H-JTAG

北京中科凌创电子

S3C2440 NAND FLASH 烧写

文档版本　A

发布日期　2013-01-10

WWW.HJTAG.COM

S3C2440 NAND FLASH 烧写

修改记录

日期	版本	改动
2013-01-10	A	发布第一版本

版权声明

官方主页

www.hjtag.com

联系电话

010-58409379

技术支持

forum.hjtag.com
qq: 2413170606
support@hjtag.com / twentyone@hjtag.com

销售咨询

qq: 2415313652
sales@hjtag.com

S3C2440 NAND FLASH 烧写

文档详细介绍了如何使用 H-JTAG/H-FLASHER 对 S2C2440+NAND FLASH 进行烧写. 文中不但介绍了基本的 NAND FLASH 操作, 也包括了用户很关心的一键烧写嵌入式系统, NAND FLASH 驱动程序定制修改等内容. 具体内容如下:

用户可以根据自身需求有选择的进行阅读. 如果用户对 NAND FLASH 不是很了解, 我们建议用户先看一下附录中关于 NAND FLASH 的基本介绍.

1. H-JTAG 介绍

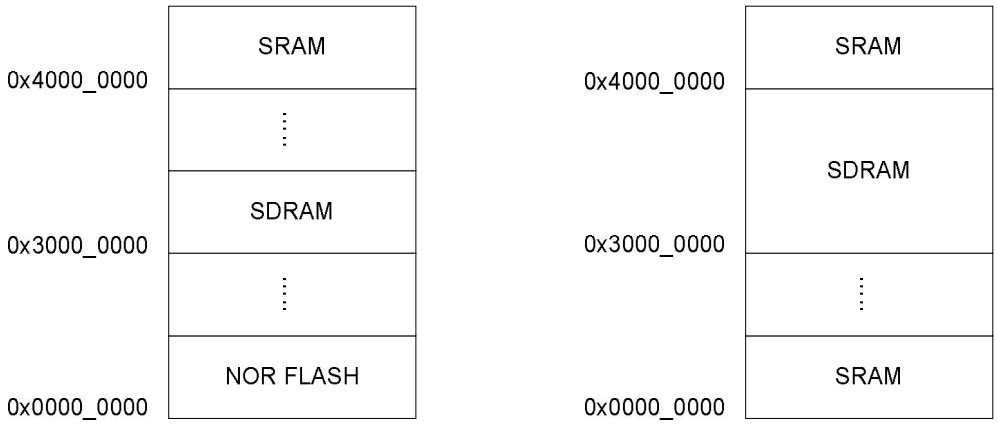
H-JTAG 是由北京中科凌创电子科技有限公司开发的一款具有完全知识产权的 ARM 仿真/烧写工具. 关于工具的介绍, 请访问我们的官方网站: WWW.HJTAG.COM. 关于 H-JTAG/H-FLASHER 使用的详细介绍, 用户也可以参考 H-JJTAG 的用户手册.

针对 NAND FLASH 本身的特点, H-JTAG/H-FLASHER 提供了多种灵活的 NAND FLASH 烧写方式. 本文将重点详细介绍如何使用 H-JTAG/H-FLASHER 对 S3C2440+NAND FLASH 进行烧写.

2. S3C2440 介绍

S3C2440 是三星的一款基于 ARM920T 的高性能 32 位嵌入式处理器. 处理器的最高主频 400MHZ, 内置 4K SRAM, 存储控制器支持外部扩展 SDRAM 和 NOR FLASH, NAND FLASH 控制器支持外接 8/16 位 512/2048 字节页面大小的 NAND. S3C2440, 并可以通过跳线选择从 NOR FLASH 或 NAND FLASH 启动.

上电后, 默认情况 (未开启 MMU) 下的存储分配和启动选择有关系. 如果选择从 NOR FLASH 启动, 地址 0x0 开始的空间是 NOR FLASH, 地址 0x30000000 开始的空间是 SDRAM, 地址 0x40000000 开始的 4K 空间是片内 SRAM. 如果选择从 NAND FLASH 启动, NOR FLASH 未分配地址, 地址 0x0 开始的 4K 空间是片内 SRAM, 地址 0x30000000 开始的空间是外部 SDRAM, 地址 0x40000000 开始的 4K 空间是片内 SRAM. 烧写 NAND FLASH 时, 可以选择使用片内 SRAM 或是外部 SDRAM. S3C2440 的具体存储分配请参考下图.



(1) NOR FLASH 启动 **(2) NAND FLASH 启动**

图 1: S3C2440 默认存储分配示意图

如果开启了 MMU, 具体的存储分配由 MMU 的配置决定. 如果 FLASH 内部已经烧写了 LINUX 或 WINCE, 上电后, 操作系统会配置 MMU 并重新分配存储. 在这种情况下, 通过 H-JTAG 进行烧写的时候, 可以通过添加 Software Reset 脚本命令来关闭 MMU, 以恢复默认的存储分配.

3. 使用 H-JTAG 烧写 S3C2440+NAND FLASH

针对 NAND FLASH 本身的特点，H-JTAG/H-FLASHER 提供了多种灵活的 NAND FLASH 烧写方式. 用户可以根据实际需求，选择不同的坏块管理方式，指定不同的烧写文件格式. 下面将简单介绍如何使用 H-JTAG 对 S3C2440+NAND FLASH 进行烧写.

3.1 准备工作

将 H-JTAG USB 仿真器通过 USB 线和电脑连接好. 然后通过 JTAG 排线将开发板和 H-JTAG USB 仿真器连接好. 连接好后，将开发板上电. 此时，H-JTAG USB 仿真器上的 USB 和 TGT 两个灯应该都是亮的. 接下来运行 H-JTAG 软件. 启动后，H-JTAG 会自动做 JTAG 检测. 正常情况下，H-JTAG 会检测到 S3C2440，并在 H-JTAG 主界面上显示其 JTAG ID 和 ARM 内核，如下图所示.

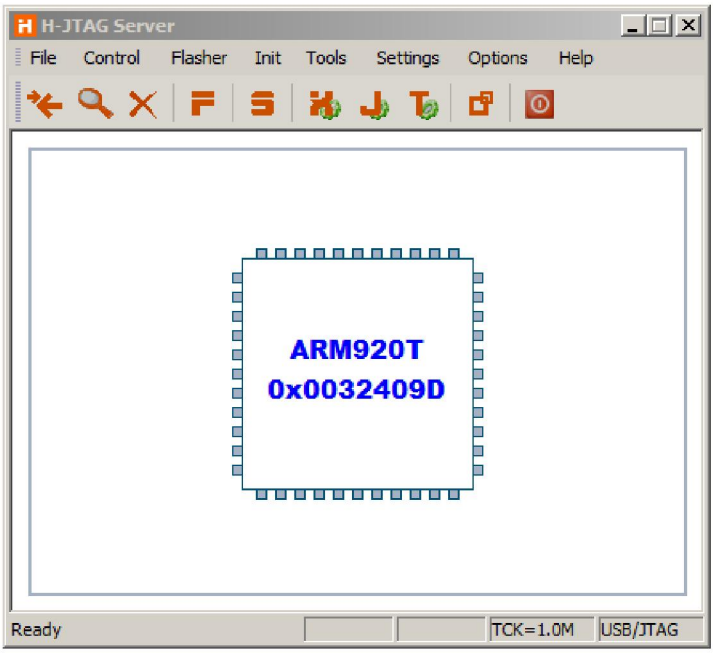


图 2：S3C2440 检测

3.2 运行 H-FLASHER 并装载 HFC 配置文件

通过 H-JTAG 检测到 S3C2440 后，需要运行 H-JTAG 的 FLASH 烧写软件 H-FLASHER，并针对目标系统做些基本的配置.

H-FLASHER 的主界面如下图所示. 主界面的顶部是菜单，左侧是编程向导，右侧是相应的操作页面. 用户可以通过编程向导选择进入到不同的操作页面.

