《嵌入式 Linux C 语言开发》

作者：华清远见

第 1 章 嵌入式 Linux C 语言开发工具

本章目标

任何应用程序的开发都离不开编辑器、编译器及调试器，嵌入式 Linux 的 C 语言开发也一样，它也有一套优秀的编辑、编译及调试工具。

掌握这些工具的使用是至关重要的，它直接影响到程序开发的效率。希望读者通过自己的实践，熟练掌握这些工具的使用。通过本章的学习，读者将会掌握如下内容：

- C 语言产生的历史背景
- 嵌入式 Linux 下 C 语言的开发环境
- 嵌入式 Linux 下的编辑器 vi
- 嵌入式 Linux 下的编译器 GCC
- 嵌入式 Linux 下的调试器 GDB
- 嵌入式 Linux 下的工程管理器 make
- 如何使用 autotools 来生成 Makefile
- 嵌入式 Linux 下的综合编辑器 Emacs

1.1 嵌入式 Linux 下 C 语言概述

读者在第一章中已经了解了嵌入式开发的基本流程，在嵌入式系统中应用程序的主体是在宿主机中开发完成的，就嵌入式 Linux 而言，此过程则一般是在安装有 Linux 的宿主机中完成。
在本章中介绍的实际是嵌入式 Linux 下 C 语言的开发工具，用户在开发时往往是在 Linux 宿主机中对程序进行调试，然后再进行交叉编译的。

1.1.1 C 语言简史

C 语言于 20 世纪 70 年代诞生于美国的贝尔实验室。在此之前，人们编写系统软件主要是使用汇编语言。汇编语言编写的程序依赖于计算机硬件，其可读性和可移植性都比较差。而高级语言的可读性和可移植性虽然较汇编语言好，但一般高级语言又不具备低级语言能够直观地对硬件实现控制和操作而且执行速度快等特点。
在这种情况下，人们迫切需要一种既具有一般高级语言特性，又具有低级语言特性的语言，于是 C 语言就应运而生了。
由于 C 语言既具有高级语言的特点又具有低级语言的特点，因此迅速普及，成为当今最有发展前途的计算机高级语言之一。C 语言既可以用来编写系统软件，也可以用来编写应用软件。现在，C 语言已经被广泛地应用在除计算机行业外的机械、建筑和电子等各个行业中。
C 语言的发展历程如下。

- C 语言最初是美国贝尔实验室的 D.M.Ritchie 在 B 语言的基础上设计出来的，此时的 C 语言只是为了描述和实现 UNIX 操作系统的一种工作语言。在一段时间里，C 语言还只在贝尔实验室内部使用。
- 1975 年，UNIX 第 6 版公布后，C 语言突出的优点引起人们的普遍注意。
- 1977 年出现了可移植的 C 语言。
- 1978 年 UNIX 第 7 版的 C 语言成为后来被广泛使用的 C 语言版本的基础，被称为标准 C 语言。
- 1983 年，美国国家标准化协会（ANSI）根据 C 语言问世以来的各种版本，对 C 语言进行发展和扩充，并制定了新的标准，称为 ANSI C。
- 1990 年，国际标准化组织 ISO 制定了 ISO C 标准，目前流行的 C 语言编译系统都是以它为标准的。

1.1.2 C 语言特点

C 语言兼有汇编语言和高级语言的优点，既适合于开发系统软件，也适合于编写应用程序。被广泛应用于事务处理、科学计算、工业控制、数据库技术等领域。
C 语言之所以能存在和发展，并具有强大的生命力，这都要归功于其鲜明的特点。这些特点是多方面的，归纳如下。

1．C 语言是结构化的语言

C 语言采用代码及数据分隔的方式，使程序的各个部分除了必要的信息交流外彼此独立。这种结构化方式可使程序层次清晰，便于使用、维护以及调试。
C 语言是以函数形式提供给用户的，这些函数可方便地调用，并具有多种循环、条件语句控制程序流向，从而使程序完全结构化。

2．C 语言是模块化的语言

C 语言主要用于编写系统软件和应用软件。一个系统软件的开发需要很多人经过几年的时间才能完成。一般来说，一个较大的系统程序往往被分为若干个模块，每一个模块用来实现特定的功能。

在 C 语言中，用函数作为程序的模块单位，便于实现程序的模块化。在程序设计时，将一些常用的功能模块编写成函数，放在函数库中供其他函数调用。模块化的特点可以大大减少重复编程。程序设计时，只要善于利用函数，就可减少劳动量、提高编程效率。

3．程序可移植性好

C 语言程序便于移植，目前 C 语言在许多计算机上的实现大都是由 C 语言编译移植得到的，不同机器上的编译程序大约有 80%的代码是公共的。程序不做任何修改就可用于各种型号的计算机和各种操作系统。因此，特别适合在嵌入式开发中使用。

4．C 语言运算符丰富、代码效率高

C 语言共有 34 种运算符，使用各种运算符可以实现在其他高级语言中难以实现的运算。在代码质量上，C 语言可与汇编语言媲美，其代码效率仅比用汇编语言编写的程序的代码低 10%～20%。

1.1.3　嵌入式 Linux C 语言编程环境

嵌入式 Linux C 语言程序设计与在其他环境中的 C 程序设计很类似，也涉及编辑器、编译链接器、调试器及项目管理工具的使用。现在我们先对这 4 种工具进行简单介绍，后面会对其一一进行讲解。

1．编辑器

嵌入式 Linux 下的编辑器就如 Windows 下的 Word、记事本等一样，完成对所录入字符的编辑功能，最常用的编辑器有 vi（vim）和 Emacs，它们功能强大，使用方便，本书重点介绍 vi 和 Emacs。

2．编译链接器

编译过程包括词法、语法和语义的分析、中间代码的生成和优化、符号表的管理和出错处理等。在嵌入式 Linux 中，最常用的编译器是 GCC 编译器。它是 GNU 推出的功能强大、性能优越的多平台编译器，其执行效率与一般的编译器相比平均效率要高 20%～30%。

3．调试器

调试器可以方便程序员在程序运行时进行源代码级的调试，但不是代码执行的必备工具。在程序开发的过程当中，调试所消耗的时间远远大于编写代码的时间。因此，有一个功能强大、使用方便的调试器是必不可少的。GDB 可以方便地设置断点、单步跟踪等，足以满足开发人员的需要。

4．项目管理器

嵌入式 Linux 中的项目管理器“make”类似于 Windows 中 Visual C++里的“工程”管理，它是一种控制编译或者重复编译代码的工具。另外，它还能自动管理软件编译的内容、方式和时机，使程序员能够把精力集中在代码的编写上而不是在源代码的组织上。

1.2 嵌入式 Linux 编辑器 vi 的使用

vi 是 Linux 系统的第一个全屏幕交互式编辑工具。它从诞生至今一直得到广大用户的青睐，历经数十年后仍然是人们主要使用的文本编辑工具，足见其生命力之强，其强大的编辑功能可以同任何一个最新的编辑器相媲美。

虽然用惯了 Windows 中的 Word 等编辑器的读者在刚刚接触 vi 时或多或少会有些不适应，但使用过一段时间后，就能感受到它的方便与快捷。

小知识 Linux 系统提供了一个完整的编辑器家族系列，如 Ed、Ex、Vi 和 Emacs 等，按功能它们可以分为两大类：行编辑器（Ed、Ex）和全屏幕编辑器（Vi、Emacs）。行编辑器每次只能对一行进行操作，使用起来很不方便。而全屏幕编辑器可以对整个屏幕进行编辑，用户编辑的文件直接显示在屏幕上，从而克服了行编辑的那种不直观的操作方式，便于用户学习和使用，具有强大的功能。

1.2.1 vi 的基本模式

vi 编辑器具有 3 种工作模式，分别是命令行模式（Command Mode）、插入模式（Insert Mode）和底行模式（Last Line Mode），各模式的功能区分如下。

1．命令行模式（Command Mode）

在该模式下用户可以输入命令来控制屏幕光标的移动，字符、单词或行的删除，移动复制某区段，也可以进入到底行模式或者插入模式下。

2．插入模式（Insert Mode）

用户只有在插入模式下才可以进行字符输入，用户按［Esc］键可回到命令行模式下。

3．底行模式（Last Line Mode）

在该模式下，用户可以将文件保存或退出 vi，也可以设置编辑环境，如寻找字符串、显示行号等。这一模式下的命令都是以“：”开始。

不过在一般使用时，人们通常把 vi 简化成两个模式，即将底行模式（Last Line Mode）也归入命令行模式中。

1.2.2 vi 的基本操作

1．进入与离开 vi

进入 vi 可以直接在系统提示符下键入 vi ＜文档名称＞，vi 可以自动载入所要编辑的文档或是创建一个新的文档。如在 shell 中键入 vi hello.c（新建文档）即可进入 vi 画面。如图 1.1 所示。

进入 vi 后屏幕最左边会出现波浪符号，凡是有该符号就代表该行目前是空的。此时进入的是命令行模式。

要离开 vi 可以在底行模式下键入“:q”（不保存离开），“:wq”（保存离开）则是存档后再离开（注意冒号）。如图 1.2 所示。

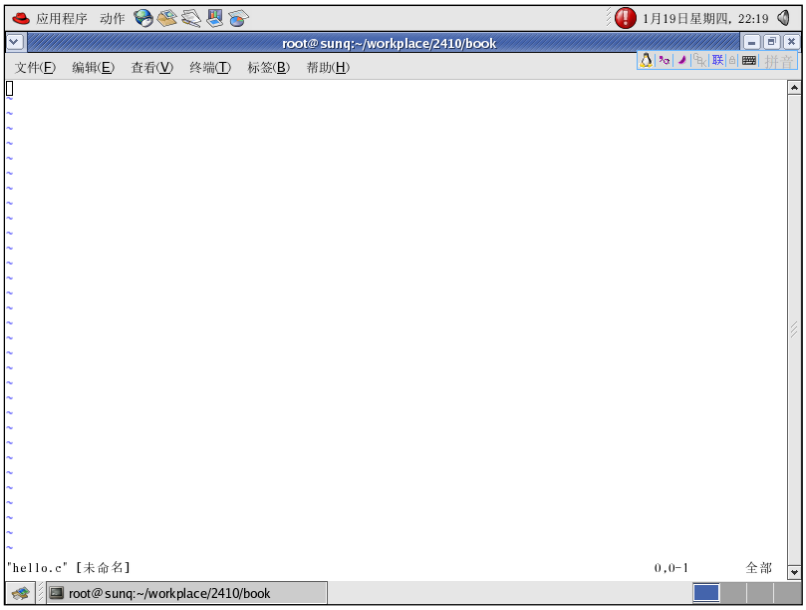


图 1.1　在 vi 中打开/新建文档

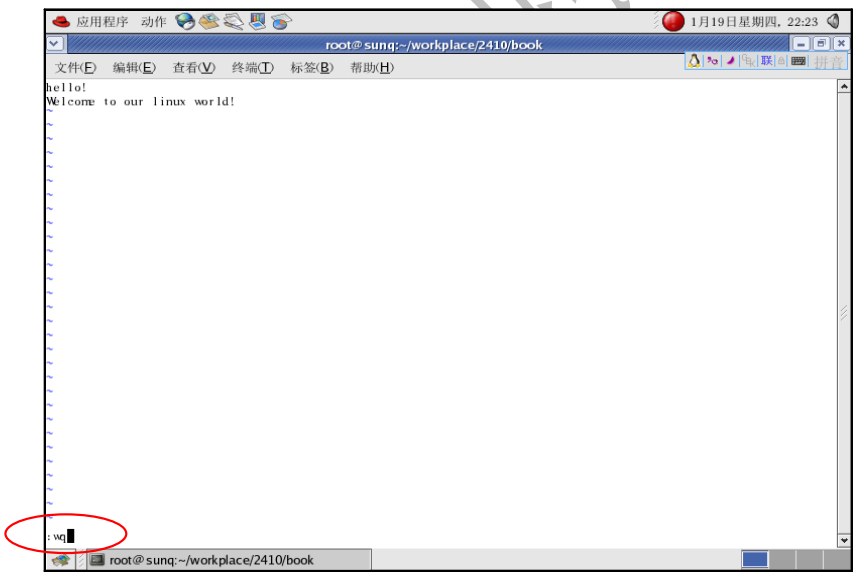


图 1.2　在 vi 中退出文档

2．vi 中 3 种模式的切换

vi 的使用中 3 种模式的切换是最为常用的，在处理的过程中，读者要时刻注意屏幕左下方的提示。在插入模式下，左下方会有“插入”字样，而在命令行或底行模式下则无提示。

