Early Camera Presentation in Linux for the Intel® Atom™ Processor E6xx Series

Application Note

March 2013
# Contents

## 1.0 About This Document
- 1.1 Document Scope ................................................................. 6
- 1.2 General Assumptions ........................................................... 6
- 1.3 Technical Assumptions .......................................................... 6
- 1.4 Terminology ........................................................................ 7
- 1.5 Reference Documents .......................................................... 7

## 2.0 High Level Description
- 2.1 Hardware Configuration ....................................................... 8
- 2.2 Platform Firmware ............................................................... 9
- 2.3 Linux Kernel and Subsystem ................................................ 9
- 2.4 Graphic System and UVC Driver ........................................... 9

## 3.0 Low Level Description
- 3.1 Linux Kernel .......................................................................... 11
  - 3.1.1 Asynchronous AHCI port probing ...................................... 11
  - 3.1.2 Asynchronous Driver Initialization .................................. 11
  - 3.1.3 Reordering Module Initialization .................................... 11
- 3.2 Graphics and UVC Camera System ....................................... 12
  - 3.2.1 Kernel Module Loaded Sequence .................................... 12
  - 3.2.2 Initializing Camera Module ........................................... 12
  - 3.2.3 Reverse Gear Detect Thread ......................................... 13
  - 3.2.4 Terminating Camera Module ........................................ 16
  - 3.2.5 Enabling or Disabling Camera Destination Colorkey ....... 16

## 4.0 Implementation Details
- 4.1 Reference Boot Loader from Intel ........................................ 17
  - 4.1.1 Reducing Cache-As-Register Size .................................. 17
  - 4.1.2 Remove Unnecessary Features ...................................... 18
  - 4.1.3 Enable Direct Boot ....................................................... 19
- 4.2 Linux Kernel .......................................................................... 20
  - 4.2.1 Kernel boot argument .................................................. 20
  - 4.2.2 include\linux\device.h .................................................. 20
  - 4.2.3 drivers\Makefile ............................................................ 21
  - 4.2.4 drivers\ata\libata-core.c .............................................. 22
  - 4.2.5 drivers\base\dd.c .......................................................... 23
  - 4.2.6 drivers\gpu\drm\drm_pci.c ........................................... 28
  - 4.2.7 drivers\usb\host\ehci-pci.c ........................................... 31
  - 4.2.8 drivers\usb\host\ohci-pci.c ........................................... 31
  - 4.2.9 init\initramfs.c ............................................................. 31
- 4.3 Graphics and UVC Camera System ..................................... 34
  - 4.3.1 Embedded Media & Graphics Driver (EMGD) .................. 34
  - 4.3.2 Video For Linux 2 (V4L2) ............................................. 35
  - 4.3.3 GPIO ........................................................................... 35
  - 4.3.4 v2g_bridge .................................................................. 36
  - 4.3.5 USB Video Capture (UVC) ........................................... 37

## 5.0 Results
- 5.1 Platform Firmware ............................................................ 39
- 5.2 Linux Kernel, EMGD & UVC ............................................... 39
  - 5.2.1 Before Optimization .................................................... 39
  - 5.2.2 Post Optimization ....................................................... 40

## 6.0 Setup for Android ............................................................. 43
7.0 Setup for Tizen Linux

Figures
1  2-way (On/Off) Switch to Emulate Reverse Gear Signal ......................................................... 8
2  Early Direct Camera Presentation Block Diagram ........................................................................ 10
3  Camera Module Initialization ........................................................................................................ 13
4  Frame Presentation (Part 1) ............................................................................................................. 15
5  Frame Presentation (Part 2) ............................................................................................................. 16
6  The Original Boot Graph .................................................................................................................. 39
7  Excerpt of Linux Boot Message (dmesg) for Camera Presentation .................................................. 40
8  Boot Graph after Optimization¹ ...................................................................................................... 41
9  Excerpt of Linux Boot Message (dmesg) for Camera Presentation after Optimization¹ .......... 41

Tables
1  Glossary ............................................................................................................................................ 7
2  Type of Asynchronous Device Probing ............................................................................................ 11
3  Time Measurement of the Reference Boot Loader from Intel Stage (in ms) ................................. 39
**Revision History**

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>March 2013</td>
<td>001</td>
<td>Initial release</td>
</tr>
</tbody>
</table>
1.0 About This Document

1.1 Document Scope

This document discusses the ingredients and best-known methods (BKM) for achieving early camera presentation in a Linux-based system on an Intel® Atom™ Processor E6XX based platform. The document includes programming guide for customers to achieve the aforesaid feature by optimizing the following 3 areas:

• Platform firmware
• Linux kernel & subsystem
• Graphic System & USB video capture (UVC)

The early camera presentation is particularly useful in automotive reverse parking guidance system as a visual aid and other applications.

This document does not discuss BKMs for optimizing Linux subsystems such as X server, Linux Init process, user shell process and others.

1.2 General Assumptions

This document assumes that the reader is familiar with:

• Reference Boot Loader from Intel
• Linux Kernel
• Video For Linux 2 (V4L2)
• Intel® Embedded Media & Graphic Drivers (Intel® EMGD)
• USB Video Capture (UVC) Driver

1.3 Technical Assumptions

• Feature development is based on the Intel® Atom™ Processor E660 with Intel® Platform Controller Hub EG20T Development Kit customer reference board.
• Feature development is based on the standard Linux-based kernel package (at minimum Linux kernel version 2.6)
• Intel® EMGD provides handshake interfaces to render V4L2 inputs to the hardware Overlay plane. This includes mapping video frame memory from the V4L2 driver to EMGD video memory, turning on or off direct video display, and directly presenting the video frame.
• For simplification, one YUV4:2:2 FOURCC format is supported: YUY2.
• An external switch was connected to the GPIO port to simulate reverse gear engagement.
1.4 Terminology

Table 1 contains definitions of the terms used later in this document.

