```
**********************************************************
   52155 Sun May  4 19:18:02 2014
new/usr/src/cmd/hal/hald/solaris/devinfo_storage.c
4846 HAL partition names don't match real partition names
**********************************************************
/***************************************************************************
 *
 * devinfo_storage.c : storage devices
 *
 * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2013 Garrett D'Amore <garrett@damore.org>
 * Copyright 2014 Andrew Stormont.
#endif /* ! codereview */
 *
 * Licensed under the Academic Free License version 2.1
 *
 ***************************************************************************/

#ifdef HAVE_CONFIG_H
#  include <config.h>
#endif

#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <ctype.h>
#include <libdevinfo.h>
#include <sys/types.h>
#include <sys/mkdev.h>
#include <sys/stat.h>
#include <sys/mntent.h>
#include <sys/mnttab.h>

#include "../osspec.h"
#include "../logger.h"
#include "../hald.h"
#include "../hald_dbus.h"
#include "../device_info.h"
#include "../util.h"
#include "../hald_runner.h"
#include "hotplug.h"
#include "devinfo.h"
#include "devinfo_misc.h"
#include "devinfo_storage.h"
#include "osspec_solaris.h"

#ifdef sparc
#define WHOLE_DISK      "s2"
#else
#define WHOLE_DISK      "p0"
#endif

/* some devices,especially CDROMs, may take a while to be probed (values in ms)
#define DEVINFO_PROBE_STORAGE_TIMEOUT   60000
#define DEVINFO_PROBE_VOLUME_TIMEOUT    60000

typedef struct devinfo_storage_minor {
        char    *devpath;
        char    *devlink;
        char    *slice;
        dev_t   dev;
        int     dosnum; /* dos disk number or -1 */
} devinfo_storage_minor_t;

HalDevice *devinfo_ide_add(HalDevice *parent, di_node_t node, char *devfs_path,
static HalDevice *devinfo_ide_host_add(HalDevice *parent, di_node_t node, char *
```



```
static HalDevice *devinfo_ide_device_add(HalDevice *parent, di_node_t node, char
static HalDevice *devinfo_ide_storage_add(HalDevice *parent, di_node_t node, cha
HalDevice *devinfo_scsi_add(HalDevice *parent, di_node_t node, char *devfs_path,
static HalDevice *devinfo_scsi_storage_add(HalDevice *parent, di_node_t node, ch
HalDevice *devinfo_blkdev_add(HalDevice *parent, di_node_t node, char *devfs_pat
static HalDevice *devinfo_blkdev_storage_add(HalDevice *parent, di_node_t node,
HalDevice *devinfo_floppy_add(HalDevice *parent, di_node_t node, char *devfs_pat
static void devinfo_floppy_add_volume(HalDevice *parent, di_node_t node);
static HalDevice *devinfo_lofi_add(HalDevice *parent, di_node_t node, char *devf
static void devinfo_lofi_add_minor(HalDevice *parent, di_node_t node, char *mino
static void devinfo_storage_minors(HalDevice *parent, di_node_t node, gchar *dev
static struct devinfo_storage_minor *devinfo_storage_new_minor(char *maindev_pat
    char *devlink, dev_t dev, int dosnum);
static void devinfo_storage_free_minor(struct devinfo_storage_minor *m);
HalDevice *devinfo_volume_add(HalDevice *parent, di_node_t node, devinfo_storage
static void devinfo_volume_preprobing_done(HalDevice *d, gpointer userdata1, gpo
static void devinfo_volume_hotplug_begin_add (HalDevice *d, HalDevice *parent, D
static void devinfo_storage_hotplug_begin_add (HalDevice *d, HalDevice *parent,
static void devinfo_storage_probing_done (HalDevice *d, guint32 exit_type, gint
const gchar *devinfo_volume_get_prober (HalDevice *d, int *timeout);
const gchar *devinfo_storage_get_prober (HalDevice *d, int *timeout);

static char *devinfo_scsi_dtype2str(int dtype);
static char *devinfo_volume_get_slice_name (char *devlink);
static gboolean dos_to_dev(char *path, char **devpath, int *partnum);
static gboolean is_dos_path(char *path, int *partnum);

static void devinfo_storage_set_nicknames (HalDevice *d);

DevinfoDevHandler devinfo_ide_handler = {
        devinfo_ide_add,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL
};
DevinfoDevHandler devinfo_scsi_handler = {
        devinfo_scsi_add,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL
};
DevinfoDevHandler devinfo_blkdev_handler = {
        devinfo_blkdev_add,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL
};
DevinfoDevHandler devinfo_floppy_handler = {
        devinfo_floppy_add,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL
};
DevinfoDevHandler devinfo_lofi_handler = {
        devinfo_lofi_add,
        NULL,
        NULL,
        NULL,
```

```
128         NULL,
129         NULL
130 };
131 DevinfoDevHandler devinfo_storage_handler = {
132         NULL,
133         NULL,
134         devinfo_storage_hotplug_begin_add,
135         NULL,
136         devinfo_storage_probing_done,
137         devinfo_storage_get_prober
138 };