（1）命令行模式、底行模式转为插入模式

在命令行模式或底行模式下转入到插入模式有 3 种方法，如表 1.1 所示。

表 1.1　　命令行模式转到插入模式

特　征	命令	作　用
新增	a	从光标所在位置后面开始新增资料，光标后的资料随新增资料向后移动
	A	从光标所在列最后面的地方开始新增资料

插入	i	从光标所在位置前面开始插入资料，游标后的资料随新增资料向后移动
	I	从光标所在列的第一个非空白字元前面开始插入资料
开始	o	在光标所在列下新增一列，并进入插入模式
	O	在光标所在列上方新增一列，并进入插入模式

在这里，最常用的是“i”，在转入插入模式后如图 1.3 所示。

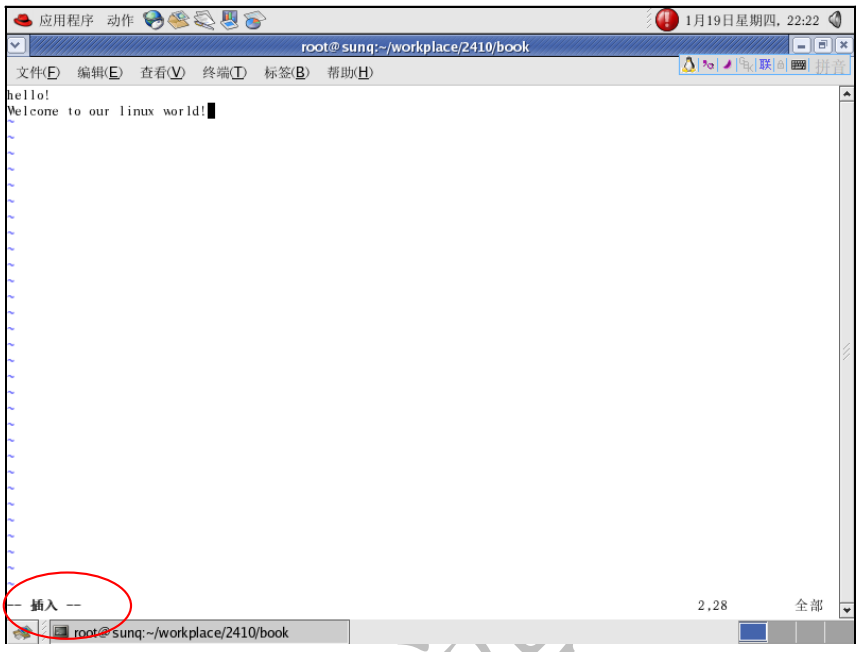


图 1.3　命令模式转入插入模式

（2）插入模式转为命令行模式、底行模式

从插入模式转为命令行模式、底行模式比较简单，只需使用［Esc］键即可。

（3）命令行模式与底行模式转换

命令行模式与底行模式间的转换不需要其他特别的命令，而只需要直接键入相应模式中的命令键即可。

3．vi 的删除、修改与复制

在 vi 中进行删除、修改都可以在插入模式下使用键盘上的方向键及［Delete］键，另外，vi 还提供了一系列的操作指令可以大大简化操作。

这些指令记忆起来比较复杂，希望读者能够配合操作来进行实验。以下命令都是在命令行模式下使用的。表 1.2 所示为 vi 的删除、修改与复制命令。

表 1.2　　vi 的删除、修改与复制命令

特　征	ARM	作　用
删除	x	删除光标所在的字符
	dd	删除光标所在的行
	s	删除光标所在的字符，并进入输入模式
	S	删除光标所在的行，并进入输入模式
修改	r 待修改字符	修改光标所在的字符，键入 r 后直接键入待修改字符
	R	进入取代状态，可移动光标键入所指位置的修改字符，该取代状态直到按[Esc]才结束
复制	yy	复制光标所在的行

	nyy	复制光标所在的行向下 n 行
	p	将缓冲区内的字符粘贴到光标所在位置

4．vi 的光标移动

由于许多编辑功能都是通过光标的定位来实现的，因此，掌握 vi 中光标移动的方法很重要。虽然使用方向键也可以实现 vi 的操作，但 vi 的指令可以实现复杂的光标移动，只要熟悉以后都非常方便，希望读者都能切实掌握。

表 1.3 所示为 vi 中的光标移动指令，这些指令都是在命令行模式下使用的。

表 1.3　　vi 中光标移动的命令

指　令	作　用
0	移动到光标所在行的最前面
$	移动到光标所在行的最后面
［Ctrl］d	光标向下移动半页
［Ctrl］f	光标向下移动一页
H	光标移动到当前屏幕的第一行第一列
M	光标移动到当前屏幕的中间行第一列
L	光标移动到当前屏幕的最后行第一列
b	移动到上一个字的第一个字母
w	移动到下一个字的第一个字母
e	移动到下一个字的最后一个字母
^	移动到光标所在行的第一个非空白字符
n-	向上移动 n 行
n+	向下移动 n 行
nG	移动到第 n 行

5．vi 的查找与替换

在 vi 中的查找与替换也非常简单，其操作有些类似在 Telnet 中的使用。其中，查找的命令在命令行模式下，而替换的命令则在底行模式下（以“：”开头），其命令如表 1.4 所示。

表 1.4　　vi 的查找与替换命令

特　征	ARM	作　用
查找	/<要查找的字符>	向下查找要查找的字符
	?<要查找的字符>	向上查找要查找的字符
替换	:0,$s/string1/string2/g	0，$：替换范围从第 0 行到最后一行 s：转入替换模式 string1/string2:把所有 string1 替换为 string2 g：强制替换而不提示

6．vi 的文件操作指令

vi 中的文件操作指令都是在底行模式下进行的，所有的指令都是以“：”开头，其命令如表 1.5 所示。

表 1.5　　vi 的文件操作指令

指　令	作　用
：q	结束编辑，退出 vi
：q!	不保存编辑过的文档
：w	保存文档，其后可加要保存的文件名
：wq	保存文档并退出
：zz	功能与“：wq”相同
：x	功能与“：wq”相同

1.2.3　vi 的使用实例分析

本节给出了一个 vi 使用的完整实例，通过这个实例，读者一方面可以熟悉 vi 的使用流程，另一方面也可以熟悉 Linux 的操作，希望读者能够首先自己思考每一步的操作，再看后面的实例解析答案。

1．vi 使用实例内容

（1）在/root 目录下建一个名为 vi 的目录。
（2）进入 vi 目录。
（3）将文件/etc/inittab 复制到当前目录下。
（4）使用 vi 编辑当前目录下的 inittab。
（5）将光标移到该行。
（6）复制该行内容。
（7）将光标移到最后一行行首。
（8）粘贴复制行的内容。
（9）撤销第 9 步的动作。
（10）将光标移动到最后一行的行尾。
（11）粘贴复制行的内容。
（12）光标移到“si::sysinit:/etc/rc.d/rc.sysinit”。
（13）删除该行。
（14）存盘但不退出。
（15）将光标移到首行。
（16）插入模式下输入“Hello,this is vi world!”。
（17）返回命令行模式。
（18）向下查找字符串“0:wait”。
（19）再向上查找字符串“halt”。
（20）强制退出 vi，不存盘。

2．vi 使用实例解析

在该实例中，每一步的使用命令如下所示。
（1）mkdir　/root/vi

（2）cd /root/vi
（3）cp /etc/inittab ./
（4）vi ./inittab
（5）17<enter>（命令行模式）
（6）yy
（7）G
（8）p
（9）u
（10）$
（11）p
（12）21G
（13）dd
（14）:w（底行模式）
（15）1G
（16）i 并输入“Hello,this is vi world!”（插入模式）
（17）Esc
（18）/0:wait（命令行模式）
（19）?halt
（20）:q!（底行模式）

1.3 嵌入式 Linux 编译器 GCC 的使用

1.3.1 GCC 概述

作为自由软件的旗舰项目，Richard Stallman 在刚开始编写 GCC 的时候，仅仅只是把它当作一个 C 程序的编译器，GCC 的意思也只是 GNU C Compiler 而已。
经过了多年的发展，GCC 除了能支持 C 语言外，目前还支持 Ada 语言、C++语言、Java 语言、Objective C 语言、PASCAL 语言、COBOL 语言，以及支持函数式编程和逻辑编程的 Mercury 语言等。GCC 也不再单指 GNU C 语言编译器的意思，而是变成了 GNU 编译器家族。
正如前文中所述，GCC 的编译流程分为了 4 个步骤，分别为

- 预处理（Pre-Processing）。
- 编译（Compiling）。
- 汇编（Assembling）。
- 链接（Linking）。

编译器通过程序的扩展名来分辨编写源程序所用的语言，由于不同的程序所需要执行编译的步骤是不同的，因此 GCC 根据不同的后缀名对它们进行相应的处理，表 1.6 指出了不同后缀名的处理方式。

表 1.6 GCC 所支持后缀名解释

后 缀 名	所对应的语言	编 译 流 程
.c	C 原始程序	预处理、编译、汇编
.C/.cc/.cxx	C++原始程序	预处理、编译、汇编
.m	Objective-C 原始程序	预处理、编译、汇编
.i	已经过预处理的 C 原始程序	编译、汇编
.ii	已经过预处理的 C++原始程序	编译、汇编

.s/.S	汇编语言原始程序	汇编
.h	预处理文件（头文件）	（不常出现在指令行）
.o	目标文件	链接
.a/.so	编译后的库文件	链接

1.3.2 GCC 编译流程分析

GCC 使用的基本语法为：

```
gcc [option | filename]
```

这里的 option 是 GCC 使用时的一些选项，通过指定不同的选项 GCC 可以实现其强大的功能。这里的 filename 则是 GCC 要编译的文件，GCC 会根据用户所指定的编译选项以及所识别的文件后缀名来对编译文件进行相应的处理。

本节从编译流程的角度讲解 GCC 的常见使用方法。

先来分析一段简单的 C 语言程序。该程序由两个文件组成，其中“hello.h”为头文件，在“hello.c”中包含了“hello.h”，其源文件如下所示。

```
/*hello.h*/
#ifndef _HELLO_H_
#define _HELLO_H_

typedef unsigned long val32_t;

#endif
/*hello.c*/
#include <stdio.h>
#include <stdlib.h>
#include "hello.h"

int main()
{
        val32_t i = 5;
        printf("hello, embedded world %d\n",i);
}
```

1．预处理阶段

GCC 的选项“-E”可以使编译器在预处理结束时就停止编译，选项“-o”是指定 GCC 输出的结果，其命令格式为如下所示。

```
gcc -E -o [目标文件] [编译文件]
```

表 1.6 指出后缀名为“.i”的文件是经过预处理的 C 原始程序。要注意，“hello.h”文件是不能进行编译的，因此，使编译器在预处理后停止的命令如下所示。

```
[root@localhost gcc]# gcc -E -o hello.i hello.c
```

在此处，选项‘-o’是指目标文件，由 1.6 表可知，‘.i’文件为已经过预处理的 C 原始程序。以下列出了 hello.i 文件的部分内容。

```
# 2 "hello.c" 2
# 1 "hello.h" 1

typedef unsigned long val32_t;
# 3 "hello.c" 2

int main()
{
 val32_t i = 5;
 printf("hello, embedded world %d\n",i);
}
```

由此可见，GCC 确实进行了预处理，它把“hello.h”的内容插入到 hello.i 文件中了。

2．编译阶段

编译器在预处理结束之后，GCC 首先要检查代码的规范性、是否有语法错误等，以确定代码的实际要做的工作，在检查无误后，就开始把代码翻译成汇编语言，GCC 的选项“-S”能使编译器在进行完编译之后就停止。由表 1.6 可知，“.s”是汇编语言原始程序，因此，此处的目标文件就可设为“.s”类型。

```
[root@localhost gcc]# gcc -S -o hello.s hello.i
```

以下列出了 hello.s 的内容，可见 GCC 已经将其转化为汇编了，感兴趣的读者可以分析一下这一行简单的 C 语言小程序用汇编代码是如何实现的。

```
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string "hello, embedded world %d\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        addl    $15, %eax
        addl    $15, %eax
        shrl    $4, %eax
        sall    $4, %eax
        subl    %eax, %esp
        movl    $5, -4(%ebp)
        subl    $8, %esp
        pushl   -4(%ebp)
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        leave
        ret
        .size   main, .-main
```

```
        .section        .note.GNU-stack,"",@progbits
        . .ident    "GCC: (GNU) 4.0.0 20050519 (Red Hat 4.0.0-8)"
```

可以看到，这一小段 C 语言的程序在汇编中已经复杂很多了，这也是 C 语言作为中级语言的优势所在。

3．汇编阶段

汇编阶段是把编译阶段生成的“.s”文件生成目标文件，读者在此使用选项“-c”就可看到汇编代码已转化为“.o”的二进制目标代码了。如下所示。

```
[root@localhost gcc]# gcc –c hello.s –o hello.o
```

