

嵌入式图形系统 $\mu\text{C}/\text{GUI}$ 的动态内存分配的研究杨少春¹, 王作鹏²

(1. 武汉职业技术学院 电子信息工程系, 湖北 武汉 430074;

2. 武汉理工大学 自动化学院, 湖北 武汉 430074)

摘要:在进行嵌入式系统的开发的过程中,都必须设计出易于操作的人机互动操作界面。完成此工作会占用开发者过多的时间,甚至以失败告终。文中为开发者论述了一款新系统,即嵌入式图形系统 $\mu\text{C}/\text{GUI}$ 。它不仅具有很好的移植性,而且还具有灵活、简洁、高效、可方便剪裁、程序占用空间小等优点,因此在嵌入式系统行业中得到了广泛的应用。为了提高 $\mu\text{C}/\text{GUI}$ 的实时性,在研究了此系统的运行机制之后,设计了一个动态内存分配的方案。此方案在实验中取得了良好的结果,是非常有效的。

关键词:动态内存分配;嵌入式系统;碎片整理

中图分类号:TP311

文献标识码:A

文章编号:1673-629X(2006)08-0091-03

Research on Dynamic Memory Distribution of Embedded $\mu\text{C}/\text{GUI}$ YANG Shao-chun¹, WANG Zuo-peng²

(1. Department of Electronic Information, Wuhan Institute of Technology, Wuhan 430074, China;

2. Automation School of Wuhan University of Technology, Wuhan 430074, China)

Abstract: When developing the embedded system, it is necessary to research the operation interface. But finished the work will take most time, even failed at last. So introduces the system called embedded graph system $\mu\text{C}/\text{GUI}$. It is transplantable, flexible, simple and efficient, which can be clipped easily and applied widely. To improve the real-time performance of $\mu\text{C}/\text{GUI}$, researches into the operating principium of this system, then designs a scheme for dynamic memory distribution, which proves to be very practical and efficient in the experiments.

Key words: dynamic memory distribution; embedded system; fragment modulation

0 引言

随着嵌入式操作系统和高端微控制器的发展,人们对显示效果的要求也越来越高,不仅期望显示简单的字母和汉字,还期望能显示图形和彩色图片,甚至是视频和动画。为实现这个目的,系统在运行过程中不可避免会占用微控制器的大量系统资源,降低了嵌入式操作系统的实时性。而文中设计的 $\mu\text{C}/\text{GUI}$ 内存分配方案不但很好地解决了上述矛盾,而且可以运行其窗口体系建立友好的人机界面,这是由于以动态的形式创建任务,完成任务之后可以迅速释放占用的内存,从而提高了操作系统的实时性能。

1 动态内存分配的含义及特点

动态内存分配是相对于静态内存分配而言的。静态内存分配是指分配固定大小的内存分配方法,但是这种内存分配的方法存在比较严重的缺陷,例如:在大多数情况下会浪费大量的内存空间;在少数情况下,如当你定义的

数组不够大时,可能引起下标越界错误,甚至会导致严重后果^[1]。那么有没有其它的方法来解决这样的问题呢?有,那就是动态内存分配。

所谓动态内存分配就是指在程序执行的过程中动态地分配或者回收存储空间的分配内存的方法。动态内存分配不像数组等静态内存分配方法那样需要预先分配存储空间,而是由系统根据程序的需要即时分配,且分配的大小就是程序要求的大小。从以上动、静态内存分配比较可以知道动态内存分配相对于静态内存分配的特点:

- (1) 不需要预先分配存储空间;
- (2) 分配的空间可以根据程序的需要扩大或缩小。

2 在 C 语言中实现动态内存分配及其管理

在 C 语言中要实现根据程序的需要动态分配存储空间,就必须用到以下几个函数:

- (1) malloc 函数。

malloc 函数的原型为: void * malloc (unsigned int size)

