熟悉 binutils 工具集

李　云
Blog: yunli.blog.51cto.com

摘要

对于嵌入式系统开发，掌握相应的工具至关重要，它能使我们解决问题的效率大大提高。目前，可以说嵌入式系统的开发工具是 GNU 的天下，因为来自 GNU 的 GCC 编译器支持大量的目标处理器。除了 GCC，还有一个非常重要的、同样来自于 GNU 的工具集（toolchain）—— binutils toolchain。这一工具集中存在的一些工具，可以说是我们开发和调试不可缺少的利器。

本文通过介绍 binutils 以及提供一定的使用实例来帮助读者熟悉这一工具集，以达到提高效率的目的。当你掌握了 binutils 后，你会发现你得到的是“渔”而不只是“鱼”。

关键词

binutils　　工具集

参考资料

《什么是 boot loader》
《堆和栈》
《程序中的段》
《C 语言中一个字节对齐问题的分析》

1　引言

对于嵌入式系统开发，掌握相应的工具至关重要，它能使我们解决问题的效率大大提高。目前，可以说嵌入式系统开发工具是 GNU（www.gnu.org）的天下，因为来自 GNU 的 GCC 编译器支持大量的目标处理器。除了 GCC，还有一个非常重要的、同样来自于 GNU 的工具集（toolchain）—— binutils toolchain。Binutils 中的工具不少和 GCC 相类似，也是针对特定的处理器的。

你可能要问：哪些嵌入式操作系统的开发是采用 GNU 工具集（包括 GCC 编译器、binutils 工具集等）的？Linux 相关的实时（Monta Vista Linux、WindRiver Linux、RTLinux 等）或非实时嵌入式系统开发就不用说了，全是采用 GNU 工具集的；最为有名的来自 WindRiver（现已被 Intel 收购）的 VxWorks 操作系统也是采用 GNU 工具集的，为了使用 GNU 工具集，VxWork 的开发 IDE 采用 Cygwin 作为其在 Windows 操作系统的支撑平台；还有就是 RTEMS（www.rtems.org）操作系统，以前是美国军方的一个实时操作系统，后来开源了，也是采用 GNU 工具集的；此外，另一个很有名的实时操作系统 —— eCos，也是采用 GNU 工具集的，如果你熟悉 Altera 的 Nios，那么对 eCos 也应当不陌生；等等。我想可以举出很多很多的例子。例子越多，说明我们学习 binutils 就越是有用！还有对于 bintuils 工具集的学习，不光是对于嵌入式系统开发有用，对于 Linux 主机或是 Solaris 服务器上的程序开发也是很有帮助的。

对于采用 C/C++从事 Windows 应用程序开发的人来说，很有可能会问：我在 Windows 上的一个目标文件其后缀是.obj，在 GNU 的工具集中仍是采用.obj 后缀吗？在 Windows 中的动态库是以.dll 结尾的，那在 GNU 的工具集中也一样吗？。这些都是很好的问题，通过类比，我们可以根据我们的经验去掌握另一类似的新东西。在 GNU 工具集中，一个源程序（.c 或是.cpp）是先被编译成.o 目标文件（对应于 Windows 中的.obj 文件）的，如果目标文件直接连接成可执行文件，则生成的是 ELF（Excutable and Linkable Format）文件。这种可执行文件对应于 Windows 中的.exe 文件，与 Windows 系统所不同的是，在 GNU 工具集中一个可执行文件并没有一个统一的后缀，甚至没有后

缀。如果要将多个.o 文件生成一个库文件，那么存在两种类型的库：一种是静态库，其后缀是.a；另一种是动态库，其后缀是.so。在 Windows 系统中，其全部都是.dll。静态库与动态库的区别是什么呢？静态库是每一个与这一库进行连接的都将有一份代码（和数据）拷贝。比如，如果 libx.a 中存在一个 foo ()函数，而程序 A 和程序 B 都需要采用 libx.a 进行连接以使用其中的 foo ()函数，那么在连接以后，程序 A 和 B 的可执行程序中都会存在一个 foo ()函数，即程序 A 和程序 B 的可执行代码中都存在 foo ()函数的一个拷贝。与静态库所不同的是，采用动态库则不会生成多个代码拷贝。采用动态库时，如前面的程序 A 和 B，所有的程序共享这个库的代码，即在内存中只存在这个库中代码的一个拷贝，但这个库中的（可读写）数据仍然是每一个程序拥有一个独立的拷贝。

在 binutils 中以下的工具是我们在做嵌入式系统开发时需要掌握的：

- *as* 是汇编器，在此我不打算对其进行讲解，因为其涉及到了处理器的指令集，我们在合适的时候再来讲。
- *addr2line* 用得到程序地址所对应源代码的文件名和行号以及所对应的函数。
- *ar* 用于创建、修改档案文件（比如.a 静态库文件）以及从档案文件中抽取文件（比如从.a 静态库中抽取.o 文件）。
- *ld* 是连接器，对其的讲解我打算采用独立的一篇文章来进行，因为连接器在嵌入式系统开发中非常重要。比如，我们需要通过写或是修改连接脚本，来定制我们的嵌入式程序中的各个段（section）。
- *nm* 用于列出目标文件、库或是可执行文件（后面统称这三种文件为***程序文件***）中的代码符号及代码符号所对应的程序开始地址。
- *objcopy* 是用来拷贝或是翻译目标文件的。
- *objdump* 帮助我们显示程序文件的相关信息。
- *ranlib* 用于生成一个档案文件的内容索引。这样做的目的是为了加快档案文件的访问速度，比如，我们常对静态库文件（.a 文件）进行 ranlib 以提高连接速度。
- *readelf* 用于显示 ELF 文件的信息。
- *size* 用于显示程序文件的段信息。
- *strings* 用于显示一个程序文件当中的可显示字符串。
- *strip* 用于剥去程序文件中的符号信息，以减小程序文件的大小。这对于存储空间有限的嵌入式系统尤为有用。

