



嵌入式Linux启动优化技术

成都华清远见

杨晓鹏

版权

- } 华清远见嵌入式培训中心版权所有；
- } 未经华清远见明确许可，不能为任何目的以任何形式复制或传播此文档的任何部分；
- } 本文档包含的信息如有更改，恕不另行通知；
- } 保留所有权利。

内容提纲

本专题将给大家介绍在嵌入式linux系统中如何优化启动速度，加快系统启动时间。同时为大家介绍一种制作启动进度条的原理和具体实现。

- } 传统嵌入式linux产品的启动流程介绍
- } 启动过程时间耗费分析
- } 启动过程优化技术介绍
- } 启动进度条实现原理分析
- } 制作启动进度条具体实现介绍

传统嵌入式linux产品的启动流程介绍

} 通用嵌入式系统硬件一般由以下部分组成

} 微控制器

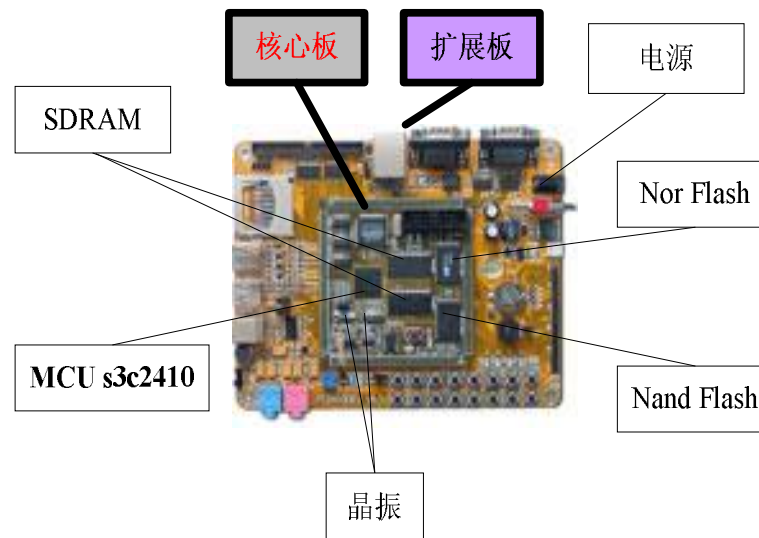
} 晶振

} 内存（如：SRAM，SDRAM）

} 存储器（如：ROM，FLASH，SD，微硬盘）

} 其他外围设备接口

} 输入、输出接口



传统嵌入式linux产品的启动流程介绍

} 通用嵌入式系统软件组成部分

图一为无os嵌入式系统组成图

图二为有os嵌入式系统组成图

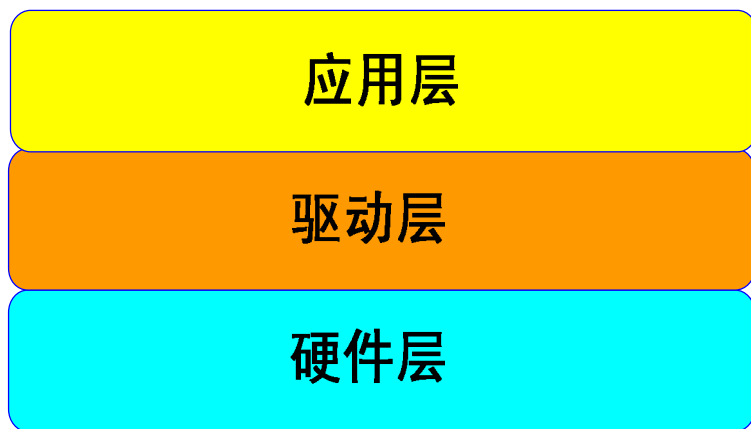


图 一

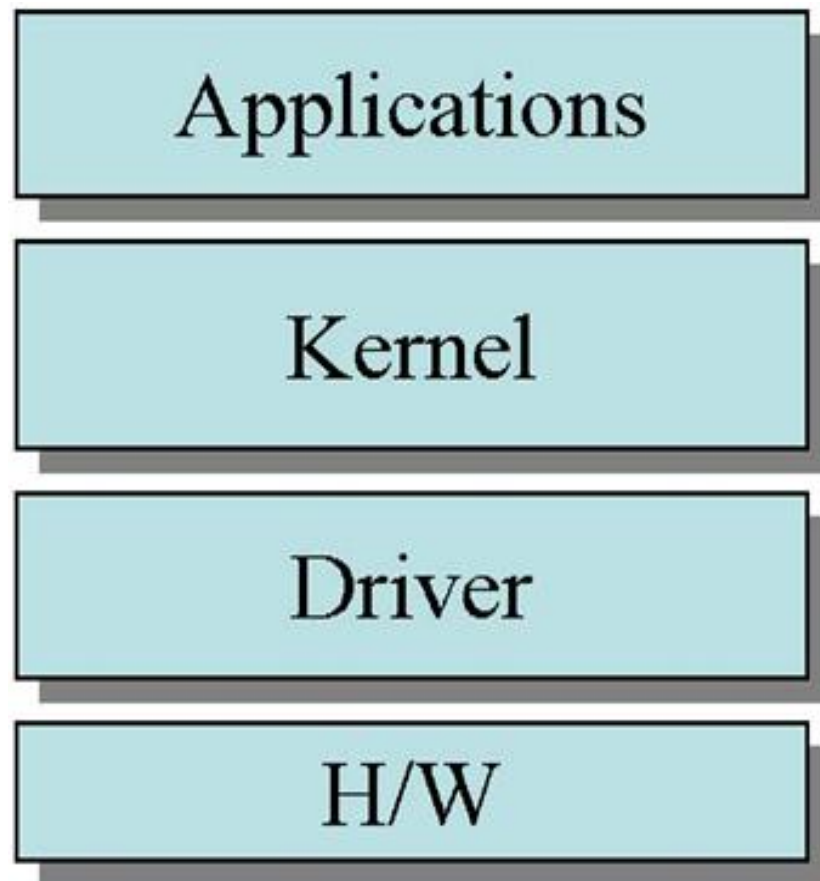


图 二

传统嵌入式linux产品的启动流程介绍

- } 传统嵌入式linux产品的启动流程
 - } 系统加电，运行Bootloader。
 - } 由Bootloader负责加载Kernel或文件系统。
 - } 运行Kernel，挂载文件系统。
 - } 进入文件系统，配置运行环境和自动启动应用程序。

传统嵌入式linux产品的启动流程介绍

} 几个概念的说明

- } Bootloader是硬件启动的引导程序，是启动操作系统的根本。Bootloader可与linux没有关系，我们可在linux发行版本里开发Bootloader，也可不用linux环境而开发支持linux系统的Bootloader。
- } 这里的Kernel指的是linux Kernel，我们说的嵌入式linux产品，其实准确的说应该是安装了linux Kernel，应用软件运行在linux Kernel上的系统。
- } 文件系统。这里指的是linux Kernel支持的文件系统。产品里常见的文件系统主要有ramdisk方式下使用的ext2，只读的cramfs，nand或nor flash下用的yaffs2和jffs2文件系统。