139 DevinfoDevHandler devinfo_volume_handler = {
140         NULL,
141         NULL,
142         devinfo_volume_hotplug_begin_add,
143         NULL,
144         NULL,
145         devinfo_volume_get_prober
146 };

148 /* IDE */

150 HalDevice *
151 devinfo_ide_add(HalDevice *parent, di_node_t node, char *devfs_path, char *devic
152 {
153         char    *s;

155         if ((device_type != NULL) && (strcmp(device_type, "ide") == 0)) {
156                 return (devinfo_ide_host_add(parent, node, devfs_path));
157         }

159         if ((di_prop_lookup_strings (DDI_DEV_T_ANY, node, "class", &s) > 0) &&
160             (strcmp (s, "dada") == 0)) {
161                 return (devinfo_ide_device_add(parent, node, devfs_path));
162         }

164         return (NULL);
165 }

167 static HalDevice *
168 devinfo_ide_host_add(HalDevice *parent, di_node_t node, char *devfs_path)
169 {
170         HalDevice *d;

172         d = hal_device_new ();

174         devinfo_set_default_properties (d, parent, node, devfs_path);
175         hal_device_property_set_string (d, "info.product", "IDE host controller"
176         hal_device_property_set_string (d, "info.subsystem", "ide_host");
177         hal_device_property_set_int (d, "ide_host.number", 0); /* XXX */

179         devinfo_add_enqueue (d, devfs_path, &devinfo_ide_handler);

181         return (d);
182 }

184 static HalDevice *
185 devinfo_ide_device_add(HalDevice *parent, di_node_t node, char *devfs_path)
186 {
187         HalDevice *d;

189         d = hal_device_new();

191         devinfo_set_default_properties (d, parent, node, devfs_path);
192         hal_device_property_set_string (parent, "info.product", "IDE device");
193         hal_device_property_set_string (parent, "info.subsystem", "ide");
```



```
194         hal_device_property_set_int (parent, "ide.host", 0); /* XXX */
195         hal_device_property_set_int (parent, "ide.channel", 0);

197         devinfo_add_enqueue (d, devfs_path, &devinfo_ide_handler);

199         return (devinfo_ide_storage_add (d, node, devfs_path));
200 }

202 static HalDevice *
203 devinfo_ide_storage_add(HalDevice *parent, di_node_t node, char *devfs_path)
204 {
205         HalDevice *d;
206         char    *s;
207         int     *i;
208         char    *driver_name;
209         char    udi[HAL_PATH_MAX];

211         if ((driver_name = di_driver_name (node)) == NULL) {
212                 return (NULL);
213         }

215         d = hal_device_new ();

217         devinfo_set_default_properties (d, parent, node, devfs_path);
218         hal_device_property_set_string (d, "info.category", "storage");

220         hal_util_compute_udi (hald_get_gdl (), udi, sizeof (udi),
221                 "%s/%s%d", hal_device_get_udi (parent), driver_name, di_instance
222         hal_device_set_udi (d, udi);
223         hal_device_property_set_string (d, "info.udi", udi);
224         PROP_STR(d, node, s, "devid", "info.product");

226         hal_device_add_capability (d, "storage");
227         hal_device_property_set_string (d, "storage.bus", "ide");
228         hal_device_property_set_int (d, "storage.lun", 0);
229         hal_device_property_set_string (d, "storage.drive_type", "disk");

231         PROP_BOOL(d, node, i, "hotpluggable", "storage.hotpluggable");
232         PROP_BOOL(d, node, i, "removable-media", "storage.removable");

234         hal_device_property_set_bool (d, "storage.media_check_enabled", FALSE);

236         /* XXX */
237         hal_device_property_set_bool (d, "storage.requires_eject", FALSE);

239         hal_device_add_capability (d, "block");

241         devinfo_storage_minors (d, node, (char *)devfs_path, FALSE);

243         return (d);
244 }

246 /* SCSI */

248 HalDevice *
249 devinfo_scsi_add(HalDevice *parent, di_node_t node, char *devfs_path, char *devi
250 {
251         int     *i;
252         char    *driver_name;
253         HalDevice *d;
254         char    udi[HAL_PATH_MAX];

256         driver_name = di_driver_name (node);
257         if ((driver_name == NULL) || (strcmp (driver_name, "sd") != 0)) {
258                 return (NULL);
259         }
```

```
261         d = hal_device_new ();

263         devinfo_set_default_properties (d, parent, node, devfs_path);
264         hal_device_property_set_string (d, "info.subsystem", "scsi");

266         hal_util_compute_udi (hald_get_gdl (), udi, sizeof (udi),
267                 "%s/%s%d", hal_device_get_udi (parent), di_node_name(node), di_i
268         hal_device_set_udi (d, udi);
269         hal_device_property_set_string (d, "info.udi", udi);

271         hal_device_property_set_int (d, "scsi.host",
272                 hal_device_property_get_int (parent, "scsi_host.host"));