在接下来的章节，我们将看一看各个工具的使用方法和使用例子。需要注意的是，本文并不是 binutils 工具集的完整参考手册，对于每一个工具的讲解都是基于其常用功能来进行的，当你需要得到更为详细的帮助信息时，完全可以参照相应工具的 man（或 info）信息。比如，你要获得 objdump 工具的 man 信息，你可以在 Linux 或是 Cygwin 中运行“man objdump”。另一种更为简单的方法是采用--help 参数运行相应工具得到简单的帮助信息，比如“objdump --help”。

需要注意的是，这不是一篇教你如何进行 Linux 程序开发的文章，相反，这里假设了你了解一些基本的 Linux 命令。同样地，这篇文章不会告诉你什么时候要用 GCC 进行编译，而什么时候又得用 G++进行编译，更不会告诉你这些编译器的具体参数的意思是什么以及如何使用。对于这些信息，你需要参考其它的文章或是书籍。

2 准备环境

在讲解 binutils 中的工具之前，我们需要有一定的环境用于练习。你可以找一台安装有 GCC 的 Linux 计算机（可以是 Wmware 上虚拟的），如果没有，你可以在你的 Windows 上安装一个 Cygwin。如果你需要安装 Cygwin，通常分为以下几个步骤：

- 从 www.cygwin.com 上下载 setup.exe，并运行它。然后选择“从 Internet 下载安装包到本地”。在下载之前，请确保你选择了下载 GCC 和 binutils 安装包。
- 安装包下载完了以后，你需要再次运行 setup.exe，且这次选择“从本地安装”。在安装时，

同样不要忘了选择安装 GCC 和 binutils。

当你安装好了 Cygwin 后，运行 Cygwin 并在其上运行如下的命令（注：美元符‘$’不是命令的一部分，它是命令提示符，这如同 Windows 命令窗口中的‘C:\>’提示符）来验证 binutils 是否已准备好。不管你使用的是 Linux 操作系统或是 Cygwin，如果你能看到命令运行后出现了对于这一命令的使用说明，那么说明 binutils 在你的环境中被正确地安装了。

```
yunli.blog.51cto.com ~
$nm -h
```

如果采用 Cygwin，由于 Cygwin 只是在 Windows 上模拟 Linux 的环境，因此，其有些行为仍然是像 Windows 的。比如，如果我们采用以下的命令来编译一个程序，在 Linux 上其生成的可执行文件名就是 test，而在 Gygwin 上则为 test.exe。在后面的使用实例中，你需要注意这一区别。了就是说，在 Linux 中可能输入的是 test，而在 Cygwin 中你必须换成输入 test.exe；反之亦然。

```
yunli.blog.51cto.com ~
$gcc main.c -o test
```

3 addr2line

addr2line 是用来将程序地址转换成其所对应的程序源文件及所对应的代码行，当然，也可以得到所对应的函数。为了说明 addr2line 是如何使用的，我们需要有一个练习用的程序。先采用编辑工具编辑一个 main.c 源文件，其内容如图 1 所示。

main.c

```
#include <stdio.h>

void foo ()
{
    printf (“The address of foo () is %p.\n”, foo);
}

int main ()
{
    foo ();
    return 0;
}
```

图 1

运行如下的命令将 main.c 编译成可执行文件，并运行之。在运行 test.exe 程序后，我们可以在其终端上看到它打印出的 foo ()函数的地址 —— 0x401100。

```
yunli.blog.51cto.com ~
$gcc -g main.c -o test
yunli.blog.51cto.com ~
$./test.exe
The address of foo () is 0x401100.
```

现在，我们可以用这一地址来看一看 addr2line 是如何使用的。在终端中运行如下的命令，从命令的运行结果来看，addr2line 工具正确的指出了地址 0x401100 所对于应的程序的具体位置是在哪以及所对应的函数名是什么。

```
yunli.blog.51cto.com ~
$addr2line 0x401100 -f -e test.exe
```

```
foo
/home/Administrator/main.c:4
```

可能有人会问了：这个 0x401100 地址是我们打印出来，即然有打印，我们一般情况下也会打印出其具体的函数位置，而不是只打印地址，我为何要这么绕一下通过 addr2line 去找到地址所对应的函数呢？其实，这里打印出地址只是为了得到一个地址以便用于练习。在现实中，地址往往是在调试过程中或是当程序崩溃时通过某种方式获得的。此外，采用 nm 工具（后面会讲到）可以得到如下的函数地址信息。

```
yunli.blog.51cto.com ~
$nm -n test.exe
…显示结果有删减…
00401100 T _foo
0040111C T _main
```

nm 命令会打印出所有的符号（包括函数和全局变量名）所对应的开始地址，需要注意的是，在 C 中源代码中的函数其所对应的 nm 输出符号前会多加一个下划线‘_’，比如 C 程序中的 main ()函数对应的是 nm 输出符号中的_main，同样地，C 程序中的 foo ()函数对应的是 nm 输出符号中的_foo。从 nm 输出的信息你可以看出，foo ()函数所对应的地址为 0x00401100，而 foo ()函数是有大小的（因为其有实现代码，且代码越是复杂或是长，则函数的大小越大），其大小是_main 的地址减_foo 的地址（_main 紧跟在_foo 的后面说明在 C 程序中 main ()函数是跟在 foo ()函数的后面的），那么是不是说我们给 add2line 的地址可以是从 0x0040100 到 0x0040111C 的任一地址呢？是的，请看下面的操作结果。

```
yunli.blog.51cto.com ~
$addr2line 0x401110 -f -e test.exe
foo
/home/Administrator/main.c:4
yunli.blog.51cto.com ~
$addr2line 0x40111B -f -e test.exe
foo
/home/Administrator/main.c:4
```

