GCC 交叉编译平台建立过程

（人工智能与机器人研究所 李国辉）

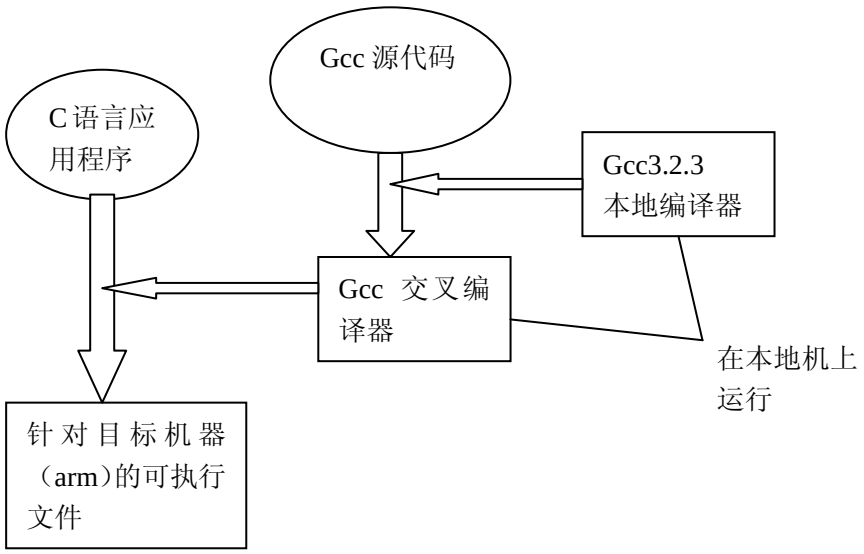
基于嵌入式系统的开发环境主要涉及到交叉编译器，汇编器、链接器等。这些工具一般由开发 cpu 的厂商提供，可以直接在 windows 下或者在 linux 下运行。本文的目的并不是具体的讲述如何去使用这些工具，而是当得不到这些现成的工具时，自己如何去构造这些工具。构造的方法就是应用 gcc 的源代码建立针对目标处理器的交叉编译平台。

GCC（GNU Compiler Collection）是现阶段被广泛使用的开放源码的编译器，它支持多种高级语言（c,c++,fortran,java,ada），同时支持多种处理器（alpha,arm,avr,IA-64,intel 386,AMD,mips,mmix,powerpc,sparc,pdp-11…）,它不仅因为其支持多目标的特性而被广大厂商使用，更是因为其在生成代码的质量、速度上的优秀表现而获得广泛的赞美。下面具体讲述交叉编译平台建立过程。

一、准备工作

1． 安装 linux 操作系统

这个平台是在 linux 下建立起来的，所用的 linux 版本为 redhatv9.2。所用的本地编译器为 gcc3.2.2（redhatv9.2 自带的 gcc）。用这个本地编译器编译 gcc 的源代码使之产生针对 arm 处理器的交叉编译器。本地编译器是指它编译出来的程序可在本地机（即运行编译器的机器）上运行；而交叉编译器编译出来的程序不可以在本地机上运行。如下图所示：



需要说明的是，由于 linux 下的一些程序并不具有很好的兼容性和稳定性，因此所用的 gcc 本地编译器最好是稳定的版本（比如 gcc3.2.2），而不要一味地追求最高的版本。

2. 为目标机器建立 binutils.

Binutils 是一个工具包，它包含汇编器、链接器以及管理静态库、动态库的一些工具。我们从网上下载的一般是 binutils 的源文件，我们需要把它编译成针对目标机（arm）的工具，这样编译后得到的汇编器是针对 arm 的汇编器，链接器也是针对 arm 的链接器。

首先从网上下载 binutils-2.13.2.1.tar.gz,执行以下命令：

```
tar –xzvf  binutils-2.13.2.1.tar.gz
```

新生成的 binutils-2.13.2.1 被称为源目录，用${srcdir}表示它。以后在出现${srcdir}的地方，应该用真实的路径来替换它。

然后建立目标目录，目标目录（用${objdir}表示）是用来存放编译结果的地方。GCC 建议编译后的文件不要放在源目录${srcdir}中（虽然这样做也可以），最好单独存放在另外一个目录中，比如${srcdir}的同级目录。用 cd 命令进入目标目录，然后执行以下命令：

```
${srcdir}/configure --prefix=/usr/local --target=arm-elf
--with-cpu=arm8
```

配置的目的是将 binutils 安装到什么地方，在这里为/usr/local,在 linux 系统下，这个目录就是用来存放安装结果的地方。--targeth 和--with-cpu 选项说明了生成的汇编器和链接器等是针对 arm 处理器的，具体是 arm 系列的 arm8 处理器,并且该处理器上还没有运行操作系统（大多数嵌入式系统都是这样的）。

