Freescale Semiconductor
Application Note

Document Number: AN3984
Rev. 0, 02/2010

i.MX35 Board Initialization and Memory Mapping Using the Linux Target Image Builder (LTIB)

by *Multimedia Applications Division*
Freescale Semiconductor, Inc.
Austin, TX

This application note provides general information regarding the board initialization process and the memory mapping implementation of the Linux kernel using the LTIB in an i.MX35 Board Support Package (BSP).

The board initialization process is relatively complex and long. Hence this application note provides a general overview of board initialization process, while explaining more about the memory mapping.

The knowledge on these aspects enable better understanding of the BSP structure. When changes such as migrations to another board or device with different memories and external board chips are needed, these are some of the elements that need to be changed on the software side.

This application note is targeted to the i.MX35 BSPs, but it is applicable to any i.MX device. The structure and architecture of the system (software) is the same for all the i.MX BSPs.

This application note covers information, and the initialization process flow of a BSP running a LTIB, on an i.MX35 platform. The focus of this application note is on memory elements and memory mapping, from bootloader startup to kernel initialization.

Contents

freescale semiconductor

The first section of this application note explains the general objectives of a standard bootloader, and how to pass information about the system memory (such as the size and start address) to the kernel.

The second section explains the board initialization function. It also explains the elements that are required to perform a board initialization (such as structures, linker sections, and functions), and their place within the flow of the Linux booting process.

The last section describes the general aspects of memory mapping on the system and the implementation of the memory map in the BSP.

1 Linux Booting Process

This section describes the main objectives of the bootloader of a Linux system for an ARM device. It also describes the structures and the flows that the bootloader needs to pass to the kernel.

1.1 General Bootloader Objectives

Even though there are many possibilities, such as loading an initial RAM, this application note explains only the basic function of the bootloader. There are five minimum steps that any bootloader needs to follow:

1. Setup and initialize the system RAM—The bootloader finds and initializes the entire RAM to provide the volatile data storage for the system. The algorithm that is used to locate and set up the RAM depends on the processor and bootloader designs.
2. Initialize one serial port (optional, but highly recommended)—The bootloader locates and enables a serial port on the target. This allows the kernel serial driver to detect the serial port that is used later as the kernel console. The bootloader passes the `console=` expression as a kernel parameter, which is recognized as a part of the tagged list.
3. Detect the machine type—The bootloader detects the type of processor that is running on the system. This information is a macro with a name in the form `MACH_TYPE_XXX`.
4. Setup the kernel tagged list—The bootloader creates and initializes a kernel tagged list that contains the information such as the size and location of the system memory (RAM). Other elements such as RAM disk creation or a console value are added to the tagged list. The tagged list concept and its characteristics are described in the Section 1.2, “Tags in the Linux Booting Process.”
5. Call or start the kernel image—The bootloader finally calls the kernel image (a compressed zImage), depending on where the zImage is stored. It is possible to call the zImage in Flash directly or store the zImage in RAM and call there. The following are some of the conditions that are to be set to call the zImage:
   a) Set the CPU in supervisor mode with IRQ disabled.
   b) Turn off the Memory Management Unit (MMU) and Data Cache. The code running in RAM does not have translated addressing yet.
   c) Set the register r0 to 0, r1 to the ARM Linux machine type, and r2 to the physical address of the parameter list (tagged list).

The bootloader is in charge of initializing the process while configuring the RAM for the system. The kernel does not have knowledge of the RAM configuration beyond what is provided by the bootloader. If there is a need to change the RAM of a system, most of the software changes apply to the bootloader.

A small exception is the usage of the machine fixup function (`fixup_mxc_board`). It is normally stored in the machine dependant code (`linux-2.6.26/arch/arm/mach-mx35/mx35_3stack.c`) and it is used inside the kernel to enable some actions of memory configuration that belong to the bootloader. It allows the user to statically fill the values of parameters such as memory data.

NOTE

Under normal circumstances, the bootloader fills the values of parameters. But fixup exists to allow flexibility for exceptions.

1.2 Tags in the Linux Booting Process

The following section provides a more detailed description of the tagged list, with emphasis on the memory tag.

1.2.1 Tags in the Bootloader Environment

The tagged list contains the information of the physical layout. The information is passed to the kernel with the ATAG_MEM parameter. This parameter is a part of the tagged list that is passed from the bootloader to the kernel. The value is overriden through the kernel command line parameter `mem=`, and by using this, the bootloader passes the size of the physical memory.