我们已经讲了对于 C 程序 addr2line 是如何使用的，那么对于 C++程序呢？现在假设我们有如图 2 所示的 C++代码。

main.cpp

```
#include <iostream>

using namespace std;

void foo ()
{
    cout << “The address of foo () is “ << hex << int (foo) << endl;
}

int main ()
{
    foo ();
    return 0;
}
```

图 2

采用 g++编译这一代码并运行之，我们得到如下的结果。

```
yunli.blog.51cto.com ~
$g++ -g main.cpp -o cpptest
yunli.blog.51cto.com ~
$./cpptest.exe
The address of foo () is 401150
```

使用 addr2line 的结果可以从下面得到。奇怪！怎么出现了乱码？应当是_foo，怎么变成了_Z3foov 了呢？

```
yunli.blog.51cto.com ~
$addr2line 0x401150 -f -e cpptest.exe
_Z3foov
/home/Administrator/main.cpp:6
```

其实，这是 C++语言的一个特点。在 GNU 工具集中存在 mangling 的这么一个称呼，而在 Windows 中称之为 decorating，也就是对 C++源程序中的函数名进行名字分裂的过程。为什么要进行名字分裂呢？还记得 C++语言中的重载（overload）吗?在 C++中，允许多个函数是重名的，但各个函数的输入参数必需是不一样的。那就有一个问题，当我们在实际的程序中调用这些重名的函数时，如何区分哪一个函数应当被调用呢？显然，应当根据不同的参数调用其相应的函数。 C++语言是在 C 语言的基础上实现的，因此，我们需要从 C 语言的角度来看这个问题。从 C 语言的角度来看，那么每个函数名必须是不一样的，即不存在重载的概念。为此，C++编译器的处理方法是，对于每一个函数根据其输入参数采用一定的编码方式，形成不同的 C 函数名，这一过程就是名字分裂过程。正如上面你所看到的，_Z3foov 其实就是 C++程序中 foo ()函数的名字分裂后的形式。如果要我们看这种“加了密”的函数名，那是很容易让人抓狂的。Addr2line 是否提供一定的选项来解决这一问题呢？有的，那就是--demangle 选项，但很遗憾的是，我发现它在我的 Cygwin 中不能正常工作，而在一台真正的 Linux 机器上，它却能正常工作。从下面在 Cygwin 中运行的结果可以看出，增加了--demangle 选项后，名字变成了 Z3foov。后面我们讲 nm 工具时，我们再看看如何用 nm 得知名字分裂后的形式所对应的真正的 C++程序中的函数。

```
yunli.blog.51cto.com ~
$addr2line 0x401150 --demangle=gnu-v3 -f -e cpptest.exe
Z3foov
/home/Administrator/main.cpp:6
```

使用 addr2line 的前提是程序文件中存在符号表，这通过给 GCC 增加一个-g 参数来达到这一目的，在讲解 objdump 时我们会更加的清楚为什么。如果没有加入-g 选项，那么 addr2line 并不能起作用。可能你会问，当我使用 GCC 或 g++进行代码优化时（比如使用-O2 选项）能否也使用-g 选项呢？在 GCC 和 g++中，代码优化与生成符号表是正交的，也就是说前面所问的问题的答案是“可以”。

除了这里所介绍的外，addr2line 还有其它的一些选项，你可以通过如下两条命令来得到其帮助信息，以便在具体使用时参照。

```
yunli.blog.51cto.com ~
$addr2line -h
yunli.blog.51cto.com ~
$man addr2line
```

4 ar

ar 是用来管理档案文件的，在嵌入式系统开发中， ar 主要是用来对静态库进行管理的。现在让我们先看一看一个静态库中到底有些什么。我们采用系统库文件/lib/libc.a 为例，对其采用 ar 的 x 参数进行解压操作，如下所示。

```
yunli.blog.51cto.com ~
$cp /lib/libc.a libc.a
yunli.blog.51cto.com ~
$ls
libc.a
yunli.blog.51cto.com ~
$ar x libc.a
yunli.blog.51cto.com ~
$ls
…显示结果有删减…
acltotext.o          fsetpos.o     initgroups.o     regcomp.o     tmpfile.o
chown.o              fstat.o       lchown.o         regerror.o    truncate.o
cygwin_attach_dll.o  ftello.o      libc.a           regexec.o
cygwin_crt0.o        ftruncate.o   libcmain.o       regfree.o
dll_entry.o          getegid.o     lseek.o          seekdir.o
```

如果你本来就是一名 C 程序开发者，相信你对于上面解压出来的.o 文件名一点也不陌生。每一个.o 文件差不多都能找到其文件名所对应的 C 库函数。采用 GNU 工具集进行开发时，一个静态库其实就是将所有的.o 文件打成一个档案包就好了。当然，为了使得连接的速度更快，我们往往还得生成内容索引，这可以通过给 ar 工具增加 s 参数来实现。

现在，为了示例我们是如何使用 ar 来生成静态库的，我们需要一些源程序文件。现在假设我们有 foo.c 和 bar.c 两个 C 程序文件，其分别实现了 foo ()和 bar ()两个函数，代码如图 3 所示。

foo.c

```
#include <stdio.h>

void foo ()
{
    printf (“This is foo ().\n”);
}
```

bar.c

```
#include <stdio.h>

void bar ()
{
    printf (“This is bar ().\n”);
}
```

图 3

我们希望将 foo ()和 bar ()函数做成一个库，为此先要将它们分别编译成.o 目标文件。有了目标文件之后，我们采用 ar 命令来生成 libmy.a 库，如下所示。其中，ar 的 c 参数表示创建一个档案文件，而 r 参数表示增加文件到所创建的库文件中，s 参数是为了生成库索引以提高连接速度。

```
yunli.blog.51cto.com ~
```

```
$gcc -c foo.c
yunli.blog.51cto.com ~
$gcc -c bar.c
yunli.blog.51cto.com ~
$ar crs libmy.a foo.o bar.o
```