273         hal_device_property_set_int (d, "scsi.bus", 0);
274         PROP_INT(d, node, i, "target", "scsi.target");
275         PROP_INT(d, node, i, "lun", "scsi.lun");
276         hal_device_property_set_string (d, "info.product", "SCSI Device");

278         devinfo_add_enqueue (d, devfs_path, &devinfo_scsi_handler);

280         return (devinfo_scsi_storage_add (d, node, devfs_path));
281 }

283 static HalDevice *
284 devinfo_scsi_storage_add(HalDevice *parent, di_node_t node, char *devfs_path)
285 {
286         HalDevice *d;
287         int     *i;
288         char    *s;
289         char    udi[HAL_PATH_MAX];

291         d = hal_device_new ();

293         devinfo_set_default_properties (d, parent, node, devfs_path);
294         hal_device_property_set_string (d, "info.category", "storage");

296         hal_util_compute_udi (hald_get_gdl (), udi, sizeof (udi),
297                 "%s/sd%d", hal_device_get_udi (parent), di_instance (node));
298         hal_device_set_udi (d, udi);
299         hal_device_property_set_string (d, "info.udi", udi);
300         PROP_STR(d, node, s, "inquiry-product-id", "info.product");

302         hal_device_add_capability (d, "storage");

304         hal_device_property_set_int (d, "storage.lun",
305                 hal_device_property_get_int (parent, "scsi.lun"));
306         PROP_BOOL(d, node, i, "hotpluggable", "storage.hotpluggable");
307         PROP_BOOL(d, node, i, "removable-media", "storage.removable");
308         hal_device_property_set_bool (d, "storage.requires_eject", FALSE);

310         /*
311          * We have to enable polling not only for drives with removable media,
312          * but also for hotpluggable devices, because when a disk is
313          * unplugged while busy/mounted, there is not sysevent generated.
314          * Instead, the HBA driver (scsa2usb, scsa1394) will notify sd driver
315          * and the latter will report DKIO_DEV_GONE via DKIOCSTATE ioctl.
316          * So we have to enable media check so that hald-addon-storage notices
317          * the "device gone" condition and unmounts all associated volumes.
318          */
319         hal_device_property_set_bool (d, "storage.media_check_enabled",
320             ((di_prop_lookup_ints(DDI_DEV_T_ANY, node, "removable-media", &i) >=
321             (di_prop_lookup_ints(DDI_DEV_T_ANY, node, "hotpluggable", &i) >= 0))

323         if (di_prop_lookup_ints(DDI_DEV_T_ANY, node, "inquiry-device-type",
324             &i) > 0) {
325                 s = devinfo_scsi_dtype2str (*i);
```



```
326                 hal_device_property_set_string (d, "storage.drive_type", s);

328                 if (strcmp (s, "cdrom") == 0) {
329                         hal_device_add_capability (d, "storage.cdrom");
330                         hal_device_property_set_bool (d, "storage.no_partitions_
331                         hal_device_property_set_bool (d, "storage.requires_eject
332                 }
333         }

335         hal_device_add_capability (d, "block");

337         devinfo_storage_minors (d, node, devfs_path, FALSE);

339         return (d);
340 }

342 static char *
343 devinfo_scsi_dtype2str(int dtype)
344 {
345         char *dtype2str[] = {
346                 "disk"  ,         /* DTYPE_DIRECT         0x00 */
347                 "tape"  ,         /* DTYPE_SEQUENTIAL     0x01 */
348                 "printer",         /* DTYPE_PRINTER        0x02 */
349                 "processor",         /* DTYPE_PROCESSOR      0x03 */
350                 "worm"  ,         /* DTYPE_WORM           0x04 */
351                 "cdrom" ,         /* DTYPE_RODIRECT       0x05 */
352                 "scanner",         /* DTYPE_SCANNER        0x06 */
353                 "cdrom" ,         /* DTYPE_OPTICAL        0x07 */
354                 "changer",         /* DTYPE_CHANGER        0x08 */
355                 "comm"  ,         /* DTYPE_COMM           0x09 */
356                 "scsi"  ,         /* DTYPE_???            0x0A */
357                 "scsi"  ,         /* DTYPE_???            0x0B */
358                 "array_ctrl",         /* DTYPE_ARRAY_CTRL     0x0C */
359                 "esi"   ,         /* DTYPE_ESI            0x0D */
360                 "disk"            /* DTYPE_RBC            0x0E */
361         };

363         if (dtype < NELEM(dtype2str)) {
364                 return (dtype2str[dtype]);
365         } else {
366                 return ("scsi");
367         }

369 }

371 /* blkdev */

373 HalDevice *
374 devinfo_blkdev_add(HalDevice *parent, di_node_t node, char *devfs_path, char *de
375 {
376         int     *i;
377         char    *driver_name;
378         HalDevice *d;
379         char    udi[HAL_PATH_MAX];

381         driver_name = di_driver_name (node);