其作用是在内存的动态存储区中分配一个长度为 size 的连续空间。其参数是一个无符号整形数,返回值是一个指向所分配的连续存储域的起始地址的指针。还有一点必

收稿日期:2006-03-17

作者简介:杨少春(1958-),男,河南南阳人,副教授,从事电子信息技术研究和教学工作。

须注意的是,当函数未能成功分配存储空间(如内存不足)就会返回一个 NULL 指针。所以在调用该函数时应该检测返回值是否为 NULL,并执行相应的操作。

(2) free 函数。

由于内存区域总是有限的,不能无限制地分配下去,而且一个程序要尽量节省资源,所以当所分配的内存区域不用时,就要释放它,以便其它的变量或者程序使用。这时就要用到 free 函数。

其函数原型是: void free(void * p)

作用是释放指针 p 所指向的内存区。其参数 p 必须是先前调用 malloc 函数或 calloc 函数(另一个动态分配存储区域的函数)时返回的指针。给 free 函数传递其它的值很可能造成死机或其它灾难性的后果。

注意:这里重要的是指针的值,而不是用来申请动态内存的指针本身。

3 嵌入式图形系统 μC/GUI 的动态内存分配方案

3.1 μC/GUI 系统有关动态内存分配的常量、结构及预定义项

3.1.1 定义常量

tALLOCINT - 用来设定记录每块内存偏移内存起点的变量类型是 2 个字节还是 4 个字节, GUI_ALLOC_SIZE 大于 32767 时,定义为 4 个字节类型。

```
# if GUI_ALLOC_SIZE < 32767
# define tALLOCINT I16
# else
# define tALLOCINT I32
# endif
```

HANDLE - 设定内存块句柄类型是 1 个字节还是 2 个字节,当 GUI_MAXBLOCKS 大于 256 时定义为 2 个字节。

```
# if GUI_MAXBLOCKS ≥ 256
# define HANDLE U16
# else
# define HANDLE U8
# endif
```

tALLOCINT, HANDLE 的定义影响到用于记录每一块内存的信息节点的大小,即用于动态内存分配的开销。

3.1.2 定义结构

记录每个内存块信息的节点结构。

```
typedef struct {
    tALLOCINT Off; /* Offset of memory area */
    tALLOCINT Size; /* usable size of allocated block */
    HANDLE Next; /* next handle in linked list */
    HANDLE Prev;
} tBlock;
/* 下面这个结构的定义在 GUI.h 当中 */
typedef union {
    int aintHeap[GUI_ALLOC_SIZE/4]; /* required for
```

```
proper alignment */
    U8 abHeap[GUI_ALLOC_SIZE];
} GUI_HEAP;
extern GUI_HEAP GUI_Heap; /* Public for debugging only
*/
static tBlock aBlock[GUI_MAXBLOCKS];
定义这样的结构,实质是通过一个全局数组 aBlock[]
来实现的,它的大小由 GUI_MAXBLOCKS = (2 + GUI_
ALLOC_SIZE/32)字节数来确定[2]。
aBlock[]是用于记录所有内存块的数组,每一个元素
记录一个内存块的信息,其所有的内存块不但被记录在这
一个数组中,而且组成了一个双链表,这样有利于遍历所
有已经分配的内存块。
struct {
    int NumUsedBlocks, NumFreeBlocks, NumFreeBlocksMin;
/* For statistical purposes only */
    tALLOCINT NumUsedBytes, NumFreeBytes, NumFree-
BytesMin;
```

```
} GUI_ALLOC;
```

同时通过 GUI_HEAP 这个共用体,以 abHeap[]来访问是以 1 个字节为单元,aintHeap[]则是以 4 个字节为单元。

3.1.3 预定义项

动态分配的预定义项如下所示:

```
# define GUI_ALLOC_SIZE 12500 /* Size of dy-
namic memory */
```

其中,GUI_ALLOC_SIZE 定义的是整个 μC/GUI 嵌入式图形系统中可用于动态分配的内存大小,也只有当这个预定义打开后,才能使 μC/GUI 系统提供的动态内存分配的功能在 GUIAlloc.c 文件中得以实现。