NOTE

For more information on the syntax of the command line parameters, see the documentation located at:
`linux-2.6.26/Documentation/kernel-parameters.txt`

ATAG_MEM is one among a set of parameters passed by the bootloader to the kernel. The parameters create a list (tagged list) that contains information, such as the command line tag associated with the kernel command line string, serial console information, RAM disk usage, or initial configuration values for the framebuffer. This tagged list (ATAG) is implemented as a structure and is stored in main memory. The address of the structure is passed to the register r2 when starting the kernel. However, in many cases, the kernel finds it in a fixed memory location (by default, both the bootloader and the kernel know where it is).

The following are the most important constraints in the tagged list:

- The list is stored in a safe place in RAM. The recommended place is the first 16 Kbytes of RAM.

NOTE

The list should not be stored where the kernel is decompressed, and where the initrd overwrite it.

- The list must not extend beyond the 0x4000 boundary where the kernel initial translation page table is created.
- The list is aligned to a 4 byte boundary.

- The list begins with a tag ATAG_CORE, contain a tag ATAG_MEM and end with a tag ATAG_NONE

Each tag in the list contains a `tag_header` structure that sets the size of the tag, and a tag value that represents the tag type. In almost all cases, each tag header has more data associated with the type of tag (except for ATAG_NONE).

Example 1 shows a section of the tag structure containing `tag_header` and several different types of tags, implemented as a union of structures. This structure is from the kernel location: `linux-2.6.26/include/asm-arm/setup.h`. However, the bootloader contains a definition very similar to the following lines of code:

Example 1. Tag Structure

```
struct tag {
        struct tag_header hdr;
        union {
                struct tag_core           core;
                struct tag_mem32          mem;
                struct tag_videotext      videotext;
                struct tag_ramdisk        ramdisk;
                struct tag_initrd         initrd;
                struct tag_serialnr       serialnr;
                struct tag_revision       revision;
                struct tag_videolfb       videolfb;
                struct tag_cmdline        cmdline;
                (…)
        } u;
};
```

The data associated with each tag (in the union part of the tag structure) contains the information related to each type. For example, in the case of the `tag_mem` (ATAG_MEM), the data is described in the union as the `tag_mem` structure (`tag_mem32` in the kernel example). This structure contains two fields, one for the size of the memory represented in this tag, and another for the physical start addresses of this memory.

The following lines of code are from the `linux-2.6.26/include/asm-arm/setup.h` file. The code contains some definitions of tags, such as `tag_mem`, and the `tag_header`. The bootloader should have an implementation of tags similar to the following lines of code:

```
/* The list ends with an ATAG_NONE node. */

#define ATAG_NONE0x00000000

struct tag_header {
        __u32 size;
        __u32 tag;
}

/* The list must start with an ATAG_CORE node */

#define ATAG_CORE0x54410001

struct tag_core {
        __u32 flags;      /* bit 0 = read-only */
        __u32 pagesize;
```

```
        __u32 rootdev;
};

/* it is allowed to have multiple ATAG_MEM nodes */

#define ATAG_MEM0x54410002

struct tag_mem32 {
        __u32    size;
        __u32    start;   /* physical start address */
};
```

1.2.2 Tags in the Kernel Environment

From the kernel standpoint, it is important to know where the tags are retrieved from, and used for the kernel internal settings. The system needs tag structures similar to the ones from the bootloader to enable this feature. All the tag structures definitions are provided in `linux-2.6.26/include/asm-arm/setup.h`.

The specific function for the tag retrieving process is `void __init setup_arch(char **cmdline_p)`, (from the file `linux2.6.26/arch/arm/kernel/setup.c`). This function is called from the function `asmlinkage void __init start_kernel(void)` (from the file `linux-2.6.26/init/main.c`).

The `start_kernel` function is called after all the assembly-oriented section of the kernel initialization is executed. This process involves the files related to the compressed kernel stage (zImage). After the function is called, the kernel relocation follows, and finally, the uncompressed kernel startup (`head-armv.S` file).

The kernel gets the memory configuration information in the following two ways:

- Getting the tagged list that contains the memory tag with the information.
- Getting the information from the kernel command line through the usage of the `mem=` parameter. Both cases are described and implemented in the file: `linux-2.6.26/arch/arm/kernel/setup.c`.