Table 1. Glossary

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRM</td>
<td>Direct Rendering Manager</td>
<td>An infrastructure to provide efficient video acceleration (especially 3D rendering) on Unix-like operating system.</td>
</tr>
<tr>
<td>EHCI</td>
<td>Enhanced Host Controller Interface</td>
<td>Register-level interface as defined in USB2.0 specification</td>
</tr>
<tr>
<td>Intel EMGD</td>
<td>Intel Embedded Media &amp; Graphics Driver</td>
<td></td>
</tr>
<tr>
<td>OHCI</td>
<td>Open Host Controller Interface</td>
<td>Register-level interface as defined in USB1.1 specification</td>
</tr>
<tr>
<td>SSD</td>
<td>Solid State Disk</td>
<td></td>
</tr>
<tr>
<td>SDHC</td>
<td>Secure Digital Host Controller</td>
<td></td>
</tr>
<tr>
<td>UVC</td>
<td>USB Video Capture</td>
<td>USB-based video capture device</td>
</tr>
<tr>
<td>V4L2</td>
<td>Video For Linux 2</td>
<td>Video capture application programming interface for Linux</td>
</tr>
</tbody>
</table>

1.5 Reference Documents

[ARJAN] Arjan van de Ven & Auke Kok, “LPC: Booting Linux in five seconds” as covered by Don Marti at LWN.net
http://lwn.net/Articles/299483/
2.0 High Level Description

To present video content from a UVC device to the display panel very early as measured from power-on, there are three main areas that are optimized:

- Platform Firmware/Boot Loader
- Linux Kernel and subsystem
- Graphic system and UVC drivers

2.1 Hardware Configuration

The following hardware components are used for early camera presentation:

- The Intel® Atom™ Processor E6XX (1.3GHz) platform with EG20T Platform Controller Hub.
- USB keyboard & mouse
- USB Video Capture device (Logitech HD Pro WebCam C920)
- SATA Solid State Disk (Kingston SSDNow V+200 120GB)
- 2-way switch to emulate reverse gear signal that is connected to GPIO pin (Vcc:Pin-40, Gnd:Pin-5 & GPIO input:Pin-9) as shown below.

Figure 1. 2-way (On/Off) Switch to Emulate Reverse Gear Signal
2.2 Platform Firmware

The Reference Boot Loader from Intel is used to provide very fast platform initialization time. Major optimization steps taken on the Reference Boot Loader from Intel are:

- CAR (Cache-As-Register) size is reduced from 128K byte to 4K byte.
- PS/2 keyboard & mouse support is removed.
- Support SD cards & SATA disks detection only.
- Turn on Direct Boot feature within the Reference Boot Loader from Intel.

When Direct Boot is enabled, the Reference Boot Loader from Intel by-passes the Master Boot Record (MBR) parsing, OS-loader loading and running (e.g. GRUB2 or SYSLINUX) which happen for legacy boot-path. In fact, the OS-loading capability is built into the Reference Boot Loader from Intel whereby the Direct Boot feature will load the Linux kernel image, bzImage, directly from boot-medium to system memory before passing control to it.

The SATA Solid State Disk (SSD) is used as a boot device due to its very high data throughput which is a critical factor for Direct Boot. SD Card support is retained because initialization time of the SDHC controller is short, i.e. not a major boot-time contributor.

2.3 Linux Kernel and Subsystem

The following optimizations are based on the work in Ref. [1]:

- Asynchronous EHCI and OHCI driver initialization
- Asynchronous EMGD graphic driver initialization
- Asynchronous AHCI port probing
- Initialization of the EMGD graphic driver & EHCI host controller driver is done earlier than other drivers.
- Linux kernel configuration (make menuconfig) is customized per system requirement

2.4 Graphic System and UVC Driver

Early direct camera presentation consists of five components:

- v2g_bridge is a newly created Kernel Mode component. It plays the role of camera application module in the Kernel Mode.
- The other four components - GPIO, EMGD, V4L2, and UVC, are kernel driver modules. New APIs/functions are added in these modules to support early direct camera presentation in the Kernel Mode.

In Figure 2, the “App” component in the User Mode is optional. It can be used to terminate the camera kernel application and dynamically enable or disable the destination colorkey via ioctl().
Figure 2. Early Direct Camera Presentation Block Diagram
3.0  Low Level Description

3.1  Linux Kernel

3.1.1  Asynchronous AHCI port probing

The time for AHCI port probing increases with the number of SATA port available in system. Ref [1] [ARJAN] has implemented a solution for probing AHCI port in a non-staggered (parallel) fashion and this is not enabled by default in the Linux kernel tree. Therefore, to enable parallel AHCI port probing, simply add “libahci.ignore_sss=1” to the kernel boot argument. See section Section 4.2.1, “Kernel boot argument” on page 20.

3.1.2  Asynchronous Driver Initialization

Ref [1] [ARJAN] has implemented asynchronous function calls to improve boot performance. This capability is extended to parallelize the driver-device binding initialization stage for EHCI, OHCI and EMGD modules. Table 2 describes the various types of asynchronous driver-device probing options.

For EHCI and OHCI, their asynchronous device probing is synchronized at their associated driver domain, that is, ASYNC_PROBE_LOCAL_WAIT as described in section Section 4.2.7, “drivers\usb\host\ehci-pci.c” on page 31 and Section 4.2.8, “drivers\usb\host\ohci-pci.c” on page 31.

For EMGD, its asynchronous device loading is synchronized at the end of kernel initialization just before the Linux init process, see section Section 4.2.6, “drivers\gpu\drm\drm_pci.c” on page 28.