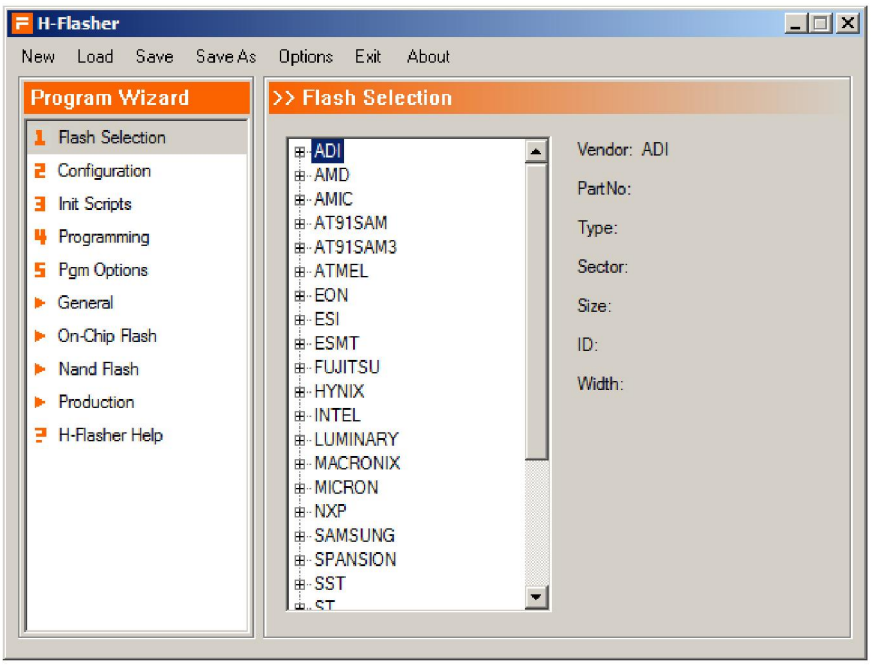


图 3: H-FLASHER 主界面

H-FLASHER 的基本配置包括选择 FLASH 型号，指定 RAM 地址，添加 S3C2440 初始化脚本等. 在这里，用户直接使用 H-JTAG 提供的 S3C2440 的配置文件. 在 H-FLASHER 的主界面上，点击 Load 菜单，然后选择 H-JTAG 安装目录下，“HFC Examples” 文件夹里面的 S3C2440+K9F1208.hfc 文件. 这个配置文件中选择的 FLASH 型号是 S3C2440+K9F1208. 如果用户开发板上的 NAND FLASH 型号不是 K9F1208，需要在 H-FLASHER 中重新选择 FLASH 型号. 选择好后，点击 Save 菜单直接保存配置文件，或是点击 Save As 将配置文件另存为其它 HFC 文件. 这样，烧写配置就完成了.

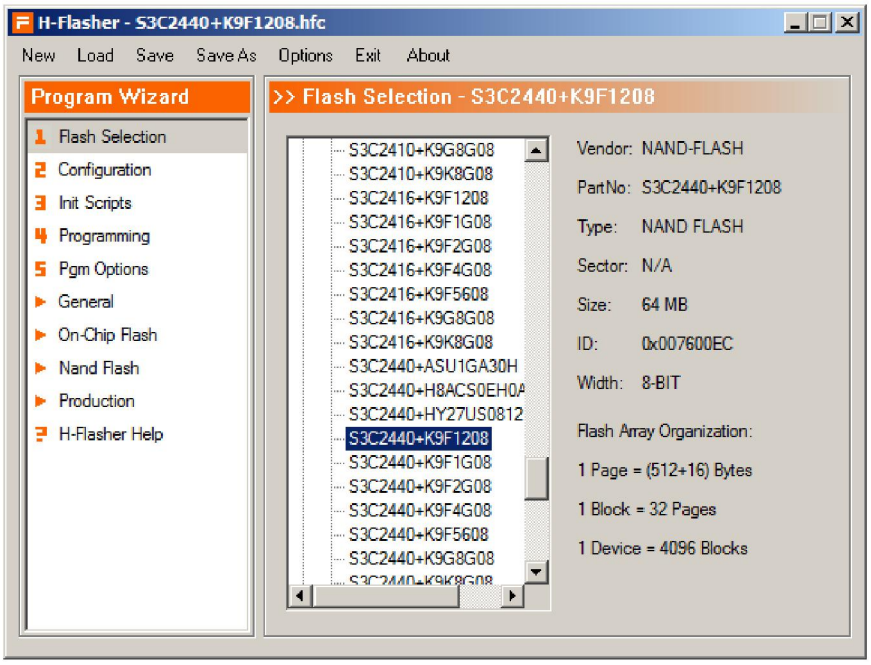


图 4: H-FLASHER 配置

3.3 通过 H-FLASHER 对 FLASH 进行基本操作

配置完成后，就可以通过 H-FLASHER 对 NAND FLASH 进行操作了．在 H-FLASHER 的主界面左侧的编程向导中点击”4 Programming”，就可以进入编程页面，如下图所示.

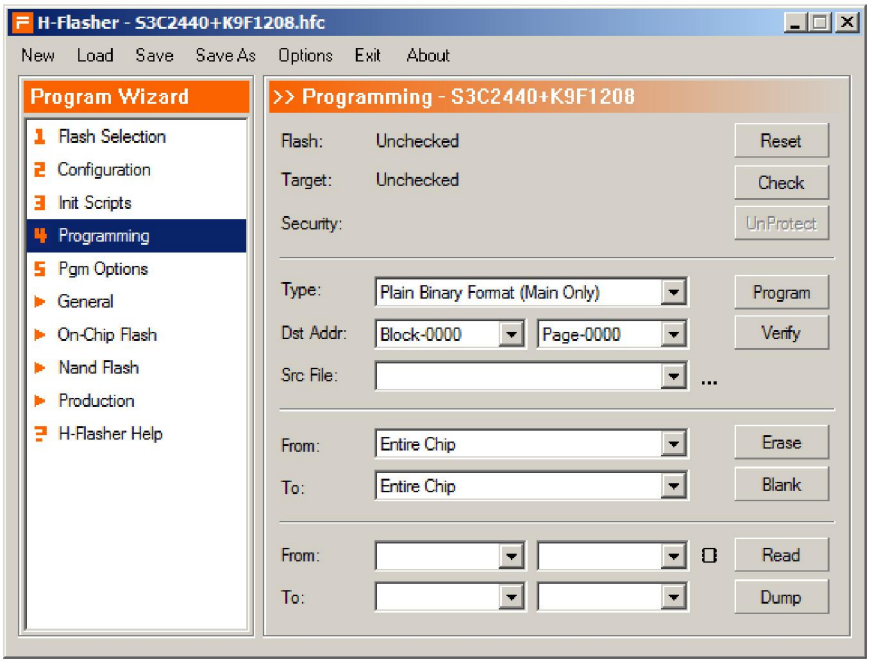


图 5: H-FLASHER 编程页面

在 H-FLASHER 的编程页面上，用户可以对 NAND FLASH 执行不同的操作，包括 Check，Program，Verify，Erase，Blank，Read 和 Dump．各个基本操作的简单说明如下表所示:

表 1: NAND FLASH 基本操作与说明

基 本 操 作	说 明
Reset	用来对目标系统执行复位操作.
Check	用来检测 FLASH 并返回读取到的 FLASH ID.
Program	用来对 FLASH 执行编程操作．该操作会将用户指定的文件烧写到指定的位置．烧写前，会自动先执行 FLASH 擦除操作.
Verify	用来执行校验操作．该操作会将指定的 FLASH 位置中的内容读取回来，并和指定的文件进行比较.
Erase	用来执行 FLASH 擦除操作．该操作会将用户指定的 BLOCK 擦掉．并在成功擦除后,提示发现的坏块.
Blank	用来检查 FLASH 是否为空．该操作会对用户指定的 BLOCK 执行是否为空操作，并在检查完成后提示结果.
Read	用来执行 READ 操作．该操作会将用户指定的 BLOCK/PAGE 范围内的数据读取回来. READ 操作只读取主数据区的数据．空闲数据区的数据不包含在该操作中．假设 PAGE 的大小为(2048+64)字节. 读取一个页面，返回的数据大小是 2048 字节.
Dump	用来执行 DUMP 操作．该操作会将用户指定的 BLOCK/PAGE 范围内的数据读取回来. DUMP 操作会同时读取主数据区和空闲数据区的数据．这也是 DUMP 操作和 READ 操作的区别．假设 PAGE 的大小为(2048+64)字节, DUMP 一个页面，返回的数据大小是 2112 字节.

本文中，测试使用的硬件平台是，S3C2440 + K9F1208. 其中 K9F1208 芯片共包含 4096 个 BLOCK，每个 BLOCK 包含 32 个 PAGE，每个 PAGE 的大小为 (512 + 16) 字节. 接下来，将基于这个硬件平台简单介绍如何通过 H-JTAG/H-FLASHER 对 NAND FLASH 进行操作.

3.3.1 Reset 操作

如果目标系统的复位信号有连接到 JTAG 接口，通过 Reset 按钮可以对目标系统执行系统复位操作. 这样可以方便用户复位目标系统.

3.3.2 Check 操作

Check 操作会读取 FLASH 的 ID 并将其返回. 通过 Check 操作，可以验证当前的烧写配置是否正确，也可以用来简单测试硬件是否工作正常. 在 S3C2440+K9F1208 的测试平台上，点击 Check，可以看到从目标系统读取的 FLASH ID. 如图 X 中所示, FLASH 型号为 K9F1208 的 FLASH ID 为 0x007600EC.