现在你应当能在你的目录下看到一个 libmy.a 文件，这就是我们的静态库。下面我们需要验证这一库确实是可用的，我们采用图 4 所示的源程序来验证它。

main.c

```
extern void foo ();
extern void bar ();

int main ()
{
	foo ();
	bar ();
	return 0;
}
```

图 4

编译我们的验证程序，并与 libmy.a 进行连接，之后运行最终的可执行程序。从程序的运行结果你确实可以看出，我们的 libmy.a 是起作用的。

```
yunli.blog.51cto.com ~
$gcc main.c libmy.a -o mylib
yunli.blog.51cto.com ~
$./mylib.exe
This is foo ().
This is bar ().
```

采用 ar 的 t 参数可以查看一个档案文件中有些什么内容，下面的操作示例了这一参数的作用。

```
yunli.blog.51cto.com ~
$ar t libmy.a
foo.o
bar.o
```

如果想删除档案文件中的文件，我们可以用 d 参数。下面示例了用 d 参数删除 libmy.a 中的 foo.o 文件。从操作的最后结果来看，当执行完了 d 操作后，libmy.a 中只存在一个 bar.o 文件了。

```
yunli.blog.51cto.com ~
$ar d libmy.a foo.o
yunli.blog.51cto.com ~
$ar t libmy.a
bar.o
```

上面我们讲解了 ar 的几个参数，这里我们再总结一下。采用 c 参数用于创建一个档案文件，r 参数表示向档案文件中增加文件，t 参数用于显示档案文件中存在哪些文件，s 参数用于指示生成索引以加块查找速度，而 d 参数用于从档案文件中删除文件，最后 x 参数用于从档案文件中解压文件。应当说 ar 还有其它的一些命令参数，但这里所讲到的几个都是最为常用的。

5 nm

前面我们提到了 nm，现在我们就来看一看 nm 的功能。总的来说，nm 用于列出程序文件中的符号，符号是指函数或是变量名什么的。下面，我们来看一看图 2 所编译出来的程序当中有些什么符号。

```
yunli.blog.51cto.com ~
$nm -n test.exe
…显示结果有删减…
00401100 T _foo
0040111c T _main
004011b8 T _printf
00401378 T _calloc
00401388 T _realloc
00401398 T _free
004013a8 T _malloc
00402000 D __data_start__
00402000 D _register_frame_info_ptr
00402004 D _deregister_frame_info_ptr
0040200c D __data_end__
```

nm 所列出的每一行有三部分组成：第一列是指程序运行时的符号所对应的地址，对于函数则地址表示的是函数的开始地址，对于变量则表示的是变量的存储地址；第二列是指相应符号是放在哪一个段的；而最后面的一列则是指符号的名称。在前面我们讲解 addr2line 时，我们提到 addr2line 是将程序地址转换成这一地址所对应的具体函数是什么，而 nm 则是全面的列出这些信息。但是，nm 不具备列出符号所在的源文件及其行号这一功能，因此，我们说每一个工具有其特定的功能，在嵌入式系统的开发过程中我们需要灵活的运用它们。

对于 nm 列出的第二列信息，非常的有用，其意义在于可以了解我们在程序中所定义的一个符号（比如变量等等）是被放在程序的哪一个段的，对于程序段的概念和解释请参照《程序中的段》一文。下表列出了第二列将会出现的部分字母的含义，要参看所有字母的意思，请在你的开发环境中运行“man nm”。

字母	说　　明
A	表示符号所对应的值是绝对的且在以后的连接过程中也不会改变
B 或 b	表示符号位于未初始化的数据段（.bss 段）中
C	表示没有被初始化的共公符号
D 或 d	表示符号位于初始化的数据段（.data 段）中
N	表示符号是调试用的
p	表示符号位于一个栈回朔段中
R 或 r	表示符号位于只读数据段（.rdata 段）中
T 或 t	表示符号位于代码段（.text 段）中
U	表示符号没有被定义

为了更清楚的理解 nm 中的符号与我们所编写的程序的关系。我们需要看一看图 5 所示的显示源程序在采用只编译而不连接与采用连接的情况下，其所对应的 nm 输出结果有什么不同。

main.c

```
#include <time.h>

int global1;
```

```
int global2 = 3;

static int static_global1;
static int static_global2 = 3;

void foo ()
{
	static int internal1;
	static int internal2 = 3;
	time (0);
}

static void bar ()
{
}

int main ()
{
	int local1;
	int local2 = 3;

	foo ();
	return 0;
}
```

图 5

```
yunli.blog.51cto.com ~
$gcc -c -g main.c
yunli.blog.51cto.com ~
$nm -n main.o
                 U _time
00000000 T _foo
00000000 D _global2
00000000 b _internal1.2129
00000004 d _static_global2
00000005 t _bar
00000008 d _internal2.2130
0000000a T _main
00000010 C _global1
00000010 b _static_global1
```

对于上面的结果，因为我们只对 main.c 进行了编译，所以我们看到上面的结果中的地址有相同的 00000000，在实际连接完成了后，地址应当是不一样的。另外，还可以看出 time ()的符号是属于没有定义的，我们知道在 C 标准库 libc.a（而不是 main.o）中提供了 time ()的实现，所以这里显示没有定义是正常的。

从 nm 的输出信息，我们可以得出以下的结论：

- 不论一个静态变量是定义在函数内的，或是函数外的，其在程序段中的分配方式是一样的。如果这一静态变量是初始化好的，那么被分配（或存放）在.data 段中，否则被分配在.bss

段中。

- 非静态的全局变量，其所分配的段也是只与其是否被初始化有关。如果是初始化了的全局变量其将被分配在.data 段中，否则是.bss 段中。
- 函数无论是静态还是非静态的其总是被分配在.text 段的，但 T（t）的大小写表示了这一符号所对应的函数是否是静态函数。
- 函数内的局部变量并不被分配在.data、.bss 和.text 段中，那分配在哪呢？在栈上，采用 nm 看不出这一信息。

