ARM 嵌入式系统 C 语言编程

姜 换 新
(惠普中国软件研发中心　上海 201206)

摘　要　无操作系统支持的嵌入式系统软件,包括系统引导(BOOT)、驱动程序、动态内存管理、I/O、通信以及应用软件等方面。本文详细介绍了嵌入式平台上用 C 语言编写系统软件和应用软件的方法。虽然是针对 ARM 平台介绍的,但基本经验和算法也适合于其他嵌入式平台的软件设计。

关键词　嵌入式系统　软件　C 语言　ARM

PROGRAMMING C ON ARM EMBEDDED PLATFORM

Jiang Huanxin
(China Software Solutions Center , Hewlett - Packard Company , Shanghai 201206)

Abstract　Programming C on ARM embedded platform is a complicated project. Modules including system boot ,drivers ,dynamic memory management ,I/O interface ,communications and applications should be considered carefully. With an excellent experience on ARM embedded system ,the author gives a detailed description in this paper on the methods and algorithms about programming ARM. Though ARM is the only discussed item ,this paper is useful for programming on any other embedded platforms.

Keywords　Embedded system　Software　C programming language　ARM

1　引　言

无操作系统支持的嵌入式软件包括系统引导(BOOT)、外围驱动程序、存储管理、系统 I/O、通信、应用程序等方面,需要结合采用汇编语言(约占 10 %)和 C 语言(约占 90 %)。本文结合作者实践,详细介绍 ARM 嵌入式平台的 C 编程方法。考虑到通信软件涉及范围较大,本文不进行讨论。

2　系统引导与 main 函数

通常 C 语言是从 main 函数开始的。main 函数的原型是:

```
int main(int argc ,char * * argv)
```

其中 argc 是参数的个数,argv 是指向各参数的指针的数组。main 函数由操作系统内核启动,操作系统内核完成函数所需的变量初始化工作,并在调用结束后检查 main 函数的返回值,若返回值为 0,表明程序运行正常,否则表明程序运行出错。在嵌入式系统中,由于没有操作系统内核存在,对 main 函数的初始化工作只能由系统引导(BOOT)模块完成。

系统引导(BOOT)部分完成系统初始化工作,用汇编语言实现。它的工作包括硬件初始化、栈寄存器的设置、全局变量的初始化或清 0、RAM 中运行的模块的加载、堆参数的初始化等。完成这些工作后,再把控制权交给 C 的 main 函数。显然,对嵌入式系统的 main 而言,argc 和 argv 这两个参数及返回值都是没有意义的(如果返回,表明系统出现严重错误)。另外,为了避免产生混淆,我们还必须给 main 函数另外取一个名字,比如 Main。否则,编译器将会给 main 函数生成一大堆初始化代码,导致 C 程序的主入口与系统引导模块的接口错误。

系统引导模块完成各种初始化工作后,用一条跳转指令进入 C 的主入口 Main,控制权从此移交给了 C 应用程序。

3　存储管理

存储管理是一个复杂的课题。从广义的角度来说,磁盘文件系统、内存、片内高速 Cache 等都属于这个范畴。嵌入式系统中,较有意义的是内存的动态分配与释放及 Flash 存储器管理两方面。本文要介绍的是我们在嵌入式系统中实现的动态内存管理。

C 语言中动态内存分配与释放主要由 malloc 和 free 两个标准库函数实现。malloc 从系统空闲内存中分配合适的内存块,free 函数完成内存块的回收。这两个函数一般需要操作系统内核的支持,在 ARM 裸平台上,不能直接调用。为此,我们编写了 m_alloc 和 m_free 两个函数,实现动态存储管理的功能。

典型应用程序内存映象分成代码区、数据区和栈区,三个区从低地址到高地址依次分布。代码区从最低地址开始,栈区

收稿日期:2002 - 05 - 10。姜换新,硕士,主研领域:数字通信、嵌入式及网络编程。

则占据最高地址。代码区和数据区可以相连,也可以分开。嵌入式系统里,代码区位于只读存储器(如 Flash)中,数据区和栈区则位于 RAM 中,因此代码区和数据区一般并不相连[1)]。数据区和栈区是分开的,它们之间的空隙称作堆。

