GNU compiler及binary utilities简介*

戴雨文

2003年11月26日

1 各个部分的职能

通常为了把一个应用程序从源文件转变为可执行的二进制代码,需要以下三个部分:

1. 编译器
2. (目标代码)连接器
3. 程序库

其中编译器部分还可细分为预处理器,C编译器和汇编编译器等. GCC的功能是C编译器[1]. Binutils 最重要的成员是汇编编译器和连接器,还包括一些二进制代码工具. 程序库通常是C或C++标准库. 注意这三部分是彼此独立的,也就是说,GCC并不是非要Binutils中的工具,也可以使用其它汇编编译器和连接器,也可以使用其它C程序库.

2 使用Binutils中的工具

Binutils中的工具可以帮助我们诊断许多问题.

2.1 readelf

readelf用来察看elf文件的内容. 用-a选项可以看见大部分内容:

```
arm-elf-readelf -a a2.elf
```

2.2 objdump

objdump的一个重要作用是反汇编目标文件:

*本文用TEX排版.

[1]GCC比较特殊,还可以作为编译器和连接器的驱动器.

```
arm-elf-objdump -S a2.o

a1.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <main>:
#include <stdio.h>
#include "uart_ev4510.h"

void main (void)
{
   0:   e1a0c00d        mov     ip, sp
   4:   e92dd800        stmdb   sp!, {fp, ip, lr, pc}
   8:   e24cb004        sub     fp, ip, #4      ; 0x4
   c:   e24dd008        sub     sp, sp, #8      ; 0x8
        int i ,j;

        i = 2;
  10:   e3a03002        mov     r3, #2  ; 0x2
  14:   e50b3010        str     r3, [fp, -#16]
        j = i * i;
  18:   e51b2010        ldr     r2, [fp, -#16]
  1c:   e51b3010        ldr     r3, [fp, -#16]
  20:   e0030392        mul     r3, r2, r3
  24:   e50b3014        str     r3, [fp, -#20]

        dev_uart_init (0, 0, 0);  /* initialise uart */
  28:   e3a00000        mov     r0, #0  ; 0x0
  2c:   e3a01000        mov     r1, #0  ; 0x0
  30:   e3a02000        mov     r2, #0  ; 0x0
  34:   ebffffe        bl      0 <main>
        crtio (19200);            /* change baund rate */
  38:   e3a00c4b        mov     r0, #19200      ; 0x4b00
  3c:   ebffffe        bl      0 <main>

        printf ("helo, world\n");
  40:   e59f0004        ldr     r0, [pc, #4]    ; 4c <main+0x4c>
  44:   ebffffe        bl      0 <main>
}
  48:   e91ba800        ldmdb   fp, {fp, sp, pc}
  4c:   00000000        andeq   r0, r0, r0
```

2.3 objcopy

objcopy有一个很重要的作用是把代码从elf文件中抽取出来,形成可执行的机器码:

```
arm-elf-objcopy -O binary -R .comment -R .note -S a2.elf a2.bin
```

形成的结果文件a2.bin可以烧到flash或下载到内存中去.

2.4 nm

nm用来列出elf文件中使用到的symbol:

```
arm-elf-nm a1.o

         U crtio
         U dev_uart_init
00000000 T main
         U printf
```

T表示定义过的函数,U表示尚未定义的函数. 我们可以看出main在a1.o里定义了, 其它几个函数crtio,dev_uart_init和printf在a1.o中被引用,但找不到定义,它们可能在其它.o文件或程序库中定义. nm对诊断连接错误很有帮助.

2.5 连接器ld

连接器的主要作用是把一个或多个目标文件(程序库)转变为一个可执行程序. 符号的重定位(relocation)是它最重要的工作. 传给ld的主要参数有目标文件,-l 选项和-L选项. 例如:

```
ld -o crt0.o myapp main.o subs.o -lm -lc -L/usr/lib
```

注意-lc的写法,它表示要与lib$c$.a连接[2];同样,-lm表示要与数学库lib$m$.a连接. -L*path*指定库的搜索路径.

