

RE: [patch 2.6.12-rc3] dell_rbu: Resubmitting patch for new DellBIOS update driver

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2005-06/0725.html>

Abhay_Salunke_at_Dell.com

Date: 06/03/05

Date: Thu, 2 Jun 2005 17:25:46 -0500

To: <marcel@holtmann.org>

Please find a new patch...

> *please fix the coding style. We use tabs instead of spaces.*

Cleaned up

>

> *Make sure that all functions are static and clean your namespace. Even*

> *if they are static it is still unclean.*

>

Yes all functions are static and also removed do_packet_read which was unused

> *It is <linux/firmware.h> and not "linux/firmware.h".*

>

done

> *The Kconfig is missing a "select FW_LOADER".*

>

Done.

This driver also has made the mono_size and packet_size entries writable to allow canceling (freeing) of RBU image downloaded.

Thanks for your inputs,

Abhay

```
diff -uprN linux-2.6.11.8.ORIG/Documentation/DELL_RBU.txt
```

```
linux-2.6.11.8/Documentation/DELL_RBU.txt
```

```
--- linux-2.6.11.8.ORIG/Documentation/DELL_RBU.txt 1969-12-31
```

```
18:00:00.000000000 -0600
```

```
+++ linux-2.6.11.8/Documentation/DELL_RBU.txt 2005-06-02
```

```
13:23:36.000000000 -0500
```

```
@@ -0,0 +1,70 @@
```

```
+Purpose:
```

```
+Demonstrate the usage of the DELL_RBU (DELL Remote BIOS Update) driver
```

```
+for updating BIOS images on Dell hardware.
```

```
+
```

```
+Scope:
```

```
+This document discusses the functionality of the DELL_RBU driver.
```

Linux-Kernel: RE: [patch 2.6.12-rc3] dell_rbu: Resubmitting patch for new DellBIOS update driver

+This driver is required by BIOS update applications shipped by DELL for updating

+BIOS on DELL servers and client systems.

+

+Overview:

+The rbu driver is designed to be running on 2.6 kernel.

+This driver is one single dell_rbu.c file (approx 800 lines total).

+This driver utilizes the hotplug interface for downloading the BIOS update image

+in the contiguous or packetized memory depending upon the update type.

+The BIOS then scans the memory to find the image and will then update itself.

+There are basically two different mechanisms for writing the BIOS image in to

+contiguous memory

+1> By writing the image to one monolithic chunk of contiguous physical memory.

+2> By writing image in to smaller packet chunks of contiguous physical memory.

+The update mechanism is determined by the update application based on the

+particular system type.

+

+Update mechanism using single physical chunk of memory:

+The rbu driver on its load time creates the following entries in sysfs

+/sys/firmware/dell_rbu/monolithic/mono_name

+/sys/firmware/dell_rbu/monolithic/mono_size

+/sys/firmware/dell_rbu/packetized/packet_name

+/sys/firmware/dell_rbu/packetized/packet_size

+

+Steps to update the BIOS image:

+

+1> Copy the image file in to /lib/firmware

+2> echo the image name in to /sys/firmware/dell_rbu/xxxx/xxxx_name

+

+This will generate a hotplug event and the data form the image file is

+transferred to the memory.

+Here xxxx stands for the type of BIOS update mechanism chosen by choosing

+monolithic the image is copied to contiguous physical pages and by choosing

+the mechanism as packetized the image is treated as one single packet and the

+packet size is set to the first packet image size. If a new image packet of

+different size form the previous is copied then all the previous packets are

+freed and this packet's size is treated as new packet size.

+

+On a driver unload all the allocated memory is freed.

+The user should not unload the driver after downloading the new BIOS

RE: [patch 2.6.12-rc3] dell_rbu: Resubmitting patch for new DellBIOS update driver

image

+if it wants to update BIOS with that image.

+The user can overwrite the previously loaded monolithic image by echoing a

+new file name string to /sys/firmware/dell_rbu/monolithic/mono_name.

Make sure

+the file is present in /lib/firmware. If the image size is more than

+previous image then the previous image is freed and the new allocation is made.

+

+The user can know of a successful allocation by reading the size files.

+cat /sys/firmware/dell_rbu/monolithic/mono_size

+

+Update using smaller chunks (packets) of contiguous memory:

+The disadvantage of contiguous allocation is that it may not be always possible

+to get that size of contiguous chunk of available physical pages as in most

+Linux systems the memory gets fragmented immediately after a reboot.