此外，需要注意的是其中的 global1 其类型是 C（公共符号），在后面进行连接以后其类型将会有所改变（变成 B）。之所以出现这一问题，是因为 C 语言中允许在多个文件中定义同样名称和类型的没有初始化的全局变量。接下来我们看一看连接以后的 nm 结果是什么。如下所示。

```
yunli.blog.51cto.com ~
$gcc -g main.c -o test
yunli.blog.51cto.com ~
$nm -n test.exe
…显示结果有删减…
00401100 T _foo
00401114 t _bar
00401119 T _main
004011b4 T _time
0040200c D _global2
00402010 d _static_global2
00402014 d _internal2.1860
00404030 b _internal1.1859
00404040 b _static_global1
00404060 B _global1
```

显然，最大的变化是所有的符号都有了具体的地址，而 global1 的分配空间也变成了 B（.bss 段），time ()从无定义变成了分配在 T（.text 段）中。

上面我们看了 C 程序是如何用 nm 来查看符号的，那么 C++有什么不同呢？假设我们有如图 6 所示的 C++源程序。

main.c

```
class thread_t
{
public:
    thread_t () {};
private:
    int thread_id_;
};
thread_t g_thread;
```

图 6

对于图 6 的程序我们只对其编译但不连接，然后用 nm 看一下有些什么输出，如下所示。对于这次的结果我们一点都不意外了，因为在 addr2line 这一章节中，我们知道了 C++语言中的名字分裂。

```
yunli.blog.51cto.com ~
$g++ -g -c main.cpp
yunli.blog.51cto.com ~
```

```
$nm -n main.o
…显示结果有删减…
00000000 t __Z41__static_initialization_and_destruction_0ii
00000000 T __ZN8thread_tC1Ev
00000000 B _g_thread
00000024 t __GLOBAL__I_thread
…显示结果有删减…
```

为了使输出的 C++符号更具可读性，我们可以采用 nm 中的--demangle 选项。显然，采用了--demangle 选项后，其输出的符号可读性有了极大的提高。

```
yunli.blog.51cto.com ~
$nm -n --demangle=gnu-v3 main.o
…显示结果有删减…
00000000 t __static_initialization_and_destruction_0(int, int)
00000000 T thread_t::thread_t()
00000000 B g_thread
00000024 t global constructors keyed to g_thread
…显示结果有删减…
```

至此，我们了解了 nm 工具的主要功能。同样，别忘了看一看其帮助信息以便在合适的时候使用其它的选项来达到所需的目的。

6 objdump

objdump 可以用来查看目标程序中的段信息和调试信息，也可以用来对目标程序进行反汇编。我们知道程序是由多个段组成的，比如.text 是用来放代码的、.data 是用来放初始化好的数据的、.bss 是用来放未初始化好的数据的，等等。在嵌入式系统的开发过程中，我们有时需要知道所生成的程序中的段信息来分析问题。比如，我们需要知道其中的某个段在程序运行时，共起始地址是什么，或者，我们需要知道正在运行的程序中是否存在调试信息等等。

下面是使用 objdump 的--h 选项来查看程序中的段信息，练习用的程序如前面的图 5，这里假设你已将其编译成了可执行文件 test.exe。

```
yunli.blog.51cto.com ~
$objdump -h test.exe

test.exe:     file format pei-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000458  00401000  00401000  00000400  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE, DATA
  1 .data         00000018  00402000  00402000  00000a00  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .rdata        00000044  00403000  00403000  00000c00  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .bss          00000090  00404000  00404000  00000000  2**3
                  ALLOC
  4 .idata        000001a4  00405000  00405000  00000e00  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

```
  5 .debug_aranges      00000020  00406000  00406000  00001000  2**0
                        CONTENTS, READONLY, DEBUGGING
  6 .debug_pubnames     00000071  00407000  00407000  00001200  2**0
                        CONTENTS, READONLY, DEBUGGING
  7 .debug_info         00000427  00408000  00408000  00001400  2**0
                        CONTENTS, READONLY, DEBUGGING
  8 .debug_abbrev       000000ed  00409000  00409000  00001a00  2**0
                        CONTENTS, READONLY, DEBUGGING
  9 .debug_line         000000b3  0040a000  0040a000  00001c00  2**0
                        CONTENTS, READONLY, DEBUGGING
 10 .debug_frame        0000006c  0040b000  0040b000  00001e00  2**2
                        CONTENTS, READONLY, DEBUGGING
 11 .debug_loc          0000009b  0040c000  0040c000  00002000  2**0
                        CONTENTS, READONLY, DEBUGGING
```

前面的 0 到 3 段是我们非常熟悉的。需要说明的是，对于.rdata，其中放的是我们程序中定义的只读初始化好的数据，在程序中采用 const 进行修饰的变量就被存放在这一段的。从 objdump 的输出信息还可以看出每一个段的大小，以及当这一程序运行时其所在的地址。对于地址，你可能注意到了存在 VMA（Virtual Memory Address，虚拟内存地址）和 LMA（Load Memory Address，加载内存地址）。对于每一个可加载的（loadable）或是可以重新分配的（allocatable）的段，其都存在一个 VMA 和一个 LMA，简单说来 VMA 指示的是段在程序运行时的开始地址，而 LMA 是指段的存放首地址。在大多数情况下 VMA 和 LMA 是一样的，尤其是采用 boot loader 的嵌入式系统，从我的使用经验来看，这一点不用太关心。由于绝大部分的嵌入式系统都不使用虚拟内存，所以 VMA 地址就是系统的实地址。

此外，我们还可以从 objdump 输出的信息中看出，其还输出 File off 信息，这一信息是指示每一个段在代码文件中的存储位置。对于 boot loader 来说，就是要通过 File off 信息从文件中读出相应段的内容，然后是将内容写到 VMA 所对应的地址块上。Align 指示了每一个段的对齐字节数是多少，对于为什么要进行字节对齐，读者或许可以参照《C 语言中一个字节对齐问题的分析》一文，其中对这一问题有所阐述。

除了这些信息，你还可以看出每一个段的属性，比如 READONLY、ALLOC 等等。在 objdump 列出的段信息中，你还可以看到很多段是以“.debug_”开头的，这些段是调试时需要使用到的，其中存储了我们程序中每一个符号的调试所需信息，以便在我们采用如 GDB 或是其它的调试工具进行调试时使用。这些调信息采用了一定的编码格式，最为常用的格式是 DWARF（Debugging With Attributed Record Formats）。对于 DWARF，你可以从其官网上找到规范，网址是 www.dwarfstd.org。采用 objdump 也可以查看程序文件中的 DWARF 信息，这需要用到-W 参数。

