```
**********************************************************
   95794 Tue Apr 16 05:30:04 2019
new/usr/src/uts/common/io/lofi.c
10567 lofi should support basic EFI ioctl()s
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright (c) 1999, 2010, Oracle and/or its affiliates. All rights reserved.
  23  *
  24  * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
  25  * Copyright (c) 2016 Andrey Sokolov
  26  * Copyright 2016 Toomas Soome <tsoome@me.com>
  27  * Copyright 2019 Joyent, Inc.
  28  */

  30 /*
  31  * lofi (loopback file) driver - allows you to attach a file to a device,
  32  * which can then be accessed through that device. The simple model is that
  33  * you tell lofi to open a file, and then use the block device you get as
  34  * you would any block device. lofi translates access to the block device
  35  * into I/O on the underlying file. This is mostly useful for
  36  * mounting images of filesystems.
  37  *
  38  * lofi is controlled through /dev/lofictl - this is the only device exported
  39  * during attach, and is instance number 0. lofiadm communicates with lofi
  40  * through ioctls on this device. When a file is attached to lofi, block and
  41  * character devices are exported in /dev/lofi and /dev/rlofi. These devices
  42  * are identified by lofi instance number, and the instance number is also used
  43  * as the name in /dev/lofi.
  44  *
  45  * Virtual disks, or, labeled lofi, implements virtual disk support to
  46  * support partition table and related tools. Such mappings will cause
  47  * block and character devices to be exported in /dev/dsk and /dev/rdsk
  48  * directories.
  49  *
  50  * To support virtual disks, the instance number space is divided to two
  51  * parts, upper part for instance number and lower part for minor number
  52  * space to identify partitions and slices. The virtual disk support is
  53  * implemented by stacking cmlb module. For virtual disks, the partition
  54  * related ioctl calls are routed to cmlb module. Compression and encryption
  55  * is not supported for virtual disks.
  56  *
  57  * Mapped devices are tracked with state structures handled with
  58  * ddi_soft_state(9F) for simplicity.
  59  *
  60  * A file attached to lofi is opened when attached and not closed until
  61  * explicitly detached from lofi. This seems more sensible than deferring
```



```
  62  * the open until the /dev/lofi device is opened, for a number of reasons.
  63  * One is that any failure is likely to be noticed by the person (or script)
  64  * running lofiadm. Another is that it would be a security problem if the
  65  * file was replaced by another one after being added but before being opened.
  66  *
  67  * The only hard part about lofi is the ioctls. In order to support things
  68  * like 'newfs' on a lofi device, it needs to support certain disk ioctls.
  69  * So it has to fake disk geometry and partition information. More may need
  70  * to be faked if your favorite utility doesn't work and you think it should
  71  * (fdformat doesn't work because it really wants to know the type of floppy
  72  * controller to talk to, and that didn't seem easy to fake. Or possibly even
  73  * necessary, since we have mkfs_pcfs now).
  74  *
  75  * Normally, a lofi device cannot be detached if it is open (i.e. busy).  To
  76  * support simulation of hotplug events, an optional force flag is provided.
  77  * If a lofi device is open when a force detach is requested, then the
  78  * underlying file is closed and any subsequent operations return EIO.  When the
  79  * device is closed for the last time, it will be cleaned up at that time.  In
  80  * addition, the DKIOCSTATE ioctl will return DKIO_DEV_GONE when the device is
  81  * detached but not removed.
  82  *
  83  * If detach was requested and lofi device is not open, we will perform
  84  * unmap and remove the lofi instance.
  85  *
  86  * If the lofi device is open and the li_cleanup is set on ioctl request,
  87  * we set ls_cleanup flag to notify the cleanup is requested, and the
  88  * last lofi_close will perform the unmapping and this lofi instance will be
  89  * removed.