该命令执行**${srcdir}**目录下的 configure 文件，同时接受用户输入的选项，然后生成 Makefile。Makefile 说明了 gcc 众多源文件的编译顺序和依赖关系。用户在编译 gcc 源代码时，不需要依次对各源文件进行编译，只需执行 make 命令即可，它能解释 makefile 里的编译顺序和依赖关系，一次性完成全部的编译过程。关于 Makefile 的更进一步的探讨，有兴趣的同学可以参考附录。

配置完成之后然后执行以下两个命令就完成了安装过程：

```
make
make install
```

其中执行 make install 要求用户具有超级权限。执行以下命令会看到编译生成的工具：

```
ls  /usr/local/bin
```

arm-elf-as
arm-elf-ld
arm-elf-ar

```
ls  /usr/local/arm-elf
```

bin lib

我们可以看到/usr/local 是 binutils 安装用到的最上层目录，在执行 configure 命令时通过--prefix 设置了这个最上层目录。

二、 构造交叉编译器

如果准备工作已经就绪，剩下的就是构造这个新的编译器了。构造编译器所需的步骤基本上与构造 binutils 的步骤相同，需要建立目标目录然后在目标目录里执行 configure、make all-gcc、make install-gcc 命令就完成了编译过程。

${srcdir}/configure --prefix=/usr/local --target=arm-elf --with-cpu=arm8 --enable-languages=c --disable-shared --disable-nls --with-gnu-as --with-gnu-ld

${srcdir}就是解压缩 gcc-3.2.3.tar.gz 之后生成的目录，比如……/gcc3.2.3.

在这里也设置了前缀（prefix）,需要注意的是，这个前缀必须与配置 binutils 时的前缀一样。因为在编译 gcc 源码的过程中需要用到 prefix 目录里的一些工具，比如 arm-elf-as、arm-elf-ld、arm-elf-ar 等等。

之后执行

Make all-gcc
make install-gcc

运行完毕之后，可以在/usr/local/bin 里找到构造好的交叉编译程序，如下：

ls /usr/local/bin

arm-elf-gcc
……

需要说明的是：

1. 如果用户设置的 prefix 选项不是/usr/local，而是其他目录，用${prefix}表示，那么在 make 之前需要执行：

PATH=$PATH: ${prefix}

这样设置了 PATH 变量之后，在编译时，arm-elf-as、arm-elf-ld 这些工具就能够被找到。

2. 在编译时，一定要是 make all-gcc,而不能是简单的 make 命令，make all-gcc 只生成交叉编译程序，不生成 c 运行时库，这在很多情况下已经足够了。

三、后续的工作

后续的工作就是生成 c 运行时库，有这些库的支持，c 语言应用程序的开发才会变得功能强大。在这里，只简单的说明库的作用，至于如何生成 c 运行时库，可参考《gcc 技术参考大全》第 17 章。

在 c 语言中，一些比较常用的函数，比如<math.h>里的函数，已经由编译器的厂商写好，并且已经编译成二进制文件（目标文件），用户如果要用这些函数，不需要再去写函数，只需调用即可。这些常用的函数范围相当广泛，涉及到输入输出，文件读写，内存分配等，极大地方便了程序员开发出功能强大的应用程序。这些函数的目标文件的集合就是库。不仅编译器或者操作系统提供这种一般功能的库，也有其他软件开发商提供专门用途的库,这种类型的库称为 sdk（software development kit），比如专门用于图像处理的工具包 VisualSDK、视频处理工具包 OPENCV、图形界面开发工具包 GTK。另外，微软开发的 MFC 类库其实也是各种

类定义的目标文件集合。使用库进行编程能省去程序员大量的重复劳动，一个高效的程序员应该在开始写程序之前，应确定要写的模块是否能用现成的库来实现，这样既保证了代码的可靠性又节省了时间。在 GCC 中生成库的例子如下：

```
gcc -c inlet.c outlet.c genspru.c
ar -r libspin.a inlet.o outlet.o genspru.o
gcc main.c libspin.a -o spinner
```

注：libspin.a 是 inlet.o outlet.o genspru.o 的集合，但在链接时，并不是这三个目标文件都被链接进来，只有 main.c 中用到的目标文件才会被链接进来。

