Newlib的研究与最小实现

张宇旻，罗 蕾

(电子科技大学计算机科学与工程学院 成都 610054)

【摘要】对嵌入式C运行库—— newlib进行了深入研究，阐述了该运行库在多任务环境下可重入性的实现方法；介绍了移植newlib到嵌入式系统上需要的桩函数及其实现方法，并重点介绍了与I/O相关的四个桩函数open、close、read和write的实现方法，以及动态内存分配器malloc的两种实现方法。

关 键 词 嵌入式系统； C运行库； 可重入性； 桩函数

中图分类号 TP393 **文献标识码** A

Research and Minimum Accomplishment of Newlib

ZHANG Yu-min，LUO Lei

(School of Computer Science and Engineering, UEST of China Chengdu 610054)

Abstract This article has a research on a embedded C Runtime Library: newlib. It presents the way to accomplish the reentry of newlib under multi-task environment. It introduces stub functions implementation which is needed by porting newlib on embedded system, especially open, close, read and write implementation which relate with I/O, and malloc two implementation ways.

Key words embedded system; newlib; c runtime library; reentry; stub

1 Newlib简介

Newlib是一个面向嵌入式系统的C运行库。最初是由Cygnus Solutions收集组装的一个源代码集合，取名为newlib，现在由Red Hat维护，目前的最新的版本是1.11.0[1]。

对于与GNU兼容的嵌入式C运行库，Newlib并不是唯一的选择，但是从成熟度来讲，newlib是最优秀的。newlib具有独特的体系结构，使得它能够非常好地满足深度嵌入式系统的要求。newlib可移植性强，具有可重入特性、功能完备等特点，已广泛应用于各种嵌入式系统中。

2 可重入性的实现

C运行库的可重入性问题主要是库中的全局变量在多任务环境下的可重入性问题，Newlib解决这个问题的方法是，定义一个struct _reent类型的结构，将运行库所有会引起可重入性问题的全局变量都放到该结构中。而这些全局变量则被重新定义为若干个宏，以errno为例，名为“errno”的宏引用指向struct _reent结构类型的一个全局指针，这个指针叫做_impure_ptr。

收稿日期：2003 – 12 – 05

基金项目：四川省重点科技攻关项目

作者简介：张宇旻(1978 –)，男，硕士，主要从事嵌入式系统软件方面的研究；罗 蕾(1967 –)，女，硕士，教授，主要从事嵌入式系统软件方面的研究.

对于用户，这一切都被errno宏隐藏了，需要检查错误时，用户只需要像其他ANSI C环境下所做的一样，检查errno“变量”就可以了。实际上，用户对errno宏的访问是返回_impure_ptr->errno的值，而不是一个全局变量的值。

Newlib定义了_reent结构类型的一个静态实例，并在系统初始化时用全局指针_impure_ptr指向它。如果系统中只有一个任务，那么系统将正常运行，不需要做额外的工作；如果希望newlib运行在多任务环境下，必须完成下面的两个步骤：

1) 每个任务提供一个_reent结构的实例并初始化；

2) 任务上下文切换的时刻重新设置_impure_ptr指针，使它指向即将投入运行任务的_reent结构实例。

这样就可以保障大多数库函数(尤其是stdio库函数)的可重入性。如果需要可重入的malloc，还必须设法实现__malloc_lock()和__malloc_unlock()函数，它们在内存分配过程中保障堆(heap)在多任务环境下的安全。

3 Newlib的移植

Newlib的所有库函数都建立在20个桩函数的基础上[2]，这20个桩函数完成一些newlib无法实现的功能：

1) 级I/O和文件系统访问(open、close、read、write、lseek、stat、fstat、fcntl、link、unlink、rename)；

2) 扩大内存堆的需求(sbrk)；

3) 获得当前系统的日期和时间(gettimeofday、times)；

4) 各种类型的任务管理函数(execve、fork、getpid、kill、wait、_exit)；

这20个桩函数在语义、语法上与POSIX标准下对应的20个同名系统调用是完全兼容的[3]。成功移植newlib的关键是在目标系统环境下，找到能够与这些桩函数衔接的功能函数并实现这些桩函数。

Newlib为每个桩函数提供了可重入的和不可重入的两种版本。两种版本的区别在于，如果不可重入版桩函数的名字是xxx，则对应的可重入版桩函数的名字是_xxx_r，如close和_close_r，open和_open_r，等等。此外，可重入的桩函数在参数表中含有一个_reent结就

对于设备名表应该实现以下两个操作：

(1) 设备名/设备号注册函数NameRegister；

(2) 从设备名到设备号的转换函数NameLookup；

2) 文件描述符表记录系统中当前打开的设备的设备号。每个表项代表一个处于打开状态的设备。每个表项的索引值就是需要返回给用户的文件描述符。

对文件描述符表需要实现以下3个操作：

(1) 文件描述符分配函数FdAllocate；

(2) 文件描述符释放函数FdFree；

(3) 从文件描述符到设备号的转换函数Fd2DevCode；

3) 驱动地址表记录系统中每个驱动程序的入口地址。每个表项代表一个驱动程序，对每个驱动程序都应该实现五个具有统一接口的操组函数：init、open、close、read、write。每个表项在表中的索引值就是该设备的设备号。需要注意是每个驱动程序都必须提供init操作。

对驱动地址表需要实现以下操作：

初始化驱动表中的所有驱动函数InitAllDrivers；

该操作对表中的每一个驱动程序调用init操作，完成表中所有驱动程序的初始化操作。

在系统初始化的时间，应该调用InitAllDrivers()操作，完成系统中所有驱动程序的初始化操作。在每个驱动程序的init操作中，应该调用NameRegister()操作，完成驱动程序对应的设备注册，以COM1驱动程序的com1_init()操作为例，它的实现如下：

```
void com1_init(int devCode)
{
    /*首先注册设备名和设备号到设备名表中*/
    NameRegister(“COM1”, devCode);
    /*然后完成其他的设备初始化操作*/
}
```