1.2.2.1 Retrieving Tag Information

The tag table is built by the linker using the `__tagtable` declarations of each tag in the `linux-2.6.26/arch/arm/kernel/setup.c` file. One of these declarations is the memory tag as follows:

```
__tagtable(ATAG_MEM, parse_tag_mem32)
```

The definition of this line of code is found in the file `linux-2.6.26/include/asm-arm/setup.h`, and its meaning is:

```
#define __tag __used __attribute__((__section__(".taglist.init")))
#define __tagtable(tag, fn) \
static struct tagtable __tagtable_##fn __tag = { tag, fn }
```

The result of this declaration is a `struct tagtable`. The structure is formed by a `__32` number and a pointer to a function that has:

- A name `__tagtable_##fn` (a concatenation with the name of the function).
- The attribute that assembly functions of code related to this declaration is placed in the section `.taglist.init` that is defined by the linker (see `vmlinux.lds`), instead of the common text section.

- The parameter values: `tag` (_u32 number, in this case ATAG_MEM) and `fn` (a pointer to a function that in this case is the `parse_tag_mem32` function)

Each declaration generates one `struct tagtable` as the following:

The tag list is retrieved in `setup_arch` (found in `setup.c`). This function has a call to the `parse_tags` function. The function `parse_tags` parses all the tags contained on the list, by using another function that parses one tag at a time. This last function takes the tags on the list as input, and calls the parse function for each input tag that has a match on the tag table.

```
/*
* Parse all tags in the list, checking both the global and
* architecture specific tag tables.
*/

static void __init parse_tags(const struct tag *t)
{
        for (; t->hdr.size; t = tag_next(t))
                if (!parse_tag(t))
                        printk(KERN_WARNING
                                "Ignoring unrecognised tag 0x%08x\n",
                                t->hdr.tag);
}

*
* Scan the tag table for this tag, and call its parse function.
* The tag table is built by the linker from all the __tagtable
* declarations.
*/

static int __init parse_tag(const struct tag *tag)
{
        extern struct tagtable __tagtable_begin, __tagtable_end;
        struct tagtable *t;
        for (t = &__tagtable_begin; t < &__tagtable_end; t++)
                if (tag->hdr.tag == t->tag) {
                        t->parse(tag);
                        break;
                }
        return t < &__tagtable_end;
}
```

The function has the elements `__tagtable_begin` and `__tagtable_end` defined in `linux-2.6.26/arch/arm/kernel/vmlinux.lds` (linker file). These are the limits of the tag list in memory.

```
.init : { /* Init code and data*/
*(.init.text) *(.cpuinit.text) *(.meminit.text)
_einittext = .;
__proc_info_begin = .;
*(.proc.info.init)
__proc_info_end = .;
__arch_info_begin = .;
*(.arch.info.init)
__arch_info_end = .;
__tagtable_begin = .;
*(.taglist.init)
__tagtable_end = .;
. = ALIGN(16);
```

```
__setup_start = .;
*(.init.setup)
__setup_end = .;
__early_begin = .;
*(.early_param.init)
__early_end = .;
(…)
```

The space between both elements (begin and end) is the data and its attributes (`taglist.init`). The first tag that needs to be parsed and recognized is ATAG_CORE. This is the first tag found according to the established protocol.

A memory tag is also found, when the match is done it calls the parse function associated with the tag. In the case of the memory tag, the parse function is `parse_tag_mem32`.

```
static int __init parse_tag_mem32(const struct tag *tag)
{
        if (meminfo.nr_banks >= NR_BANKS) {
                printk(KERN_WARNING
                       "Ignoring memory bank 0x%08x size %dKB\n",
                         tag->u.mem.start, tag->u.mem.size / 1024);
                return -EINVAL;
        }
        arm_add_memory(tag->u.mem.start, tag->u.mem.size);
        return 0;
}
```

Near the end, `arm_add_memory` is called. This is the function that sets the memory information (start address and size) in the `membank` structure.

```
static void __init arm_add_memory(unsigned long start, unsigned long size)
{
        struct membank *bank;
        /*
         * Ensure that start/size are aligned to a page boundary.
         * Size is appropriately rounded down, start is rounded up.
         */
        size -= start & ~PAGE_MASK;
        bank = &meminfo.bank[meminfo.nr_banks++];
        bank->start = PAGE_ALIGN(start);
        bank->size  = size & PAGE_MASK;
        bank->node  = PHYS_TO_NID(start);
}
```