启动过程时间耗费分析

- } 真实的嵌入式linux产品有时很看重系统启动时间，启动时间越短越好。有些产品启动时间超过40秒，有些可以做到10秒以下。那些启动时间超短的系统，他们是怎样做到的？

根据上面介绍的嵌入式linux产品启动流程可以看出，影响系统启动时间的主要会有这几个地方：

- } 运行Bootloader所耗费的时间；
- } Bootloader加载Kernel或文件系统所耗费的时间；
- } Kernel运行和挂载文件系统所耗费的时间；
- } 文件系统挂上后，配置运行环境和自动启动应用程序所耗费的时间。

启动过程时间耗费分析

- } 我们通过实际的实验平台和测量数据来比较不同启动方式所耗费的启动时间。
- | 实验平台：
 - 硬件环境：华清远见的FS2410开发板，s3c2410 Cpu，2MB Nor Flash，64MB Nand Flash，64 MB SDRAM
 - 软件环境：移植好的u-boot-1.1.3 + linux-2.6.22.6 和交叉编译工具 arm-linux-gcc 3.3.2 /3.4.1。
 - 交叉编译开发环境：VMware Workstation 6.5 + Ubuntu8.10
- | Flash规划：
 - 实验采用nor flash引导，nor flash里装Bootloader，nand flash储存Kernel和文件系统。nor flash和nand flash的内容如下所示：

启动过程时间耗费分析

} Nor flash: 0-256 KB放u-boot, 256KB – 512KB作为参数区, 512KB-2MB处没有使用。

Nand flash: 0-16MB 存放kernel和ramdisk压缩文件, 其中2MB-6MB存放linux kernel, 6MB-16MB存放ramdisk.gz。16MB – 32MB作为cramfs文件系统, 32-64MB作为jffs2文件系统。

} 首先在u-boot下测试一下命令

```
nand read 30008000 200000 400000 ; go 30008000
```

这个命令的意思是将nand flash里2MB处的内容拷贝4MB到内存0x30008000处, 再跳到0x30008000处开始执行。正常情况下这个命令执行完成后我们应该看到著名的

```
Uncompressing Linux.....  
done, booting the kernel
```

