

第7章 嵌入式Linux系统移植及调试



目 录

- 7.1 Boot Loader基本概念与典型结构
- 7.2 U-Boot
- 7.3 交叉开发环境的建立
- 7.4 交叉编译工具链
- 7.5 嵌入式Linux系统移植过程
- 7.6 Gdb调试器
- 7.7 远程调试
- 7.8 内核调试

一个嵌入式linux系统通常由引导程序及参数、linux内核、文件系统和用户应用程序组成。

由于嵌入式系统与开发主机运行的环境不同，这就为开发嵌入式系统提出了开发环境特殊化的要求。交叉开发环境正是在这种背景下应运而生。

7.1

Part One

7.1 Boot Loader基本概念与典型结构

PC 机中的引导加载程序

- 流程
 - 系统加电启动时，系统BIOS负责检测并启动加载控制卡上的BIOS
 - BIOS在完成硬件检测和资源分配后，将硬盘 MBR 中的 Boot Loader 读到系统的 RAM 中，然后将控制权交给 OS Boot Loader
 - Boot Loader 的主要运行任务就是将内核映象从硬盘上读到 RAM 中，然后跳转到内核的入口点去运行，即开始启动操作系统。
- BIOS常驻系统内存的高端(C0000-FFFFFF)
- BIOS的扩充方法
 - 1.直接修改，然后固化
 - 2.软件方法，TSR技术
- BIOS目前逐步扩展到用固件实现的UEFI，“统一的可扩展固件接口”(Unified Extensible Firmware Interface)。

嵌入式系统中引导加载程序

- 没 **BIOS** 那样的固件程序
 - 有的嵌入式 **CPU** 也会内嵌一段短小的启动程序
- 系统的加载启动任务就完全由 **Boot Loader** 来完成
 - 如**ARM7TDMI**中，系统在上电或复位时从地址 **0x00000000** 处开始执行
 - 这个地址是**Boot Loader** 程序

7.1.1 Boot Loader基本概念

| | | | | |
|------------|----|----------|------|--------|
| Bootloader | 参数 | Linux 内核 | 文件系统 | 用户应用程序 |
|------------|----|----------|------|--------|

一个嵌入式Linux系统通常可以分为以下几个部分：

(1) 引导加载程序及其环境参数。这里通常是指BootLoader以及相关环境参数。

(2) Linux内核。基于特定嵌入式开发板的定制内核以及内核的相关启动参数。

(3) 文件系统。主要包括根文件系统和一般建立于Flash内存设备之上文件系统。

(4) 用户应用程序。基于用户的应用程序。有时在用户应用程序和内核层之间可能还会包括一个嵌入式图形用户界面程序（GUI）。常见的嵌入式GUI有QT和MiniGUI等。

- 在嵌入式操作系统中，**BootLoader**是在操作系统内核运行之前运行的一小段程序，可以初始化硬件设备、建立内存空间映射图，从而将系统的软硬件环境带到一个适合的状态，以便为最终调用操作系统内核准备好正确的环境。
- 在嵌入式系统中，通常并没有像通用计算机中BIOS那样的固件程序，因此整个系统的加载启动任务就完全由BootLoader来完成。

BootLoader是嵌入式系统在加电后执行的第一段代码，在它完成CPU和相关硬件的初始化之后，再将操作系统映像或固化的嵌入式应用程序装载到内存中然后跳转到操作系统所在的空间，启动操作系统运行。

对于嵌入式系统而言，**BootLoader**是基于特定硬件平台来实现的。因此，几乎不可能为所有的嵌入式系统建立一个通用的**BootLoader**，不同的处理器架构都有不同的**BootLoader**。

BootLoader不仅依赖于**CPU**的体系结构，而且依赖于嵌入式系统板级设备的相关配置。

7.1.2 BootLoader的操作模式

大多数BootLoader都包含两种不同的操作模式：自启动模式和交互模式。这种划分仅仅对于开发人员才有意义。

(1) 自启动模式

自启动模式也叫启动加载模式。在这种模式下，**BootLoader**自动从目标机上的某个固态存储设备上将操作系统加载到**RAM**中运行，整个过程并没有用户的介入。这种模式是**BootLoader**的正常工作模式

2) 交互模式

交互模式也叫下载模式。在这种模式下，目标机上的 **BootLoader** 将通过串口或网络等通信手段从开发主机上下载内核映像、根文件系统到 **RAM** 中。然后再被 **BootLoader** 写到目标机上的固态存储媒质（如 **FLASH**）中，或者直接进入系统的引导。交互模式也可以通过接口（如串口）接收用户的命令。这种模式在初次固化内核、根文件系统时或者更新内核及根文件系统时都会用到。

7.1.3 BootLoader的典型结构

BootLoader启动大多数都分为两个阶段。第一阶段主要包含依赖于**CPU**的体系结构硬件初始化的代码，通常都用汇编语言来实现。这个阶段的任务有：

基本的硬件设备初始化（屏蔽所有的中断、关闭处理器内部指令/数据Cache等）。

为第二阶段准备RAM空间。

如果是从某个固态存储媒质中，则复制BootLoader的第二阶段代码到RAM。

设置堆栈。

跳转到第二阶段的C程序入口点。

第二阶段通常用C语言完成，以便实现更复杂的功能，也使程序有更好的可读性和可移植性。这个阶段的任务有：

初始化本阶段要使用到的硬件设备。

检测系统内存映射。

将内核映像和根文件系统映像从Flash读到RAM。

为内核设置启动参数。

调用内核。

7.1.4常见的BootLoader

表 7-1 常见开源 BootLoader

| BootLoa der | 描述 | X86 | ARM | POWER PC |
|----------------|-----------------|-----|-----|----------|
| LILO | Linux 磁盘引导程序 | 是 | 否 | 否 |
| GRUB | GNU 的 LILO 替代程序 | 是 | 否 | 否 |
| BLOB | LART 等硬件平台的引导程序 | 否 | 是 | 否 |
| U-Boot | 通用引导程序 | 是 | 是 | 是 |
| Redboot | 基于 ecos 的引导程序 | 是 | 是 | 是 |

(1) Redboot

Redboot (Red Hat Embedded Debug and Bootstrap)是Red Hat公司开发的一个独立运行在嵌入式系统上的BootLoader程序，是目前比较流行的一个功能、可移植性好的BootLoader。

Redboot是一个采用eCos开发环境开发的应用程序，并采用了eCos的硬件抽象层作为基础，但它完全可以摆脱eCos环境运行，可以用来引导任何其他嵌入式操作系统，如Linux、Windows CE等。

Redboot支持的处理器构架有ARM, MIPS, MN10300, PowerPC, Renesas SHx, v850, x86等，是一个完善的嵌入式系统 BootLoader。

（2）U-Boot

U-Boot（Universal BootLoader）于2002年12月17日发布第一个版本U-Boot-0.2.0。U-Boot自发布以后已更新多次，其支持具有持续性。U-Boot是在GPL下资源代码最完整的一个通用Boot Loader。

（3）Blob

Blob(Boot Loader Object)是由Jan-Derk Bakker 和Erik Mouw发布的，是专门为StrongARM 构架下的LART设计的Boot Loader。Blob的最后版本是blob-2.0.5。

Blob功能比较齐全，代码较少，比较适合做修改移植，用来引导Linux，目前大部分S3C44B0板都用Blob修改移植后加载uClinux。

(4) vivi

vivi是韩国mizi 公司开发的BootLoader, 适用于ARM9处理器, 现在已经停止开发了。它是三星官方板SMDK2410采用的BootLoader。Vivi最主要的特点就是代码小巧, 有利于移植新的处理器。同时vivi的软件架构和配置方法类似Linux风格, 对于有过编译Linux内核经验的用户, vivi更容易上手。

7.2 **Part Two**

U-Boot

7.2.1 U-Boot概述

U-Boot, 全称 Universal Boot Loader, 是遵循GPL条款的开放源码项目。从FADSROM、8xxROM、PPCBOOT逐步发展演化而来。其源码目录、编译形式与Linux内核很相似, 事实上, 不少U-Boot源码就是根据相应的Linux内核源程序进行简化而形成的, 尤其是一些设备的驱动程序。

U-Boot支持多种嵌入式操作系统, 主要有OpenBSD, NetBSD, FreeBSD, 4.4BSD, Linux, SVR4, Esix, Solaris, Irix, SCO, Dell, NCR, VxWorks, LynxOS, pSOS, QNX, RTEMS, ARTOS, android等。同时, U-Boot除了支持PowerPC系列的处理器外, 还能支持MIPS、x86、ARM、NIOS、XScale等诸多常用系列的处理器。

U-Boot的主要特点有：

源码开放，目前有些版本的未开源。

支持多种嵌入式操作系统内核和处理器架构。

可靠性和稳定性均较好。

功能设置高度灵活，适合调试、产品发布等；

设备驱动源码十分丰富，支持绝大多数常见硬件外设；
并将对于与硬件平台相关的代码定义成宏并保留在配置文件中，开发者往往只需要修改这些宏的值就能成功使用这些硬件资源，简化了移植工作。

表 7-2 U-Boot 主要目录

| 目录 | 说明 |
|------------|-------------------------------------------------------------|
| board | 目标机相关文件，主要包含 SDRAM、FLASH 驱动等 |
| common | 独立于处理器体系结构的通用代码，如内存大小探测与故障检测 |
| arch/./cpu | 与处理器相关的文件。如 s5p1cxx 子目录下含串口、网口、LCD 驱动及中断初始化等文件 |
| driver | 通用设备驱动 |
| doc | U-Boot 的说明文档 |
| examples | 可在 U-Boot 下运行的示例程序；如 hello_world.c，timer.c |
| include | U-Boot 头文件；尤其 configs 子目录下与目标机相关的配置头文件是移植过程中经常要修改的文件 |
| lib_XXX | 处理器体系相关的文件，如 lib_ppc, lib_arm 目录分别包含与 PowerPC、ARM 体系结构相关的文件 |
| net | 与网络功能相关的文件目录，如 bootp,nfs,tftp |
| post | 上电自检文件目录 |
| rtc | RTC 驱动程序 |
| tools | 用于创建 U-Boot S-RECORD 和 BIN 镜像文件的工具 |

在U-Boot的这些源文件中，以S5PV210为例几个比较重要的源文件如下所示：

(1) **start.S(arch\arm\cpu\armv7\start.S)**

通常情况下start.S是U-Boot上电后执行的第一个源文件。该汇编文件包括定义了异常向量入口、相关的全局变量、禁用L2缓存、关闭MMU等，之后跳转到lowlevel_init()函数中继续执行。

(2)

lowlevel_init.S(board\samsung\smdkv210\lowlevel_init.S)

该源文件用汇编代码编写，其中只定义了一个函数lowlevel_init()。该函数实现对平台硬件资源的一系列初始化过程，包括关看门狗、初始化系统时钟、内存和串口。

(3) `board.c`(`arch\arm\lib\board.c`)

`Board.c`主要实现了U-Boot第二阶段启动过程，包括初始化环境变量、串口控制台、Flash和打印调试信息等，最后调用`main_loop()`函数。

(4) `smdkv210.h`(`include\configs\Smdkv210.h`)

该文件与具体平台相关，比如这里就是S5PV210平台的配置文件，该源文件采用宏定义了一些与CPU或者外设相关的参数。

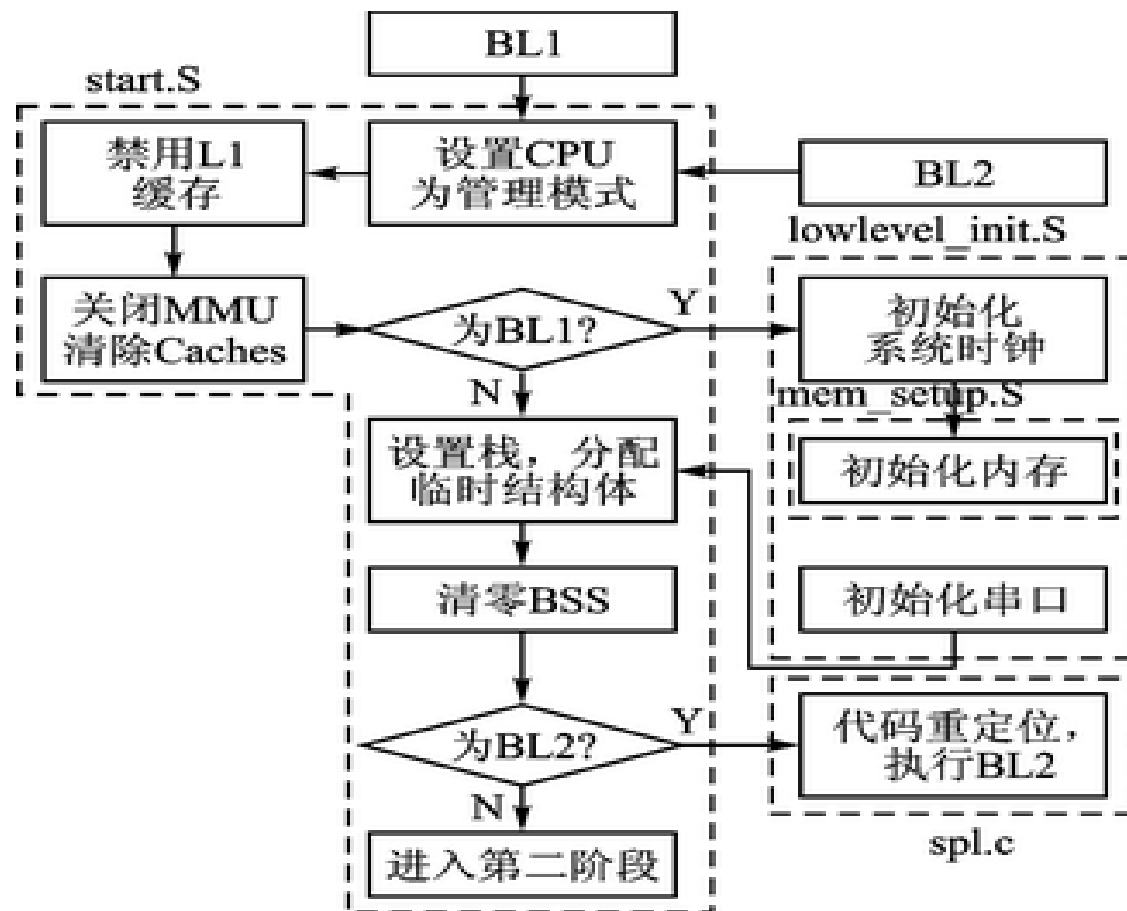
7.2.2 U-Boot启动的一般流程

跟大多数BootLoader的启动过程相似，U-Boot的启动过程分为两个阶段：

第一阶段主要由汇编代码实现，负责对CPU及底层硬件资源的初始化；

第二阶段用C语言实现，负责使能Flash、网卡等重要硬件资源和引导操作系统等。

1. 第一阶段初始化



7-3 U-Boot第一阶段启动流程

与U-Boot第一阶段有关的文件主要有start.s和lowlevel_init.s。上电后,U-Boot首先会设置CPU为管理模式、禁用L1缓存、关闭MMU和清除Caches，之后调用底层初始化函数lowlevel_init()。该函数部分实现如下。

```
.globl lowlevel_init
lowlevel_init:
push{lr}
#if defined(CONFIG_SPL_BUILD)
/*初始化时钟 */
blsystem_clock_init
/*初始化内存 */
blmem_ctrl_asm_init
/*初始化串口 */
bluart_asm_init
#endif
pop{pc}
```