1.2.2.2 Retrieving Memory Information from the Command Line

The other possibility for retrieving the memory information is getting the data from the kernel command line and the parameter `mem=`. This process is similar to the retrieving of tag information from the tagged list. Some characteristics are:

- Starts in `setup_arch` and finishes in `arm_add_memory`
- Needs a particular section in memory, also set by the `vmlinux.lds` file, but now it is named `early_param.init` and the limits are `__early_begin __early_end`
- Instead of the `parse_tags` function, there is now a `parse_cmdline`

- The content of `early_param.init` is filled by the declaration `__early_param(mem=, early_mem)` using the same elements as `__tagtable` (`__attribute__` and a special section).
- In summary the flow is that the `setup_arch function` calls the `parse_cmdline`. When parsing the command line, and if the `mem=` parameter is found, it calls the `early_mem` function, and the `arm_add_memory` to fill the `membank` structure.

All the code are found in `linux-2.6.26/arch/arm/kernel/setup.c` and definitions in `linux-2.6.26/include/asm-arm/setup.h` (`__early_param` or `early_params` structures).

2 Board Initialization Process

This section explains the process by which the board elements are initialized on a Linux system.

There are some elements that are to be set before the board is initialized. This section describes the elements that are related to the Linux booting process. Some of these are the `machine_desc` structure, or the process that the kernel uses to confirm the CPU and the machine type used in the system (in the current board).

This section also explains briefly the contents of the function related to the board (system) initialization.

2.1 MACHINE_START Description and Flow

The `MACHINE_START` definition is the declaration of the `machine_desc` structure holding the name of the board currently used. Besides having the name of the board in use, it is also set in a particular section declared in the `vmlinux.lds` file. It has the `MACH_TYPE` and the name of the system as parameters. The definition is located in: `linux-2.6.26/include/asm-arm/mach/arch.h`. The `MACH_TYPE` parameter passed in `MACHINE_START` is stored in: `linux-2.6.26/include/asm-arm/mach_types.h`.

```
#define MACHINE_START(_type,_name)\
static const struct machine_desc __mach_desc_##_type\
__used                                                          \
__attribute__((__section__(".arch.info.init"))) = {\
        .nr             = MACH_TYPE_##_type,\
        .name           = _name,

#define MACHINE_END                \
};

#endif
```

The `MACHINE_START` macro becomes a data structure when the compiler builds the file that holds it. Usually this structure is declared in a file where the initialization of the current board is made. This means the file is inside the mach-xxx folder. For this application note, the board used is the i.MX35 PDK. The file where the declaration is made is: `linux-2.6.26/arch/arm/mach-mx35/mx35_3stack.c`.

This macro is defined as the structure that describes the machine, or the board. It contains more members than a name and a type. These members are part of the `machine_desc` structure that is declared with the macro. The definition of the `machine_desc` structure is located in:
`linux-2.6.26/include/asm-arm/mach/arch.h`.

```
struct machine_desc {
/*
```

```
* Note! The first four elements are used
* by assembler code in head.S, head-common.S
*/
        unsigned int     nr;              /* architecture number*/
        unsigned int     phys_io;         /* start of physical io*/
        unsigned int     io_pg_offst;     /* byte offset for io * page tabe entry*/

        const char       *name;                    /* architecture name*/
        unsigned longboot_params;  /* tagged list    */

        unsigned intvideo_start;   /* start of video RAM*/
        unsigned intvideo_end;             /* end of video RAM*/

        unsigned intreserve_lp0 :1;/* never has lp0*/
        unsigned intreserve_lp1 :1;/* never has lp1*/
        unsigned intreserve_lp2 :1;/* never has lp2*/
        unsigned intsoft_reboot :1;/* soft reboot*/

        void     (*fixup)(struct machine_desc *, struct tag *, char **, struct meminfo *);
        void     (*map_io)(void);  /* IO mapping function*/
        void     (*init_irq)(void);
        struct sys_timer*timer;    /* system tick timer*/
        void     (*init_machine)(void);
};
```

As seen, there are several members of the structure, and not all of them are filled in the final `MACHINE_START` declaration. For the current system, the declaration is located in:

