```
**********************************************************
   33462 Thu Jun 19 12:36:21 2014
new/usr/src/cmd/rmvolmgr/rmm_common.c
4845 rm(u)mount don't always print mount/unmount errors
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 *
 *
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 *
 * Copyright 2014 Andrew Stormont.
#endif /* ! codereview */
 */

#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <strings.h>
#include <stdarg.h>
#include <fcntl.h>
#include <libintl.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mnttab.h>

#include <dbus/dbus.h>
#include <dbus/dbus-glib.h>
#include <dbus/dbus-glib-lowlevel.h>
#include <libhal.h>
#include <libhal-storage.h>

#include "rmm_common.h"

#define RMM_PRINT_DEVICE_WIDTH  20

extern int rmm_debug;

static const char *action_strings[] = {
        "eject",
        "mount",
        "remount",
        "unmount",
        "clear_mounts",
        "closetray"
};


LibHalContext *
rmm_hal_init(LibHalDeviceAdded devadd_cb, LibHalDeviceRemoved devrem_cb,
    LibHalDevicePropertyModified propmod_cb, LibHalDeviceCondition cond_cb,
    DBusError *error, rmm_error_t *rmm_error)
{
        DBusConnection  *dbus_conn;
        LibHalContext   *ctx;
        char            **devices;
        int             nr;

        dbus_error_init(error);

        /*
         * setup D-Bus connection
         */
        if (!(dbus_conn = dbus_bus_get(DBUS_BUS_SYSTEM, error))) {
                dprintf("cannot get system bus: %s\n", rmm_strerror(error, -1));
                *rmm_error = RMM_EDBUS_CONNECT;
                return (NULL);
        }
        rmm_dbus_error_free(error);

        dbus_connection_setup_with_g_main(dbus_conn, NULL);
        dbus_connection_set_exit_on_disconnect(dbus_conn, B_TRUE);

        if ((ctx = libhal_ctx_new()) == NULL) {
                dprintf("libhal_ctx_new failed");
                *rmm_error = RMM_EHAL_CONNECT;
                return (NULL);
        }

        libhal_ctx_set_dbus_connection(ctx, dbus_conn);

        /*
         * register callbacks
         */
        if (devadd_cb != NULL) {
                libhal_ctx_set_device_added(ctx, devadd_cb);
        }
        if (devrem_cb != NULL) {
                libhal_ctx_set_device_removed(ctx, devrem_cb);
        }
        if (propmod_cb != NULL) {
                libhal_ctx_set_device_property_modified(ctx, propmod_cb);
                if (!libhal_device_property_watch_all(ctx, error)) {
                        dprintf("property_watch_all failed %s",
                            rmm_strerror(error, -1));
                        libhal_ctx_free(ctx);
                        *rmm_error = RMM_EHAL_CONNECT;
                        return (NULL);
                }
        }
        if (cond_cb != NULL) {
                libhal_ctx_set_device_condition(ctx, cond_cb);
        }

        if (!libhal_ctx_init(ctx, error)) {
                dprintf("libhal_ctx_init failed: %s", rmm_strerror(error, -1));
                libhal_ctx_free(ctx);
                *rmm_error = RMM_EHAL_CONNECT;
```

```
126                 return (NULL);
127         }
128         rmm_dbus_error_free(error);

130         /*
131          * The above functions do not guarantee that HAL is actually running.
132          * Check by invoking a method.
133          */
134         if (!(devices = libhal_get_all_devices(ctx, &nr, error))) {
135                 dprintf("HAL is not running: %s", rmm_strerror(error, -1));
136                 libhal_ctx_shutdown(ctx, NULL);
137                 libhal_ctx_free(ctx);
138                 *rmm_error = RMM_EHAL_CONNECT;
139                 return (NULL);
140         } else {
141                 rmm_dbus_error_free(error);
142                 libhal_free_string_array(devices);
143         }

145         return (ctx);
146 }


149 void
150 rmm_hal_fini(LibHalContext *hal_ctx)
151 {
152         DBusConnection  *dbus_conn = libhal_ctx_get_dbus_connection(hal_ctx);

154         (void) dbus_connection_unref(dbus_conn);
155         (void) libhal_ctx_free(hal_ctx);
156 }


159 /*
160  * find volume from any type of name, similar to the old media_findname()
161  * returns the LibHalDrive object and a list of LibHalVolume objects.
162  */
163 LibHalDrive *
164 rmm_hal_volume_find(LibHalContext *hal_ctx, const char *name, DBusError *error,
165     GSList **volumes)
166 {
167         LibHalDrive     *drive;
168         char            *p;
169         char            lastc;

171         *volumes = NULL;

173         /* temporarily remove trailing slash */
174         p = (char *)name + strlen(name) - 1;
175         if (*p == '/') {
176                 lastc = *p;
177                 *p = '\0';
178         } else {
179                 p = NULL;
180         }

182         if (name[0] == '/') {
183                 if (((drive = rmm_hal_volume_findby(hal_ctx,
184                     "info.udi", name, volumes)) != NULL) ||
185                     ((drive = rmm_hal_volume_findby(hal_ctx,
186                     "block.device", name, volumes)) != NULL) ||
187                     ((drive = rmm_hal_volume_findby(hal_ctx,
188                     "block.solaris.raw_device", name, volumes)) != NULL) ||
189                     ((drive = rmm_hal_volume_findby(hal_ctx,
190                     "volume.mount_point", name, volumes)) != NULL)) {
191                         goto out;
```



```
192                 } else {
193                         goto out;
194                 }
195         }

197         /* try volume label */
198         if ((drive = rmm_hal_volume_findby(hal_ctx,
199             "volume.label", name, volumes)) != NULL) {
200                 goto out;
201         }

203         drive = rmm_hal_volume_findby_nickname(hal_ctx, name, volumes);

205 out:
206         if (p != NULL) {
207                 *p = lastc;
208         }
209         return (drive);
210 }

212 /*
213  * find default volume. Returns volume pointer and name in 'name'.