启动过程时间耗费分析

} 从敲回车到看到“done, booting the kernel”，大约用了3.5秒，这里我们可以进一步缩短时间吗？

我们分析，内核解压时即从看到Uncompressing Linux到看到done, booting the kernel，用了近2.5秒的时间。

我们可以不使用zImage，可以用Image来替换zImage。它们的区别仅仅是zImage是Image压缩后的产物。

替换后，我们从上面测试出的3.5秒减少到1秒，节省了2.5秒的启动时间。

启动过程时间耗费分析

} 文件系统加载。

我们测试3种文件系统的加载时间，为了有可比性，我们必须保证3种文件系统的内容要完全一样。我们在切换使用不同文件方式时，需要相应的内核参数。

1 使用ramdisk方式：在u-boot下运行命令

```
setenv bootargs root=/dev/ram init=/linuxrc load_ramdisk=1
```

```
initrd=0x30800000 ramdisk_size=28672 console=ttySAC0,115200 mem=65536K
```

```
nand read 30008000 200000 400000 ;
```

```
nand read 30800000 600000 800000 ; go 30008000
```

启动完成共用约25秒，其中主要费时间的部分为：

从nand flash拷贝Image和ramdisk.gz 4秒，

停在 RAMDISK: Compressed image found at block 0用了14秒

启动过程时间耗费分析

使用jffs2文件系统启动方式：在u-boot下运行命令
setenv bootargs root=1f02 noinitrd rootfstype=jffs2 rw
console=ttySAC0,115200 mem=65536K

nand read 30008000 200000 400000 ; go 30008000

启动完成共用约15秒，其中主要费时间的部分为：

从nand flash拷贝Image只用时1秒，

停在文件系统加载那里，大约7秒。从这个比较可以看出，比ramdisk节省启动时间，主要是nand flash拷贝节省了3秒和文件系统加载节省了约7秒。

启动过程时间耗费分析

使用cramfs文件系统启动方式：在u-boot下运行命令
setenv bootargs root=1f01 noinitrd rootfstype=cramfs ro
console=ttySAC0,115200 mem=65536K init=/linuxrc

nand read 30008000 200000 400000 ; go 30008000

启动完成共用约11秒，其中主要费时间的部分为：

从nand flash拷贝Image只用时1秒，

停在文件系统加载那里，大约3秒。从这个比较可以看出，比jffs2节省启动时间，文件系统加载节省了4秒。

启动过程时间耗费分析

实验结论：

内核采用Image非压缩内核可以节省kernel运行时间；文件系统采用cramfs可以节省Bootloader拷贝flash时间和kernel加载文件系统时间。

需要注意的问题：采用非压缩内核和压缩内核的选择要根据flash的空间来判断。cramfs文件系统只读，实际产品中可能还会用到yaffs，jffs2和ramfs等可读写的文件系统。

启动过程时间耗费分析

} 还可以进一步减少系统的启动时间吗？

答案是肯定的。可以考虑：

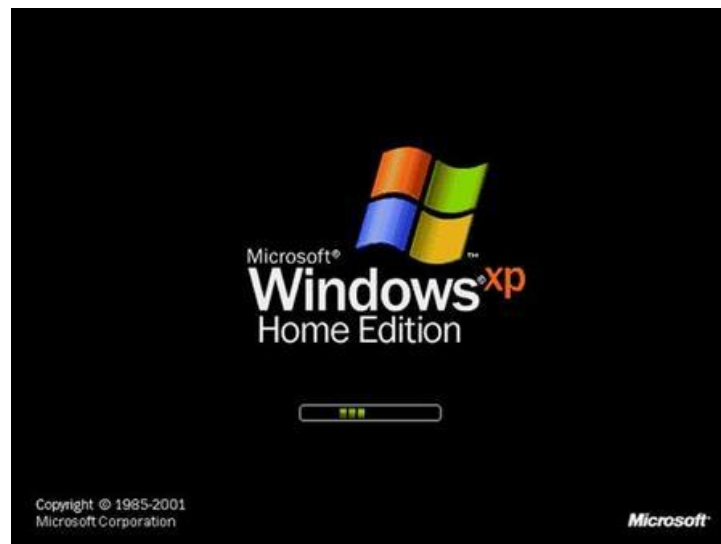
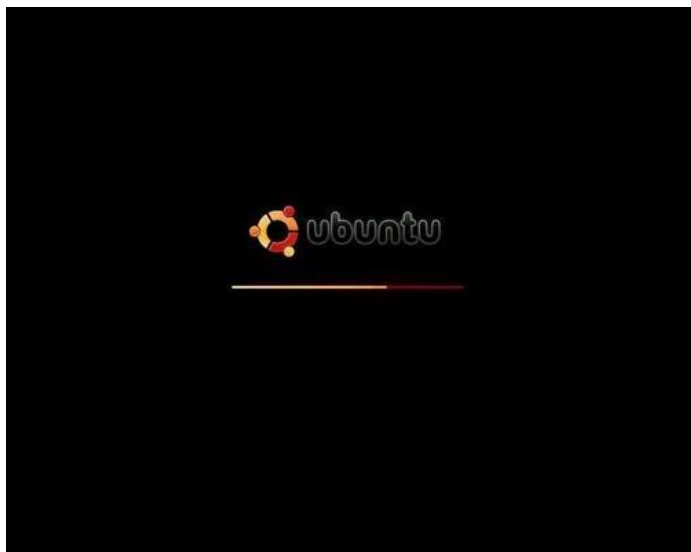
- | 优化u-boot，减少u-boot运行时间；
- | 减少kernel的体积，将不是在启动阶段必须加载的内容驱动模块化，留在文件系统后面加载。减少kernel运行时间；
- | 优化或者关闭调试串口的输出，至少可以节省1秒的启动时间；
- | 驱动程序放到文件系统后面加载，先显示应用界面，再在后台加载必要的内容驱动模块；
- | 将整个系统运行环境保存到RAM中，加电只是一个系统恢复过程；
- | 仔细优化kernel和各种驱动程序，将每个部分的延时操作调试到最快；
- | 应用程序环境的优化，减少应用程序启动时间。
- |