3.2 μC/GUI 系统中实现动态内存分配功能的函数

在 GUI.h 文件中定义相关函数来实现动态内存分配,GUI.h 文件的形式如下所示:

```
/* * * * * *
* Dynamic memory management
* * * * *
# if ! defined(GUI_ALLOC_ALLOC)
void GUI_ALLOC_Init(void);
void * GUI_ALLOC_h2p(GUI_HMEM hMem);
void GUI_ALLOC_Free(GUI_HMEM hMem);
void GUI_ALLOC_FreePtr(GUI_HMEM * phMem);
GUI_HMEM GUI_ALLOC_Alloc(int size);
/* diagnostics */
int GUI_ALLOC_GetUsed(void);
int GUI_ALLOC_GetNumFreeBytes(void);
int GUI_ALLOC_GetMaxSize(void);
/* macros */
# define GUI_ALLOC_ALLOC(size) GUI_ALLOC_Alloc
(size)
# define GUI_ALLOC_H2P(h) GUI_ALLOC_h2p(h)
```

```
# define GUI_ALLOC_FREE(handle) GUI_ALLOC_
Free(handle)
# define GUI_ALLOC_LOCK(handle) GUI_ALLOC_
h2p(handle)
# define GUI_ALLOC_UNLOCK(handle)
# endif
```

3.2.1 编写初始化的函数

```
void GUI_ALLOC_Init(void);
```

主要初始化 GUI_ALLOC 这个整体内存分配信息结构,并置已经初始化状态,初始化了第一个内存分配节点:

```
aBlock[0].Size = (1 < GUI_BLOCK_ALIGN); /
* occupy minimum for a block */
aBlock[0].Off = 0;
aBlock[0].Next = 0;
```

这个节点是用于双链表的头节点,它一直存在,不会有任何变化,其作用是为了维护内存分配信息节点双链表^[3]。

3.2.2 内存分配的函数定义

```
GUI_HMEM GUI_ALLOC_Alloc(int size);
```

这个函数是通过调用 Alloc() 进行内存分配的,这个函数主要完成以下的功能:

(1)调用函数 Size2LegalSize() 将要分配内存大小调整至最小内存单元的整数倍。

(2)寻找可用于记录此块分配信息的节点,在函数 FindFreeHandle() 中完成。

(3)寻找在整个内存分配空间中从低至高可满足此次分配的区域,在函数 FindHole() 中完成。

(4)如果剩余字节不够分配,当预定义碎片整理时,调用函数 CreateHole() 进行碎片整理。

(5)将分配所取得的内存分配信息节点加入双链表、初始化此次分配内存为 0 值、更新 GUI_ALLOC 这个整体分配结构体信息。

函数 FindFreeHandle() 作用是从数组 aBlock[] 中找出未用的节点(注意是从节点 1 开始找,节点 0 已经使用了)。如果找不到未用的节点返回 GUI_HMEM_NULL(0),函数 GUI_ALLOC_Alloc() 检测到此值时即返回已分配内存句柄为 0。GUI_ALLOC_H2P() 中转换此内存句柄时,如果检测到内存句柄为 0,则会返回此内存句柄帧内对应内存地址值为 0。

FindHole() 遍历双链表中的所有已分配节点,寻找此次要分配的内存的区域,有两种情况:

·所找到的区域在已经分配节点之间,这种情况是经过释放过内存之后再分配新的内存之时,查找方法是由后一分配节点偏移量减去当前分配节点偏移量与大小和的差,即 $aBlock[iNext].Off - (aBlock[i].Off + aBlock[i].Size)$, 确定后一节点与前一点之间是否有间隙,且此间隙是否满足此次分配要求。这种间隙其实是由释放内存块后引起的^[4,5]。