4．链接阶段

在成功编译之后，就进入了链接阶段。在这里涉及一个重要的概念：函数库。
在这个程序中并没有定义“printf”的函数实现，在预编译中包含进的“stdio.h”中也只有该函数的声明，而没有定义函数的实现，那么，是在哪里实现“printf”函数的呢？
最后的答案是：系统把这些函数实现都已经放入名为 libc.so.6 的库文件中去了，在没有特别指定时，GCC 会到系统默认的搜索路径“/usr/lib”下进行查找，也就是链接到 libc.so.6 库函数中去，这样就能实现函数“printf”了，而这也就是链接的作用。
完成了链接之后，GCC 就可以生成可执行文件，其命令如下所示。

```
[root@localhost gcc]# gcc hello.o –o hello
```

运行该可执行文件，出现正确的结果。

```
[root@localhost gcc]# ./hello
hello, embedded world 5
```

1.3.3 GCC 警告提示

本节主要讲解 GCC 的警告提示功能。GCC 包含完整的出错检查和警告提示功能，它们可以帮助 Linux 程序员写出更加专业和高效的代码。
读者千万不能小瞧这些警告信息，在很多情况下，含有警告信息的代码往往会有意想不到的运行结果。
首先读者可以先看一下以下这段代码：

```
#include<stdio.h>

void main(void)
{
        long long tmp = 1;
        printf("This is a bad code!\n");
}
```

虽然这段代码运行的结果是正确的，但还有以下问题。

- main 函数的返回值被声明为 void，但实际上应该是 int。
- 使用了 GNU 语法扩展，即使用 long long 来声明 64 位整数，不符合 ANSI/ISO C 语言标准。
- main 函数在终止前没有调用 return 语句。

GCC 的警告提示选项有很多种类型，主要可分为“-Wall”类和非“-Wall”类。

1．Wall 类警告提示

这一类警告提示选项占了 GCC 警告选项的 90%以上，它不仅包含打开所有警告等功能，还可以单独对常见错误分别指定警告，这些常见的警告选项如表 1.7 所示（这些选项可供读者在实际操作时查阅使用）。

表 1.7　　GCC 的 Wall 类警告提示选项

选　　项	作　　用
-Wall	打开所有类型语法警告，建议读者养成使用该选项的习惯
-Wchar-subscripts	如果数组使用 char 类型变量做为下标值的话，则发出警告。因为在某些平台上可能默认为 signed char，一旦溢出，就可能导致某些意外的结果
-Wcomment	当'/*'出现在 '/* ... */'注释中，或者'\'出现在'// ...'注释结尾处时，使用-Wcomment 会给出警告，它很可能会影响程序的运行结果
-Wformat	检查 printf 和 scanf 等格式化输入输出函数的格式字符串与参数类型的匹配情况，如果发现不匹配则发出警告。某些时候格式字符串与参数类型的不匹配会导致程序运行错误，所以这是个很有用的警告选项
-Wimplicit	该警告选项实际上是-Wimplicit-int 和-Wimplicit-function-declaration 两个警告选项的集合。前者在声明函数却未指明函数返回类型时给出警告，后者则是在函数声明前调用该函数时给出警告
-Wmissing-braces	当聚合类型或者数组变量的初始化表达式没有充分用括号{}括起时，给出警告
-Wparentheses	这是一个很有用的警告选项，它能帮助用户从那些看起来语法正确但却由于操作符优先级或者代码结构“障眼”而导致错误运行的代码中解脱出来
-Wsequence-point	关于顺序点（sequence point），在 C 标准中有解释，不过很晦涩。我们在平时编码中尽量避免写出与实现相关、受实现影响的代码便是了。而-Wsequence-point 选项恰恰可以帮我们这个忙，它可以帮我们查出这样的代码来，并给出其警告
-Wswitch	这个选项的功能浅显易懂，通过文字描述也可以清晰地说明。当以一个枚举类型（enum）作为 switch 语句的索引时但却没有处理 default 情况，或者没有处理所有枚举类型定义范围内的情况时，该选项会给出警告
-Wunused-function	警告存在一个未使用的 static 函数的定义或者存在一个只声明却未定义的 static 函数
-Wunused-label	用来警告存在一个使用了却未定义或者存在一个定义了却未使用的 label
-Wunused-variable	用来警告存在一个定义了却未使用的局部变量或者非常量 static 变量
-Wunused-value	用来警告一个显式计算表达式的结果未被使用
-Wunused-parameter	用来警告一个函数的参数在函数的实现中并未被用到
-Wuninitialized	该警告选项用于检查一个局部自动变量在使用之前是否已经初始化了或者在一个 longjmp 调用可能修改一个 non-volatile automatic variable 时给出警告

这些警告提示读者可以根据自己的不同情况进行相应的选择，这里最为常用的是“-Wall”，上面的这一小段程序使用该警告提示后的结果是：

```
[root@ft charpter2]# gcc -Wall wrong.c -o wrong
wrong.c:4: warning: return type of 'main' is not 'int'
wrong.c: In function 'main':
wrong.c:5: warning: unused variable 'tmp'
```

可以看出，使用‘-Wall’选项找出了未使用的变量 tmp 以及返回值的问题，但没有找出无效数据类型的错误。

2．非 Wall 类警告提示

非 Wall 类的警告提示中最为常用的有以下两种：“-ansi”和“-pedantic”。

（1）“-ansi”

该选项强制 GCC 生成标准语法所要求的告警信息，尽管这还并不能保证所有没有警告的程序都是符合 ANSI C 标准的。使用该选项的运行结果如下所示：

```
[root@ft charpter2]# gcc -ansi wrong.c -o wrong
wrong.c: In function 'main':
wrong.c:4: warning: return type of 'main' is not 'int'
```

可以看出，该选项并没有发现“long long”这个无效数据类型的错误。

（2）“-pedantic”

该选项允许发出 ANSI C 标准所列的全部警告信息，同样也保证所有没有警告的程序都是符合 ANSI C 标准的。使用该选项的运行结果如下所示：

```
[root@ft charpter2]# gcc -pedantic wrong.c -o wrong
wrong.c: In function 'main':
wrong.c:5: warning: ISO C90 does not support 'long long'
wrong.c:4: warning: return type of 'main' is not 'int'
```

可以看出，使用该选项查看出了“long long”这个无效数据类型的错误。

1.3.4 GCC 使用库函数

1．Linux 函数库介绍

函数库可以看做是事先编写的函数集合，它可以与主函数分离，使得程序模块化，从而增加代码的复用性。Linux 中函数库包括两类：静态库和共享库。

静态库的代码在编译时就已连接到开发人员开发的应用程序中，而共享库是在程序开始运行时被加载。由于在使用共享库时程序中并不包括库函数的实现代码，只是包含了对库函数的引用，因此程序代码的规模比较小。

系统中可用的库都安装在/usr/lib 和/lib 目录下。库文件名由前缀 lib 和库名以及后缀组成。根据库的类型不同，后缀名也不一样。

注意 共享库的后缀名由.so 和版本号组成。
静态库的后缀名为.a。

如：数学共享库的库名为 libm.so.5，这里的标识字符为 m，版本号为 5，libm.a 则是静态数学库。在 Linux 系统中系统所用的库都存放在/usr/lib 和/lib 目录中。

2．相关路径选项

有些时候库文件并不存放在系统默认的路径下。因此，要通过路径选项来指定相关的库文件位置，这里首先介绍两个常用选项的使用方法。

（1）“-I <dir>”

GCC 使用缺省的路径来搜索头文件，如果想要改变搜索路径，用户可以使用“-I”选项。“-I<dir>”选项可以在头文件的搜索路径列表中添加<dir>目录。这样，GCC 就会到指定的目录去查找相应的头文件。

比如在“/root/workplace/gcc”下有两个文件：

```
hello.c
#include <my.h>
int main()
{
     printf("Hello!!\n");
     return 0;
}
```

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

这样，就可在 GCC 命令行中加入“-I”选项，其命令如下所示。

```
[root@localhost gcc] gcc hello.c –I/root/workplace/gcc/ -o hello
```

这样，GCC 就能够执行出正确结果。

小技巧 在 include 语句中，“<>”表示在标准路径中搜索头文件，在 Linux 中默认为“/usr/include”。故在上例中，可把 hello1.c 的“#include <my.h>”改为“#include "my.h"”，这样就不需要加上“-I”选项了。

（2）“-L <dir>”

选项“-L <dir>”的功能与“-I <dir>”类似，其区别就在于“-L”选项是用于指明库文件的路径。例如有程序 hello_sq.c 需要用到目录“/root/workspace/gcc/lib”下的一个动态库 libsunq.so，则只需键入如下命令即可。

```
[root@localhost gcc] gcc hello_sq.c –L/root/workspace/gcc/lib –lsunq –o hello_sq
```

注意 ‘-I <dir>’和‘-L< dir>’都只是指定了路径，而没有指定文件，因此不能在路径中包含文件名。

3．使用不同类型链接库

使用不同类型的链接库的方法很相似，都是使用选项是“-l”（注意这里是小写的“L”）。该选项是用于指明具体使用的库文件。由于在 Linux 中函数库的命名规则都是以“lib”开头的，因此，这里的库文件只需填写 lib 之后的内容即可。

如：有静态库文件 libm.a，在调用时只需写作“-lm”；同样对于共享库文件 libm.so，在调用时也只需写作“-lm”即可，其整体调用命令类似如下：

```
[root@localhost gcc] gcc -o dynamic -L /root/lq/testc/lib/dynamic.o -lmydynamic
```

那么，若系统中同时存在库名相同的静态库文件和共享库文件时，该链接选项究竟会调用静态库文件还是共享库文件呢？

经测试后可以发现，系统缺省链接的是共享库，这是由于 Linux 系统中默认的是采用动态链接的方式。如果用户要链接同名的静态库，则在“-l”之前需要添加选项“-static”。例如：链接 libm.a 库文件的选项是“-static -lm”。

1.3.5 GCC 代码优化

GCC 可以对代码进行优化，它通过编译选项-On 来控制优化代码的生成，其中 n 是一个代表优化级别的整数。对于不同版本的 GCC 来讲，n 的取值范围及其对应的优化效果可能并不完全相同，比较典型的范围是从 0 到 2 或 3。

不同的优化级别对应不同的优化处理工作。如使用优化选项-O 主要进行线程跳转（Thread Jump）和延迟退栈（Deferred Stack Pops）两种优化。使用优化选项-O2 除了完成所有-O1 级别的优化之外，同时还要进行一些额外的调整工作，如处理器指令调度等；选项-O3 则还包括循环展开和其他一些与处理器特性相关的优化工作。

虽然优化选项可以加快代码的运行速度，但对于调试而言将是一个很大的挑战。因为代码在经过优化之后，原先在源程序中声明和使用的变量很可能不再使用，控制流也可能会突然跳转到其他的地方，循环语句也有可能因为循环展开而变得到处都有，所有这些都将使调试工作异常艰难。
建议开发人员在调试程序的时候不使用任何优化选项，只有当程序完成调试，最终发行的时候再考虑对其进行优化。

1.4 嵌入式 Linux 调试器 GDB 的使用

在程序编译通过生成可执行文件之后，就进入了程序的调试环节。调试一直是程序开发中的重中之重，如何使程序员能够迅速找到错误的原因是一款调试器的首要目标。
GDB 是 GNU 开源组织发布的一个 Linux 下的程序调试工具，它是一种强大的命令行调试工具。
一个出色的调试器需要有以下几项功能。

- 能够运行程序，设置所有能影响程序运行的参数。
- 能够让程序在指定的条件下停止运行。
- 能够在程序停止时检查所有参数的情况。
- 能够根据指定条件改变程序的运行。

1.4.1 GDB 使用实例

下面通过一个简单的实例使读者对 GDB 有一个感性的认识，这里所介绍的指令都是 GDB 中最为基本也是最为常用的指令，希望读者能够动手操作，掌握 GDB 的使用方法。
首先，有以下程序段。

```
#include <stdio.h>

/*子函数 add: 将自然数从 1～m 相加*/
int add(int m)
{
         int i,n=0;
         for(i=1; i<=m;i++)
             n += i;
         printf("The sum of 1-%d in add is %d\n", m,n);
}

int main()
{
       int i,n=0;
       add(50);
       for(i=1; i<=50; i++)
           n += i;
       printf("The sum of 1-50 is %d \n", n );
}
```

注意将此程序用 GCC 进行编译时要加上“-g”选项。

1．进入 GDB

进入 GDB 只需输入 GDB 和要调试的可执行文件即可，如下所示：

```
[root@localhost gdb]# gdb test
GNU gdb Red Hat Linux (6.3.0.0-1.21rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library
"/lib/libthread_db.so.1".
(gdb)
```