堆作为一个连续的可利用空间,是系统的初始可分配块。每次应用程序申请内存,m_alloc 便从堆中分割出一块(从低地址开始)给它。随着申请次数的增加,原来一个完整的内存块便被分割为多个独立的块分配给应用程序。由于内存释放的先后顺序是随机的,因此一定时间后,系统中将存在多个互不相连的内存块。这就使得整个内存区呈现出占用块和空闲块犬牙交错的状态,如图 1 所示。图中灰色部分表示内存被占用,白色部分表示未被占用。

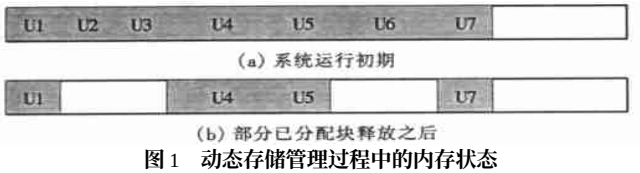

图 1　动态存储管理过程中的内存状态

为了进行内存动态管理,需要维护两张全局表,一张是可利用空间表(avail_list),管理空闲内存块的信息,另一张是已分配空间表(used_list),管理占用内存块。这两张表都用双向循环链表实现。随着系统的运行,可利用空间表中往往会有多个空闲块存在,究竟分配哪一块呢?文[1]介绍了三种不同的分配策略,即首次拟合法、最佳拟合法和最差拟合法,各有优缺点。笔者实现的是首次拟合法。

可利用空间表和已分配空间表采用相同的"表元"数据结构,定义如下:

```
struct mblock{
    struct mblock  *next;
    struct mblock  *prev;
    size_t size;
    char  *space;
};
```

在系统初始化时,整个可分配内存块是一个连续的存储区,可利用空间表的元素只有一个。m_alloc 函数每次分配内存时,先检查 size(m_alloc 的参数)是否合法(如是否超出堆的范围),若合法,再将其与 32 - bit 字对齐,然后从 avail_list 中搜索合适的内存块,并将其分配给应用程序。如果内存块的大小比 size 大得较多,则对内存块进行分裂,低地址的一块分配给应用程序,高地址的一块仍然放入 avail_list 中。如果搜索不到合适的空闲块,m_alloc 返回(void *)0。

m_free 函数释放内存时,根据参数 addr 给定的地址,在 used_list 中搜索相应的表元,找到后,将它标识的内存块释放,并插入到 avail_list 中去。然后,在 avail_list 中检查是否有相邻的空闲块,并进行空闲块的合并。有三种不同的情况要分别处理:(1)左相邻:相邻块在当前释放块的低地址端。(2)右相邻:相邻块在当前释放块的高地址端。(3)左右相邻:当前释放块的低地址端和高地址端都有相邻块。

在具体的分配算法上,文[1]介绍了边界标识法和伙伴系统。前者直接将链表管理信息插入到内存块的前端和后端,回收算法效率较高,但如果应用程序改写了超出它所申请范围的内存区,则会破坏整个数据结构,鲁棒性差一些。后者是笔者采用的算法之一,但使用下来发现它没有本文所描述的算法的效率高,且容易形成很多内存碎片。

4　LCD 终端(系统 I/O)

LCD 终端软件是系统 I/O 范畴的重要内容,主要包括 LCD 字符显示(英文 8 ×16 点阵,汉字 16 ×16 点阵),LCD 绘图(点、线、圆、面、位图、图形旋转等)。320 ×240 象素的 LCD 显示器,能显示 15 行 ×40 列英文字符,或 15 行 ×20 列汉字字符,并基本实现有较好分辨率的图形/图像的显示。

