嵌入式操作系统

2 GNU（交叉）开发工具链简介

陈香兰（xlanchen@ustc.edu.cn）

计算机应用教研室@计算机学院
嵌入式系统实验室@苏州研究院
中国科学技术大学

Fall 2014

November 28, 2014

Outline

“工欲善其事，必先利其器”

——《论语》

Outline

Outline

Outline

交叉开发

- 本地开发 vs 交叉平台开发
  1. 本地开发：
     一般软件的开发属于本地开发，即
     开发软件的系统与运行软件的系统是相同的。
  2. 交叉平台开发：
     本课程所涉及到的嵌入式系统开发属于交叉平台开发，即
     开发软件的系统与运行软件的系统不同。

- 交叉开发平台
  - 主机：
    开发软件的平台，称为主机，往往是通用电脑；
  - 目标机：
    运行软件的平台，称为目标机，在这里是嵌入式系统。

嵌入式交叉开发工具

- 掌握嵌入式开发工具的使用是进行嵌入式开发的前提条件之一
- 与主流开发工具类似，嵌入式交叉开发工具也包括
  - 编译器
    即能够把一个源程序编译生成一个可执行程序的软件
  - 调试工具
    即能够对执行程序进行源码或汇编级调试的软件
  - 软件工程工具
    用于协助多人开发或大型软件项目的管理的软件

GNU tools

- GNU tools和其他一些优秀的开源软件可以完全覆盖上述软件开发工具。
- 为了更好的开发嵌入式系统，需要熟悉如下一些软件
  - GCC——GNU编译器集
  - Binutils——辅助GCC的主要软件
  - Gdb——调试器
  - make——软件工程工具
  - diff, patch——补丁工具
  - CVS——版本控制系统
  - ...

Outline

Outline

GCC——The GNU Compiler Collection

- 不仅仅是C语言编译器
- 目前，GCC可以支持多种高级语言，如
  - C、C++
  - ADA
  - Objective-C、Objective-C++
  - JAVA
  - Fortran
  - PASCAL

GCC下的工具

- GCC下的工具包括
  - cpp — 预处理器
    GNU C编译器在编译前自动使用cpp对用户程序进行预处理
  - gcc — 符合ISO等标准的C编译器
  - g++ — 基本符合ISO标准的C++编译器
  - gcj — GCC的java前端
  - gnat — GCC的GNU ADA 95前端
  - 等等

GNU Tools—gcc

- gcc
  是一个强大的工具集合，它包含了预处理器、编译器、汇编器、链接器等组件。它会在需要的时候调用其他组件。输入文件的类型和传递给gcc的参数决定了gcc调用具体的哪些组件。
- 对于开发者，它提供的足够多的参数，可以让开发者全面控制代码的生成，这对嵌入式系统级的软件开发非常重要

gcc使用举例

源程序：gcctest.c

```
#include <stdio.h>

int main(void) {
      int i,j;
      i=0;j=0;
      i=j+1;
      printf( “Hello World!\n” );
      printf( “i=j+1=%d\n” ,i);
}
```

- 编译和运行

```
gcc  –o gcctest gcctest.c
./gcctest
```

gcc的工作过程

- 如果使用-v选项，则可以看到许多被隐藏的信息。例如

```
gcc -o gcctest gcctest.c -v
```

gcc的编译过程

- 一般情况下，c程序的编译过程为
  1. 预处理
  2. 编译成汇编代码
  3. 汇编成目标代码
  4. 链接

1、预处理

- 预处理：使用-E参数
  输出文件的后缀为“.cpp”

```
gcc -E -o gcctest.cpp gcctest.c
```

- 使用wc命令比较预处理后的文件与源文件，可以看到两个文件的差异

wc命令：wc gcctest.c gcctest.cpp

```
  8   12   117 gcctest.c
851 2096 17605 gcctest.cpp
859 2108 17722 总用量
```

关于wc命令

wc —help

用法：wc [选项]... [文件]...
  或：wc [选项]... —files0-from=F
Print newline, word, and byte counts for each FILE, and a total line if more than one FILE is specified. With no FILE, or when FILE is -, read standard input. A word is a non-zero-length sequence of characters delimited by white space.
The options below may be used to select which counts are printed, always in the following order: newline, word, character, byte, maximum line length.

-c, —bytes	print the byte counts
-m, —chars	print the character counts
-l, —lines	print the newline counts
—files0-from=文件	从指定文件读取以NUL 终止的名称，如果该文件被指定为”-”则从标准输入读文件名
-L, —max-line-length	显示最长行的长度
-w, —words	显示单词计数
—help	显示此帮助信息并退出
—version	显示版本信息并退出

2、编译成汇编代码

预处理文件→汇编代码

1. 使用-x参数说明根据指定的步骤进行工作，
   cpp-output指明从预处理得到的文件开始编译
2. 使用-S说明生成汇编代码后停止工作

```
gcc -x cpp-output -S -o gcctest.s gcctest.cpp
```

也可以直接编译到汇编代码

```
gcc -S gcctest.c
```

3、编译成目标代码

汇编代码→目标代码

```
gcc -x assembler -c gcctest.s
```

直接编译成目标代码

```
gcc -c gcctest.c
```

使用汇编器生成目标代码

```
as -o gcctest.o gcctest.s
```

4、编译成执行代码

目标代码→执行代码

```
gcc -o gcctest gcctest.o
```

直接生成执行代码

```
gcc -o gcctest gcctest.c
```

gcc的高级选项

- -Wall：打开所有的警告信息

```
gcc -o gcctest gcctest.c -Wall
```

```
gcctest.c: In function ‘main’:
gcctest.c:8:1: warning: control reaches end of non-void
function [-Wreturn-type]
}
^
```

gcc的高级选项

- 根据警告信息检查源程序

源程序：gcctest.c

```
#include <stdio.h>

int main(void){
        int i,j;
        i=0;j=0;
        i=j+1;
        printf( “Hello World!\n” );
        printf( “i=j+1=%d\n” ,i);
}
```