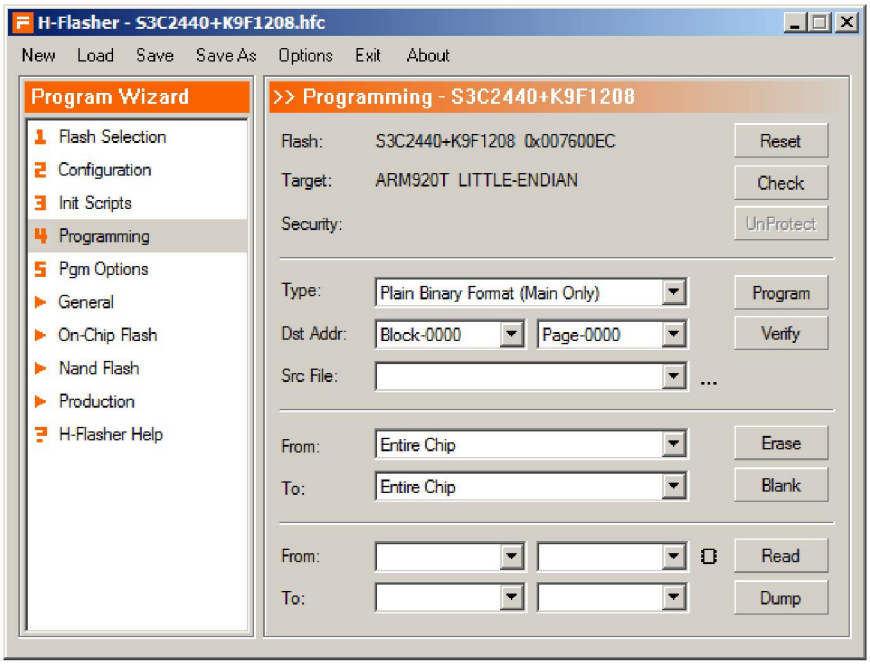


图 6: Check 操作

3.3.3 Program/Verify 操作

如果要将文件烧写到 NAND FLASH 中去，只需要指定烧写文件的格式和路径，以及烧写到哪个 BLOCK/PAGE 即可. 如果烧写文件不包含空闲数据区数据，请选择 Plain Binary Format (Main Only) 格式. 如果烧写文件本身包含 ECC 校验码等信息，请选择 Plain Binary Format (Main + Spare/Oob) 格式.

选择 Main Only 格式的时候, H-FLASHER 认为烧写文件只包含要烧写到主数据区的数据. 空闲数据区用来存放动态实时计算的 ECC 校验码，或是以 0xFF 填充.

选择 Main + Spare/Oob 格式的时候, H-FLASHER 认为烧写文件本身包含了主数据区数据和空闲数据区数据. 所以，H-FLASHER 会检查烧写文件的大小是否是页面的主数据区加空闲数据区的大小的整数倍. 以

K9F1208 为例，该 FLASH 的页面主数据区和空闲数据区的大小是 512 + 16 = 528 字节. 那选择 Main + Spare/Oob 格式时，要烧写的二进制的文件的大小必须是 528 的整数倍，否则会提示错误.

假设要将 U-BOOT 烧写到 K9F1208 的 BLOCK-0 的 PAGE-0 去. 因为 U-BOOT 本身只包含主数据区的数据，不包含 ECC 校验码. 所以烧写格式选择: Plain Binary Format (Main Only). 烧写时的设置如下图所示:

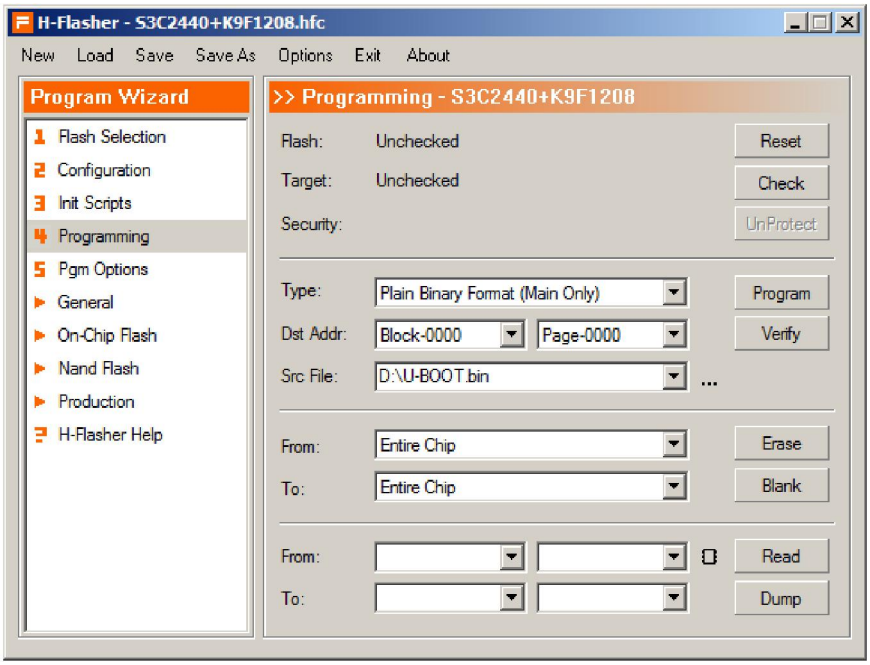


图 7: 烧写 U-BOOT

在编程页面中，点击 Program 按钮，就可以开始烧写. 烧写过程中，会显示当前的操作，时间及进度. 完成后, Program 对话框如下图所示:

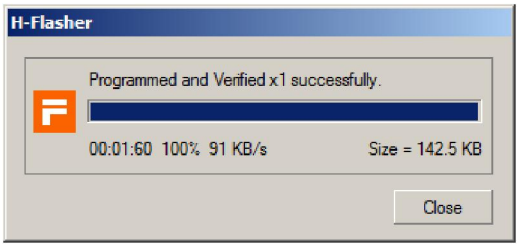


图 8: 烧写完成

烧写完成后，点击 Verify 按钮，就可以执行相应的校验操作. 校验操作会将数据从 FLASH 中读取回来，并和烧写文件进行比较. 烧写完 U-BOOT 后, Verify 对话框如下图所示.

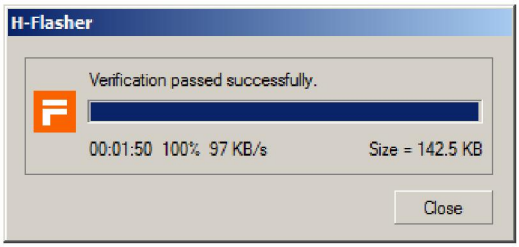


图 9: 校验完成

3.3.4 Erase / Blank 操作

在 H-FLASHER 的编程页面，还可以对 FLASH 执行擦除操作和检查 FLASH 是否为空的操作．操作范围可以由用户指定．下面先执行一个整片擦除操作．擦除完成后，再对整片 FLASH 做一个是否为空的检查．范围选择为 Entire Chip，如下图所示．

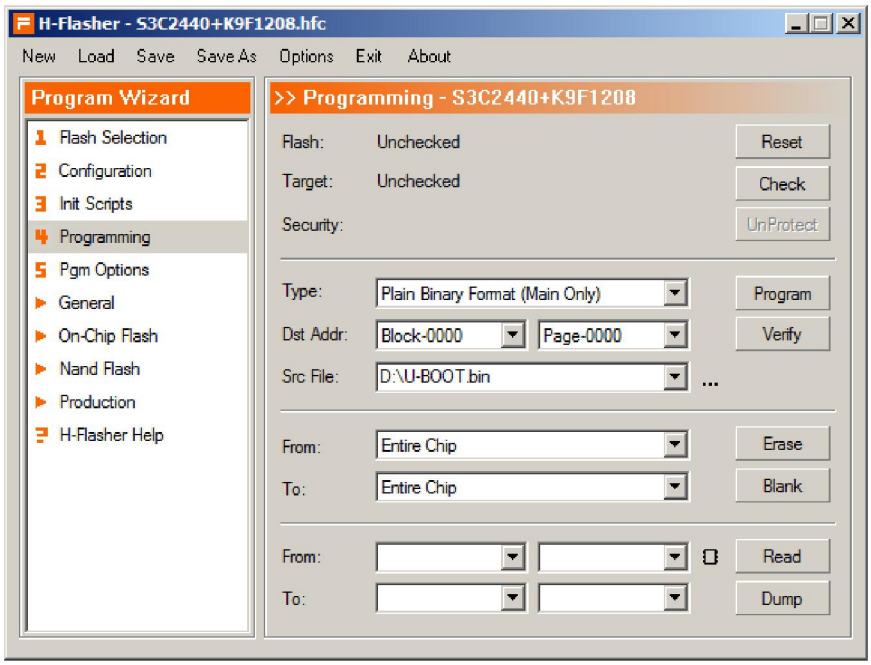


图 10:　Erase/Blank 设置

在编程页面点击 Erase 后，H-FLASHER 就会对指定的 BLOCK 执行擦除操作．擦除过程中，会实时显示进度与时间．完成后，Erase 对话框显示如下图所示．从图中可以看出，擦除操作成功，且该 FLASH 没有坏块．