Table 2.  Type of Asynchronous Device Probing

<table>
<thead>
<tr>
<th>Asynchronous Probe Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASYNC_PROBE_NO_ASYNC</td>
<td>Asynchronous device probing is not supported.</td>
</tr>
<tr>
<td>ASYNC_PROBE_LOCAL_WAIT</td>
<td>Device probing of a driver is executed in parallel and synchronized at the end of driver_attach().</td>
</tr>
<tr>
<td>ASYNC_PROBE_LOCAL_NO_WAIT</td>
<td>Device probing of a driver is scheduled to be executed in parallel. The device probing of a driver is not synchronized at the end of driver_attach(). This allows the kernel to continue to the next step without waiting for a driver-device binding routine to complete.</td>
</tr>
<tr>
<td>ASYNC_PROBE_GLOBAL</td>
<td>Device probing of a driver is scheduled to be executed in parallel. The device probing of a driver is not synchronized at the end of driver_attach(). However, the all asynchronously scheduled routines are synchronized at the end of kernel initialization, kernel_init(), just before executing the Linux init process.</td>
</tr>
</tbody>
</table>

3.1.3  Reordering Module Initialization

Initialization of the EMGD graphic driver & EHCI host controller driver are done as early as possible before other drivers because these modules are needed by the UVC driver. This is done by changing the list in linux/drivers/Makefile.
3.2 Graphics and UVC Camera System

Assumptions:
- The camera module is a Kernel Mode application.
  - Kicked off by the EMGD driver. EMGD driver called v2g_bridge to start the camera module.
  - Can be terminated via v2g_bridge IOCTL.
- The Reverse Gear is connected to the GPIO interface. Reverse Gear On/Off state is represented by different values from the GPIO port.
- The camera is started when the Reverse Gear is ON. Camera is stopped when the Reverse Gear is OFF.
- The Graphics hardware Overlay plane is used to display the frame from the camera.

3.2.1 Kernel Module Loaded Sequence

v2g_bridge, EMGD, V4L2, UVC and GPIO are kernel built-in modules. They are loaded automatically as kernel boots. They are loaded with module_init priority level. In order to make those modules load earlier than usual and thus camera presented earlier, these modules are changed to be loaded with higher priority level, such as fs_initcall_sync or rootfs_initcall.

3.2.2 Initializing Camera Module

1. v2g_bridge exports v2g_start_camera() for external kernel module to start the camera. The EMGD driver calls v2g_start_camera() after it initializes all the necessary modules.
2. Camera initialization: v2g_bridge calls APIs exported by V4L2 and EMGD.
   - Retrieve camera list via v4l2 Enumerate Camera.
3. GPIO initialization (Reverse Gear On/Off detection initialization): v2g_bridge calls API exported by GPIO.
   - The GPIO driver is PCH GPIO (drivers/gpio/pch_gpio.c in Linux kernel).
   - v2g_bridge sets GPIO port direction as INPUT by calling pch_gpio_direction_input_kuse().
4. Display initialization: v2g_bridge calls API exported by EMGD
   - Open Display Handle of EMGD by calling emgd_get_display_handle().
   - Get screen size by calling emgd_get_screen_size().
5. Start “Reverse Gear detection” thread. The thread keeps detecting if GPIO Pin-9 is ON (meaning reverse gear is engaged) by calling pch_gpio_get_kuse(). If yes, it presents the frame from camera.
3.2.3 Reverse Gear Detect Thread

1. Read GPIO value via `pch_GPIO_get_kuse()`.
   - If “ON”, calls `v2gcamera_start_video()` to start camera
   - If “OFF”, calls `v2gcamera_stop_video()` to stop camera
2. Continue to poll GPIO pin at regular intervals
3. Within `v2gcamera_start_video()`
   a. Open camera via `v4l2_open()` exported by V4L2 module.
   b. Set camera format via `uvc_v4l2_ioctl_kernel()` with command id `VIDIOC_S_FMT` exported by UVC module.
   c. Get camera format via `uvc_v4l2_ioctl_kernel()` with command id `VIDIOC_G_FMT` exported by UVC module.
   d. Request buffer via `uvc_v4l2_ioctl_kernel()` with command id `VIDIOC_REQBUFS`.
   e. UVC driver allocates frame buffers in `uvc_alloc_buffers()` in response to command id `VIDIOC_REQBUFS`. EMGD requires that the buffer must be physically contiguous.
   f. Query buffer via `uvc_v4l2_ioctl_kernel()` with command id `VIDIOC_QUERYBUF`. Then get virtual kernel address of the buffer.
   g. Queue the buffer via `uvc_v4l2_ioctl_kernel()` with command id `VIDIOC_QBUF`.
   h. Map the buffer to GTT via `gmm_map_to_graphics()` exported by EMGD.
   i. Enable direct display via `enable_direct_display_tnc()` exported by EMGD.
   j. Start camera via `uvc_v4l2_ioctl_kernel()` with command id `VIDIOC_STREAMON`.
   k. Call `stream_video_camera()` to present camera video.

Figure 3. Camera Module Initialization
4. Within `stream_video_camera()`
   a. Reads GPIO value via `pch_gpio_get_kuse()`
      — If "OFF", sleep for certain duration and then continue to poll GPIO value.
      — If "ON":
         1. Wait for frame arrival via `uvc_v4l2_ioctl_kernel` with command ID VIDIOC_DQBUF.
         2. Present the frame via `direct_display_frame_tnc()` exported by EMGD.
         3. Queue the frame via `uvc_v4l2_ioctl_kernel()` with command ID VIDIOC_QBUF.
   b. Continue to check the GPIO value.

5. Within `v2gcamera_stop_video()`
   a. Inform Frame Delivery Thread that camera is off
   b. Disable Direct Display via `disable_direct_display_tnc()` exported by EMGD
   c. Unmap the buffer from GTT via `gmm_unmap_from_graphics()` exported by EMGD
   d. Stop the camera via `uvc_v4l2_ioctl_kernel()` with command id VIDIOC_STREAMOFF
   e. Release buffer
   f. Release camera via `v4l2_release()` exported by V4L2
Low Level Description

Figure 4. Frame Presentation (Part 1)
3.2.4 Terminating Camera Module

The application in User Mode can terminate the camera application via ioctl of \texttt{v2g\_bridge}.

1. Terminate "Reverse Gear Detect" thread
2. Call \texttt{v2gcamera\_stop\_video()}
3. Release v2g module via \texttt{v2g\_release()}

3.2.5 Enabling or Disabling Camera Destination Colorkey

The application in User Mode can set destination colorkey via IOCTL of \texttt{v2g\_bridge} to enable or disable colorkey.