- Main函数的返回值为int
- 在函数的末尾应当添加返回一个值

gcc的高级选项

- 修改源程序

源程序(修改)：gcctest.c

```
#include <stdio.h>

int main(void){
        int i,j;
        i=0;j=0;
        i=j+1;
        printf( “Hello World!\n” );
        printf( “i=j+1=%d\n” ,i);
        return 0;
}
```

再次运行命令：

```
gcc -o gcctest gcctest.c -Wall
```

无上述警告信息。

优化编译

- 优化编译选项有：
  - -O0
    缺省情况，不优化
  - -O1
  - -O2
  - -O3
  - 等等

不同程度的优化
不同的优化内容

gcc的优化编译举例

源代码：mytest.c

```
#include <stdio.h>
#include <math.h>
int main(void) {
      int i,j;
      double k=0.0,k1,k2,k3;
      k1=k2=k3=1.0;
      for (i=0;i<50000;i++)
            for (j=0;j<50000;j++) {
                  k+=k1+k2+k3;
                  k1 += 0.5;
                  k2 += 0.2;
                  k3 = k1+k2;
                  k3 -= 0.1;
            }
      return 0;
}
```

gcc的优化编译举例

- 使用不同的优化选项，分别生成不同的可执行文件

```
gcc -O0 -o m0 mytest.c

gcc -O1 -o m1 mytest.c

gcc -O2 -o m2 mytest.c

gcc -O3 -o m3 mytest.c
```

gcc的优化编译举例

- 使用time命令统计程序的运行

time ./m0

```
real 0m17.068s
user 0m17.056s
sys 0m0.000s
```

time ./m1

```
real 0m1.720s
user 0m1.716s
sys 0m0.000s
```

time ./m2

```
real 0m0.001s
user 0m0.000s
sys 0m0.000s
```

time ./m3

```
real 0m0.001s
user 0m0.000s
sys 0m0.000s
```

gcc的优化编译举例

- 一个以前机器上的结果

```
[donger@donger gcctest]$ ls
gcctest.c  m0  m1  m2  m3  mytest.c
[donger@donger gcctest]$ time ./m3

real    0m2.756s
user    0m2.658s
sys     0m0.042s
[donger@donger gcctest]$ time ./m2

real    0m2.733s
user    0m2.643s
sys     0m0.037s
[donger@donger gcctest]$ time ./m1

real    0m1.829s
user    0m1.767s
sys     0m0.022s
[donger@donger gcctest]$ time ./m0

real    0m40.808s
user    0m39.632s
sys     0m0.337s
[donger@donger gcctest]$
```

Outline

GNU binutils I

binutils是一组二进制工具程序集，是辅助GCC的主要软件
主要包括

1. addr2line
   把程序地址转换为文件名和行号。
   在命令行中给它一个地址和一个可执行文件名，
   它就会使用这个可执行文件的调试信息指出在给出的地址上是
   哪个文件以及行号。
2. ar
   建立、修改、提取归档文件。
   归档文件是包含多个文件内容的一个大文件，其结构保证了
   可以恢复原始文件内容。
3. as
   GNU汇编器，主要用来编译GNU C编译器gcc输出的汇编文件，
   它将汇编代码转换成二进制代码，并存放到一个object文件中，
   该目标文件将由连接器ld连接

GNU binutils II

4. C++filt
   解码C++符号名，连接器使用它来过滤 C++ 和 Java 符号，防止重载函数冲突。
5. gprof
   显示程序调用段的各种数据。
6. ld
   是连接器，它把一些目标和归档文件结合在一起，重定位数据，并链接符号引用，最终形成一个可执行文件。
   通常，建立一个新编译程序的最后一步就是调用ld。
7. nm
   列出目标文件中的符号。
8. objcopy
   把一种目标文件中的内容复制到另一种类型的目标文件中。

GNU binutils III

9. objdump
   显示一个或者更多目标文件的信息。
   使用选项来控制其显示的信息。它所显示的信息通常只有编写编译工具的人才感兴趣。
10. ranlib
    产生归档文件索引，并将其保存到这个归档文件中。
    在索引中列出了归档文件各成员所定义的可重分配目标文件。
11. readelf
    显示elf格式可执行文件的信息。
12. size
    列出目标文件每一段的大小以及总体的大小。
    默认情况下，对于每个目标文件或者一个归档文件中的每个模块只产生一行输出。

GNU binutils IV

13. strings
    打印某个文件的可打印字符串，这些字符串最少4个字符长，也可以使用选项-n设置字符串的最小长度。
    默认情况下，它只打印目标文件初始化和可加载段中的可打印字符；对于其它类型的文件它打印整个文件的可打印字符，这个程序对于了解非文本文件的内容很有帮助。
14. strip
    丢弃目标文件中的全部或者特定符号。
15. libiberty
    包含许多GNU程序都会用到的函数，这些程序有：
    getopt, obstack, strerror, strtol 和 strtoul.
16. libbfd
    二进制文件描述库.
17. libopcodes
    用来处理opcodes的库，在生成一些应用程序的时候也会用到它，比如objdump.Opcodes是文本格式可读的处理器操作指令.

binutils开发工具使用举例

1. ar
2. nm
3. Objcopy
4. Objdump
5. readelf

1、ar

- ar用于建立、修改、提取归档文件(archive)。
  一个归档文件，是包含多个被包含文件的单个文件
  （也可以认为归档文件是一个库文件）。
- 被包含的原始文件的内容、权限、时间戳、所有者等
  属性都保存在归档文件中，并且在提取之后可以还原