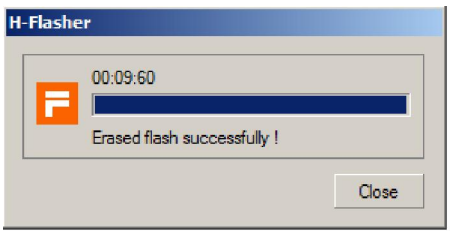


图 11:　擦除完成

擦除完成后，点击 Blank 按钮，开始对整片 FLASH 执行是否为空检查．检查过程中，会实时显示当前所花费的时间．检查完成后，Blank 对话框如下图所示．检查结果显示，FLASH 为空．

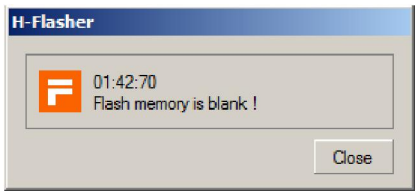


图 12:　检查完成

3.3.5 Read / Dump 操作

用户需要从 NAND FLASH 中读取数据的时候，可以使用 Read/Dump 操作. 根据需求，可以选择 Read 或是 Dump. Read 和 Dump 的区别在于: Read 操作只读取主数据区的数据，而 Dump 操作会同时读取主数据区和空闲数据区的数据. 提供 Dump 操作的目的是方便用户在需要的时候查看与分析空闲数据区的数据，或是做完全拷贝.

接下来，尝试从 FLASH 中将部分数据 Read 和 Dump 回来. 指定操作范围为 Block-0 的 Page-0 到 Block-7 的 Page-31，整整 8 个 BLOCK 的数据. 如下图所示.

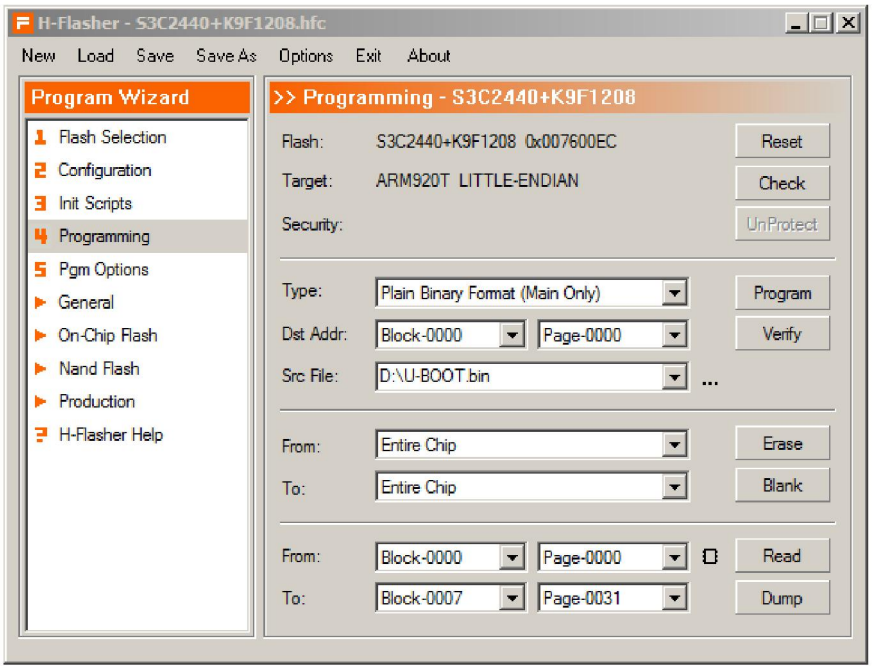


图 13: Read/Dump 设置

点击 Read 按钮，开始 Read 指定的 8 个 BLOCK 的数据. Read 过程中，会实时显示时间，速度和进度. 完成后, Read 对话框如下图所示.

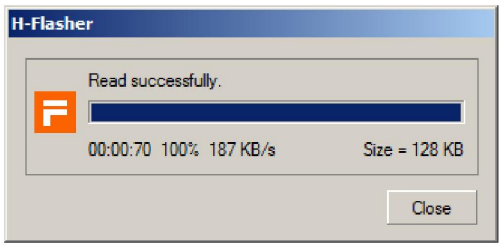


图 14: Read 完成

点击 Dump 按钮，开始 Dump 指定的 8 个 BLOCK 的数据. Dump 过程中，会实时显示时间，速度和进度. 完成后, Dump 对话框如下图所示.

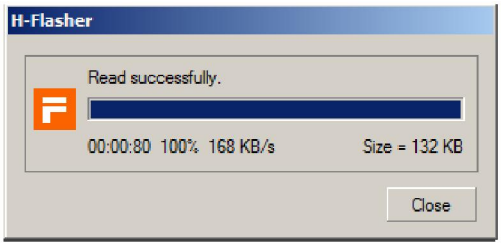


图 15: Dump 完成

Read 的文件大小为 8 x 32 x 512 = 131072 字节，而 Dump 的文件大小为 8 x 32 x (512 + 16) = 135168 字节. Read 的文件比 Dump 的文件要小，原因是 Dump 操作包含了空闲数据区的数据. Read 文件的大小是页面的主数据区大小的整数倍，而 Dump 文件的大小是页面的主数据区加空闲数据区大小的整数倍.

4. 一键烧写嵌入式系统

嵌入式操作系统一般都包括启动程序，内核镜像文件和文件系统等多个模块. 烧写时，一般都是先通过 H-JTAG 将启动程序烧写好，然后再通过启动程序来烧写内核镜像和文件系统等其它模块. 在做产品烧写时，这种方法效率比较低. 如果能一次将启动程序，内核镜像和文件系统一次烧写到 NAND FLASH 中去，会方便很多. H-FLASHER 支持一键烧写嵌入式系统. 具体的实现，请参考专门的文档**<<使用 H-JTAG 一键烧写嵌入式系统>>**. 用户可以从我们官方网站的下载中心下载该文档.

5. 产品烧写模式

H-FLASHER 支持产品烧写模式. 在 H-FLASHER 左侧的编程导航中点击 Production, 就会进入烧写模式设置页面. 在该页面中, 选择 “Enable production mode for flash programming” 即可使能产品烧写模式.

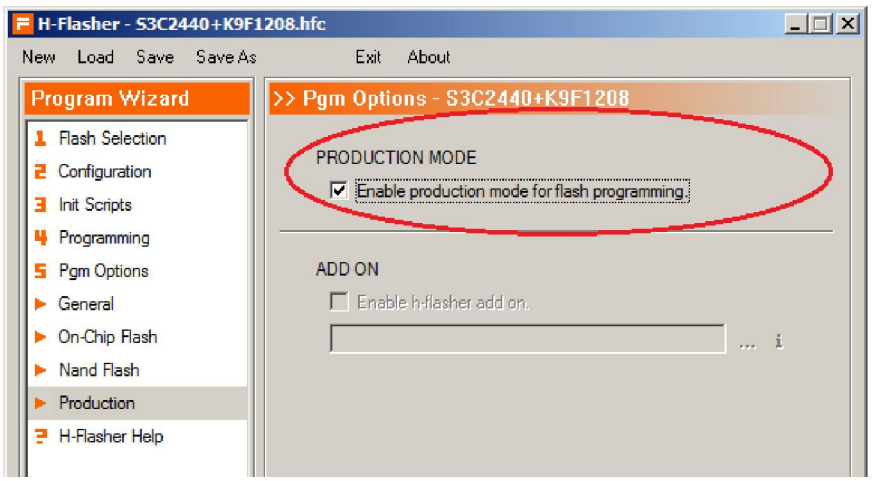


图 16: 使能产品烧写模式

使能产品烧写模式后, 在编程页面上点击 Program 按钮, H-FLASHER 就会自动进入产品烧写模式, 如下图所示.

