第5章 chapter 5

基于ARM的嵌入式软件开发

学习嵌入式程序开发的起点是最简单的程序。一个基本的Linux应用程序可以涵盖编程的所有基础知识，通过编写Linux应用程序，可以帮助读者快速入门程序开发。

本章主要侧重于嵌入式C语言程序设计的基础知识、设计技巧，以及C语言与汇编混合编程的内容，旨在引导读者进入嵌入式程序开发的领域。

5.1 嵌入式C语言程序设计基础

很多编程书籍都以输出“Hello，World!”向初学者展示如何编写程序。虽然这个程序很简单，但它却展示了C程序的基本要素：语法格式、引用头文件、调用库函数等。本节将展示程序的编辑、编译和执行的相关知识。

5.1.1 Hello World

1. 用VIM编辑源代码文件hello.c

在Linux终端中输入vim hello.c。

2. 编写源代码

在屏幕的左下角会出现文件名为hello.c的标识，表示一个新建的文件，名称为hello.c。接下来，在键盘上输入小写字母i，屏幕的左下角将显示“插入”，表示目前进入了插入模式，此时可以输入源代码。根据示例，输入相应的源代码。

```
#include <stdio.h>

int main(void)
{
    printf("Hello,World\r\n");
    return 0;
}
```

3. 保存退出

输入实例所示的源代码后，在当前状态下按Esc键，输入“:wq”，按Enter键，将保存

文件并退出 VIM。

4. 用 GCC 编译程序

编辑好源文件 hello.c 文件后，需要把它编译成可执行文件才可以在 Linux 主机上运行。在控制终端当前目录下，输入以下命令完成编译。

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

GCC 编译器会将源代码文件编译连接成 Linux 可以执行的二进制文件。其中，-o 表示输出的二进制文件名字为 hello。

此时，可以在终端输入“./hello”命令来运行编译好的程序。

```
root@altaslesson:~/c#vim hello.c
root@altaslesson:~/c#gcc -o hello hello.c
root@altaslesson:~/c#./hello
Hello,Worid!
root@altaslesson:~/c#
```

5.1.2 GCC 与交叉编译器

在 5.1.2 节中已经使用 GCC 编译器编译.c 文件，接下来介绍编译器的使用方法。

1. gcc 命令

gcc 命令的格式如下：

```
gcc [选项] [文件名字]
```

主要选项如下。

- -c：只编译，不链接为可执行文件，编译器将输入的.c 文件编译为.o 的目标文件。
- -o<输出文件名>：用来指定编译结束以后的输出文件名，如果不使用这个选项，则 GCC 默认编译出来的可执行文件名字为 a.out。
- -g：添加调试信息，如果使用调试工具(如 GDB)，就必须加入此选项，此选项指示编译的时候生成调试所需的符号信息。
- -O：对程序进行优化编译，如果使用此选项，那么整个源代码在编译、链接的时候都会进行优化，这样产生的可执行文件的执行效率更高。
- -O2：比-O 有更大幅度的优化，生成的可执行文件的执行效率更高，但是整个编译过程会很长。

2. 编译流程

GCC 编译器的工作流程一般包括预处理、汇编、编译和链接阶段。预处理阶段主要对程序中的宏定义等相关内容进行初步处理。汇编阶段将 C 文件转换为汇编文件。编译阶段将 C 源文件编译为以.o 结尾的目标文件。生成的目标文件无法直接执行，需要进行链接操作。如果项目中包含多个 C 源文件，则会生成多个目标文件，这些目标文件需

