Chipcon Products from Texas Instruments

应用程序说明：
堆内存管理

文件编号：F8W-2006-0026

德州仪器公司
美国加利福尼亚州圣迭戈
(619) 497-3845

版本	描述	日期
1.0	初始发布。	12/06/2006
1.1	修改了文件名称。更新了标题页。	05/21/2007

目录

i、缩略语

API	应用程序编程接口
BSP	板支持包—HAL 和 OSAL 两者一起，组成了一个基本的操作系统，被称作 BSP。
HAL	硬件（H/W）抽象层
OSAL	操作系统（OS）抽象层
OTA	无线

1、堆内存管理

1.1 简介

OSAL 堆内存管理提供了一个类似于 POSIX 的 API，用于分配和循环使用动态的堆内存。在一个低成本、资源有限的嵌入式系统中执行堆内存管理时，两个重要的考虑，即规模和速度，已经被适当解决。

- 架空内存成本管理，每个分配的块已经达到最小——CPU 的分配块只有 2 字节，字节对齐内存访问。
- 分配和释放操作的中断延时已经减少到最低限度——在大量 OTA 信息负载期间，分配和释放一个内存块，平均合计只用 26 *微秒*。

1.2 API

1.2.1 osal_mem_alloc()

osal_mem_alloc()函数请求内存管理器为一个堆保留一个块。

1.2.1.1 原型

```
void  *osal_mem_alloc  (  uint16  size  );
```

1.2.1.2 参数

size—请求的动态内存的字节数。

1.2.1.3 返回值

如果找到一个足够大的空闲块，函数返回一个空指针，指向保留用于堆内存位置的 RAM。如果没有足够的内存可以分配，返回一个 NULL 指针。返回的任何非 NULL 的指针必须通过调用 osal_mem_free()[1] 释放，以重新使用。

1.2.2 osal_mem_free()

osal_mem_free()函数请求内存管理器释放之前为一个堆保留的一个内存块，这样内存可以重复使用。

1.2.2.1 原型

```
void  osal_mem_free  (  void  *ptr  );
```

1.2.2.2 参数

ptr——一个指向缓冲区的指针，以释放重复使用——这个指针必须是之前调用 osal_mem_alloc()返回的非 NULL 指针。

1.2.2.3 返回值

无。

1.3 策略

内存管理应该在堆中、在尽可能少的块中，努力保持连续的空闲空间，每个块要尽可能大。如果堆的总大小已经根据应用程序的使用模式合适地设置，这样的一个总体策略有助于确保较大的内存块请求总是成功的。

执行以下具体策略：

- 内存分配以在堆中第一个空闲块中寻找一个足够大的空闲块开始。
- 内存分配尝试合并所有连续的空闲块，以为一个分配请求形成一个足够大的空闲块。
- 内存分配使用遇到的（或合并得到的）足够大可以满足要求的第一个空闲块，如果大于所请求分配的内存块，内存块就被分割。

1.4 讨论

系统任务初始化之后，“堆开始”标志就立即被设置为有效，作为第一个空闲块。由于内存管理总是启动一个“行走”，从上述标志开始，寻找一个足够大的空闲块，如果所有长寿命的堆分配都紧密地放在堆的起始位置，这样它们就不必经过每个内存分配，这将大大降低行走的运行开销。因此，任何应用程序应在其各自的系统初始化例程中，执行所有长寿命的动态内存分配。

应用程序执行者必须确保使用动态内存不会对 Z-Stack 基本层的操作产生不利影响。Z-Stack 是根据最低限度使用堆内存的示例应用程序进行测试和认证的。因此，使用的堆明显多于示例应用程序的用户应用程序，或用户应用程序 MAXMEMHEAP 设置的值小于示例应用程序，可能会无意中造成 Z-Stack 的下层没有足够的内存，以致它们不能有效工作或根本不能工作。例如，应用程序可能分配了太多的动态内存，以致栈的底层无法分配足够的内存来发送和/或接收任何 OTA 信息——设备不会被看作参与 OTA。

1.5 配置

1.5.1 MAXMEMHEAP

MAXMEMHEAP 常量通常定义在 OnBoard.h 中。它必须定义为小于 **32768**。

MAXMEMHEAP 是内存管理器为堆保留的 RAM 字节数——它不能在运行期间动态改变——它必须在编译期间定义。如果 MAXMEMHEAP 定义为大于或等于 32768，就会在 osal_memory.c 中发生一个编译错误。MAXMEMHEAP 不会反映用户预期可以使用的动态内存的总数量，因为每个内存分配都有开销成本。

1.5.2 OSALMEM_PROFILER

OSALMEM_PROFILER 常量本地定义在 osal_memory.c 中，默认为 **FALSE**。

执行完毕一个用户应用程序之后，为了在 MAXMEMHEAP 定义的限制下，达到最佳运行性能，OSAL 内存管理器可能需要重新调整。通过定义 OSALMEM_PROFILER 常量为 **TRUE** 来使能代码，允许用户收集所需的经验、运行结果来为应用程序调整内存管理器。代码的设置如下。