1. Transfer video context, colorkey enable or disable flag, and colorkey value to EMGD.
2. EMGD sets the overlay destination colorkey to overlay or Sprite-C plane.
4.0 Implementation Details

4.1 Reference Boot Loader from Intel

4.1.1 Reducing Cache-As-Register Size

Objective: In BWG32.INC, set CAS size based on code size. The smaller the CAS size, the faster the boot time.

```
# Define the tile size
# The tile size and tile placement are critical to ensuring
# that no data loss occurs
# See BWG - chapter "Determining Tile Size"
#
-.equ TILE_SIZE , 0x000020000 # 128K
+.equ TILE_SIZE , 0x00001000 # 4K
```
4.1.2 Remove Unnecessary Features

Objective: In osv\brd_crown_bay\Makefile, remove unnecessary features that are not needed by the platform.

```c
# Remove unnecessary features from Configuration file (makefile)
CFG_USB=0          # Disabled USB Booting
CFG_ATA=1
CFG_BMP_SUPPORT=1
CFG_CONSOLE_LOCAL=0
CFG_CONSOLE_SERIAL=1
CFG_CONSOLE_SERIAL_PORT=0
CFG_CONSOLE_SERIAL_BAUD=115200
CFG_SFX=1
CFG_KEYBD_PS2_ENABLE=0   # Skip initialize for PS2 Keyboard and mouse
CFG_LEGACY=1
CFG_LEGACY_PCI32=0
CFG_PNP_BOOT=0
CFG_PROC_UPDATE=0
CFG_RTC=1
CFG_SD=1
CFG_SHELL=0            # Disappear User shell
CFG_SIO_PRESENT=1
CFG_SMM=1
CFG_SHI=1
CFG_USER_SHELL=1
CFG_VBIOS=0            # Skip Video ROM initialization
CFG_CPU_REV=1
```

### 4.1.3 Enable Direct Boot

**Objective:** In osv\brd_crown_bay\user_init.c, enable direct boot in userInit().

**Note:** Ensure that the SATA solid disk is formatted to the multi-boot file system as declared in osv\core_src\boot\protocol\intel_mbfs.h.

```c
Void userInit()
{
   #if CFG_SD
       /* Attempt to boot from SD card first */
       bootSD(MBFS_TYPE_ACTIVE, BOOT_DIRECT);
       bootSD(0, BOOT_LEGACY);
   #endif

   #if CFG_ATA
       /* Attempt an ATA boot first */
       bootATA(MBFS_TYPE_ACTIVE, BOOT_DIRECT);
       bootATA(0, BOOT_LEGACY);
   #endif
}
```
4.2 Linux Kernel

Note: The code change location described below may vary depending on the kernel version the reader intends to modify.

4.2.1 Kernel boot argument

Objective: To enable the AHCI port probing to be asynchronous, please add the following option to the kernel boot argument:

```
libahci.ignore_sss=1
```

4.2.2 include/linux/device.h

Objective: To add various levels of asynchronous support to the device driver probing mechanism:

```c
+ enum {
+   /* flags for async probe, do not use async probing for USB device */
+   ASYNC_PROBE_NO_ASYNC = 0,        /* probes sequentially */
+   ASYNC_PROBE_LOCAL_WAIT,          /* returns after all local probes are done*/
+   ASYNC_PROBE_LOCAL_NO_WAIT,       /* returns after all local probes are
+                                           scheduled */
+   ASYNC_PROBE_GLOBAL,              /* returns after all probes are scheduled
+                                           globally */
+   };;

struct device_driver {
   const char *name;
   struct bus_type *bus;
   struct module *owner;
   const char *mod_name; /* used for built-in modules */
   bool suppress_bind_attrs; /* disables bind/unbind via sysfs */

   const struct of_device_id *of_match_table;
   int (*probe) (struct device *dev);
   int (*remove) (struct device *dev);
   void (*shutdown) (struct device *dev);
   int (*suspend) (struct device *dev, pm_message_t state);
   int (*resume) (struct device *dev);
   const struct attribute_group **groups;
   const struct dev_pmops *pm;
   struct driver_private *p;
+   unsigned int async_probe; /* Use ASYNC_PROBE XXX to control */
};
```
4.2.3 **drivers\Makefile**

**Objective:** To sequence the module loading for ATA & SCSI to earlier than before.

```
obj-$(CONFIG_VIRTIO) += virtio/
obj-$(CONFIG_XEN) += xen/

# regulators early, since some subsystems rely on them to initialize
obj-$(CONFIG_REGULATOR) += regulator/

# load modules earlier
+obj-$(CONFIG_SCSI) += scsi/
+obj-$(CONFIG_ATA) += ata/

# tty/ comes before char/ so that the VT console is the boot-time
# default.
obj-y += tty/
obj-y += char/

# Ay line-43
obj-$(CONFIG_NUBUS) += nubus/
obj-y += macintosh/
obj-$(CONFIG_IDE) += ide/
-obj-$(CONFIG_SCSI) += scsi/ /* Move module loading earlier */
-obj-$(CONFIG_ATA) += ata/ /* Move module loading earlier */
obj-$(CONFIG_TARGET_CORE) += target/
obj-$(CONFIG_MTD) += mtd/
```
4.2.4  drivers\ata\libata-core.c

Objective: To add module parameters that allow users to disable SATA probing on a port that is not attached to actual SATA disk.

```c
# At line-151
static int ata_probe_timeout;
module_param(ata_probe_timeout, int, 0444);
MODULE_PARM_DESC(ata_probe_timeout, "Set ATA probing timeout (seconds)");

+static int libata_disable_ports;
+module_param_named(disable_ports, libata_disable_ports, int, 0444);
+MODULE_PARM_DESC(disable_ports, "Disable specified ports (as bitmask) at boot time");

int libata_noacpi = 0;
module_param(acpi, libata_noacpi, int, 0444);
MODULE_PARM_DESC(noacpi, "Disable the use of ACPI in probe/suspend/resume (0-off [default], 1=on)");

# At line-5966
int ata_host_register(struct ata_host *host, struct scsi_host_template *sht)
{

# At line-6034
    /* perform each probe asynchronously */
    for (i = 0; i < host->n_ports; i++) {
        struct ata_port *ap = host->ports[i];
        /*
        * check if the port is specified as
        * 'to disable' by module parameter
        */
        if (libata_disable_ports & (1 << ap->port_no)) {
            continue;
        }
        async_schedule(async_port_probe, ap);
    }
}
4.2.5  drivers\base\dd.c

Objective: To implement an asynchronous driver loading mechanism to driver_attach().
```c
#include <linux/async.h>
#include <linux/pm_runtime.h>
#include <linux/slab.h>
#include "base.h"

int driver_probe_device(struct device_driver *drv, struct device *dev)
{
}

+/**
+ * struct async_probe_data - used in async probe
+ * @dev: driver to bind a device to
+ * @dev: device to try to bind to the driver
+ * @dev: domain in which async probing runs in
+ */
+struct async_probe_data {
+    struct device_driver *drv;
+    struct device *dev;
+    struct list_head running;
+    unsigned int async探 probe;
+};