要链接在一起，组成完整的可执行文件。

在上一部分的示例程序中，只包含一个简单的文件，因此可以直接使用 gcc 命令生成可执行文件。

3. 交叉编译器安装

在进行 ARM 裸机、Uboot 移植以及 Linux 移植等任务时，需要在 Linux 环境下进行编译操作。编译过程需要使用编译器。之前已经介绍了如何在 Linux 环境下进行 C 语言开发，并使用了 GCC 编译器编译代码。然而，Linux 系统自带的 GCC 编译器主要用于 X86 架构，因此，使用该编译器生成的程序无法在 ARM 架构上运行。为了在 X86 架构的操作系统上进行 ARM 架构目标主机的编译工作，需要借助交叉编译工具链。交叉编译工具链包含交叉编译器 GCC。在交叉编译器中，“交叉”一词表示在一个架构上编译另一个架构的代码，实际上是将两种架构“交叉”起来。通过使用交叉编译工具链，可以在 Linux 环境下为 ARM 架构生成可执行程序，从而在 ARM 主机上运行。交叉编译工具链中的交叉编译器 GCC 就提供了这样的能力。

交叉编译器有很多种，这里以 Linaro 出品的交叉编译器为例进行介绍。Linaro 是一家非营利性质的开放源代码软件工程公司，开发了很多软件，最著名的就是 Linaro GCC 编译工具链（编译器），关于 Linaro 的详细介绍可以到 Linaro 官网查阅。Linaro GCC 编译器的下载地址如下：http://releases.linaro.org/components/toolchain/binaries/5.4-2017.05/aarch64-linux-gnu/gcc-linaro-5.4.1-2017.05-x86_64_aarch64-linux-gnu.tar.xz。在交叉编译器下载完成后进行交叉编译工具链的安装，以下是安装步骤。

（1）登录 Linux 服务器。

（2）执行如下命令，切换至 root 用户。

```
su -root
```

（3）执行如下命令，创建/opt/compiler 目录。

```
mkdir /opt/compiler
```

（4）使用文件传输工具将交叉编译工具链上传至/opt/compiler 目录。

（5）进入/opt/compiler 目录。

```
cd /opt/compiler
```

（6）执行如下命令，解压缩交叉编译工具。

```
tar -xvf 交叉编译工具链 -C ./ --strip-components 1
```

（7）在配置文件中增加交叉编译工具链的路径。

```
echo "export PATH=\$PATH:/opt/compiler/bin" >>/etc/profile
```

（8）执行如下命令，使环境变量生效。

```
source /etc/profile
```

(9) 执行如下命令,查看交叉编译工具链的版本。

```
aarch64-linux-gnu-gcc -v
```

(10) 如果显示版本信息,则表明工具链安装成功。

5.1.3 Makefile

在前一部分中,介绍了如何在 Linux 环境下使用 GCC 编译器进行 C 语言编译,通过在终端中执行 gcc 命令可以完成单个或少量.c 文件的编译。然而,在工程规模较大的情况下,例如存在数十、数百甚至数千个源代码文件时,在终端中逐个输入 gcc 命令显然是不切实际的。

为了解决这个问题,可以编写一个描述编译源代码文件以及编译规则的文件,使用该文件可以指定编译的源代码文件与编译方式。这样,每次需要编译整个工程时,只需要执行该文件即可。这个文件的解决方案就是 Makefile。

Makefile 文件的作用是描述需要编译的文件和重新编译的条件。类似于脚本文件,Makefile 中可以执行系统命令。使用 Makefile,只需要执行 make 命令,整个工程就会自动编译,大幅提高了软件开发的效率。

通过使用 Makefile,程序员可以方便地管理包含大量源代码文件的工程,并定义编译规则,使得编译过程自动化。这样,无论工程规模大小,都能更加高效地进行软件开发。接下来以一个例子介绍 make 工具和 Makefile 语法。项目需要完成以下任务:通过键盘输入两个整型数字,然后计算它们的和,并将结果显示在屏幕上,在这个工程中有 main.c、input.c 和 calcu.c 三个.c 文件,以及 input.h、calcu.h 两个头文件。其中 main.c 是主体,input.c 负责接收从键盘输入的数值,calcu.h 进行任意两个数相加的操作,其中 main.c 文件的内容如下:

```
#include <stdio.h>
#include "input.h"
#include "calcu.h"

int main(void)
{
    int a,b,num;
    input_int(&a, &b);
    num =calcu(a, b);
    printf("%d +%d =%d\r\n", a, b, num);
}
```

input.c 文件的内容如下:

```
#include <stdio.h>
#include "input.h"
void input_int(int *a, int *b)
{
    printf("input two num:");
```

```
    scanf("%d %d", a, b);
    printf("\r\n");
}
```

calcu.c 文件的内容如下：

```
#include "calcu.h"
int calcu(int a, int b)
{
    return (a +b);
}
```

input.h 文件的内容如下：

```
#ifndef _INPUT_H
#define _INPUT_H
void input_int(int * a, int * b);
#endif
```

calcu.h 文件的内容如下：

```
#ifndef _CALCU_H
#define _CALCU_H
int calcu(int a, int b);
#endif
```