·所找到的区域在所有已经分配节点之后,在剩余的空间由函数 FindHole() 找到可满足分配的区域时,其返回值是可分配区域的最邻近区域的内存句柄,剩余空间不够才返回 -1。即在全局内存中没有一整块如此大的内存能满足此次分配,无法满足分配的原因可能是由于过多的小块内存被释放后形成了碎片,这些碎片夹杂在整个内存之间,所以解决办法就是将这些碎片合成一块大的内存。

在整理碎片,CreateHole() 要完成如下几件事:

①首先要从已分配节点中找出间隙,找出间隙的方法就是 $space = aBlock[iNext].Off - (aBlock[i].Off + aBlock[i].Size)$, 当 space 小于要分配的内存大小,将成为整理的对象。

②整理的方法,在保证已分配内存的数据和正常访问的情况下,将有间隙的两个节点的后一节点数据前移,并调整后一节点的偏移,这是注意的地方。

③最后此次内存是在所有已分配节点之后的,当 $GUI_ALLOC_SIZE - (aBlock[i].Off + aBlock[i].Size) \geq Size$ 这个条件满足,即调整碎片后所得的剩余空间满足此次分配,那么就返回 i 值, i 值即为双链表中最后一结点;如果调整碎片后还是无法满足些次分配,那上面那个条件不成立,则还是返回 -1,即此次分配失败。

3.2.3 内存释放的函数

释放与分配比起来,所做的工作少多了。GUI_ALLOC_Free() 与 GUI_ALLOC_FreePtr(), 两者完成同样功能,只是参数不同而已。

(1)根据参数中指定中的内存句柄,将这些内存句柄指对应分配节点 size 清零,对应内存清为 0xcc,并将节点从双链表中清除。

(2)更新 GUI_ALLOC 中记录的整体内存使用情况信息。

3.2.4 获取整体内存使用情况的函数

这一组函数比较简单,只作简短说明,它的信息基本上从 GUI_ALLOC 这个结构中取得。

GUI_GetUsedMem() - 获取已用内存字节数 NumUsedBytes。

GUI_ALLOC_GetNumFreeBytes() - 获取剩余内存字节数 NumFreeBytes。

GUI_ALLOC_GetMaxSize() - 遍历所有已分配节点,找出分配节点之间最大剩余一个区域的字节数,并与最后一个节点后剩余的内存比较,找出最大的剩余一块内存字节数。

4 结 论

文中系统地阐述了一个用于嵌入式操作系统的图形系统 $\mu\text{C}/\text{GUI}$ 的动态内存分配的方案,通过与 C 语言中实现动态内存分配及其管理的函数进行对比,可以发现采用

(下转第 96 页)

au1500_mii_init 的调用。

Au1500 各有 4 个 DMA 收发 FIFO,一共是 8 个通道,收发过程对 4 个通道是轮流使用的。

U-Boot 实现了一个简单的 TCP/IP 协议栈,整个 U-Boot 是不使用中断的,因此对于数据包的接收 U-Boot 采用轮循的办法。U-Boot 没有实现 TCP 协议,只有 UDP 协议,这已经足够使用了。

5 对 /include/configs/dbau1x00.h 的修改

这个文件是整个 U-Boot 的配置文件,U-Boot 的很多功能和参数都可以在这里进行调节。下面只举出其中的一部分作为例子来说明。

* CFG_MALLOC_LEN: malloc 区域可用的空间大小,这里设为 128 * 1024,即 128kB;

* CFG_HZ: CPU 的主频,设为 396000000;

* CFG_SDRAM_BASE: 内存基址,设为 0x80000000;

* CFG_MAX_FLASH_BANKS: 可以看成是 flash 的片数,设为 1,只用了一片 flash;

* PHYS_FLASH_1: 第一片 flash 的起始地址,为 0xbfc00000;

* CFG_ENV_IS_IN_FLASH: 将 CFG_ENV_IS_NOWHERE 改为 CFG_ENV_IS_IN_FLASH,其值设为 1,这样使 U-Boot 支持从 flash 读取环境变量;

* CFG_ENV_ADDR: 环境变量保存在 flash 中的起始地址,为 0xbfc80000;

