Freescale Semiconductor
Application Note

Document Number: AN3980
Rev. 0, 03/2010

i.MX51 Board Initialization and Memory Mapping Using the Linux Target Image Builder (LTIB)

by *Multimedia Application 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.MX51 Board Support Package (BSP).

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

The knowledge of these aspects enable better understanding of the BSP structure. When changes such as migrations to another board or device with the memories, 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.MX51 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 the information, and the initialization process flow of a BSP running a LTIB, on an i.MX51 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 (size, start address) to the kernel.

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

The last section describes 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 flow that are to be executed, and the information that the bootloader needs to pass to the kernel.

1.1 General Bootloader Objectives

This section explains the general objectives of a Linux bootlader for ARM devices. There are five minimum elements that a bootloader needs to accomplish:

1. Setup and initialize the system RAM—The boot loader finds and initializes the entire RAM to provide the volatile data storage in the system. The algorithm that is used to locate and set up the RAM depends on the processor and bootloader designs.
2. Initialize 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 `console=` 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 the processor that is running on the system. This information is a `MACH_TYPE_XXX` macro.
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 systems memory (RAM). Other elements such a RAM disk creation and 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 Linux Booting Process.”
5. Call or start the kernel image—The bootloader finally calls the kernel image (an 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 to supervisor mode with IRQ disabled.
   b) Turn off the Memory Management Unit (MMU). The code running in RAM does not have translated addressing yet.
   c) Turn off the data cache
   d) 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 the need of a change in 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 placed normally in the machine dependant code (`linux-2.6.28/arch/arm/mach-mx51/mx51_babbage.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 values for the parameters such as memory data. It is not recommended to use all the time.

1.2 Tags in 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 part of the tagged list that is passed from the bootloader to the kernel. The parameter is manipulated through the kernel command line option `mem=`, and by using this, the bootloader passes size and physical memory.

NOTE

For more information on the syntax of the parameters go to the documentation inside the kernel `linux-2.6.28/Documentation/kernel-parameters.txt`.

ATAG_MEM is a part of the parameter passed by the bootloader to the kernel. The parameters create a list (tagged list) that contain information such as the command line tag associated with the kernel command line string, serial console information, RAM disk usage, initial configuration values for the framebuffer. This tagged list (ATAG) 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 previously set fixed value (by default, both the bootloader and the kernel knows where it is).

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

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

  NOTE

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

- The list should 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 represent the tag type. In almost all cases, each tag header has more data associated with the type of tag (except for ATAG_NONE).

The following are the lines of code 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-2.6.28/arch/arm/include/asm/setup.h`. However, the bootloader contains a definition very similar to the following lines of code:

```
struct tag {
         struct tag_header hdr;
         union {
                  struct tag_core   core;
                  struct tag_mem32mem;
                  struct tag_videotextvideotext;
                  struct tag_ramdiskramdisk;
                  struct tag_initrd initrd;
                  struct tag_serialnrserialnr;
                  struct tag_revisionrevision;
                  struct tag_videolfbvideolfb;
                  struct tag_cmdlinecmdline;
                  …..
         } 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. This structure contains two fields, one for the size of the memory represented in this tag, and another for the physical address of this memory.

The following lines of code are from the `linux-2.6.28/arch/arm/include/asm/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:

```
struct tag_header {
         __u32 size;
         __u32 tag;
};

……

/* 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-2.6.28/arch/arm/include/asm/setup.h`.

The file from where the tag is retrieved is: `linux2.6.28/arch/arm/kernel/setup.c`. The specific function inside this file is `void __init setup_arch(char **cmdline_p)`, and this function is called from `asmlinkage void __init start_kernel(void)` function (from the file `linux2.6.28/initi/main.c`).

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

The kernel gets the information from the memory configuration 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 `mem=––`. Both the cases are described and implemented in `linux-2.6.28/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: `linux2.6.28/arch/arm/kernel/setup.c`. 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-2.6.28/arch/arm/include/asm/setup.h`, and its meaning is:

```
#define __tag __attribute_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).
- An attribute that assembly functions of code related to this declaration will be placed in the section `.taglist.init` that is defined by the linker (see `vmlinux.lds` in the following pages) instead of the common text section.
- It has as 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`).

Each declaration generate one `struct tagtable` as the following.

The tag list is retrieved in `setup_arch`. In the body of the function there is a call to the function `parse_tags`. This function parses all the tags contained in the list by using a function that parses a tag by each time (`parse_tag`). This last function looks for the tags and calls the parse function of each tag if there is a match.