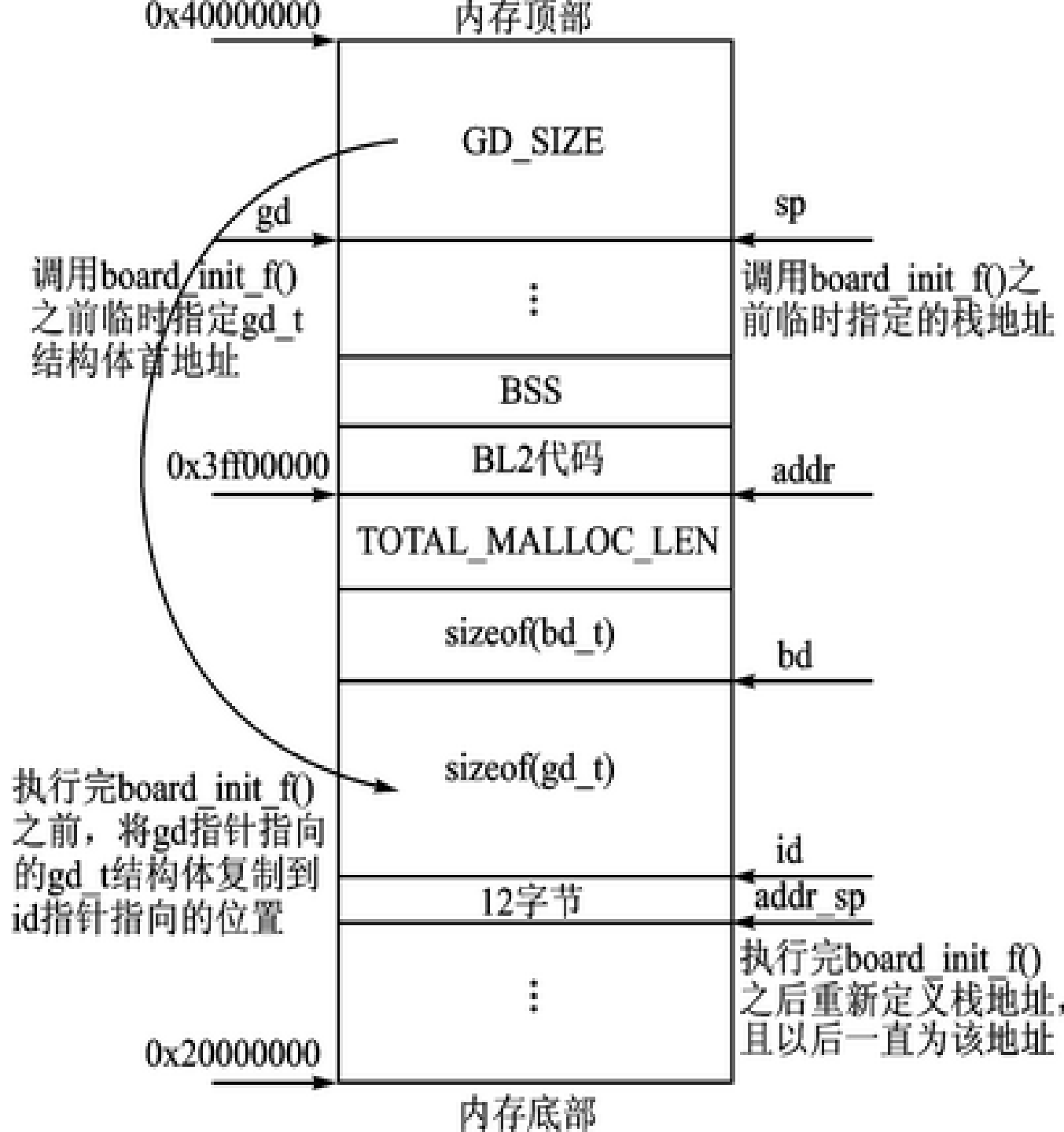
初始化完成之后，U-Boot首先调用拷贝函数将BL2拷贝到内存地址为0x 3FF00000处，然后跳转到该位置执行BL2。在U-Boot中，BL1和BL2是基于相同的一些源文件编译生成的。开发者在编写代码时需要使用预编译宏CONFIG_SPL_BUILD来实现BL1和BL2不同的功能。

2. 第二阶段初始化

进入第二阶段后，U-Boot首先声明一个gd_t结构体类型的指针指向内存地址（0x 40000000~GD_SIZE）处。0x 40000000为内存结束地址，GD_SIZE为结构体gd_t的大小，这样相当于在内存最顶端分配了一段空间用于存放一个临时结构体gd_t。该结构体在global_data.h中被定义，U-Boot用它来存储所有的全局变量。之后U-Boot会调用board_init_f()和board_init_r()两个函数进一步对底板进行初始化。

(1) board_init_f()

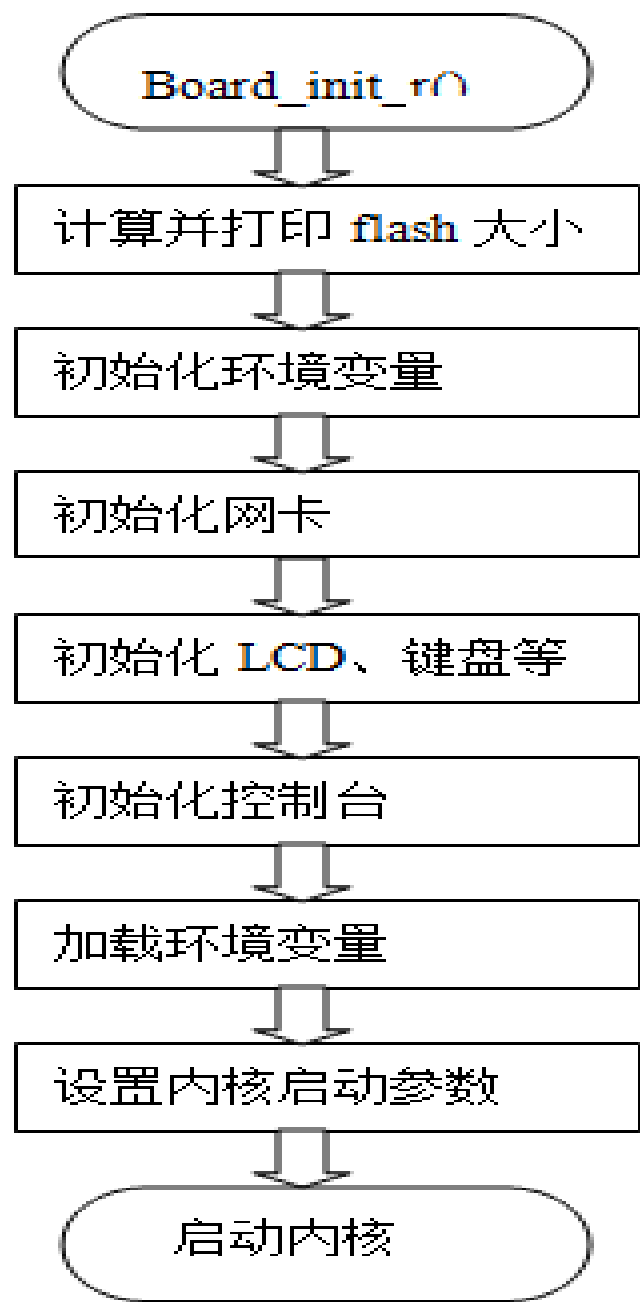
进入board_init_f()之后，U-Boot首先设置之前分配的临时结构体，然后开始划分内存空间



从图7-4中可以发现，gd指针指向的临时结构体存放于内存的最顶部。**BL2**代码存放在内存地址**0x 3ff00000**处，即距离内存顶部**1 MB**空间的位置，接下来依次分配**malloc**空间、**bd_t**结构体空间和**gd_t**结构体空间，并且重新设置栈，最后将临时结构体拷贝到**ID**指针所指向的位置。

(2) board_init_r()

board_init_r()负责对其他硬件资源进行初始化，如网卡、Flash、MMC、中断等，最后调用main_loop()，等待用户输入命令。



7.2.3 U-Boot环境变量

U-Boot的环境变量是使用U-Boot的关键，它可以由用户定义并遵守约定俗成的一些用法，也有部分是U-Boot定义并不得更改。

值得注意的是在未初始化的开发板中并不存在环境变量。U-Boot在缺省的情况下会存在一些基本的环境变量，当用户执行了`saveenv`命令之后，环境变量会第一次保存到flash中，之后用户对环境变量的修改和保存都是基于保存在flash中的环境变量的操作。

表7-3 U-Boot常用环境变量

| 环境变量名称 | 相关描述 |
|-----------|-------------|
| bootdelay | 执行自动启动的等候秒数 |
| baudrate | 串口控制台的波特率 |
| netmask | 以太网接口的掩码 |
| ethaddr | 以太网卡的网卡物理地址 |
| bootfile | 缺省的下载文件 |
| bootargs | 传递给内核的启动参数 |
| bootcmd | 自动启动时执行的命令 |
| serverip | 服务器端的ip地址 |
| ipaddr | 本地ip 地址 |
| stdin | 标准输入设备 |
| stdout | 标准输出设备 |
| stderr | 标准出错设备 |

U-Boot的环境变量中最重要的两个变量是：
bootcmd 和 **bootargs**。

Bootcmd是自动启动时默认执行的一些命令，因此用户可以在当前环境中定义各种不同配置，不同环境的参数设置，然后通过bootcmd配置好参数。

Bootargs是环境变量中的重中之重。

7.2.4 U-Boot命令

U-Boot上电启动后，**按任意键退出自启动状态，进入命令行状态。**

在提示符下，可以输入U-Boot特有的命令完成相应的功能。U-Boot提供了更加周详的命令帮助，通过**help**命令不仅可以得到当前U-Boot的所有命令列表，还能够查看每个命令的参数说明。

7.3

Part Three

7.3 交叉开发环境的建立

Linux本地软件开发模式

1、程序编辑

vi debug.c

```
1 #include <stdio.h>
2 int func(int n)
3 {
4     int sum = 0, i;
5     for(i=0; i<n; i++)
6     {
7         sum += i;
8     }
9     return sum;
10 }
11
12 main()
13 {
14     int i;
15     long result=0;
16     for(i=1; i<=100; i++)
17     {
18         result+=i;
19     }
20     printf("result[1-100]=%d \n", result);
21     printf("result[1-250]=%d \n", func(250));
22 }
```