214  */
215 LibHalDrive *
216 rmm_hal_volume_find_default(LibHalContext *hal_ctx, DBusError *error,
217     const char **name_out, GSList **volumes)
218 {
219         LibHalDrive     *drive;
220         static const char *names[] = { "floppy", "cdrom", "rmdisk" };
221         int             i;

223         *volumes = NULL;

225         for (i = 0; i < NELEM(names); i++) {
226                 if ((drive = rmm_hal_volume_findby_nickname(hal_ctx,
227                     names[i], volumes)) != NULL) {
228                         /*
229                          * Skip floppy if it has no media.
230                          * XXX might want to actually check for media
231                          * every time instead of relying on volcheck.
232                          */
233                         if ((strcmp(names[i], "floppy") != 0) ||
234                             libhal_device_get_property_bool(hal_ctx,
235                             libhal_drive_get_udi(drive),
236                             "storage.removable.media_available", NULL)) {
237                                 *name_out = names[i];
238                                 break;
239                         }
240                 }
241                 rmm_dbus_error_free(error);
242         }

244         return (drive);
245 }

247 /*
248  * find volume by property=value
249  * returns the LibHalDrive object and a list of LibHalVolume objects.
250  * XXX add support for multiple properties, reduce D-Bus traffic
251  */
252 LibHalDrive *
253 rmm_hal_volume_findby(LibHalContext *hal_ctx, const char *property,
254     const char *value, GSList **volumes)
255 {
256         DBusError       error;
257         LibHalDrive     *drive = NULL;
```

```
258         LibHalVolume    *v = NULL;
259         char            **udis;
260         int             num_udis;
261         int             i;
262         int             i_drive = -1;

264         *volumes = NULL;

266         dbus_error_init(&error);

268         /* get all devices with property=value */
269         if ((udis = libhal_manager_find_device_string_match(hal_ctx, property,
270             value, &num_udis, &error)) == NULL) {
271                 rmm_dbus_error_free(&error);
272                 return (NULL);
273         }

275         /* find volumes and drives among these devices */
276         for (i = 0; i < num_udis; i++) {
277                 rmm_dbus_error_free(&error);
278                 if (libhal_device_query_capability(hal_ctx, udis[i], "volume",
279                     &error)) {
280                         v = libhal_volume_from_udi(hal_ctx, udis[i]);
281                         if (v != NULL) {
282                                 *volumes = g_slist_prepend(*volumes, v);
283                         }
284                 } else if ((*volumes == NULL) &&
285                     libhal_device_query_capability(hal_ctx, udis[i], "storage",
286                     &error)) {
287                         i_drive = i;
288                 }
289         }

291         if (*volumes != NULL) {
292                 /* used prepend, preserve original order */
293                 *volumes = g_slist_reverse(*volumes);

295                 v = (LibHalVolume *)(*volumes)->data;
296                 drive = libhal_drive_from_udi(hal_ctx,
297                     libhal_volume_get_storage_device_udi(v));
298                 if (drive == NULL) {
299                         rmm_volumes_free (*volumes);
300                         *volumes = NULL;
301                 }
302         } else if (i_drive >= 0) {
303                 drive = libhal_drive_from_udi(hal_ctx, udis[i_drive]);
304         }

306         libhal_free_string_array(udis);
307         rmm_dbus_error_free(&error);

309         return (drive);
310 }

312 static void
313 rmm_print_nicknames_one(LibHalDrive *d, LibHalVolume *v,
314     const char *device, char **drive_nicknames)
315 {
316         const char      *volume_label = NULL;
317         const char      *mount_point = NULL;
318         boolean_t       comma;
319         int             i;

321         (void) printf("%-*s ", RMM_PRINT_DEVICE_WIDTH, device);
322         comma = B_FALSE;
```



```
324         if (drive_nicknames != NULL) {
325                 for (i = 0; drive_nicknames[i] != NULL; i++) {
326                         (void) printf("%s%s", comma ? "," : "",
327                             drive_nicknames[i]);
328                         comma = B_TRUE;
329                 }
330         }

332         if ((v != NULL) &&
333             ((volume_label = libhal_volume_get_label(v)) != NULL) &&
334             (strlen(volume_label) > 0)) {
335                 (void) printf("%s%s", comma ? "," : "", volume_label);
336                 comma = B_TRUE;
337         }

339         if ((v != NULL) &&
340             ((mount_point = libhal_volume_get_mount_point(v)) != NULL) &&
341             (strlen(mount_point) > 0)) {
342                 (void) printf("%s%s", comma ? "," : "", mount_point);
343                 comma = B_TRUE;
344         }

346         (void) printf("\n");
347 }

349 /*
350  * print nicknames for each available volume
351  *
352  * print_mask:
353  *   RMM_PRINT_MOUNTABLE        print only mountable volumes
354  *   RMM_PRINT_EJECTABLE        print volume-less ejectable drives
355  */
356 void
357 rmm_print_volume_nicknames(LibHalContext *hal_ctx, DBusError *error,
358     int print_mask)
359 {
360         char            **udis;
361         int             num_udis;
362         GSList          *volumes = NULL;
363         LibHalDrive     *d, *d_tmp;
364         LibHalVolume    *v;
365         const char      *device;
366         char            **nicknames;
367         int             i;
368         GSList          *j;
369         int             nprinted;

371         dbus_error_init(error);

373         if ((udis = libhal_find_device_by_capability(hal_ctx, "storage",
374             &num_udis, error)) == NULL) {
375                 rmm_dbus_error_free(error);
376                 return;
377         }

379         for (i = 0; i < num_udis; i++) {
380                 if ((d = libhal_drive_from_udi(hal_ctx, udis[i])) == NULL) {
381                         continue;
382                 }

384                 /* find volumes belonging to this drive */
385                 if ((d_tmp = rmm_hal_volume_findby(hal_ctx,
386                     "block.storage_device", udis[i], &volumes)) != NULL) {
387                         libhal_drive_free(d_tmp);
388                 }
```

```
390                 nicknames = libhal_device_get_property_strlist(hal_ctx,
391                     udis[i], "storage.solaris.nicknames", NULL);

393                 nprinted = 0;
394                 for (j = volumes; j != NULL; j = g_slist_next(j)) {
395                         v = (LibHalVolume *)(j->data);

397                         if ((device = libhal_volume_get_device_file(v)) ==
398                             NULL) {
399                                 continue;
400                         }
401                         if ((print_mask & RMM_PRINT_MOUNTABLE) &&
402                             (libhal_volume_get_fsusage(v) !=
403                             LIBHAL_VOLUME_USAGE_MOUNTABLE_FILESYSTEM)) {
404                                 continue;
405                         }

407                         rmm_print_nicknames_one(d, v, device, nicknames);
408                         nprinted++;
409                 }

411                 if ((nprinted == 0) &&
412                     (print_mask & RMM_PRINT_EJECTABLE) &&
413                     libhal_drive_requires_eject(d) &&
414                     ((device = libhal_drive_get_device_file(d)) != NULL)) {
415                         rmm_print_nicknames_one(d, NULL, device, nicknames);
416                 }

418                 libhal_free_string_array(nicknames);
419                 libhal_drive_free(d);
420                 rmm_volumes_free(volumes);
421                 volumes = NULL;
422         }

424         libhal_free_string_array(udis);
425 }

427 /*
428  * find volume by nickname
429  * returns the LibHalDrive object and a list of LibHalVolume objects.