`linux-2.6.26/arch/arm/mach-mx35/mx35_3stack.c`:

```
/*
* The following uses standard kernel macros define in arch.h in order to
* initialize __mach_desc_MX35_3DS data structure.
*/
/* *INDENT-OFF* */
MACHINE_START(MX35_3DS, "Freescale MX35 3-Stack Board")
        /* Maintainer: Freescale Semiconductor, Inc. */
        .phys_io = AIPS1_BASE_ADDR,
        .io_pg_offst = ((AIPS1_BASE_ADDR_VIRT) >> 18) & 0xfffc,
        .boot_params = PHYS_OFFSET + 0x100,
        .fixup = fixup_mxc_board,
        .map_io = mxc_map_io,
        .init_irq = mxc_init_irq,
        .init_machine = mxc_board_init,
        .timer = &mxc_timer,
MACHINE_END
```

The data obtained from the declaration are as follows:

- The `MACH_TYPE` and architecture number (`nr`) is: `MACH_TYPE_MX35_3DS`
- The name of the `mach_desc` structure is: `__mach_desc_MX35_3DS`
- The name parameter of the `mach_desc` structure is: `Freescale MX35 3-Stack Board`
- The physical address of the I/O bank is (`phys_io`): `AIPS1_BASE_ADDR`
- The I/O page offset that allows providing virtual memory is (`io_pg_offst`):
  `((AIPS1_BASE_ADDR_VIRT) >> 18) & 0xfffc`

- The boot parameters (address of the tagged list used in the process of retrieving the tagged list for the kernel) are (`boot_params`): `PHYS_OFFSET + 0x100`
- The fixup function reference is (`fixup`): `fixup_mxc_board`
- The I/O memory mapping function is (`map_io`): `mxc_map_io`
- The IRQ initialization function is (`init_irq`): `mxc_init_irq`
- The machine initialization function (board initialization) is (`init_machine`): `mxc_board_init`
- The timer structure is (`timer`): `&mxc_timer`

Some of these elements are extremely important for the development of this application note. For example, the `boot_params` provide the location of the tag structure created in `setup_arch` function covered in the Section 1.2.2.2, "Retrieving Memory Information from the Command Line," which passes the information about the memory layout to the system.

The `init_machine` parameter provides the reference to the function that initializes the system. The objective of this section is to explain how the Linux booting process gets to that function and describe it briefly.

The `map_io` parameter provides the reference to the function for the memory mapping process. This function is explained in the following section.

2.1.1 Recognizing the CPU and Machine

In the Linux kernel boot, after passing the stage of uncompressing the kernel, the kernel initializes the hardware using the `init_machine` parameter. Before initializing the hardware, the kernel validates if it is running on the CPU that it was compiled for. To know if this is true, the kernel gets the processor ID and compares it to the data contained in the `proc.info.init` section (see `vmlinux.lds`). This verification is followed by the initialization of caches and the MMU.

The process is accomplished as follows:

- Function `setup_processor()` is called from `setup_arch()`, (file `linux-2.6.26/arch/arm/kernel/setup.c`).
- `setup_processor` uses function `__lookup_processor_type` (located in: `linux-2.6.26/arch/arm/kernel/head-common.S`) to get the processor ID. The `proc.info` section is filled with information from the file `linux-2.6.26/arch/arm/mm/proc-v6.S`, which sets the `.proc.info.init` section and contains the processor ID information.
- The kernel compares the machine ID given by the bootloader to the kernel with the information contained in `.arch.info.init` section (see `vmlinux.lds`). This also occurs inside `setup_arch()`.
- After `setup_processor` is called, the function `setup_machine(machine_arch_type)` is called (the parameter holds the machine type obtained from the `MACHINE_START` macro declaration). This function reaches `lookup_machine_type` (`nr`) that gets the machine type and compares it with the information in `.arch.info.init` (filled with the `MACHINE_START` macro declaration).

2.2 Board Initialization Function

See Section 2, "Board Initialization Process," for information about the board initialization function. The function is `mxc_board_init` and it is found in `linux-2.6.26/arch/arm/mach-mx35/mx35_3stack.c`. The function is also referenced as `mdesc->init_machine`.