可以看出，在 GDB 的启动画面中指出了 GDB 的版本号、使用的库文件等信息，接下来就进入了由“(gdb)”开头的命令行界面了。

2．查看文件

在 GDB 中键入‘l’（list）就可以查看所载入的文件，如下所示：

```
(gdb) l
4       {
5               int i,n=0;
6               for(i=1; i<=m;i++)
7                 n += i;
8               printf("The sum of 1-%d in add is %d\n", m,n);
9       }
10
11      int main()
12      {
13          int i,n=0;
(gdb) l
14          add(50);
15          for(i=1; i<=50; i++)
16           {
17             n += i;
18           }
19          printf("The sum of 1-50 is %d \n", n );
20
21      }
22
```

可以看出，GDB 列出的源代码中明确地给出了对应的行号，这样可以大大地方便代码的定位。

> **注意** 在一般情况下，源代码中的行号与用户书写程序中的行号是一致的，但有时由于用户的某些编译选项会导致行号不一致的情况，因此，一定要查看在 GDB 中的行号。

3．设置断点

设置断点可以使程序执行到某个位置时暂时停止，程序员在该位置处可以方便地查看变量的值、堆栈情况等，从而找出问题的症结所在。

在 GDB 中设置断点非常简单，只需在“b”后加入对应的行号即可（这是最常用的方法），其命令如下所示：

```
(gdb) b 6
Breakpoint 1 at 0x804846d: file test.c, line 6.
```

要注意的是，在 GDB 中利用行号设置断点是指代码运行到对应行之前暂停，如上例中，代码运行到第 6 行之前暂停（并没有运行第 6 行）。

4．查看断点处情况

在设置完断点之后，用户可以键入“info b”来查看断点设置情况。在 GDB 中可以设置多个断点。

```
(gdb) info b
Num Type         Disp Enb Address    What
1   breakpoint    keep y   0x0804846d in main at test.c:6
```

5．运行代码

接下来就可运行代码了，GDB 默认从首行开始运行代码，可键入“r”（run）即可，在“r”后面加上行号即可从程序中指定行开始运行。

```
(gdb) r
Starting program: /home/yul/book/test

Breakpoint 1, add (m=50) at test.c:6
6            for(i=1; i<=m;i++)
```

可以看到，程序运行到断点处就停止了。

6．查看变量值

在程序停止运行之后，程序员可以查看断点处的相关变量值。在 GDB 中只需键入“p 变量名”即可，如下所示：

```
(gdb) p n
$1 = 0
(gdb) p i
$2 = 134518440
```

在此处，为什么变量 i 的值是如此奇怪的一个数字呢？原因就在于程序是在断点设置的对应行之前停止的，此时代码没有把变量 i 的值赋为 0，而只是一个随机的数字。但变量 n 是在程序第 5 行赋值的，所以此时 n 的值已经为 0。

小技巧 GDB 在显示变量值时都会在对应值之前加上“$N”标记，它是当前变量值的引用标记，所以以后若想再次引用此变量就可以直接写作“$N”，而无需写冗长的变量名。

7．观察变量

在某一循环处，程序员往往希望能够观察一个变量的变化情况，这时就可以键入命令“watch”来观察变量的变化情况，如下所示：

```
(gdb) watch n
Hardware watchpoint 2: n
```

可以看到，GDB 在“n”设置了观察点。

注意 在此处必须键入完整的命令“watch”，因为在 GDB 中有不少以‘w’开头的命令，如“where”、“while”等。

8．单步运行

单步运行是指一次只运行一条语句，这样可以方便程序员来查看程序运行的结果，在此处只需键入“n”（next）即可。

```
(gdb) n
```

```
7               n += i;
(gdb) n
Hardware watchpoint 2: n

Old value = 15
New value = 21
```

可以看到，随着程序的单步运行，当变量 n 的值发生变化时，GDB 就会自动显示出 n 的变化情况。

9．程序继续运行

命令“c”（continue）可以使程序继续往下运行，直到再次遇到断点或程序结束。如下所示：

```
(gdb) c
Continuing.
The sum of 1-50 is 1275

Program exited with code 031.
```

10．退出 GDB

退出 GDB 只需使用指令“q”（quit）即可，如下所示：

```
(gdb) q
[root@localhost gcc]
```

以上所介绍的是 GDB 中最为常见的命令，下面几节将会详细讲解其他的一些命令。

1.4.2 设置/删除断点

GDB 中有丰富的断点设置、删除命令，可以满足用户各个方面的需求。表 1.8 列出了 GDB 中常见的断点设置及删除命令。

表 1.8 GCC 中常见断点设置与删除指令

命 令 格 式	作 用
break+设置断点的行号	用于在程序中对应行设置断点
tbreak+行号或函数名	设置临时断点，到达后被自动删除
break+filename+行号	用于在指定文件的对应行设置断点
break+<0x…>	用于在内存某一位置处暂停
break+行号+if+条件	用于设置条件断点，在循环中使用非常方便
info breakpoints/watchpoints	查看断点/观察点的情况
clear+要清除断点的行号	用于清除对应行的断点
delete+要清除断点的编号	用于清除断点和自动显示的表达式的命令。与 clear 的不同之处：clear 要给出断点的行号，delete 要给出断点的编号。用 clear 命令清除断点时 GDB 会给出提示，而用 delete 清除断点时 GDB 不会给出任何提示
disable+断点编号	让所设断点暂时失效。如果要让多个编号处的断点失效可将编号之间用空格隔开
enable+断点编号	与 disable 相反
awatch+变量	设置一个观察点，当变量被读出或写入时程序被暂停
rwatch+变量	设置一个观察点，当变量被程序读时，程序被暂停
watch+变量	同 awatch

小知识 在多线程的程序中，观察点的作用很有限，gdb 只能观察在一个线程中的表达式的值。如果用户确信表达式只被当前线程所存取，那么使用观察点才有效。gdb 不能注意一个非当前线程对表达式值的改变。

1.4.3 数据相关命令

在 GDB 中也有丰富的数据显示相关命令，他们可以使用户可以以各种形式显示所要查看的数据，数据相关命令如表 1.9 所示。

表 1.9 GDB 中数据相关指令

命 令 格 式	作 用
display+表达式	该命令用于显示表达式的值，使用了该命令后，每当程序运行到断点处都会显示表达式的值
info display	用于显示当前所有要显示值的表达式的有关情况
delete+display 编号	用于删除一个要显示值的表达式，调用这个命令删除一个表达式后，被删除的表达式将不被显示
disable+display 编号	使一个要显示的表达式暂时无效
enable+display 编号	disable diplay 的反操作
undisplay+display 编号	用于结束某个表达式值的显示
whatis+变量	显示某个表达式的数据类型
print(p)+变量或表达式	用于打印变量或表达式的值
set+变量＝变量值	改变程序中一个变量的值

小技巧 在使用 print 命令时，可以对变量按指定格式进行输出，其命令格式为：print /变量名+格式
其中格式有以下几种方式。
X：十六进制；d：十进制；u：无符号数；o：八进制；
T：二进制；a：十六进制打印；c：字符格式；f：浮点数。

1.4.4 调试运行环境相关命令

在 GDB 中控制程序的运行也是非常方便地，用户可以自行设定变量值、调用函数等，其具体命令如表 1.10 所示。

表 1.10 GDB 调试运行环境相关命令

命 令 格 式	作 用
set args	设置运行参数
show args	参看运行参数
set width+数目	设置 GDB 的行宽
cd+工作目录	切换工作目录
run	程序开始执行
step（s）	进入式（会进入到所调用的子函数中）单步执行
next（n）	非进入式（不会进入到所调用的子函数中）单步执行
finish	一直运行到函数返回
until+行数	运行到函数某一行

continue（c）	执行到下一个断点或程序结束
return<返回值>	改变程序流程，直接结束当前函数，并将指定值返回
call+函数	在当前位置执行所要运行的函数

1.4.5 堆栈相关命令

gdb 中也提供了多种堆栈相关的命令，可以查看堆栈的情况、寄存器的情况等，其具体命令如表 1.11 所示。

表 1.11 GDB 中堆栈相关命令

命 令 格 式	作 用
backtrace 或 bt	用来打印栈侦指针，也可以在该命令后加上要打印的栈侦指针的个数
frame	该命令用于打印栈侦
info reg	查看寄存器使用情况
info stack	查看堆栈情况
up	跳到上一层函数
down	与 up 相对

1.5 make 工程管理器

前面几节主要介绍如何在 Linux 环境下使用文本编辑器，如何使用 GCC 编译出可执行文件，以及如何使用 GDB 来调试程序。既然所有的工作都已经完成了，为什么还需要 make 这个工程管理器呢？

工程管理器可以用来管理较多的文件。读者可以试想一下：一个由上百个源文件构成的项目，如果其中只有一个或少数几个文件进行了修改，按照之前所学的 GCC 的用法，就不得不把这所有的文件重新编译一遍。原因就在于编译器并不知道哪些文件是最近更新的，所以，程序员就不得不处理所有的文件来完成重新编译工作。

显然，开发人员需要一个能够自动识别出那些被更新的代码文件并实现整个工程自动编译的工具。

实际上，make 就是一个自动编译管理器，能够根据文件时间戳自动发现更新过的文件从而减少编译的工作量。同时，它通过读入 Makefile 文件的内容来执行大量的编译工作，用户只需一次编写简单的编译语句即可。它大大提高了项目开发和维护的工作效率，几乎所有嵌入式 Linux 下的项目编程均会涉及 make 管理器，希望读者能够认真学习本节内容。

1.5.1 Makefile 基本结构

Makefile 用来告诉 make 如何编译和连接一个程序，是 make 读入的惟一配置文件，本节主要讲解 Makefile 的编写规则。

在一个 Makefile 中通常包含如下内容。

- 需要由 make 工具创建的目标体（target），目标体通常是目标文件、可执行文件或是一个标签。
- 要创建的目标体所依赖的文件（dependency_file）。
- 创建每个目标体时需要运行的命令（command）。

它的格式为：

```
target: dependency_files
```

```
    command
```

例如，有两个文件分别为 hello.c 和 hello.h，希望创建的目标体为 hello.o，执行的命令为 gcc 编译指令：gcc –c hello.c，那么，对应的 Makefile 就可以写为以下形式：

```
#The simplest example
hello.o: hello.c hello.h
    gcc -c hello.c -o hello.o
```

接着就可以使用 make 了。使用 make 的格式为：make target，这样 make 就会自动读入 Makefile（也可以是首字母小写 makefile）执行对应 target 的 command 语句，并会找到相应的依赖文件，如下所示：

```
[root@localhost makefile]# make hello.o
gcc -c hello.c -o hello.o
[root@localhost makefile]# ls
hello.c  hello.h  hello.o  Makefile
```

可以看到，Makefile 执行了“hello.o”对应的命令语句，并生成了“hello.o”目标体。

注意 在 Makefile 中的每一个 command 前必须有“Tab”符，否则在运行 make 命令时会出错。

上面示例的 Makefile 在实际中是几乎不存在的，因为它过于简单，仅包含两个文件和一个命令，在这种情况下完全不需要编写 Makefile 而只需在 Shell 中直接输入命令即可。在实际中使用的 Makefile 往往是包含很多的命令的，一个项目也会包含多个 Makefile。

