Booting ARM Linux

Vincent Sanders <vince@arm.linux.org.uk>

Review and advice, large chunks of the ARM Linux kernel, all around good guy: Russell King

Review, advice and numerous clarifications.: Nicolas Pitre

Review and advice: Erik Mouw, Zwane Mwaikambo, Jeff Sutherland, Ralph Siemsen, Daniel Silverstone, Martin Michlmayr, Michael Stevens, Lesley Mitchell, Matthew Richardson

Review and referenced information (see bibliography): Wookey

2004-06-04

Revision History

Revision 1.00	10th May 2004	VRS
	Initial Release.	
Revision 1.10	4th June 2004	VRS
	Update example code to be more complete. Improve wording in places, changes suggested by Nicolas Pitre. Update Section 2, “Other bootloaders”. Update acknowledgements.	

Table of Contents

Abstract

This document defines in clear concise terms, with implementation guidance and examples, the requirements and procedures for a bootloader to start an ARM Linux kernel.

1. About this document

This document describes the "new" booting procedure which all version 2.4.18 and later kernels use. The legacy "struct" method must *not* be used.

This document contains information from a wide variety of sources (see the Bibliography) and authors, you are encouraged to consult these sources for more information before asking questions of the Maintainers, or on the ARM Linux mailing lists. Most of these areas have been covered repeatedly in the past and you are likely to be ignored if you haven't done at least basic research.

Additionally it should be noted that provided the guidance in this document is followed, there should be no need for an implementor to understand every nuance of the assembler that starts the kernel. Experience has shown on numerous occasions that most booting problems are unlikely to be related to this code, said code is also quite tricky and unlikely to give any insight into the problem.

2. Other bootloaders

Before embarking on writing a new bootloader a developer should consider if one of the existing loaders is appropriate. There are examples of loaders in most areas, from simple GPL loaders to full blown commercial offerings. A short list is provided here but the documents in the Bibliography offer more solutions.

Table 1. Bootloaders

Name	URL	Description
Blob	Blob bootloader [http://www.sf.net/projects/blob/]	GPL bootloader for SA11x0 (StrongARM) platforms.
Bootldr	Bootldr [http://www.handhelds.org/sources.html]	Both GPL and non-GPL versions available, mainly used for handheld devices.
Redboot	Redboot [http://sources.redhat.com/redboot/]	Redhat loader released under their eCos licence.
U-Boot	U-Boot [http://sourceforge.net/projects/u-boot/]	GPL universal bootloader, provides support for several CPUs.
ABLE	ABLE bootloader [http://www.simtec.co.uk/products/SWABLE/]	Commercial bootloader with comprehensive feature set

3. Overview

ARM Linux cannot be started on a machine without a small amount of machine specific code to initialise the system. ARM Linux *requires* the bootloader code to do very little, although several bootloaders do provide extensive additional functionality. The minimal requirements are:

Configure the memory system.
Load the kernel image at the correct memory address.
Optionally load an initial RAM disk at the correct memory address.
Initialise the boot parameters to pass to the kernel.

Obtain the ARM Linux machine type
Enter the kernel with the appropriate register values.

It is usually expected that the bootloader will initialise a serial or video console for the kernel in addition to these basic tasks. Indeed a serial port is almost considered mandatory in most system configurations.

Each of these steps will be examined in the following sections.

4. Configuring the system's memory

The bootloader is expected to find and initialise all RAM that the kernel will use for volatile data storage in the system. It performs this in a machine dependent manner. It may use internal algorithms to automatically locate and size all RAM, or it may use knowledge of the RAM in the machine, or any other method the bootloader designer sees fit.

In all cases it should be noted that all setup is performed by the bootloader. The kernel should have no knowledge of the setup or configuration of the RAM within a system other than that provided by the bootloader. The use of machine_fixup() within the kernel is most definitely not the correct place for this. There is a clear distinction between the bootloaders responsibility and the kernel in this area.

The physical memory layout is passed to the kernel using the ATAG_MEM parameter. Memory does not necessarily have to be completely contiguous, although the minimum number of fragments is preferred. Multiple ATAG_MEM blocks allow for several memory regions. The kernel will coalesce blocks passed to it if they are contiguous physical regions.