使用ar建立库文件

源程序add.c

```
// add.c
int Add(int a, int b) {
        int result;
        result = a+b;
        return result;
}
```

源程序minus.c

```
// minus.c
int Minus(int a, int b) {
        int result;
        result = a-b;
        return result;
}
```

- Step1：使用命令

```
gcc -c add.c minus.c
```

生成add.o和minus.o两个目标文件

使用ar建立库文件

- step2：使用命令

```
ar rv libtest.a add.o minus.o
```

生成库文件libtest.a，并使用命令

```
sudo cp libtest.a /usr/lib/
```

复制到/usr/lib/目录下

```
ar rv libtest.a add.o minus.o
```

```
ar: 正在创建 libtest.a
a - add.o
a - minus.o
```

```
sudo cp libtest.a /usr/lib/
```

关于ar命令：

```
r    - replace existing or insert new file(s) into the archive
v    - be verbose
```

库文件使用举例：

- step1：在代码中使用Add和Minus函数

源代码test.c

```
#include <stdio.h>

int main(void)
{
        int a=8;
        int b=3;
        printf( “a=%d\tb=%d\n” ,a,b);
        int sum = Add(a,b);
        printf( “a+b=%d\n” ,sum);
        int cha = Minus(a,b);
        printf( “a-b=%d\n” ,cha);
        return 0;
}
```

库文件使用举例：

- step2：在编译时指定库文件。
  在链接时，使用“-l<name>”选项来指明库文件

```
gcc -o test test.c -ltest
```

- step3：运行

```
./test
a=8         b=3
a+b=11
a-b=5
```

2、nm

- nm的主要功能是列出目标文件中的符号。
  程序员可以使用此工具定位和分析执行程序和目标文件中的符号信息和它的属性

nm显示的符号类型

- A：符号的值是绝对值，并且不会被将来的链接所改变
- B：符号位于未初始化数据部分（BSS段）
- C：符号是公共的。公共符号是未初始化的数据。
  在链接时，多个公共符号可能以相同的名字出现。如果符号在其他地方被定义，则该文件中的这个符号会被当作引用来处理
- D：符号位于已初始化的数据部分
- T：符号位于代码部分
- U：符号未被定义
- ？：符号类型未知，或者目标文件格式特殊

nm使用举例

1. 分别查看add.o和minus.o中的符号

```
nm add.o minus.o
```

```
add.o:
00000000 T Add

minus.o:
00000000 T Minus
```

2. 重新编译test.c为test.o，然后查看test.o的符号

```
gcc -c test.c
nm test.o
```

```
         U Add
00000000 T main
         U Minus
         U printf
```

作业：使用nm列出可执行文件test的符号，与test.o进行比较

3、objcopy

- 可以将一种格式的目标文件内容进行转换，并输出为另一种格式的目标文件。
- 它使用GNU BFD(binary format description)库读/写目标文件，通过这个BFD库，objcopy能以一种不同于源目标文件的格式生成新的目标文件

```
objcopy -h
```

- 在makefile里面用

  -o binary

  选项来生成原始的二进制文件，即通常说的image文件

Objcopy使用举例

- step1：查看可执行文件test的文件类型相关信息

在ubuntu-14.04中，命令：file test

```
test: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked
(uses shared libs), for GNU/Linux 2.6.24, BuildID[sha1]=f2eaec4b4747af071d6b309f2d1
3cff079a9f1bb, not stripped
```

- step2：将test分别转换为srec格式的文件ts和bianry格式的文件test.bin

```
objcopy -O srec test ts

objcopy -O binary test test.bin
```

- step3：使用file命令查看ts和test.bin的文件类型信息

file ts

```
ts: Motorola S-Record; binary data in text format
```

file test.bin

```
test.bin: data
```

文件格式

- a.out：
  assembler and link editor output
  汇编器和链接编辑器的输出
- coff
  common object file format
  一种通用的对象文件格式
- ELF
  excutive linked file
  Linux系统所采用的一种通用文件格式，支持动态连接。
  ELF格式可以比COFF格式包含更多的调试信息
- Flat
  elf格式有很大的文件头，flat文件对文件头和一些段信息做了简化
  μClinux系统使用flat可执行文件格式
- SREC
  MOTOROLA S-Recoder格式（S记录格式文件）
- 等等

4、objdump

- 显示一个或多个目标文件的信息，由其选项来控制显示哪些信息。
- 一般来说，objdump只对那些要编写编译工具的程序员有帮助，但是我们通过这个工具可以方便的查看执行文件或者库文件的信息

Objdump使用举例

1. “-f”选项显示文件头的内容

objdump -f test

```
test： 文件格式 elf32-i386
体系结构：i386， 标志 0x00000112：
EXEC_P, HAS_SYMS, D_PAGED
起始地址 0x08048320
```

objdump -f ts

```
ts： 文件格式 srec
体系结构：UNKNOWN!， 标志 0x00000000：

起始地址 0x08048320
```

objdump -f test.bin

```
objdump: test.bin: 不可识别的文件格式
```

Objdump使用举例

2. “-d”选项进行反汇编

```
objdump -d add.o
```

```
add.o：  文件格式 elf32-i386

Disassembly of section .text:

00000000 <Add>:
   0:  55 push %ebp
   1:  89 e5 mov %esp,%ebp
   3:  83 ec 10 sub $0x10,%esp
   6:  8b 45 0c mov 0xc(%ebp),%eax
   9:  8b 55 08 mov 0x8(%ebp),%edx
   c:  01 d0 add %edx,%eax
   e:  89 45 fc mov %eax,-0x4(%ebp)
  11:  8b 45 fc mov -0x4(%ebp),%eax
  14:  c9 leave
  15:  c3 ret
```

