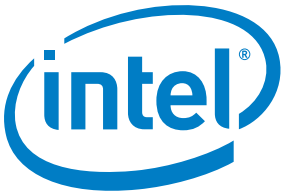


Reference Boot Loader from Intel

Application Note

March 2013

Document Number: 328739 -001

Contents

Figures

Revision History

Date	Revision	Description
March 2013	1.0	Initial release.

§

1 Introduction

This document describes the Reference Boot Loader from Intel. This includes:

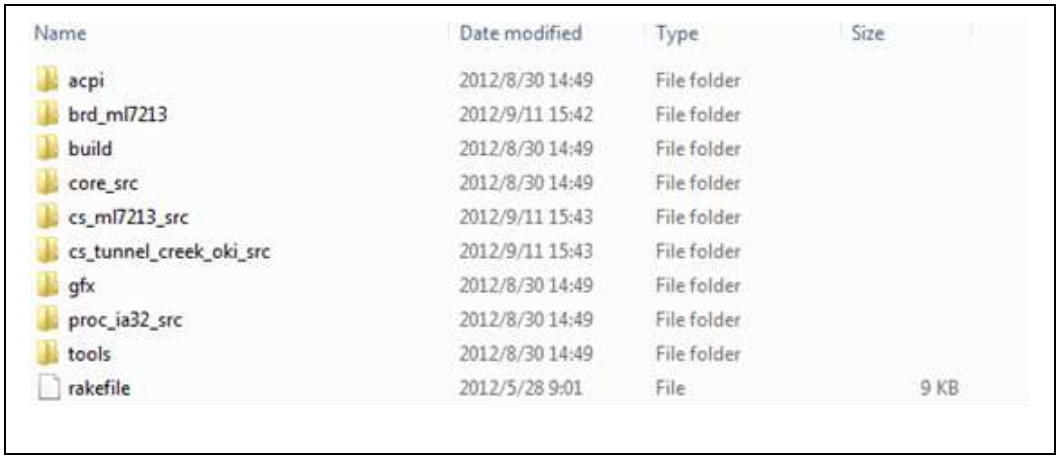
- The source code directory structure,
- The development environment,
- How to build the firmware image,
- An illustration of the boot sequence (with pseudo-code), and
- A list of ways that developers can customize the firmware for a specific platform.

§

2 Source code directory structure

The directory structure of the source code folder is shown in Figure 1.

Figure 1. Folder structure



- **acpi** is the folder to build the ACPI table.
- **brd_ml7213** is the folder for the specified platform code.
- **build** is the folder that includes the files that control the build process.
- **core_src** is the folder that includes the code for various common functions, such as PCI enumeration, device drivers, debug, memory, console, compression, etc.
- **cs_ml7213_src** and **cs_tunnel_creek_oki_src** are the directories that include the code for the ML7213 IOH and the Intel® Atom™ Processor E6xx Series based processor.
- **gfx** is the folder that includes the code and library for the graphics controller.
- **proc_ia32_src** is the folder that includes the code for initializing and configuring the CPU.
- **tools** is the folder that includes the build tools.
- **rakefile** is the Ruby rake file for building the firmware image.

§

3 Development environment

The development environment for building the reference boot loader firmware requires a 32-bit or 64-bit version of Fedora 14 Linux with the development package (including C, Perl, and rake) installed. Use of other Linux distributions is not guaranteed. The Ruby rake tool can be installed by using the yum command. Below are the steps to install the tool.

```
[root@localhost /]# yum install rake
Loaded plugins: langpacks, presto, refresh-packagekit
Resolving Dependencies
--> Running transaction check
---> Package rubygem-rake.noarch 0:0.9.2.2-15.fc17 will be installed

...

Transaction Summary
================================================================================
Install  1 Package  (+7 Dependent packages)
Upgrade             ( 1 Dependent package)

Total download size: 4.6 M
Is this ok [y/N]: y

...

Importing GPG key 0x1ACA3465:
 Userid     : "Fedora (17) <fedora@fedoraproject.org>"
 Fingerprint: cac4 3fb7 74a4 a673 d81c 5de7 50e9 4c99 1aca 3465
 Package    : fedora-release-17-1.noarch (@anaconda-0)
 From       : /etc/pki/rpm-gpg/RPM-GPG-KEY-fedora-i386
Is this ok [y/N]: y

...

Dependency Updated:
  ruby-libs.i686 0:1.9.3.194-15.fc17

Complete!
[root@localhost /]#

...
```