The bootloader may also manipulate the memory with the kernels command line, using the 'mem=' parameter, the options for this parameter are fully documented in `linux/Documentation/kernel-parameters.txt`

The kernel command line 'mem=' has the syntax `mem=<size>[KM][,@<phys_offset>]` which allows the size and physical memory location for a memory area to be defined. This allows for specifying multiple discontigous memory blocks at differing offsets by providing the mem= parameter multiple times.

5. Loading the kernel image

Kernel images generated by the kernel build process are either uncompressed "Image" files or compressed zImage files.

The uncompressed Image files are generally not used, as they do not contain a readily identifiable magic number. The compressed zImage format is almost universally used in preference.

The zImage has several benefits in addition to the magic number. Typically, the decompression of the image is *faster* than reading from some external media. The integrity of the image can be assured, as any errors will result in a failed decompress. The kernel has knowledge of its internal structure and state, which allows for better results than a generic external compression method.

The zImage has a magic number and some useful information near its beginning.

Table 2. Useful fields in zImage head code

Offset into zImage	Value	Description
0x24	0x016F2818	Magic number used to identify this is an ARM Linux zImage
0x28	start address	The address the zImage starts at
0x2C	end address	The address the zImage ends at

The start and end offsets can be used to determine the length of the compressed image (size = end - start). This is used by several bootloaders to determine if any data is appended to the kernel image. This data is typically used for an initial RAM disk (initrd). The start address is usually 0 as the zImage code is position independent.

The zImage code is Position Independent Code (PIC) so may be loaded anywhere within the available address space. The maximum kernel size after decompression is 4Megabytes. This is a hard limit and would include the initrd if a bootpImage target was used.

Note

Although the zImage may be located anywhere, care should be taken. Starting a compressed kernel requires additional memory for the image to be uncompressed into. This space has certain constraints.

The zImage decompression code will ensure it is not going to overwrite the compressed data. If the kernel detects such a conflict it will uncompress the image immediately *after* the compressed zImage data and relocate the kernel after decompression. This obviously has the impact that the memory region the zImage is loaded into *must* have up to 4Megabytes of space after it (the maximum uncompressed kernel size), i.e. placing the zImage in the same 4Megabyte bank as its ZRELADDR would probably not work as expected.

Despite the ability to place zImage anywhere within memory, convention has it that it is loaded at the base of physical RAM plus an offset of 0x8000 (32K). This leaves space for the parameter block usually placed at offset 0x100, zero page exception vectors and page tables. This convention is *very* common.

6. Loading an initial RAM disk

An initial RAM disk is a common requirement on many systems. It provides a way to have a root filesystem available without access to other drivers or configurations. Full details can be obtained from `linux/Documentation/initrd.txt`

There are two methods available on ARM Linux to obtain an initial RAM disk. The first is a special build target bootpImage which takes an initial RAM disk at *build* time and appends it to a zImage. This method has the benefit that it needs no bootloader intervention, but requires the kernel build process to have knowledge of the physical address to place the ramdisk (using the INITRD_PHYS definition). The hard size limit for the uncompressed kernel and initrd of 4Megabytes applies. Because of these limitations this target is rarely used in practice.

The second and much more widely used method is for the bootloader to place a given initial ramdisk image, obtained from whatever media, into memory at a set location. This location is passed to the kernel using ATAG_INITRD2 and ATAG_RAMDISK.

Conventionally the initrd is placed 8Megabytes from the base of physical memory. Wherever it is placed there must be sufficient memory after boot to decompress the initial ramdisk into a real ramdisk i.e. enough memory for zImage + decompressed zImage + initrd + uncompressed ramdisk. The compressed initial ramdisk memory will be freed after the decompression has happened. Limitations to the position of the ramdisk are:

It must lie completely within a single memory region (must not cross between areas defined by different ATAG_MEM parameters)
It must be aligned to a page boundary (typically 4k)
It must not conflict with the memory the zImage head code uses to decompress the kernel or it *will* be overwritten as no checking is performed.