5、readelf

- readelf： 显示一个或多个ELF格式的目标文件信息。

例如：readelf -h test

```
ELF 头：
 Magic：  7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
 Class:                             ELF32
 Data:                              2's complement, little endian
 Version:                           1 (current)
 OS/ABI:                            UNIX - System V
 ABI Version:                       0
 Type:                              EXEC (可执行文件)
 Machine:                           Intel 80386
 Version:                           0x1
 入口点地址：                       0x8048320
 程序头起点：                       52 (bytes into file)
 Start of section headers:          4428 (bytes into file)
 标志：                             0x0
 本头的大小：                       52 (字节)
 程序头大小：                       32 (字节)
 Number of program headers:         9
 节头大小：                         40 (字节)
 节头数量：                         30
 字符串表索引节头：                 27
```

Outline

GNU Toolchain—gdb

- Gdb = GNU debuger
- GNU tools中的调试器，功能强大
  - 设置断点
  - 监视、修改变量
  - 单步执行
  - 显示/修改寄存器的值
  - 堆栈查看
  - 远程调试

gdb使用举例

源代码bug.c

```
#include <stdio.h>
#include <stdlib.h>

static char buff[256];
static char* string;

int main(void){
      printf( “input a string:” );
      scanf( “%s\n” ,string);
      printf( “\n Your string is:%s\n” ,string);
      return 0;
}
```

编译并运行：

```
gcc -o bug bug.c
./bug
```

```
input a string:hello
段错误（核心已转储）
```

使用gdb调试bug

gdb bug

```
...
(gdb) run
Starting program: /home/xlanchen/workspace/gdb_test/bug
input a string:hello

Program received signal SIGSEGV, Segmentation fault.
_IO_gets (buf=0x0) at iogets.c:54
54 iogets.c: 没有那个文件或目录.
(gdb) list
49 in iogets.c
```

静态编译bug.c并调试

```
gcc -static -o bug bug.c
```

```
...
(gdb) run
Starting program: /home/xlanchen/workspace/gdb_test/bug
input a string:hello

Program received signal SIGSEGV, Segmentation fault.
0x0804f75a in gets ()
(gdb) where
#0 0x0804f75a in gets ()
#1 0x08048e66 in main ()
```

能否查看bug.c源代码？

使用gcc的-g参数生成调试信息

```
gcc -g -static -o bug bug.c
```

```
(gdb) run
Starting program: /home/xlanchen/workspace/gdb_test/bug
input a string:hello

Program received signal SIGSEGV, Segmentation fault.
0x0804f75a in gets ()
(gdb) list
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 static char buff[256];
5 static char* string;
6
7 int main(void){
8 printf(" input a string:" );
9 gets(string);
10 printf(" \n Your string is:%s\n" ,string);
(gdb) print string
$1 = 0x0 (gdb)
```

Outline

使用GNU make管理项目 GNU I

- make是一种代码维护工具，在使用GNU编译器开发大型应用时，往往要使用make管理项目。
  - 如果不使用make管理项目，就必须重复使用多个复杂的命令行维护项目和生成目标代码。
- Make通过将命令行保存到makefile中简化了编译工作。
- Make的主要任务是
  根据makefile中定义的规则和步骤，根据各个模块的更新情况，自动完成整个软件项目的维护和代码生成工作。
- Make可以识别出makefile中哪些文件已经被修改，并且在再次编译的时候只编译这些文件，从而提高编译的效率
  - Make会检查文件的修改和生成时间戳，如果目标文件的修改或者生成时间戳比它的任意一个依赖文件旧，则make就执行makefile文件中描述的相应命令，以便更新目的文件
  - 只更新那些需要更新的文件，而不重新处理那些并不过时的文件

使用GNU make管理项目 GNU II

- 特点：
  - 适合于支持多文件构成的大中型软件项目的编译、链接、清除中间文件等管理工作
  - 提供和识别多种默认规则，方便对大型软件项目的管理
  - 支持对多目录的软件项目进行递归管理
  - 对软件项目具有很好的可维护性和扩展性

makefile

- Makefile告诉make该做什么、怎么做
- makefile主要定义了
  1. 依赖关系
     即有关哪些文件的最新版本是依赖于哪些别的文件产生或者组成的
  2. 需要用什么命令来产生目标文件的最新版本
  3. 以及一些其他的功能

Makefile的规则

规则

- 一条规则包含3个方面的内容，
  1. 要创建的目标（文件）
  2. 创建目标（文件）所依赖的文件列表；
  3. 通过依赖文件创建目标文件的命令组

规则一般形式

```
target ... : prerequisites ...
<tab>command
<tab>...
<tab>...
```

例如

```
test:test.c;gcc -O -o test test.c
```

一个简单的makefile

```
edit : main.o kbd.o command.o display.o insert.o \
        search.o files.o utils.o
cc -o edit main.o kbd.o command.o display.o insert.o \
        search.o files.o utils.o

main.o : main.c defs.h
cc -c main.c

kbd.o : kbd.c defs.h command.h
cc -c kbd.c

command.o : command.c defs.h command.h
cc -c command.c

display.o : display.c defs.h buffer.h
cc -c display.c

insert.o : insert.c defs.h buffer.h
cc -c insert.c

search.o : search.c defs.h buffer.h
cc -c search.c

files.o : files.c defs.h buffer.h command.h
cc -c files.c

utils.o : utils.c defs.h
cc -c utils.c

clean :
rm edit main.o kbd.o command.o display.o insert.o \
        search.o files.o utils.o
```