2、程序编译

gcc debug.c -o debug -g

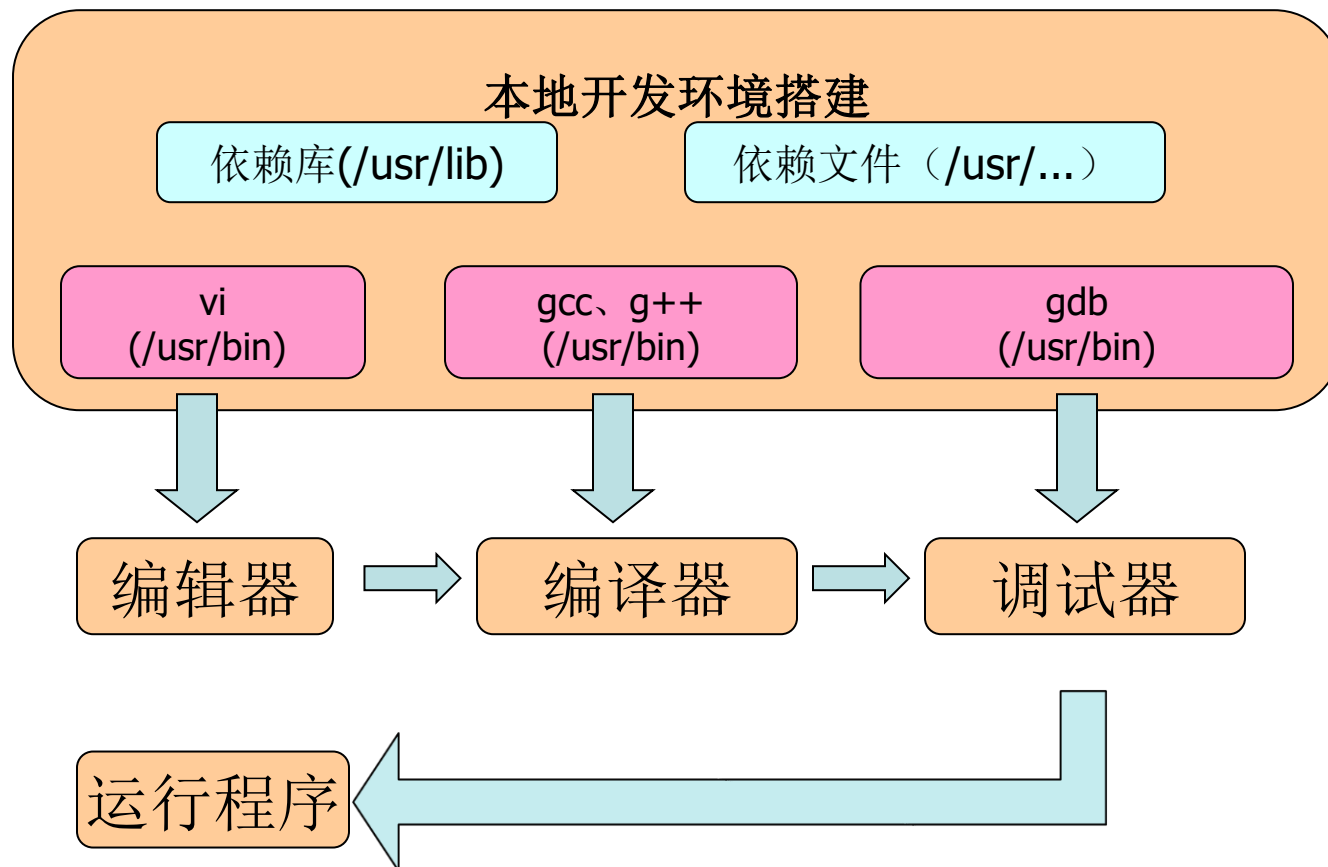
3、程序运行

./debug

4、程序调试

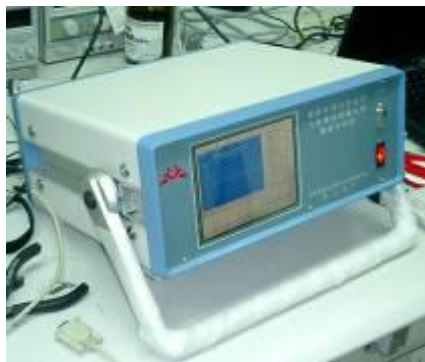
gdb debug

Linux本地软件开发环境



嵌入式系统不具备自举开发能力

◆ 由于计算、存储、显示等资源受限，嵌入式系统无法完成自举开发。

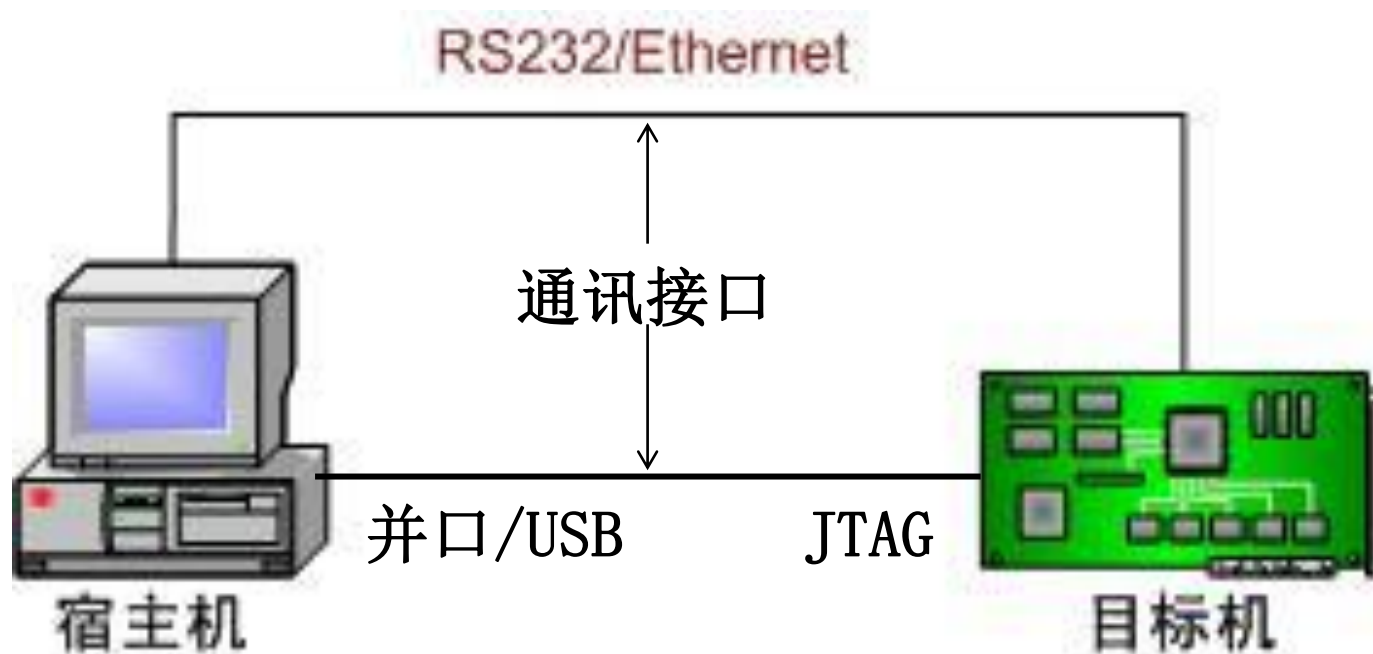


嵌入式软件开发模式

- ◆ 嵌入式系统资源受限，直接在嵌入式系统硬件平台上编写软件较为困难。
- ◆ 解决方法
 - 首先在通用计算机上编写软件
 - 然后通过本地编译或者交叉编译生成目标平台上可以运行的二进制代码格式
 - 最后再下载到目标平台上运行

主机—目标机交叉开发模式

- ◆ 嵌入式系统采用双机开发模式：主机—目标机开发模式，利用资源丰富的PC机来开发嵌入式软件。



宿主机：资源丰富

目标机：资源受限

主机-目标机交叉开发环境模式是由**开发主机**和**目标机**两套计算机系统内组成的。

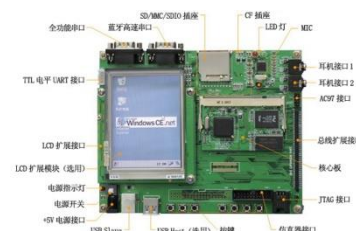
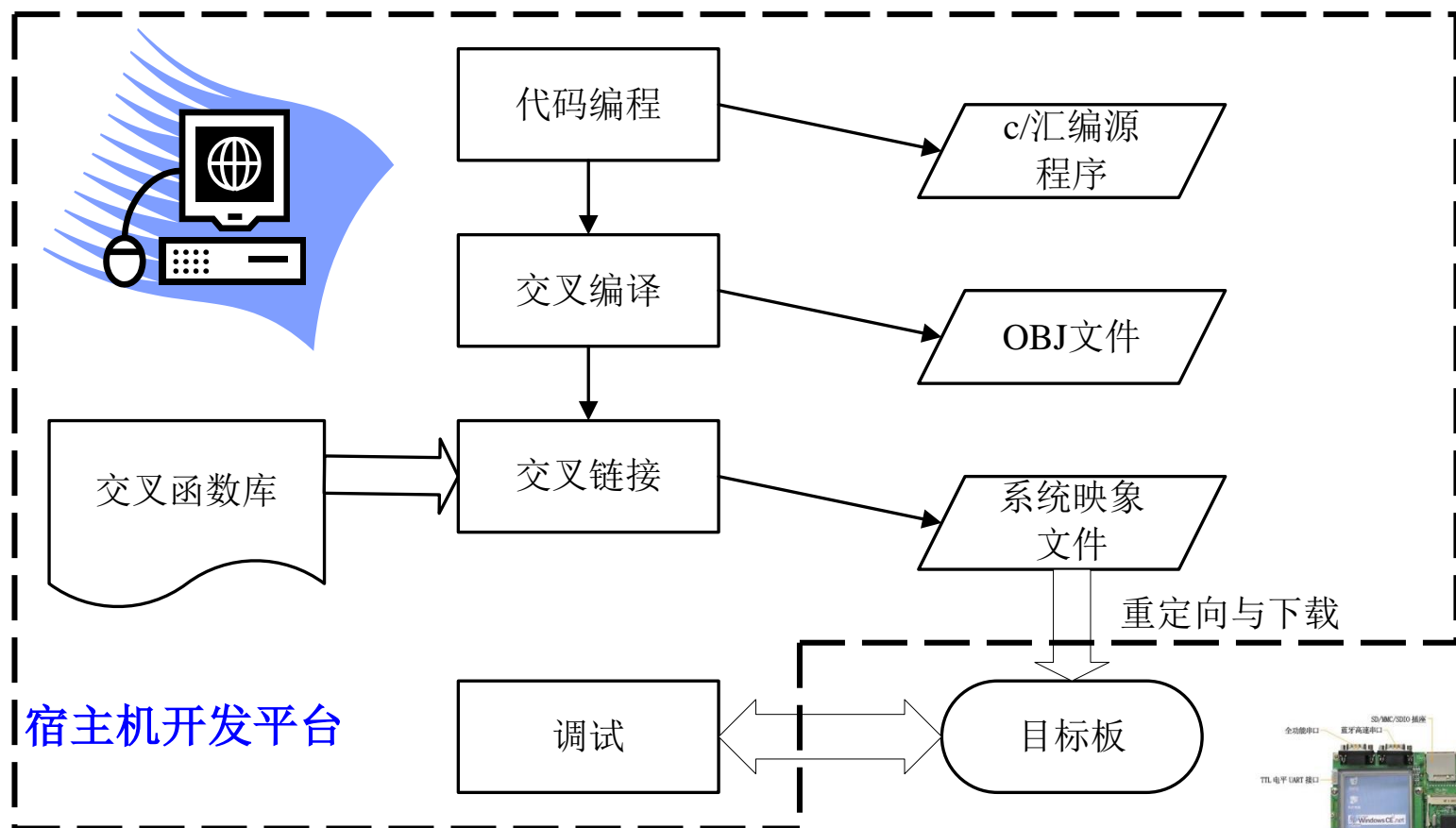
开发主机一般指通用计算机，如**PC**等，
目标机指嵌入式开发板（系统）。

通过交叉开发环境，在主机上使用开发工具（如各种**SDK**），针对目标机设计应用系统进行设计工作，然后下载到目标机上运行。

交叉开发模式一般采用以下3个步骤：

- (1) 在主机上编译 BootLoader (引导加载程序)，然后通过 JTAG 接口烧写到目标板。
- (2) 在主机上编译 Linux 内核，然后通过 BootLoader 下载到目标板以启动或烧写到 Flash。
- (3) 在主机上编译各类应用程序，通过 NFS 运行、调试这些程序，验证无误后再将制作好的文件系统映像烧写到目标板。

嵌入式软件开发流程



7.3.1. 主机与目标机的连接方式

主机与目标机的连接方式主要有串口、以太网接口、USB接口、JTAG接口等方式连接。

主机可以使用minicom、kermit或者Windows超级终端等工具，通过串口发送文件。

目标机亦可以把程序运行结果通过串口返回并显示。

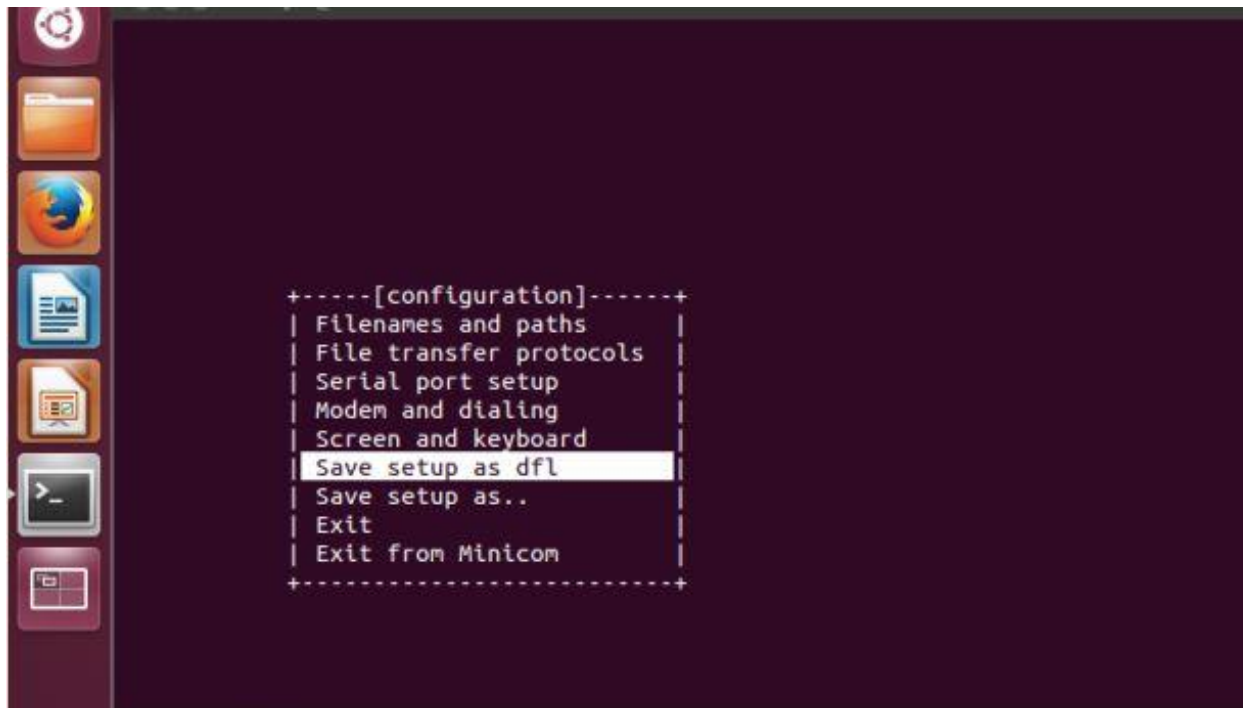
以太网接口方式使用简单，配置灵活，支持广泛，传输速率快，缺点是网络驱动的实现比较复杂。

Linux宿主机串口通讯简介—minicom

◆ Minicom对串口数据传输的配置

```
[root@XSBase home]# minicom -s
```

- 若目标机接在COM1上，则输入/dev/ttyS0;若接在COM2上则输入/dev/ttyS1。
- Speed为115200
- Parity bit为No
- Data bit为8
- Stop bits为1



Linux宿主机串口通讯简介—minicom

- 设置正确后，目标板启动显示信息如下：

```
pyl@mc: ~  
[ 2.400000] VFS: Mounted root (nfs filesystem) on device 0:13.  
[ 2.407000] devtmpfs: mounted  
[ 2.411000] Freeing init memory: 224K  
[ 2.415000] Write protecting the kernel text section 80008000 - 808d2000  
[ 2.422000] rodata_test: attempting to write to read-only section:  
[ 2.428000] write to read-only section trapped, success  
  
Please press Enter to activate this console. [ 5.827000] PHY: stmmac-0:07 - l  
  
/ # ls  
bin      etc      lib      proc     sys      usr  
dev      home    linuxrc  sbin     tmp  
/ # uname -r  
3.4.39  
/ # uname -a  
Linux 192.168.1.200 3.4.39 #1 SMP PREEMPT Sun Nov 29 17:13:58 CST 2015 armv7l Gx  
/ # cd /dev/  
/dev # ls  
alarm          loop-control   network_throughput  usbdev1.3  
android_adb    loop0          null                usbdev2.1  
ashmem         loop1          nxp-scaler          v4l-subdev0  
autofs         loop2          ppp                 v4l-subdev1  
binder         loop3          psaux               v4l-subdev10  
bus            loop4          ptmx                v4l-subdev11
```