下面就对较复杂的 Makefile 进行讲解。以下这个工程包含有 3 个头文件和 8 个 C 文件，其 Makefile 如下所示：

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

main.o : main.c defs.h
         gcc -c main.c –o main.o
kbd.o : kbd.c defs.h command.h
        gcc -c kbd.c –o kbd.o
command.o : command.c defs.h command.h
            gcc -c command.c –o command.o
display.o : display.c defs.h buffer.h
            gcc -c display.c –o display.o
insert.o : insert.c defs.h buffer.h
           gcc -c insert.c –o insert.o
search.o : search.c defs.h buffer.h
           gcc -c search.c –o search.o
files.o : files.c defs.h buffer.h command.h
          gcc -c files.c –o files.o
utils.o : utils.c defs.h
          gcc -c utils.c –o utils.o
clean :
         rm edit main.o kbd.o command.o display.o \
            insert.o search.o files.o utils.o
```

这里的反斜杠“\”是换行符的意思，用于增加 Makefile 的可读性。读者可以把这些内容保存在文件名为 Makefile 或 makefile 的文件中，然后在该目录下直接输入命令 make 就可以生成可执行文件 edit。如果想要删除可执行文件和所有的中间目标文件，只需要简单地执行一下 make clean 即可。

在这个 makefile 中，目标文件（target）包含以下内容：可执行文件 edit 和中间目标文件*.o，依赖文件（dependency_file）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 文件又是可执行文件 edit 的依赖文件。依赖关系表明目标文件是由哪些文件生成的。换言之，目标文件是由哪些文件更新的。

在定义好依赖关系后，后面的一行命令定义了如何生成目标文件。请读者注意，这些命令都是以一个 Tab 键作为开头的。

值得注意的是，make 工程管理器并不关心命令是如何工作的，它只负责执行用户事先定义好的命令。同时，make 还会比较目标文件和依赖文件的最后修改日期，如果依赖文件的日期要比目标文件的日期更新，或者目标文件并不存在的话，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称，也可称其为标签，不依赖于其他任何文件。

若用户想要执行其后的命令，就要在 make 命令后显示地指出这个标签的名字。这个方法非常有用，通常用户可以在一个 Makefile 中定义一些和编译无关的命令，比如程序的打包、备份或删除等等。

1.5.2 Makefile 变量

为了进一步简化 Makefile 的编写和维护，make 允许在 Makefile 中创建和使用变量。变量是在 Makefile 中定义的名字，用来代替一个文本字符串，该文本字符串称为该变量的值。

变量的值可以用来代替目标体、依赖文件、命令以及 Makefile 文件中其他部分。在 Makefile 中的变量定义有两种方式：一种是递归展开方式，另一种是简单方式。

递归展开方式定义的变量是在引用该变量时进行替换的，即如果该变量包含了对其他变量的引用，则在引用该变量时一次性将内嵌的变量全部展开。虽然这种类型的变量能够很好地完成用户的指令，但是它也有严重的缺点，如不能在变量后追加内容，因为语句“CFLAGS = $(CFLAGS) –O”在变量扩展过程中可能导致无穷循环。

为了避免上述问题，简单扩展型变量的值在定义处展开，并且只展开一次，因此它不包含任何对其他变量的引用，从而消除了变量的嵌套引用。

递归展开方式的定义格式为：VAR=var。

简单扩展方式的定义格式为：VAR：=var。

Make 中的变量使用均使用格式为：$(VAR)

注意

变量名是不包括‘:’，‘#’，‘=’、结尾空格的任何字符串。同时，变量名中包含字母、数字以及下划线以外的情况应尽量避免，因为它们可能在将来被赋予特别的含义，

变量名是大小写敏感的，例如变量名‘foo’、‘FOO’和‘Foo’代表不同的变量。

推荐在 Makefile 内部使用小写字母作为变量名，预留大写字母作为控制隐含规则参数或用户重载命令选项参数的变量名。

在上面的例子中，先来看看 edit 这个规则：

```
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
```

读者可以看到“.o”文件的字符串被重复了两次，如果在工程需要加入一个新的“.o”文件，那么用户需要在这两处分别加入（其实应该是有 3 处，另外一处在 clean 中）。

当然，这个实例的 Makefile 并不复杂，所以在这两处分别添加也没有太多的工作量。但如果 Makefile 变得复杂，那么用户就很有可能会忽略一个需要加入的地方，从而导致编译失败。所以，为了使 Makefile 易维护，推荐在 Makefile 中尽量使用变量这种形式。

这样，用户在这个实例中就可以按以下的方式来定义变量：

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

这里是以递归展开的方式来进行定义的。在此之后，用户就可以很方便地在 Makefile 中以 $(objects) 的方式来使用这个变量了，于是改良版 Makefile 就变为：

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

edit : $(objects)
       gcc -o edit $(objects)
main.o : main.c defs.h
       gcc -c main.c –o main.o
kbd.o : kbd.c defs.h command.h
       gcc -c kbd.c –o kbd.o
command.o : command.c defs.h command.h
       gcc -c command.c –o command.o
display.o : display.c defs.h buffer.h
       gcc -c display.c –o display.o
insert.o : insert.c defs.h buffer.h
       gcc -c insert.c –o insert.o
search.o : search.c defs.h buffer.h
       gcc -c search.c –o search.o
files.o : files.c defs.h buffer.h command.h
       gcc -c files.c –o files.o
utils.o : utils.c defs.h
       gcc -c utils.c –o utils.o
clean :
       rm edit $(OBJS)
```

可以看到，如果这时又有新的“.o”文件需要加入，用户只需简单地修改一下变量 OBJS 的值就可以了。Makefile 中的变量分为用户自定义变量、预定义变量、自动变量及环境变量。如上例中的 OBJS 就属于用户自定义变量，其值由用户自行设定。预定义变量和自动变量无需定义就可以在 Makefile 中使用，其中部分有默认值，当然用户也可以对其进行修改。

预定义变量包含了常见编译器、汇编器的名称及编译选项，表 1.12 列出了 Makefile 中常见预定义变量及其部分默认值。

表 1.12　　Makefile 中常见预定义变量

命 令 格 式	含　义
AR	库文件维护程序的名称，默认值为 ar
AS	汇编程序的名称，默认值为 as
CC	C 编译器的名称，默认值为 cc
CPP	C 预编译器的名称，默认值为$(CC) –E
CXX	C++编译器的名称，默认值为 g++
FC	FORTRAN 编译器的名称，默认值为 f77
RM	文件删除程序的名称，默认值为 rm –f
ARFLAGS	库文件维护程序的选项，无默认值
ASFLAGS	汇编程序的选项，无默认值
CFLAGS	C 编译器的选项，无默认值

CPPFLAGS	C 预编译的选项，无默认值
CXXFLAGS	C++编译器的选项，无默认值
FFLAGS	FORTRAN 编译器的选项，无默认值

上例中的 CC 和 CFLAGS 是预定义变量，其中由于 CC 没有采用默认值，因此，需要把“CC=gcc”明确列出来。

由于常见的 gcc 编译语句中通常包含了目标文件和依赖文件，而这些文件在 Makefile 文件中目标体的一行已经有所体现，因此，为了进一步简化 Makefile 的编写，引入了自动变量。

自动变量通常可以代表编译语句中出现的目标文件和依赖文件等，并且具有本地含义（即下一语句中出现的相同变量代表的是下一语句的目标文件和依赖文件），表 1.13 列出了 Makefile 中常见自动变量。

表 1.13　　Makefile 中常见自动变量

命 令 格 式	含　　义
$*	不包含扩展名的目标文件名称
$+	所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件
$<	第一个依赖文件的名称
$?	所有时间戳比目标文件晚的依赖文件，并以空格分开
$@	目标文件的完整名称
$^	所有不重复的依赖文件，以空格分开
$%	如果目标是归档成员，则该变量表示目标的归档成员名称

自动变量的书写比较难记，但是在熟练了之后会非常地方便，请读者结合下例中的自动变量改写的 Makefile 进行记忆。

```
OBJS = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
CC = gcc
CFLAGS = -Wall -O -g
edit : $(objects)
          $(CC) $^ -o $@
main.o : main.c defs.h
        (CC) $(CFLAGS) -c $< -o $@
kbd.o : kbd.c defs.h command.h
        (CC) $(CFLAGS) -c $< -o $@
command.o : command.c defs.h command.h
        (CC) $(CFLAGS) -c $< -o $@
display.o : display.c defs.h buffer.h
        (CC) $(CFLAGS) -c $< -o $@
insert.o : insert.c defs.h buffer.h
        (CC) $(CFLAGS) -c $< -o $@
search.o : search.c defs.h buffer.h
        (CC) $(CFLAGS) -c $< -o $@
files.o : files.c defs.h buffer.h command.h
        (CC) $(CFLAGS) -c $< -o $@
utils.o : utils.c defs.h
        (CC) $(CFLAGS) -c $< -o $@
clean :
        rm edit $(OBJS)
```

另外，在 Makefile 中还可以使用环境变量。使用环境变量的方法相对比较简单，make 在启动时会自动读取系统当前已经定义了的环境变量，并且会创建与之具有相同名称和数值的变量。但是，如果用户在 Makefile 中定义了相同名称的变量，那么用户自定义变量将会覆盖同名的环境变量。

1.5.3 Makefile 规则

Makefile 的规则包括目标体、依赖文件及其间的命令语句，是 make 进行处理的依据。Makefile 中的一条语句就是一个规则。
在上面的例子中显示地指出了 Makefile 中的规则关系，如“$(CC) $(CFLAGS) -c $< -o $@”，为了简化 Makefile 的编写，make 还定义了隐式规则和模式规则，下面就分别对其进行讲解。

1．隐式规则

隐含规则能够告诉 make 怎样使用传统的技术完成任务，这样，当用户使用它们时就不必详细指定编译的具体细节，而只需把目标文件列出即可。make 会自动搜索隐式规则目录来确定如何生成目标文件，如上例可以写成：

```
OBJS = main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
CC = gcc
CFLAGS = -Wall -O -g
edit :$(objects)
          $(CC) $^ -o $@
main.o : main.c defs.h
kbd.o : kbd.c defs.h command.h
command.o : command.c defs.h command.h
display.o : display.c defs.h buffer.h
insert.o : insert.c defs.h buffer.h
search.o : search.c defs.h buffer.h
files.o : files.c defs.h buffer.h command.h
utils.o : utils.c defs.h
clean :
          rm edit $(OBJS)
```

为什么可以省略“(CC) $(CFLAGS) -c $< -o $@”这句呢？
因为 make 的隐式规则指出：所有“.o”文件都可自动由“.c”文件使用命令“$(CC) $(CPPFLAGS) $(CFLAGS) -c file.c –o file.o”生成。因此，Makefile 就可以进一步地简化了。

注意 在隐式规则只能查找到相同文件名的不同后缀名文件，如“kang.o”文件必须由“kang.c”文件生成。

表 1.14 给出了常见的隐式规则目录。

表 1.14 Makefile 中常见隐式规则目录

对应语言后缀名	规　则
C 编译：.c 变为.o	$(CC) –c $(CPPFLAGS) $(CFLAGS)
C++编译：.cc 或.C 变为.o	$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)
Pascal 编译：.p 变为.o	$(PC) -c $(PFLAGS)
Fortran 编译：.r 变为-o	$(FC) -c $(FFLAGS)。

2．模式规则

隐式规则仅仅能够用 make 默认的变量来进行操作。
模式规则不同于隐式规则，是用来定义相同处理规则的多个文件的，模式规则能引入用户自定义变量，为多个文件建立相同的规则，简化 Makefile 的编写。
模式规则的格式类似于普通规则，这个规则中的相关文件前必须用“%”标明，然而在这个实例中，并不能使用这个模式规则。

1.5.4 make 使用

使用 make 管理器非常简单，只需在 make 命令的后面键入目标名即可建立指定的目标，如果直接运行 make，则建立 Makefile 中的第一个目标。
此外 make 还有丰富的命令行选项，可以完成各种不同的功能，表 1.15 列出了常用的 make 命令行选项。

表 1.15 make 的命令行选项

命 令 格 式	含 义
-C dir	读入指定目录下的 Makefile
-f file	读入当前目录下的 file 文件作为 Makefile
-i	忽略所有的命令执行错误
-I dir	指定被包含的 Makefile 所在目录
-n	只打印要执行的命令，但不执行这些命令
-p	显示 make 变量数据库和隐含规则
-s	在执行命令时不显示命令
-w	如果 make 在执行过程中改变目录，打印当前目录名

1.6 Eclipse 集成开发环境

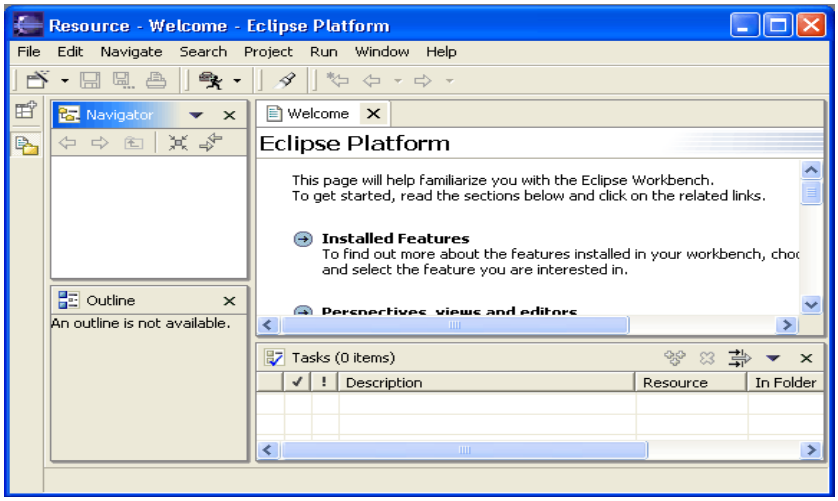
1.6.1 eclipse 简介

eclipse 是著名的跨平台的集成开发环境（IDE），最初是由 IBM 公司开发的替代商业软件 Visual Age For Java 的下一代 IDE 开发环境。2001 年 11 月贡献给开源社区，现在它由非营利软件供应商联盟 eclipse 基金会（Eclipse Foundation）管理。2005 年 7 月，稳定版 3.1.0 发布，目前最新的稳定版本为 3.4。