7. Initialising a console

A console is highly recommended as a method to see what actions the kernel is performing when initialising a system. This can be any input output device with a suitable driver, the most common cases are a video framebuffer driver or a serial driver. Systems that ARM Linux runs on tend to almost always provide a serial console port.

The bootloader should initialise and enable one serial port on the target.This includes enabling any hardware power management etc., to use the port. This allows the kernel serial driver to automatically detect which serial port it should use for the kernel console (generally used for debugging purposes, or communication with the target.)

As an alternative, the bootloader can pass the relevant 'console=' option to the kernel, via the command line parameter specifying the port, and serial format options as described in `linux/Documentation/kernel-parameters.txt`

8. Kernel parameters

The bootloader must pass parameters to the kernel to describe the setup it has performed, the size and shape of memory in the system and, optionally, numerous other values.

The tagged list should conform to the following constraints

The list must be stored in RAM and placed in a region of memory where neither the kernel decompresser nor initrd manipulation will overwrite it. The recommended placement is in the first 16KiB of RAM, usually the start of physical RAM plus 0x100 (which avoids zero page exception vectors).
The physical address of the tagged list must be placed in R2 on entry to the kernel, however historically this has not been mandatory and the kernel has used the fixed value of the start of physical RAM plus 0x100. This must *not* be relied upon in the future.
The list must not extend past the 0x4000 boundary where the kernel's initial translation page table is created. The kernel performs no bounds checking and will overwrite the parameter list if it does so.
The list must be aligned to a word (32 bit, 4byte) boundary (if not using the recommended location)
The list must begin with an ATAG_CORE and end with ATAG_NONE
The list must contain at least one ATAG_MEM

Each tag in the list consists of a header containing two unsigned 32 bit values, the size of the tag (in 32 bit, 4 byte words) and the tag value

```
struct atag_header {
        u32 size; /* legth of tag in words including this header */
        u32 tag;  /* tag value */
};
```

Each tag header is followed by data associated with that tag, excepting ATAG_NONE which has no data and ATAG_CORE where the data is optional. The size of the data is determined by the size field in header, the minimum size is 2 as the headers size is included in this value. The ATAG_NONE is unique in that its size field is set to zero.

A tag may contain additional data after the mandated structures provided the size is adjusted to cover the extra information, this allows for future expansion and for a bootloader to extend the data provided to the kernel. For example a bootloader may provide additional serial number information in an ATAG_SERIAL which could them be interpreted by a modified kernel.

The order of the tags in the parameter list is unimportant, they may appear as many times as required although interpretation of duplicate tags is tag dependant.

The data for each individual tag is described in the Appendix A, *Tag Reference* section.

Table 3. List of usable tags

Tag name	Value	Size	Description
ATAG_NONE	0x00000000	2	Empty tag used to end list
ATAG_C	0x54410001	5 (2 if	First tag used to start list

Tag name	Value	Size	Description
ORE		empty)	
ATAG_MEM	0x54410002	4	Describes a physical area of memory
ATAG_VIDEOTEXT	0x54410003	5	Describes a VGA text display
ATAG_RAMDISK	0x54410004	5	Describes how the ramdisk will be used in kernel
ATAG_INITRD2	0x54420005	4	Describes where the compressed ramdisk image is placed in memory
ATAG_SERIAL	0x54410006	4	64 bit board serial number
ATAG_REVISION	0x54410007	3	32 bit board revision number
ATAG_VIDEOLFB	0x54410008	8	Initial values for vesafb-type framebuffers
ATAG_CMDLINE	0x54410009	2 + ((length_of_cmdline + 3) / 4)	Command line to pass to kernel

For implementation purposes a structure can be defined for a tag

```
struct atag {
        struct atag_header hdr;
        union {
                struct atag_core         core;
                struct atag_mem          mem;
                struct atag_videotext    videotext;
                struct atag_ramdisk      ramdisk;
                struct atag_initrd2      initrd2;
                struct atag_serialnr     serialnr;
                struct atag_revision     revision;
                struct atag_videolfb     videolfb;
                struct atag_cmdline      cmdline;
        } u;
};
```