+The update using smaller chunks fixes this issue; it also requires the BIOS on

+the system to support this feature; the update application needs to query this

+with the BIOS on the system before using this technique.

+

+

+NOTE:

+After updating the BIOS image the application needs to communicate with the BIOS

+for enabling the update on the next reboot. The application can then choose to

+reboot the system immediately or not reboot the system and leave up to the user

+to do a reboot.

+

+

diff -uprN linux-2.6.11.8.ORIG/drivers/firmware/dell_rbu.c

linux-2.6.11.8/drivers/firmware/dell_rbu.c

--- linux-2.6.11.8.ORIG/drivers/firmware/dell_rbu.c 1969-12-31

18:00:00.000000000 -0600

+++ linux-2.6.11.8/drivers/firmware/dell_rbu.c 2005-06-02

17:01:54.632598440 -0500

@@ -0,0 +1,681 @@

+/*

+ * dell_rbu.c

+ * Bios Update driver for Dell systems

+ * Author: Dell Inc

+ * Abhay Salunke <abhay_salunke@dell.com>

+ *

+ * Copyright (C) 2005 Dell Inc.

+ *

Linux-Kernel: RE: [patch 2.6.12-rc3] dell_rbu: Resubmitting patch for new DellBIOS update driver

```
+ * Remote BIOS Update (rbu) driver is used for updating DELL BIOS by
creating
+ * entries in the /sys file systems on Linux 2.6 and higher kernels.
+ * It uses the hotplug interface for getting the image in to memory.
+ * The driver supports two mechanism to update the BIOS namely
contiguous and
+ * packetized. Both these methods still require to have some
application to set
+ * the CMOS bit indicating the BIOS to update itself after a reboot.
+ * In both the methods the image file name needs to be specified for
hot plugging
+ * the image file.
+ *
+ * Contiguous method:
+ * This driver writes the incoming data in a monolithic image by
allocating
+ * contiguous physical pages large enough to accommodate the incoming
BIOS image
+ * size.
+ *
+ * Packetized method:
+ * The driver writes the incoming packet image by allocating a new
packet on
+ * every time the packet image name is written.
+ * This driver requires an application to break the BIOS image in to
fixed sized
+ * packet chunks and each packet is written as a hotplug image.
+ *
+ * This program is free software; you can redistribute it and/or modify
+ * it under the terms of the GNU General Public License v2.0 as
published by
+ * the Free Software Foundation
+ *
+ * This program is distributed in the hope that it will be useful,
+ * but WITHOUT ANY WARRANTY; without even the implied warranty of
+ * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
+ * GNU General Public License for more details.
+ */
+#include <linux/version.h>
+#include <linux/config.h>
+#include <linux/init.h>
+#include <linux/module.h>
+#include <linux/string.h>
+#include <linux/errno.h>
+#include <linux/blkdev.h>
+#include <linux/device.h>
+#include <linux/spinlock.h>
+#include <linux/moduleparam.h>
+#include <linux/firmware.h>
+
+MODULE_AUTHOR("Abhay Salunke <abhay_salunke@dell.com>");
```

RE: [patch 2.6.12-rc3] dell_rbu: Resubmitting patch for new DellBIOS update driver


```

+static int fill_last_packet(void *data, size_t length)
+{
+ struct list_head *ptemp_list;
+ struct packet_data *ppacket = NULL;
+ int packet_count = 0;
+
+ pr_debug("fill_last_packet: entry \n");
+
+ /* check if we have any packets */
+ if (0 == rbu_data.num_packets) {
+ pr_debug("fill_last_packet: num_packets=0\n");
+ return -ENOMEM;
+ }
+
+ packet_count = rbu_data.num_packets;
+
+ ptemp_list = (&packet_data_head.list)->next;
+
+ while(--packet_count) {
+ ptemp_list = ptemp_list->next;
+ }
+
+ ppacket = list_entry(ptemp_list,struct packet_data, list);
+
+ if ((rbu_data.packet_write_count + length) >
rbu_data.packet_size) {
+ printk(KERN_WARNING "fill_last_packet: packet size data
"
+ "overrun\n");
+ return -ENOMEM;
+ }
+
+ pr_debug("fill_last_packet : buffer = %p\n", ppacket->data);
+
+ /* copy the incoming data in to the new buffer */
+ memcpy((ppacket->data + rbu_data.packet_write_count),
+ data, length);
+
+ if ((rbu_data.packet_write_count + length) ==
rbu_data.packet_size) {
+ /*
+ this was the last data chunk in the packet
+ so reinitialize the packet data counter to zero
+ */
+ rbu_data.packet_write_count = 0;
+ } else {
+ rbu_data.packet_write_count += length;
+ }
+ pr_debug("fill_last_packet: exit \n");
+ return 0;
+}