382         if ((driver_name == NULL) || (strcmp (driver_name, "blkdev") != 0)) {
383                 return (NULL);
384         }

386         d = hal_device_new ();

388         devinfo_set_default_properties (d, parent, node, devfs_path);
389         hal_device_property_set_string (d, "info.subsystem", "pseudo");

391         hal_util_compute_udi (hald_get_gdl (), udi, sizeof (udi),
```

```
392                 "%s/%s%d", hal_device_get_udi (parent), di_node_name(node), di_i
393         hal_device_set_udi (d, udi);
394         hal_device_property_set_string (d, "info.udi", udi);
395         hal_device_property_set_string (d, "info.product", "Block Device");

397         devinfo_add_enqueue (d, devfs_path, &devinfo_blkdev_handler);

399         return (devinfo_blkdev_storage_add (d, node, devfs_path));
400 }

402 static HalDevice *
403 devinfo_blkdev_storage_add(HalDevice *parent, di_node_t node, char *devfs_path)
404 {
405         HalDevice *d;
406         char    *driver_name;
407         int     *i;
408         char    *s;
409         char    udi[HAL_PATH_MAX];

411         d = hal_device_new ();

413         devinfo_set_default_properties (d, parent, node, devfs_path);
414         hal_device_property_set_string (d, "info.category", "storage");

416         hal_util_compute_udi (hald_get_gdl (), udi, sizeof (udi),
417                 "%s/blkdev%d", hal_device_get_udi (parent), di_instance (node));
418         hal_device_set_udi (d, udi);
419         hal_device_property_set_string (d, "info.udi", udi);

421         hal_device_add_capability (d, "storage");

423         hal_device_property_set_int (d, "storage.lun", 0);

425         PROP_BOOL(d, node, i, "hotpluggable", "storage.hotpluggable");
426         PROP_BOOL(d, node, i, "removable-media", "storage.removable");

428         hal_device_property_set_bool (d, "storage.requires_eject", FALSE);
429         hal_device_property_set_bool (d, "storage.media_check_enabled", TRUE);
430         hal_device_property_set_string (d, "storage.drive_type", "disk");

432         hal_device_add_capability (d, "block");

434         devinfo_storage_minors (d, node, devfs_path, FALSE);

436         return (d);
437 }

439 /* floppy */

441 HalDevice *
442 devinfo_floppy_add(HalDevice *parent, di_node_t node, char *devfs_path, char *de
443 {
444         char    *driver_name;
445         char    *raw;
446         char    udi[HAL_PATH_MAX];
447         di_devlink_handle_t devlink_hdl;
448         int     major;
449         di_minor_t minor;
450         dev_t   dev;
451         HalDevice *d = NULL;
452         char    *minor_path = NULL;
453         char    *devlink = NULL;

455         driver_name = di_driver_name (node);
456         if ((driver_name == NULL) || (strcmp (driver_name, "fd") != 0)) {
457                 return (NULL);
```



```
458         }

460         /*
461          * The only minor node we're interested in is /dev/diskette*
462          */
463         major = di_driver_major(node);
464         if ((devlink_hdl = di_devlink_init(NULL, 0)) == NULL) {
465                 return (NULL);
466         }
467         minor = DI_MINOR_NIL;
468         while ((minor = di_minor_next(node, minor)) != DI_MINOR_NIL) {
469                 dev = di_minor_devt(minor);
470                 if ((major != major(dev)) ||
471                     (di_minor_type(minor) != DDM_MINOR) ||
472                     (di_minor_spectype(minor) != S_IFBLK) ||
473                     ((minor_path = di_devfs_minor_path(minor)) == NULL)) {
474                         continue;
475                 }
476                 if ((devlink = get_devlink(devlink_hdl, "diskette.+" , minor_pat
477                         break;
478                 }
479                 di_devfs_path_free (minor_path);
480                 minor_path = NULL;
481                 free(devlink);
482                 devlink = NULL;
483         }
484         di_devlink_fini (&devlink_hdl);

486         if ((devlink == NULL) || (minor_path == NULL)) {
487                 HAL_INFO (("floppy devlink not found %s", devfs_path));
488                 goto out;
489         }

491         d = hal_device_new ();

493         devinfo_set_default_properties (d, parent, node, devfs_path);
494         hal_device_property_set_string (d, "info.category", "storage");
495         hal_device_add_capability (d, "storage");
496         hal_device_property_set_string (d, "storage.bus", "platform");
497         hal_device_property_set_bool (d, "storage.hotpluggable", FALSE);
498         hal_device_property_set_bool (d, "storage.removable", TRUE);
499         hal_device_property_set_bool (d, "storage.requires_eject", TRUE);
500         hal_device_property_set_bool (d, "storage.media_check_enabled", FALSE);