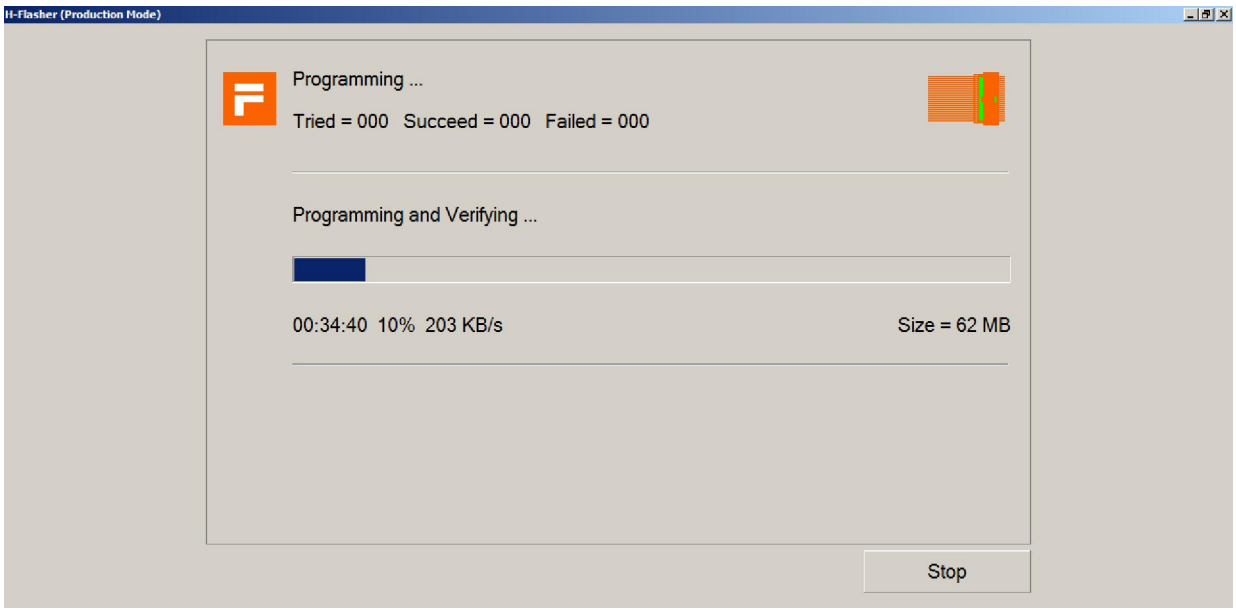


图 17: 产品烧写模式

在产品烧写模式下, 用户操作被简化, H-FLASHER 完全通过检测 JTAG 的插拔来自动控制烧写, 以提高效率. 当 H-FLASHER 检测到有新的连接时, 自动开始烧写. 烧写完成后, 提示用户断开当前连接. H-FLASHER 检测到连接断开后, 会进入新的循环, 等待下一个新连接. 整个过程都由 H-FLASHER 自动控制, 用户无需在 PC 端做任何鼠标/键盘操作.

6．高级应用: 修改 NAND FLASH 驱动

因为 NAND FLASH 本身的特点，在实际应用中，需要引入坏块管理和 ECC 校验以保证数据的可靠性. H-FLASHER 自带的驱动程序能满足用户的大部分需求，但其中关于坏块标识和 ECC 处理可能和用户采用的算法有差别. 用户可以基于我们提供的代码进行修改，以生成符合自己需求的驱动程序.

H-FLASHER 自带的 NAND FLASH 驱动程序都是基于固定的模板，使用 ADS 1.2 开发的. 用户可以在 H-JTAG 安装目录下找到所有 NAND FLASH 驱动的源代码. **(H-JTAG 安装路径\FDevice\NAND-FLASH\SourceCodes)**. 关于 NAND FLASH 驱动程序的介绍，用户可以参考**<<NAND FLASH DRIVER ADDING GUIDE>>**.

下面，将以一个实际的例子，一步一步详细介绍如何根据实际需求修改 NAND FLASH 驱动程序，以便能使用该驱动程序直接烧写 LINUX. 我们使用的硬件测试平台是 S3C2440 +K9F1208. H-FLASHER 自带的 S3C2440+K9F1208 的驱动程序中，没有引入 ECC 处理. 在下面的例子中，我们将如何修改 H-FLASHER 自带的驱动程序，添加 ECC 处理，从而能够使用 H-FLASHER 直接烧写启动程序, LINUX 内核镜像和文件系统.

需要注意的是，下面的介绍只是针对我们测试使用的硬件平台及配套的软件. 其中涉及的具体代码及使用的烧写地址，并不具有通用性. 用户修改驱动程序的时候，需要参考实际使用的软件实现.

6.1 新建一个 NAND FLASH 描述文件

首先，我们需要建立一个新的 NAND FLASH 描述文件，然后基于这个新的描述文件进行修改. 这样可以保留 H-FLASHER 自带的设备描述文件.

进入目录 “H-JTAG 安装路径\FDevice\NAND-FLASH\”，并找到文件 S3C2440+K9F1208. 将该文件复制一份，并将其命名为 S3C2440+K9F1208-ECC. 然后用记事本打开 S3C2440+K9F1208-ECC 这个文件. 在文件中，找到 FLASH_DRIVER=1102/0/0 这行，并将其修改为 FLASH_DRIVER=1150/0/0，然后保存. 这样，我们就新建了一个 NAND FLASH 描述文件. 该文件中关于 FLASH 的描述和 S3C2440+K9F1208 是一样的，只不过其对应的 NAND FLASH 驱动程序是 1150.

新建了 NAND FLASH 描述文件后，重新运行 H-FLASHER 后，用户就应该能在 FLASH 列表的 NAND-FLASH 下面发现新添加的 S3C2440+K9F1208-ECC.

6.2 新建一个 NAND FLASH 驱动程序

在新建了 NAND FLASH 描述文件后，我们需要新建一个与其对应的驱动程序. 然后基于这个新的驱动程序进行修改.

进入目录 ”H-JTAG 安装路径\FDevice\NAND-FLASH\SourceCodes (ADS1.2)\”，找到文件夹”DRIVER-1102 S3C2440+K9F1208” . 拷贝该文件夹，将其复制到合适的位置，并将其重新命名为 DRIVER-1150 S3C2440+K9F1208-ECC. 然后进入文件夹内，找到 ADS1.2 工程项目文件 DRIVER.mcp. 双击 MCP 文件, ADS1.2 会自动运行. 如果 ADS1.2 运行后没有打开工程文件，可以直接在 ADS1.2 中使用 FILE -> OPEN 打开.

打开工程项目文件后，还需要在 ADS1.2 中设置一下编译链接后，输出的二进制文件的名称. 在菜单上选择 Edit -> ARM Settings -> Linker -> ARM FromELF -> Output File Name. 然后将输出文件名修改为

Driver1150. 然后点击 OK.

设置好输出文件名后，在 ADS1.2 中选择 Project -> Make，重新编译工程项目. 成功编译后，会在 Driver_Data\ARM\目录下，生成新的驱动程序二进制文件 Driver1150.

将新生成的 Driver1150 拷贝到"H-JTAG 安装路径\FDevice\NAND-FLASH\Drivers\"下去. 这样，用户选择 FLASH 为 S3C2440+K9F1208-ECC 的时候, H-FLASHER 就会使用 Driver1150 来对 FLASH 进行操作.

因为我们没有修改过驱动程序的源代码，所以 Driver1150 和 Driver1102 实际上还是一样的. 下面将介绍如何修改源代码，以满足我们的要求.

6.3 添加 ECC 处理

到目前为止，我们已经新建了 NAND FLASH 描述文件和与其对应的新的驱动程序. 下面，我们将修改驱动 1150，引入 ECC 处理，以方便我们通过 H-FLASHER 直接烧写 LINUX.

我们使用的测试平台中, LINUX 系统是由 U-BOOT 来加载和启动的. 所以，我们首先需要了解 U-BOOT 中的 ECC 处理方式. 通过分析 U-BOOT 的代码，发现其并没有采用 S3C2440 内置的硬件 ECC 模块来计算 ECC 校验码，而是采用了软件来生成. 计算 ECC 校验码时，每 256 字节的数据，生成 3 个字节的 ECC 校验码. 因为 K9F1208 每个页面的主数据区为 512 个字节，所以总共有 6 个字节的 ECC 校验码. 这 6 个字节的校验码按顺序存放在页面的空闲数据区的字节 0, 1, 2, 3, 6, 7.