* CFG_ENV_SIZE: 保存环境变量所占空间,为 0x10000;

* CFG_DCACHE_SIZE, CFG_ICACHE_SIZE, CFG_CACHELINE_SIZE: 对 cache 大小的设置。

6 U-Boot 对 Linux 的引导

在调试阶段,可以通过 tftp 将内核上传到指定的内存地址。之后可以用 go 命令执行上传的文件,这是针对纯二进制文件,go 命令直接跳转到指定地址执行上传的文件。U-Boot 还带有一个 mkimage 工具,它可以在文件开头加一个 U-Boot 可以识别的文件头,这样使用起来更

加方便。对 Linux 的引导是采取的这种方法。这种文件只能用 bootm 命令启动。

首先获得没有压缩的 Linux 内核纯二进制文件^[5],用 gzip 对其压缩,再用 mkimage 打包。打包时注意 -a 和 -e 这两个选项,-a 后面跟的是加载地址(load address),这里的加载地址并不是指的 tftp 的上载地址,bootm 执行后上传的文件首先被解压缩,-a 后面跟的地址即是解压地址。由于解压需要较大的空间,解压地址和 RAM 的起始地址之间应该留有足够的空间。-e 后面跟的是解压之后的跳转进入地址(entry point)。

文中使用的 mkimage 命令如下:

```
mkimage - AMIPS - O linux - T kernel - C gzip - a
0x82000000 - e 0x81000000 \ - n "Linux Kernel Image" - d linux.
bin. gz uImage
```

再说一下 bootargs 环境变量的设置,以笔者为例:

```
setenv bootargs root = /dev/nfs rw nfsroot = 191.168.
123.146:/home/au1500 nfsaddrs=191.168.123.152:191.
168.123.146
```

这里是用 nfs 的方式加载文件系统,nfsaddrs 中前一个地址是传给内核的板子的 IP,后一个是 nfs 服务器的 IP 地址。

7 结束语

通过移植,U-Boot 可以正常运行,一方面借助其强大的功能方便了后续的开发工作,另一方面,以后的 U-Boot 维护工作也可以借助开源社区的力量来完成。

参考文献:

- [1] Advanced Micro Devices, Inc. AMD Alchemy Solutions Au-1500 Processor Data Book [EB/OL]. <http://www.amd.com>, 2003.
- [2] MIPS Technologies, Inc. MIPS32 Architecture For Programmers [EB/OL]. <http://www.mips.com>, 2001.
- [3] Sweetman D. MIPS 处理器设计透视 [M]. 赵俊良等译. 北京:北京航空航天大学出版社, 2005.
- [4] Stevens R. TCP/IP 详解 [M]. 谢希仁等译. 北京:机械工业出版社, 2004.
- [5] MIPS Technologies, Inc. Linux Kernel Archives [EB/OL]. <http://www.linux-mips.org>, 2005.

(上接第 93 页)

这种分配方案使其具有占用 RAM 和 ROM 空间小的特点,从而能够极大地节省开发成本,丰富人机交互信息。因此可以得到结论:此方案可以有效地提高 $\mu C/GUI$ 系统的实时性,使之可以带来更大的经济效益,提高产品的性能。

参考文献:

- [1] 罗 蕾. 嵌入式实时操作系统及应用开发 [M]. 北京:北京航空航天大学出版社, 2005. 23-26.

- [2] 谭浩强. C 程序设计 (第 2 版) [M]. 北京:清华大学出版社, 2000. 101-108.
- [3] 探砂工作室. 嵌入式系统开发圣经 [M]. 北京:中国青年出版社, 2002. 139-149.
- [4] Mercer C W. An Introduction to Real-Time Operating Systems: Scheduling Theory [Z]. School of Computer Science, Carnegie Mellon University, 1992.
- [5] Bruyninckx H, Leuven K U. Real-Time and Embedded Guide [J]. Mechanical Engineering, 2001, 56: 5-9.