430  */
431 LibHalDrive *
432 rmm_hal_volume_findby_nickname(LibHalContext *hal_ctx, const char *name,
433     GSList **volumes)
434 {
435         DBusError       error;
436         LibHalDrive     *drive = NULL;
437         LibHalDrive     *drive_tmp;
438         char            **udis;
439         int             num_udis;
440         char            **nicknames;
441         int             i, j;

443         *volumes = NULL;

445         dbus_error_init(&error);

447         if ((udis = libhal_find_device_by_capability(hal_ctx, "storage",
448             &num_udis, &error)) == NULL) {
449                 rmm_dbus_error_free(&error);
450                 return (NULL);
451         }

453         /* find a drive by nickname */
454         for (i = 0; (i < num_udis) && (drive == NULL); i++) {
455                 if ((nicknames = libhal_device_get_property_strlist(hal_ctx,
```



```
456                     udis[i], "storage.solaris.nicknames", &error)) == NULL) {
457                         rmm_dbus_error_free(&error);
458                         continue;
459                 }
460                 for (j = 0; (nicknames[j] != NULL) && (drive == NULL); j++) {
461                         if (strcmp(nicknames[j], name) == 0) {
462                                 drive = libhal_drive_from_udi(hal_ctx, udis[i]);
463                         }
464                 }
465                 libhal_free_string_array(nicknames);
466         }
467         libhal_free_string_array(udis);

469         if (drive != NULL) {
470                 /* found the drive, now find its volumes */
471                 if ((drive_tmp = rmm_hal_volume_findby(hal_ctx,
472                     "block.storage_device", libhal_drive_get_udi(drive),
473                     volumes)) != NULL) {
474                         libhal_drive_free(drive_tmp);
475                 }
476         }

478         rmm_dbus_error_free(&error);

480         return (drive);
481 }

483 void
484 rmm_volumes_free(GSList *volumes)
485 {
486         GSList  *i;

488         for (i = volumes; i != NULL; i = g_slist_next(i)) {
489                 libhal_volume_free((LibHalVolume *)(i->data));
490         }
491         g_slist_free(volumes);
492 }

494 /*
495  * Call HAL's Mount() method on the given device
496  */
497 boolean_t
498 rmm_hal_mount(LibHalContext *hal_ctx, const char *udi,
499     char **opts, int num_opts, char *mountpoint, DBusError *error)
500 {
501         DBusConnection  *dbus_conn = libhal_ctx_get_dbus_connection(hal_ctx);
502         DBusMessage     *dmesg, *reply;
503         char            *fstype;

505         dprintf("mounting %s...\n", udi);

507         if (!(dmesg = dbus_message_new_method_call("org.freedesktop.Hal", udi,
508             "org.freedesktop.Hal.Device.Volume", "Mount"))) {
509                 dprintf(
510                     "mount failed for %s: cannot create dbus message\n", udi);
511                 return (B_FALSE);
512         }

514         fstype = "";
515         if (mountpoint == NULL) {
516                 mountpoint = "";
517         }

519         if (!dbus_message_append_args(dmesg, DBUS_TYPE_STRING, &mountpoint,
520             DBUS_TYPE_STRING, &fstype,
521             DBUS_TYPE_ARRAY, DBUS_TYPE_STRING, &opts, num_opts,
```

```
522             DBUS_TYPE_INVALID)) {
523                 dprintf("mount failed for %s: cannot append args\n", udi);
524                 dbus_message_unref(dmesg);
525                 return (B_FALSE);
526         }

528         dbus_error_init(error);
529         if (!(reply = dbus_connection_send_with_reply_and_block(dbus_conn,
530             dmesg, RMM_MOUNT_TIMEOUT, error))) {
531                 dprintf("mount failed for %s: %s\n", udi, error->message);
532                 dbus_message_unref(dmesg);
533                 return (B_FALSE);
534         }

536         dprintf("mounted %s\n", udi);

538         dbus_message_unref(dmesg);
539         dbus_message_unref(reply);

541         rmm_dbus_error_free(error);

543         return (B_TRUE);
544 }


547 /*
548  * Call HAL's Unmount() method on the given device
549  */
550 boolean_t
551 rmm_hal_unmount(LibHalContext *hal_ctx, const char *udi, DBusError *error)
552 {
553         DBusConnection *dbus_conn = libhal_ctx_get_dbus_connection(hal_ctx);
554         DBusMessage *dmesg, *reply;
555         char **opts = NULL;

557         dprintf("unmounting %s...\n", udi);

559         if (!(dmesg = dbus_message_new_method_call("org.freedesktop.Hal", udi,
560             "org.freedesktop.Hal.Device.Volume", "Unmount"))) {
561                 dprintf(
562                     "unmount failed %s: cannot create dbus message\n", udi);
563                 return (B_FALSE);
564         }

566         if (!dbus_message_append_args(dmesg, DBUS_TYPE_ARRAY, DBUS_TYPE_STRING,
567             &opts, 0, DBUS_TYPE_INVALID)) {
568                 dprintf("unmount failed %s: cannot append args\n", udi);
569                 dbus_message_unref(dmesg);
570                 return (B_FALSE);
571         }

573         dbus_error_init(error);
574         if (!(reply = dbus_connection_send_with_reply_and_block(dbus_conn,
575             dmesg, RMM_UNMOUNT_TIMEOUT, error))) {
576                 dprintf("unmount failed for %s: %s\n", udi, error->message);
577                 dbus_message_unref(dmesg);