首先，在驱动程序的 Flash.c 中，添加计算 ECC 的相关代码. 如下:

```
static const U8 nand_ecc_precalc_table[] = {
    0x00, 0x55, 0x56, 0x03, 0x59, 0x0c, 0x0f, 0x5a,
    0x5a, 0x0f, 0x0c, 0x59, 0x03, 0x56, 0x55, 0x00,
    0x65, 0x30, 0x33, 0x66, 0x3c, 0x69, 0x6a, 0x3f,
    0x3f, 0x6a, 0x69, 0x3c, 0x66, 0x33, 0x30, 0x65,
    0x66, 0x33, 0x30, 0x65, 0x3f, 0x6a, 0x69, 0x3c,
    0x3c, 0x69, 0x6a, 0x3f, 0x65, 0x30, 0x33, 0x66,
    0x03, 0x56, 0x55, 0x00, 0x5a, 0x0f, 0x0c, 0x59,
    0x59, 0x0c, 0x0f, 0x5a, 0x00, 0x55, 0x56, 0x03,
    0x69, 0x3c, 0x3f, 0x6a, 0x30, 0x65, 0x66, 0x33,
    0x33, 0x66, 0x65, 0x30, 0x6a, 0x3f, 0x3c, 0x69,
    0x0c, 0x59, 0x5a, 0x0f, 0x55, 0x00, 0x03, 0x56,
    0x56, 0x03, 0x00, 0x55, 0x0f, 0x5a, 0x59, 0x0c,
    0x0f, 0x5a, 0x59, 0x0c, 0x56, 0x03, 0x00, 0x55,
    0x55, 0x00, 0x03, 0x56, 0x0c, 0x59, 0x5a, 0x0f,
    0x6a, 0x3f, 0x3c, 0x69, 0x33, 0x66, 0x65, 0x30,
    0x30, 0x65, 0x66, 0x33, 0x69, 0x3c, 0x3f, 0x6a,
    0x6a, 0x3f, 0x3c, 0x69, 0x33, 0x66, 0x65, 0x30,
    0x30, 0x65, 0x66, 0x33, 0x69, 0x3c, 0x3f, 0x6a,
    0x0f, 0x5a, 0x59, 0x0c, 0x56, 0x03, 0x00, 0x55,
```

```
    0x55, 0x00, 0x03, 0x56, 0x0c, 0x59, 0x5a, 0x0f,
    0x0c, 0x59, 0x5a, 0x0f, 0x55, 0x00, 0x03, 0x56,
    0x56, 0x03, 0x00, 0x55, 0x0f, 0x5a, 0x59, 0x0c,
    0x69, 0x3c, 0x3f, 0x6a, 0x30, 0x65, 0x66, 0x33,
    0x33, 0x66, 0x65, 0x30, 0x6a, 0x3f, 0x3c, 0x69,
    0x03, 0x56, 0x55, 0x00, 0x5a, 0x0f, 0x0c, 0x59,
    0x59, 0x0c, 0x0f, 0x5a, 0x00, 0x55, 0x56, 0x03,
    0x66, 0x33, 0x30, 0x65, 0x3f, 0x6a, 0x69, 0x3c,
    0x3c, 0x69, 0x6a, 0x3f, 0x65, 0x30, 0x33, 0x66,
    0x65, 0x30, 0x33, 0x66, 0x3c, 0x69, 0x6a, 0x3f,
    0x3f, 0x6a, 0x69, 0x3c, 0x66, 0x33, 0x30, 0x65,
    0x00, 0x55, 0x56, 0x03, 0x59, 0x0c, 0x0f, 0x5a,
    0x5a, 0x0f, 0x0c, 0x59, 0x03, 0x56, 0x55, 0x00
};

int nand_calculate_ecc(U8 *dat, U8 *ecc_code)
{
    int i;
    U8 idx, reg1, reg2, reg3, tmp1, tmp2;

    /* Initialize variables */
    reg1 = reg2 = reg3 = 0;

    /* Build up column parity */
    for(i = 0; i < 256; i++) {
        /* Get CP0 - CP5 from table */
        idx = nand_ecc_precalc_table[*dat++];
        reg1 ^= (idx & 0x3f);

        /* All bit XOR = 1 ? */
        if (idx & 0x40) {
            reg3 ^= (unsigned char) i;
            reg2 ^= ~((unsigned char) i);
        }
    }

    /* Create non-inverted ECC code from line parity */
    tmp1  = (reg3 & 0x80) >> 0; /* B7 -> B7 */
    tmp1 |= (reg2 & 0x80) >> 1; /* B7 -> B6 */
    tmp1 |= (reg3 & 0x40) >> 1; /* B6 -> B5 */
    tmp1 |= (reg2 & 0x40) >> 2; /* B6 -> B4 */
    tmp1 |= (reg3 & 0x20) >> 2; /* B5 -> B3 */
    tmp1 |= (reg2 & 0x20) >> 3; /* B5 -> B2 */
    tmp1 |= (reg3 & 0x10) >> 3; /* B4 -> B1 */
```

```
    tmp1 |= (reg2 & 0x10) >> 4; /* B4 -> B0 */

    tmp2  = (reg3 & 0x08) << 4; /* B3 -> B7 */
    tmp2 |= (reg2 & 0x08) << 3; /* B3 -> B6 */
    tmp2 |= (reg3 & 0x04) << 3; /* B2 -> B5 */
    tmp2 |= (reg2 & 0x04) << 2; /* B2 -> B4 */
    tmp2 |= (reg3 & 0x02) << 2; /* B1 -> B3 */
    tmp2 |= (reg2 & 0x02) << 1; /* B1 -> B2 */
    tmp2 |= (reg3 & 0x01) << 1; /* B0 -> B1 */
    tmp2 |= (reg2 & 0x01) << 0; /* B7 -> B0 */

    /* Calculate final ECC code */
    ecc_code[0] = ~tmp1;
    ecc_code[1] = ~tmp2;
    ecc_code[2] = ((~reg1) << 2) | 0x03;

    return 0;
}
```

接下来，我们需要修改函数 nand_program_page()，以添加 ECC 处理．该函数调用 nand_calculate_ecc()，以获取 ECC 校验码，并将其写到空闲数据区中去．具体代码如下:

```
U32 nand_program_page(U32 blockidx, U32 pageidx)
{
    int i;
    U32 blkpgnum;
    U32 *ptr32;
    U8  ecc[6];
    U8  spare[16];
    U8  data[512];

    ptr32 = (U32*)data;
    blkpgnum = blockidx*NAND_BLOCK_PAGE_NUM + pageidx;

    for(i = 0; i < 16; i++)
        spare[i] = 0xFF;

    //Program a page
    nand_wait();

    *SysAddr16(NFCMD) = 0x00;
    *SysAddr16(NFCMD) = 0x80;

    *SysAddr16(NFADDR) = 0x00;
```

```
    *SysAddr16(NFADDR) = (blkpgnum>>0)  & 0xFF;
    *SysAddr16(NFADDR) = (blkpgnum>>8)  & 0xFF;
    *SysAddr16(NFADDR) = (blkpgnum>>16) & 0xFF;

    for(i = 0; i < NAND_PAGE_SIZE; i += 4){
        *ptr32 = read_from_host();
        *SysAddr8(NFDATA) = data[i+0];
        *SysAddr8(NFDATA) = data[i+1];
        *SysAddr8(NFDATA) = data[i+2];
        *SysAddr8(NFDATA) = data[i+3];
        ptr32 += 1;
    }

    nand_calculate_ecc(data, ecc);
    nand_calculate_ecc(data+256, ecc+3);

    spare[0] = ecc[0];
    spare[1] = ecc[1];
    spare[2] = ecc[2];
    spare[3] = ecc[3];
    spare[6] = ecc[4];
    spare[7] = ecc[5];

    for(i = 0; i < NAND_SPARE_SIZE; i++)
        *SysAddr8(NFDATA) = spare[i];

    *SysAddr16(NFCMD) = 0x10;

    nand_wait();

    //Read status
    *SysAddr16(NFCMD) = 0x70;

    nand_wait();

    if(*SysAddr8(NFDATA) & 0x1)
        return 1;

    return 0;
}
```

同时，因为驱动程序添加了比较多的代码，为了保证不会出现堆栈溢出的异常情况，需要将堆栈空间设置大一些. 打开 INIT.s 文件，将堆栈设置修改成如下所示:

```
    ;//Set SP
    MOV SP, PC
    ADD SP, SP, #0x2000
```

H-JTAG 提供的 FLASH 驱动程序，大部分都是与位置无关(POSITION INDEPENDENT)的. 这样驱动程序可以下载到任何 RAM 地址上运行. 但因为修改了驱动程序，新添加的代码不一定是与位置无关的. 为了简单起见，在重新编译前，把 RO_BASE 设置成 0x30000000. 烧写时，利用位于地址 0x30000000 的 SDRAM 来完成烧写. 在 ADS 的菜单上选择 Edit -> ARM Settings -> Linker -> ARM Linker -> RO_BASE -> 0x30000000.

最后，重新编译工程. 生成新的 Driver1150. 将新生成的 Driver1150 拷贝到”H-JTAG 安装路径\FDevice\NAND-FLASH\Drivers\”下去，覆盖掉原来的文件.