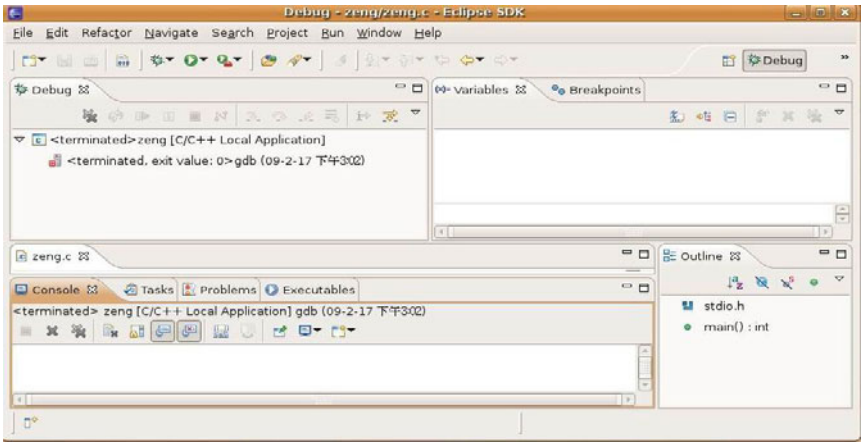
eclipse 本身只是一个框架平台，最初主要用于 Java 语言的开发。但是众多插件的支持使得 eclipse 可以支持其他语言的开发，如 C/C++、C#、Perl、Cobol 等等。许多软件开发商以 eclipse 为框架开发了己的 IDE。 由于 eclipse 平台用 Java 实现，所以运行时需要 jre(Java Runtime Environment)的支持。

1.6.2 eclipse 相关术语

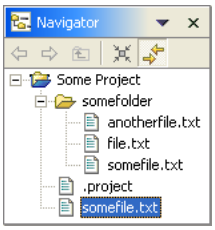
- 工作台(workbench) ：提供了一个或多个透视图。

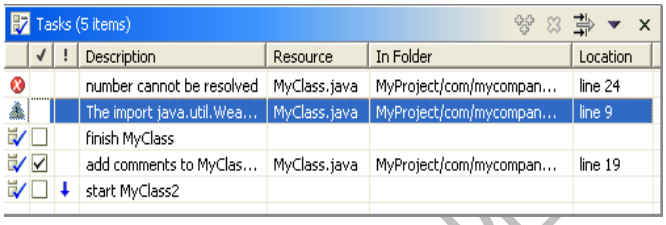


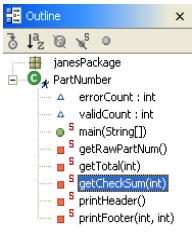
- 透视图(perspective) ：工作台中的一组视图和编辑器。每个透视图包括一组不同的视图，它定义了视图在工作台中的布局。

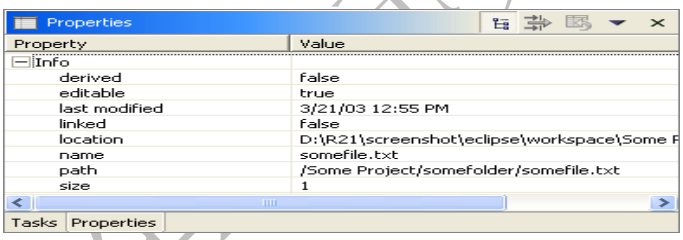


♦ 视图(view) ：工作台内的可视组件，通常用来浏览分层信息。

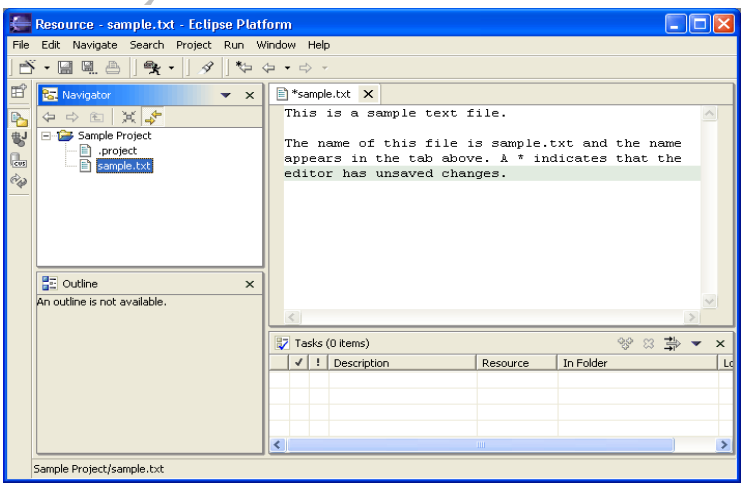








♦ 编辑器(editor) ：工作台内的可视组件，通常用来编辑或浏览信息。



1.6.3 安装 eclipse 集成开发环境（假设宿主机环境为 ubuntu8.10）

1. 下载相关软件包

运行 eclipse 需要依次安装 jre(Java 运行环境)、eclipse 开发包和 cdt(c/c++开发插件)。软件下载链接如下所示：

- ✧ jdk-6u7-linux-i586.bin (http://java.sun.com/javase/downloads/index.jsp)
- ✧ eclipse-SDK-3.4-linux-gtk.tar.gz (http://www.eclipse.org/downloads)
- ✧ cdt-master-5.0.0.zip (http://www.eclipse.org/cdt/downloads.php)

2. 安装 jdk

```
# cd  /opt
# bash  jdk-6u7-linux-i586.bin
# mv  jdk1.6.0_07  java
# cd  /opt/java
# mv  /etc/alternatives/java  /etc/alternatives/java.gnu
# ln  -s  /opt/java/bin/java  /etc/alternatives/java
# export  JAVA_HOME=/opt/java/
```

3. 安装 eclipse

```
# cd  /opt
# tar  zxvf  eclipse-SDK-3.4-linux-gtk.tar.gz
# export  PATH=/opt/eclipse:$PATH
```

4. 安装 cdt（c/c++ development toolkit）

```
# mkdir  -p  /opt/cdt
# cd  /opt/cdt
# unzip  cdt-master-5.0.0.zip
# cp  -r  plugins/*  /opt/eclipse/plugins/
# cp  -r  features/*  /opt/eclipse/features/
```