501         hal_device_property_set_string (d, "storage.drive_type", "floppy");

503         hal_device_add_capability (d, "block");
504         hal_device_property_set_bool (d, "block.is_volume", FALSE);
505         hal_device_property_set_int (d, "block.major", major(dev));
506         hal_device_property_set_int (d, "block.minor", minor(dev));
507         hal_device_property_set_string (d, "block.device", devlink);
508         raw = dsk_to_rdsk (devlink);
509         hal_device_property_set_string (d, "block.solaris.raw_device", raw);
510         free (raw);

512         devinfo_add_enqueue (d, devfs_path, &devinfo_storage_handler);

514         /* trigger initial probe-volume */
515         devinfo_floppy_add_volume(d, node);

517 out:
518         di_devfs_path_free (minor_path);
519         free(devlink);

521         return (d);
522 }
```

```
524 static void
525 devinfo_floppy_add_volume(HalDevice *parent, di_node_t node)
526 {
527         char    *devlink;
528         char    *devfs_path;
529         int     minor, major;
530         dev_t   dev;
531         struct devinfo_storage_minor *m;

533         devfs_path = (char *)hal_device_property_get_string (parent, "solaris.de
534         devlink = (char *)hal_device_property_get_string (parent, "block.device"
535         major = hal_device_property_get_int (parent, "block.major");
536         minor = hal_device_property_get_int (parent, "block.minor");
537         dev = makedev (major, minor);

539         m = devinfo_storage_new_minor (devfs_path, WHOLE_DISK, devlink, dev, -1)
540         devinfo_volume_add (parent, node, m);
541         devinfo_storage_free_minor (m);
542 }

544 /*
545  * After reprobing storage, reprobe its volumes.
546  */
547 static void
548 devinfo_floppy_rescan_probing_done (HalDevice *d, guint32 exit_type, gint return
549     char **error, gpointer userdata1, gpointer userdata2)
550 {
551         void *end_token = (void *) userdata1;
552         const char *devfs_path;
553         di_node_t node;
554         HalDevice *v;

556         if (!hal_device_property_get_bool (d, "storage.removable.media_available
557                 HAL_INFO (("no floppy media", hal_device_get_udi (d)));

559                 /* remove child (can only be single volume) */
560                 if (((v = hal_device_store_match_key_value_string (hald_get_gdl(
561                     "info.parent", hal_device_get_udi (d))) != NULL) &&
562                     ((devfs_path = hal_device_property_get_string (v,
563                     "solaris.devfs_path")) != NULL)) {
564                         devinfo_remove_enqueue ((char *)devfs_path, NULL);
565                 }
566         } else {
567                 HAL_INFO (("floppy media found", hal_device_get_udi (d)));

569                 if ((devfs_path = hal_device_property_get_string(d, "solaris.dev
570                         HAL_INFO (("no devfs_path", hal_device_get_udi (d)));
571                         hotplug_event_process_queue ();
572                         return;
573                 }
574                 if ((node = di_init (devfs_path, DINFOCPYALL)) == DI_NODE_NIL) {
575                         HAL_INFO (("di_init %s failed %d", devfs_path, errno));
576                         hotplug_event_process_queue ();
577                         return;
578                 }

580                 devinfo_floppy_add_volume (d, node);

582                 di_fini (node);
583         }

585         hotplug_event_process_queue ();
586 }
587
588 /* lofi */
```



```
590 HalDevice *
591 devinfo_lofi_add(HalDevice *parent, di_node_t node, char *devfs_path, char *devi
592 {
593         return (devinfo_lofi_add_major(parent,node, devfs_path, device_type, FAL
594 }

596 HalDevice *
597 devinfo_lofi_add_major(HalDevice *parent, di_node_t node, char *devfs_path, char
598     gboolean rescan, HalDevice *lofi_d)
599 {
600         char    *driver_name;
601         HalDevice *d = NULL;
602         char    udi[HAL_PATH_MAX];
603         di_devlink_handle_t devlink_hdl;
604         int     major;
605         di_minor_t minor;
606         dev_t   dev;
607         char    *minor_path = NULL;
608         char    *devlink = NULL;

610         driver_name = di_driver_name (node);
611         if ((driver_name == NULL) || (strcmp (driver_name, "lofi") != 0)) {
612                 return (NULL);
613         }

615         if (!rescan) {
616                 d = hal_device_new ();

618                 devinfo_set_default_properties (d, parent, node, devfs_path);
619                 hal_device_property_set_string (d, "info.subsystem", "pseudo");

621                 hal_util_compute_udi (hald_get_gdl (), udi, sizeof (udi),