Once these structures have been defined an implementation needs to create the list this can be implemented with code similar to

```
#define tag_next(t)     ((struct tag *)((u32 *)(t) + (t)->hdr.size))
#define tag_size(type)  ((sizeof(struct tag_header) + sizeof(struct type)) >> 2
static struct atag *params; /* used to point at the current tag */

static void
setup_core_tag(void * address,long pagesize)
{
    params = (struct tag *)address;         /* Initialise parameters to start a

    params->hdr.tag = ATAG_CORE;            /* start with the core tag */
    params->hdr.size = tag_size(atag_core); /* size the tag */

    params->u.core.flags = 1;               /* ensure read-only */
    params->u.core.pagesize = pagesize;     /* systems pagesize (4k) */
    params->u.core.rootdev = 0;             /* zero root device (typicaly overi

    params = tag_next(params);              /* move pointer to next tag */
}

static void
```

```
setup_mem_tag(u32_t start, u32_t len)
{
    params->hdr.tag = ATAG_MEM;             /* Memory tag */
    params->hdr.size = tag_size(atag_mem);  /* size tag */

    params->u.mem.start = start;            /* Start of memory area (physical a
    params->u.mem.size = len;               /* Length of area */

    params = tag_next(params);              /* move pointer to next tag */
}

static void
setup_end_tag(void)
{
    params->hdr.tag = ATAG_NONE;            /* Empty tag ends list */
    params->hdr.size = 0;                   /* zero length */
}


static void
setup_tags(void)
{
    setup_core_tag(0x100, 4096);            /* standard core tag 4k pagesize */
    setup_mem_tag(0x10000000, 0x400000);    /* 64Mb at 0x10000000 */
    setup_mem_tag(0x18000000, 0x400000);    /* 64Mb at 0x18000000 */
    setup_end_tag(void);                    /* end of tags */
}
```

While this code fragment is complete it illustrates the absolute minimal requirements for a parameter set and is intended to demonstrate the concepts expressed earlier in this section. A real bootloader would probably pass additional values and would probably probe for the memory actually in a system rather than using fixed values. A more complete example can be found in Appendix B, *Complete example*

9. Obtaining the ARM Linux machine type

The only additional information the bootloader needs to provide is the machine type, this is a simple number unique for each ARM system often referred to as a MACH_TYPE.

The machine type number is obtained via the ARM Linux website Machine Registry [http://www.arm.linux.org.uk/developer/machines/]. A machine type should be obtained as early in a projects life as possible, it has a number of ramifications for the kernel port itself (machine definitions etc.) and changing definitions afterwards may lead to a number of undesirable issues. These values are represented by a list of defines within the kernel source (linux/arch/arm/tools/mach-types)

The boot loader must obtain the machine type value by some method. Whether this is a hard coded value or an algorithm that looks at the connected hardware. Implementation is completely system specific and is beyond the scope of this document.

10. Starting the kernel

Once the bootloader has performed all the other steps it must start execution of the kernel with the correct values in the CPU registers.

The entry requirements are:

The CPU must be in SVC (supervisor) mode with both IRQ and FIQ interrupts disabled.
The MMU must be *off*, i.e. code running from physical RAM with no translated addressing.
Data cache must be *off*
Instruction cache may be either on or off
CPU register 0 must be 0
CPU register 1 must be the ARM Linux machine type

CPU register 2 must be the physical address of the parameter list

The bootloader is expected to call the kernel image by jumping directly to the first instruction of the kernel image.

Tag Reference

ATAG_CORE

ATAG_CORE -- Start tag used to begin list

ATAG_CORE

Value

0x54410001

Size

5 (2 if no data)

Structure members

```
struct atag_core {
        u32 flags;              /* bit 0 = read-only */
        u32 pagesize;           /* systems page size (usually 4k) */
        u32 rootdev;            /* root device number */
};
```

Description

This tag *must* be used to start the list, it contains the basic information any bootloader must pass, a tag length of 2 indicates the tag has no structure attached.

ATAG_NONE

ATAG_NONE -- Empty tag used to end list

ATAG_NONE

Value

0x00000000

Size

2

Structure members

None

Description

This tag is used to indicate the list end. It is unique in that its size field in the header should be set to 0 (not 2).