578                 return (B_FALSE);
579         }

581         dprintf("unmounted %s\n", udi);

583         dbus_message_unref(dmesg);
584         dbus_message_unref(reply);

586         rmm_dbus_error_free(error);
```



```
588         return (B_TRUE);
589 }


592 /*
593  * Call HAL's Eject() method on the given device
594  */
595 boolean_t
596 rmm_hal_eject(LibHalContext *hal_ctx, const char *udi, DBusError *error)
597 {
598         DBusConnection  *dbus_conn = libhal_ctx_get_dbus_connection(hal_ctx);
599         DBusMessage     *dmesg, *reply;
600         char            **options = NULL;
601         uint_t          num_options = 0;

603         dprintf("ejecting %s...\n", udi);

605         if (!(dmesg = dbus_message_new_method_call("org.freedesktop.Hal", udi,
606             "org.freedesktop.Hal.Device.Storage", "Eject"))) {
607                 dprintf("eject %s: cannot create dbus message\n", udi);
608                 return (B_FALSE);
609         }

611         if (!dbus_message_append_args(dmesg,
612             DBUS_TYPE_ARRAY, DBUS_TYPE_STRING, &options, num_options,
613             DBUS_TYPE_INVALID)) {
614                 dprintf("eject %s: cannot append args to dbus message ", udi);
615                 dbus_message_unref(dmesg);
616                 return (B_FALSE);
617         }

619         dbus_error_init(error);
620         if (!(reply = dbus_connection_send_with_reply_and_block(dbus_conn,
621             dmesg, RMM_EJECT_TIMEOUT, error))) {
622                 dprintf("eject %s: %s\n", udi, error->message);
623                 dbus_message_unref(dmesg);
624                 return (B_FALSE);
625         }

627         dprintf("ejected %s\n", udi);

629         dbus_message_unref(dmesg);
630         dbus_message_unref(reply);

632         rmm_dbus_error_free(error);

634         return (B_TRUE);
635 }

637 /*
638  * Call HAL's CloseTray() method on the given device
639  */
640 boolean_t
641 rmm_hal_closetray(LibHalContext *hal_ctx, const char *udi, DBusError *error)
642 {
643         DBusConnection  *dbus_conn = libhal_ctx_get_dbus_connection(hal_ctx);
644         DBusMessage     *dmesg, *reply;
645         char            **options = NULL;
646         uint_t          num_options = 0;

648         dprintf("closing tray %s...\n", udi);

650         if (!(dmesg = dbus_message_new_method_call("org.freedesktop.Hal", udi,
651             "org.freedesktop.Hal.Device.Storage", "CloseTray"))) {
652                 dprintf(
653                     "closetray failed for %s: cannot create dbus message\n",
```

```
654                     udi);
655                 return (B_FALSE);
656         }

658         if (!dbus_message_append_args(dmesg,
659             DBUS_TYPE_ARRAY, DBUS_TYPE_STRING, &options, num_options,
660             DBUS_TYPE_INVALID)) {
661                 dprintf("closetray %s: cannot append args to dbus message ",
662                     udi);
663                 dbus_message_unref(dmesg);
664                 return (B_FALSE);
665         }

667         dbus_error_init(error);
668         if (!(reply = dbus_connection_send_with_reply_and_block(dbus_conn,
669             dmesg, RMM_CLOSETRAY_TIMEOUT, error))) {
670                 dprintf("closetray failed for %s: %s\n", udi, error->message);
671                 dbus_message_unref(dmesg);
672                 return (B_FALSE);
673         }

675         dprintf("closetray ok %s\n", udi);

677         dbus_message_unref(dmesg);
678         dbus_message_unref(reply);

680         rmm_dbus_error_free(error);

682         return (B_TRUE);
683 }

685 /*
686  * Call HAL's Rescan() method on the given device
687  */
688 boolean_t
689 rmm_hal_rescan(LibHalContext *hal_ctx, const char *udi, DBusError *error)
690 {
691         DBusConnection  *dbus_conn = libhal_ctx_get_dbus_connection(hal_ctx);
692         DBusMessage     *dmesg, *reply;

694         dprintf("rescanning %s...\n", udi);

696         if (!(dmesg = dbus_message_new_method_call("org.freedesktop.Hal", udi,
697             "org.freedesktop.Hal.Device", "Rescan"))) {
698                 dprintf("rescan failed for %s: cannot create dbus message\n",
699                     udi);
700                 return (B_FALSE);
701         }

703         dbus_error_init(error);
704         if (!(reply = dbus_connection_send_with_reply_and_block(dbus_conn,
705             dmesg, -1, error))) {
706                 dprintf("rescan failed for %s: %s\n", udi, error->message);
707                 dbus_message_unref(dmesg);
708                 return (B_FALSE);
709         }

711         dprintf("rescan ok %s\n", udi);

713         dbus_message_unref(dmesg);
714         dbus_message_unref(reply);

716         rmm_dbus_error_free(error);

718         return (B_TRUE);
719 }
```



```
721 boolean_t
722 rmm_hal_claim_branch(LibHalContext *hal_ctx, const char *udi)
723 {
724         DBusError error;
725         DBusConnection  *dbus_conn = libhal_ctx_get_dbus_connection(hal_ctx);
726         DBusMessage *dmesg, *reply;
727         const char *claimed_by = "rmvolmgr";

729         dprintf("claiming branch %s...\n", udi);

731         if (!(dmesg = dbus_message_new_method_call("org.freedesktop.Hal",
732             "/org/freedesktop/Hal/Manager", "org.freedesktop.Hal.Manager",
733             "ClaimBranch"))) {