```
/*
* 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;
}
```

As a limit it has elements such as `__tagtable_begin`, `__tagtable_end` defined in `linux-2.6.28/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 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;
}
```

This function calls the `arm_add_memory` at the end. This is the function that sets the memory information (start address and size memory) 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=`. The 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 set as well by the `vmlinux.lds`, but now it is named `early_param.init` and the limits are `__early_begin __early_end`
- Instead of a `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` calls `parse_cmdline`. When parsing the command line, and if the `mem=` is found, it calls the `early_mem` function, that at the end `arm_add_memory` is called to fill the `membank` structure.

All the code related are found in `linux-2.6.28/arch/arm/kernel/setup.c` and definitions in `linux-2.6.28/arch/arm/include/asm/setup.h` (`__early_param` or `early_param` 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 (current board) used in the system.

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-2.6.28/arch/arm/include/asm/mach/arch.h`. The `MACH_TYPE` parameter passed in the `MACHINE_START` is stored in `linux-2.6.28/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. In this application note, the i.MX51 board is the Babbage board. The file where the declaration is made is: `linux-2.6.28/arch/arm/mach-mx51/mx51_babbage.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-2.6.28/arch/arm/include/asm/mach/arch.h`.

```
struct machine_desc {
/*
* Note! The first four elements are used
* by assembler code in head.S, head-common.S
*/
unsigned intnr;                         /* architecture number*/
        unsigned intphys_io;            /* start of physical io*/
        unsigned intio_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 shown, there are several members of the structure, not all of them are filled in the final `MACHINE_START` declaration. For the current system, the declaration is located in `linux-2.6.28/arch/arm/mach-mx51/mx51_babbage.c`:

```
/*
* The following uses standard kernel macros define in arch.h in order to
* initialize __mach_desc_MX51_3STACK data structure.
*/
/* *INDENT-OFF* */
MACHINE_START(MX51_BABBAGE, "Freescale MX51 Babbage 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__MX51_BABBAGE`
- The name of the `mach_desc` structure is: `__mach_desc_MX51_BABBAGE`
- The name parameter of the `mach_desc` structure is: `Freescale MX51 Babbage 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 to 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 process of uncompressing the kernel, the kernel initializes the hardware using init_machine parameter. Before initializing the hardware, the kernel validates if it is running in the CPU that it was compiled for. This verification is followed by the initialization of caches and the MMU. To know if this is true, the kernel gets the processor ID and it is compared to the data contained in the proc.info.init section (see vmlinux.lds).

The process begins in the setup_processor() function from the setup_arch() in linux-2.6.28/arch/arm/kernel/setup.c. This setup_processor() function uses the lookup_processor_type (located in: arch/arm/kernel/head-common.S) and read_cpuid_id (located in: linux-2.6.28/arch/arm/include/asm/cputype.h) functions to get the processor ID. The proc.info section is filled with information from the file linux-2.6.28/arch/arm/mm/proc-v7.S, which sets the .proc.info.init section, and contains the information about the processor ID.

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 process is also inside the setup_arch() in linux2.6.28/arch/arm/kernel/setup.c. After the setup_processor function 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 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 more information about the board initialization function. The function is mxc_board_init and it is found in linux-2.6.28/arch/arm/mach-mx51/mx51_babbage.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 `linux2.6.28/arch/arm/kernel/setup.c` there is a function named `customize_machine`. This function contains a call to `init_machine()`, which is defined as:
`static void (*init_machine)(void) __initdata.`

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

The relationship between `init_machine` and the board initialization board is given in `setup_arch()`. At the end of the function `init_machine` is assigned with a reference to the initialization board:
`init_machine = mdesc->init_machine.`