```

```
+
+/*
+ get_free_pages_limited:
+ This is a helper function which allocates free pages based on an upper
limit.
+ On x86_64 or 64 bit arch the memory allocation goes above 4GB space
which is
+ not addressable by the BIOS. This function tries to get allocation
below the
+ limit (4GB) address. It first tries to allocate memory normally using
the
+ GFP_KERNEL argument if the incoming limit is non-zero and if the
returned
+ physical memory address exceeds the upper limit, the allocated pages
are freed
+ and the memory is reallocated using the GFP_DMA argument.
+*/
+static void *get_free_pages_limited(unsigned long size,
+ int *ordernum,
+ unsigned long limit)
+{
+ unsigned long img_buf_phys_addr;
+ void *pbuf = NULL;
+
+ *ordernum = get_order(size);
+ /*
+ Check if we are not getting a very large file.
+ This can happen as a user error in entering the file size
+ */
+ if (*ordernum == BITS_PER_LONG) {
+ pr_debug("get_free_pages_limited: Incoming size is"
+ " very large\n");
+ return NULL;
+ }
+
+ /* try allocating a new buffer to fit the request */
+ pbuf =(unsigned char *)__get_free_pages(GFP_KERNEL, *ordernum);
+
+ if (pbuf != NULL) {
+ /* check if the image is with in limits */
+ img_buf_phys_addr = (unsigned long)virt_to_phys(pbuf);
+
+ if ((limit != 0) && ((img_buf_phys_addr + size) >
limit)) {
+ pr_debug("Got memory above 4GB range, free this
"
+ "and try with DMA memory\n");
+ /* free this memory as we need it with in 4GB
range */
+ free_pages ((unsigned long)pbuf, *ordernum);
+ }
+ }
+*/
```

```
+ Try allocating a new buffer from the GFP_DMA
range
+ as it is with in 16MB range.
+ */
+ pbuf =(unsigned char
*)__get_free_pages(GFP_DMA,
+ *ordernum);
+ if (pbuf == NULL)
+ pr_debug("Failed to get memory of size
%ld "
+ "using GFP_DMA\n", size);
+ }
+ }
+ return pbuf;
+}
+
+static int create_packet(size_t length)
+{
+ struct packet_data *newpacket;
+ int ordernum = 0;
+
+ pr_debug("create_packet: entry \n");
+
+ if (rbu_data.packetsize == 0 ) {
+ pr_debug("create_packet: packetsize not specified\n");
+ return -EINVAL;
+ }
+
+ newpacket = kmalloc(sizeof(struct packet_data) ,GFP_KERNEL);
+ if(newpacket == NULL) {
+ printk(KERN_WARNING "create_packet: failed to allocate
new "
+ "packet\n");
+ return -ENOMEM;
+ }
+
+ /* there is no upper limit on memory address for packetized
mechanism*/
+ newpacket->data = get_free_pages_limited(rbu_data.packetsize,
+ &ordernum, 0);
+ pr_debug("create_packet: newpacket %p\n", newpacket->data);
+
+ if(newpacket->data == NULL) {
+ printk(KERN_WARNING "create_packet: failed to allocate
new "
+ "packet\n");
+ kfree(newpacket);
+ return -ENOMEM;
+ }
+
+ newpacket->ordernum = ordernum;
```

```
+ ++rbu_data.num_packets;
+ /* initialize the newly created packet headers */
+ INIT_LIST_HEAD(&newpacket->list);
+ list_add_tail(&newpacket->list, &packet_data_head.list);
+ /* packets have fixed size*/
+ newpacket->length = rbu_data.packet_size;
+
+ pr_debug("create_packet: exit \n");
+
+ return 0;
+}
+
+static int packetize_data(void *data, size_t length)
+{
+ int rc = 0;
+
+ if (rbu_data.packet_write_count == 0) {
+ if ((rc = create_packet(length)) != 0 )
+ return rc;
+ }
+ /* fill data in to the packet */
+ if ((rc = fill_last_packet(data, length)) != 0)
+ return rc;
+
+ return rc;
+}
+
+static void packet_empty_list(void)
+{
+ struct list_head *ptemp_list;
+ struct list_head *pNext_list;
+ struct packet_data *newpacket;
+
+ ptemp_list = (&packet_data_head.list)->next;
+ while(!list_empty(ptemp_list)) {
+ newpacket = list_entry(ptemp_list, struct packet_data,
+ list);
+ pNext_list = ptemp_list->next;
+ list_del(ptemp_list);
+ ptemp_list = pNext_list;
+ }
+ /*
+ zero out the RBU packet memory before freeing to make
+ sure
+ there are no stale RBU packets left in memory
+ */
+ memset(newpacket->data, 0, rbu_data.packet_size);
+ free_pages((unsigned long)newpacket->data,
+ newpacket->ordernum);
+ kfree(newpacket);
+ }
+ rbu_data.packet_write_count = 0;
```

```
+ rbu_data.packet_read_count = 0;
+ rbu_data.num_packets = 0;
+ rbu_data.packet_size = 0;
+}
+
+/*
+ img_update_free:
+ Frees the buffer allocated for storing BIOS image
+ Always called with lock held and returned with lock held
+*/
+static void img_update_free( void)
+{
+ if (rbu_data.image_update_buffer == NULL)
+ return;
+
+ /*
+ zero out this buffer before freeing it to get rid of any stale
+ BIOS image copied in memory.
+ */
+ memset(rbu_data.image_update_buffer, 0,
+ rbu_data.image_update_buffer_size);
+ free_pages((unsigned long)rbu_data.image_update_buffer,
+ rbu_data.image_update_order_number);
+ /* Re-initialize the rbu_data variables after a free */
+ rbu_data.image_update_buffer = NULL;
+ rbu_data.image_update_buffer_size = 0;
+ rbu_data.bios_image_size = 0;
+}
+
+/*
+ img_update_realloc:
+ This function allocates the contiguous pages to accommodate the
+ requested
+ size of data. The memory address and size values are stored globally
+ and
+ on every call to this function the new size is checked to see if more
+ data is required than the existing size. If true the previous memory
+ is
+ freed and new allocation is done to accommodate the new size. If the
+ incoming size is less than the already allocated size, then that
+ memory is reused.
+ This function is called with lock held and return with lock held.
+*/
+static int img_update_realloc(unsigned long size)
+{
+ unsigned char *image_update_buffer = NULL;
+ unsigned long rc;
+ int ordernum =0;
+
+ /* check if the buffer of sufficient size has been already
+ allocated */
```

```
+ if (rbu_data.image_update_buffer_size >= size) {
+ /* check for corruption */
+ if ((size != 0) && (rbu_data.image_update_buffer ==
NULL)) {
+ printk(KERN_ERR "img_update_realloc: "
+ "corruption check failed\n");
+ return -EINVAL;
+ }
+ /* we have a valid pre-allocated buffer with sufficient
size */
+ return 0;
+ }
+
+ /* free any previously allocated buffer */
+ img_update_free();
+
+ /*
+ This has already been called as locked so we can now unlock
+ and proceed to calling get_free_pages_limited as this function
+ can sleep
+ */
+ spin_unlock(&rbu_data.lock);
+
+ image_update_buffer = (unsigned char
*)get_free_pages_limited(size,
+ &ordernum,
+ BIOS_SCAN_LIMIT);
+
+ /* acquire the spinlock again */
+ spin_lock(&rbu_data.lock);
+
+ if (image_update_buffer != NULL) {
+ rbu_data.image_update_buffer = image_update_buffer;
+ rbu_data.image_update_buffer_size = PAGE_SIZE <<
ordernum;
+ rbu_data.image_update_order_number = ordernum;
+ memset(rbu_data.image_update_buffer,0,
+ rbu_data.image_update_buffer_size);
+ rc = 0;
+ } else {
+ pr_debug("Not enough memory for image update:order"
+ " number = %d,size = %ld\n",ordernum, size);
+ rc = -ENOMEM;
+ }
+
+ return rc;
+} /* img_update_realloc */
+
+ /*
+ rbu_store_image_size:
+ This is primarily for canceling any RBU updates.
```

```
+*/
+static ssize_t rbu_store_image_size (struct rbu_download_device
+rbu_dev,
+ const char *buf, size_t count, int type)
+{
+ int size = 0;
+
+ sscanf(buf, "%d",&size);
+ if (size != 0)
+ return -EINVAL;
+
+ spin_lock(&rbu_data.lock);
+ if (type == MONOLITHIC )
+ img_update_free();
+ else
+ packet_empty_list();
+
+ spin_unlock(&rbu_data.lock);
+ return count;
+}
+
+static ssize_t rbu_show_image_size (struct rbu_download_device
+rbu_dev,
+ char *buf, int type)
+{
+ unsigned int size = 0;
+ if ( type == MONOLITHIC )
+ size = sprintf(buf, "%lu\n", rbu_data.bios_image_size);
+ else
+ size = sprintf(buf, "%lu\n", rbu_data.packet_size);
+ return size;
+}
+
+static ssize_t rbu_show_image_name (struct rbu_download_device
+rbu_dev,
+ char *buf, int type)
+{
+ unsigned int size = 0;
+ char *image;
+
+ if (type == MONOLITHIC )
+ image = rbu_data.mono_image;
+ else
+ image = rbu_data.packet_image;
+
+ size = sprintf(buf, "%s\n", image);
+ return size;
+}
+
+static ssize_t rbu_store_image_name (struct rbu_download_device
+rbu_dev,
```

```

+ const char *buf, size_t count, int
type)
+{
+ int rc = count;
+ const struct firmware *fw_entry;
+ char *image_name = NULL;
+
+ spin_lock(&rbu_data.lock);
+
+ if (strlen(buf) >= 256 ) {
+ spin_unlock(&rbu_data.lock);
+ rc = -ENOMEM;
+ }
+
+ if (type == MONOLITHIC ){
+ sscanf(buf, "%s",rbu_data.mono_image);
+ image_name = rbu_data.mono_image;
+ } else {
+ sscanf(buf, "%s",rbu_data.packet_image);
+ image_name = rbu_data.packet_image;
+ }
+
+ spin_unlock(&rbu_data.lock);
+
+ rc = request_firmware(&fw_entry, image_name,
+ &rbu_device);
+ if (rc) {
+ printk(KERN_ERR "rbu_store_image_name: "
+ "Firmware not available %d\n", rc);
+ return rc;
+ }
+
+ pr_debug("rbu_store_image_name: "
+ "request_firmware is successful "
+ "fw->size = %lu\n", fw_entry->size);
+
+ spin_lock(&rbu_data.lock);
+ if (type == MONOLITHIC ){
+ rbu_data.bios_image_size = fw_entry->size;
+ if (fw_entry->size == 0 )
+ img_update_free();
+ else {
+ rc = img_update_realloc(fw_entry->size);
+ if (rc == 0) {
+ memcpy(rbu_data.image_update_buffer,
+ fw_entry->data,
+ fw_entry->size);
+ rc = count;
+ }
+ }
+ } else {