622                         "%s/%s%d", hal_device_get_udi (parent), di_node_name(nod
623                 hal_device_set_udi (d, udi);
624                 hal_device_property_set_string (d, "info.udi", udi);

626                 devinfo_add_enqueue (d, devfs_path, &devinfo_lofi_handler);
627         } else {
628                 d = lofi_d;
629         }

631         /*
632          * Unlike normal storage, as in devinfo_storage_minors(), where
633          * sd instance -> HAL storage, sd minor node -> HAL volume,
634          * lofi always has one instance, lofi minor -> HAL storage.
635          * lofi storage never has slices, but it can have
636          * embedded pcfs partitions that fstyp would recognize
637          */
638         major = di_driver_major(node);
639         if ((devlink_hdl = di_devlink_init(NULL, 0)) == NULL) {
640                 return (d);
641         }
642         minor = DI_MINOR_NIL;
643         while ((minor = di_minor_next(node, minor)) != DI_MINOR_NIL) {
644                 dev = di_minor_devt(minor);
645                 if ((major != major(dev)) ||
646                     (di_minor_type(minor) != DDM_MINOR) ||
647                     (di_minor_spectype(minor) != S_IFBLK) ||
648                     ((minor_path = di_devfs_minor_path(minor)) == NULL)) {
649                         continue;
650                 }
651                 if ((devlink = get_devlink(devlink_hdl, NULL, minor_path)) == NU
652                         di_devfs_path_free (minor_path);
653                         continue;
654                 }
```

```
656                 if (!rescan ||
657                     (hal_device_store_match_key_value_string (hald_get_gdl (),
658                     "solaris.devfs_path", minor_path) == NULL)) {
659                         devinfo_lofi_add_minor(d, node, minor_path, devlink, dev
660                 }
661
662                 di_devfs_path_free (minor_path);
663                 free(devlink);
664         }
665         di_devlink_fini (&devlink_hdl);
666
667         return (d);
668 }
669
670 static void
671 devinfo_lofi_add_minor(HalDevice *parent, di_node_t node, char *minor_path, char
672 {
673         HalDevice *d;
674         char    *raw;
675         char    *doslink;
676         char    dospath[64];
677         struct devinfo_storage_minor *m;
678         int     i;
679
680         /* add storage */
681         d = hal_device_new ();
682
683         devinfo_set_default_properties (d, parent, node, minor_path);
684         hal_device_property_set_string (d, "info.category", "storage");
685         hal_device_add_capability (d, "storage");
686         hal_device_property_set_string (d, "storage.bus", "lofi");
687         hal_device_property_set_bool (d, "storage.hotpluggable", TRUE);
688         hal_device_property_set_bool (d, "storage.removable", FALSE);
689         hal_device_property_set_bool (d, "storage.requires_eject", FALSE);
690         hal_device_property_set_string (d, "storage.drive_type", "disk");
691         hal_device_add_capability (d, "block");
692         hal_device_property_set_int (d, "block.major", major(dev));
693         hal_device_property_set_int (d, "block.minor", minor(dev));
694         hal_device_property_set_string (d, "block.device", devlink);
695         raw = dsk_to_rdsk (devlink);
696         hal_device_property_set_string (d, "block.solaris.raw_device", raw);
697         free (raw);
698         hal_device_property_set_bool (d, "block.is_volume", FALSE);
699
700         devinfo_add_enqueue (d, minor_path, &devinfo_storage_handler);
701
702         /* add volumes: one on main device and a few pcfs candidates */
703         m = devinfo_storage_new_minor(minor_path, WHOLE_DISK, devlink, dev, -1);
704         devinfo_volume_add (d, node, m);
705         devinfo_storage_free_minor (m);
706
707         doslink = (char *)calloc (1, strlen (devlink) + sizeof ("pNN") + 1);
  7      doslink = (char *)calloc (1, strlen (devlink) + sizeof (":NNN") + 1);
708         if (doslink != NULL) {
709                 for (i = 1; i < 16; i++) {
710                         snprintf(dospath, sizeof (dospath), "p%d", i);
711                         sprintf(doslink, "%sp%d", devlink, i);
 10                     snprintf(dospath, sizeof (dospath), WHOLE_DISK":%d", i);
 11                     sprintf(doslink, "%s:%d", devlink, i);
712                         m = devinfo_storage_new_minor(minor_path, dospath, dosli
713                         devinfo_volume_add (d, node, m);
714                         devinfo_storage_free_minor (m);
715                 }
716                 free (doslink);
717         }
718 }
_____unchanged_portion_omitted_
```



```
797 /*
798  * Storage minor nodes are potential "volume" objects.