ATAG_MEM

ATAG_MEM -- Tag used to describe a physical area of memory.

ATAG_MEM

Value

0x54410002

Size

4

Structure members

```
struct atag_mem {
        u32     size;   /* size of the area */
        u32     start;  /* physical start address */
};
```

Description

Describes an area of physical memory the kernel is to use.

ATAG_VIDEOTEXT

ATAG_VIDEOTEXT -- Tag used to describe VGA text type displays

ATAG_VIDEOTEXT

Value

0x54410003

Size

5

Structure members

```
struct atag_videotext {
        u8              x;           /* width of display */
        u8              y;           /* height of display */
        u16             video_page;
        u8              video_mode;
        u8              video_cols;
        u16             video_ega_bx;
        u8              video_lines;
        u8              video_isvga;
        u16             video_points;
};
```

Description

ATAG_RAMDISK

ATAG_RAMDISK -- Tag describing how the ramdisk will be used by the kernel

ATAG_RAMDISK

Value

0x54410004

Size

5

Structure members

```
struct atag_ramdisk {
        u32 flags;      /* bit 0 = load, bit 1 = prompt */
        u32 size;       /* decompressed ramdisk size in _kilo_ bytes */
        u32 start;      /* starting block of floppy-based RAM disk image */
};
```

Description

Describes how the (initial) ramdisk will be configured by the kernel, specifically this allows for the bootloader to ensure the ramdisk will be large enough to take the *decompressed* initial ramdisk image the bootloader is passing using ATAG_INITRD2.

ATAG_INITRD2

ATAG_INITRD2 -- Tag describing the physical location of the compressed ramdisk image

ATAG_INITRD2

Value

0x54420005

Size

4

Structure members

```
struct atag_initrd2 {
        u32 start;      /* physical start address */
        u32 size;       /* size of compressed ramdisk image in bytes */
};
```

Description

Location of a compressed ramdisk image, usually combined with an ATAG_RAMDISK. Can be used as an initial root file system with the addition of a command line parameter of 'root=/dev/ram'. This tag *supersedes* the original ATAG_INITRD which used virtual addressing, this was a mistake and produced issues on some systems. All new bootloaders should use this tag in preference.

ATAG_SERIAL

ATAG_SERIAL -- Tag with 64 bit serial number of the board

ATAG_SERIAL

Value

0x54410006

Size

4

Structure members

```
struct atag_serialnr {
        u32 low;
        u32 high;
};
```

Description

ATAG_REVISION

ATAG_REVISION -- Tag for the board revision

ATAG_REVISION

Value

0x54410007

Size

3

Structure members

```
struct atag_revision {
        u32 rev;
};
```

Description

ATAG_VIDEOLFB

ATAG_VIDEOLFB -- Tag describing parameters for a framebuffer type display

ATAG_VIDEOLFB

Value

0x54410008

Size

8

Structure members

```
struct atag_videolfb {
        u16             lfb_width;
        u16             lfb_height;
        u16             lfb_depth;
        u16             lfb_linelength;
        u32             lfb_base;
        u32             lfb_size;
        u8              red_size;
        u8              red_pos;
        u8              green_size;
        u8              green_pos;
        u8              blue_size;
        u8              blue_pos;
        u8              rsvd_size;
        u8              rsvd_pos;
};
```

Description

ATAG_CMDLINE

ATAG_CMDLINE -- Tag used to pass the commandline to the kernel

ATAG_CMDLINE

Value

0x54410009

Size

2 + ((length_of_cmdline + 3) / 4)

Structure members

```
struct atag_cmdline {
        char    cmdline[1];     /* this is the minimum size */
};
```

Description

Used to pass command line parameters to the kernel. The command line must be NULL terminated. The length_of_cmdline variable should include the terminator.

Complete example

This is a worked example of a simple bootloader and shows all the information explained throughout this document. More code would be required for a real bootloader this example is purely illustrative.