+static void driver_probe_device_async(void *data, async_cookie_t cookie)
+{
+    struct async_probe_data *probe_data = data;
+    /*
+    * Holding dev->parent will prevent us from running asynchronously.
+    * Since it is only required in probing USB device, we remove the locking.
+    * Thus, do not use asynchronous probing for USB device.
+    */
+    device_lock(probe_data->dev);
+    if (!probe_data->dev->driver)
+        driver_probe_device(probe_data->drv, probe_data->dev);
+    device_unlock(probe_data->dev);
```
Implementation Details

```c
+   /* got what we need, free the memory */
+   kfree(data);
+

static int __device_attach(struct device_driver *drv, void *data)
{
}

# At line-265
static int __driver_attach(struct device *dev, void *data)
{
}

+static int __driver_attach_async(struct device *dev, void *probe_data)
+{
+   struct async_probe_data *per_driver_probe_data = probe_data;
+   struct async_probe_data *per_device_probe_data;
+   /*
+   * Lock device and try to bind to it. We drop the error
+   * here and always return 0, because we need to keep trying
+   * to bind to devices and some drivers will return an error
+   * simply if it didn't support the device.
+   *
+   * driver_probe_device_async() will spit a warning if there
+   * is an error.
+   */
+   if (!driver_match_device(per_driver_probe_data->drv, dev))
+      return 0;
+
+   /* allocate memory for each probing thread */
+   per_device_probe_data = kmalloc(sizeof(struct async_probe_data),
+                                GFP_ATOMIC);
+   if (!per_device_probe_data) {
+      pr_err("%s: failed to allocate memory for async_probe_data\n", __func__);
+      return -ENOMEM;
+   }
+
+   /* copy the memory */
```
```c
+    memcpy(per_device_probe_data, per_driver_probe_data,
+            sizeof(struct async_probe_data));
+    /* assign dev */
+    per_device_probe_data->dev = dev;
+
+    switch (per_driver_probe_data->async_probe) {
+      case ASYNC_PROBE_LOCAL_WAIT:
+        /* scheduled in shared local domain */
+        async_schedule_domain(driver_probe_device_async,
+                               per_device_probe_data,
+                               &per_driver_probe_data->running);
+        break;
+      case ASYNC_PROBE_LOCAL_NO_WAIT:
+        /* scheduled in own domain */
+        INIT_LIST_HEAD(&per_device_probe_data->running);
+        async_schedule_domain(driver_probe_device_async,
+                               per_device_probe_data,
+                               &per_device_probe_data->running);
+        break;
+      case ASYNC_PROBE_GLOBAL:
+        /* scheduled in global domain */
+        async_schedule(driver_probe_device_async, per_device_probe_data);
+        break;
+      default:
+        /* should not reach here... */
+        break;
+    }
+    return 0;
+}

# At line-303
int driver_attach(struct device_driver *drv) {
    return bus_for_each_dev(drv->bus, NULL, drv, __driver_attach);
    struct async_probe_data per_driver_probe_data;
    
    /* do sequential probing if parallel is not supported */
    if (drv->async_probe == ASYNC_PROBE_NO_ASYNC)
        return bus_for_each_dev(drv->bus, NULL, drv, __driver_attach);
    
    /* fill in the struct */
```
+ per_driver_probe_data.drv = drv;
+ per_driver_probe_data.dev = NULL;
+ INIT_LIST_HEAD(&per_driver_probe_data.running);
+ per_driver_probe_data.async_probe = drv->async_probe;
+
+ bus_for_each_dev(drv->bus, NULL, &per_driver_probe_data,
+                  driver_attach_async);
+ /* wait for return if required */
+ if (drv->async_probe == ASYNC_PROBE_LOCAL_WAIT)
+    async_synchronize_full_domain(&per_driver_probe_data.running);
+
+ return 0;
}

EXPORT_SYMBOL_GPL(driver_attach);
4.2.6 drivers\gpu\drm\drm_pci.c

**Objective:** To make `drm_pci_init()` load the DRM device asynchronously so as to not block other kernel module initialization.
Implementation Details

# At line-41
#include <linux/slab.h>
#include <linux/dma-mapping.h>
#include <linux/async.h>
#include "drmP.h"

int dm_get_pci_dev(struct pci_dev *pdev, const struct pci_device_id *ent,
                   struct drm_driver *driver)
{
}

EXPORT_SYMBOL(dm_get_pci_dev);

# At line-390
+struct async_arguments {
  + struct pci_dev *pdev;
  + const struct pci_device_id *pid;
  + struct drm_driver *driver;
+};

+static void dm_get_pci_dev_async(void *data, async_cookie_t cookie)
+{
  + struct async_arguments *args = (struct async_arguments *)data;
  + struct pci_dev *pdev = args->pdev;
  + const struct pci_device_id *pid = args->pid;
  + struct drm_driver *driver = args->driver;
  +
  + /* got what we need, free the allocated-per-thread structure */
  + kfree(data);
  + dm_get_pci_dev(pdev, pid, driver);
+}

# At line-403
int dm_pci_init(struct drm_driver *driver, struct pci_driver *pdev)
{
  struct pci_dev *pdev = NULL;
  const struct pci_device_id *pid;
  + struct async_arguments *args = NULL;
```c
int i;
DRM_DEBUG("\n");
INIT_LIST_HEAD(idriver->device_list);
driver->kdriver.pci = pdriver;
driver->bus = &drm_pci_bus;
if (driver->driver_features & DRIVER_MODESET)
    return pci_register_driver(pdriver);
/* If not using KMS, fall back to stealth mode manual scanning. */
for (i = 0; pdriver->id_table[i].vendor != 0; i++) {
    pid = &pdriver->id_table[i];