那么我们该选择哪种优化方式？

答案在于我们是否还要去关注最快的系统启动方式，有时我们的产品启动时间低于某个值就已经达到设计要求，追求最短的系统启动时间，是一个很费资源的工作，完全没有必要。

启动过程优化技术介绍

- } 系统启动过程优化，除了启动时间优化外，我们还要考虑启动界面的优化。启动界面目前常见的有：
- 1) 开机显示静态的logo，再进入操作界面；
 - 2) 开机显示动态的logo，再进入操作界面；
 - 3) 开机显示静态的logo，启动过程中显示动态的加载进度条。



启动过程优化技术介绍

有些设备开机时除了可以显示图片，还可以播放动画和播放音乐。我们今天主要介绍常见的第3种启动界面技术的基本原理，并通过一个实例来说明其应用方式。

启动进度条实现原理分析

- } Logo和进度条显示用到的知识点主要是Framebuffer即帧缓存。
 - } 可以把帧缓存看成是一块内存区，这个内存区可以叫做显存。往这个内存区写进的任何数据，都将被LCD控制器送往LCD显示出来。
 - } 我们的实验FS2410实验平台使用的LCD显示采用320 x 240的16位模式，数据格式是565 RGB模式。那么显示完整的1帧即1屏图像，将会使用 $320 \times 240 \times 2 = 153600$ 字节内存区。每个像素点用2个字节表示。
 - } Logo和进度条显示的核心就是操作这个帧缓存内存区。

启动进度条实现原理分析

- } 嵌入式linux系统启动时先要启动Bootloader，再启动linux kernel，所以我们一般先要在Bootloader里显示logo，在启动kernel时再显示进度条。
- } 直接在Bootloader里初始化CPU的LCD控制器，把Logo的图像数据写进显存即可显示静态的logo。
- } Kernel启动后一般会重新初始化CPU的LCD控制器，然后在LCD的左上角显示小企鹅。这时我们要去掉小企鹅显示，还需要重新把Logo的图像数据写进显存显示出静态的logo。
- } 进度条其实就是一个小区域的图像数据。将进度条移动起来，我们可以采用内核定时器技术，让内核每隔一段时间，刷新一下进度条显示区域里的图像数据，每次向前移动一点距离，这样看起来进度条图像就移动起来了。

制作启动进度条具体实现介绍

} Bootloader里的静态Logo图像

将Logo图像制作成二进制数组，写进bootloader里。初始化完lcd控制器后，将二进制数组写到显存里即可。在本次试验里，我们计划显示右面的logo，它是一幅180x60的图像，那么我们的在bootloader里显示logo的代码看起来像如下这样，这里的0x32540000是帧缓存的起始地址：

```
{
unsigned char *dest;
int pixel = 0,i;
int x,y,
dest = (unsigned char *) 0x32540000;
memset(dest, 0, 320 * 240 * 2);
dest += (320 * 50 + 70)* 2;
for (y = 0; y < 60; y++) {
    for (x = 0; x < 180; x++) {
        *(dest++) =logoimage[pixel++];
        *(dest++) = logoimage [pixel++];
        dest += (140) * 2;
    }
}
}
```



制作启动进度条具体实现介绍

} Kernel里的处理

} 首先用make menuconfig 将启动时显示小企鹅图标功能去掉，不选择如下项即可：

device drivers->

 Graphics support->

 [] Boot logo->

 然后make，重新生成内核文件。

} 驱动程序里的处理

} 由于我们的内核已经打开了帧缓存的支持，而且是在内核启动时很早加载的驱动，而且会重新初始化LCD控制器并清除帧缓存，所以我们要重新画logo，并且加入进度条显示功能。主要修改的文件是drivers/video/s3c2410fb.c。具体做的工作是在LCD控制器并清除帧缓存完成后，重画logo，画进度条，初始化内核定时器。在定时器响应函数里移动进度条。

制作启动进度条具体实现介绍

- } 在s3c2410fb.c里的s3c2410fb_probe最后return 0前面加上一个内核定时器初始化函数timer_init(), 这个函数主要做的工作是重新显示logo, 显示进度条, 初始化内核定时, 其代码看起来像这样:

```
struct timer_list timer;  
init_timer(&timer);  
timer.data = 5;  
timer.expires = jiffies + 100;  
timer.function = timer_function;  
add_timer(&timer);
```