JTAG（**Joint Test Action Group**，联合测试行动小组）是一种国际标准测试协议（**IEEE1149.1**标准），主要是用于对目标机系统中的各芯片的简单调试，和对**BootLoader**的下载两个功能。**JTAG**连接器中，其芯片内部封装了专门的测试电路**TAP**（**Test Access Port**,测试访问口），通过专用的**JTAG**测试工具对内部节点进行测试。因而该方式是开发调试嵌入式系统的一种简洁高效的手段。**JTAG**有两种标准，14针接口和20针接口。

JTAG接口一端与PC机并口相连，另一端是面向用户的**JTAG**测试接口，通过本身具有的边界扫描功能便可以对芯片进行测试，从而达到处理器的启动和停滞、软件断点、单步执行和修改寄存器等功能的调试目的。其内部主要是由**JTAG**状态机和**JTAG**扫描链的组成。

虽然**JTAG**调试不占用系统资源，能够调试没有外部总线的芯片，代价也非常小，但是**JTAG**只能提供一种静态的调试方式，不能提供处理器实时运行时的信息。它是通过串行方式依次传递数据的，所以传送信息速度比较慢。

7.3.2. 主机-目标机的文件传输方式

主机-目标机的文件传输方式主要有串口传输方式、网络传输方式、**USB**接口传输方式、**JTAG**接口传输方式、移动存储设备方式。

串口传输协议常见的有kermit、Xmodem、Ymoderm、Zmoderm等。串口驱动程序的实现相对简单，但是速度慢，不适合较大文件的传输。

USB接口方式通常将主机设为主设备端，目标机设为从设备端。与其他通信接口相比，**USB**接口方式速度快，配置灵活，易于使用。如果目标机上有移动存储介质如U盘等，可以制作启动盘或者复制到目标机上，从而引导启动。

网络传输方式一般采用TFTP (trivial file transport protocol) 协议。TFTP是一个传输文件的简单协议，是TCP/IP协议族中的一个用来在客户机与服务器之间进行简单文件传输的协议，提供不复杂、开销不大的文件传输服务。端口号为69。此协议只能从文件服务器上获得或写入文件，不能列出目录，不进行认证，它传输8位数据。

传输中有三种模式：

`netascii`，这是8位的ASCII码形式；

另一种是`octet`，这是8位源数据类型；

最后一种`mail`已经不再支持，它将返回的数据直接返回给用户而不是保存为文件。

7.3.3.文件系统的挂接-配置网络文件系统NFS

NFS（Network File System）即网络文件系统，是**FreeBSD**支持的文件系统中的一种，它允许网络中的计算机之间通过**TCP/IP**网络共享资源。在**NFS**的应用中，本地**NFS**的客户端应用可以透明地读写位于远端**NFS**服务器上的文件，就像访问本地文件一样。

NFS的优点主要有：

(1) 节省本地存储空间，将常用的数据存放在一台NFS服务器上且可以通过网络访问，那么本地终端将可以减少自身存储空间的使用。

(2) 用户不需要在网络中的每个机器上都建有Home目录，Home目录可以放在NFS服务器上且可以在网络上被访问使用。

(3) 一些存储设备如软驱、CDROM和Zip等都可以在网络上被别的机器使用。这可以减少整个网络上可移动介质设备的数量。

NFS体系至少有两个主要部分：
一台NFS服务器和若干台客户机。

客户机通过TCP/IP网络远程访问存放在NFS服务器上的数据。在NFS服务器正式启用前，需要根据实际环境和需求，配置一些NFS参数。

7.3.4. 交叉编译环境的建立

交叉编译是在一个平台上生成另一个平台上执行代码。在宿主机上对即将运行在目标机上的应用程序进行编译,生成可在目标机上运行的代码格式。交叉编译环境是由一个编译器、连接器和解释器组成的综合开发环境。交叉编译工具主要包括针对目标系统的编译器、目标系统的二进制工具、目标系统的标准库和目标系统的内核头文件。

- 在**PC平台(X86)**上编译出能运行在**ARM**平台上的程序,即编译得到的程序在**X86**平台上不能运行,必须放到**ARM**平台上才能运行;
- 用来编译这种程序的编译器就叫交叉编译器;
- 为了不与本地编译器混淆,交叉编译器的名字一般都有前缀,例如: **arm-linux-gcc**。

7.4 Part Four

交叉编译工具链

7.4.1交叉编译工具链概述

在一种计算机环境中运行的编译程序，能编译出在另外一种环境下运行的代码，我们就称这种编译器支持交叉编译。这个编译过程就叫交叉编译。

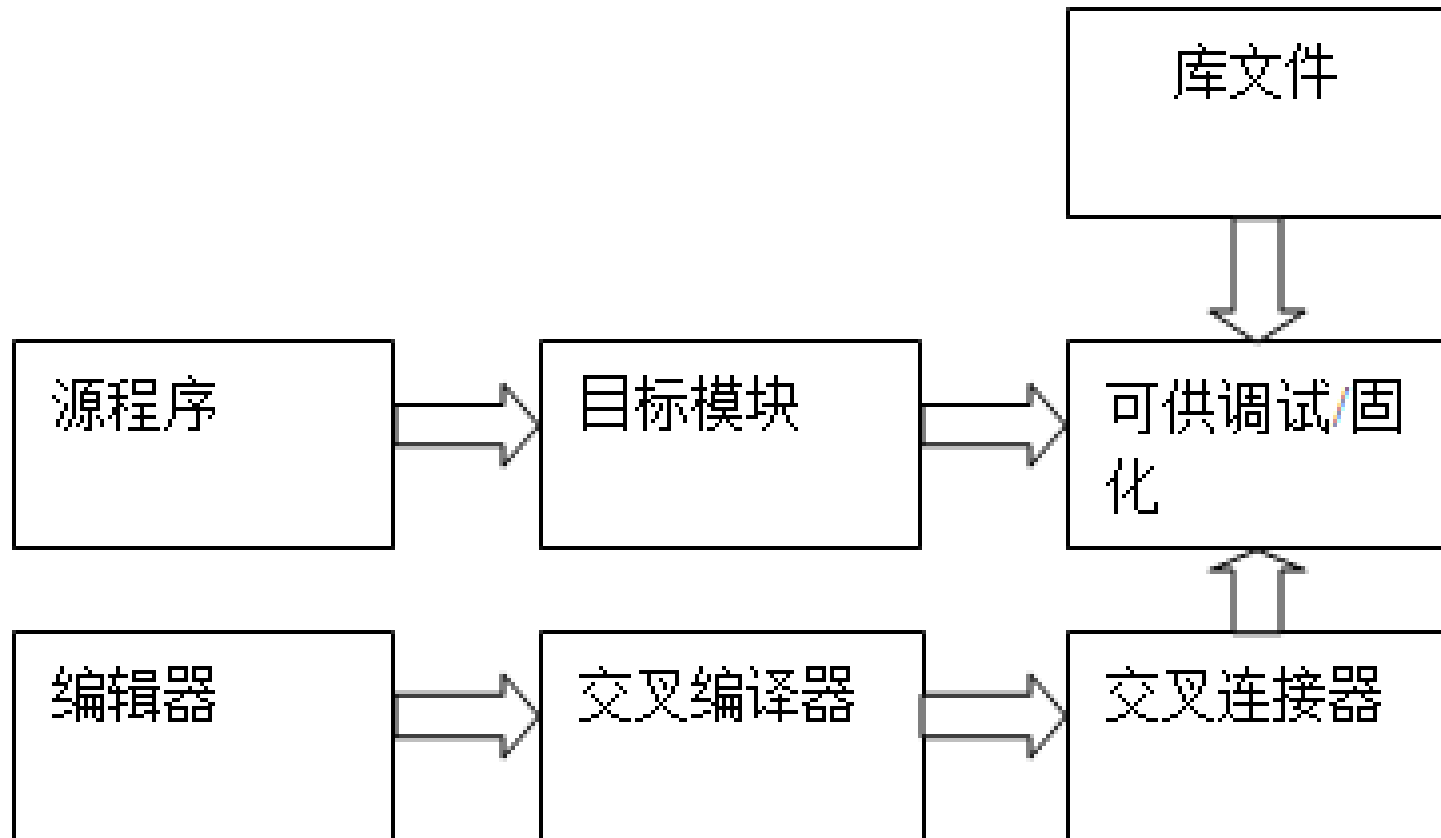
要进行交叉编译，我们需要在主机平台上安装对应的交叉编译工具链（**cross compilation tool chain**），然后用这个交叉编译工具链编译链接源代码，最终生成可在目标平台上运行的程序。

常见的交叉编译例子如下：

（1）在Windows PC上，利用诸如类似ADS、RVDS等软件，使用armcc编译器，则可编译出针对ARM CPU的可执行代码。

（2）在Linux PC上，利用arm-linux-gcc编译器，可编译出针对Linux ARM平台的可执行代码。

（3）在Windows PC上，利用cygwin环境，运行arm-elf-gcc编译器，可编译出针对ARM CPU的可执行代码。

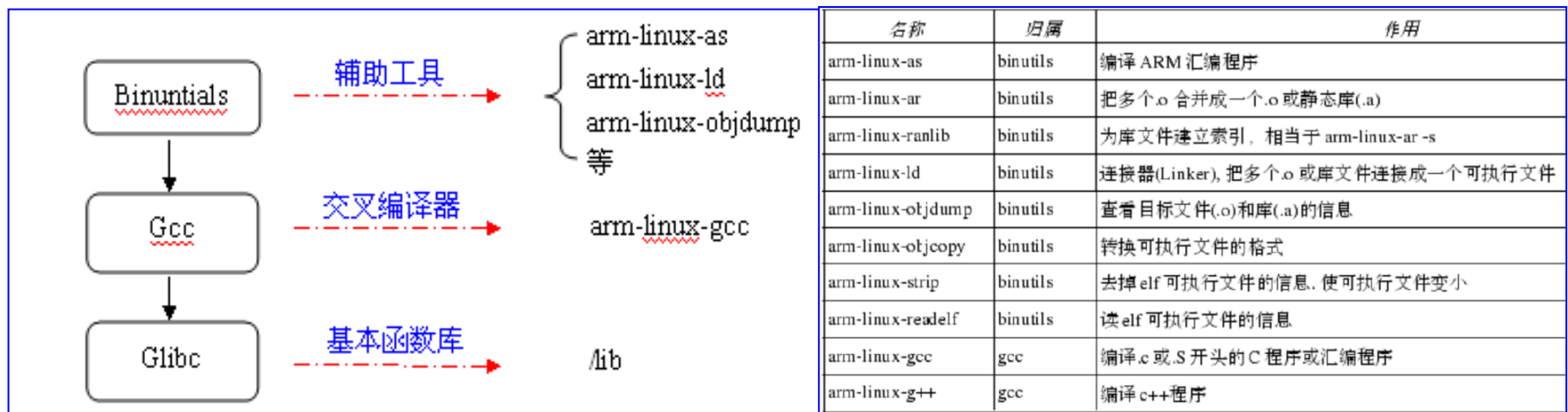


交叉开发工具链就是为了编译、链接、处理和调试跨平台体系结构的程序代码。每次执行工具链软件时，通过带有不同的参数，可以实现编译、链接、处理或者调试等不同的功能。从工具链的组成上来说，它一般由多个程序构成，分别对应着各个功能。

构建交叉编译环境

构建交叉编译环境所需的工具链主要包括：

- 交叉编译器，例如**arm-linux-gcc**
- 交叉汇编器，例如**arm-linux-as**
- 交叉链接器，例如**arm-linux-ld**
- 用于处理可执行程序 and 库的一些基本工具，例如**arm-linux-strip**



7.4.2 工具链的构建方法

通常构建交叉工具链有如下三种方法。

方法一：分步编译和安装交叉编译工具链所需要的库和源代码，最终生成交叉编译工具链。该方法相对比较困难，适合想深入学习构建交叉工具链的读者及用户。如果只是想使用交叉工具链，建议使用下列的方法二构建交叉工具链。

方法二：通过Crosstool脚本工具来实现一次编译，生成交叉编译工具链，该方法相对于方法一要简单许多，并且出错的机会也非常少，建议大多数情况下使用该方法构建交叉编译工具链。

方法三：直接通过网上下载已经制作好的交叉编译工具链。该方法的优点是简单可靠，缺点也比较明显，扩展性不足，对特定目标没有针对性，而且也存在许多未知错误的可能，建议读者慎用此方法。

交叉编译器的安装

1) 安装交叉编译工具链(注意：此版本可能与后续的交叉编译链版本不同)：

将光盘中交叉编译工具的源码 `arm-none-linux-gnueabi-arm-2010-09-50-for-linux.tar.bz2` 拷贝到用户目录下（文档以 `/usr/local/arm` 为例），并解压：

```
#tar jxvf arm-none-linux-gnueabi-arm-2010-09-50-for-linux.tar.bz2 -C /usr/local/arm
```

2) 修改环境变量：

```
#vim ~/.bashrc
```

在文件末添加

```
export PATH=/usr/local/arm/4.5.1/bin/:$PATH
```

编译器安装成功。

```
#source ~/.bashrc
```


7.4.3 交叉编译工具链的主要工具

交叉编译工具主要包括针对目标系统的编译器、目标系统的二进制工具、调试器、目标系统的标准库和目标系统的内核头文件。主要由glibc、gcc、binutils和gdb四个软件提供。Gdb调试器作为单独一小节在7.6小节介绍。

1. GCC

通常所说的GCC是GNU Compiler Collection的简称，除了编译程序之外，它还含其他相关工具，所以它能把高级语言编写的源代码构建成计算机能够直接执行的二进制代码。GCC是Linux平台下最常用的编译程序，它是Linux平台编译器的实际上的事实标准。同时，在Linux平台下的嵌入式开发领域，GCC也是用得最普遍的一种编译器。

对于GNU编译器来说，GCC的编译要经历四个相互关联的步骤：**预处理**(也称预编译，Preprocessing)、**编译**(Compilation)、**汇编**(Assembly)和**链接**(Linking)。

源代码（这里以file.c为例）经过四个步骤后从而产生一个可执行文件，各部分对应不同的文件类型，具体如下：

| | |
|------------------------|---------------------------------|
| file.c | c程序源文件 |
| file.i | c程序预处理后文件 |
| file.cxx / file.c++ | c++程序源文件，也可以是file.cc / file.cpp |
| file.ii | c++程序预处理后文件 |
| file.h | c/c++头文件 |
| file.s | 汇编程序文件 |
| file.o | 目标代码文件 |