2.2.1 Calling the Function

The function is called in a specific way. In the file `linux-2.6.26/arch/arm/kernel/setup.c` there is a function named `customize_machine`.

```
static int __init customize_machine(void)
{
        /* customizes platform devices, or adds new ones */
        if (init_machine)
                init_machine();
        return 0;
}
arch_initcall(customize_machine);
```

This function contains a call to `init_machine()`, which is defined as:

```
static void (*init_machine)(void) __initdata;
```

The relationship between `init_machine` and the board initialization is given in `setup_arch()`. At the end of the function, `init_machine` gets a reference to the board initialization:

```
init_machine = mdesc->init_machine;
```

This function gets called in a particular way. This function is a part of a group of functions that get initialized through a table built by the linker. The group have the `__initcalls()` or `module_init()` calls.

The function `customize_kernel` is part of the `__initcall` group because of the line of code `arch_initcall` (`customize_machine`). The definition of `arch_initcall` is found in: `linux-2.6.26/include/linux/init.h`. The result expands in a `__define_initcall` that is placed in the section `".initcall" level ".init"`, and it has a value of the function (in this case `customize_machine`).

```
#define arch_initcall(fn)__define_initcall("3",fn,3)
(…)
* initcalls are now grouped by functionality into separate
* subsections. Ordering inside the subsections is determined
* by link order.
* For backwards compatibility, initcall() puts the call in the device init subsection.
*

* The `id' arg to __define_initcall() is needed so that multiple initcalls
* can point at the same handler without causing duplicate-symbol build errors.
*/

#define __define_initcall(level,fn,id) \
        static initcall_t __initcall_##fn##id __used \}
__attribute__((__section__(".initcall" level ".init"))) = fn
```

The function is added in the table built by the linker. The following code is an excerpt from:
`linux-2.6.26/arch/arm/kernel/vmlinux.lds`

```
__initcall_start = .;
*(.initcall0.init) *(.initcall0s.init) *(.initcall1.init)
*(.initcall1s.init) *(.initcall2.init) *(.initcall2s.init)
*(.initcall3.init) *(.initcall3s.init) *(.initcall4.init)
*(.initcall4s.init) *(.initcall5.init) *(.initcall5s.init)
*(.initcallrootfs.init) *(.initcall6.init) *(.initcall6s.init)
*(.initcall7.init) *(.initcall7s.init)
__initcall_end = .;
```

2.2.2 Board Initialization Content

The content of the board initialization function (`mxc_board_init`) is a set of initialization routines for the systems, modules and integrated chips that are on the board. The initialization is not that the drivers for each module are described and coded in this file, but it is actually the opposite. This function is where all the devices that are represented on the board are getting initialized and registered so they are accessible by the kernel.

The attributes are passed to each device, and the resources are provided. Some important cases are as follows:

- GPIO are assigned for each module
- Partitions for MTD devices are made
- Devices for buses are registered (such as I^2C or SPI)

Most of the routines used by the board initialization function are also defined in `linux-2.6.26/arch/arm/mach-mx35/mx35_3stack.c`. The following function is a summary of the elements that are enabled on the system, and its characteristics.

```
static void __init mxc_board_init(void)
{
        mxc_cpu_common_init();
        mxc_clocks_init();
        early_console_setup(saved_command_line);
        mxc_gpio_init();
        mxc_init_devices();
        if (!board_is_mx35(BOARD_REV_2))
                mx35_3stack_fixup_for_board_v1();
        mx35_3stack_gpio_init();
        mxc_init_enet();
        mxc_init_nor_mtd();
        mxc_init_nand_mtd();
        mxc_init_lcd();
        mxc_init_fb();
        mxc_init_bl();
        mxc_sgtl5000_init();

        i2c_register_board_info(0, mxc_i2c_board_info,ARRAY_SIZE(mxc_i2c_board_info));
        spi_register_board_info(mxc_spi_board_info,ARRAY_SIZE(mxc_spi_board_info));
        mxc_init_mmc();
        mxc_init_pata();
        mxc_init_bluetooth();
        mxc_init_gps();
        mxc_init_mlb();
}
```

3 Memory Map

Linux runs in virtual address space and hence, the Memory Management Unit (MMU) provides the virtual to physical address mapping defined by a memory map page table. This page table is a pre-defined memory map definition that maps the virtual memory to physical memory, so the device drivers access the device registers.

In the i.MX35 platform, the table is defined in: `linux-2.6.26/arch/arm/mach-mx35/mm.c`. This location is under machine dependent code (Machine Specific Layer or MSL). The header files that provide macros for all the I/O modules (physical and virtual addresses or conversion macros) are stored in `linux-2.6.26/include/asm-arm/hardware.h` or `/include/asm-arm/arch-mxc/mx35.h`.