734                 dprintf("cannot create dbus message\n");
735                 return (B_FALSE);
736         }

738         if (!dbus_message_append_args(dmesg, DBUS_TYPE_STRING, &udi,
739             DBUS_TYPE_STRING, &claimed_by, DBUS_TYPE_INVALID)) {
740                 dprintf("cannot append args to dbus message\n");
741                 dbus_message_unref(dmesg);
742                 return (B_FALSE);
743         }

745         dbus_error_init(&error);
746         if (!(reply = dbus_connection_send_with_reply_and_block(dbus_conn,
747             dmesg, -1, &error))) {
748                 dprintf("cannot send dbus message\n");
749                 dbus_message_unref(dmesg);
750                 rmm_dbus_error_free(&error);
751                 return (B_FALSE);
752         }

754         dprintf("claim branch ok %s\n", udi);

756         dbus_message_unref(dmesg);
757         dbus_message_unref(reply);

759         return (B_TRUE);
760 }

762 boolean_t
763 rmm_hal_unclaim_branch(LibHalContext *hal_ctx, const char *udi)
764 {
765         DBusError error;
766         DBusConnection  *dbus_conn = libhal_ctx_get_dbus_connection(hal_ctx);
767         DBusMessage *dmesg, *reply;
768         const char *claimed_by = "rmvolmgr";

770         dprintf("unclaiming branch %s...\n", udi);

772         if (!(dmesg = dbus_message_new_method_call("org.freedesktop.Hal",
773             "/org/freedesktop/Hal/Manager", "org.freedesktop.Hal.Manager",
774             "UnclaimBranch"))) {
775                 dprintf("cannot create dbus message\n");
776                 return (B_FALSE);
777         }

779         if (!dbus_message_append_args(dmesg, DBUS_TYPE_STRING, &udi,
780             DBUS_TYPE_STRING, &claimed_by, DBUS_TYPE_INVALID)) {
781                 dprintf("cannot append args to dbus message\n");
782                 dbus_message_unref(dmesg);
783                 return (B_FALSE);
784         }
```

```
786         dbus_error_init(&error);
787         if (!(reply = dbus_connection_send_with_reply_and_block(dbus_conn,
788             dmesg, -1, &error))) {
789                 dprintf("cannot send dbus message\n");
790                 dbus_message_unref(dmesg);
791                 rmm_dbus_error_free(&error);
792                 return (B_FALSE);
793         }

795         dprintf("unclaim branch ok %s\n", udi);

797         dbus_message_unref(dmesg);
798         dbus_message_unref(reply);

800         return (B_TRUE);
801 }

803 static boolean_t
804 rmm_action_one(LibHalContext *hal_ctx, const char *name, action_t action,
805     const char *dev, const char *udi, LibHalVolume *v,
806     char **opts, int num_opts, char *mountpoint)
807 {
808         char            dev_str[MAXPATHLEN];
809         char            *mountp;
810         DBusError       error;
811         boolean_t       ret = B_FALSE;

813         dprintf("rmm_action_one %s %s\n", name, action_strings[action]);

815 #endif /* ! codereview */
816         if (strcmp(name, dev) == 0) {
817                 (void) snprintf(dev_str, sizeof (dev_str), name);
818         } else {
819                 (void) snprintf(dev_str, sizeof (dev_str), "%s %s", name, dev);
820         }

822         dbus_error_init(&error);

824         switch (action) {
825         case EJECT:
826                 ret = rmm_hal_eject(hal_ctx, udi, &error);
827                 break;
828         case INSERT:
829         case REMOUNT:
 24                 if (libhal_volume_is_mounted(v)) {
 25                         goto done;
 26                 }
830                 ret = rmm_hal_mount(hal_ctx, udi,
831                     opts, num_opts, mountpoint, &error);
832                 break;
833         case UNMOUNT:
 31                 if (!libhal_volume_is_mounted(v)) {
 32                         goto done;
 33                 }
834                 ret = rmm_hal_unmount(hal_ctx, udi, &error);
835                 break;
836         case CLOSETRAY:
837                 ret = rmm_hal_closetray(hal_ctx, udi, &error);
838                 break;
839         }

841         if (!ret) {
842                 (void) fprintf(stderr, gettext("%s of %s failed: %s\n"),
843                     action_strings[action], dev_str, rmm_strerror(&error, -1));
844                 goto done;
845         }
```



```
847         switch (action) {
848         case EJECT:
849                 (void) printf(gettext("%s ejected\n"), dev_str);
850                 break;
851         case INSERT:
852         case REMOUNT:
853                 mountp = rmm_get_mnttab_mount_point(dev);
854                 if (mountp != NULL) {
855                         (void) printf(gettext("%s mounted at %s\n"),
856                             dev_str, mountp);
857                         free(mountp);
858                 }
859                 break;
860         case UNMOUNT:
861                 (void) printf(gettext("%s unmounted\n"), dev_str);
862                 break;
863         case CLOSETRAY:
864                 (void) printf(gettext("%s tray closed\n"), dev_str);
865                 break;
866         }

868 done:
869         rmm_dbus_error_free(&error);
870         return (ret);
871 }

873 /*
874  * top level action routine
875  *
876  * If non-null 'aa' is passed, it will be used, otherwise a local copy
877  * will be created.