```
yunli.blog.51cto.com ~
$objdump -W test.exe
…显示结果有删减…
<0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)
    < c>    DW_AT_producer      : GNU C 4.3.2 20080827 (beta) 2
    <2a>    DW_AT_language      : 1          (ANSI C)
    <2b>    DW_AT_name          : main.c
    <32>    DW_AT_comp_dir      : /home/Administrator
    <46>    DW_AT_low_pc        : 0x401100
    <4a>    DW_AT_high_pc       : 0x401142
    <4e>    DW_AT_stmt_list     : 0x0
```

```
…显示结果有删减…
<1><103>: Abbrev Number: 4 (DW_TAG_subprogram)
    <104>    DW_AT_external      : 1
    <105>    DW_AT_name          : foo
    <109>    DW_AT_decl_file     : 1
    <10a>    DW_AT_decl_line     : 10
    <10b>    DW_AT_low_pc        : 0x401100
    <10f>    DW_AT_high_pc       : 0x401114
    <113>    DW_AT_frame_base    : 0x0       (location list)
    <117>    DW_AT_sibling       : <0x14a>
```

从上面显示的信息来看，尽管我们不熟悉 DWARF 规范，但是我们可以看出，调试信息中记录了源程序所在的路径，源程序所编译出来的程序起始地址（DW_AT_low_pc 和 DW_AT_high_pc），以及每一个函数的程序起始地址等等。这也就是为什么采用 addr2line 能将一个程序地址转换成其所对应的函数或是源程序所在的行号的原因。总而言之，所有我们在调试时能查看的信息，都采用 DWARF 格式放在调试段中，当然，运行时的值不可能存放在调试段中。我们也不难猜出，当调试器得到一个 PC（Program Counter，程序计数指针）值时，可以通过 DWARF 格式的调试信息反向的查出其所对应的函数是什么，当然，还可以具体定位到其所对应的源程序代码行。

采用-d 选项可以显示程序文件的汇编代码，下面是采用-d 选项所显示的 test.exe 的内容。

```
yunli.blog.51cto.com ~
$objdump -d test.exe
…显示结果有删减…
00401100 <_foo>:
  401100:       55                      push   %ebp
  401101:       89 e5                   mov    %esp,%ebp
  401103:       83 ec 08                sub    $0x8,%esp
  401106:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  40110d:       e8 a2 00 00 00          call   4011b4 <_time>
  401112:       c9                      leave
  401113:       c3                      ret
```

从显示的汇编来看，foo ()函数的起始和终止地址分别是 0x401100 和 0x401113，与前面所显示的 DWARF 格式的调试信息相对照，的确是与之对应的。我们除了可以看出汇编代码外，还可以看出汇编代码所对应的机器指令是什么。

在使用-d 选项进行反汇编时，还有一个非常有用的选项是-S，其用途是告诉 objdump 在反汇编时同时显示 C/C++源程序和与之对应的汇编代码，我们看看采用这一选项后的结果有什么不同。

```
yunli.blog.51cto.com ~
$objdump -S -d test.exe
…显示结果有删减…
00401100 <_foo>:

static int static_global1;
static int static_global2 = 3;

void foo ()
{
  401100:       55                      push   %ebp
```

```
  401101:       89 e5                   mov     %esp,%ebp
  401103:       83 ec 08                sub     $0x8,%esp
    static int internal1;
    static int internal2 = 3;
    time (0);
  401106:       c7 04 24 00 00 00 00    movl    $0x0,(%esp)
  40110d:       e8 a2 00 00 00          call    4011b4 <_time>
}
  401112:       c9                      leave
  401113:       c3                      ret
```

采用将汇编与源代码相结合显示的方式，有助于我们去了解高级语言的语法从汇编语言的角度是如何实现的。与 nm 工具相似的是，我们也可以运用--demangle 来帮助提高 C++程序在采用-S 选项进行反汇编时的可读性。

采用-f 选项可以显示目标文件的头信息。其中最要注意的是 start address，其指示了这一程序被执行时的入口地址是什么。比如，从显示可以看出 test.exe 的入口地址 0x00401000。对于嵌入式系统，当 boot loader 加载完程序后，就会调转到 start address 运行被加载的程序。

```
yunli.blog.51cto.com ~
$objdump -f test.exe

test.exe:       file format pei-i386
architecture: i386, flags 0x0000013a:
EXEC_P, HAS_DEBUG, HAS_SYMS, HAS_LOCALS, D_PAGED
start address 0x00401000
```

另一个非常有用的选项是-s，将它与-j 参数配合使用能查看某一个段中的具体内容。下面示例了，如何查看 test.exe 中.data 段的内容。

```
yunli.blog.51cto.com ~
$objdump -s -j .data test.exe

test.exe:       file format pei-i386

Contents of section .data:
 402000 00000000 00000000 00000000 03000000    ................
 402010 03000000 03000000                      ........
```

除了这里所说到的 objdump 中的常用选项外，objdump 还有很多其它的选项，读者同样在需要时可以参考帮助信息获取帮助。

7 objcopy

objcopy 的功能非常的强大，它可以对最后生成的程序文件进行一定的编辑。先来看一看采用 objcopy 如何生成一个只包含.text 段的目标文件。所需用到的练习程序如图 5 所示，这里假设你已将其编译成了可执行文件 test.exe。