§

4 Building the firmware image

The following commands could be used to clean, build and rebuild the source code.

1. To build the firmware image:

```
%> rake build_oki_ivi
```

2. To rebuild the firmware image:

```
%> rake rebuild_oki_ivi
```

3. To cleanup after a previous build:

```
%> rake clean_all
```

§

5 Boot sequence

The boot sequence consists of four phases.

5.1 Reset phase

The first instruction that is executed after power is applied or the system is reset (the reset vector) is located in file protected_mode_switch.s. The label for the reset vector is `protected_mode_switch`. The reset phase saves the BIST, enters protected mode, and then checks whether the system is doing a cold boot or a warm boot. Depending on the result, it then goes to either the basic initialization phase (cold boot) or the advanced initialization phase (warm boot).

Pseudo code:

```
protected_mode_switch:
  Save BIST
  Enter protected mode
  Check if cold or warm boot
```

5.2 Basic initialization phase

The entry point of this phase is located in file basic_init.s. The label for the start of this phase is `basic_init`. This phase loads the CPU microcode, and then does the basic chipset and CPU initialization. After the memory is initialized, it then relocates the data section into RAM, because the data cannot be updated while the data section is located in ROM. When complete it then goes to the advanced initialization phase.

Pseudo code:

```
basic_init:
  Load Micro Code
  Basic initialization for Chipset and CPU
  Relocate data which are needed to update
  Go to the advanced initialization phase
```

5.3 Advanced initialization phase

The entry point of this phase is located in file init.c. The label for the start of this phase is `advancedInit`. This phase initializes and configures the chipset and the CPU, enables interrupts, initializes the graphics controller, and displays a splash screen. When complete it then goes to the load OS phase.

Pseudo code:

```
advancedInit:
  Initialize Chipset and Cpu
  Enable interrupts
  Initialize graphics controller
  Display splash screen
  Go to userInit phase
```

5.4 Load OS phase

The entry point of this phase is located in file user_init.c. The label for the starting point of this phase is `userInit`. The main purpose of this phase is to load an OS image from a storage device such as a hard drive, a USB storage device, an SD card or MMC card, or to boot into an internal Shell.

Pseudo code:

```
userInit:
  Boot from Hard disk, USB storage, SD Card, or MMC card
```

§

6 Customizing the firmware

This section will show how to customize the firmware for a specific platform, including feature enabling and disabling, and configuring the GPIO, SVID, SID, PCI IRQ routing, option ROM, Microcode, SMI handler, and boot device.

6.1 Enable or disable features

Features can be enabled or disabled in file brd_ml7213\makefile. A feature is enabled or disabled by setting the corresponding configuration flag to either "1" or "0". The example below shows how to disable boot from HDD and enable boot from MMC and SD cards.

Example:

```
CFG_ATA=0
CFG_MMC=1
CFG_SD=1
```

6.2 IRQ routing configuration

The IRQ routing configuration is located in file board_pci_irq.c. It is an array that contains the PCI device interrupt line and PIRQ information.