2. Binutils

Binutils提供了一系列用来创建、管理和维护二进制目标文件的工具程序，如汇编（**as**）、连接（**ld**）、静态库归档（**ar**）、反汇编（**objdump**）、**elf**结构分析工具（**readelf**）、无效调试信息和符号的工具（**strip**）等。通常**binutils**与**gcc**是紧密相集成的，没有**binutils**的话，**gcc**是不能正常工作的。

(1) 编译单个文件

`vi hello.c` //创建源文件hello.c

`gcc -o hello hello.c` //编译为可执行文件hello，在默认情况下产生的可执行文件名为a.out

`./hello` //执行文件，如果只写hello是错误的，因为系统会将hello当指令来执行，然后报错

(2) 编译多个源文件

vi message.c

gcc -c message.c //输出message.o文件，是一个已编译的
目标代码文件

vi main.c

gcc -c main.c //输出main.o文件

gcc -o all main.o message.o //执行连接阶段的工作，然后生成
all可执行文件

./all

注意：gcc对如何将多个源文件编译成一个可执行文件有内置的规则，所以前面的多个单独步骤可以简化为一个命令。

vi message.c

vi main.c

gcc -o all message.c main.c

./all

(3) 使用外部函数库

GCC常常与包含标准例程的外部软件库结合使用，几乎每一个linux应用程序都依赖于**GNU C**函数库**GLIBC**。

```
vi trig.c
```

```
gcc -o trig -lm trig.c
```

GCC的-lm选项，告诉**GCC**查看系统提供的数学库libm。函数库一般会位于目录/lib或者/usr/lib中。

（4）共享函数库和静态函数库

静态函数库：每次当应用程序和静态连接的函数库一起编译时，任何引用的库函数的代码都会被直接包含进最终二进制程序。

共享函数库：包含每个库函数的单一全局版本，它在所有应用程序之间共享。

```
vi message.c
```

```
vi hello.c
```

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

```
gcc -fPIC -c message.c
```

```
gcc -shared -o libmessage.so message.o
```

其中，**PIC**命令行标记告诉**GCC**产生的代码不要包含对函数和变量具体内存位置的引用，这是因为现在还无法知道使用该消息代码的应用程序会将它链接到哪一段地址空间。这样编译输出的文件**message.o**可以被用于建立共享函数库。**-shared**标记将某目标代码文件转换成共享函数库文件。

```
gcc -o all -lmessage -L. hello.o
```

-lmessage标记来告诉**GCC**在连接阶段使用共享数据库**libmessage.so**，**-L.**标记告诉**GCC**函数库可能在当前目录中，首先查找当前目录，否则**GCC**连接器只会查找系统函数库目录，在本例情况下，就找不到可用的函数库了。

3. glibc

Glibc是gnu发布的**libc**库，也即**c**运行库。**Glibc**是linux系统中最底层的应用程序开发接口，几乎其它所有的运行库都倚赖于**glibc**。**Glibc**除了封装linux操作系统所提供的系统服务外，它本身也提供了许多其它一些必要功能服务的实现，比如**open**, **malloc**, **printf**等等。**Glibc**是**GNU**工具链的关键组件，用于和二进制工具和编译器一起使用，为目标架构生成用户空间应用程序。

7.4.4 Makefile

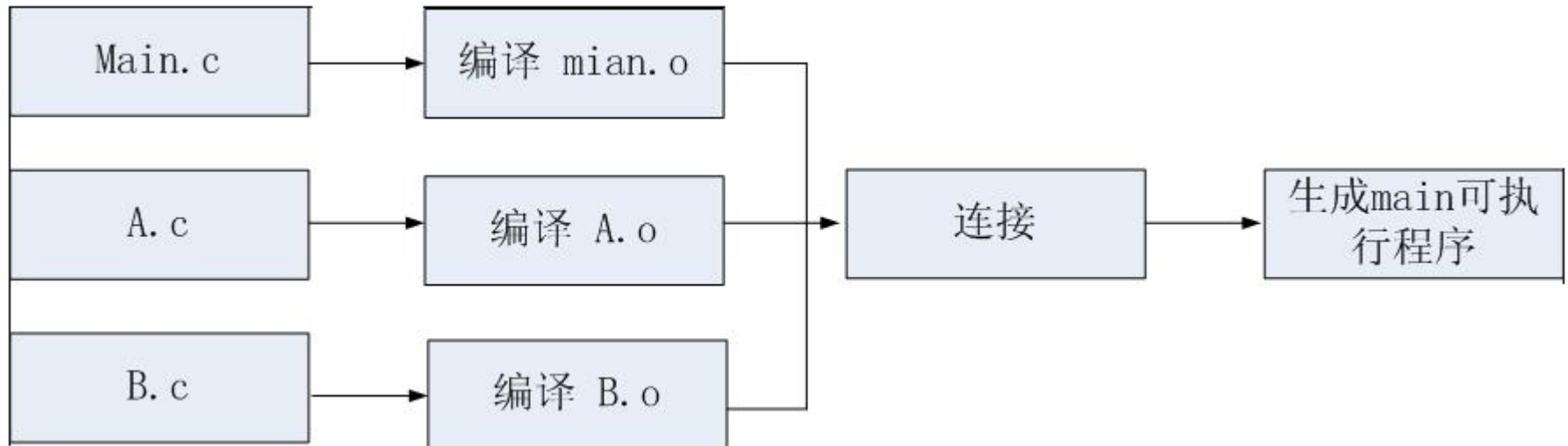
GNU make是一种常用的编译工具，通过它，开发人员可以很方便地管理软件编译的内容、方式和时机，从而能够把主要精力集中在代码的编写上。**Gnu make**的主要工作是读取一个文本文件**makefile**。这个文件里主要是有关目的文件是从哪些依赖文件中产生的，以及用什么命令来进行这个产生过程。

在嵌入式系统的程序开发中，通常一个较大的程序都会使用到不同的小程序或函数，所以在编译时就要将这些不同的程序编译，产生不同的目标文件，然后再执行连续的动作，最后才能生成可执行的二进制程序。

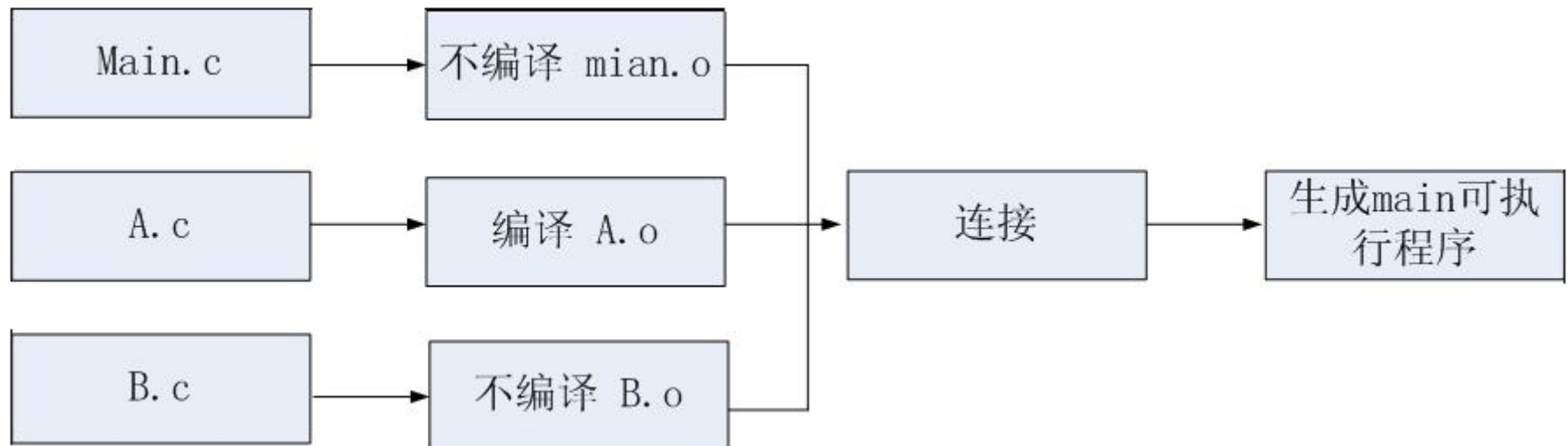
Make工程管理器

- 例如有一个主程序为**main.c**，需要使用到**A.c**和**B.c**的程序，因此在编译时就要执行如下命令才能产生可执行的二进制程序**main**。

- `gcc -c main.c` （生成main.o目标文件）
- `gcc -c A.c` （生成A.o目标文件）
- `gcc -c B.c` （生成B.o目标文件）
- `gcc -o main main.o A.o B.o`
- 最后根据main.o、 A.o 、 B.o这3个目标文件，才能生成main可执行二进制程序。



- 但是当在执行程序时发现程序执行的结果是由于A.c程序源代码有错误，此时就要修改A.c程序代码后再执行上图的编译过程，由于main.c及B.c程序并没有错误，且在第一次执行编译时已经有目标文件main.o和B.o产生了，为了提高编译效率，GNU gcc提供了自动化编译工具Make，其功能就是在执行编译时，只针对修改的部分进行编译，没修改的程序部分不做编译。



- 自动化编译工具**Make**（工程管理器），是指管理较多的文件的编译工具。
- 实际上，**Make**工程管理器也就是个“**自动编译管理器**”，这里的“自动”是指它能够根据文件时间戳自动发现更新过的文件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。用户只需编写一次简单的编译语句就可以了。它大大提高了实际项目的工作效率，而且几乎所有**Linux**下的项目编程均会涉及它。

Make的优点

Make的优点如下：

- 1) 对庞大及复杂的c源代码文件进行有效的维护。
- 2) 减少程序编译的次数。
- 3) 使源代码的编译、连接、管理更加有效。
- 4) 具有编辑自动化的功能，将编译(Compiler)、连接(Link)、产生可执行二进制程序的动作自动化完成。

Make工具需要Makefile文件

1) 首先编译一个名为Makefile的文件。

Makefile文件主要描述了各个文件间的依赖关系和更新命令。

2) 编辑好了Makefile文件后，每次更新程序源代码后，只要输入make命令就可以进行编译了。

Makefile书写规则

Makefile文件含有一系列的规则，规则内容：

- 一个目标（target），即make最终需要创建的文件，如可执行文件和目标文件；目标也可以是要执行的动作，如clean。
- 一个或多个依赖文件（dependency）列表，通常是编译目标文件所需要的其他文件。
- 一系列命令（command），是make执行的动作，通常是把指定的相关文件编译成目标文件的编译命令，每个命令占一行，且每个命令行起始字符必须为TAB字符。

Makefile规则的一般形式如下：

**target: dependency dependency
(tab)<command>**

例1： 有以下的Makefile文件：

hello: hello1.o hello2.o hello3.o

(tab) gcc -o hello hello1.o hello2.o hello3.o

第一行表示hello目标文件需要hello1.o hello2.o hello3.o这3个目标文件。

第二行表示编译生成hello可执行二进制程序时，gcc指令需要连接hello1.o hello2.o hello3.o这3个目标文件。

例2： 有以下的Makefile文件：

```
hello.o: hello.c hello.h
```

```
(tab) gcc -c hello.c
```

表示hello.o目标文件需要依赖hello.c和hello.h文件，若hello.c和hello.h文件在编译时有一个被修改，执行make时就会自动重新编译生成hello.o目标文件，若hello.c和hello.h文件在编译时，没有被修改，则执行make时就不会再重新编译。

举例

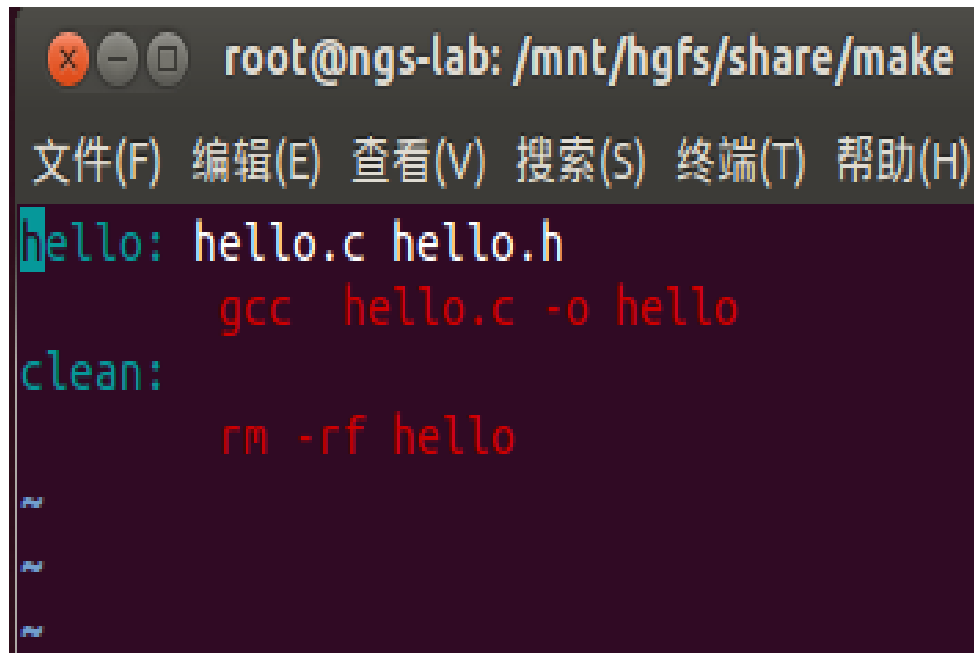
例题1：有以下hello.c源代码和hello.h头文件，代码都在不同的两个文件中，编写makefile文件，使用make工程管理器对它们进行编译。

hello.c源代码：

```
/*hello.c*/  
int main()  
{  
    printf("Hello!This is our embedded world!\n");  
    return 0;  
}
```

hello.h头文件：

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



```
root@ngs-lab: /mnt/hgfs/share/make
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
hello: hello.c hello.h
    gcc hello.c -o hello
clean:
    rm -rf hello
~
~
~
```

图1 例题1的makefile文件

举例

例题2：如图有一个主程序（main.c）可输入两个整数a和b，其中主程序会执行一个求两整数和的函数add()后输出其和，然后再执行一个求两整数差的函数sub()后输出其差，add()和sub()这两个函数分别定义在add.c和sub.c的文件中，这两个函数的声明是定义在main.h的头文件中，其程序源代码分别如下所述。