```
yunli.blog.51cto.com ~
$objcopy -j .text test.exe onlytext.exe
```

你可以看出通过-j 参数我们可以指定哪一个段是我们所需要抽取的。为了验证最后生成的 onlytext.exe 是否是我们所期望的，我们可以用 objdump 命令来查看其段信息。从下面的输出结果

来看，onlytext.exe 中确实只包含一个.text 段。

```
yunli.blog.51cto.com ~
$objdump -h onlytext.exe

onlytext.exe:     file format pei-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000458  00401000  00401000  00000200  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE, DATA

yunli.blog.51cto.com ~
$objdump -f onlytext.exe

onlytext.exe:     file format pei-i386
architecture: i386, flags 0x00000132:
EXEC_P, HAS_SYMS, HAS_LOCALS, D_PAGED
start address 0x00401000
```

如果要指定多个段需要拷贝，比如，我们希望最后产生的 onlytext.exe（或许名字不应当再叫 onlytext 了）包含.text、.data 或是.bss 段，那么我们可以使用多个-j 参数的方法。

```
yunli.blog.51cto.com ~
$objcopy -j .text -j .data -j .bss test.exe onlytext.exe
```

与-j 参数相反的是，采用-R 可以删除一个段，我们看一看如果在 test.exe 中去除.text 段从而生成一个新的 notext.exe 程序。同样地，我们可以采用 objdump 工具查看 notext.exe 中是否存在.text 段。

```
yunli.blog.51cto.com ~
$objcopy -R .text test.exe notext.exe
yunli.blog.51cto.com ~
$objdump -h notext.exe

notext.exe:     file format pei-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .data         00000018  00402000  00402000  00000400  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  1 .rdata        00000044  00403000  00403000  00000600  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .bss          00000090  00404000  00404000  00000000  2**3
                  ALLOC
  3 .idata        000001a4  00405000  00405000  00000800  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  4 .debug_aranges 00000020  00406000  00406000  00000a00  2**0
                  CONTENTS, READONLY, DEBUGGING
  5 .debug_pubnames 00000071  00407000  00407000  00000c00  2**0
                  CONTENTS, READONLY, DEBUGGING
```

```
  6 .debug_info   00000427  00408000  00408000  00000e00  2**0
                  CONTENTS, READONLY, DEBUGGING
  7 .debug_abbrev 000000ed  00409000  00409000  00001400  2**0
                  CONTENTS, READONLY, DEBUGGING
  8 .debug_line   000000b3  0040a000  0040a000  00001600  2**0
                  CONTENTS, READONLY, DEBUGGING
  9 .debug_frame  0000006c  0040b000  0040b000  00001800  2**2
                  CONTENTS, READONLY, DEBUGGING
 10 .debug_loc    0000009b  0040c000  0040c000  00001a00  2**0
                  CONTENTS, READONLY, DEBUGGING
```

在嵌入式系统中，资源往往是有限的，有时为了减小程序文件所占用的空间（比如 FLASH），我们可以将程序中的调试信息去除，最为常用的是采用 strip 工具达到这一目的。但是，采用 objcopy 的--strip-debug 选项也可以达到同样的目的。现在让我们看一看，采用这一选项对 notext.exe 进行操作后的结果是什么。

```
yunli.blog.51cto.com ~
$objcopy --strip-debug notext.exe
yunli.blog.51cto.com ~
$objdump -h notext.exe

notext.exe:     file format pei-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .data         00000018  00402000  00402000  00000400  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  1 .rdata        00000044  00403000  00403000  00000600  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .bss          00000090  00404000  00404000  00000000  2**3
                  ALLOC
  3 .idata        000001a4  00405000  00405000  00000800  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

可以看出其中已经没有调试信息了，那我们再看一看此时运行 addr2line 命令的结果会是什么呢？下面的操作告诉了我们结果，正如前面所提到的，addr2line 需要依赖程序文件中的调试信息去得到地址所对应的函数名或是源程序的具体位置，当我们将目标文件中的调试信息去掉了以后，addr2line 自然就不能正常工作了。

```
yunli.blog.51cto.com ~
$addr2line -f 0x401100 -e notext.exe
??
??:0
```

objcopy 最为重要的功能就是能按照我们的需要抽取程序文件中的段。在有的嵌入式系统中，比如制作 boot loader 时就会需要用到 objcopy，以便将代码段抽取出来，然后使用烧写器将代码烧到系统的启动运行地址处（通常是一块 FLASH 中）。objcopy 还提供其它的一些有用的功能，比如，改变段的地址，但在我的工作经验中没有用过这些功能，所以在此也不多讲。你在需要时可以参看帮助，看看如何使用这些功能。使用这些功能的前提往往是我们要知道我们的嵌入式系统为什么要采用这种方式去改变地址，一旦明白了，用 objcopy 只是几个参数的问题。

8 ranlib

ranlib 的功能相对的简单，就是用于在档案文件中生成文件索引。前面在讲 ar 时我们也提到，ar 中的 s 参数也是具有同样的功能。当档案文件增加了索引后，对于其中文件的存取速度将更快。如果档案文件是一个静态库，那么我们在使用静态库进行连接时，其速度将会有所加快。

```
yunli.blog.51cto.com ~
$ranlib libmy.a
```

我们可以用 nm 加上一个-s 参数来查看档案文件中的索引信息。

```
yunli.blog.51cto.com ~
$nm -s libmy.a

Archive index:
_bar in bar.o
_foo in foo.o

bar.o:
00000000 b .bss
00000000 d .data
00000000 r .rdata
00000000 t .text
00000000 T _bar
         U _puts

foo.o:
00000000 b .bss
00000000 d .data
00000000 r .rdata
00000000 t .text
00000000 T _foo
         U _puts
```