6.4 新驱动测试

通过前面的修改，新的 NAND FLASH 描述文件和驱动程序都已经准备就绪. 接下来，让我们通过 H-FLASHER 直接烧写 LINUX，测试一下修改的驱动程序是否正确工作. 测试中，我们需要将启动程序，内核镜像和文件系统三个文件烧写到模板系统中去. 要烧写的 3 个文件及其对应的烧写地址如下所示:

烧写文件	文件描述	文件大小	烧写地址	OOB/ECC
u-boot.bin	启动程序	143 KB	Block-000/Page-0	不包含
uImage	Linux 内核镜像文件	1.86 MB	Block-024/Page-0	不包含
filesystem.yaffs	文件系统(YAFFS)	58.5 MB	Block-152/Page-0	包含

首先，运行 H-FLASHER，装载 S3C2440+K9F1208.hfc 配置文件，并重新选择 FLASH 为 S3C2440+K9F1208-ECC. 如下图所示:

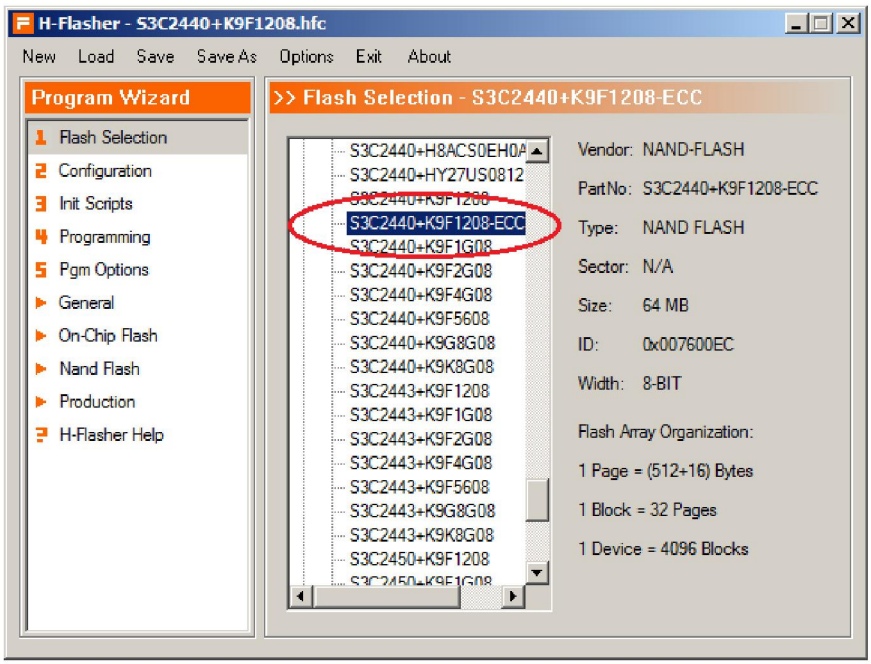


图 18: 选择 S3C2440+K9F1208-ECC

接下来，先烧写 U-BOOT. 在编程页面，选择烧写类型为”Plain Binary Format (Main Only)”，烧写地址为 Block-0/Page-0，烧写文件为 u-boot.bin. 设置如下图所示. 然后点击 Program 按钮开始烧写 U-BOOT.

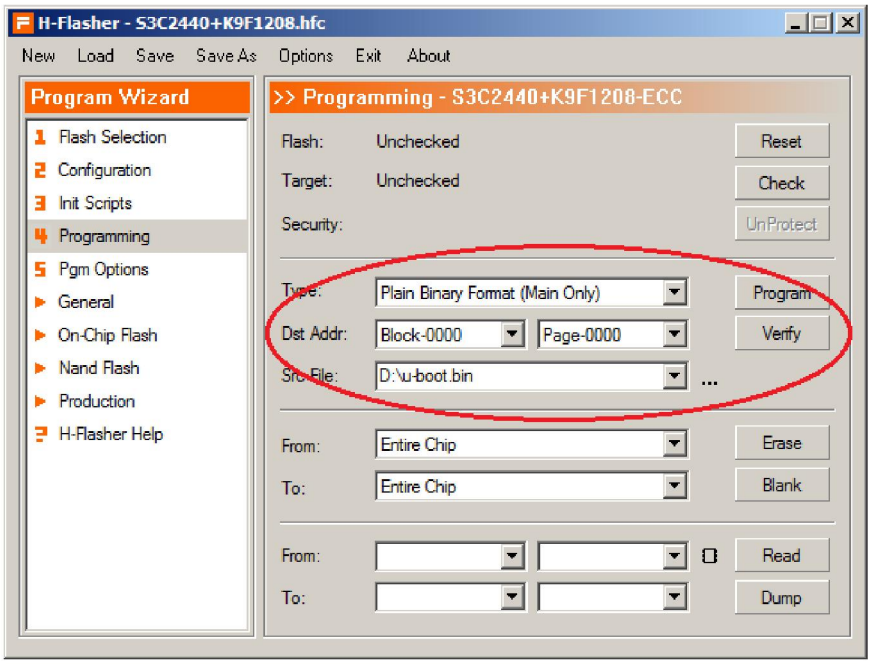


图 19: 烧写 U-BOOT

然后烧写内核镜像. 编程设置中，选择烧写类型为”Plain Binary Format (Main Only)”，烧写地址为 Block-24/Page-0，烧写文件为 uImage. 设置如下图所示. 然后点击 Program 按钮开始烧写内核镜像.

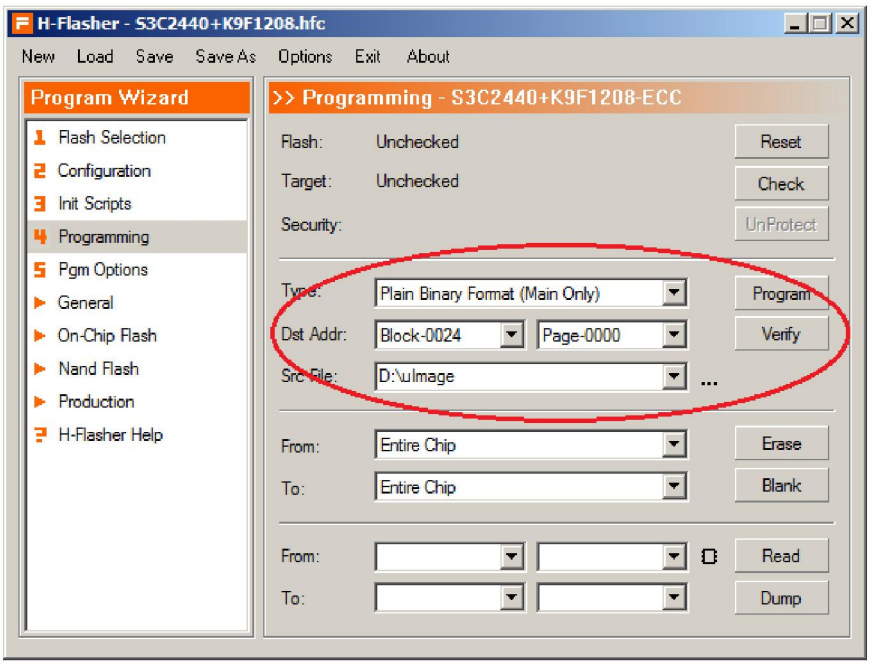


图 20: 烧写内核镜像

最后，烧写文件系统．需要注意的是，测试中使用的文件系统是 YAFFS 格式的，文件本身包含了 OOB/ECC 数据．在编程设置中，选择烧写类型为”Plain Binary Format (Main + Spare/Oob)”，烧写地址为 Block-152/Page-0，烧写文件为 filesystem.yaffs．设置如下图所示．然后点击 Program 按钮开始烧写文件系统．