1.5.2.1 OSALMEM_INIT

OSALMEM_INIT 常量本地定义在 osal_memory.c 中，以 ASCII 码“**X**”表示。

内存管理器初始化将设置堆中所有字节为 OSALMEM_INIT。

1.5.2.2 OSALMEM_ALOC

OSALMEM_INIT 常量本地定义在 osal_memory.c 中，以 ASCII 码“**A**”表示。

任何分配块中用户可用的字节设置为 OSALMEM_ALOC。

1.5.2.3 OSALMEM_REIN

OSALMEM_INIT 常量本地定义在 osal_memory.c 中，以 ASCII 码“**F**”表示。

每当释放一个块，无论用户可用的字节是什么值，都被设置为 OSALMEM_REIN。

1.5.2.4 OSALMEM_PROMAX

OSALMEM_PROMAX 常量本地定义在 osal_memory.c 中，以 **8** 表示。

OSALMEM_PROMAX 是要配置的不同型号的 bucket 空间。Bucket 空间由一个数组定义：

```
static   uint16   proCnt[OSALMEM_PROMAX] = { OSALMEM_SMALL_BLKSZ,
                                              48, 112, 176, 192, 224, 256, 65535 };
```

bucket 空间配置根据应用程序的调整设置，但是最后一个 bucket 必须总是 65535 作为全部选择。每个 bucket 有三个指标。

- proCur–适合相应 bucket 空间的当前已分配块的个数。
- proMax–一次对应 bucket 空间的已分配块的最大个数。
- proTot–对应 bucket 空间的块被分配的总次数。

另外，有一个计数器，记录的是为“小块”保留的堆的部分太满而不能允许请求分配小块: proSmallBlkMiss 的总次数。

1.5.3 OSALMEM_MIN_BLKSZ

OSALMEM_MIN_BLKSZ 常量本地定义在 osal_memory.c 中，默认以 **4** 表示。

OSALMEM_MIN_BLKSZ 是通过把一个空闲块分割成两个新块创建的一个块的最小字节数。第一个新块的大小是内存分配所请求的大小，会被标记为正在使用。无论第二个块剩余大小是多少都会被标记为空闲。较大的数值可能会导致应用程序总的运行速度明显加快，而总的堆大小不需要更多或不是很多。例如，如果应用程序有大量相互混合、每个 2 &4 个字节的短寿命内存分配，每个相应的块就有 4&6 个字节的开销。内存管理器可能要花费许多时间来处理，因为在堆的同一个一般区域，为了能够容纳相互混合的大小的请求，它要被多次分割和合并。

1.5.4 OSALMEM_SMALL_BLKSZ

OSALMEM_SMALL_BLKSZ 常量本地定义在 osal_memory.c 中，默认以 **16** 表示。

堆内存使用的 Z-Stack 是运用 TransmitApp 示例应用程序配置的，它根据经验确定在大量 OTA 负载期间，内存分配和释放最好的最坏的情况的平均结合时间，可以通过把空闲堆分裂为两部分来实现。第一部分保留分配给较小的块，第二部分用于分配给较大的块，而且如果第一部分满了，也分配给较小的块。OSALMEM_SMALL_BLKSZ 是第一部分可以分配的最大块的字节数。

1.5.5 SMALLBLKHEAP

SMALLBLKHEAP 常量本地定义在 osal_memory.c 中，默认以 **232** 表示。

SMALLBLKHEAP 是堆前面描述的堆的第一部分的字节数，为较小的块保留。当配置

堆内存为使用 TransmitApp 时，同时可以分配最多 18 个 OSALMEM_SMALL_BLKSZ（比如，16）大小的块。尽管根据前面的说明可以定义 SMALLBLKHEAP 为 288（比如，16*18），进一步的工作表明，最好的最坏的情况次数只是稍微小于默认设置下的次数[2]。

1.5.6 OSALMEM_NODEBUG

OSALMEM_NODEBUG 常量本地定义在 osal_memory.c 中，默认以 **TRUE** 表示。

Z-Stack 和实例应用程序不会滥用堆内存 API。[3] 同样用户应用程序也有这个责任：为了尽可能提供最小的吞吐量延时，不对 API 的正确使用做运行期间检查。通过定义 OSALMEM_NODEBUG 常量为 FALSE，可以证明应用程序是正确的。这样的设置将使能下述滥用情况中的代码。

- 调用 osal_mem_alloc()，大小等于零。
- 调用 osal_mem_free()，有一个为 NULL 的指针。
- 调用 osal_mem_free()，有一个已释放的指针。

警告：调用含有一个悬挂或无效指针的 osal_mem_free()是不能被检测出来的。

1.5.7 OSALMEM_GUARD

OSALMEM_GUARD 常量本地定义在 osal_memory.c 中，默认以 **TRUE** 表示。

1.4 Z-Stack 发布时有一个已知的漏洞：是否定义了一个常量，LCD_SUPPORTED。初始化堆内存管理之前，LCD 的系统初始化支持使用堆内存 API。为了防止在堆内存 API 初始化之前使用任何堆内存 API，通过 OSALMEM_GUARD 使能的代码将导致产生运行吞吐量成本以检查一个标志。如果应用程序不使用 LCD，或如果用户修复了本地漏洞，那么通过定义 OSALMEM_GUARD 为 FALSE，可以消除运行检查（和随之产生的吞吐量成本）。

附录 A 适用文件

内部文件

1、Z-Stack OSAL API , F8W-2002-0002
2、Z-Stack 编译选项，F8W-2005-0038

---

1 内存管理器上没有实施自动垃圾收集。因此任何不再使用但是没有手动释放的堆内存将永远丢失，被视作一个“内存漏洞”。Z-Stack 和实例应用程序是不会泄漏的。内存漏洞最终使应用程序出现“锁定”—最终甚至没有足够的空闲堆来设置一个 OSAL 定时器，或写入一个调试信息到串行端口。

2 这一直观次数结果的分析超出了本文件的范围。

3 除了下面 1.4.6 节指出的滥用情况。