9 readelf

Oops! 在写这一章节时发现，直接安装的 Cygwin 中的 GCC 不能生成 ELF 格式的程序文件，它所生成的是 PEI 格式的文件，这一格式是来自于 Microsoft 的 COFF （Common Object File Format，公共目标格式文件）。如果要使 Cygwin 中的 GCC 能生成 ELF 文件那么必须重新编译 binutils 工具集。

总体上说来，readelf 工具的功能 objdump 都有。如果需要使用它，你可以在你的系统中看一看 readelf 的帮助。

10 size

size 工具也很简单，就是列程序文件中各段的大小。在前面的章节中，我们看到，当使用 objdump 查看段信息时，除了这三个段还有.rdata 和.idata 两个段，其中.rdata 段被归类到.text 段中，而.idata 段被归类到.data 段中。下面是采用 size 工具所显示出的 test.exe 中的段大小信息。

```
yunli.blog.51cto.com ~
$size test.exe
    text        data        bss        dec        hex filename
    1180        444         144        1768       6e8 test.exe
```

如果采用-A 选项，size 将显示出与 objdump 相同的段和段大小信息。

```
yunli.blog.51cto.com ~
$size -A test.exe
test.exe  :
section                 size      addr
.text                   1112      4198400
.data                   24        4202496
.rdata                  68        4206592
.bss                    144       4210688
.idata                  420       4214784
.debug_aranges          32        4218880
.debug_pubnames         113       4222976
.debug_info             1063      4227072
.debug_abbrev           237       4231168
.debug_line             179       4235264
.debug_frame            108       4239360
.debug_loc              155       4243456
Total                   3655
```

11 strings

strings 用于查看我们的程序文件中的可显示字符。先假设我们有所示的 string.c 源程序。

string.c

```
#include <stdio.h>

#define VERSION       “2.3.7”

#define PASSWORD    “admin”

const char *get_password ()
{
      return PASSWORD;
}

int main ()
{
      printf (“Version: %s\n”, VERSION);
      printf (“Password: %s\n”, get_password ());
      return 0;
}
```

图 7

编译这一程序并运行它，输出如下。你可以看到这个程序输出的版本和密码信息。

```
yunli.blog.51cto.com ~
```

```
$gcc -g string.c string.exe
yunli.blog.51cto.com ~
$./string.exe
Version: 2.3.7
Password: admin
```

现在我们用 strings 工具来看一看我们的 string.exe 中有些什么可显示的字符串。

```
yunli.blog.51cto.com ~
$strings string.exe
$h0@
$h0@
$L0@
$Y0@
%|P@
C$0@@
C@`@@
$l0@
cyggcc_s-1.dll
__register_frame_info
__deregister_frame_info
admin
2.3.7
Version: %s
Password: %s
__main
_impure_ptr
calloc
cygwin_internal
dll_crt0__FP11per_process
free
malloc
printf
realloc
GetModuleHandleA
GetProcAddress
cygwin1.dll
KERNEL32.dll
```

你如果仔细看会发现，版本信息和密码信息都可以从 strings 的输出结果中找到。还有，你可以发现很多的函数名也在其中，这告诉我们什么呢？程序文件中的很多信息，是可以通过 strings 看到的，原因是这些信息都是放在.data 段（或是.rdata）中的，比如，我们在 C/C++程序中使用__FILE__宏时，就会在.rdata 段中生成函数名字符串。也就是说，即使你用 strip 将程序文件中的调试信息都去掉，你仍然可以通过 strings 看到这些信息。当然，如果你的程序中没有去除调试信息，那么，所有的文件名信息也可以采用前面说到的 nm 工具看出。

如何想将别人设计的程序（或库）作为你的程序的一部分，而你又不想让别人知道，那你得小心了，因为通过 strings 可以看到别人代码的一些信息。此外，如果程序中想定义密钥，那最好不要用字符串，或说不要直接用定义的字符串作为密钥。而是，因当采用一定的算法对这一可见字符串

进行加工，从而让别人即使是从 strings 的输出结果中看到了这一信息，也无法猜出密钥。

strings 对于我们调试也是很有用的。比如，你发布了一个软件，但是在现场你并不知道其版本是多少，因为你的程序没有提供人机接口方法去显示版本信息，但是你知道代码中定义了版本信息。知道了 strings 后，你遇到这种情况就有方法了，只需要用 strings 看一看就可以了。

由于 strings 是输出.data 段中的字符串信息的，因此，我们可以想像 strings 工具与具体的处理器是无关的。也就是说，我们可以用在 x86 上运行的 strings 程序去查看在 PowerPC 上运行的程序中的字符串。

12 strip

strip 的功能也相对的简单，主要用于去除程序文件中的调试信息以便减小文件的大小。对于 strip 的功能，其与 objcopy 带--strip-debug 参数时的功能是一样的，这我们前面也有提及。strip 所具有的功能，objcopy 也都有。

```
yunli.blog.51cto.com ~
$strip test.exe
```

致读者

如果你觉得本文的哪些地方需要改进或是存在一些不明白的地方，请点击这里并留言。如果你想参与讨论嵌入式系统开发相关的话题，请加入技术圈（g.51cto.com/UltraEmbedded）。

修订历史

日 期	修 订 说 明
2009-07-28	新文档
2009-07-29	新增了关于 ar 的章节
2009-07-30	新增了关于 nm 的章节
2009-07-31	新增了关于 objdump 的章节
2009-08-01	新增了关于 objcopy 的章节
2009-08-02	1) 新增了关于 ranlib、readelf、size、strings 和 strip 的五个章节 2) 修订了不少语句不通和错别字问题 3) 从文章全局的角度使用了更为一致的用词，比如，引入***程序文件***一词，指出其是目标文件、可执行文件和库文件的一个总称
2009-08-05	增加了参考资料的引用、增加了 objdump 的-s 参数的说明以及修订了一些错误