2.5.1 连接脚本

ld知道如何连接各段(section)以及各段的起始位置,因为这些信息在ld本身被编译的时候已经内置进ld了. 有时你可能需要自己来安排可执行文件中各段的分布. 你可以自己写一个连接脚本(link script),用-T选项传给ld. 这是一个简单的连接脚本:

```
MEMORY
{
        rom (rx) : ORIGIN = 0, LENGTH = 2M
        ram (!rx): ORIGIN = 0x1000000, LENGTH = 8M
}
```

[2]如果ld缺省支持动态连接库,首先寻找的是libc.so.

```
SECTIONS
{
        .text : {
                _ftext = . ;
                *( .text )
        }
        . = ALIGN( 4 );
        _etext  = .;
        PROVIDE(etext  = .);

        .data :
        AT (_etext)
        {
                *( .data )
        } >ram

        _edata = .;
        .bss : {
                *(.bss)
        }>ram
        . = ALIGN( 4 );
        _end = .;
}
```

它只指定了3个最重要的段:text,data和bss以及它们的位置. 定制的连接脚本在嵌入式系统中经常要用到,因为ld内置的脚本可能不符合需要.

2.5.2 部分连接

ld是连接器. 有时我们可以用它来‘‘部分连接’’几个文件以生成一个目标文件, 该目标文件以后还可以再次与其它目标文件连接. ‘‘部分连接’’在uClinux中的应用:

```
genromfs -d romfs -f romfs.img
ld -r -o romfs.o romfs.img
```

ld并不关心romfs.img的文件格式是什么. 但生成的romfs.o是ELF格式,它以后还会和其它.o文件连接最终生成linux.elf.

2.6 ar

ar通常用来制作库文件---即含有许多.o文件的.a文件.如果你想了解某个库文件的内容,可以

```
ar tv libabc.a
```

制作一个库文件也很简单:

```
ar rs libabc.a a.o b.o c.o
```

事实上你可以把任何文件打包成一个存档文件,ar并不关心a.o,b.o,c.o等的格式. 你还可以把存档文件里的文件解开:

```
ar x libabc.a a.o
```

3 GCC作为编译器和连接器的大门

3.1 统一的入口

GCC本身并不是编译器或连接器. 它可以作为预处理器,C编译器,汇编编译器和连接器的入口. 例如,你运行命令

```
gcc -o a1.o -c a1.c
```

的时候,gcc自动调用了预处理器cpp,汇编编译器as. 你也可以用gcc来编译汇编文件:

```
gcc -o a1.o -c a1.S
```

在Makefile中我们通常定义变量LD为gcc,用gcc来连接目标文件:

```
LD      = arm-elf-gcc
...
$(OUTPUT_NAME): $(OBJECTS)
        echo "Linking... "
        echo "Creating file $@..."
        $(LD) -o $@ $(STARTUP) $^ $(LDFLAGS)
```

用gcc作为编译器和连接器的入口的好处是:它会传一些额外的参数给编译器和连接器. 比如,通常程序会与标准C程序库相连接. 如果你用ld来连接,必须传给ld 许多参数,象crt0.o,-lc等等. 因为这些东西并不是生成可执行文件所必需的,因此这些信息不会内嵌在ld中. 如果你用gcc作为连接器,这些选项就会自动被加上去.

3.2 gcc的specs文件

gcc的上述行为是通过它的specs文件实现的. 我们打开specs文件[3],选取一些片断看一下:

```
*startfile:
 crti%O%s crtbegin%O%s crt0%O%s
```

此条信息告诉我们,startfile是crti.o,crtbegin.o和crt0.o. 在连接的时候它们会被置于你的应用程序的最前面.

```
*predefines:
-D__ELF__
```

[3]specs 存放在/usr/lib/gcc-lib/$(ARCH)-$(OS)/$(VERSION)下面.