以上是这个工程的所有源文件，接下来使用前面介绍的方法进行编译，在终端输入如下命令：

```
gcc main.c calcu.c input.c -o main
```

上面命令的意思就是使用 GCC 编译器对 main.c、calcu.c 和 input.c 这三个文件进行编译，编译生成的可执行文件叫作 main。编译完成以后执行 main 这个程序，测试软件是否工作正常。

```
root@altaslesson:~/c#
root@altaslesson:~/c#gcc main.c calcu.c input.c -o main
root@altaslesson:~/c#
root@altaslesson:~/c#./main
input two num:5 6

5+6=11
root@altaslesson:~/c#
```

然而，在工程规模庞大的情况下，例如当拥有数千个源文件时，若仅使用之前提到的命令编译方式，那么一旦任何一个文件发生修改，所有文件都将被重新编译。如果工程中拥有数万个源文件（例如 Linux 源码），那么每次重新编译数万个文件将耗费大量时间。为了更高效地处理这种情况，最理想的方式是仅编译已被修改的文件，而无须重新编译未被修改的文件。为此，需要改变编译的方法。在首次编译工程时，先编译所有源文件。之后，在某个文件发生修改时，可以仅对这个修改的文件进行编译，而无须重新编译其他未更改的文件。具体的命令如下：

```
gcc -c main.c
gcc -c input.c
gcc -c calcu.c
gcc main.o input.o calcu.o -o main
```

上述命令的前三行分别是将 main.c、input.c 和 calcu.c 编译成对应的.o 文件，所以使用了-c 选项，只进行编译而不链接。最后一行命令是将编译出来的所有.o 文件链接成可执行文件 main。假如现在修改了 calcu.c 这个文件，只需要将 caclu.c 这个文件重新编译成.o 文件，然后将所有的.o 文件链接成可执行文件即可：

```
gcc -c calcu.c
gcc main.o input.o calcu.o -o main
```

但是这样又会产生一个问题，如果修改了大量的文件，可能都不记得哪个文件修改过了，然后忘记编译，为此需要这样一个工具：

- 如果工程没有编译过，那么工程中的所有.c 文件都要被编译并链接成可执行程序；
- 如果工程中只有个别.c 文件被修改了，那么只编译这些被修改的.c 文件即可；
- 如果工程的头文件被修改了，那么需要编译所有引用这个头文件的.c 文件并链接成可执行文件。

很明显，能够完成这个功能的就是 Makefile，需要在工程目录下创建名为 Makefile 的文件。

在 Makefile 中输入如下代码：

```
main: main.o input.o calcu.o
	gcc -o main main.o input.o calcu.o
main.o: main.c
	gcc -c main.c
input.o: input.c
	gcc -c input.c
calcu.o: calcu.c
	gcc -c calcu.c
clean:
	rm * .o
	rm main
```

上述代码中，所有行首需要空出来的地方一定要使用 Tab 键实现。

Makefile 编写好后，可以使用 make 命令编译工程，直接在命令行中输入 make 即可，make 命令会在当前目录下查找是否存在 Makefile 这个文件，如果存在，就会按照 Makefile 中定义的编译方式进行编译。