上面代码的意思是建立了一个内核定时器, 每隔100x10毫秒也就是1秒以后执行一次timer_function这个函数。这里的时间间隔与内核的HZ数有关, 一般来说基于ARM的linux内核这个值是1/100秒, 也就是10毫秒。

制作启动进度条具体实现介绍

} timer_function里的代码主要的工作是更新进度条的显示，使之看上去移动起来，然后判断是否关闭定时器，定时器操作代码看起来像这样的：

.....

```
if(flag == 2)
```

```
    del_timer(&timer); //删除定时器，停止进度条
```

```
else
```

```
    mod_timer(&timer, jiffies + 5); //重新设置定时器
```

.....

这里的mod_timer(&timer, jiffies + 5)意思是继续激活定时器，隔5x10毫秒后再次执行timer_function。这样进度条就运动起来了，看起来就像动画一样。所以进度条显示的核心技术就是内核定时器技术。至于怎样画进度条以及显示，其实就是在不同位置画同样图像，定时更新图像即可达到动画效果。

制作启动进度条具体实现介绍

} 我们来看一段动画效果演示。

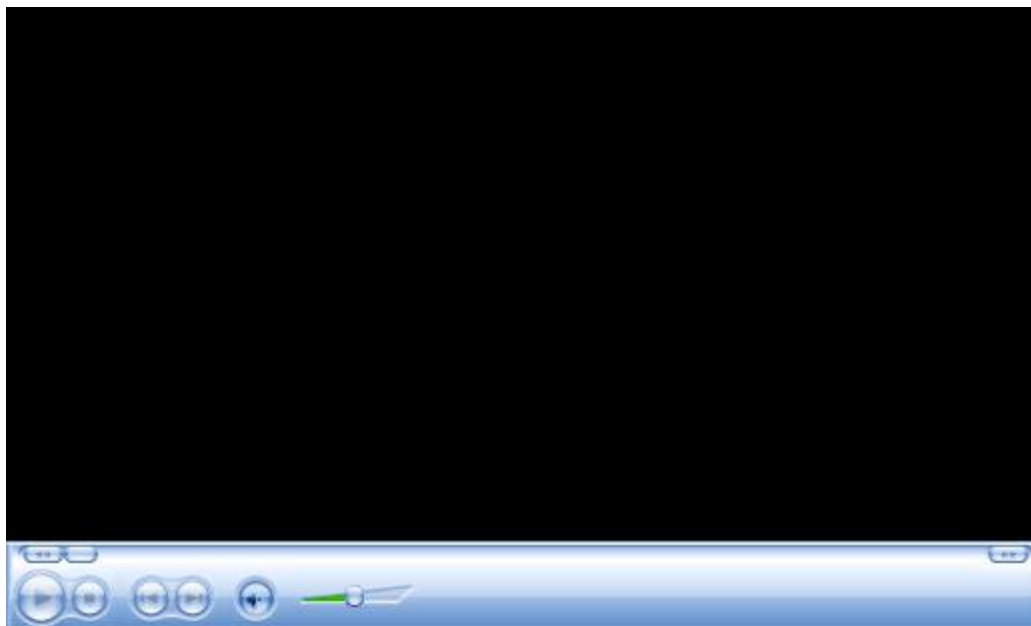
这里的用硬件和软件系统就是前面我们介绍的，安装的软件如下：

Bootloader: u-boot-1.1.3

Kernel: linux-2.6.22.6 for FS2410

File system: nand flash + 32MB jffs2

Application: minigui-1.6.10 demo



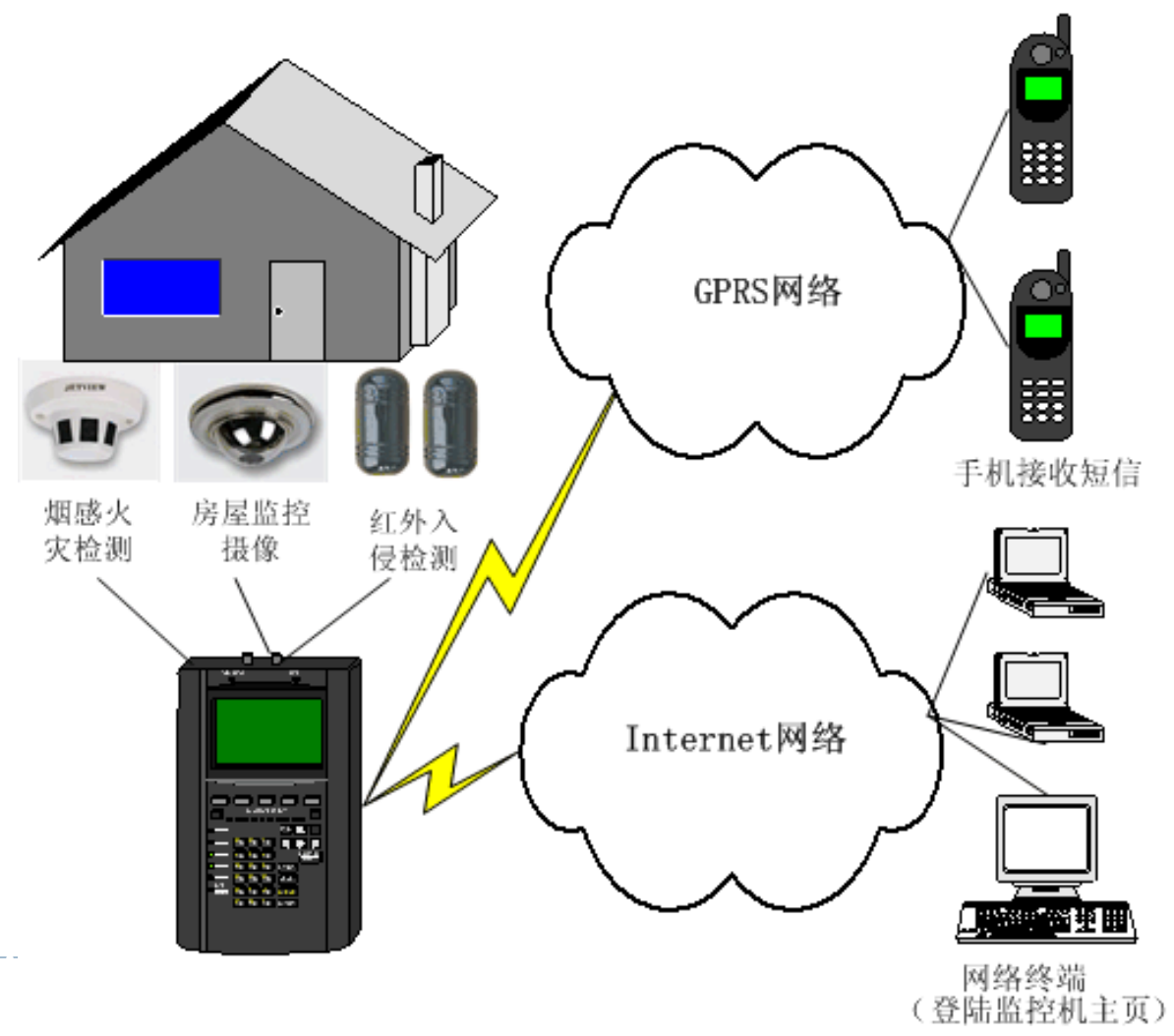
华清远见嵌入式培训课程介绍

- } 嵌入式linux系统开发班
- } 嵌入式linux驱动开发班
- } ARM工程师开发班
- } 嵌入式linux就业班
- } Andriod系统开发班
- } WinCE系统定制与驱动开发班
- } Windows Modbiles开发班
- } Symbian系统开发班

案例介绍 (一)