The `linux-2.6.26/arch/arm/mach-mx35/mm.c` file contains the memory map of the system, and the `mxc_map_io` function, which is responsible for I/O memory mapping. This function is also found as `mdesc->map_io()`, in other words, this function is the I/O memory mapping function from the machine description structure. The following sections explain how to call the I/O mapping function, and describe the content of the memory map table.

3.1 I/O Mapping Function Flow and Description

The flow of calling the `mxc_map_io` function is as follows:

- Call the function `paging_init` (`&meminfo, mdesc`) from within `setup_arch()` (inside file `linux-2.6.26/arch/arm/kernel/setup.c`).
- The `paging_init` function calls `devicemaps_init` (both functions are located in `linux-2.6.26/arch/arm/mm/mmu.c`), and from there function the `mdesc->map_io()` is called.
- The function `mxc_map_io()` calls `iotable_init`, which gets the mapping using the `create_mapping` function (located in the file `linux-2.6.26/arch/arm/mm/mmu.c`).

```
static void __init devicemaps_init(struct machine_desc *mdesc)
{
        struct map_desc map;
        unsigned long addr;
(…)
        /*
         * Ask the machine support to map in the statically mapped devices.
         */
        if (mdesc->map_io)
                mdesc->map_io();
(…)
```

```
/*!
* This function initializes the memory map. It is called during the
* system startup to create static physical to virtual memory map for
* the IO modules.
*/

void __init mxc_map_io(void)
{
        iotable_init(mxc_io_desc, ARRAY_SIZE(mxc_io_desc));
}

/*
```

```
* Create the architecture specific mappings
*/
void __init iotable_init(struct map_desc *io_desc, int nr)
{
        int i;
        for (i = 0; i < nr; i++)
                create_mapping(io_desc + i);
}
```

3.2 Memory Map on i.MX35

The memory map is formed by an array of `map_desc` structures. This structure (`map_desc`) is defined in `linux-2.6.26/include/asm-arm/mach/map.h` and contains only four elements. These elements are `unsigned long` types for a virtual address, length, page frame number, and an `unsigned int` for the type.

```
struct map_desc {
        unsigned long virtual;
        unsigned long pfn;
        unsigned long length;
        unsigned int type;
};
```

The information obtained from the memory map are as follows:

- There are eight `map_desc` structures inside the array.
- The `.virtual` field is the virtual address where that `map_desc` is defined.
- The `.pfn` field is the address of the `map_desc` in terms of page frame number. The following code is in: `linux-2.6.26/include/asm-arm/memory.h`. The page frame numbers are the physical addresses with the offset values taken out, and the page values shifted to the right.

```
/*
* Convert a physical address to a Page Frame Number and back
*/
#define  __phys_to_pfn(paddr)((paddr) >> PAGE_SHIFT)
#define  __pfn_to_phys(pfn)((pfn) << PAGE_SHIFT)
```

The memory map structure is defined in the following lines of code (it is found in the file `linux-2.6.26/arch/arm/mach-mx35/mm.c`).

- Most of the macros used in the memory map structure is seen in the headers: `linux-2.6.26/include/asm-arm/arch-mxc/mx35.h`.
- `IRAM_BASE_ADDR_VIRT` represents the internal RAM. The virtual address is `0xF8400000` (physical address `0x10000000`). The length of the memory represented by the `map_desc` is 128KB.
- `X_MEMC_BASE_ADDR_VIRT` represents the control registers of the memory controllers. The virtual address is `0xF8A00000` (physical address `0xB8000000`). The length of the memory represented by the `map_desc` is 1MB.
- `NFC_BASE_ADDR_VIRT` represents the NAND flash controller. The virtual address is `0xF8B00000` (physical address `0xBB000000`). The length of the memory represented by the `map_desc` is 1MB.
- `ROMP_BASE_ADDR_VIRT` represents the platform ROMPATCH. The virtual address is `0xF8800000` (physical address `0x60000000`). The length of the memory represented by the `map_desc` is 1MB.