__ELF__会被预定义. 因此如果你在程序中有如下语句:

```
#ifdef __ELF__
do_something();
#endif
```

do_something肯定会被编译进代码.

```
*link:
%{mbig-endian:-EB} -X
```

如果命令行中有形如-mbig-endian的参数,就传给连接器‘‘-EB -X’’的参数

```
*lib:
%{!shared:%{g*:-lg} %{!p:%{!pg:-lc}}%{p:-lc_p}%{pg:-lc_p}}
```

连接的时候,会根据不同的选项,添加库libc或libc_p等.

```
*link_command:
%{!fsyntax-only:%{!c:%{!M:%{!MM:%{!E:%{!S:    %(linker) %l %X %{o*}
%{A} %{d} %{e*} %{m} %{N} %{n} %{r} %{s} %{t}    %{u*} %{x} %{z} %{Z}
%{!A:%{!nostdlib:%{!nostartfiles:%S}}}    %{static:} %{L*}
%(link_libgcc) %o
%{!nostdlib:%{!nodefaultlibs:%(link_gcc_c_sequence)}}
%{!A:%{!nostdlib:%{!nostartfiles:%E}}} %{T*} }}}}}}
```

这是连接的规则,比较复杂. 不过我们可以看出,gcc为我们额外做了许多事情.

4 制作交叉编译器

4.1 一些基本概念

制作交叉编译器的时候,GCC有所谓的target,host和build的概念. 交叉编译器所生成的代码运行在target上面,交叉编译器本身运行在host上,而制作交叉编译器的机器是build. 因此我们可以在i386-pc-linux系统上制作一套在cygwin上运行的交叉编译器,该编译器生成在ARM处理器上运行的代码.

4.2 准备工作

我们将在cygwin环境下制作一套arm-elf的交叉编译器. 我们需要的软件包有: GCC, Binutils和libc. libc可以有多种选择,我们在此选用newlib. 首先把下载好的软件包解开:

```
mkdir c:/build
cd c:/build
tar -jxf binutils-2.13.1.tar.bz2
tar -zxf gcc-3.2.3.tar.gz
tar -zxf newlib-1.11.0.tar.gz
```

目录binutils-2.13.1,gcc-3.2.3和newlib-1.11.0会生成. 这三个软件包都建议源文件目录和编译目录分开. 因此我们要另建三个目录:

```
mkdir build-bin build-gcc build-newlib
```

4.3 开始编译

制作交叉编译器的步骤是:

1. 编译binutils
2. 编译一个最简单的gcc
3. 编译libc
4. 再次编译gcc,生成功能完整的gcc

4.3.1 编译binutils

编译binutils很简单:

```
cd /cygdrive/c/build/build-bin

/cygdrive/c/build/binutils-2.13.1/configure --target=arm-elf \
--prefix=/cygdrive/c/bar --nfp

make all install
```

4.3.2 编译最简gcc

然后我们编译一个最简的gcc,它只能编译C程序:

```
cd /cygdrive/c/build/build-gcc

/cygdrive/c/build/gcc-3.2.3/configure --target=arm-elf \
--prefix=/cygdrive/c/bar --with-newlib --without-headers \
--enable-languages=c --disable-threads --nfp

make all install
```

4.3.3 编译newlib

再用刚才编译好的gcc和binutils编译newlib:

```
cd /cygdrive/c/build/build-newlib

CFLAGS=-O2 CXXFLAGS=-O2 /cygdrive/c/build/newlib-1.11.0/configure
--target=arm-elf --prefix=/cygdrive/c/bar \
--srcdir=/cygdrive/c/build/newlib-1.11.0 --nfp
```