Makefile 概述

什么是 makefile？或许很多 Winodws 的程序员都不知道这个东西，因为那些 Windows 的 IDE 都为你做了这个工作，但我觉得要作一个好的和 professional 的程序员，makefile 还是要懂。这就好像现在有这么多的 HTML 的编辑器，但如果你想成为一个专业人士，你还是要了解 HTML 的标识的含义。特别在 Unix 下的软件编译，你就不能不自己写 makefile 了，会不会写 makefile，从一个侧面说明了一个人是否具备完成大型工程的能力。

因为，makefile 关系到了整个工程的编译规则。一个工程中的源文件不计数，其按类型、功能、模块分别放在若干个目录中，makefile 定义了一系列的规则来指定，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作，因为 makefile 就像一个 Shell 脚本一样，其中也可以执行操作系统的命令。

makefile 带来的好处就是——“自动化编译”，一旦写好，只需要一个 make 命令，整个工程完全自动编译，极大的提高了软件开发的效率。make 是一个命令工具，是一个解释 makefile 中指令的命令工具，一般来说，大多数的 IDE 都有这个命令，比如：Delphi 的 make，Visual C++ 的 nmake，Linux 下 GNU 的 make。可见，makefile 都成为了一种在工程方面的编译方法。

现在讲述如何写 makefile 的文章比较少，这是我想写这篇文章的原因。当然，不同产商的 make 各不相同，也有不同的语法，但其本质都是在“文件依赖性”上做文章，这里，我仅对 GNU 的 make 进行讲述，我的环境是 RedHat Linux 8.0，make 的版本是 3.80。必竟，这个 make 是应用最为广泛的，也是用得最多的。而且其还是最遵循于 IEEE 1003.2-1992 标准的（POSIX.2）。

在这篇文档中，将以 C/C++的源码作为我们基础，所以必然涉及一些关于 C/C++的编译的知识，相关于这方面的内容，还请各位查看相关的编译器的文档。这里所默认的编译器是 UNIX 下的 GCC 和 CC。

关于程序的编译和链接

在此，我想多说关于程序编译的一些规范和方法，一般来说，无论是 C、C++、还是 pas，首先要把源文件编译成中间代码文件，在 Windows 下也就是 .obj 文件，UNIX 下是 .o 文件，即 Object File，这个动作叫做编译（compile）。然后再把大量的 Object File 合成执行文件，这个动作叫作链接（link）。

编译时，编译器需要的是语法的正确，函数与变量的声明的正确。对于后者，通常是你需要告诉编译器头文件的所在位置（头文件中应该只是声明，而定义应该放在 C/C++文件中），只要所有的语法正确，编译器就可以编译出中间目标文件。一般来说，每个源文件都应该对应于一个中间目标文件（O 文件或是 OBJ 文件）。

链接时，主要是链接函数和全局变量，所以，我们可以使用这些中间目标文件（O 文件或是 OBJ 文件）来链接我们的应用程序。链接器并不管函数所在的源文件，只管函数的中间目标文件（Object File），在大多数时候，由于源文件太多，编译生成的中间目标文件太多，而在链接时需要明显地指出中间目标文件名，这对于编译很不方便，所以，我们要给中间目标文件打个包，在 Windows 下这种包叫“库文件”（Library File)，也就是 .lib 文件，在 UNIX 下，是 Archive File，也就是 .a 文件。

总结一下，源文件首先会生成中间目标文件，再由中间目标文件生成执行文件。在编译时，编译器只检测程序语法，和函数、变量是否被声明。如果函数未被声明，编译器会给出一个警告，但可以生成 Object File。而在链接程序时，链接器会在所有的 Object File 中找寻函数的实现，如果找不到，那到就会报链接错误码（Linker Error），在 VC 下，这种错误一般是：Link 2001 错误，意思说是说，链接器未能找到函数的实现。你需要指定函数的 Object File.

好，言归正传，GNU 的 make 有许多的内容，闲言少叙，还是让我们开始吧。

Makefile 介绍

make 命令执行时，需要一个 Makefile 文件，以告诉 make 命令需要怎么样的去编译和链接程序。

首先，我们用一个示例来说明 Makefile 的书写规则。以便给大家一个感兴认识。这个示例来源于 GNU 的 make 使用手册，在这个示例中，我们的工程有 8 个 C 文件，和 3 个头文件，我们要写一个 Makefile 来告诉 make 命令如何编译和链接这几个文件。我们的规则是：

1）如果这个工程没有编译过，那么我们的所有 C 文件都要编译并被链接。

2）如果这个工程的某几个 C 文件被修改，那么我们只编译被修改的 C 文件，并链接目标程序。

3）如果这个工程的头文件被改变了，那么我们需要编译引用了这几个头文件的 C 文件，并链接目标程序。

只要我们的 Makefile 写得够好，所有的这一切，我们只用一个 make 命令就可以完成，make 命令会自动智能地根据当前的文件修改的情况来确定哪些文件需要重编译，从而自己编译所需要的文件和链接目标程序。