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

The function `customize_kernel` is a part of the `__initcall` group because of the line of code `arch_initcall` (`customize_machine`). The definition of `arch_initcall` is found in: `linux-2.6.28/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 from: `vmlinux.lds` (`linux-2.6.28/arch/arm/kernel/`).

```
__initcall_start = .;
*(.initcallearly.init) __early_initcall_end = .; *(.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 in 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 in the board are getting initialized or registered.

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

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

Most of the routines inside the board initialization function are also in `linux-2.6.28/arch/arm/mach-mx51/mx51_babbage.c`. The following function is a summary for the elements that are enabled on the system and its characteristics:

```
static void __init mxc_board_init(void)
{
        int err;

        mxc_cpu_common_init();
        mxc_gpio_init();
        mx51_3stack_io_init();
        early_console_setup(saved_command_line);
        mxc_init_devices();

        mxc_expio_init();
        mxc_init_enet();
        mxc_init_pata();
        mxc_init_fb();
        mxc_init_bl();
        mxc_init_keypad();
        mxc_init_nand_mtd();
        mxc_init_mmc();
        mxc_init_sim();
        mxc_init_srpgconfig();
        mx51_3stack_init_mc13892();

#if defined(CONFIG_I2C_MXC) || defined(CONFIG_I2C_MXC_MODULE)

#ifdef CONFIG_I2C_MXC_SELECT1
        i2c_register_board_info(0, mxc_i2c0_board_info,
                                ARRAY_SIZE(mxc_i2c0_board_info));
#endif
#ifdef CONFIG_I2C_MXC_SELECT2
        i2c_register_board_info(1, mxc_i2c1_board_info,
                                ARRAY_SIZE(mxc_i2c1_board_info));
#endif
#if defined(CONFIG_I2C_MXC_HS) || defined(CONFIG_I2C_MXC_HS_MODULE)
        i2c_register_board_info(3, mxc_i2c_hs_board_info,
                                ARRAY_SIZE(mxc_i2c_hs_board_info));
#endif

#endif
```

```
        mxc_init_touchscreen();
        mxc_init_wm8903();
        mxc_init_sgtl5000();
        mxc_init_bluetooth();

        err = mxc_request_iomux(MX51_PIN_EIM_D19, IOMUX_CONFIG_GPIO);
        if (err)
                printk(KERN_ERR "Error: bt reset request gpio failed!\n");
        else
        mxc_set_gpio_direction(MX51_PIN_EIM_D19, 0);
}
```

3 Memory Map

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

In the i.MX platform, the table is defined in `linux-2.6.28/arch/arm/mach-mx51/mm.c`. This location is under machine dependant code (MSL or Machine Specific Layer). The header files that provide macros for all the IO modules (physical and virtual addresses or conversion macros) are stored in `linux-2.6.28/arch/arm/plat-mxc/include/mach/hardware.h` or `linux-2.6.28/arch/arm/plat-mxc/include/mach/mx51.h`.

The `linux-2.6.28/arch/arm/mach-mx51/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 the I/O mapping function is called, 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 starts from the `setup_arch()` function inside the `linux2.6.28/arch/arm/kernel/setup.c`. The call that initiates the process is `paging_init(&meminfo, mdesc)` (it is the function that is called after `parse_cmdline`). The `paging_init` function is located in `linux-2.6.28/arch/arm/mm/mmu.c`, and this function calls `devicemaps_init` (located in `linux-2.6.28/arch/arm/mm/mmu.c`), and from there the function `mdesc->map_io()` is called.

```
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();
…
```

The `mxc_map_io` function has a simple objective. This function calls `iotable_init`, which gets the mapping using the `create_mapping` function (`linux-2.6.28/arch/arm/mm/mmu.c`). In i.MX51, the

mxc__map_io function checks the revision of the chip before calling iotable_init, and changes the tzic_addr.

```
/*!
* 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)
{
        u32 tzic_addr;
        if (cpu_is_mx51_rev(CHIP_REV_2_0) < 0)
                tzic_addr = 0x8FFFC000;
        else
                tzic_addr = 0xE0003000;

        mxc_io_desc[2].pfn =  __phys_to_pfn(tzic_addr);
        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.MX51

The memory map is formed by an array of map_desc structures. This structure located in linux-2.6.28/arch/arm/include/asm/map.h, contains only four elements. These elements are unsigned long types for a virtual address, length and page frame number, while the unsigned int is 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 seven 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 page frame number. The page frame numbers are the physical address with the offset values taken out, and the page values shifted to the right.

  ```
  arch/arm/include/asm/memory.h
  /*
  * 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)
```