```

```

+ /* if a new packet is entered free all previous
+ packets and start over.
+ */
+ if ( rbu_data.packet_size != fw_entry->size )
+ packet_empty_list();
+
+ rbu_data.packet_size = fw_entry->size;
+ rc = packetize_data(fw_entry->data, fw_entry->size);
+ if ( rc == 0 )
+ rc = count;
+ }
+ spin_unlock(&rbu_data.lock);
+ release_firmware(fw_entry);
+
+ return rc;
+}
+
+/* no default attributes yet. */
+static struct attribute * def_attrs[] = { NULL, };
+
+struct rbu_attribute {
+ struct attribute attr;
+ ssize_t (*show) (struct rbu_download_device *rbu_dev,
+ char *buf, int type);
+ ssize_t (*store)(struct rbu_download_device *rbu_dev,
+ const char *buf, size_t count, int type);
+ int type;
+};
+
+#define RBU_DEVICE_ATTR(_name, _mode, _show, _store, _type) \
+struct rbu_attribute rbu_attr_##_name = { \
+ .attr = { .name = __stringify(_name), .mode = _mode, .owner = \
+ THIS_MODULE }, \
+ .show = _show, \
+ .store = _store, \
+ .type = _type, \
+};
+
+#define to_rbu_attr(_attr) container_of(_attr, struct \
+ rbu_attribute, attr)
+#define to_rbu_download_device(obj) \
+ container_of(obj, struct rbu_download_device, kobj)
+
+static RBU_DEVICE_ATTR(mono_name, 0644, rbu_show_image_name,
+ rbu_store_image_name, MONOLITHIC);
+
+static RBU_DEVICE_ATTR(mono_size, 0644, rbu_show_image_size,
+ rbu_store_image_size, MONOLITHIC);
+
+static RBU_DEVICE_ATTR(packet_name, 0644, rbu_show_image_name,