主程序main.c源代码：

```
/*main.c*/  
#include<stdlib.h>  
#include"main.h"  
int main()  
{  
    int a,b,c,d;  
    printf("Please input two parms:");  
    scanf("%d%d",&a,&b);  
    c=add(a,b);  
    printf("the sum result is %d\n",c);  
    d=sub(a,b);  
    printf("the dif result is %d\n",d);  
    return 0;  
}
```

- 头文件main.h中包含求两整数和及差的原型声明，其程序源代码如下所述。

- **/*main.h*/**

- **int add(int,int);**

- **int sub(int,int);**

- 求两整数和的函数add()，定义在add.c程序中，其程序源代码如下所述。

- **/*add.c*/**

- **int add(int a,int b)**

- **{ int s;**

- **s=a+b;**

- **return(s);**

- **}**

- 求两整数差的函数sub()，定义在sub.c程序中，其程序源代码如下所述。

- **/*sub.c*/**

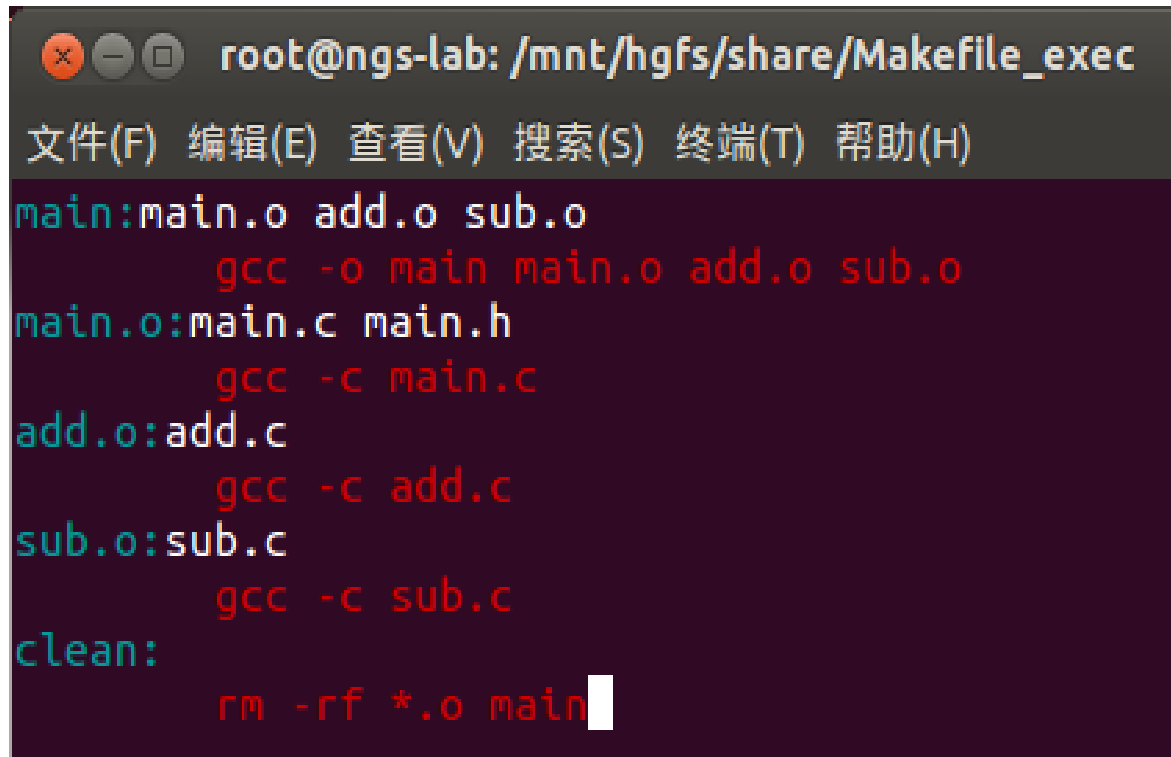
- **int sub(int c,int d)**

- **{ int dif;**

- **dif=c-d;**

- **return(dif);**

- **}**

A terminal window with a dark background and light-colored text. The title bar shows a window icon, a close button, and the text 'root@ngs-lab: /mnt/hgfs/share/Makefile_exec'. Below the title bar is a menu bar with the text '文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)'. The main area of the terminal displays a Makefile with the following content:

```
main:main.o add.o sub.o
    gcc -o main main.o add.o sub.o
main.o:main.c main.h
    gcc -c main.c
add.o:add.c
    gcc -c add.c
sub.o:sub.c
    gcc -c sub.c
clean:
    rm -rf *.o main
```

图2 例题2的makefile文件

Makefile变量的应用

在Makefile中可使用一些变量来代表一串文字。

1、objects变量主要用于代表一些目标文件。

如：objects=hello.o hello1.o hello2.o

hello:\$ (objects)

2、当在Makefile文件中希望同时编译很多个输出文件时则可使用all。

如：all:hello hello3

执行make命令后，就会产生hello和hello3两个可执行的二进制程序。

3、clean

在执行make clean命令时会执行clean后面的命令。

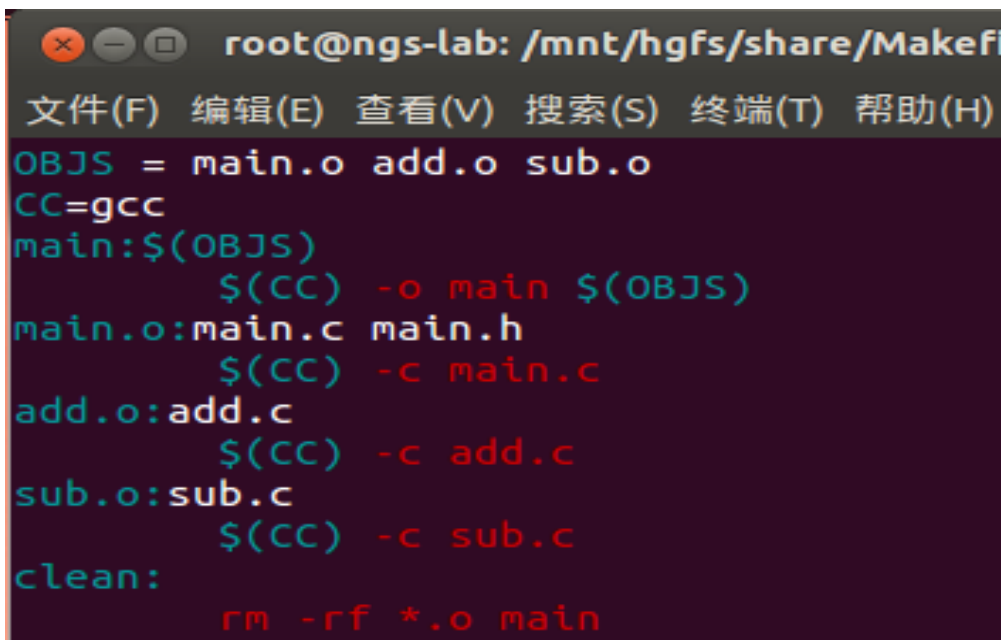
如：clean:

```
rm hello *.o
```

Makefile变量分类

1、用户自定义变量

- **OBJS**变量就是用户自定义变量，自定义变量的值由用户自行设定。
- 如下面的**OBJS**就是用户自定义变量。



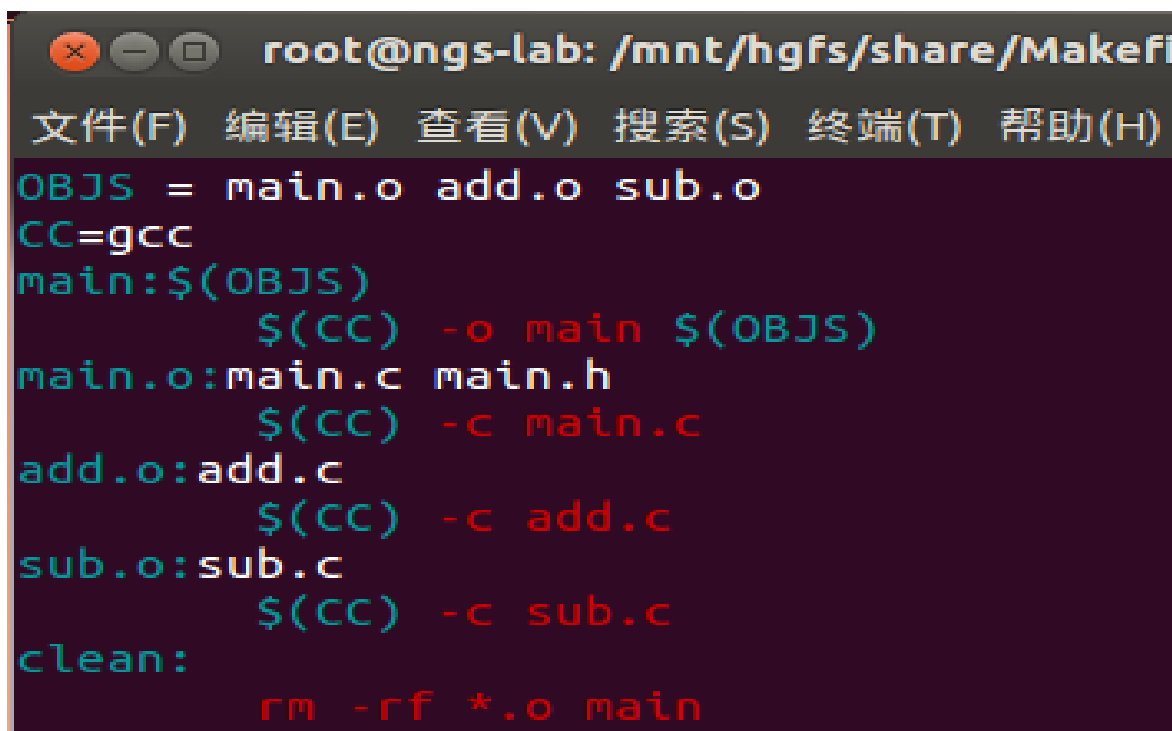
```
root@ngs-lab: /mnt/hgfs/share/Makefi
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
OBJS = main.o add.o sub.o
CC=gcc
main:$(OBJS)
    $(CC) -o main $(OBJS)
main.o:main.c main.h
    $(CC) -c main.c
add.o:add.c
    $(CC) -c add.c
sub.o:sub.c
    $(CC) -c sub.c
clean:
    rm -rf *.o main
```

• 2、预定义变量

- 预定义变量包含了常见编译器、汇编器的名称及其编译选项。

| 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↵ | FORTTRAN 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↵ | FORTTRAN 编译器的选项，无默认值↵ |

- 如下图的**CC**就是预定义变量。



```
root@ngs-lab: /mnt/hgfs/share/Makefi
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
OBJS = main.o add.o sub.o
CC=gcc
main:$(OBJS)
    $(CC) -o main $(OBJS)
main.o:main.c main.h
    $(CC) -c main.c
add.o:add.c
    $(CC) -c add.c
sub.o:sub.c
    $(CC) -c sub.c
clean:
    rm -rf *.o main
```

Make管理器的使用

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。

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，则打印当前目录名↵ |

7.5 Part Five

嵌入式Linux系统移植过程

嵌入式Linux系统的移植主要针对BootLoader（最常用的是U-Boot）、Linux内核、文件系统这三部分展开工作。

嵌入式 Linux 系统移植的一般流程是：

首先构建嵌入式 Linux 开发环境，包括硬件环境和软件环境；其次，移植引导加载程序BootLoader；

然后，移植 Linux 内核和构建根文件系统；

最后，一般还要移植或开发设备驱动程序。

这几个步骤完成之后，嵌入式 Linux 已经可以在目标板上运行起来，开发人员能够在串口控制台进行命令行操作。如果需要图形界面支持，还需要移植位于用户应用程序层次的GUI(Graphical User Interface)，比如 Qtopia、Mini GUI 等。

7.5.1 U-Boot 移植

开始移植U-Boot之前，要先熟悉处理器和开发板。确认U-Boot是否已经支持新开发板的处理器和I/O设备，如果U-Boot已经支持该开发板或者十分相似的开发板，那么移植的过程就将非常简单。整体看来，移植U-Boot就是添加开发板硬件需要的相关文件、配置选项，然后编译和烧写到开发板。

U-Boot 的移植过程主要包括以下四个步骤：

1. 下载U-Boot源码

U-Boot的源码包可以从SourceForge网站下载，具体地址为
<http://sourceforge.net/project/U-Boot>。

2. 修改相应的文件代码

U-Boot 源码文件下包括一些目录文件和文本文件，这些文件可分为“与平台相关的文件”和“与平台无关的文件”，在移植的过程中，需要修改的文件也就是这些与平台相关的文件。检查源代码里面是否有CPU级相关代码，下一步就是查看板级相关代码了。