Make的工作过程

- default goal
  - 在缺省的情况下，make从makefile中的第一个目标开始执行
- Make的工作过程类似一次深度优先遍历过程

Makefile 中的变量

- 使用变量可以
  - 降低错误风险
  - 简化makefile

例：objects变量（$(objects)）

```
objects = main.o kbd.o command.o \
          display.o insert.o search.o files.o utils.o
edit : $(objects)
       cc -o edit $(objects)
```

- 有点像环境变量
  - 环境变量在make 过程中被解释成make的变量
- 可以被用来
  - 贮存一个文件名列表。
  - 贮存可执行文件名。如用变量代替编译器名。
  - 贮存编译器FLAG

预定义变量

- Make使用了许多预定义的变量，如
  - AR
  - AS
  - CC
  - CXX
  - CFLAGS
  - CPPFLAGS
  - 等等

简化后的makefile文件

```
objects = main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o

edit : $(objects)
cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        rm edit $(objects)
```

内部变量

- $@扩展成当前规则的目的文件名
- $<扩展成依赖列表中的第一个依赖文件
- $^扩展成整个依赖列表（除掉了里面所有重复的文件名）
- 等等

- 不需要括号括住

例如：

```
CC = gcc
CFLAGS = -Wall -O -g
foo.o : foo.c foo.h bar.h
        $(CC) $(CFLAGS) -c $< -o $@
```

隐含规则（Implicit Rules）

- 内置的规则
- 告诉make当没有给出某些命令的时候，应该怎么办。
- 用户可以使用预定义的变量改变隐含规则的工作方式，如
  - 一个C编译的具体命令将会是：

```
$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c $< -o $@
```

设定目标（Phony Targets）

- 设定目标
  - 目标不是一个文件
  - 其目的是为了让一些命令得以执行
- 使用PHONY显式声明设定目标

```
.PHONY: clean
```

- 使用设定目标实现多个目的

```
all: prog1 prog2
```

典型的设定目标

- 设定目标也可以用来描述一些其他的动作。例如，想把中间文件和可执行文件删除，可以在 makefile 里设立这样一个规则：

```
clean:
        rm *.o exec_file
```

前提是没有其它的规则依靠这个‘clean’目标，它将永远不会被执行。但是，如果你明确的使用命令‘make clean’，make会把这个目标做为它的主要目标，执行那些 rm 命令

Makefile中的函数（Functions） I

- 用来计算出要操作的文件、目标或者要执行的命令
- 使用方法：

$(function arguments)

- 几个典型函数
  1. $(substfrom,to,text)

例如：

$(subst ee,EE,feet on the street)

相当于‘fEEt on the strEEt’

Makefile中的函数（Functions） II

2. $(patsubst pattern,replacement,text)

例如：

```
$(patsubst %.c,%.o,x.c.c bar.c)
```

相当于‘x.c.o bar.o‘

3. $(wildcard pattern)

例如：

```
$(wildcard *.c)
```

又如：

```
objects := $(wildcard *.o)
```

makefile中的条件语句

```
conditional-directive
      text-if-true
endif
```

or

```
conditional-directive
      text-if-true
else
      text-if-false
endif
```

四种条件语句

- ifeq...else...endif
- ifneq…else…endif
- ifndef…else…endif
- ifndef...else…endif

实际项目中的makefiles

找到Linux或者μClinux源代码中所有的makefile，分析它们的功能

Outline

GNU Tools—ld

- ld，The GNU Linker
  Linux上常用的链接器
- ld软件的作用是把各种目标文件（.o文件）和库文件链接在一起，并定位数据和函数地址，最终生成可执行程序
- gcc可以间接的调用ld，使用gcc的-Wl参数可以传递参数给ld
- 使用下面的命令可以列出ld常用的一些选项：

```
ld --help
```

ld使用举例 I

源程序

```
//hello.c

#include <stdio.h>
int main(void)
{
        printf( "\n\nHello World!\n\n" );
}
```

编译hello.c到hello.o

```
gcc -c hello.c
```

ld使用举例 II

链接（在ubuntu-14.04中）

```
ld -dynamic-linker /lib/ld-linux.so.2
/usr/lib/i386-linux-gnu/crt1.o /usr/lib/i386-linux-gnu/crti.o
/usr/lib/i386-linux-gnu/crtn.o hello.o -lc -o hello
```

运行

```
./hello
```

目标文件

- ld通过BFD库可以读取和操作coff、elf、a.out等各种执行文件格式的目标文件
  - BFD（Binary File Descriptor）
- 目标文件（object file）
  - 由多个节(section)组成，常见的节有：
    - ★ text节保存了可执行代码，
    - ★ data节保存了有初值的全局标量，
    - ★ bss节保存了无初值的全局变量。

使用objdump查看目标文件的信息

objdump – hhello.o

```
hello.o：  文件格式 elf32-i386

节：
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000017  00000000  00000000  00000034  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  0000004b  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  0000004b  2**0
                  ALLOC
  3 .rodata       0000000f  00000000  00000000  0000004b  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      00000025  00000000  00000000  0000005a  2**0
                  CONTENTS, READONLY
  5 .note.GNU-stack 00000000  00000000  00000000  0000007f  2**0
                  CONTENTS, READONLY
  6 .eh_frame     00000038  00000000  00000000  00000080  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

作业：比较hello

链接描述文件（ Linker script ）

- 可以使用链接描述文件控制ld的链接过程。

链接描述文件，command file

又称为链接脚本，Linker script

- 用来控制ld的链接过程
  - 描述各输入文件的各节如何映射到输出文件的各节
  - 控制输出文件中各个节或者符号的内存布局
- 使用的语言为：
  - The ld command language，链接命令语言

- ld命令的-T commandfile选项指定了链接描述文件名
  - 如果不指定链接描述文件，ld就会使用一个默认的描述文件来产生执行文件

作业：找到Linux或者μClinux中的链接描述文件并分析。

链接描述文件的命令

- 链接描述文件的命令主要包括如下几类：
  - 设置入口点命令
  - 处理文件的命令
  - 处理文件格式的命令
  - 其他

常用的命令

- 设置入口点

格式：

ENTRY(symbol)

  - 设置symbol的值为执行程序的入口点。

- ld有多种方法设置执行程序的入口点，确定程序入口点的顺序如下：
  - ld命令的-e选项指定的值
  - Entry(symbol)指定的值
  - .text节的起始地址
  - 入口点为0

常用的命令

包含其他filename的链接描述文件