Example:

```
IrqRoutingEntry pciIrqRouting[] = {
  { 3, 0, /* PCIE1 Port :  PCIE -> PCI Bridge */ \
    { \
      { PIRQB, PCI_IRQS }, /* A */ \
      { 0, 0 }, /* B */ \
      { 0, 0 }, /* C */ \
      { 0, 0 }, /* D */ \
    }, \
    0, 0 \
  }, \
 ...
}
```

6.3 Intel® Atom™ Processor E6xx Series GPIO, SVID, and SID initialization

The initialization of the processor GPIO, SVID, and SID is handled in function `boardPreInit`, which is located in file board_pre_init.c.

Example:

```
void boardPreInit()
{
  // Init GPIO
   data32 = inl(GpioBase + 0x00) & 0xFFFFFFE0;
   data32 |= gpioEnable1;
   outl(GpioBase + 0x00, data32);
   // Init SVID and SID
   pciwrite32(0,0x1A,0,0x94,pciread32(0,0x1A,0,0));
}
```

6.4 IOH GPIO initialization

The initialization of the IOH GPIO is handled in function `iohGpioInit`, which is located in file Ioh_gpio.c.

Example:

```
void iohGpioInit()
{
   ...
  mmioWrite16(iohGpioBase+i*0x30, R_IOH_GPIO_PM, iohGpioPM[i]);
  ...
  return;
}
```

6.5 IOH SVID and SID initialization

The initialization of the IOH SVID and SID is handled in function `iohMiscInit`, which is located in file Ioh.c.

Example:

```
void iohMiscInit (void)
{
  ...
  pciwrite32(2,0x00,0,0x2C,pciread32(2,0x00,0,0)); Packet Hub
  ...
}
```

6.6 Splash screen

The configuration of a splash screen is located in file init.c. The call to function `epog_driver_init` initializes the graphic controller. The splash screen is then displayed by the call to function `EPOG_START`.

Example:

```
#if CFG_GFX
   ...
   /* call board specific splash screen init hook */
   GFX_handle = epog_driver_init();
   EPOG_START(GFX_handle);
   ...
#endif /* CFG_GFX */
```

These two functions are included in library libepog.a, which is located in directory brd_ml7213. This library can be generated and configured by using the EMGD configuration editor. This tool can integrate a custom picture into the library that can be used for the splash screen.

6.7 CPU microcode location

The CPU microcode is located in file board_proc_update.s. New microcode could replace the data shown in the example below for a microcode update.

Example:

```
.globl M0220661104
M0220661104:
.long 0x00000001 # Header Version
.long 0x00000104 # Patch ID
.long 0x10232009 # DATE
.long 0x00020661 # CPUID
.long 0xd89229bc # Checksum
.long 0x00000001 # Loader Version
.long 0x00000002 # Platform ID
.long 0x000013d0 # Data size
.long 0x00001400 # Total size
.long 0x00000000 # reserved
.long 0x00000000 # reserved
.long 0x00000000 # reserved
```

6.8 SMI handler

The SMI is handled by function `smmSmiHandler`, which is located in file std_smi_hndlr.c.

Example:

```
void smmSmiHandler() {
  ...
}
```

6.9 Boot device selection

The selection of the boot device is located in file user_init.c. A boot device is enabled or disabled if its corresponding configuration flag is set to "1" or "0" (see section 6.1).

Example:

```
#if CFG_ATA
   bootAta();
#endif
#if CFG_SD
   bootSd();
#endif
#if CFG_MMC
   bootMmc();
#endif
```

You can change the order in which the boot loader searches for a bootable OS image by changing the order of the calls to the various boot functions in this file.

§

7 Results

The Reference Boot Loader from Intel has a simple boot flow, low complexity, and is easy to understand and customize. The boot loader firmware can be easily compiled using open source Linux tools. Compared to UEFI and Legacy BIOS firmware, development and maintenance of firmware based on the Reference Boot Loader from Intel requires less effort and investment, leaving more engineering resources available to focus on operating system, device driver, and application development.

The Reference Boot Loader from Intel is a good choice for embedded IA platforms due to its simplicity and open source development environment. Currently, it supports booting Linux operating systems. Support for booting Windows* is planned in a future release.

§