  90  *
  91  * If the lofi device is open and the li_force is set on ioctl request,
  92  * we set ls_cleanup flag to notify the cleanup is requested,
  93  * we also set ls_vp_closereq to notify IO tasks to return EIO on new
  94  * IO requests and wait in process IO count to become 0, indicating there
  95  * are no more IO requests. Since ls_cleanup is set, the last lofi_close
  96  * will perform unmap and this lofi instance will be removed.
  97  * See also lofi_unmap_file() for details.
  98  *
  99  * Once ls_cleanup is set for the instance, we do not allow lofi_open()
 100  * calls to succeed and can have last lofi_close() to remove the instance.
 101  *
 102  * Known problems:
 103  *
 104  *      UFS logging. Mounting a UFS filesystem image "logging"
 105  *      works for basic copy testing but wedges during a build of ON through
 106  *      that image. Some deadlock in lufs holding the log mutex and then
 107  *      getting stuck on a buf. So for now, don't do that.
 108  *
 109  *      Direct I/O. Since the filesystem data is being cached in the buffer
 110  *      cache, _and_ again in the underlying filesystem, it's tempting to
 111  *      enable direct I/O on the underlying file. Don't, because that deadlocks.
 112  *      I think to fix the cache-twice problem we might need filesystem support.
 113  *
 114  * Interesting things to do:
 115  *
 116  *      Allow multiple files for each device. A poor-man's metadisk, basically.
 117  *
 118  *      Pass-through ioctls on block devices. You can (though it's not
 119  *      documented), give lofi a block device as a file name. Then we shouldn't
 120  *      need to fake a geometry, however, it may be relevant if you're replacing
 121  *      metadisk, or using lofi to get crypto.
 122  *      It makes sense to do lofiadm -c aes -a /dev/dsk/c0t0d0s4 /dev/lofi/1
 123  *      and then in /etc/vfstab have an entry for /dev/lofi/1 as /export/home.
 124  *      In fact this even makes sense if you have lofi "above" metadisk.
 125  *
 126  * Encryption:
 127  *      Each lofi device can have its own symmetric key and cipher.
```

```
 128  *      They are passed to us by lofiadm(1m) in the correct format for use
 129  *      with the misc/kcf crypto_* routines.
 130  *
 131  *      Each block has its own IV, that is calculated in lofi_blk_mech(), based
 132  *      on the "master" key held in the lsp and the block number of the buffer.
 133  */

 135 #include <sys/types.h>
 136 #include <netinet/in.h>
 137 #include <sys/sysmacros.h>
 138 #include <sys/uio.h>
 139 #include <sys/kmem.h>
 140 #include <sys/cred.h>
 141 #include <sys/mman.h>
 142 #include <sys/errno.h>
 143 #include <sys/aio_req.h>
 144 #include <sys/stat.h>
 145 #include <sys/file.h>
 146 #include <sys/modctl.h>
 147 #include <sys/conf.h>
 148 #include <sys/debug.h>
 149 #include <sys/vnode.h>
 150 #include <sys/lofi.h>
 151 #include <sys/lofi_impl.h>      /* for cache structure */
 152 #include <sys/fcntl.h>
 153 #include <sys/pathname.h>
 154 #include <sys/filio.h>
 155 #include <sys/fdio.h>
 156 #include <sys/open.h>
 157 #include <sys/disp.h>
 158 #include <vm/seg_map.h>
 159 #include <sys/ddi.h>
 160 #include <sys/sunddi.h>
 161 #include <sys/zmod.h>
 162 #include <sys/id_space.h>
 163 #include <sys/mkdev.h>
 164 #include <sys/crypto/common.h>
 165 #include <sys/crypto/api.h>
 166 #include <sys/rctl.h>
 167 #include <sys/vtoc.h>
 168 #include <sys/scsi/scsi.h>      /* for DTYPE_DIRECT */
 169 #include <sys/scsi/impl/uscsi.h>
 170 #include <sys/sysevent/dev.h>
 171 #include <sys/efi_partition.h>
 172 #include <sys/note.h>