799  * This function also completes building the parent object (main storage device)
800  */
801 static void
802 devinfo_storage_minors(HalDevice *parent, di_node_t node, gchar *devfs_path, gbo
803 {
804         di_devlink_handle_t devlink_hdl;
805         gboolean is_cdrom;
806         const char *whole_disk;
807         int     major;
808         di_minor_t minor;
809         dev_t   dev;
810         char    *minor_path = NULL;
811         char    *maindev_path = NULL;
812         char    *devpath, *devlink;
813         int     doslink_len;
814         char    *doslink;
815         char    dospath[64];
816         char    *slice;
817         int     pathlen;
818         int     i;
819         char    *raw;
820         boolean_t maindev_is_d0;
821         GQueue  *mq;
822         HalDevice *volume;
823         struct devinfo_storage_minor *m;
824         struct devinfo_storage_minor *maindev = NULL;
825
826         /* for cdroms whole disk is always s2 */
827         is_cdrom = hal_device_has_capability (parent, "storage.cdrom");
828         whole_disk = is_cdrom ? "s2" : WHOLE_DISK;
829
830         major = di_driver_major(node);
831
832         /* the "whole disk" p0/s2/d0 node must come first in the hotplug queue
833          * so we put other minor nodes on the local queue and move to the
834          * hotplug queue up in the end
835          */
836         if ((mq = g_queue_new()) == NULL) {
837                 goto err;
838         }
839         if ((devlink_hdl = di_devlink_init(NULL, 0)) == NULL) {
840                 g_queue_free (mq);
841                 goto err;
842         }
843         minor = DI_MINOR_NIL;
844         while ((minor = di_minor_next(node, minor)) != DI_MINOR_NIL) {
845                 dev = di_minor_devt(minor);
846                 if ((major != major(dev)) ||
847                     (di_minor_type(minor) != DDM_MINOR) ||
848                     (di_minor_spectype(minor) != S_IFBLK) ||
849                     ((minor_path = di_devfs_minor_path(minor)) == NULL)) {
850                         continue;
851                 }
852                 if ((devlink = get_devlink(devlink_hdl, NULL, minor_path)) == NU
853                         di_devfs_path_free (minor_path);
854                         continue;
855                 }
856
857                 slice = devinfo_volume_get_slice_name (devlink);
858                 if (strlen (slice) < 2) {
859                         free (devlink);
860                         di_devfs_path_free (minor_path);
861                         continue;
```

```
862                 }

164                 /* ignore p1..N - we'll use p0:N instead */
165                 if ((strlen (slice) > 1) && (slice[0] == 'p') && isdigit(slice[1
166                     ((atol(&slice[1])) > 0)) {
167                         free (devlink);
168                         di_devfs_path_free (minor_path);
169                         continue;
170                 }

864                 m = devinfo_storage_new_minor(minor_path, slice, devlink, dev, -
865                 if (m == NULL) {
866                         free (devlink);
867                         di_devfs_path_free (minor_path);
868                         continue;
869                 }

871                 /* main device is either s2/p0 or d0, the latter taking preceden
872                 if ((strcmp (slice, "d0") == 0) ||
873                     (((strcmp (slice, whole_disk) == 0) && (maindev == NULL))))
874                         if (maindev_path != NULL) {
875                                 di_devfs_path_free (maindev_path);
876                         }
877                         maindev_path = minor_path;
878                         maindev = m;
879                         g_queue_push_head (mq, maindev);
880                 } else {
881                         di_devfs_path_free (minor_path);
882                         g_queue_push_tail (mq, m);
883                 }

885                 free (devlink);
886         }
887         di_devlink_fini (&devlink_hdl);

889         if (maindev == NULL) {
890                 /* shouldn't typically happen */
891                 while (!g_queue_is_empty (mq)) {
892                         devinfo_storage_free_minor (g_queue_pop_head (mq));
893                 }
894                 goto err;
895         }

897         /* first enqueue main storage device */
898         if (!rescan) {
899                 hal_device_property_set_int (parent, "block.major", major);