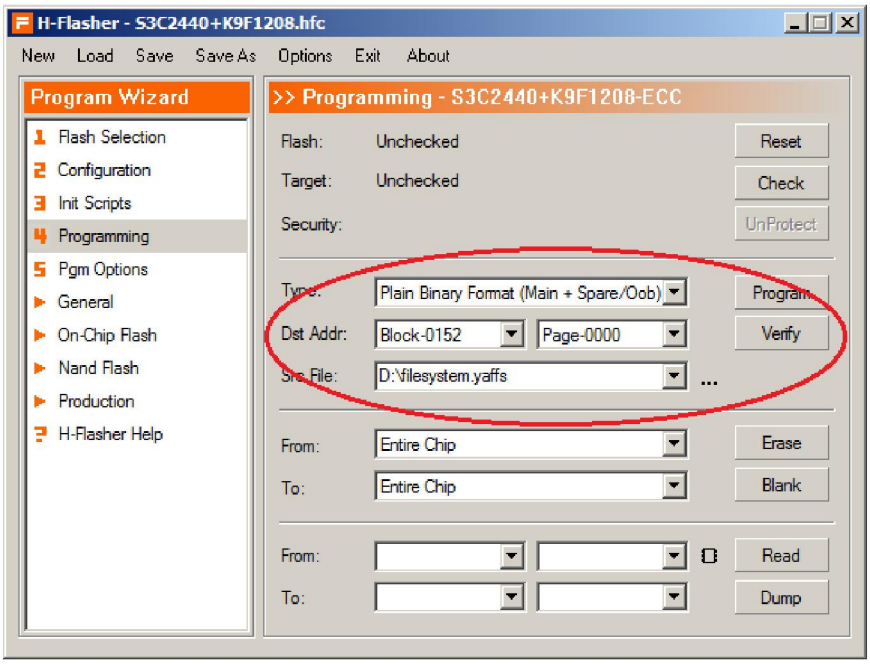


图 21:　烧写文件系统

烧写完成后，给目标系统重新上电，LINUX 系统正常启动运行．测试说明 NAND FLASH 驱动程序修改成功，引入的 ECC 处理测试通过．

附录 A: NAND FLASH 介绍

相对于 NOR FLASH 而言，NAND FLASH 低价格，高容量的特点，在嵌入式系统设计中得到越来越多的使用. 同时，因为 NAND FLASH 的设计和工艺的原因, NAND FLASH 芯片不支持片上执行 (Execute In Place)，不支持随机访问，而且会有随机坏块.

因为不支持片上执行，如果要从 NAND FLASH 启动，需要 MCU 本身提供一定的辅助机制. 以 S3C2440 为例，内部的 4K SRAM 也称作 SteppingStone. 如果选择从 NAND FLASH 启动，上电后，该 4K SRAM 会被映射到地址 0X0. 同时 MCU 会将 NAND FLASH 前 4K 的数据读取并拷贝到 SRAM，然后从 SRAM 开始执行程序，实现从 NAND FLASH 启动.

NAND FLASH 是基于 BLOCK/PAGE (块/页面)结构的. 一块芯片包含一定数量的 BLOCK，每个 BLOCK 包含一定数量的 PAGE. 每个 PAGE 包含一定数量的存储单元. 每个 PAGE 的存储单元又分为两个部分: MAIN AREA (主数据区) 和 SPARE AREA (空闲数据区) . 一般来说，主数据区用来存储用户数据，空闲数据区用来存放坏块标识, ECC 校验码等其它附加信息.

以三星的 K9F2G08 为例，该芯片有 2048 个 BLOCK，每个 BLOCK 包含 64 个 PAGE，每个 PAGE 包含 2K 字节的主数据区,和 64 字节的空闲数据区. 该 FLASH 的结构如下.

SAMSUNG K9F2G08		
1 Page	**= (2K + 64) Bytes**	每个页面的大小为 2K + 64 字节，其中 2K 字节为主数据区, 64 字节为空闲数据区
1 Block	**= 64 Pages x (2K + 64)B** **= (128K + 4K) Bytes**	每个块包含 64 个页面. 所以每个块的大小为 128K + 4K 字节. 主数据区的大小为 128K，空闲数据区的大小为 4K.
1 Device	**= 2048 Blocks x 64Pages x (2K+64)B** **= (256M + 8M) Bytes**	整片 FLASH 共包含 2048 个块，所以总容量为 256M +8M 字节，其中主数据区的大小为 256M，空闲数据区的大小为 8M.

NAND FLASH 芯片一般只提供 8 位或是 16 位的数据线，且地址线和数据线复用的. 所以, NAND FLASH 的读写需要多个地址周期，一般都通过 NAND FLASH 控制器来完成.

NAND FLASH 的擦除操作是以 BLOCK 为基本单位的. 对一个 BLOCK 执行擦除操作，会将该 BLOCK 包含的所有 PAGE 的数据都擦除掉. 编程操作是以 PAGE 为单位的. 在编程前，需要先执行擦除操作，以保证数据能够正确写入.

A1. NAND FLASH 坏块

如果一个 BLOCK 中包含一个或多个无效的比特，那该块就被定义为一个坏块. 实际上，每片出厂的 NAND FLASH 芯片都有可能会包含坏块 (Bad/Invalid Block). 而且在使用的过程中，也可能会产生新的坏块. 但对于出厂的每片 NAND FLASH，厂商会保证芯片的第一个 BLOCK 不是坏块. 这样可以保证存储在第一个 BLOCK 的数据的安全性和可靠性，使其适合用于存储 BOOT 程序或是坏块表/替换表等比较重要的数据.

每片出厂的 NAND FLASH 芯片，厂商都会做坏块检测，并做好标识．通过读取坏块标识，可以判断一个 BLOCK 是否是坏块．在对 NAND FLASH 执行 BLOCK 擦除操作后，同样也可以通过读取坏块标识来判断其是否是坏块．以三星的 K9F2G08 为例．对一个 BLOCK 执行擦除操作后，如果该 BLOCK 的第一个 PAGE 或是第二个 PAGE 的空闲区域的第一个 BYTE 不是 0xFF，那这个 BLOCK 就是坏块．

坏块标识一般都是存放在空闲数据区的．但需要注意的是，坏块标识在空闲数据区内的具体位置，不同芯片型号，不同芯片厂商可能会有不同的定义．在使用的时候，需要具体参考芯片的手册．在有些情况下，BOOT 程序/操作系统本身也可能会对坏块进行特别标识．在这种情况下，坏块标识的位置是 BOOT 程序/操作系统决定的，和芯片本身规定的坏块标识位置可能会不一样．

A2. NAND FLASH ECC 校验

在 NAND FLASH 使用过程中，应当避免将数据写到坏块中去．但 NAND FLASH 在使用过程中，可能会产生新的坏块．在读取数据的时候，为了保证数据的可靠性，需要引入 ECC 校验．比较常见的是 1-BIT ECC 校验．通过校验，可以检测出任何 BIT 的错误．如果是 1-BIT 错误，还可以通过算法进行纠正，提高可靠性．

在将用户数据写如到一个 PAGE 之前，会先根据用户的数据，通过 ECC 算法计算出相应的 ECC 校验码．然后将用户数据和 ECC 校验码一起烧写进去．其中，用户数据写入到主数据区，ECC 校验码写入到空闲数据区．ECC 校验码一般都是存放在空闲数据区的，但存放位置并没有统一的标准，具体的存放位置是由用户/应用决定的．

在从一个 PAGE 读取数据的时候，会将存储在主数据区的用户数据和存储在空闲数据区的 ECC 校验码一并读取回来．根据读取回来的用户数据，重新计算一个 ECC 校验码．然后比较新计算的校验码和读取回来的校验码．通过比较，就可以判断数据的正确性．如果有错，执行相应的纠错操作．如果不能纠正，提示并丢弃该数据．

ECC 的计算可以通过软件和硬件模块来实现．有些 MCU 的 NAND FLASH 控制器本身集成了 ECC 硬件模块，支持 ECC 硬件编码/解码/纠错，可以直接使用．如果 MCU 本身不提供 ECC 硬件模块，或是用户希望选择不同的 ECC 算法，也可以通过软件来实现．

在实际应用中，选择硬件或是软件来计算 ECC 校验码，使用什么样 ECC 算法，将 ECC 校验码存放在空闲数据区的什么位置，都是由用户或是应用决定的．在很多情况下，都是由使用的 BOOTLOADER 及其具体实现决定的．

A3. NAND FLASH 坏块管理

在 NAND FLASH 使用当中，应当避免把数据写到坏块中去．因为坏块的存在，需要引入坏块管理，以规避坏块的使用．坏块管理有很多方式，但常见的有两种方式．

第一种是顺序替换．执行写操作的时候，如果遇到一个坏块，就将数据写到下一个好的 BLOCK 中去．读取数据的时候，如果遇到坏块，就从下一个好的 BLOCK 中读取．这是最简单，也是最常用的一种方式．

另一种是建立替换表(Relocation Table)的方式．在 NAND FLASH 中，保留一定数量的 BLOCK 用于替换 (通常位于 NADN FLASH 的末端)．执行写操作的时候，如果遇到一个坏块，就从保留的 BLOCK 中选取一个可以

使用的 BLOCK，用于替换该坏块．数据都写入到该替换 BLOCK 中去．同时，记录下坏块到替换 BLOCK 的映射关系．在完成所有写操作后，将替换表存储到 FLASH 中特定的位置中去．读取数据的时候，如果目标 BLOCK 是坏块，就通过替换表找到其对应的替换 BLOCK．然后从替换 BLOCK 中读取相应的数据．

在实际使用中，坏块管理也没有统一的实现方式．具体选择都是由用户或是应用决定的．很多时候，象 ECC 校验一样，使用的管理方式取决于使用的 BOOTLOADER 及其具体实现．