使用 make 命令编译完成后会在当前工程目录下生成各种.o 文件和可执行文件，说明编译成功，接下来就可以运行程序了。

```
root@altaslesson:~/c#ls
calcu.c calcu.h input.c input.h main .c Makefile
root@altaslesson:~/c#
```

```
root@altaslesson:~/c#make
gcc -c main .c
gcc-c input.c
gcc-c calcu.c
gcc -o main main.o input.o calcu.o
root@altaslesson:~/c#./main
input two num:5 6

5+6=11
root@altaslesson:~/c#ls
calcu.c calcu.h calcu.o input.c input.h input.o main main.c main.o Makefile
root@altaslesson:~/c#
```

由于生成了大量.o 文件，故可以执行 make clean 命令将可执行程序和.o 文件一起删除。

```
root@altaslesson:~/c#ls
calcu.c calcu.h calcu.o input.c input.h input.o main main.c main.o Makefile
root@altaslesson:~/c#
root@altaslesson:~/c#make clean
rm *.o
rm main
root@altaslesson:~/c#ls
calcu.c calcu.h input.c input.h main.c Makefile
root@altaslesson:~/c#
```

5.1.4 CMake

不同的 IDE 集成的 make 工具所遵循的规范和标准都不同，导致其语法、格式不同，也就不能很好地跨平台编译，会再次使得工作烦琐起来。

cmake 为了解决这个问题而诞生了，其允许开发者指定整个工程的编译流程，然后根据编译平台生成本地化的 Makefile 和工程文件，最后只需 make 编译即可。

简而言之，可以把 cmake 看成一款自动生成 Makefile 的工具，所以编译流程就变成了 cmake→make→用户代码→可执行文件。

1. 编写 CMakeLists.txt

首先编写 C 语言程序，使用前面使用过的 main.c 函数，然后编写 CMakeLists.txt 文件，并保存在与 main.c 源文件同一个目录下。

```
#CMake 最低版本要求
cmake_minimum_required (VERSION 2.8)

#项目信息
project (Demo)

#指定生成目标
add_executable(Demo main.c)
```

CMakeLists.txt 的语法比较简单，由命令、注释和空格组成，其中命令是不区分大小写的。符号“#”后面的内容是注释。命令由命令名称、小括号和参数组成，参数之间使用空格分隔。对于上面的 CMakeLists.txt 文件，依次出现了以下几个命令。

（1）cmake_minimum_required：指定运行此配置文件所需的 CMake 的最低版本。

（2）project：参数值是 Demo，该命令表示项目的名称是 Demo。

（3）add_executable：将名为 main.c 的源文件编译成一个名称为 Demo 的可执行文件。在本例中传入了两个参数，第一个参数表示生成的可执行文件对应的文件名，第二个参数表示对应的源文件。

2. 编译项目

在当前目录执行 cmake，得到 Makefile 后再使用 make 命令编译得到 Demo 可执行文件。

```
root@altaslesson:~/c/cmake# cmake .
--The C compiler identification is GNU 5.4.0
--The CXX compiler identification is GNU 5.4.0
--Check for working C compiler: /usr/bin/cc
--Check for working C compiler: /usr/bin/cc --works
--Detecting C compiler ABI info
--Detecting C compiler ABI info -done
--Detecting C compile features
--Detecting C compile features -done
--Check for working CXX compiler: /usr/bin/c++
--Check for working CXX compiler: /usr/bin/c++--works
--Detecting CXX compiler ABI info
--Detecting CXX compiler ABI info -done
--Detecting CXX compile features
--Detecting CXX compile features -done
--Configuring done
--Generating done
--Build files have been written to: /root/c/cmake
root@altaslesson:~/c/cmake# make
Scanning dependencies of target Demo
[ 50%] Building C object CMakeFiles/Demo.dir/main.c.o
[100%] Linking C executable Demo
[100%] Built target Demo
root@altaslesson:~/c/cmake# ./Demo
Hello,World
```

3. 运行程序

执行./Demo 可执行文件即可。

4. 多个源文件

若存在多个源文件，且都在同一个目录下，则可以修改 CMakeLists.txt 为如下：

```
#查找当前目录下的所有源文件
#并将名称保存到 DIR_SRCS 变量
aux_source_directory(. DIR_SRCS)

#指定生成目标
add_executable(Demo ${DIR_SRCS})
```