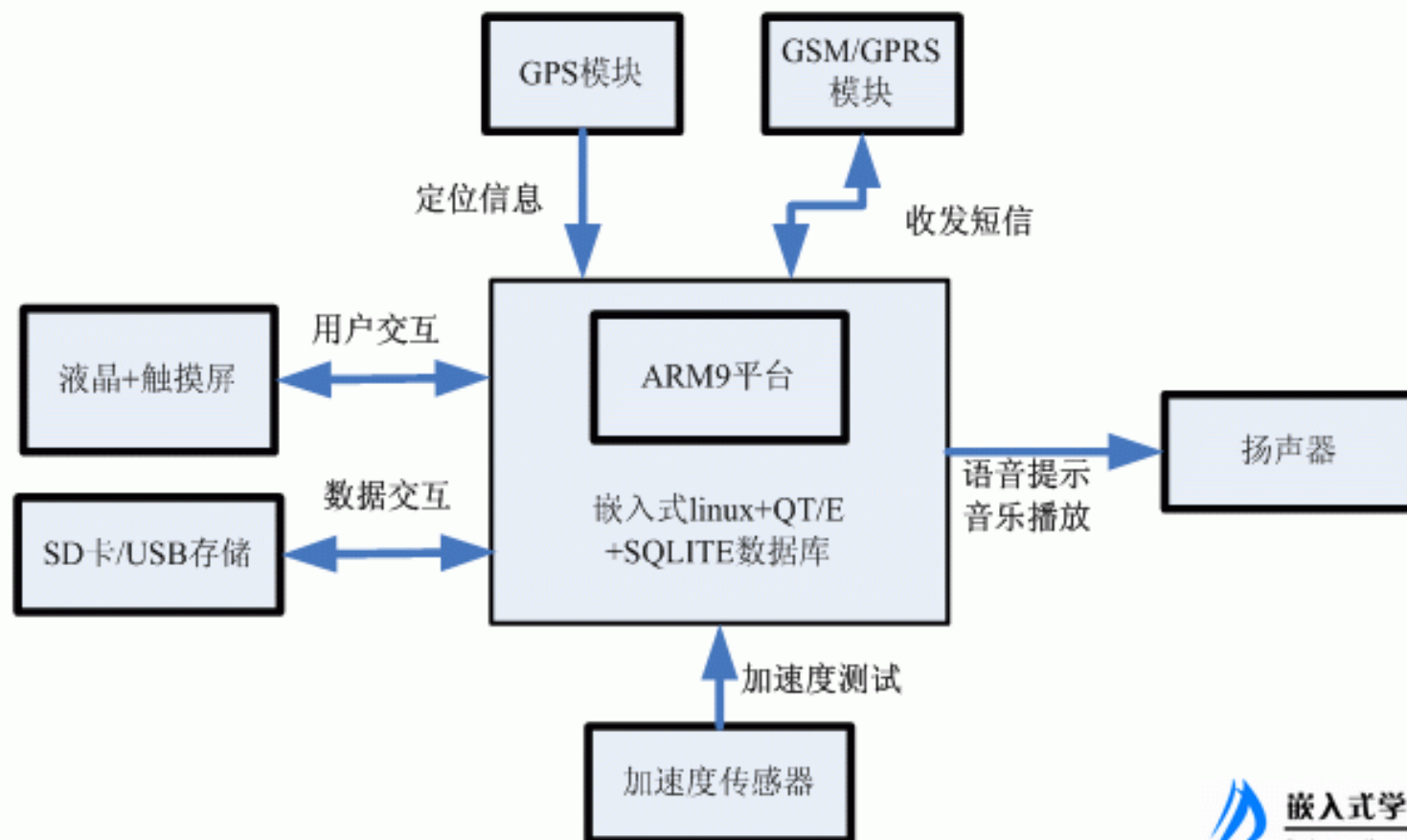
基于GPRS 的远程安防 监控系统

基于GPRS的远程安防监控系统
(华清远见嵌入式学院学员实践项目)



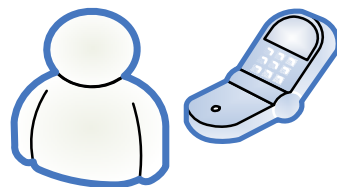
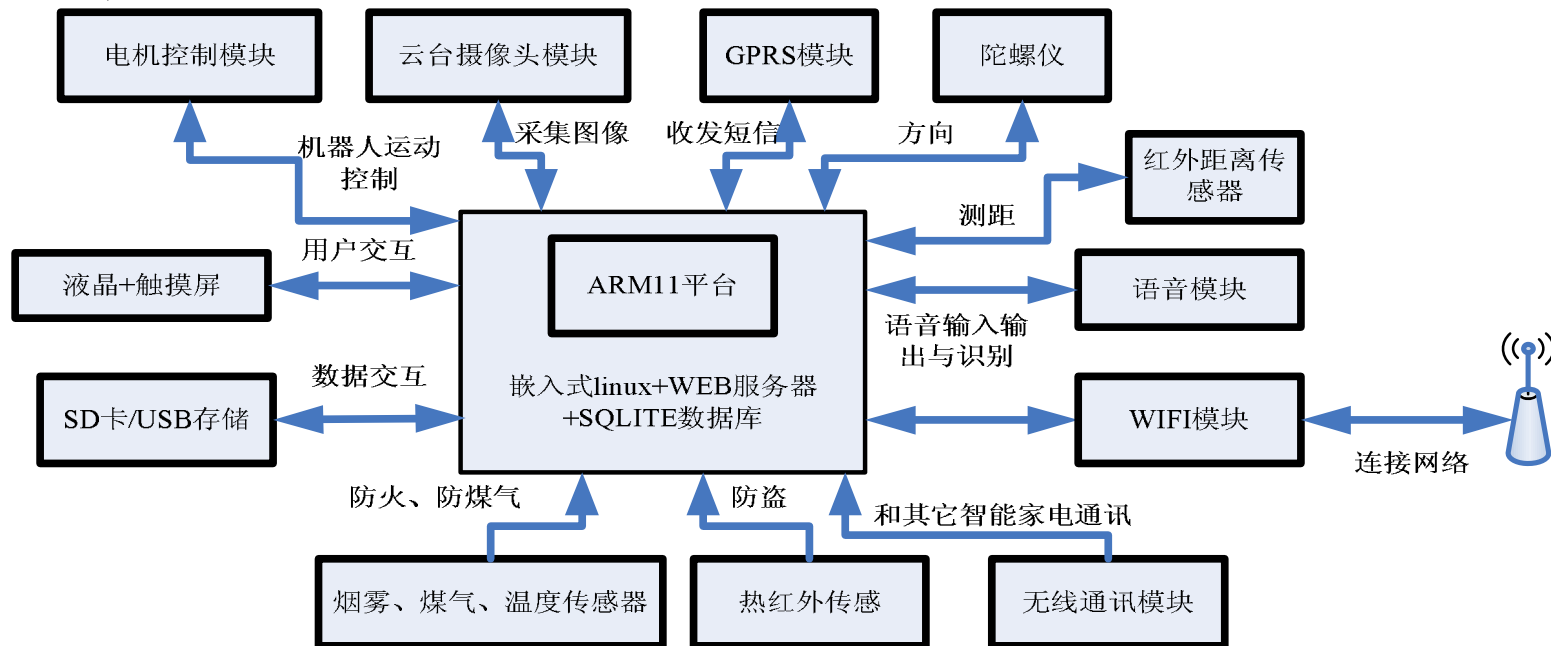
案例介绍（二）