一、Makefile 的规则

在讲述这个 Makefile 之前，还是让我们先来粗略地看一看 Makefile 的规则。

```
target ... : prerequisites ...
        command
        ...
```

...

target 也就是一个目标文件，可以是 Object File，也可以是执行文件。还可以是一个标签（Label），对于标签这种特性，在后续的“伪目标”章节中会有叙述。

prerequisites 就是，要生成那个 target 所需要的文件或是目标。

command 也就是 make 需要执行的命令。（任意的 Shell 命令）

这是一个文件的依赖关系，也就是说，target 这一个或多个的目标文件依赖于 prerequisites 中的文件，其生成规则定义在 command 中。说白一点就是说，prerequisites 中如果有一个以上的文件比 target 文件要新的话，command 所定义的命令就会被执行。这就是 Makefile 的规则。也就是 Makefile 中最核心的内容。

说到底，Makefile 的东西就是这样一点，好像我的这篇文档也该结束了。呵呵。还不尽然，这是 Makefile 的主线和核心，但要写好一个 Makefile 还不够，我会以后面一点一点地结合我的工作经验给你慢慢到来。内容还多着呢。：）

二、一个示例

正如前面所说的，如果一个工程有 3 个头文件，和 8 个 C 文件，我们为了完成前面所述的那三个规则，我们的 Makefile 应该是下面的这个样子的。

```
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
```

反斜杠（\）是换行符的意思。这样比较便于 Makefile 的易读。我们可以把这个内容保存在文件为“Makefile”或“makefile”的文件中，然后在该目录下直接输入命令“make”就可以生成执行文件 edit。如果要删除执行文件和所有的中间目标文件，那么，只要简单地执行一下“make clean”就可以了。

在这个 makefile 中，目标文件（target）包含：执行文件 edit 和中间目标文件（*.o），依赖文件（prerequisites）就是冒号后面的那些 .c 文件和 .h 文件。每一个 .o 文件都有一组依赖文件，而这些 .o 文件又是执行文件 edit 的依赖文件。依赖关系的实质上就是说明了目标文件是由哪些文件生成的，换言之，目标文件是哪些文件更新的。

在定义好依赖关系后，后续的那一行定义了如何生成目标文件的操作系统命令，一定要以一个 Tab 键作为开头。记住，make 并不管命令是怎么工作的，他只管执行所定义的命令。make 会比较 targets 文件和 prerequisites 文件的修改日期，如果 prerequisites 文件的日期要比 targets 文件的日期要新，或者 target 不存在的话，那么，make 就会执行后续定义的命令。

这里要说明一点的是，clean 不是一个文件，它只不过是一个动作名字，有点像 C 语言中的 lable 一样，其冒号后什么也没有，那么，make 就不会自动去找文件的依赖性，也就不会自动执行其后所定义的命令。要执行其后的命令，就要在 make 命令后明显得指出这个 lable 的名字。这样的方法非常有用，我们可以在一个 makefile 中定义不用的编译或是和编译无关的命令，比如程序的打包，程序的备份，等等。

函数调用时的栈分配过程

对于如下函数：

```
------------------sum.c--------------------
int main()
{
    int b=1,c=2;
    if(b>c)
        return b;
    else return c;
}
--------------------sum.c  end-------------------
```

使用命令：**arm-elf-gcc -S sum.c**
得到编译之后的汇编文件 sum.s

```
---------------------------sum.s----------------
main:
    @ args = 0, pretend = 0, frame = 8
    @ frame_needed = 1, current_function_anonymous_args = 0
    mov ip, sp
    stmfd  sp!, {fp, ip, lr, pc}
    sub fp, ip, #4
    sub sp, sp, #8
    bl  __gccmain
    mov r3, #1
    str r3, [fp, #-16]
    mov r3, #2
    str r3, [fp, #-20]
    ldr r3, [fp, #-16]
    ldr r2, [fp, #-20]
    cmp r3, r2
    ble .L3
    ldr r3, [fp, #-16]
    mov r0, r3
    b   .L2
    b   .L4
.L3:
    ldr r3, [fp, #-20]
    mov r0, r3
    b   .L2
.L4:
.L2:
    ldmea  fp, {fp, sp, pc}
```

```
.Lfe1:
    .size   main,.Lfe1-main
------------sum.s  end--------------------------
```

可用下图来说明栈的分配过程：