878  */
879 boolean_t
880 rmm_action(LibHalContext *hal_ctx, const char *name, action_t action,
881     struct action_arg *aap, char **opts, int num_opts, char *mountpoint)
882 {
883         DBusError       error;
884         GSList          *volumes, *i;
885         LibHalDrive     *d;
886         LibHalVolume    *v;
887         const char      *udi, *d_udi;
888         const char      *dev, *d_dev;
889         struct action_arg aa_local;
890         boolean_t       ret = B_FALSE;

892         dprintf("rmm_action %s %s\n", name, action_strings[action]);

894         if (aap == NULL) {
895                 bzero(&aa_local, sizeof (aa_local));
896                 aap = &aa_local;
897         }

899         dbus_error_init(&error);

901         /* find the drive and its volumes */
902         d = rmm_hal_volume_find(hal_ctx, name, &error, &volumes);
903         rmm_dbus_error_free(&error);
904         if (d == NULL) {
905                 (void) fprintf(stderr, gettext("cannot find '%s'\n"), name);
906                 return (B_FALSE);
907         }
908         d_udi = libhal_drive_get_udi(d);
909         d_dev = libhal_drive_get_device_file(d);
910         if ((d_udi == NULL) || (d_dev == NULL)) {
911                 goto out;
```

```
912         }

914         /*
915          * For those drives that do not require media eject,
916          * EJECT turns into UNMOUNT.
917          */
918         if ((action == EJECT) && !libhal_drive_requires_eject(d)) {
919                 action = UNMOUNT;
920         }

922         /*
923          * Assume anything other than EJECT and CLOSETRAY is a
924          * variant of mount or unmount and requires the device
925          * to have at least one volume.
926          */
927         if (volumes == NULL && (action != EJECT && action != CLOSETRAY)) {
928                 (void) fprintf(stderr,
929                     gettext("cannot %s device '%s' with no volumes\n"),
930                         action_strings[action], name);
931                 goto out;
932         }

934 #endif /* ! codereview */
935         /* per drive action */
936         if ((action == EJECT) || (action == CLOSETRAY)) {
937                 ret = rmm_action_one(hal_ctx, name, action, d_dev, d_udi, NULL,
938                     opts, num_opts, NULL);

940                 if (!ret || (action == CLOSETRAY)) {
941                         goto out;
942                 }
943         }

945         /* per volume action */
946         for (i = volumes; i != NULL; i = g_slist_next(i)) {
947                 v = (LibHalVolume *)i->data;
948                 udi = libhal_volume_get_udi(v);
949                 dev = libhal_volume_get_device_file(v);

951                 if ((udi == NULL) || (dev == NULL)) {
952                         continue;
953                 }
954                 if (aap == &aa_local) {
955                         if (!rmm_volume_aa_from_prop(hal_ctx, udi, v, aap)) {
956                                 dprintf("rmm_volume_aa_from_prop failed %s\n",
957                                     udi);
958                                 continue;
959                         }
960                 }
961                 aap->aa_action = action;

963                 /* ejected above, just need postprocess */
964                 if (action != EJECT) {
965                         ret = rmm_action_one(hal_ctx, name, action, dev, udi, v,
966                             opts, num_opts, mountpoint);
967                 }
968                 if (ret) {
969                         (void) vold_postprocess(hal_ctx, udi, aap);
970                 }

972                 if (aap == &aa_local) {
973                         rmm_volume_aa_free(aap);
974                 }
975         }

977 out:
```



```
978         if (volumes != NULL)
979 #endif /* ! codereview */
980                 rmm_volumes_free(volumes);
981         libhal_drive_free(d);

983         return (ret);
984 }


987 /*
988  * rescan by name
989  * if name is NULL, rescan all drives
990  */
991 boolean_t
992 rmm_rescan(LibHalContext *hal_ctx, const char *name, boolean_t query)
993 {
994         DBusError       error;
995         GSList          *volumes;
996         LibHalDrive     *drive = NULL;
997         const char      *drive_udi;
998         char            **udis;
999         int             num_udis;
1000        char            *nickname;
1001        char            **nicks = NULL;
1002        boolean_t       do_free_udis = FALSE;
1003        int             i;
1004        boolean_t       ret = B_FALSE;

1006        dprintf("rmm_rescan %s\n", name != NULL ? name : "all");

1008        dbus_error_init(&error);

1010        if (name != NULL) {
1011                if ((drive = rmm_hal_volume_find(hal_ctx, name, &error,
1012                    &volumes)) == NULL) {
1013                        rmm_dbus_error_free(&error);
1014                        (void) fprintf(stderr,
1015                            gettext("cannot find '%s'\n"), name);
1016                        return (B_FALSE);
1017                }
1018                rmm_dbus_error_free(&error);
1019                g_slist_free(volumes);

1021                drive_udi = libhal_drive_get_udi(drive);
1022                udis = (char **)&drive_udi;
1023                num_udis = 1;
1024        } else {
1025                if ((udis = libhal_find_device_by_capability(hal_ctx,
1026                    "storage", &num_udis, &error)) == NULL) {
1027                        rmm_dbus_error_free(&error);
1028                        return (B_TRUE);
1029                }
1030                rmm_dbus_error_free(&error);
1031                do_free_udis = TRUE;
1032        }

1034        for (i = 0; i < num_udis; i++) {
1035                if (name == NULL) {
1036                        nicks = libhal_device_get_property_strlist(hal_ctx,
1037                            udis[i], "storage.solaris.nicknames", NULL);
1038                        if (nicks != NULL) {
1039                                nickname = nicks[0];
1040                        } else {
1041                                nickname = "";
1042                        }
1043                }
```

```
1044                 if (!(ret = rmm_hal_rescan(hal_ctx, udis[i], &error))) {
1045                         (void) fprintf(stderr,
1046                             gettext("rescan of %s failed: %s\n"),
1047                             name ? name : nickname,
1048                             rmm_strerror(&error, -1));