    /* Loop around setting up a DRM device for each PCI device
       * matching our ID and device class. If we had the internal
       * function that pci_get_subsys and pci_get_class used, we'd
       * be able to just pass pid in instead of doing a two-stage
       * thing.
       */
    pdev = NULL;
    while ((pdev =
               pci_get_subsys(pid->vendor, pid->device, pid->subvendor,
               pid->subdevice, pdev)) != NULL) {
        if (((pdev->class & pid->class_mask) != pid->class)
            continue;
        /* stealth mode requires a manual probe */
        drm_get_pci_dev(pdev, pid, driver);
        /*
        * allocate per-thread structure to pass the arguments
        * which will be freed later in each of the function thread
        */
        args = kmalloc(sizeof(*args), GFP_KERNEL);
        if (args == NULL)
            return -ENOMEM;
        args->pdev = pdev;
        args->pid = pid;
        args->driver = driver;
        async_schedule(drm_get_pci_dev_async, args);
    }
    return 0;
}
```
Implementation Details

4.2.7 drivers\usb\host\ehci-pci.c

**Objective:** To load the EHCI PCI driver asynchronously.

```c
# At line-491
static struct pci_driver ehci_pci_driver = {
    .name = (char *) hcd_name,
    .id_table = pci_ids,
    .probe = usb_hcd_pci_probe,
    .remove = usb_hcd_pci_remove,
    .shutdown = usb_hcd_pci_shutdown,
    .driver = {
        #ifdef CONFIG_PM_SLEEP
        .pm = &usb_hcd_pci_pm_ops,
        #endif
        + async_probe = ASYNC_PROBE_LOCAL_WAIT,
    },
};
```

4.2.8 drivers\usb\host\ohci-pci.c

**Objective:** To load the OHCI PCI driver asynchronously.

```c
# At line-433
static struct pci_driver ohci_pci_driver = {
    .name = (char *) hcd_name,
    .id_table = pci_ids,
    .probe = usb_hcd_pci_probe,
    .remove = usb_hcd_pci_remove,
    .shutdown = usb_hcd_pci_shutdown,
    .driver = {
        #ifdef CONFIG_PM_SLEEP
        .pm = &usb_hcd_pci_pm_ops,
        #endif
        + async_probe = ASYNC_PROBE_LOCAL_NO_WAIT,
    },
};
```

4.2.9 init\initramfs.c

**Objective:** To add asynchronous capability to initramfs/initrd loading for boot performance.

In systems such as Android that do not use initramfs, this change is not necessary.
```c
#include <linux/dirent.h>
#include <linux/syscalls.h>
#include <linux/utime.h>
+#include <linux/async.h>     // Added in for asynchronous function call

+static void __init unpack_rootfs(void *data, async_cookie_t cookie)
+{
+    char *err;
+
+    +#ifdef CONFIG_BLK_DEV_RAM
+    +    int fd;
+    +    printk(KERN_INFO "Trying to unpack rootfs image as initramfs\n");
+    +    err = unpack_to_rootfs((char *)initrd_start,
+    +        initrd_end - initrd_start);
+    +    if (!err) {
+    +        free_initrd();
+    +        return;
+    +    } else {
+    +        clean_rootfs();
+    +        unpack_to_rootfs(__initramfs_start, __initramfs_size);
+    +    }
+    +    printk(KERN_INFO "rootfs image is not initramfs (\%s)"
+    +        "; looks like an initrd\n", err);
+    +    fd = sys_open((const char _)user__force *) "/initrd.image",
+    +        O_WRONLY|O_CREAT, 0700);
+    +    if (fd >= 0) {
+    +        sys_write(fd, (char *)initrd_start,
+    +            initrd_end - initrd_start);
+    +        sys_close(fd);
+    +        free_initrd();
+    +    }
+    +#else
+    +    printk(KERN_INFO "Unpacking initramfs...\n");
+    +    err = unpack_to_rootfs((char *)initrd_start,
+    +        initrd_end - initrd_start);
+    +    if (err)
+    +        printk(KERN_EMERG "Iniramfs unpacking failed: \%s\n", err);
+    +        free_initrd();
+    +#endif
```
Implementation Details

```c
+

static int __init populate_rootfs(void)
{
    char *err = unpack_to_rootfs(__initramfs_start, __initramfs_size);
    if (err)
        panic(err); /* Failed to decompress INTERNAL initramfs */
    if (initrd_start) {
        // The original logics are moved to unpack_rootfs() function
        // so that we can asynchronously call it at boot time
        // for boot performance.
        async_schedule(unpack_rootfs, NULL);
    }
    return 0;
}
```
4.3 Graphics and UVC Camera System

4.3.1 Embedded Media & Graphics Driver (EMGD)

/*@ 
* emgd_get_display_handle (): Get screen width and height.
* 
* Parameter:
* display_handle - OUT display handle
* screen_num - IN screen id
* Return: 0 on success
*/
int emgd_get_display_handle (void **display_handle, int screen_number)

/*@ 
* emgd_get_screen_size(): Get screen width and height.
* 
* Parameter:
* screen_num - IN screen id
* width - OUT width
* height - OUT height
* Return: 0 on success
*/
int emgd_get_screen_size (int screen_num, unsigned short * width,
unsigned short * height);

/*@ 
* ovl_set_dest_colorkey_tnc (): Enable or disable destination colorkey.
* 
* Parameter:
* igd_dd_context_t dd_context: IN video context.
* igd_oval_color_key_info_t *color_key: IN colorkey flag and value structure.
* Return: 0 on success
*/
int ovl_set_dest_colorkey_tnc (igd_dd_context_t dd_context,
igd_oval_color_key_info_t *color_key);

/*@ 
* omm_map_to_graphics(): Facilitates direct display of contiguous video input
* buffers by mapping the specified block into the "graphics aperture" via the
* GIT.
* Parameter:
* phys_addr: IN physical address of the buffer to map to GIT.
* size: IN size of the buffer.
* offset: OUT GIT offset of the buffer.
* Return: 0 on success
*/
int omm_map_to_graphics (unsigned long phys_addr,
unsigned long size,
unsigned long *offset)
4.3.2 Video For Linux 2 (V4L2)