只要所有的设备驱动程序都遵守这个约定，在系统初始化完成之后，系统中所有的驱动程序就得到了初始化，并且系统中所有的设备都注册到了设备名表中。后续的I/O桩函数的实现就非常容易了。

设备名表、文件描述符表和驱动地址表3个表的结构及相关操作如图1所示。

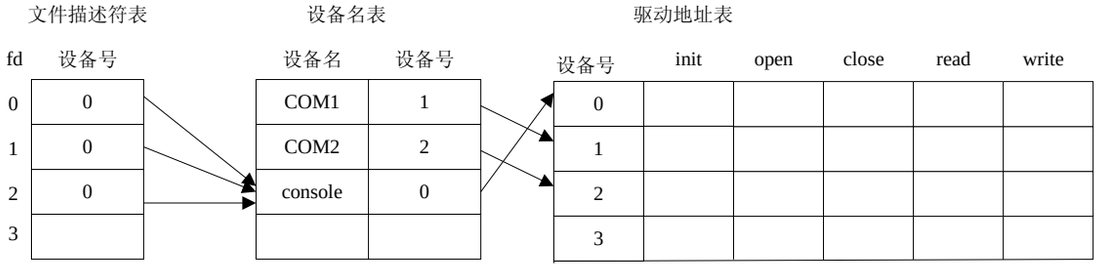

图1　文件描述符表、设备名表和驱动地址表

4.2　open 桩函数的实现

open桩函数的实现流程如下：

1) 用NameLookup()操作在设备名表中搜索匹配的设备名，并获得对应的设备号；

2) 用FdAllocate()操作从文件描述符表中分配一个空的表项，填入设备号，并获得对应的索引号即fd；

3) 通过设备号直接调用驱动地址表中对应驱动程序的open操作；

4) 返回fd。

4.3 read、write和close桩函数的实现

read和write桩函数的实现方法完全相同，流程如下：

1) 调用Fd2DevCode()操作获得与输入参数fd对应的设备号devCode；

2) 通过设备号直接调用驱动地址表中对应驱动的read或write操作；

3) 返回实际交换的数据量。

close桩函数的实现与read、write几乎完全相同，唯一不同之处在于最后调用FdFree()操作，释放fd而不是返回实际交换的数据量，流程如下：

1) 调用Fd2DevCode()操作获得与输入参数fd对应的设备号devCode；

2) 通过设备号直接调用驱动地址表中对应驱动的close操作；

3) 调用FdFree()操作释放fd。

至此，与设备I/O相关的四个桩函数open、close、read和write的实现就全部完成了。

本文没有介绍驱动程序的实现方法，并不是驱动程序不重要，恰恰相反，驱动程序中必须完成可靠高效的设备操作，保证驱动程序的各项操作在语义上与上面4个桩函数完全一致，并且实质性的操作都在驱动程序中完成。因此，在驱动程序的实现上必须仔细斟酌。由于篇幅的原因，不再赘述。

5 关于malloc

大多数嵌入式操作系统都实现了自己的动态内存分配机制，并且提供了多任务环境下对内存分配机制的保护措施，如果移植newlib到这样的系统时，可以放弃newlib自带的malloc函数。尽管newlib自带的malloc非常高效，但是几乎所有的用户都习惯使用malloc来作为动态内存分配器。在这种情况下，最好对系统自带的动态内存分配API进行封装，使它不论在风格、外观上，还是在语义上都与malloc完全相同，这对于提高应用程序的可移植性大有好处。

对于那些没有实现动态内存分配机制的嵌入式系统环境来说，newlib的malloc是一个非常好的选择，只需实现sbrk桩函数，malloc就可以非常好地工作起来。与之同名的POSIX系统调用的作用是从系统中获得一块内存，每当malloc需要更多的内存时，都会调用sbrk函数。

在单任务环境下，只需实现sbrk桩函数，malloc就可以正常运行；但在多任务环境下，还需实现__malloc_lock()和__malloc_unlock()函数，newlib用这两个函数来保护内存堆免受冲击。用户可利用目标环境中的互斥信号量机制来实现这两个函数，在__malloc_lock()函数中申请互斥信号量，而在__malloc_unlock()函数中释放同一个互斥信号量。

6 结 束 语

本文中详细介绍了newlib作为一个面向嵌入式系统的C运行库所具备的各种特点，重点介绍了ewlib在多任务环境下的可重入性和实现方法，以及ewlib的移植方法两方面的内容。尤其是针对newlib的移植，详细介绍了每一个桩函数的实现方法，并给出了许多的代码实例。但是并不是说对newlib只能像本文中所描述的那样进行移植，本文给出的仅仅是移植newlib到特定系统之上的一个最小实现。Newlib的体系结及其清晰简洁的实现方法决定了newlib具有非常的灵活性，能够适应各式各样的系统和目标环境。随着研究工作的深入，newlib在目标系统中将会发挥更大的作用。

参 考 文 献

[1] Gatliff B. Embedding GNU: newlib: [J/OL]. http: //www. embedded.com/story/OEG20020103S0073.htm, 2003-11-30

[2] Gatliff B. Embedding GNU: newlib, part 2[J/OL]. http: //www. embedded com/story/OEG20011220S0058. htm, 2003-11-30

[3] Stevens W R. Advanced programming in the UNIX environment: [M]. 北京：机械工业出版社, 2000. 35 ~ 52

编 辑 熊思亮