5. 生成并添加库文件

在平时的开发过程中，也有很多场景需要将源码编译成库文件以供使用，这个需求也可以使用 cmake 做到，这就需要用到以下命令：

```
add_library(libhello 静态/动态库 hello.c)
```

若没有设置参数，则默认生成静态库文件，可以通过增加参数的方式设置指定的库文件。

```
add_library(MathFunctions SHARED hello.c)          #生成动态库文件
add_library(MathFunctions STATIC hello.c)          #生成静态库文件
```

在生成之后，用如下命令设置目标文件需要链接的库。

```
target_link_libraries (Demo MathFunctions)
```

6. 指定头文件搜索路径

```
include_directories(
../include/
/usr/local/include/
…
)
```

指定编译时，头文件的路径先搜索“../include”和“/usr/local/include”，然后到系统的默认路径搜索。

7. 指定库文件搜索路径

```
link_directories(
    ${LIB_PATH}
    $ENV{HOME}/ascend_ddk/${ARCH}/lib/
    ${INC_PATH}/atc/lib64
)
```

在编译时会先到上述目录搜索库文件，然后到系统的默认路径搜索库文件。

5.2 嵌入式 C 语言程序设计技巧

本节将介绍嵌入式 C 语言程序设计的技巧，包括 C 编译器及其优化算法，以及面向对象编程和模块化编程的思想。

5.2.1 C编译器及其优化方法

本节将帮助读者在ARM处理器上编写高效的C代码。本节涉及的一些技术不仅适用于ARM处理器，也适用于其他RISC处理器。本节首先从ARM编译器及其优化方法入手，讲解C编译器在优化代码时碰到的一些问题。理解这些问题将有助于编写出在提高执行速度和减少代码尺寸方面更高效的C源代码。

本节假定读者熟悉C语言，并且有一些汇编语言编程方面的知识。

1. 为编译器选择处理器结构

在编译C源文件时，必须为编译器指定正确的处理器类型，这样可以使编译的代码最大限度地利用处理器的硬件结构，如对半字加载(halfword load)、存储指令(store instructions)和指令调度(instruction scheduling)的支持。所以编译程序时，应该尽量准确地告诉编译器该代码运行在什么类型的处理器上。有些类型的编译器不能直接支持，如SA-1100，这时可以使用与该类型处理器为同一指令集的基本处理器，例如对于SA-100，可以使用StrongARM。

指定目标处理器可能使代码与其他ARM处理器不兼容。例如，编译时指定了ARMv6体系结构的代码，可能不能运行在ARM920T的处理器上(当代码中使用了ARMv6体系结构中特有的指令)。

选择处理器类型可以使用--cpu name编译选项，该选项可以生成用于特定ARM处理器或体系结构的代码。

- 输入名称必须和ARM数据表中所示严格一致，例如ARM7TDMI。该选项不接受通配符字符。有效值是任何ARM6或更高版本的ARM处理器。
- 选择处理器操作会选择适当的体系结构、浮点单元(FPU)以及存储结构。
- 某些--cpu选项暗含--fpu选项。
- ARM1136JF-S选项暗含--fpu vfpv2选项。隐式FPU只覆盖命令行上出现在--cpu选项前面的显式--fpu选项。如果没有指定--fpu选项和--cpu选项，则使用--fpu softvfp选项。

2. 调试选项

如果在编译C源程序时设置了调试选项，则将很大程度地影响最终代码的大小和执行效率。因为为了能够在调试程序时正确地显示变量或设置断点，带调试信息的代码映像会包含很多冗余的代码和数据，所以如果想最大限度地提高程序执行效率、减少代码尺寸，就要在编译源文件时去除编译器的调试选项。

以下选项指定调试表的生成方法。

- -g(--debug)：该选项启用生成当前编译的调试表。无论是否使用-g选项，编译器生成的代码都是相同的。唯一的区别是调试表是否存在。编译器是否对代码进行优化是由-O选项指定的。默认情况下，使用-g选项等价于使用-g -dwarf2 --debug_macros。注意，编译程序时，只使用-g选项而没有使用优化选项，编译器