```c
/*
 * v4l2 Enumerate Camera[]): V4L2 Exported API. Get v4l2 camera list.
 * Parameter:
 * minors: OUT camera device minor list
 * count: OUT camera quantity
 * Return: 0 on success.
 */
int v4l2 Enumerate Camera(unsigned int* minors, int* count);
```

4.3.3 GPIO

```c
/*
 * pch_gpio_direction_input_kuse []): GPIO Exported API. Set GPIO Pin as input.
 * Parameter:
 * nr: IN GPIO Pin to set
 * Return: 0 on success.
 */
int pch_gpio_direction_input_kuse(unsigned nr)

/*
 * int pch_gpio_get_kuse(): GPIO Exported API. Read a GPIO Pin value.
 * Parameter:
 * nr: IN GPIO Pin ID
 * Return: pin value.
 */
int pch_gpio_get_kuse(unsigned nr)
```
4.3.4 v2g_bridge

```c
int v2g_camera_uninit(void);

ioctl() supports V2G_TERMINATE_CAMERAVIDEO - Terminate camera application.
/
* v2g_camera_uninit(): v2g_bridge module exported API. Terminate camera
* application.
* Parameter: void
* Return: 0 on success.
*/

ioctl() supports V2G_SET_DEST_COLORKEY - Enable or Disable destination colorkey
* for early camera.
/
* v2g_camera_set_dest_colorkey(): v2g_bridge module exported API. Enable or
* Disable destination colorkey for early camera.
* Parameter: igd_ovl_color关键_info_t *br: in parameter colorkey flag and value
* structure.
* Return: 0 on success.
*/
typedef struct igd_ovl_color_key_info {
   /*! Low end src color key value */
   unsigned long src_lo;
   /*! High end src color key value */
   unsigned long src_hi;
   /*! Dest color key value. If the destination pixel matches the
   * dest color key then the source pixel from the video surface is displayed.
   * Otherwise, the destination pixel is displayed */
   unsigned long dest;
   /*! Enable and disable the src and dest color key.
   * See @ref igd_ovl_color_key_info_flags */
   unsigned long flags;
} igd_ovl_color_key_info_t;

int v2g_camera_set_dest_colorkey(igd_ovl_color_key_info_t *br);
```
4.3.5 USB Video Capture (UVC)