1.6.4 eclipse 的使用

在上一节中我们学习了如何搭建 eclipse 集成开发环境。下面我们将和读者一起学习如何在 eclipse 中创建工程，编译和调试应用程序。

- ➢ 在命令行下输入 eclipse 后回车，进入开发界面，如图 1.4

```
# eclipse
```

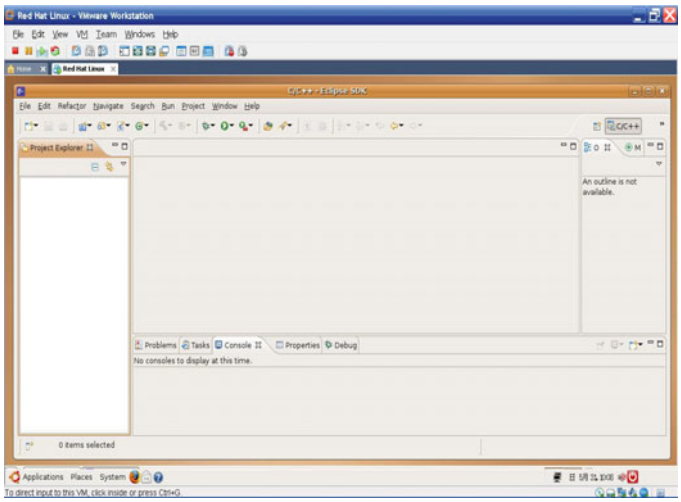


图 1.4

- 选择 File --> New --> Project --> C Project

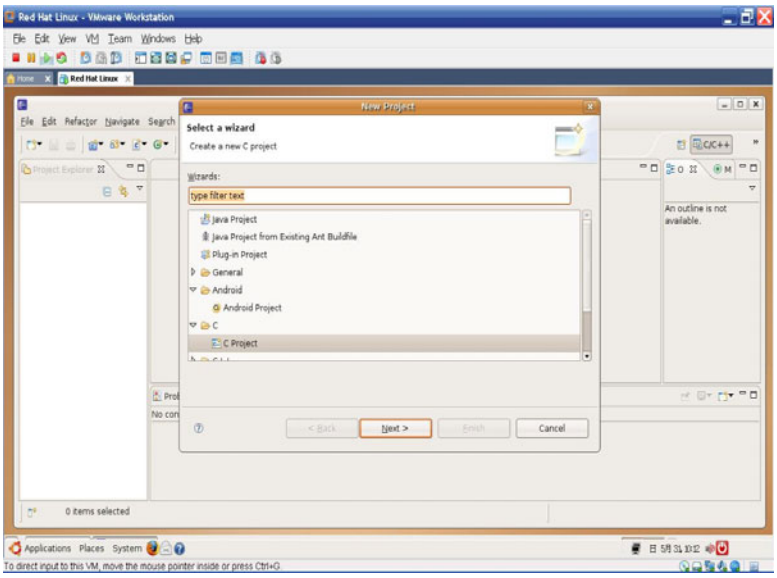


图 1.5

- 设定工程名称，选择工具链，如图 1.6

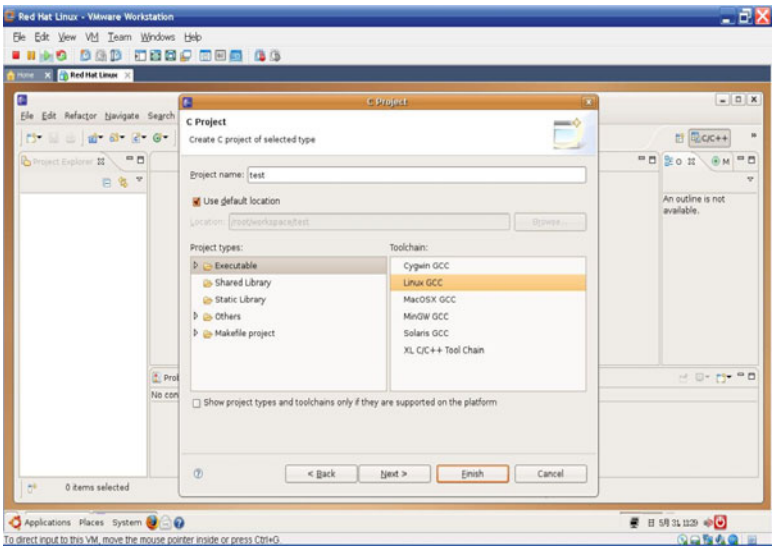


图 1.6

- 创建了新的工程，如图 1.7

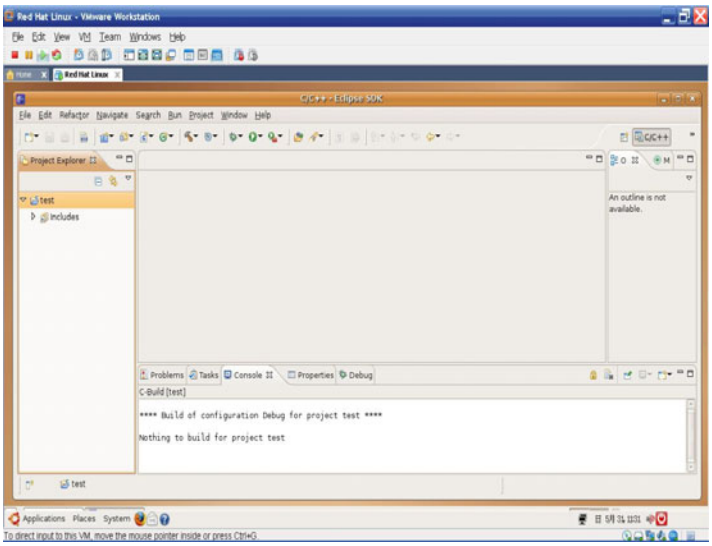


图 1.7

- 添加源文件，如图 1.8、图 1.9

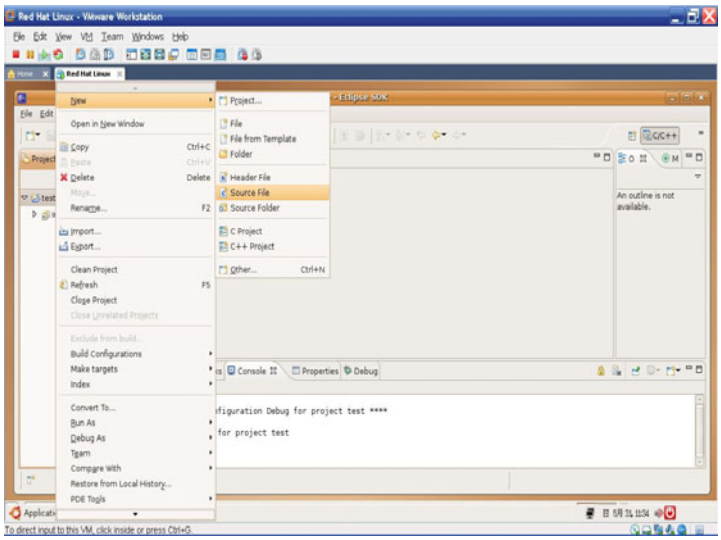


图 1.8

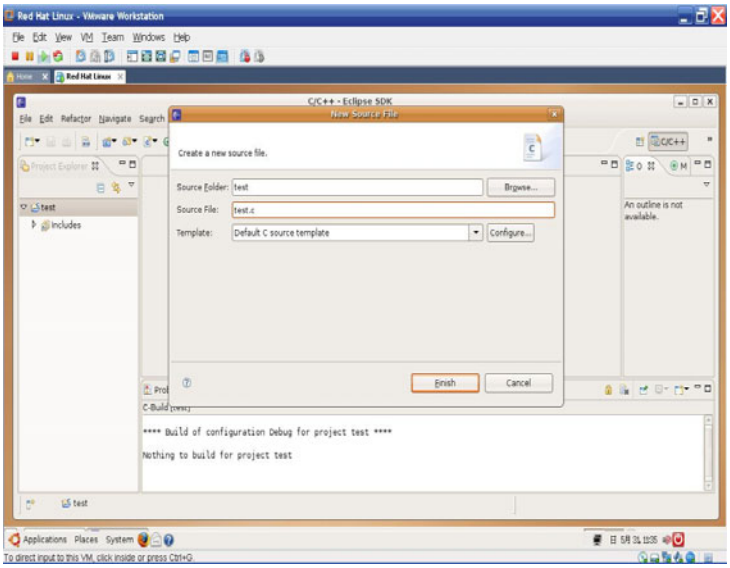


图 1.9

- 编辑代码，如图 1.10

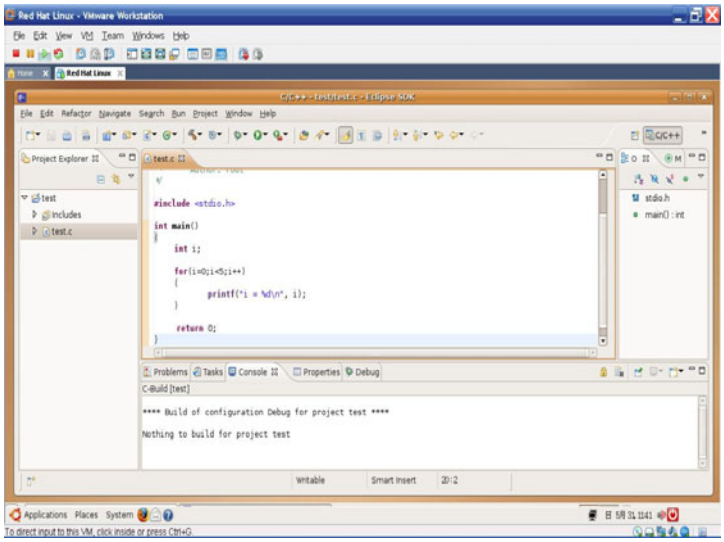


图 1.10

- 编译工程，如图 1.11

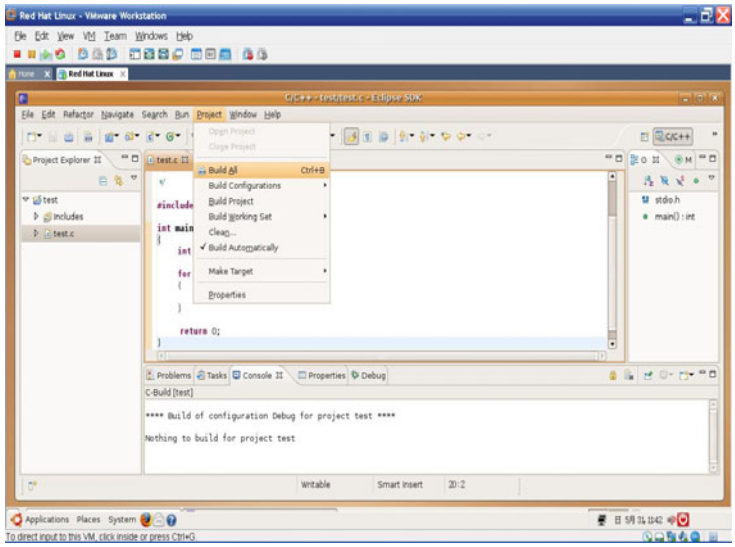


图 1.11

- 运行程序，如图 1.12

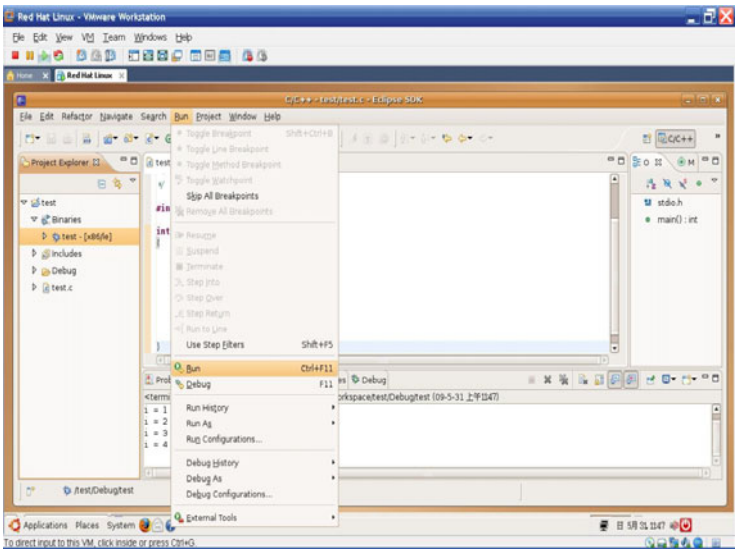


图 1.12

- 右键单击编辑框最左边浅黄色区域，设置断点，如图 1.13

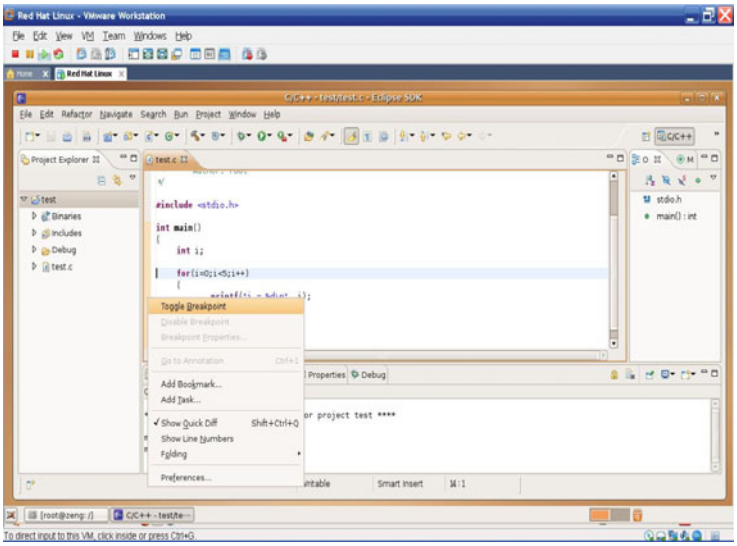


图 1.13

- 调试程序，如图 1.14、图 1.15

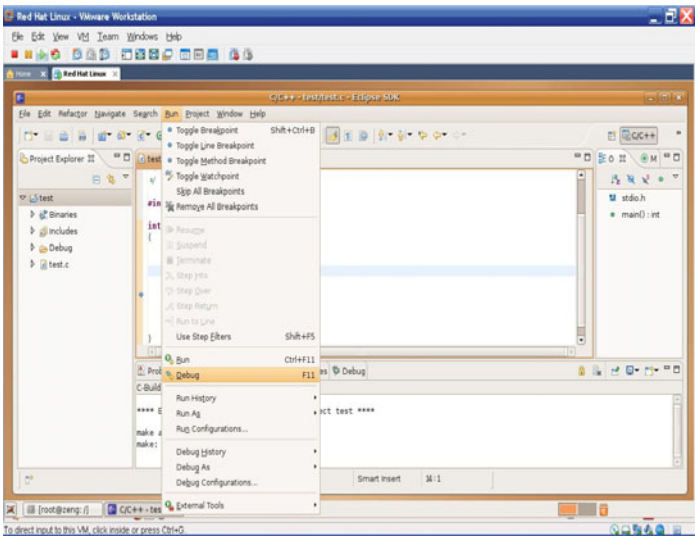


图 1.14

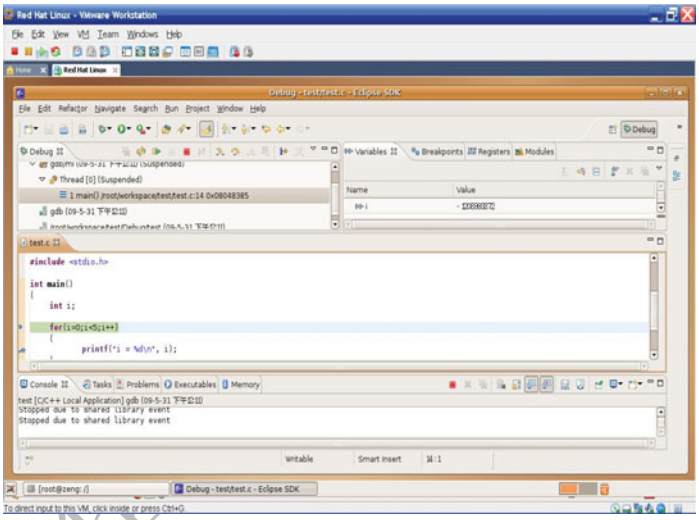


图 1.15

以上介绍的是如何使用 eclipse 进行本机上的开发。对于嵌入式系统来说，更多的时候要进行交叉编译和调试。下面将要图示如何在 eclipse 里为 arm 平台开发应用程序。