The code in this example is distributed under a BSD licence, it may be freely copied and used if necessary.

```
/* example.c
 * example ARM Linux bootloader code
 * this example is distributed under the BSD licence
 */

/* list of possible tags */
#define ATAG_NONE       0x00000000
#define ATAG_CORE       0x54410001
#define ATAG_MEM        0x54410002
#define ATAG_VIDEOTEXT  0x54410003
#define ATAG_RAMDISK    0x54410004
#define ATAG_INITRD2    0x54420005
#define ATAG_SERIAL     0x54410006
#define ATAG_REVISION   0x54410007
#define ATAG_VIDEOLFB   0x54410008
#define ATAG_CMDLINE    0x54410009

/* structures for each atag */
struct atag_header {
        u32 size; /* length of tag in words including this header */
        u32 tag;  /* tag type */
};
```

```
struct atag_core {
        u32 flags;
        u32 pagesize;
        u32 rootdev;
};

struct atag_mem {
        u32     size;
        u32     start;
};

struct atag_videotext {
        u8              x;
        u8              y;
        u16             video_page;
        u8              video_mode;
        u8              video_cols;
        u16             video_ega_bx;
        u8              video_lines;
        u8              video_isvga;
        u16             video_points;
};

struct atag_ramdisk {
        u32 flags;
        u32 size;
        u32 start;
};

struct atag_initrd2 {
        u32 start;
        u32 size;
};

struct atag_serialnr {
        u32 low;
        u32 high;
};

struct atag_revision {
        u32 rev;
};

struct atag_videolfb {
        u16             lfb_width;
        u16             lfb_height;
        u16             lfb_depth;
        u16             lfb_linelength;
        u32             lfb_base;
        u32             lfb_size;
        u8              red_size;
        u8              red_pos;
        u8              green_size;
        u8              green_pos;
        u8              blue_size;
        u8              blue_pos;
        u8              rsvd_size;
        u8              rsvd_pos;
};

struct atag_cmdline {
        char    cmdline[1];
};

struct atag {
        struct atag_header hdr;
        union {
                struct atag_core         core;
                struct atag_mem          mem;
```

```
                struct atag_videotext    videotext;
                struct atag_ramdisk      ramdisk;
                struct atag_initrd2      initrd2;
                struct atag_serialnr     serialnr;
                struct atag_revision     revision;
                struct atag_videolfb     videolfb;
                struct atag_cmdline      cmdline;
        } u;
};


#define tag_next(t)     ((struct tag *)((u32 *)(t) + (t)->hdr.size))
#define tag_size(type)  ((sizeof(struct tag_header) + sizeof(struct type)) >> 2
static struct atag *params; /* used to point at the current tag */

static void
setup_core_tag(void * address,long pagesize)
{
    params = (struct tag *)address;         /* Initialise parameters to start a

    params->hdr.tag = ATAG_CORE;            /* start with the core tag */
    params->hdr.size = tag_size(atag_core); /* size the tag */

    params->u.core.flags = 1;               /* ensure read-only */
    params->u.core.pagesize = pagesize;     /* systems pagesize (4k) */
    params->u.core.rootdev = 0;             /* zero root device (typicaly overi

    params = tag_next(params);              /* move pointer to next tag */
}

static void
setup_ramdisk_tag(u32_t size)
{
    params->hdr.tag = ATAG_RAMDISK;         /* Ramdisk tag */
    params->hdr.size = tag_size(atag_ramdisk);  /* size tag */

    params->u.ramdisk.flags = 0;            /* Load the ramdisk */
    params->u.ramdisk.size = size;          /* Decompressed ramdisk size */
    params->u.ramdisk.start = 0;            /* Unused */

    params = tag_next(params);              /* move pointer to next tag */
}

static void
setup_initrd2_tag(u32_t start, u32_t size)
{
    params->hdr.tag = ATAG_INITRD2;         /* Initrd2 tag */
    params->hdr.size = tag_size(atag_initrd2);  /* size tag */

    params->u.initrd2.start = start;        /* physical start */
    params->u.initrd2.size = size;          /* compressed ramdisk size */

    params = tag_next(params);              /* move pointer to next tag */
}

static void
setup_mem_tag(u32_t start, u32_t len)
{
    params->hdr.tag = ATAG_MEM;             /* Memory tag */
    params->hdr.size = tag_size(atag_mem);  /* size tag */

    params->u.mem.start = start;            /* Start of memory area (physical a

    params->u.mem.size = len;               /* Length of area */

    params = tag_next(params);              /* move pointer to next tag */
}

static void
setup_cmdline_tag(const char * line)
```

```
{
    int linelen = strlen(line);

    if(!linelen)
        return;                              /* do not insert a tag for an empty

    params->hdr.tag = ATAG_CMDLINE;          /* Commandline tag */
    params->hdr.size = (sizeof(struct atag_header) + linelen + 1 + 4) >> 2;

    strcpy(params->u.cmdline.cmdline,line); /* place commandline into tag */

    params = tag_next(params);               /* move pointer to next tag */
}

static void
setup_end_tag(void)
{
    params->hdr.tag = ATAG_NONE;             /* Empty tag ends list */
    params->hdr.size = 0;                    /* zero length */
}


#define DRAM_BASE 0x10000000
#define ZIMAGE_LOAD_ADDRESS DRAM_BASE + 0x8000
#define INITRD_LOAD_ADDRESS DRAM_BASE + 0x800000

static void
setup_tags(parameters)
{
    setup_core_tag(parameters, 4096);       /* standard core tag 4k pagesize */
    setup_mem_tag(DRAM_BASE, 0x4000000);    /* 64Mb at 0x10000000 */
    setup_mem_tag(DRAM_BASE + 0x8000000, 0x4000000); /* 64Mb at 0x18000000 */
    setup_ramdisk_tag(4096);                 /* create 4Mb ramdisk */
    setup_initrd2_tag(INITRD_LOAD_ADDRESS, 0x100000); /* 1Mb of compressed data
    setup_cmdline_tag("root=/dev/ram0");    /* commandline setting root device
    setup_end_tag(void);                     /* end of tags */
}

int
start_linux(char *name,char *rdname)
{
    void (*theKernel)(int zero, int arch, u32 params);
    u32 exec_at = (u32)-1;
    u32 parm_at = (u32)-1;
    u32 machine_type;

    exec_at = ZIMAGE_LOAD_ADDRESS;
    parm_at = DRAM_BASE + 0x100

    load_image(name, exec_at);               /* copy image into RAM */

    load_image(rdname, INITRD_LOAD_ADDRESS);/* copy initial ramdisk image into

    setup_tags(parm_at);                     /* sets up parameters */

    machine_type = get_mach_type();          /* get machine type */

    irq_shutdown();                          /* stop irq */

    cpu_op(CPUOP_MMUCHANGE, NULL);           /* turn MMU off */

    theKernel = (void (*)(int, int, u32))exec_at; /* set the kernel address */

    theKernel(0, machine_type, parm_at);    /* jump to kernel with register set

    return 0;
}
```