下面举例简要列举移植2014.07版本到S5PV210处理器上时修改（或添加）的文件。

以下文件均为与CPU级相关的文件：

- U-Boot2014.07/arch/arm/cpu/armv7/start.s

- U-Boot2014.07/arch/arm/cpu/armv7/Makefile

- U-Boot2014.07/arch/arm/include/asm/arch-s5pc1xx/hardware.h

- U-Boot2014.07/arch/arm/lib/board.c

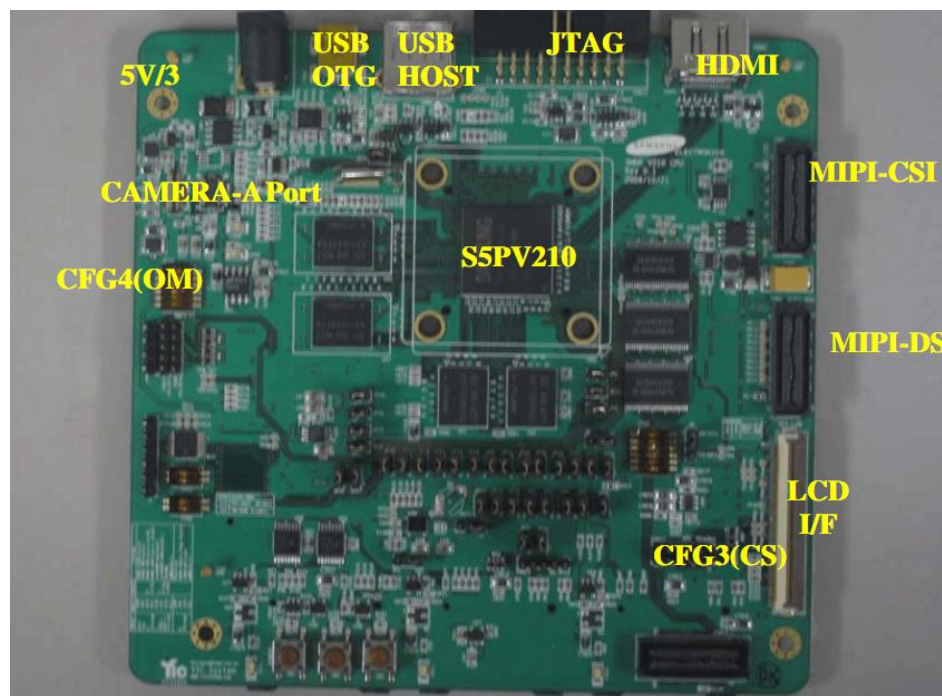
- U-Boot2014.07/arch/arm/lib/Makefile

- U-Boot2014.07/arch/arm/config.mk

以下文件均为与板级相关的文件：

- U-Boot2014.07/board/samsung/SMDKV210/tools/mkv210_image.c
- U-Boot2014.07/board/samsung/SMDKV210/lowlevel_init.S
- U-Boot2014.07/board/samsung/SMDKV210/mem_setup.S
- U-Boot2014.07/board/samsung/SMDKV210/SMDKV210.c
- U-Boot2014.07/board/samsung/SMDKV210/SMDKV210_val.h
- U-Boot2014.07/board/samsung/SMDKV210/mmc_boot.c
- U-Boot2014.07/board/samsung/SMDKV210/Makefile
- U-Boot2014.07/drivers/mtd/nand/s5pc1xx_nand.c
- U-Boot2014.07/drivers/mtd/nand/Makefile
- U-Boot2014.07/include/configs/SMDKV210.h
- U-Boot2014.07/include/s5pc110.h
- U-Boot2014.07/include/s5pc11x.h
- U-Boot2014.07/Makefile

移植过程最主要的就是代码的修改与文件的配置。国内嵌入式厂商研发的**S5PV210**开发板大都基于**SMDKV210评估板**做了减法和调整，所以三星提供的U-Boot、内核、文件系统大都适用于这些**S5PV210**开发板，因而在开发者在此基础上只需要根据相应的makefile文件修改配置即可。



3. 编译U-Boot

U-Boot编译工程通过Makefile来组织编译。顶层目录下的Makefile和boards.cfg中包含开发板的配置信息。从顶层目录开始递归地调用各级子目录下的Makefile，最后链接成U-Boot映像。

U-Boot的编译命令比较简单，主要分两步进行。

第一步是配置，如`make smdkv210_config`;

第二步是编译，执行`make`就可以了。

如果一切顺利，则可以得到U-Boot镜像。

U-BOOT编译生成的映像文件

| 文件名称 | 说明 |
|-------------------|----------------------------|
| System.map | U-Boot映像的符号表 |
| U-Boot | U-Boot映像的ELF格式 |
| U-Boot.bin | U-Boot映像原始的二进制格式 |
| U-Boot.src | U-Boot影响的S-Record格式 |

4. 烧写到开发板上，运行和调试

新开发的板子没有任何程序可以执行，也不能启动，需要先**将U-Boot烧写到flash或者SD卡中**。这里使用最为广泛的硬件设备就是前文介绍过的**JTAG接口**。

表7-8 U-Boot常用工具

| 工具名称 | 说明 |
|---------------------|-------------------|
| bmp_logo | 制作标记的位图结构体 |
| envcrc | 检验U-Boot内部嵌入的环境变量 |
| gen_eth_addr | 生成以太网接口MAC地址 |
| Img2srec | 转换SREC格式映像 |
| mkimage | 转换U-Boot格式映像 |
| updater | U-Boot自动更新升级工具 |

7.5.2 内核的配置、编译和移植

1. Makefile

Linux 内核中的哪些文件将被编译？怎样编译这些文件？连接这些文件的顺序如何？

其实所有这些都是通过Makefile来管理的。在内核源码的各级目录中含有很多个 Makefile 文件，有的还要包含其它的配置文件或规则文件。所有这些文件一起构成了Linux 的 Makefile 体系

表7-9 Linux内核源码Makefile体系的5个部分

| 名称 | 描述 |
|------------------------|---------------------------------------------------------------|
| 顶层Makefile | Makefile 体系的核心，从总体上控制内核的编译、连接 |
| .config | 配置文件，在配置内核时生成。所有的 Makefile 文件都根据 .config的内容来决定使用哪些文件 |
| Arch/\$(ARCH)/Makefile | 与体系结构相关的 Makefile ，用来决定由哪些体系结构相关的文件参与生成内核 |
| Scripts/Makefile.* | 所有 Makefile 共用的通用规则，脚本等 |
| Kbuild Makefile | 各级子目录下的 Makefile ，它们被上一层 Makefile 调用以编译当前目录下的文件 |

Makefile 编译、连接的大致工作流程为：

（1）内核源码根目录下的`.config`文件中定义了很多变量，**Makefile**通过这些变量的值来决定源文件编译的方式(编译进内核、编译成模块、不编译)，以及涉及哪些子目录和源文件。

（2）根目录下的顶层的 **Makefile** 决定根目录下有哪些子目录将被编译进内核，`arch/$(ARCH)/Makefile` 决定 `arch/$(ARCH)`目录下哪些文件和目录被编译进内核。

（3）各级子目录下的 **Makefile** 决定所在目录下的源文件的编译方式，以及进入哪些子目录继续调用它们的 **Makefile**。

（4）在顶层 **Makefile** 和 `arch/$(ARCH)/Makefile` 中还设置了全局的编译、连接选项：**CFLAGS**(编译 C 文件的选项)、**LDFLAGS**(连接文件的选项)、**AFLAGS**(编译汇编文件的选项)、**ARFLAGS**(制作库文件的选项)。

（5）各级子目录下的 **Makefile** 可设置局部的编译、连接选项：**EXTRA_CFLAGS**、**EXTRA_LDFLAGS**、**EXTRA_AFLAGS**、**EXTRA_ARFLAGS**。

（6）最后，顶层 **Makefile** 按照一定的顺序组织文件，根据连接脚本生成内核映像文件。

2. 内核的 **Kconfig** 分析

为了理解 **Kconfig** 文件的作用，需要先了解内核配置界面。在内核源码的根目录下运行命令：

```
# make menuconfig ARCH=arm CROSS_COMPILE=arm-linux-
```

这样会出现一个菜单式的内核配置界面，通过它就可以对支持的芯片类型和驱动程序进行选择，或者去除不需要的选项等，这个过程就称为“配置内核”

在内核源码的绝大多数子目录中，都具有一个 **Makefile** 文件和 **Kconfig** 文件。**Kconfig** 就是内核配置界面的源文件，它的内容被内核配置程序读取用以生成配置界面，从而供开发人员配置内核，并根据具体的配置在内核源码根目录下生成相应的配置文件 **config**。

Kconfig 文件的基本要素是 **config** 条目(**entry**)，它用来配置一个选项，或者说，它用于生成一个变量，这个变量会连同它的值一起被写入配置文件 **.config** 中。

3. 内核的配置选项

Linux 内核配置选项非常多，一般是在某个缺省配置文件的基础上进行修改。

```
ARCH    ?= arm
```

```
CROSS_COMPILE ?= arm-linux-
```

原生的内核源码根目录下是没有配置文件.config 的，一般通过加载某个缺省的配置文件来创建.config 文件，然后再通过命令“make menuconfig”来修改配置。

内核配置的基本原则是把不必要的功能都去掉，不仅可以减小内核大小，还可以节省编译内核和内核模块的时间。

.config - Linux Kernel v2.6.35.6-45.fc14.i686 Configuration**Linux Kernel Configuration**

Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
<M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] built-in [] excluded <M> module < >

General setup --->

- [*] **E**nable loadable module support --->
- *- **E**nable the block layer --->
 - P**rocessor type and features --->
 - P**ower management and ACPI options --->
 - B**us options (PCI etc.) --->
 - E**xecutable file formats / Emulations --->
- *- **N**etworking support --->
 - D**evice Drivers --->
 - F**irmware Drivers --->

v(+)

< **S**elect >< **E**xit >< **H**elp >

4. 内核移植

对于内核移植而言，主要是添加**开发板初始化和驱动程序的代码**，这些代码大部分是跟体系结构相关。

具体到cortex-A8型开发板来说，linux已经有了较好的支持。比如从Kernel 官方维护网站 kernel.org 上下载到 2.6.35的源代码，解压后查看arch/arm/目录下已经包含了三星 S5PV210 的支持，即三星官方评估开发板 SMDK210的相关文件 mach-smdkv210 了。移植 Kernel 只需要修改两个开发板之间的差别之处就可以了。

7.6 Part Six

Gdb调试器

GDB（GNU Debugger）是自由软件基金会（**Free Software Foundation, FSF**）的软件工具之一。它的作用是协助程序员找到代码中的错误。

（1）进入GDB

Gdb test

test是要调试的程序，由gcc test.c -g -o test生成。进入后提示符变为(Gdb)。

（2）查看源码

(Gdb) l

源码会进行行号提示。

如果需要查看在其他文件中定义的函数，在l后加上函数名即可定位到这个函数的定义及查看附近的其他源码。或者：使用断点或单步运行，到某个函数处使用s进入这个函数。

(3) 设置断点

(Gdb) b 6

这样会在运行到源码第6行时停止，可以查看变量的值、堆栈情况等；这个行号是Gdb的行号。

(4) 查看断点处情况

(Gdb) info b

可以键入"info b"来查看断点处情况，可以设置多个断点；

(5) 运行代码

(Gdb) r

(6) 显示变量值

(Gdb) p n

在程序暂停时，键入"p 变量名"(print)即可；

GDB在显示变量值时都会在对值之前加上"\$N"标记，它是当前变量值的引用标记，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

(Gdb) watch n

在某一循环处，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，GDB在"n"设置了观察点；

（8）单步运行

(Gdb) n

（9）程序继续运行

(Gdb) c

使程序继续往下运行，直到再次遇到断点或程序结束；

（10）退出GDB

(Gdb) q

Gdb中，输入命令时，可以不用打全命令，只用打命令的前几个字符就可以了，当然，命令的前几个字符应该要标志着一个唯一的命令，在**Linux**下，可以敲击两次**TAB**键来补齐命令的全称，如果有重复的，那么**Gdb**会把其例出来。

7.7

Part Seven

远程调试

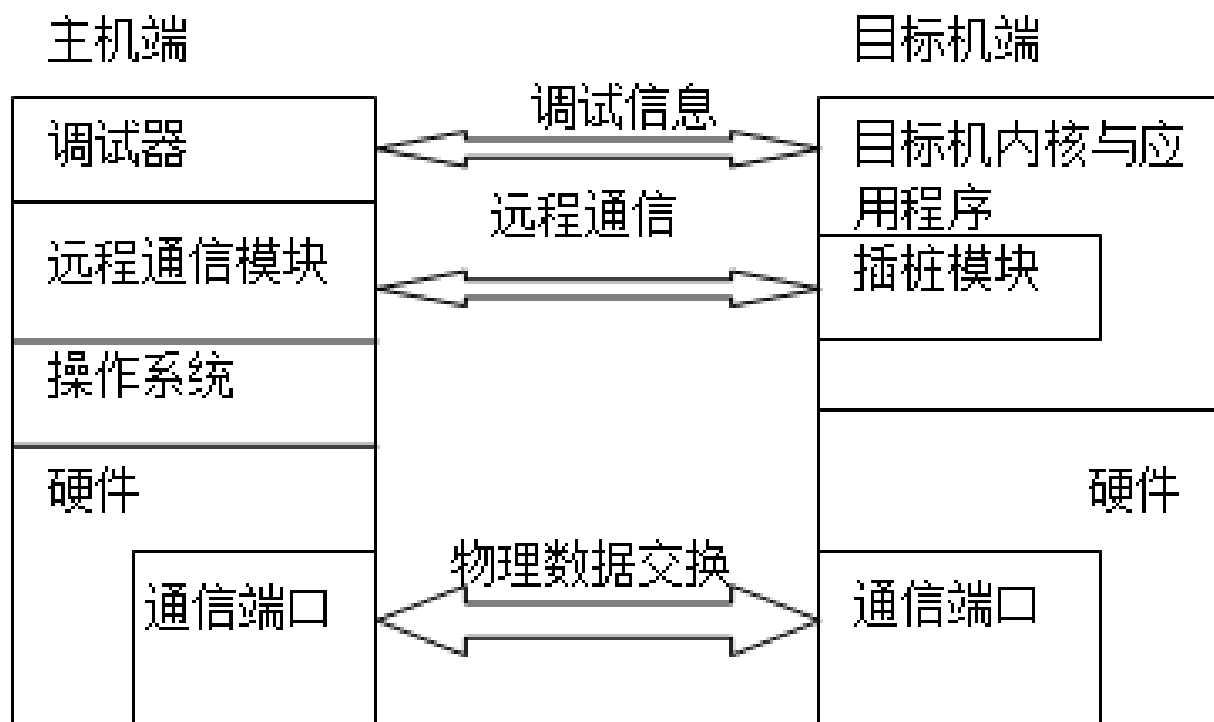
在嵌入式软件调试过程中，调试器通常运行于主机环境中，被调试的软件则运行于基于特定硬件平台的目标机上。主机上的调试器通过串口、并口或网卡接口等通信方式与目标机进行通信，控制目标机上程序的运行，实现对目标程序的调试，这种调试方式称为远程调试。

常用的远程调试技术主要有**插桩(stub)**和**片上调试**（On Chip Debugging, OCD）两种。

插桩(stub)指在目标操作系统和调试器内分别加入某些软件模块实现调试；

片上调试指在微处理器芯片内嵌入额外的控制电路实现对目标程序的调试。片上调试方式不占用目标平台的通信端口，但它依赖于硬件。插桩方式仅需要一个用于通信的端口，其他全部由软件实现。