```
INCULDE filename
```

指定多个输入文件名

```
INPUT(file,file,…)
```

常用的命令

指定输出文件的格式

```
OUTPUT_FORMAT(bfdname)
```

指定目标机器体系结构

```
OUTPUT_ARCH ( bfdname )
```

例如：

```
OUTPUT_ARCH(arm)
```

常用的命令

- MEMORY：
  这个命令在用于嵌入式系统的链接描述文件中经常出现，它描述了各

```
MEMORY
{
    name [(attr)]:ORIGIN = origin,LENGTH = len
    ...
}
```

例如：

```
MEMORY
{
        rom : ORIGIN=0x1000, LENGTH=0x1000
}
```

Memory举例

```
//标注嵌入式设备中各个内存块的地址划分情况
MEMORY
{
  //标注flash中断向量表的起始地址为0x01000000,长度为0x0400
  romvec: ORIGIN  = 0x01000000, LENGTH  = 0x0400
  //标注flash的起始地址为0x01000400,长度为0x011fffff-0x01000400
  flash: ORIGIN = 0x01000400, LENGTH = 0x011fffff-0x01000400
  //标注flash的结束地址在0x011fffff
  eflash : ORIGIN = 0x011fffff, LENGTH = 1
  ramvec:ORIGIN = 0x00000000, LENGTH = 0x0400
  ram : ORIGIN = 0x00000400, LENGTH = 0x0003ffff-0x00000400
  eram : ORIGIN = 0x0003ffff, LENGTH = 1
}
```

SECTIONS命令 I

- SECTIONS
  告诉ld如何把输入文件的各个节映射到输出文件的各个节中。
  - 在一个链接描述文件中只能有一个SECTIONS命令
- 在SECTIONS命令中可以使用的命令有三种：
  - 定义入口点
  - 赋值
  - 定义输出节

SECTIONS命令 II

定义输出节

```
SECTIONS
{
    ...
    secname :
    {
        contents
    }
    ...
}
```

SECTIONS命令 III

例如：

```
SECTIONS
{
        ROM:{*(.text)}>rom
}
```

定位计数器

- 定位计数器，The Location Counter
  - 一个特殊的ld变量，使用“.”表示
  - 总是在SECTIONS中使用

例如：

```
SECTIONS
{
        output:
        {
                file1(.text);
                . = . + 1000;
                file2(.text);
                . = . + 1000;
                file3(.text);
        } = 0x1234;
}
```

一个简单例子

- 下面是一个简单的例子：
  例中，输出文件包含text，data，bss三个节，
  而输入文件也只包含这3个节：

```
SECTIONS
{
    .=0x01000000;
    .text:{*(.text)};
    .=0x08000000;
    .data:{*(.data)};
    .bss:{*(.bss)};
}
```

SECTIONS举例（对应于上面的MEMORY例子）

```
SECTIONS
{
  //定义输出文件的ramvec节
  .ramvec:
  {
  //设定一个变量_ramvec来代表当前的位置，即ramvec节的开始处
  _ramvec=.;
  }>ramvec     //把该节定义到MEMORY中ramvec所代表的内存块中

  //定义输出文件的data节
  .data:
  {
  //设定一个变量_data_start来代表当前的位置，即data节的开始处
  _data_start = .;
  //把所有输入文件中的.data节的数据放在此处
  *(.data)
  //设定一个变量_edata为.data节的结束地址
  _edata = .;
  //把_edata按16位对齐
  _edata = ALIGN(0x10);
  }>ram //把.data节的内容放到ram定义的MEMORY中

  ......
```

linux-2.6.26, ARCH=arm

作业：分析链接描述文件（提交电子版）

- Vmlinux：
  arch/$(ARCH)/kernel/vmlinux.lds
- 制作压缩版映像时：
  linux/arch/arm/boot/compressed/vmlinux.lds
- 制作bootp时：
  linux/arch/arm/boot/bootp/bootp.lds

Outline

Outline

GNU tools 交叉开发环境的安装

- 交叉开发环境
  - 源代码配置安装，or
  - 直接安装二进制工具
- 第一种方法比较复杂，如果有现成的二进制交叉环境，建议直接使用

使用源代码安装交叉开发环境

- GNU tools的各个软件包相对独立，
  - 在选择时要注意各个软件包的版本号及其依赖关系
  - 如果全部是最新版本，也并不能保证可以配置并安装成功
- 在安装GNU tools交叉开发环境之前，首先必须建立本地GNU tools环境

一个可行的GNU tools与 Linux内核之间关系表

Table 4-2. Known functional package version combinations

Host	Target	Kernel	binutils	gcc	glibc	Patches
i386	PPC		2.10.1	2.95.3	2.2.1	No
PPC	i386		2.10.1	2.95.3	2.2.3	No
PPC	i386		2.13.2.1	3.2.1	2.3.1	No
i386	ARM	2.4.1-rmk1	2.10.1	2.95.3	2.1.3	Yes[3]
PPC	ARM		2.10.1	2.95.3	2.2.3	Yes[3]
i386	MIPS		2.8.1	egcs-1.1.2	2.0.6	Yes[4]
i386	SuperH		2.11.2	3.0.1	2.2.4	Yes[5]
Sparc (Solaris)	PPC	2.4.0	2.10.1	2.95.2	2.1.3	No

交叉开发环境的安装顺序

- 需要5个步骤完成整个GNU Tools的配置/编译/安装：
  1. 内核头文件配置
  2. binutils软件包安装
  3. Bootstrap GNU编译器（可完成基本C语言编译工作的编译器）
  4. C library的安装，一般是glibc
  5. 完整的GNU编译器安装

软件包安装步骤

- 对每个单独的软件包，一般安装过程包括下面4步
  - 下载并解压软件包
  - 配置软件包
  - 编译软件包
  - 安装软件包
- 在kubuntu等环境下，有些交叉环境也可以使用apt-get等命令安装

安装现成的二进制交叉环境

- 根据来源，可以有2种：
  1. 安装ubuntu-14.04源自带的交叉编译环境
  2. 安装其他现成的二进制交叉编译环境

1、安装ubuntu-14.04源自带的交叉编译环境

- 安装Ubuntu-14.04中自带的交叉编译工具：