```

```
+ rbu_store_image_name, PACKETIZED);
+
+static RBU_DEVICE_ATTR(packet_size, 0644, rbu_show_image_size,
+ rbu_store_image_size, PACKETIZED);
+
+static ssize_t rbu_attr_store (struct kobject *kobj,
+ struct attribute *attr,
+ const char *buf,
+ size_t count)
+{
+ struct rbu_download_device *rbu_dev =
to_rbu_download_device(kobj);
+ struct rbu_attribute *rbu_attr = to_rbu_attr(attr);
+ ssize_t rc = count;
+
+ pr_debug("rbu_attr_store: entry type = %d\n", rbu_attr->type);
+
+ if (rbu_attr->store)
+ rc = rbu_attr->store(rbu_dev, buf, count,
rbu_attr->type);
+
+ return rc;
+}
+
+static ssize_t rbu_attr_show (struct kobject * kobj,
+ struct attribute *attr,
+ char *buf)
+{
+ struct rbu_download_device *rbu_dev =
to_rbu_download_device(kobj);
+ struct rbu_attribute *rbu_attr = to_rbu_attr(attr);
+ ssize_t rc = 0;
+
+ pr_debug("rbu_attr_show: entry type = %d\n", rbu_attr->type);
+
+ if (rbu_attr->show)
+ rc = rbu_attr->show(rbu_dev, buf, rbu_attr->type);
+ return rc;
+}
+
+static struct sysfs_ops rbu_attr_ops = {
+ .show = rbu_attr_show,
+ .store = rbu_attr_store,
+};
+
+static struct kobj_type ktype_dell_rbu = {
+ .sysfs_ops = &rbu_attr_ops,
+ .default_attrs = def_attrs,
+};
+static decl_subsys(dell_rbu,&ktype_dell_rbu,NULL);
+
```

```

+static int rbu_download_device_register(struct rbu_download_device
*rbu_dev,
+ int type)
+{
+ int rc = 0;
+ if (!rbu_dev)
+ return 1;
+ memset(rbu_dev, 0, sizeof (*rbu_dev));
+ if (type == MONOLITHIC)
+ kobject_set_name(&rbu_dev->kobj, "monolithic");
+ else
+ kobject_set_name(&rbu_dev->kobj, "packetized");
+ kobj_set_kset_s(rbu_dev,dell_rbu_subsys);
+ rc = kobject_register(&rbu_dev->kobj);
+ if (!rc) {
+ if (type == MONOLITHIC ) {
+ sysfs_create_file(&rbu_dev->kobj,
+ &rbu_attr_mono_name.attr);
+ sysfs_create_file(&rbu_dev->kobj,
+ &rbu_attr_mono_size.attr);
+ } else {
+ sysfs_create_file(&rbu_dev->kobj,
+ &rbu_attr_packet_name.attr);
+ sysfs_create_file(&rbu_dev->kobj,
+ &rbu_attr_packet_size.attr);
+ }
+ } else
+ pr_debug("rbu_download_device_register: "
+ "kobject_register %d \n", rc);
+ pr_debug("rbu_download_device_register: rbu_dev addr %p\n",
rbu_dev);
+ return rc;
+}
+
+static struct rbu_download_device *create_rbu_download_entry(int type)
+{
+ struct rbu_download_device *rbu_dev = NULL;
+ rbu_dev = kmalloc( sizeof(struct rbu_download_device),
+ GFP_KERNEL);
+ if (!rbu_dev) {
+ printk(KERN_ERR "create_rbu_download_entry: kmalloc
failed\n");
+ return NULL;
+ }
+ rbu_dev->type = type;
+ if (rbu_download_device_register(rbu_dev, type)) {
+ pr_debug("create_rbu_download_entry: "
+ "rbu_download_device_register failed \n");
+ kfree(rbu_dev);
+ }
+}