7.7.1 远程调试工具的构成



插桩方式下的调试环境构成

远程调试工具由三部分构成：主机端的调试器、远程通信协议和插桩模块。前两部分可采用**GDB**（**GNU debugger**）调试器来解决。

为了监控和调试程序, 主机**GDB** 通过串行协议使用内存读写命令, 无损害地将目标程序原指令用一个**trap** 指令代替, 从而完成断点设置动作。

7.7.2 通信协议-RSP

$\$ < d a t a > \# [chksum]$

该数据包分为四部分：

第一部分是包头，由字符“\$”构成；

第二部分是数据包内容，对应调试信息，它可以是调试器发布的命令串，也可以是目标机的应答信息，数据包中应该至少有一个字节；

第三部分是字符“#”，它是调试信息的结束标志；

第四部分是由两位十六进制数的ASCII 码字符构成的校验码，

7.7.3 远程调试的实现方法及设置

相对于宿主机远程调试环境的建立过程，目标机调试**stub**的实现更要复杂，它要提供一系列实现与主机**GDB** 的通信和对被调试程序的控制功能的函数。这些功能函数**GDB**有的已经提供,如**Gdb**文件包中的**m68k-stub.c**、**i386-stub.c**等文件提供了一些相应目标平台的**stub**子函数，有的函数需要开发者根据特定目标平台自行设计实现。

stub的主要子函数：

sets_debug_traps()：函数指针初始化，捕捉调试中断进入handle.exception（）函数。

Handle_exception()：该函数是stub的核心部分。程序运行被中断时，首先发送一些主机的状态信息，如寄存器的值，然后在主机Gdb的控制下执行程序，并检索和发送Gdb需要的数据信息，直到主机Gdb要求程序继续运行，handle_exception（）交还控制权给程序。

Breakpoint（）：该功能函数可以被调试程序中设置断点。

handle_exception()

保存现场

接收并解析调试命令

设置
处理器
寄存
器

读寄
存器
的值

设置
指定
内存
地址

读取
指定
内存
地址

清理
调试
信息

设置
断点

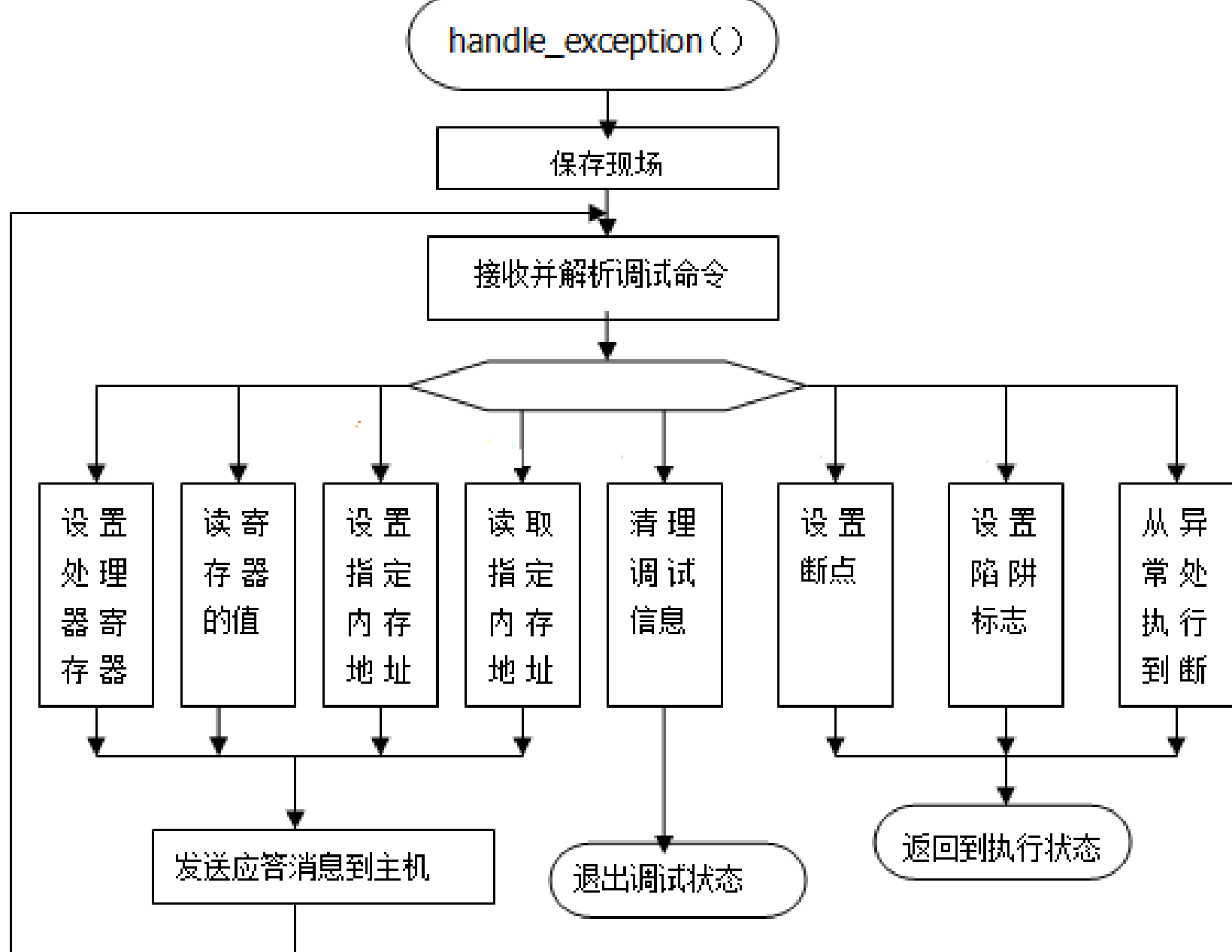
设置
陷阱
标志

从异
常处
执行
到断

发送应答消息到主机

退出调试状态

返回到执行状态



除以上函数外, 开发人员需要针对特定目标平台, 为 **stub** 实现以下底层功能函数, 才能使调试**stub**正常与主机**GDB** 协同工作, 比如:

GetDebugChar ()、**putDebugChar ()**: 读写通过 **Gdb**远程串行协议与主机交互的数据。

ExceptionHandler (): 各目标平台对系统中断向量的组织安排是不一样的, 该函数要能够使得系统中断发生时, 程序可以正常获得中断服务程序的入口地址。

Memset (): 标准库函数, 保证对特定目标平台的内存操作。

开发者就可以按以下步骤使用**stub**对目标程序进行远程调试：

（1）在被调试程序开始处，插入两个函数调用：**sets_debug_traps()**和**handle.exception()**。

（2）将被调试程序、**GDB**提供的**stub**功能函数和上述目标系统中实现的**stub**底层子函数一起编译链接生成一个包含调试**stub**的目标程序。

（3）建立主机与目标机的串口或以太口连接，保证通信物理链路的顺畅。

（4）将被调试目标代码下载到目标系统并运行该程序，它会被内部**stub** 函数中断在开始处，等待宿主机**GDB** 的连接请求。

（5）在宿主机运行针对目标平台编译链接**Gdb**。用**target remote**命令连接目标机**stub**，然后就使用相应**GDB** 命令对目标程序进行跟踪和调试了。

7.7.4 远程调试应用实例方法

目前嵌入式linux系统中，主要有三种远程调试方法，分别适合与不同场合的调试工作：**用ROM Monitor调试内核装载程序、用KGDB调试系统内核和用GDBserver调试用户空间程序**。这三种调试的方法的区别主要在于目标机远程调试stub的存在形式不同，而设计思路和实现方法则大致相同，并且它们配合工作的主机Gdb是同一个程序。

1. 用**ROM Monitor**调试目标机程序

在嵌入式linux内核运行前的状态中，程序的装载、运行和调试一般都由**Rom Monitor**实现。系统一旦加电后，包含了远程调试stub的**ROM Monitor**即可首先获得系统控制权，对**CPU**、内存、中断、串口、网络等重要硬件资源进行初始化并下载、运行和监控目标代码，内核的装载和引导也由**ROM Monitor**完成。

2. 用KGDB 调试系统内核

系统内核与硬件体系关系密切, 因而其调试stub的实现也会因具体目标平台的差异而存在一些不同, 嵌入式linux开发团队针对大多数流行的目标平台采用源码补丁形式对linux内核远程调试stub给予了实现及发布。用户只需正确编译链接打好补丁的内核, 就可对内核代码进行灵活的调试。

基于PC平台的linux 内核开发人员所熟知的KGDB就是这种实现形式, 该方法也同样用于嵌入式linux系统中

3. 用**GDBserver** 调试用户空间程序

在linux内核已经正常运行基础上，用户可以使用**Gdbserver**作为远程调试**stub**的实现工具。**GDBserver**是**GDB** 自带的、针对用户程序的远程调试**stub**，它具有良好的可移植性，可交叉编译到多种目标平台上运行。开发者可以在宿主机上用**Gdbserver**方便地监控目标机用户空间程序的运行过程。由于有操作系统的支持，它的实现要比一般的调试**stub**简单很多。

7.8 Part Eight

内核调试

7.8.1 printk()

printk类函数语法要点

| | | |
|-------|-----------------------------------|----------------------------------|
| 所需头文件 | #include <linux/kernel> | |
| 函数原型 | int printk(const char * fmt, ...) | |
| 函数传入值 | fmt: 日志级别 | KERN_EMERG: 紧急时间消息 |
| | | KERN_ALERT: 需要立即采取行动的情况 |
| | | KERN_CRIT: 临界状态，通常涉及严重的硬件或软件操作失败 |
| | | KERN_ERR: 错误报告 |
| | | KERN_WARNING: 对可能出现的问题提出警告 |
| | | KERN_NOTICE: 有必要进行提示的正常情况 |
| | | KERN_INFO: 提示性信息 |
| | KERN_DEBUG: 调试信息 | |
| | ...: 与printf()相同 | |
| 函数返回值 | 成功: 0 失败: -1 | |

7.8.2 KDB

Linux 内核调试器kdb（Linux kernel debugger）是Linux 内核的补丁，它提供了一种在系统能运行时对内核内存和数据结构进行检查的办法。Kdb是一个功能较强的工具，它允许进行多个重要操作，如内存和寄存器修改、应用断点和堆栈跟踪等。

7.8.3 Kprobes

Kprobes是一款功能强大的内核调试工具。它提供了一个可以强行进入任何内核的例程及从中断处理器无干扰地收集信息的接口。使用 **Kprobes** 可以轻松收集处理器寄存器和全局数据结构等调试信息，而无需对Linux内核频繁编译和启动。

从实现方法上来看，**Kprobes** 向运行的内核中给定地址写入断点指令并插入一个探测器，执行被探测的指令会产生断点错误，从而钩住（hook in）断点处理器并收集调试信息。**Kprobes** 吸引人的另一个重要之处在于它甚至可以单步执行被探测的指令。

7.8.4

KGDB



KGDB构成环境

安装Kgdb调试环境需要为linux内核应用Kgdb补丁。补丁实现的Gdb远程调试所需要的功能包括命令处理、陷阱处理及串口通讯3个主要部分。Kgdb补丁的主要作用是在Linux内核中添加了一个调试stub。

7.9

Part Nine

小结

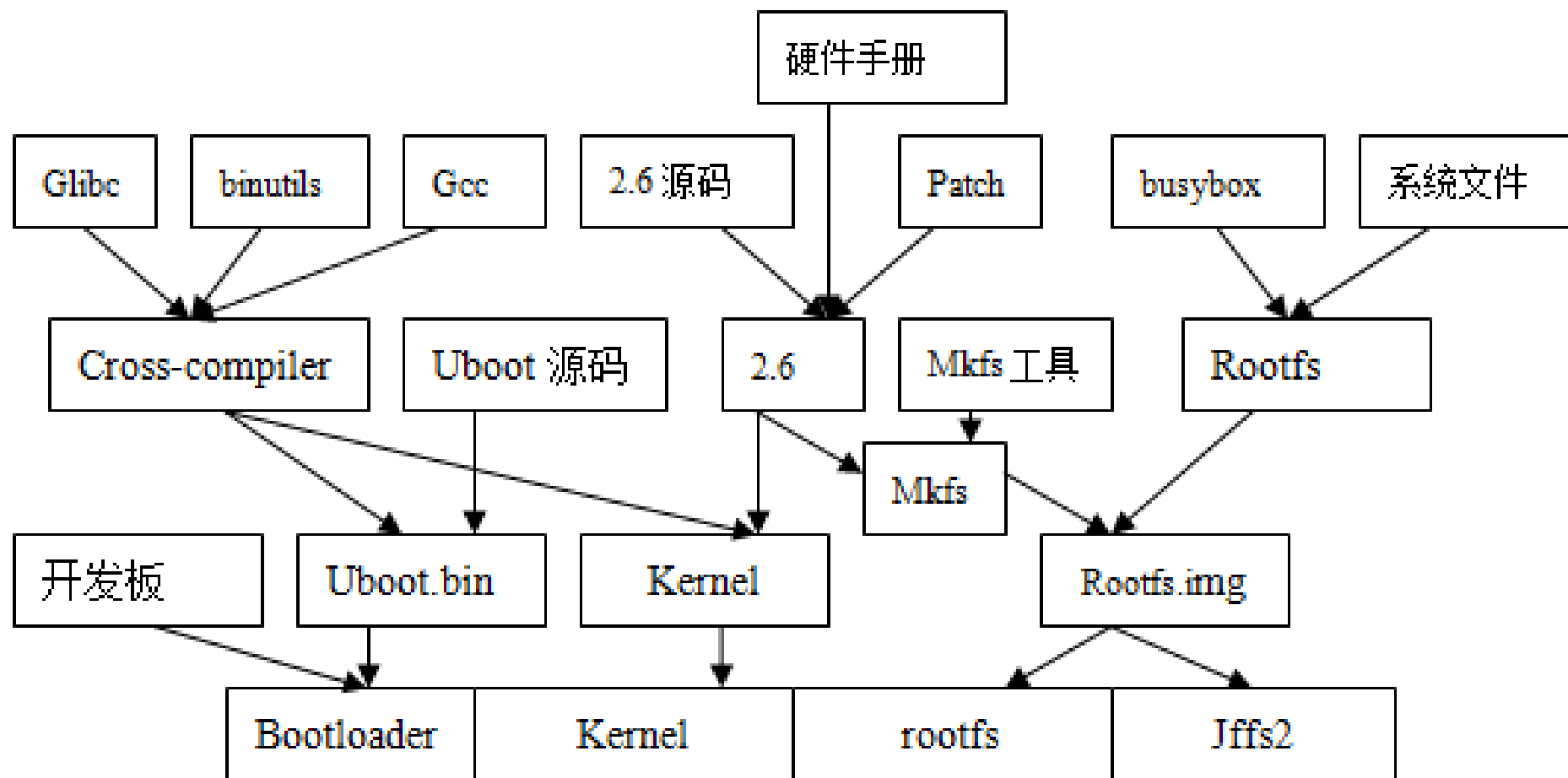


图7-14嵌入式Linux系统的基本组成和开发流程图