```
sudo apt-get install gcc-arm-linux-gnueabi
```

- 查看是否安装成功：

```
arm-linux-gnueabi-<tab>
```

arm-linux-gnueabi-<tab>

```
arm-linux-gnueabi-addr2line       arm-linux-gnueabi-gcov-4.7
arm-linux-gnueabi-ar              arm-linux-gnueabi-gprof
arm-linux-gnueabi-as              arm-linux-gnueabi-ld
arm-linux-gnueabi-c++filt         arm-linux-gnueabi-ld.bfd
arm-linux-gnueabi-cpp             arm-linux-gnueabi-ld.gold
arm-linux-gnueabi-cpp-4.7         arm-linux-gnueabi-nm
arm-linux-gnueabi-dwp             arm-linux-gnueabi-objcopy
arm-linux-gnueabi-elfedit         arm-linux-gnueabi-objdump
arm-linux-gnueabi-gcc             arm-linux-gnueabi-ranlib
arm-linux-gnueabi-gcc-4.7         arm-linux-gnueabi-readelf
arm-linux-gnueabi-gcc-ar-4.7      arm-linux-gnueabi-size
arm-linux-gnueabi-gcc-nm-4.7      arm-linux-gnueabi-strings
arm-linux-gnueabi-gcc-ranlib-4.7  arm-linux-gnueabi-strip
arm-linux-gnueabi-gcov
```

1、安装ubuntu-14.04源自带的交叉编译环境

- 查看交叉编译器的版本信息

arm-linux-gnueabi-gcc -v

arm-linux-gnueabi-gcc -v

```
使用内建 specs。
COLLECT_GCC=arm-linux-gnueabi-gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc-cross/arm-linux-gnueabi/4.7/lto-wrapper
目标：arm-linux-gnueabi
配置为：../src/configure -v —with-pkgversion=’Ubuntu/Linaro 4.7.3-12ubuntu1’
—with-bugurl=file:///usr/share/doc/gcc-4.7/README.Bugs
—enable-languages=c,c++,go,fortran,objc,obj-c++ —prefix=/usr —program-suffix=-4.7
—enable-shared —enable-linker-build-id —libexecdir=/usr/lib
—without-included-gettext —enable-threads=posix
—with-gxx-include-dir=/usr/arm-linux-gnueabi/include/c++/4.7.3 —libdir=/usr/lib
—enable-nls —with-sysroot=/ —enable-clocale=gnu —enable-libstdcxx-debug
—enable-gnu-unique-object —disable-libmudflap —disable-libitm —enable-plugin
—with-system-zlib —enable-objc-gc —with-cloog —enable-cloog-backend=ppl
—disable-cloog-version-check —disable-ppl-version-check —enable-multiarch
—enable-multilib —disable-sjlj-exceptions —with-arch=armv5t —with-float=soft
—disable-werror —enable-checking=release —build=i686-linux-gnu
—host=i686-linux-gnu —target=arm-linux-gnueabi —program-prefix=arm-linux-gnueabi-
—includedir=/usr/arm-linux-gnueabi/include
线程模型：posix
```

2、安装其他现成的二进制交叉编译环境

- 以arm-elf-tools为例，其他用到时再说
- 下载arm-elf-tools的最新版本或合适的版本
  如：arm-elf-tools-20030314.sh
  - 地址1：arm-elf-tools-20030314.sh
  - 或者：arm-elf-tools-20030314.sh
- 安装
  - 在root权限下运行

    ```
    sh ./arm-elf-tools-20030314.sh
    ```

  - 这个命令会在开发主机上自动建立一个μClinux-ARM的交叉编译环境

安装成功？ I

- 检查一下

arm-elf-<tab>

```
xlanchen@xlanchen-System-Product-Name:~/workspace$ arm-elf-
arm-elf-addr2line    arm-elf-gasp       arm-elf-protoize
arm-elf-ar           arm-elf-gcc        arm-elf-ranlib
arm-elf-as           arm-elf-gdb        arm-elf-readelf
arm-elf-c++          arm-elf-ld         arm-elf-run
arm-elf-c++filt      arm-elf-ld.real    arm-elf-size
arm-elf-elf2flt      arm-elf-nm         arm-elf-strings
arm-elf-flthdr       arm-elf-objcopy    arm-elf-strip
arm-elf-g++          arm-elf-objdump    arm-elf-unprotoize
xlanchen@xlanchen-System-Product-Name:~/workspace$ arm-elf-
```

对 arm-elf运用shell的tab功能可以看到一系列arm-elf打头的程序，如binutils系列、gcc、gdb等等

安装成功？ II

- 运行

arm-elf-gcc -v

arm-elf-gcc -v

```
Reading specs from /usr/local/lib/gcc-lib/arm-elf/2.95.3/specs
gcc version 2.95.3 20010315 (release)(ColdFire patches -
20010318 from http://fiddes.net/coldfire/)(uClinux XIP and
shared lib patches from http://www.snapgear.com/)
```

Outline

使用安装好的交叉编译器编译hello

hello.c循环输出“HelloWorld!”