1049                         libhal_free_string_array(nicks);
1050                         continue;
1051                 }
1052                 if (query) {
1053                         ret = libhal_device_get_property_bool(hal_ctx, udis[i],
1054                             "storage.removable.media_available", NULL);
1055                         if (ret) {
1056                                 printf(gettext("%s is available\n"),
1057                                     name ? name : nickname);
1058                         } else {
1059                                 printf(gettext("%s is not available\n"),
1060                                     name ? name : nickname);
1061                         }
1062                 }
1063                 libhal_free_string_array(nicks);
1064         }

1066         if (drive != NULL) {
1067                 libhal_drive_free(drive);
1068         }
1069         if (do_free_udis) {
1070                 libhal_free_string_array(udis);
1071         }

1073         return (ret);
1074 }


1077 /*
1078  * set action_arg from volume properties
1079  */
1080 boolean_t
1081 rmm_volume_aa_from_prop(LibHalContext *hal_ctx, const char *udi_arg,
1082     LibHalVolume *volume_arg, struct action_arg *aap)
1083 {
1084         LibHalVolume    *volume = volume_arg;
1085         const char      *udi = udi_arg;
1086         const char      *drive_udi;
1087         char            *volume_label;
1088         char            *mountpoint;
1089         int             len;
1090         int             ret = B_FALSE;

1092         /* at least udi or volume must be supplied */
1093         if ((udi == NULL) && (volume == NULL)) {
1094                 return (B_FALSE);
1095         }
1096         if (volume == NULL) {
1097                 if ((volume = libhal_volume_from_udi(hal_ctx, udi)) == NULL) {
1098                         dprintf("cannot get volume %s\n", udi);
1099                         goto out;
1100                 }
1101         }
1102         if (udi == NULL) {
1103                 if ((udi = libhal_volume_get_udi(volume)) == NULL) {
1104                         dprintf("cannot get udi\n");
1105                         goto out;
1106                 }
1107         }
1108         drive_udi = libhal_volume_get_storage_device_udi(volume);
```



```
1110         if (!(aap->aa_symdev = libhal_device_get_property_string(hal_ctx,
1111             drive_udi, "storage.solaris.legacy.symdev", NULL))) {
1112                 dprintf("property %s not found %s\n",
1113                     "storage.solaris.legacy.symdev", drive_udi);
1114                 goto out;
1115         }
1116         if (!(aap->aa_media = libhal_device_get_property_string(hal_ctx,
1117             drive_udi, "storage.solaris.legacy.media_type", NULL))) {
1118                 dprintf("property %s not found %s\n",
1119                     "storage.solaris.legacy.media_type", drive_udi);
1120                 goto out;
1121         }

1123         /* name is derived from volume label */
1124         aap->aa_name = NULL;
1125         if ((volume_label = (char *)libhal_device_get_property_string(hal_ctx,
1126             udi, "volume.label", NULL)) != NULL) {
1127                 if ((len = strlen(volume_label)) > 0) {
1128                         aap->aa_name = rmm_vold_convert_volume_label(
1129                             volume_label, len);
1130                         if (strlen(aap->aa_name) == 0) {
1131                                 free(aap->aa_name);
1132                                 aap->aa_name = NULL;
1133                         }
1134                 }
1135                 libhal_free_string(volume_label);
1136         }
1137         /* if no label, then unnamed_<mediatype> */
1138         if (aap->aa_name == NULL) {
1139                 aap->aa_name = (char *)calloc(1, sizeof ("unnamed_floppyNNNN"));
1140                 if (aap->aa_name == NULL) {
1141                         goto out;
1142                 }
1143                 (void) snprintf(aap->aa_name, sizeof ("unnamed_floppyNNNN"),
1144                     "unnamed_%s", aap->aa_media);
1145         }

1147         if (!(aap->aa_path = libhal_device_get_property_string(hal_ctx, udi,
1148             "block.device", NULL))) {
1149                 dprintf("property %s not found %s\n", "block.device", udi);
1150                 goto out;
1151         }
1152         if (!(aap->aa_rawpath = libhal_device_get_property_string(hal_ctx, udi,
1153             "block.solaris.raw_device", NULL))) {
1154                 dprintf("property %s not found %s\n",
1155                     "block.solaris.raw_device", udi);
1156                 goto out;
1157         }
1158         if (!(aap->aa_type = libhal_device_get_property_string(hal_ctx, udi,
1159             "volume.fstype", NULL))) {
1160                 dprintf("property %s not found %s\n", "volume.fstype", udi);
1161                 goto out;
1162         }
1163         if (!libhal_device_get_property_bool(hal_ctx, udi,
1164             "volume.is_partition", NULL)) {
1165                 aap->aa_partname = NULL;
1166         } else if (!(aap->aa_partname = libhal_device_get_property_string(
1167             hal_ctx, udi, "block.solaris.slice", NULL))) {
1168                 dprintf("property %s not found %s\n",
1169                     "block.solaris.slice", udi);
1170                 goto out;
1171         }
1172         if (!(mountpoint = libhal_device_get_property_string(hal_ctx, udi,
1173             "volume.mount_point", NULL))) {
1174                 dprintf("property %s not found %s\n",
1175                     "volume.mount_point", udi);
```

```
1176                    goto out;
1177            }
1178            /*
1179             * aa_mountpoint can be reallocated in rmm_volume_aa_update_mountpoint()
1180             * won't have to choose between free() or libhal_free_string() later on
1181             */
1182            aap->aa_mountpoint = strdup(mountpoint);
1183            libhal_free_string(mountpoint);
1184            if (aap->aa_mountpoint == NULL) {
1185                    dprintf("mountpoint is NULL %s\n", udi);
1186                    goto out;
1187            }

1189            ret = B_TRUE;

1191    out:
1192            if ((volume != NULL) && (volume != volume_arg)) {
1193                    libhal_volume_free(volume);
1194            }
1195            if (!ret) {
1196                    rmm_volume_aa_free(aap);
1197            }
1198            return (ret);