- AVIC_BASE_ADDR_VIRT represents the platform AVIC. The virtual address is 0xF8900000 (physical address 0x68000000). The length of the memory represented by the map_desc is 1MB.
- AIPS1_BASE_ADDR_VIRT represents the area for the first section of the ARM IP Bus (AIPS) control registers. Some of them are for I2C, UART, SSI, IOMUX, and so on. The virtual address is 0xF8500000 (physical address 0x43F00000) and the length of the memory represented by the map_desc is 1MB.
- SPBA0_BASE_ADDR_VIRT represents the shared peripheral bus arbiter (SPBA) registers. Some of them are for CSPI, UART, SSI, ATA, and so on. The virtual address is 0xF8600000 (physical address 0x50000000) and the length of the memory represented by the map_desc is 1MB.
- AIPS2_BASE_ADDR_VIRT represents the area for the next section of the AIPS control registers. Some of them are for GPIO, SDMA, WDOG, CAN, and so on.The virtual address is 0xF8700000 (physical address 0x53F00000) and the length of the memory represented by the map_desc is 1MB.

```
/*This structure defines the MX35 memory map.*/
static struct map_desc mxc_io_desc[] __initdata = {
        {
         .virtual = IRAM_BASE_ADDR_VIRT,
         .pfn = __phys_to_pfn(IRAM_BASE_ADDR),
         .length = IRAM_SIZE,
         .type = MT_NONSHARED_DEVICE},
        {
         .virtual = X_MEMC_BASE_ADDR_VIRT,
         .pfn = __phys_to_pfn(X_MEMC_BASE_ADDR),
         .length = X_MEMC_SIZE,
         .type = MT_DEVICE},
        {
         .virtual = NFC_BASE_ADDR_VIRT,
         .pfn = __phys_to_pfn(NFC_BASE_ADDR),
         .length = NFC_SIZE,
         .type = MT_NONSHARED_DEVICE},
        {
         .virtual = ROMP_BASE_ADDR_VIRT,
         .pfn = __phys_to_pfn(ROMP_BASE_ADDR),
         .length = ROMP_SIZE,
         .type = MT_NONSHARED_DEVICE},
        {
         .virtual = AVIC_BASE_ADDR_VIRT,
         .pfn = __phys_to_pfn(AVIC_BASE_ADDR),
         .length = AVIC_SIZE,
         .type = MT_NONSHARED_DEVICE},
        {
         .virtual = AIPS1_BASE_ADDR_VIRT,
         .pfn = __phys_to_pfn(AIPS1_BASE_ADDR),
         .length = AIPS1_SIZE,
         .type = MT_NONSHARED_DEVICE},
        {
         .virtual = SPBA0_BASE_ADDR_VIRT,
         .pfn = __phys_to_pfn(SPBA0_BASE_ADDR),
         .length = SPBA0_SIZE,
         .type = MT_NONSHARED_DEVICE},
        {
         .virtual = AIPS2_BASE_ADDR_VIRT,
         .pfn = __phys_to_pfn(AIPS2_BASE_ADDR),
         .length = AIPS2_SIZE,
```

```
        .type = MT_NONSHARED_DEVICE},
};
```

NOTE

The memory mapping represents the static I/O section from the file `linux-2.6.26/Documentation/arm/memory.txt`, having `VMALLOC_END` and `feffffff` as limits.

4 References

The following reference documents are used in conjunction with this application note for board initialization and memory mapping using LTIB.

4.1 Freescale Semiconductor Documents

The following i.MX reference manuals are found at Freescale Semiconductor Inc. World Wide Web site at http://www.freescale.com.

- *i.MX35 PDK 1.5 Linux Reference Manual*. Chapter 5: Machine Specific Layer, 5.3 Memory Map, at http://www.freescale.com/files/32bit/doc/support_info/PDK_IMX35_LINUXDOCS_BUNDLE.zip
- *i.MX35 (MCIMX35) Multimedia Applications Processor Reference Manual* (IMX35RM). Chapter 2: Memory Maps.

4.2 Standard Documents

The following standard documentations are used as reference for this application note and are found at their respective Web sites.

- *Booting ARM Linux* (June 2004), at http://www.simtec.co.uk/products/SWLINUX/files/booting_article.html#ATAG_MEM#ATAG_MEM
- *Booting and Porting Linux and uCLinux on a new Platform* (February 2006), at http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2006/RR2006-08.pdf

5 Revision History

Table 1 provides a revision history for this application note.

Table 1. Document Revision History

Rev. Number	Date	Substantive Change(s)
0	02/2010	Initial release.

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: AN3984
Rev. 0
02/2010