放定时器句柄。

返回值：

true 成功 false 失败。

说明：

硬件定时器释放，依赖芯片驱动实现。

24.1.3.4 操作定时器

typedef bool_t (*fnTimerHardCtrl)(ptu32_t timerhandle,
enum _ENUM_TIMER_CTRL_TYPE_ ctrlcmd,
ptu32_t inoutpara)

头文件：

timer_hard.h

参数：

timerhandle：待操作的定时器句柄；
ctrlcmd： 操作命令；
inoutpara：输入输出参数，根据不同的情况而定。

返回值：

true 操作成功 ； false 操作失败。

说明：

Timer 的操作码包含以下：

```
enum _ENUM_TIMER_CTRL_TYPE_
{
    EN_TIMER_STARTCOUNT = 0,        //使能计数
    EN_TIMER_PAUSECOUNT,            //停止计数
```

```
    EN_TIMER_SETCYCLE,              //设置周期
    EN_TIMER_SETRELOAD,             //设置定时器为单次触发，还是自动触发
    EN_TIMER_ENINT,                 //中断使能
    EN_TIMER_DISINT,                //中断禁止
    EN_TIMER_SETINTPRO,             //中断属性设置
    EN_TIMER_GETTIME,               //获取计时时间
    EN_TIMER_GETCYCLE,              //获取定时周期
    EN_TIMER_GETID,                 //获取定时器 ID，高 16 位为 intID，低 16 为 timerID
    EN_TIMER_GETSTATE,              //获取定时器状态
};
```

底层芯片驱动需要在该钩子函数中分别实现上述各个操作码对应的时钟芯片的操作功能。

24.1.4 使用实例

以 stm32f103 为例，采用 stm32f103 片上时钟芯片，在硬件层(时钟芯片驱动,cpu_peri_timer.c)中实现了三个钩子函数__STM32Timer_Alloc、__STM32Timer_Alloc、__STM32Timer_Ctrl，并在 Timer_ModuleInit 初始化函数中注册时钟芯片，部分代码如下：

```
void Timer_ModuleInit(void)
{
    struct tagTimerChip    STM32timer;
    STM32timer.chipname = "STM32TIMER";
    STM32timer.timerhardalloc = __STM32Timer_Alloc;
    STM32timer.timerhardfree = __STM32Timer_Free;
    STM32timer.timerhardctrl = __STM32Timer_Ctrl;
    TimerHard_RegisterChip(&STM32timer);
    return ;
}
```

详细代码请查看 Djyos 相关工程。

接下来就可以使用 Djyos 的定时器功能，可以基于一个硬件定时器使用多个软件定时器。

Step1：在 critical 函数中调用 Timer_ModuleInit 对定时器进行初始化并将其加载到系统定时器模块中；

Step2：在组件初始化阶段调用 TimerSoft_ModuleInit 初始化软件定时器；

Step3：调用 TimerSoft_Create 函数创建一个软件定时器；

Step4：调用 TimerSoft_Ctrl 函数对创建的软件定时器进行相应的操作(若使用默认设备，此步可不需要)；

Step5：定时时间到，处理相应的中断函数；

Step6：调用 TimerSoft_Delete 删除某个软件定时器。

24.2 定时器 API

24.2.1 TimerHard_RegisterChip：注册定时器芯片

bool_t TimerHard_RegisterChip(struct tagTimerChip *timerchip)

头文件：
timer_hard.h
参数：
timerchip：定时器芯片结构体指针。
返回值：
true 成功 false 失败。
说明：
注册定时器芯片到系统定时器模块。该函数

24.2.2 TimerHard_UnRegisterChip：注销定时器芯片

bool_t TimerHard_UnRegisterChip(void)
头文件：
timer_hard.h
参数：无。
返回值：
true 成功 false 失败。
说明：
定时器芯片注销，目前而言是没有用，主要是 register 之后必然有 unregister。

24.2.3 TimerHard_Alloc：硬件定时器分配

ptu32_t TimerHard_Alloc(u32 cycle, fnTimerIsr timerisr)
头文件：
timer_hard.h
参数：
cycle：指定分配定时器的定时周期，该属性可以使用API函数进行更改设定（单位为微秒）；
timerisr：分配的定时器的服务函数，中断中调用。
返回值：
NULL 分配不成功，否则返回定时器句柄，该句柄的结构由定时器芯片自己定义。
说明：
硬件定时器分配，一般而言刚分配的定时器，其计时停止，中断禁止）依赖芯片驱动实现刚开始分配的定时器应该是各种属性都关闭的，因此属性必须自己重新设置。默认：停止计数，异步中断，reload,中断禁止。

24.2.4 TimerHard_Free：硬件定时器释放

bool_t TimerHard_Free(ptu32_t timerhandle);
头文件：
timer_hard.h
参数：

timerhandle，待释放定时器句柄。
返回值：
true 成功；false 失败。
说明：
硬件定时器释放，依赖芯片驱动实现。

24.2.5 TimerHard_Ctrl：操作硬件定时器

```
bool_t TimerHard_Ctrl(ptu32_t timerhandle,
                      enum _ENUM_TIMER_CTRL_TYPE_ ctrlcmd,
                      ptu32_t inoutpara)
```