1199    }

1201    /* ARGSUSED */
1202    void
1203    rmm_volume_aa_update_mountpoint(LibHalContext *hal_ctx, const char *udi,
1204        struct action_arg *aap)
1205    {
1206            if (aap->aa_mountpoint != NULL) {
1207                    free(aap->aa_mountpoint);
1208            }
1209            aap->aa_mountpoint = rmm_get_mnttab_mount_point(aap->aa_path);
1210    }

1212    void
1213    rmm_volume_aa_free(struct action_arg *aap)
1214    {
1215            if (aap->aa_symdev != NULL) {
1216                    libhal_free_string(aap->aa_symdev);
1217                    aap->aa_symdev = NULL;
1218            }
1219            if (aap->aa_name != NULL) {
1220                    free(aap->aa_name);
1221                    aap->aa_name = NULL;
1222            }
1223            if (aap->aa_path != NULL) {
1224                    libhal_free_string(aap->aa_path);
1225                    aap->aa_path = NULL;
1226            }
1227            if (aap->aa_rawpath != NULL) {
1228                    libhal_free_string(aap->aa_rawpath);
1229                    aap->aa_rawpath = NULL;
1230            }
1231            if (aap->aa_type != NULL) {
1232                    libhal_free_string(aap->aa_type);
1233                    aap->aa_type = NULL;
1234            }
1235            if (aap->aa_media != NULL) {
1236                    libhal_free_string(aap->aa_media);
1237                    aap->aa_media = NULL;
1238            }
1239            if (aap->aa_partname != NULL) {
1240                    libhal_free_string(aap->aa_partname);
1241                    aap->aa_partname = NULL;
```



```
1242            }
1243            if (aap->aa_mountpoint != NULL) {
1244                    free(aap->aa_mountpoint);
1245                    aap->aa_mountpoint = NULL;
1246            }
1247    }

1249    /*
1250     * get device's mount point from mnttab
1251     */
1252    char *
1253    rmm_get_mnttab_mount_point(const char *special)
1254    {
1255            char            *mount_point = NULL;
1256            FILE            *f;
1257            struct mnttab   mnt;
1258            struct mnttab   mpref = { NULL, NULL, NULL, NULL, NULL };

1260            if ((f = fopen(MNTTAB, "r")) != NULL) {
1261                    mpref.mnt_special = (char *)special;
1262                    if (getmntany(f, &mnt, &mpref) == 0) {
1263                            mount_point = strdup(mnt.mnt_mountp);
1264                    }
1265                    fclose(f);
1266            }

1268            return (mount_point);
1269    }


1272    /*
1273     * get human readable string from error values
1274     */
1275    const char *
1276    rmm_strerror(DBusError *dbus_error, int rmm_error)
1277    {
1278            const char      *str;

1280            if ((dbus_error != NULL) && dbus_error_is_set(dbus_error)) {
1281                    str = dbus_error->message;
1282            } else {
1283                    switch (rmm_error) {
1284                    case RMM_EOK:
1285                            str = gettext("success");
1286                            break;
1287                    case RMM_EDBUS_CONNECT:
1288                            str = gettext("cannot connect to D-Bus");
1289                            break;
1290                    case RMM_EHAL_CONNECT:
1291                            str = gettext("cannot connect to HAL");
1292                            break;
1293                    default:
1294                            str = gettext("undefined error");
1295                            break;
1296                    }
1297            }

1299            return (str);
1300    }

1302    void
1303    rmm_dbus_error_free(DBusError *error)
1304    {
1305            if (error != NULL && dbus_error_is_set(error)) {
1306                    dbus_error_free(error);
1307            }
```

```
}

static int
rmm_vold_isbadchar(int c)
{
        int     ret_val = 0;


        switch (c) {
        case '/':
        case ';':
        case '|':
                ret_val = 1;
                break;
        default:
                if (iscntrl(c) || isspace(c)) {
                        ret_val = 1;
                }
        }

        return (ret_val);
}

char *
rmm_vold_convert_volume_label(const char *name, size_t len)
{
        char    buf[MAXNAMELEN+1];
        char    *s = buf;
        int     i;

        if (len > MAXNAMELEN) {
                len = MAXNAMELEN;
        }

        for (i = 0; i < len; i++) {
                if (name[i] == '\0') {
                        break;
                }
                if (isgraph((int)name[i])) {
                        if (isupper((int)name[i])) {
                                *s++ = tolower((int)name[i]);
                        } else if (rmm_vold_isbadchar((int)name[i])) {
                                *s++ = '_';
                        } else {
                                *s++ = name[i];
                        }
                }
        }
        *s = '\0';
        s = strdup(buf);

        return (s);
}

/*
 * swiped from mkdir.c
 */
int
makepath(char *dir, mode_t mode)
{
        int             err;
        char            *slash;


        if ((mkdir(dir, mode) == 0) || (errno == EEXIST)) {
                return (0);
```



```
        }
        if (errno != ENOENT) {
                return (-1);
        }
        if ((slash = strrchr(dir, '/')) == NULL) {
                return (-1);
        }
        *slash = '\0';
        err = makepath(dir, mode);
        *slash++ = '/';

        if (err || (*slash == '\0')) {
                return (err);
        }

        return (mkdir(dir, mode));
}


void
dprintf(const char *fmt, ...)
{

        va_list         ap;
        const char      *p;
        char            msg[BUFSIZ];
        char            *errmsg = strerror(errno);
        char            *s;

        if (rmm_debug == 0) {
                return;
        }

        (void) memset(msg, 0, BUFSIZ);

        /* scan for %m and replace with errno msg */
        s = &msg[strlen(msg)];
        p = fmt;

        while (*p != '\0') {
                if ((*p == '%') && (*(p+1) == 'm')) {
                        (void) strcat(s, errmsg);
                        p += 2;
                        s += strlen(errmsg);
                        continue;
                }
                *s++ = *p++;
        }
        *s = '\0';      /* don't forget the null byte */

        va_start(ap, fmt);
        (void) vfprintf(stderr, msg, ap);
        va_end(ap);
}
```