} 车载导航系统



案例介绍 (三)

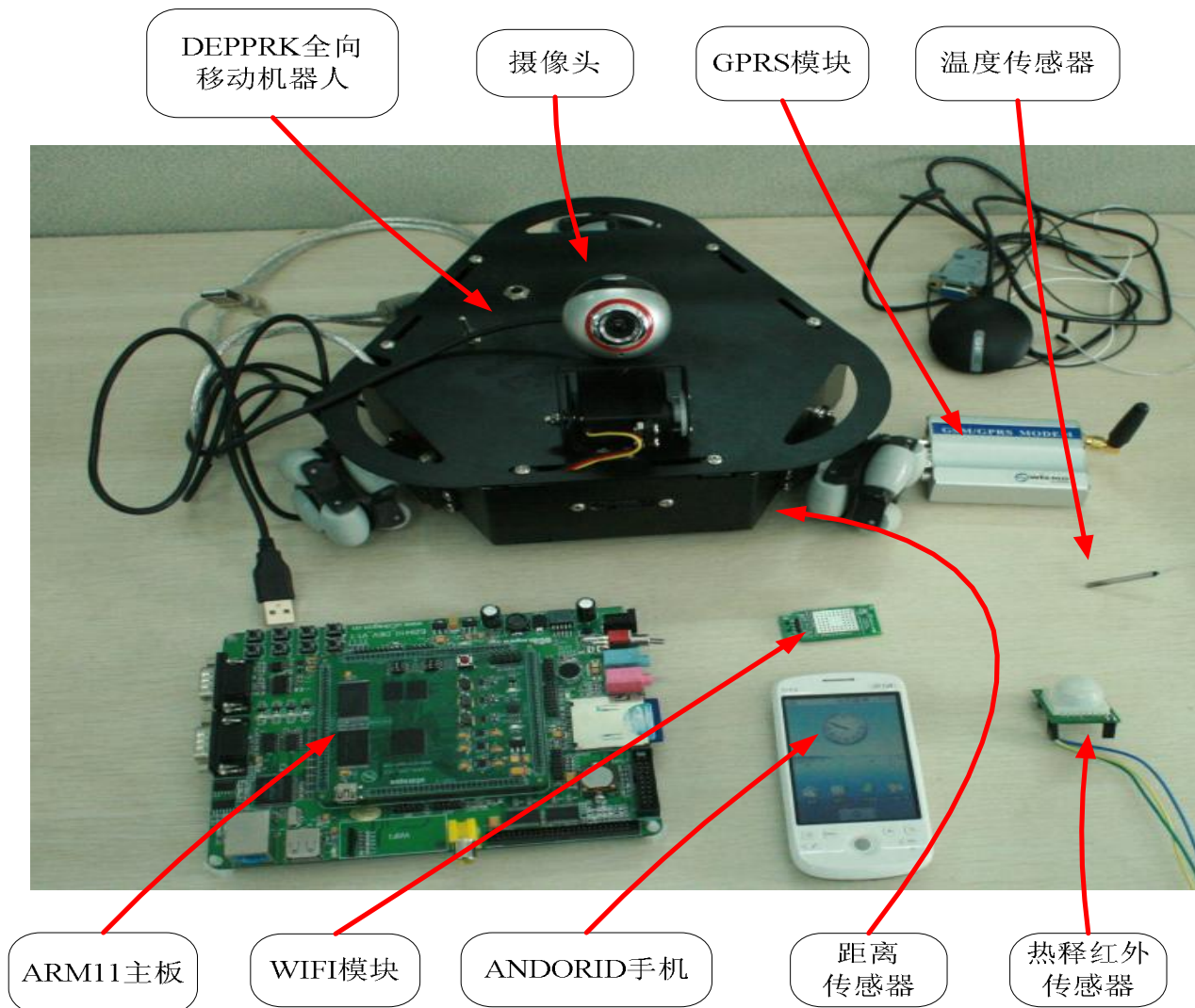
3G机器人



基于ANDROID的3G手机
 (项目采用S3C6410平台搭建)

案例介绍

} 实物图片



} 鸣谢:

感谢北京华清总部和成都分公司的各位老师 and 同事的大力协助。
十分期望今后与大家能够再次相聚。

谢谢!



Q&A