LCD 显示的最基本程序是画点程序,其原型如下:

```
void LCDPixel (int x, int y,char color)
```

其中,x 和 y 是点的坐标,坐标原点在左上角,color 是点的灰度。

字符和位图的显示利用了点阵方式。线、圆和面则利用相应的算法实现。图形旋转需要使用坐标变换函数。

这里要详细介绍的是把 LCD 作为(英文)字符型终端时的相应软件设计。把 LCD 作为字符型终端时,一个关键点是定义好光标:

```
static unsigned CurrentLine,CurrentColumn
```

这里 CurrentLine 和 CurrentColumn 分别定义了光标的横坐标和纵坐标(坐标原点在左上角),取值范围分别是(0～39)和(0～14),对应于横行 40 个字符和纵列 15 个字符。

定义好光标后,每次向屏幕输出字符时,总是从光标处开始,这样就保证了输出的有序性和连贯性。

向屏幕输出字符串的基本函数是 Printf,其原型如下:

```
void Printf(const char *fmt, ···)
```

这是一个可变参数函数,功能上与 printf 标准库函数完全相似。为了实现可变参数的处理,要使用 stdarg.h 中定义一些宏。Printf 分析每个格式字符,并对各转义字符(如\n,\t,\b,\r,\v 等)进行相应处理。在屏幕的合适位置打印格式化后的字符串。Printf 还调用一个滚屏函数 ScreenScroll,当光标位于末行时让屏幕向上滚动若干行。

Printf 函数不仅为 LCD 作为字符型终端提供了一个好的手段,同时也为程序的调试提供了便利。我们可以在程序可能出错的地方用 Printf 函数打印一些信息,这为我们对程序的跟踪提供了相当大的方便。Printf 函数在嵌入式系统编程中使用是十分明显的。

5　驱动程序设计

驱动程序包括最底层的中断处理程序设计和建立在其上的驱动程序设计两个部分,其实现与具体的外围设备有关,复杂性较大。这里只介绍用 C 语言设计驱动程序时需要注意的一些方面。

外围硬件设备一般通过中断与 CPU 进行通信。中断是一种外部异步事件。在处理与中断相关的变量时,需要小心。通常,编译器的优化选项打开后,对变量的操作,将尽量安排在寄

(下转第 53 页)

1) 使用 ARM720T 内核中的 MMU(内存管理单元),可以把各物理上分开的内存块映射成逻辑上连续的内存空间。这样,从编程角度看,内存布局是连续的。这实际上是虚拟存储的概念。

```
Energy3dlibInitialize();
MlfEnergy3d(pr_Height,pr_Width,pr_Space,pr_Intensity);
mxDestroyArray(pr_Length);
mxDestroyArray(pr_Space);
mxDestroyArray(pr_Intensity);……}
```

其中 CSpot 是为了方便分析而写的描述微光斑各个属性的 C++ 类。Energy3dlibTerminate() 在 CMyApp 类的重载函数 ExitInstance() 中调用。

要特别注意:1) 传递的矩阵大小必须相匹配,要不然运行时会出错。2) 注意数据数组下标,MATLAB 数组是按列存储的,而 VC 中是按行存储的。

2) 微光斑能量分布图

上述代码绘制的微光斑某个截面和三维能量分布图见图 3、图 4。

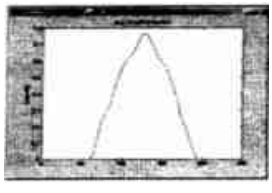

图 3 微光斑截面能量分布图

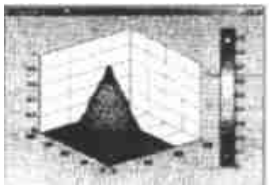

图 4 微光斑三维能量分布图

5 结 论

由微光斑分析软件的实例可以看出本文提出的用 Microsoft Visual C++ 编写光学测量分析软件的主要部分,而经常涉及到的光能量二维及三维分布图则由 VC 调用 MATLAB 编写、编译生成的 C/C++ Shared Library(DLL)绘制的方法不仅可以完全脱离 MATLAB 运行环境,可以根据需要传递参数,而且确实非常简单方便、可视性强,从而也有利于光学分析。同时这种方法也可以应用于光学以外的其它领域,提高编程速度,缩短开发周期。

参 考 文 献

[1] MathWorks 公司主页,http://www.mathworks.com.