Bibliography

ARM Linux website Documentation [http://www.arm.linux.org.uk/developer/]. Russell M King.

Linux Kernel Documentation/arm/booting.txt [http://www.arm.linux.org.uk/developer/booting.php]. Russell M King.

Setting R2 correctly for booting the kernel [http://lists.arm.linux.org.uk/pipermail/linux-arm-kernel/2003-January/013126.html] (explanation of booting requirements). Russell M King.

Wookey's post summarising booting [http://lists.arm.linux.org.uk/pipermail/linux-arm-kernel/2002-April/008700.html]. Wookey.

Makefile defines and symbols [http://lists.arm.linux.org.uk/pipermail/linux-arm-kernel/2001-July/004064.html]. Russell M King.

Bootloader guide [http://www.aleph1.co.uk/armlinux/docs/ARMbooting/t1.html]. Wookey.

Kernel boot order [http://lists.arm.linux.org.uk/pipermail/linux-arm-kernel/2001-October/005212.html]. Russell M King.

Advice for head.S Debugging [http://lists.arm.linux.org.uk/pipermail/linux-arm-kernel/2002-January/006730.html]. Russell M King.

Linux kernel 2.4 startup [http://billgatliff.com/articles/emb-linux/startup.html/index.html]. Bill Gatliff.

Blob bootloader [http://www.sf.net/projects/blob/]. Erik Mouw.

Blob bootloader on lart [http://www.lart.tudelft.nl/lartware/blob/]. Erik Mouw.