头文件：
timer_hard.h
参数：
timerhandle：待操作的定时器句柄；
ctrlcmd： 操作命令；
inoutpara：输入输出参数，根据不同的情况而定。
返回值：
true 操作成功 ；false 操作失败。
说明：
ctrlcmd 对应的 inoutpara 的属性定义说明如下：
EN_TIMER_STARTCOUNT：使能计数，inoutpara 无意义；
EN_TIMER_PAUSECOUNT：停止计数，inoutpara 无意义；
EN_TIMER_SETCYCLE：设置周期，inoutpara 为 u32,待设置的周期（微秒）；
EN_TIMER_SETRELOAD：reload 模式 or not！，inoutpara 为 bool_t,true 代表 reload；
EN_TIMER_ENINT：中断使能，inoutpara 无意义；
EN_TIMER_DISINT：中断禁止，inoutpara 无意义；
EN_TIMER_SETINTPRO：中断属性设置，inoutpara 为 bool_t,true 代表实时信号；
EN_TIMER_GETTIME：获取计时时间，inoutpara 为 u32 *,单位为微秒；
EN_TIMER_GETCYCLE：获取定时周期，inoutpara 为 u32 *,单位为微秒；
EN_TIMER_GETID：获取定时器 ID，inoutpara 为 u32 *，高 16 位为 intID，低 16 为 timerID；
EN_TIMER_GETSTATE：获取定时器状态，inoutpara 为 u32 *。

24.2.6 TimerSoft_ModuleInit：软件定时器初始化

```
ptu32_t  TimerSoft_ModuleInit(ptu32_t para)
```

头文件：
timer_soft.h
参数：
para：无意义。
返回值：
true 操作成功 ；false 操作失败。
说明：

软件定时初始化。

24.2.7 TimerSoft_Create：创建一个软件定时器(内核)

```
TIMERSOFT_HANDLE    TimerSoft_Create(char *name,
                                     u32 timercycle,
                                     fnTimerSoftAlarmHandler fntimerisr)
```

头文件：
timer_soft.h
参数：
name：定时器名字；
timercycle：定时器周期；
fntimerisr：定时器定时时间到执行 HOOk，中断中可能被调用。
返回值：
NULL 分配失败　否则返回分配到的定时器句柄。
说明：
创建一个定时器，创建的定时器默认的 reload 模式，如果需要手动的话，那么创建之后自己设置；创建的定时器还是处于 pause 状态，需要手动开启该定时器。

24.2.8 TimerSoft_Create_R：创建一个软件定时器

```
TIMERSOFT_HANDLE    TimerSoft_Create_R(char *name,
                                       u32 timercycle,
                                       fnTimerSoftAlarmHandler fntimerhandler,
                                       TIMERSOFT_RSCTYPE *timerspace);
```

头文件：
timer_soft.h
参数：
name：定时器名字；
timercycle：定时器周期；
timerflag：　定时器标记，主要指 autoload 还是 master set counter 值；
fntimerhandler：定时器定时时间到执行 HOOk，中断中可能被调用；
timersoft：提供的虚拟定时器的资源空间。
返回值：
NULL 分配失败　否则返回分配到的定时器句柄。
说明：
创建一个定时器，创建的定时器默认的 reload 模式，如果需要手动的话，那么创建之后自己设置；创建的定时器还是处于 pause 状态，需要手动开启该定时器。

24.2.9 TimerSoft_Delete：删除一个软件定时器(内核)

```
bool_t TimerSoft_Delete(TIMERSOFT_HANDLE timerhandle)
```

头文件：
timer_soft.h
参数：
timerhandle：待删除的定时器句柄。
返回值：
true 成功；false 失败。
说明：
删除一个软件定时器。

24.2.10 TimerSoft_Delete_R：删除一个软件定时器

TIMERSOFT_HANDLE TimerSoft_Delete_R(TIMERSOFT_HANDLE timerhandle)
头文件：
timer_soft.h
参数：
timerhandle：待删除的定时器句柄。
返回值：
true 成功；false 失败。
说明：
删除一个软件定时器。

24.2.11 TimerSoft_Ctrl：软件定时器控制

```
bool_t   TimerSoft_Ctrl(TIMERSOFT_HANDLE timerhandle,
                        u32 optcmd,
                        ptu32_t inoutpara);
```
头文件：
timer_soft.h
参数：
timerhandle：待操作的定时器句柄；
optcmd： 操作命令；
inoutpara：输入输出参数，根据不同的情况而定。
返回值：
true 操作成功 ；false 操作失败。
说明：
optcmd 对应的 inoutpara 的属性定义说明如下：
EN_TIMER_STARTCOUNT：使能计数，inoutpara 无意义；
EN_TIMER_PAUSECOUNT：停止计数，inoutpara 无意义；
EN_TIMER_SETCYCLE：设置周期，inoutpara 为 u32,待设置的周期（微秒）；
EN_TIMER_SETRELOAD：reload 模式 or not！，inoutpara 为 bool_t,true 代表 reload；
EN_TIMER_GETTIME：获取计时时间，inoutpara 为 u32 *,单位为微秒；
其他 CMD 不支持。