```
#include <stdio.h>

int main(void){
    while(1)
        printf(" HelloWorld!\n" );
    return 0;
}
```

- 分别使用几个不同的编译器编译hello.c

```
gcc -o hello0 hello.c
arm-linux-gnueabi-gcc -o hello1 hello.c
arm-elf-gcc -elf2flt -o hello2 hello.c
```

问题：hello1和hello2是否能在主机上直接运行？为什么？

使用安装好的交叉编译器编译hello

file hello0

```
hello0: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked
(uses shared libs), for GNU/Linux 2.6.24, BuildID[sha1]=a34dd50d1215638bc8a7b3a61ffe
d2e8ded77f18, not stripped
```

file hello1

```
hello1: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked
(uses shared libs), for GNU/Linux 2.6.32, BuildID[sha1]=e1c7864cacb6b576c1d48bc116d7
191d72614c83, not stripped
```

file hello2

```
hello2: BFLT executable - version 4 ram
```

问题：hello1和hello2能在什么样的平台上运行？

Outline

使用安装好的交叉编译器编译linux

- 尝试一下缺省编译

```
cd linux-2.6.26
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
s3c2410_defconfig
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
```

- 编译的产物：
  - 源代码根目录下的vmlinux
  - arch/arm/boot/zImage

Outline

编译μClinux: 1、准备交叉编译环境 I

1. 下载arm-linux-tools-20061213.tar.gz
2. 在主机的系统根目录下或在当前目录下解压缩

   sudo tar zvxf arm-linux-tools-20061213.tar.gz

3. 查看安装是否成功

./usr/local/bin/arm-linux-<tab>

```
arm-linux-addr2line          arm-linux-g77          arm-linux-jv-scan
arm-linux-addr2name.awk      arm-linux-gcc          arm-linux-ld
arm-linux-ar                 arm-linux-gcc-3.4.4     arm-linux-ld.real
arm-linux-arm-linux-gcjh     arm-linux-gccbug       arm-linux-nm
arm-linux-as                 arm-linux-gcj          arm-linux-objcopy
arm-linux-c++                arm-linux-gcjh         arm-linux-objdump
arm-linux-c++filt            arm-linux-gcov         arm-linux-ranlib
arm-linux-cpp                arm-linux-gnatbind     arm-linux-readelf
arm-linux-elf2flt            arm-linux-grepjar      arm-linux-size
arm-linux-flthdr             arm-linux-jar          arm-linux-strings
arm-linux-g++                arm-linux-jcf-dump     arm-linux-strip
```

编译μClinux：1、准备交叉编译环境 II

4. 查看版本信息

```
./usr/local/bin/arm-linux-gcc -v
```

```
Reading specs from
/home/xlanchen/workspace/usr/local/bin/../lib/gcc/arm-linux/3.4.4/specs
Configured with: ../configure —target=arm-linux —disable-shared —prefix=/usr/local
—with-headers=/home/gerg/new-wave.ixdp425/linux-2.4.x/include —with-gnu-as
—with-gnu-ld —enable-multilib
Thread model: posix
gcc version 3.4.4
```

编译μClinux：2、配置并编译μClinux

1. 下载源代码uClinux-dist-20140504.tar.bz2
2. 解压缩

```
tar -jvxf uClinux-dist-20140504.tar.bz2
```

3. 配置并编译

```
export LDLIBS=-ldl（可能需要）
（还可能需要安装一些软件，请根据错误信息提示安装所需软件）

make xconfig
（1）在vendor/product选项中选择GDB/ARMulator
（2）Kernel版本选择2.4.x
（3）其他选项不变（使用缺省选项）

make dep; make
```

编译μClinux：2、配置并编译μClinux

4. 查看编译出来的内核映像
   - images目录下：

```
images/
├── boot.rom
├── linux
└── romfs-inst.log
```

   - linux-2.4.x目录下：linux

编译μC1inux:
3、运行uC1inux的内核（不使用根文件系统）

- 编写skyeye.conf，内容如下：

skyeye.conf

```
#skyeye config file sample
cpu: arm7tdmi
mach: at91
mem_bank: map=M, type=RW, addr=0x00000000, size=0x00004000
mem_bank: map=M, type=RW, addr=0x01000000, size=0x00400000
mem_bank: map=I, type=RW, addr=0xf0000000, size=0x10000000
```

- 运行：

```
skyeye -e linux -c skyeye.conf
```

Outline

小结

1. 前言
   - 交叉开发
2. GNU Tools简介
   - GCC
   - GNU binutils
   - Gdb—调试器
   - GNU make——软件工程工具
   - GNU ld——链接器
3. GNU tools 交叉开发环境的安装和试用
   - GNU tools 交叉开发环境的安装
   - 使用安装好的交叉编译器编译hello
   - 使用安装好的交叉编译器编译linux
   - 编译μClinux
4. 小结和作业

作业：

1. 在课件中
2. 注意：两个对链接描述文件的分析作业，
   选择Linux/μClinux/arm1inux的一个就可以了。
   （推荐：arm1inux）

Thanks !

The end.