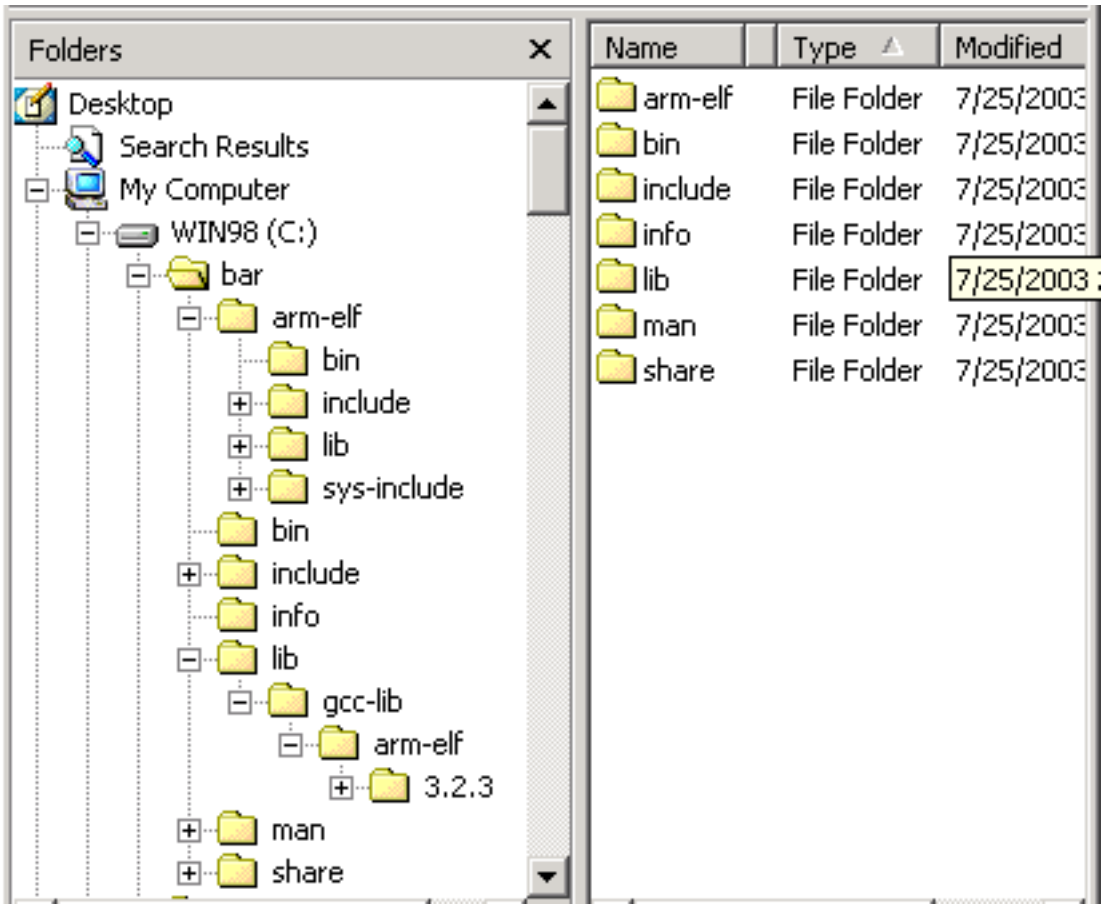

图 1: 交叉编译器的目录结构

```
make all install \
        CC_FOR_TARGET=/cygdrive/c/bar/bin/arm-elf-gcc \
        AS_FOR_TARGET=/cygdrive/c/bar/bin/arm-elf-as \
        LD_FOR_TARGET=/cygdrive/c/bar/bin/arm-elf-ld \
        AR_FOR_TARGET=/cygdrive/c/bar/bin/arm-elf-ar \
        RANLIB_FOR_TARGET=/cygdrive/c/bar/bin/arm-elf-ranlib
```

4.3.4 再次编译gcc

最后重新编译gcc,这次它充分利用了刚编好的newlib,可以支持C之外的其它语言了:

```
cd /cygdrive/c/build/build-gcc
```

```
/cygdrive/c/build/gcc-3.2.3/configure --target=arm-elf \
--prefix=/cygdrive/c/bar --with-newlib \
--with-headers=/cygdrive/c/build/newlib-1.11.0/newlib/libc/include \
--enable-languages=c++ --disable-threads --nfp

make all install
```

这样我们的交叉编译器就做好了,它生成在目录c:/bar里. 如第 8页的图 1.