```

```
+ pr_debug("create_rbu_download_entry: rbu_dev %p\n", rbu_dev);
+ return rbu_dev;
+ }
+
+static void remove_rbu_download_entry(struct rbu_download_device
*rbu_dev,
+ int type)
+{
+ pr_debug("remove_rbu_download_entry: rbu_dev ptr %p \n",
rbu_dev);
+ if (rbu_dev != NULL) {
+ kobject_unregister(&rbu_dev->kobj);
+ kfree(rbu_dev);
+ }
+
+
+static int __init dcdrbu_init(void)
+{
+ int rc = 0;
+ spin_lock_init(&rbu_data.lock);
+
+ init_packet_head();
+
+ device_initialize(&rbu_device);
+
+ rc = firmware_register(&dell_rbu_subsys);
+ if (rc < 0) {
+ printk(KERN_WARNING "dcdrbu_init: firmware_register"
+ " dell_rbu failed\n");
+ return rc;
+ }
+
+
+ rbu_download_mono = create_rbu_download_entry (MONOLITHIC);
+ if (rbu_download_mono == NULL) {
+ firmware_unregister(&dell_rbu_subsys);
+ return -ENOMEM;
+ }
+
+
+ rbu_download_packet= create_rbu_download_entry (PACKETIZED);
+ if (rbu_download_packet == NULL) {
+ remove_rbu_download_entry(rbu_download_mono,
MONOLITHIC);
+ firmware_unregister(&dell_rbu_subsys);
+ return -ENOMEM;
+ }
+ strncpy(rbu_device.bus_id,"firmware", BUS_ID_SIZE);
+ return rc;
+ }
+
+static __exit void dcdrbu_exit( void)
```