```
/*
 * uvc_v412_ioctl_kernel UVC exported API. UVC ioctl.
 * Parameter:
 * file: IN camera handle
 * cmd: IN command id
 * arg: IN/OUT data structure pointer.
 * Return: 0 on success.
 */
long uvc_v412_ioctl_kernel(struct file *file,
               unsigned int cmd, unsigned long arg);
```
5.0 Results

The total time taken for early camera presentation depends on the time taken in the following stages:

1. Platform firmware (platform basic and advanced initialization + kernel image direct-boot)
2. Kernel image decompression and kernel relocation to high-memory
3. Kernel initialization sequence, such as, kernel stack, page table, GDT & IDT setup, traps, IRQs, soft-irqs, etc.
4. Driver initialization and device probing, such as, EHCI, OHCI, AHCI & EMGD
5. V4L2 subsystem start-up
6. V2G_bridge kernel module start-up

This paper does not discuss the tool used to capture measurement. For the readers’ reference, we have used the following tools:

- Reference Boot Loader from Intel: Enable “Profile Macro” from the makefile. This will display boot measurement on shell.
- Linux OS:
  - Bootchart (http://www.bootchart.org/) to profile kernel boot sequence
  - Grabserial (http://elinux.org/Grabserial) to collect messages written to serial port from target board.

Boot measurement for stage (1) and stage (4) to stage (6) above are easily analyzed by the above-mentioned tool. However, for stage (2) and stage (3) the boot measurement is not trivial and no optimization has been focused in these two stages yet. Based on our crude measurement, these two stages take about 1s to 1.3s.
5.1 Platform Firmware

Table 3. Time Measurement of the Reference Boot Loader from Intel Stage (in ms)

<table>
<thead>
<tr>
<th></th>
<th>Before Tuning</th>
<th>After Tuning</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>112</td>
<td>57</td>
<td>-55</td>
</tr>
<tr>
<td>Advanced</td>
<td>426</td>
<td>390</td>
<td>-36</td>
</tr>
<tr>
<td>Loading Kernel Image†</td>
<td>662</td>
<td>412</td>
<td>-250</td>
</tr>
<tr>
<td>Total</td>
<td>1200</td>
<td>859</td>
<td>-341</td>
</tr>
</tbody>
</table>

Note: Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Note: The measurement is taken under the following conditions:
- Intel® Atom Processor™ Processor E6XX (1.3GHz) with Intel® Platform Controller Hub EG20T.
- Intel® Atom™ Processor E660 with Intel® Platform Controller Hub EG20T Development Kit Platform Reference Boot Loader from Intel version 4.3.6.1 with splash screen support.
- Android 2.3.7 kernel image is loaded from Kingston* Solid State Disk (Kingston SSDNow V+200 120GB).
- Tested time measurement of the Reference Boot Loader from Intel Stage (in ms).
- † The measurement is taken with splash screen support. Without splash screen feature, it saves another 200ms.

5.2 Linux Kernel, EMGD & UVC

5.2.1 Before Optimization

From Figure 6, it is clear that initialization for EHCI, EMGD and OHCI are noticeably sequential. Figure 6 also shows that the Linux DRM graphic driver, emgd_init(), is initialized after USB EHCI initialization for both port-0 & port-2 are completed at about 0.24s. Another observation is that the SATA controller probing for port 0 and port 1 is staggered in nature, that is, the SATA controller probing for port 1 starts after port 0 probing is complete. As a result the SATA probing phase takes up about 600ms in total.

In Figure 7, the v2g_bridge kernel module starts streaming video frame from the UVC device to display at 1.16s. Note that Figure 6 does not show the USB device probing process whereby UVC device is loaded.

Figure 6. The Original Boot Graph
5.2.2 Post Optimization

By exploiting multi-thread support on the Linux system, the initialization of EHCI, EMGD and OHCI modules are done asynchronously as shown in Figure 8. The DRM graphic module, `emgd_init()`, is initialized in parallel with EHCI controller, `ehci_hcd_init()`, at 0.18s, that is, about 60ms earlier than original boot performance shown in Figure 6. Another observation is that OHCI initializations for port 1 to port 4 are also executed in parallel.

In Figure 8, the SATA controller AHCI device is probed in a non-staggered manner and this leads to time reduction of about 300ms for the AHCI device probing process. See note below.

The aforementioned synchronism in module initialization has reduced the total Linux kernel initialization time by 500ms, noticeable in the ending time of Linux boot graph in Figure 6 and Figure 8. See note below.

As the UVC device driver loading is done by the USB bus driver in an asynchronous fashion post Linux kernel initialization, the time saving of 300ms in AHCI device probing does not contribute directly towards faster camera frame streaming. In Figure 9, the `v2g_bridge` kernel module starts streaming video frames from the UVC device to display at 1.06s, which is about 100ms faster. See note below.

Note: Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Note: The measurement is taken under the following conditions:

- Intel® Atom Processor™ Processor E6XX (1.3GHz) with Intel® Platform Controller Hub EG20T.
- Intel® Atom™ Processor E660 with Intel® Platform Controller Hub EG20T Development Kit Platform Reference Boot Loader from Intel version 4.3.6.1 with splash screen support
- Android 2.3.7 kernel image is loaded from Kingston* Solid State Disk (Kingston SSDNow V+200 120GB).
- Tested time measurement of the Reference Boot Loader from Intel Stage (in ms).
- The measurement is taken with splash screen support. Without splash screen feature, it saves another 200ms.
Results

1 Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

The measurement was done on the following:

Hardware configuration:
- Intel® Atom™ Processor E660 (1.3GHz) with Intel® Platform Controller Hub EG20T Development Kit
- USB Video Capture device (Logitech HD Pro WebCam C920)
• SATA Solid State Disk (Kingston SSDNow V+200 120GB)
• LVDS Panel

Software environment:
• Android 2.3.7 (Ginger Bread) - Linux 2.6
• Android BSP 1.2 for Intel Atom Processor E6XX

Tested time measurement of the Reference Boot Loader from Intel Stage (in ms).
The measurement is taken with splash screen support. Without splash screen feature, it saves another 200ms.
6.0 Setup for Android

1. Check out Android 2.3.7 source code repository and Android BSP 1.2
2. Patch Android_Early_Camera.patch. Go to the directory of your Android source code repository ($Android_Repo) and run:
   a. cd hardware/intel/linux-2.6
   b. patch -p1 <<location_of_the_patch>/Android_Early_Camera.patch
3. Build the image
   a. cd $Android_Repo
   b. source build/envsetup.sh
   c. lunch
   d. select 7 e660 eg20t-eng
   e. make menuconfig
   f. Select "Driver Drivers -->", "Staging drivers -->", "Intel V2G_BRIDGE Kernel Module Driver"
   g. Select "*" for it. And then select "Build V2G_BRIDGE kernel driver as (Release)-->
   h. Exit and save your configuration.
   i. make -j4
4. Use $Android_Repo/scripts/doit.sh to install the image to a hard disk or SD memory card.
5. Insert the hard disk or memory card to the board.
6. Connect the GPIO switch to the board.
7. Plug a USB camera onto the board.
8. Power on the board. Turn the GPIO switch On or Off to open or close early camera.
7.0 Setup for Tizen Linux

1. Checkout the Tizen kernel source
2. Patch the kernel source with Tizen_Early_Camera.patch
   Go into the kernel source directory, and run
   a. cd <kernel_source_directory>
   b. patch -p1 < <location_of_the_patch>/Tizen_Early_Camera.patch
3. Use the kernel configuration, fast-camera-config-USB
   a. cp <location_of_the_config_file>/fast-camera-config-USB .config
4. Compile a new kernel
   a. ARCH=i386 make -j4
5. Prepare Tizen boot disk using image downloaded from http://download.tizen.org
   a. bunzip2 -ck <location_of_the_image>/<image_name> > /dev/sdX; sync
      * /dev/sdX – the device node name of the boot disk
6. Prepare the Master Boot Record file which will be used to create direct boot image.
   Create a temporary folder, and do everything there.
   a. mkdir ~/temp; cd ~/temp
   b. dd if=/dev/sdX of=~/temp/mbr.bin bs=512 count=1; sync
7. Preparation for Direct-Boot disk. These files are required:
   a. the compiled kernel, kernel
   b. cp <kernel_source_directory>/arch/x86/boot/bzImage ~/temp/kernel
   c. the ruby script to convert the boot disk, journal.rb
   d. cp journal.rb ~/temp
   e. the direct boot image configuration, tizen.xml
   f. cp tizen.xml ~/temp
   g. the kernel command line, cmdline
      echo "ro root=/dev/sda3 rootwait rootfstype=ext4
      emgd.configid=1 libahci.ignore_sss=1" > ~/temp/cmdline
8. Execute the ruby script, install ruby interpreter with "yum install ruby" if it was not
   installed.
   a. cd ~/temp
   b. ruby journal.rb ifile=./tizen.xml device=/dev/sdX
9. Edit the /etc/fstab in platform partition in Direct Boot disk
   a. mount /dev/sdX3 /mnt
   b. Edit /mnt/etc/fstab, comment the second line by adding #
#UUID=cbaba816-52f4-4011-a7a3-52ff09184c66 /boot ext4 defaults,noatime
0 0
c. umount /mnt

10. Insert the hard disk or memory card to the board.
11. Connect the GPIO switch to the board.
12. Plug a USB camera onto the board.
13. Power on the board. Turn the GPIO switch On or Off to open or close early camera.