- Most of the macros used in the structure is seen in the headers: `linux-2.6.28/arch/arm/plat-mxc/include/mach/mx51.h`.
- `IRAM_BASE_ADDR_VIRT` represents the internal RAM (physical address `0x1FFE0000`), the virtual address is `0xFA3E0000` is and the length of the `map_desc` is 128 KB.
- `DEBUG_BASE_ADDR_VIRT` represents the area for debugging modules on the chip (physical address `0x60000000`), the virtual address is `0xFA200000` and the length of the `map_desc` is 1 MB.
- `TZIC_BASE_ADDR_VIRT` represents trust zone aware interrupt controller. The TZIC collects interrupt request for all i.MX51 sources to the core (physical address `0x8FFFC000`), the virtual address is `0xFA100000` and the length of the `map_desc` is 16 KB.
- `AIPS1_BASE_ADDR_VIRT` represents the area for the AIPS section 1 of module registers (physical address `0x73F00000` some of them are USBs, UARTs, GPIOs KPP, WDOGs GPT, IOMUX, PWMs, EPITs), the virtual address is `0xFB000000` and the length of the `map_desc` is 1 MB.
- `SPBA0_BASE_ADDR_VIRT` represents the global module registers area (physical address `0x70000000` some of them are ESDH, UART, eCSPI, SSI, SPBA, PATA), the virtual address is `0xFB100000` and the length of the `map_desc` is 1 MB.
- `AIPS2_BASE_ADDR_VIRT` represents the area for the AIPS section 2 of module registers (physical address `0x83F00000` some of them are DPLLIPs, IIM, SDMA I2Cs, FEC, VPU, SAHARA, EMI, AUDMUX), the virtual address is `0xFB200000` and the length of the `map_desc` is 1 MB.
- `NFC_BASE_ADDR_AXI_VIRT` represents the NAND Flash register area (physical address `0xCFFF0000`), the virtual address is `0xF9000000` and the length of the `map_desc` is 64 KB.

```
/*!
* This structure defines the MX51 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_DEVICE},
         {
          .virtual = DEBUG_BASE_ADDR_VIRT,
          .pfn = __phys_to_pfn(DEBUG_BASE_ADDR),
          .length = DEBUG_SIZE,
          .type = MT_DEVICE},
         {
          .virtual = TZIC_BASE_ADDR_VIRT,
          .pfn = __phys_to_pfn(TZIC_BASE_ADDR),
          .length = TZIC_SIZE,
          .type = MT_DEVICE},
         {
          .virtual = AIPS1_BASE_ADDR_VIRT,
          .pfn = __phys_to_pfn(AIPS1_BASE_ADDR),
          .length = AIPS1_SIZE,
          .type = MT_DEVICE},
         {
          .virtual = SPBA0_BASE_ADDR_VIRT,
```

```
            .pfn = __phys_to_pfn(SPBA0_BASE_ADDR),
            .length = SPBA0_SIZE,
            .type = MT_DEVICE},
           {
            .virtual = AIPS2_BASE_ADDR_VIRT,
            .pfn = __phys_to_pfn(AIPS2_BASE_ADDR),
            .length = AIPS2_SIZE,
            .type = MT_DEVICE},
           {
            .virtual = NFC_BASE_ADDR_AXI_VIRT,
            .pfn = __phys_to_pfn(NFC_BASE_ADDR_AXI),
            .length = NFC_AXI_SIZE,
            .type = MT_DEVICE},
};
```

The memory mapping represents the static I/O section that has `0xfeffffff` and `VMALLOC_END` as limits. See `memory.txt` from the `linux-2-6.28/Documentation/arm/`.

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.MX Family Linux Software Development Kit Reference Manual*. Chapter 3: Machine Specific Layer, 3.3 Memory Map, at
  http://www.freescale.com/files/32bit/doc/support_info/EVK16_IMX51_LINUXDOCS_BUNDLE.zip
- *MCIMX51 Multimedia Applications Processor Reference Manual* (MCIMX51RM). Chapter 2: Memory Map.

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	03/2010	Initial Release.

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: AN3980
Rev. 0
03/2010