[2] 潘卫明、赵敏、张进芳,"VC++ 下如何利用 MATLAB 工具箱进行数字信号处理",《计算机与信息技术》,2000,9.

[3] 齐波、董能力,"通过 VC++ 调用 MATLAB",《计算机世界》,2000,29.

[4] 亓波、董能力,"实现 Visual C++ 6.0 与 MATLAB 的混合编程",《电脑编程技巧与维护》.2000,12:62.

[5] 薛定宇,科学运算语言 MATLAB5.3 程序设计与应用,北京:清华大学出版社,2000.

(上接第 16 页)存器中。中断服务程序常常通过改变一些全局变量来通知应用程序某个外部事件已经发生,这些全局变量是不应该被优化的。解决的办法是在声明变量时加上 volatile 修饰符,以通知编译器这是一个可能被异步事件改变的量。这个问题看似简单,但如果不注意,实际运行时,程序将出现错误,且调试时很难定位故障。

运行效率是设计驱动程序的另外一个问题。中断比较频繁的外设,其中断处理程序的速度对整个系统的性能影响是很大的。这些模块应该直接用汇编语言编写,并尽可能优化算法。

C 语言的编写风格也要为效率考虑。例如对数组元素的操作"Array[idx/4] = &~1;"就不如改为"Array[idx >>2] = &~1;"这里。">>"是移位运算,有相应的机器指令,而"/"是除法运算,算法要复杂得多。当然,先进的编译器一般能优化这类的语句,但不能保证所有编译器都有此功能。

6 应用程序设计

嵌入式裸平台上的应用程序设计也有与 PC 机上的应用程序设计不同的地方,需要格外注意。

首先,凡是由需操作系统支持的标准库函数均不能使用,除非自己编写(如 m_alloc 和 m_free)。

其次,由于内存资源有限,栈容量有限且不能自动扩展,使用时要格外小心。常常能见到这样的局部变量的应用:

```
int buf[2048]
```

其目的是要申请一个 2048word(8192Byte)的缓冲区,对于嵌入式系统来说,开销过于庞大。同时,栈空间中用于嵌套调用的开销是不可见的,在嵌套层数较多时尤其如此。一下子申请这么大的栈空间,对系统是一个大的挑战,搞不好系统会崩溃。我们可以采用类似于下面的替代方式:

```
int *buf;
if((buf = m_alloc(2048 *sizeof(int))) == NULL)
    return ERROR;
/*other processing */
m_free(buf);
……
```

堆的操作比栈更灵活,也更好控制。如果 m_alloc 调用成功,它将返回分配的内存块的地址,否则返回 0。如果返回 0,表明系统内存已经所剩不多,这时程序员可以采取别的措施来解决问题,而不至于使系统崩溃。

第三,同样的道理,坚决避免使用递归函数。

第四,使用 m_alloc 函数时要注意两点:一是要检查返回值是否为 0;二是要适时调用 m_free 函数释放内存。前者可以避免系统陷入不必要的崩溃,后者可以防止出现内存泄漏。

第五,对于编译器给出的警告信息不要忽略。有良好的编程习惯的程序员是不会放过任何一个警告的。实际上,警告常常隐含着严重的逻辑错误。即使是无关痛痒的警告,比如变量声明了却没有使用的警告,也要予以重视,因为这类警告多了之后,会把一些有意义的警告掩盖掉,最终导致错误出现。

第六,对一些速度要求较高的关键模块,采用汇编语言实现。

7 结束语

目前,嵌入式应用日渐普及,嵌入式软件越来越受到关注。本文仅从作者的实践出发,谈了一些粗浅经验,不当之处请读者指正。

参 考 文 献

[1] 严蔚敏、吴伟民,1992,数据结构(第二版),清华大学出版社.

[2] W. Richard Stevens, 1992, Advanced Programming in the UNIX Environment, Addison Wesly Publishing Company.

[3] ARM7TDMI Data Sheet(ARM DDI 0029E).