Linux-Kernel: RE: [patch 2.6.12-rc3] dell_rbu: Resubmitting patch for new DellBIOS update driver

```
+{
+ spin_lock(&rbu_data.lock);
+ packet_empty_list();
+ img_update_free();
+ spin_unlock(&rbu_data.lock);
+ remove_rbu_download_entry(rbu_download_packet, PACKETIZED);
+ remove_rbu_download_entry(rbu_download_mono, MONOLITHIC);
+ firmware_unregister(&dell_rbu_subsys);
+}
+
+module_exit(dcdrbu_exit);
+module_init(dcdrbu_init);
+
diff -uprN linux-2.6.11.8.ORIG/drivers/firmware/Kconfig
linux-2.6.11.8/drivers/firmware/Kconfig
--- linux-2.6.11.8.ORIG/drivers/firmware/Kconfig 2005-05-13
12:07:58.000000000 -0500
+++ linux-2.6.11.8/drivers/firmware/Kconfig 2005-06-02
17:01:29.072484168 -0500
@@ -58,4 +58,17 @@ config EFI_PCDP
```

See

<http://www.dig64.org/specifications/DIG64_HCDPv20_042804.pdf>

```
+config DELL_RBU
+ tristate "BIOS update support for DELL systems via sysfs"
+ default n
+ select FW_LOADER
+ help
+ Say Y if you want to have the option of updating the BIOS for
your
+ DELL system. Note you need a supporting application to
communicate
+ with the BIOS regarding the new image for the image update to
+ take effect.
+
+ See <file:Documentation/DELL_RBU.txt> for more details on the
driver.
+
+
endmenu
diff -uprN linux-2.6.11.8.ORIG/drivers/firmware/Makefile
linux-2.6.11.8/drivers/firmware/Makefile
--- linux-2.6.11.8.ORIG/drivers/firmware/Makefile 2005-05-13
12:08:12.000000000 -0500
+++ linux-2.6.11.8/drivers/firmware/Makefile 2005-05-09
15:15:16.000000000 -0500
@@ -4,3 +4,4 @@
obj-$(CONFIG_EDD) += edd.o
obj-$(CONFIG_EFI_VARS) += efivars.o
obj-$(CONFIG_EFI_PCDP) += pcdp.o
```

RE: [patch 2.6.12-rc3] dell_rbu: Resubmitting patch for new DellBIOS update driver

Linux-Kernel: RE: [patch 2.6.12-rc3] dell_rbu: Resubmitting patch for new DellBIOS update driver

```
+obj-$(CONFIG_DELL_RBU) += dell_rbu.o
```

-

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to majordomo@vger.kernel.org

More majordomo info at <http://vger.kernel.org/majordomo-info.html>

Please read the FAQ at <http://www.tux.org/lkml/>