900                 hal_device_property_set_int (parent, "block.minor", minor(mainde
901                 hal_device_property_set_string (parent, "block.device", maindev-
902                 raw = dsk_to_rdsk (maindev->devlink);
903                 hal_device_property_set_string (parent, "block.solaris.raw_devic
904                 free (raw);
905                 hal_device_property_set_bool (parent, "block.is_volume", FALSE);
906                 hal_device_property_set_string (parent, "solaris.devfs_path", ma
907                 devinfo_add_enqueue (parent, maindev_path, &devinfo_storage_hand
908         }

910         /* add virtual dos volumes to enable pcfs probing */
911         if (!is_cdrom) {
912                 doslink_len = strlen (maindev->devlink) + sizeof ("pNN") + 1;
220                 doslink_len = strlen (maindev->devlink) + sizeof (":NNN") + 1;
913                 if ((doslink = (char *)calloc (1, doslink_len)) != NULL) {
914                         for (i = 1; i < 16; i++) {
915                                 snprintf(dospath, sizeof (dospath), "%sp%d", mai
916                                 snprintf(doslink, doslink_len, "%sp%d", maindev-
223                                 snprintf(dospath, sizeof (dospath), "%s:%d", mai
224                                 snprintf(doslink, doslink_len, "%s:%d", maindev-
```



```
917                                 m = devinfo_storage_new_minor(maindev_path, dosp
918                                 g_queue_push_tail (mq, m);
919                         }
920                         free (doslink);
921                 }
922         }

924         maindev_is_d0 = (strcmp (maindev->slice, "d0") == 0);

926         /* enqueue all volumes */
927         while (!g_queue_is_empty (mq)) {
928                 m = g_queue_pop_head (mq);

930                 /* if main device is d0, we'll throw away s2/p0 */
931                 if (maindev_is_d0 && (strcmp (m->slice, whole_disk) == 0)) {
932                         devinfo_storage_free_minor (m);
933                         continue;
934                 }
935                 /* don't do p0 on cdrom */
936                 if (is_cdrom && (strcmp (m->slice, "p0") == 0)) {
937                         devinfo_storage_free_minor (m);
938                         continue;
939                 }
940                 if (rescan) {
941                         /* in rescan mode, don't reprobe existing volumes */
942                         /* XXX detect volume removal? */
943                         volume = hal_device_store_match_key_value_string (hald_g
944                             "solaris.devfs_path", m->devpath);
945                         if ((volume == NULL) || !hal_device_has_capability(volum
946                                 devinfo_volume_add (parent, node, m);
947                         } else {
948                                 HAL_INFO(("rescan volume exists %s", m->devpath)
949                         }
950                 } else {
951                         devinfo_volume_add (parent, node, m);
952                 }
953                 devinfo_storage_free_minor (m);
954         }

956         if (maindev_path != NULL) {
957                 di_devfs_path_free (maindev_path);
958         }

960         return;

962 err:
963         if (maindev_path != NULL) {
964                 di_devfs_path_free (maindev_path);
965         }
966         if (!rescan) {
967                 devinfo_add_enqueue (parent, devfs_path, &devinfo_storage_handle
968         }
969 }
______unchanged_portion_omitted_

1395 static gboolean
1396 is_dos_path(char *path, int *partnum)
1397 {
1398         char *p;

1400         if ((p = strrchr (path, 'p')) == NULL) {
 708         if ((p = strrchr (path, ':')) == NULL) {
1401                 return (FALSE);
1402         }
1403         return ((*partnum = atoi(p + 1)) != 0);
1404 }
```

```
1406 static gboolean
1407 dos_to_dev(char *path, char **devpath, int *partnum)
1408 {
1409         char *p;

1411         if ((p = strrchr (path, 'p')) == NULL) {
 719         if ((p = strrchr (path, ':')) == NULL) {
1412                 return (FALSE);
1413         }
1414         if ((*partnum = atoi(p + 1)) == 0) {
1415                 return (FALSE);
1416         }
1417         p[0] = '\0';
1418         *devpath = strdup(path);
1419         p[0] = 'p';
 727         p[0] = ':';
1420         return (*devpath != NULL);
1421 }
```

*unchanged_portion_omitted_*