- 修改工程属性中的编译器、链接器和汇编器，如图 1.16、图 1.17、图 1.18

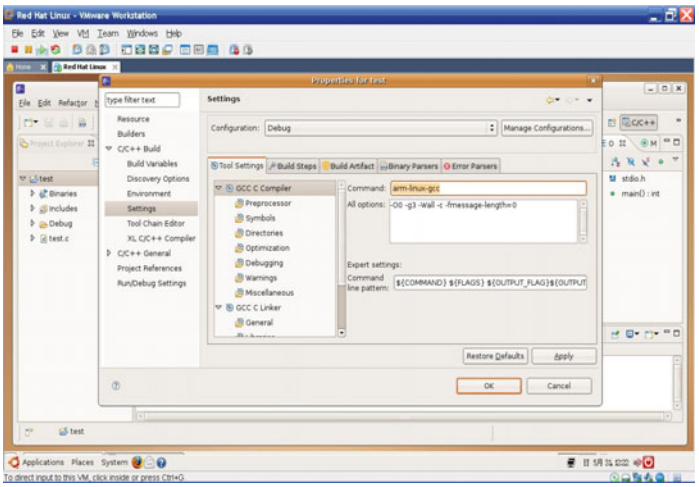


图 1.16

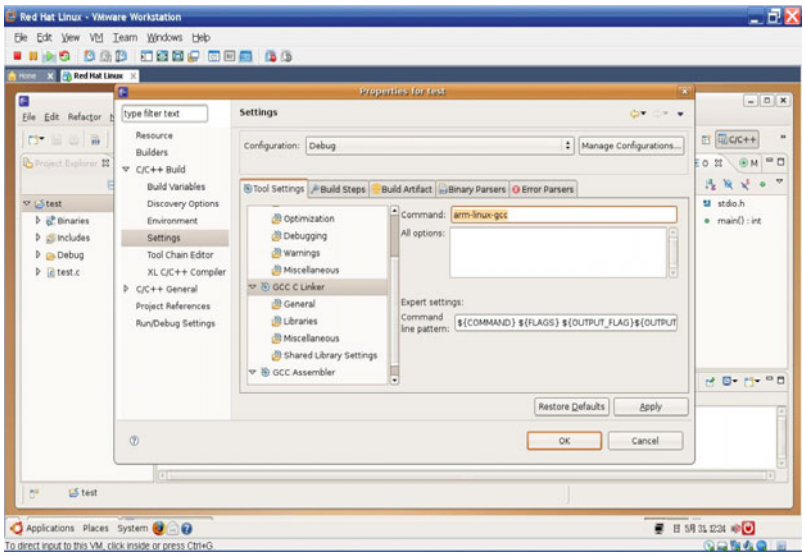


图 1.17

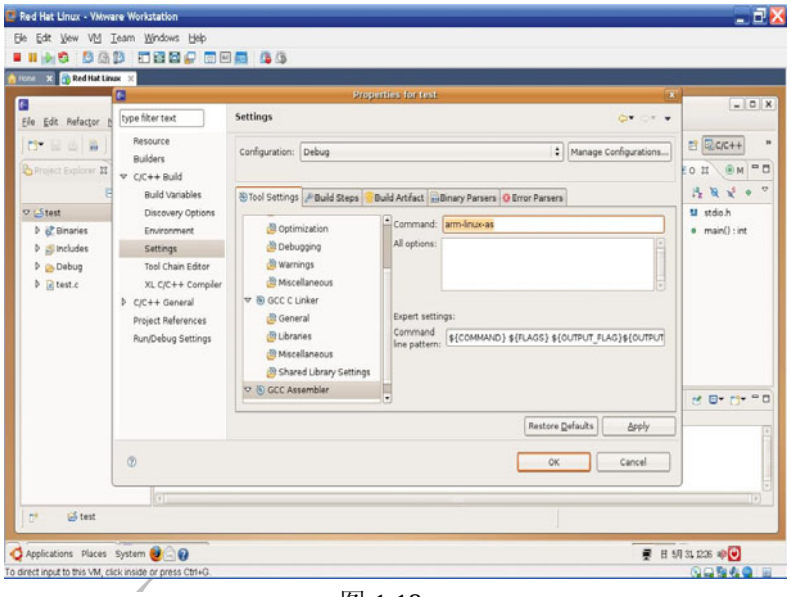


图 1.18

➢ 设置完成后重新编译，如图 1.19

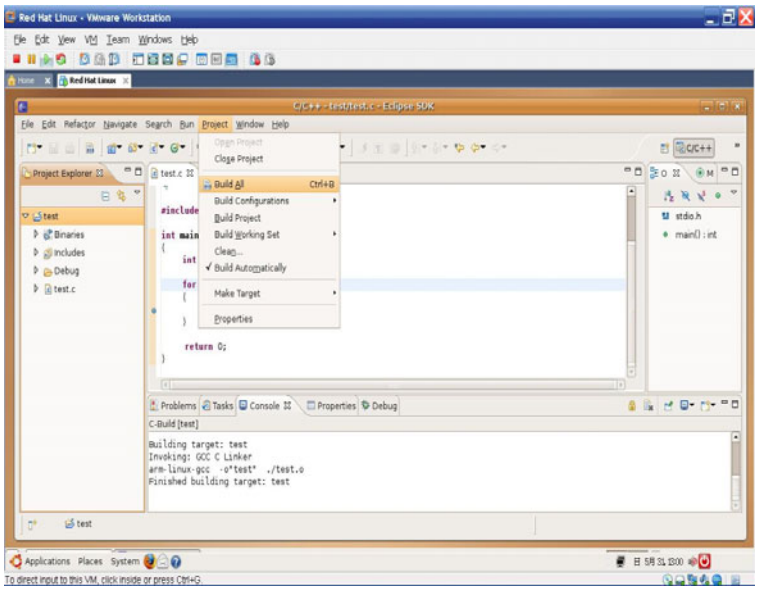


图 1.19

- 在目标平台上运行起编译好的程序，等待主机端调试器的连接，如图 1.20

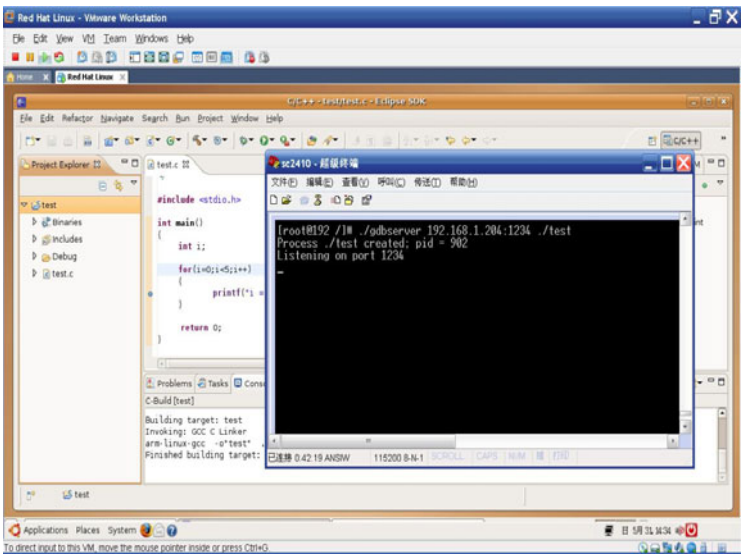


图 1.20

- 主机端 eclipse 里设置调试选项，如图 1.21

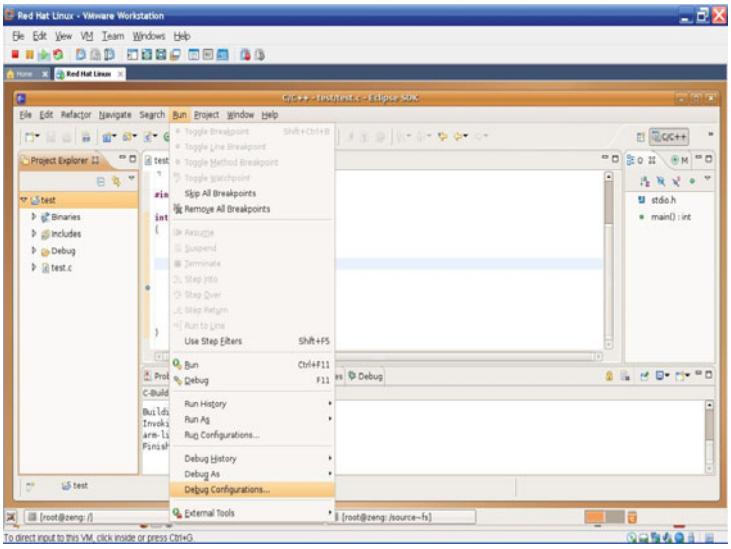


图 1.21

- 指定调试器为 arm-linux-gdb，如图 1.22

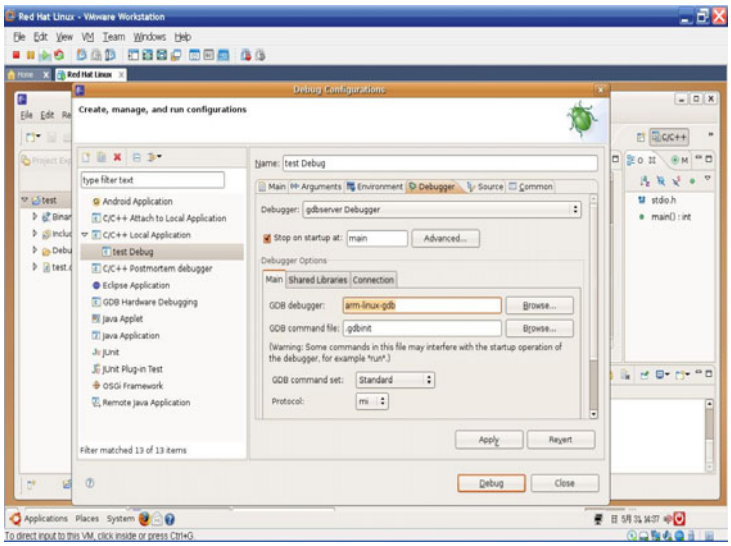


图 1.22

- 设定通过网络进行交叉调试，如图 1.23

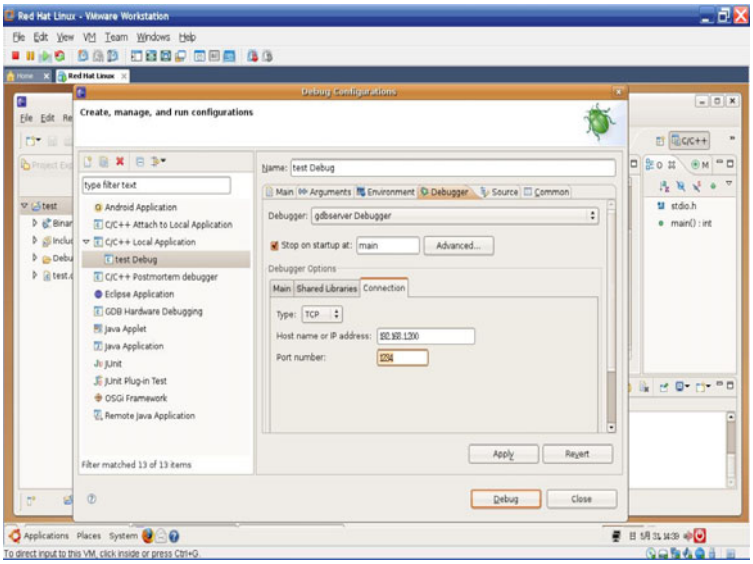


图 1.23

- 开始交叉调试，如图 1.24、图 1.25

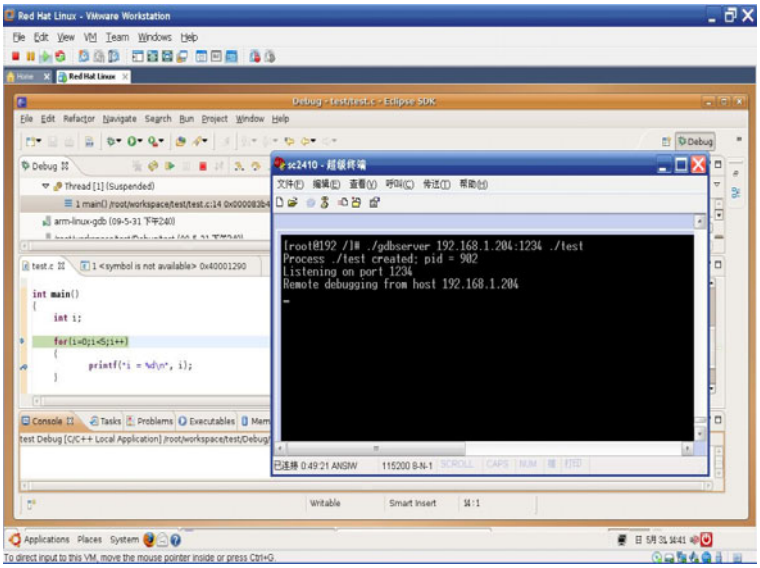


图 1.24

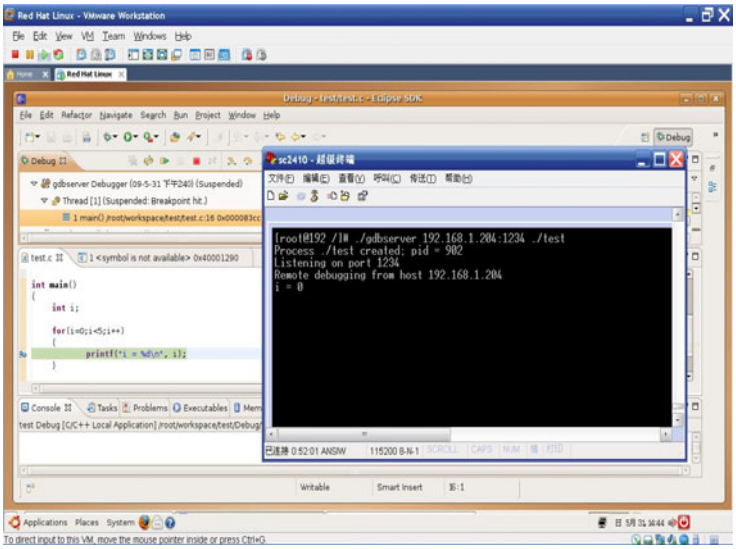


图 1.25

本 章 小 结

熟练使用开发工具是进行嵌入式 Linux C 语言开发的第一步。本章详细介绍了嵌入式 Linux C 语言开发常见的编辑器 vi、编译器 GCC、调试器 GDB、工程管理器 make 和集成开发环境 eclipse。
对于这些工具的使用方法，读者一定要通过实际动手操作来熟练掌握。本章在每个工具的讲解中都提供了完整的实例供读者参考。

动 手 练 练

1．在 vi 中编辑如下代码（命名为 test.c），并自行编写 Makefile 运行该程序。

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

int x = 0;
int y = 5;

int fun1()
{
        extern p, q;
        printf("p is %d, q is %d\n", p, q);
        return 0;
}
int p = 8;
int q = 10;

int main()
{
        fun1();
        printf("x is %d, y is %d\n", x, y);
}
```

2．在 eclipse 中创建工程，添加同样的代码，使用 GCC 编译，GDB 调试该程序。

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路 250 号银海大厦 11 层 B 区，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-25590506

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218

广州地址：广州市天河区中山大道 268 号天河广场 3 层，电话：020-28916067