 173 #include <LzmaDec.h>

 175 #define NBLOCKS_PROP_NAME       "Nblocks"
 176 #define SIZE_PROP_NAME          "Size"
 177 #define ZONE_PROP_NAME          "zone"

 179 #define SETUP_C_DATA(cd, buf, len)              \
 180         (cd).cd_format = CRYPTO_DATA_RAW;       \
 181         (cd).cd_offset = 0;                     \
 182         (cd).cd_miscdata = NULL;                \
 183         (cd).cd_length = (len);                 \
 184         (cd).cd_raw.iov_base = (buf);           \
 185         (cd).cd_raw.iov_len = (len);

 187 #define UIO_CHECK(uio)  \
 188         if (((uio)->uio_loffset % DEV_BSIZE) != 0 || \
 189             ((uio)->uio_resid % DEV_BSIZE) != 0) { \
 190                 return (EINVAL); \
 191         }

 193 #define LOFI_TIMEOUT    30
```



```
 195 static void *lofi_statep;
 196 static kmutex_t lofi_lock;              /* state lock */
 197 static id_space_t *lofi_id;             /* lofi ID values */
 198 static list_t lofi_list;
 199 static zone_key_t lofi_zone_key;

 201 /*
 202  * Because lofi_taskq_nthreads limits the actual swamping of the device, the
 203  * maxalloc parameter (lofi_taskq_maxalloc) should be tuned conservatively
 204  * high.  If we want to be assured that the underlying device is always busy,
 205  * we must be sure that the number of bytes enqueued when the number of
 206  * enqueued tasks exceeds maxalloc is sufficient to keep the device busy for
 207  * the duration of the sleep time in taskq_ent_alloc().  That is, lofi should
 208  * set maxalloc to be the maximum throughput (in bytes per second) of the
 209  * underlying device divided by the minimum I/O size.  We assume a realistic
 210  * maximum throughput of one hundred megabytes per second; we set maxalloc on
 211  * the lofi task queue to be 104857600 divided by DEV_BSIZE.
 212  */
 213 static int lofi_taskq_maxalloc = 104857600 / DEV_BSIZE;
 214 static int lofi_taskq_nthreads = 4;     /* # of taskq threads per device */

 216 const char lofi_crypto_magic[6] = LOFI_CRYPTO_MAGIC;

 218 /*
 219  * To avoid decompressing data in a compressed segment multiple times
 220  * when accessing small parts of a segment's data, we cache and reuse
 221  * the uncompressed segment's data.
 222  *
 223  * A single cached segment is sufficient to avoid lots of duplicate
 224  * segment decompress operations. A small cache size also reduces the
 225  * memory footprint.
 226  *
 227  * lofi_max_comp_cache is the maximum number of decompressed data segments
 228  * cached for each compressed lofi image. It can be set to 0 to disable
 229  * caching.
 230  */

 232 uint32_t lofi_max_comp_cache = 1;

 234 static int gzip_decompress(void *src, size_t srclen, void *dst,
 235         size_t *destlen, int level);

 237 static int lzma_decompress(void *src, size_t srclen, void *dst,
 238         size_t *dstlen, int level);

 240 lofi_compress_info_t lofi_compress_table[LOFI_COMPRESS_FUNCTIONS] = {
 241         {gzip_decompress,       NULL,   6,      "gzip"}, /* default */
 242         {gzip_decompress,       NULL,   6,      "gzip-6"},
 243         {gzip_decompress,       NULL,   9,      "gzip-9"},
 244         {lzma_decompress,       NULL,   0,      "lzma"}
 245 };
_____unchanged_portion_omitted_

1744 /*ARGSUSED2*/
1746 static int
1747 lofi_read(dev_t dev, struct uio *uio, struct cred *credp)
1748 {
1749         _NOTE(ARGUNUSED(credp));

1751         if (getminor(dev) == 0)
1752                 return (EINVAL);
1753         UIO_CHECK(uio);
1754         return (physio(lofi_strategy, NULL, dev, B_READ, minphys, uio));
1755 }
```

```
1754 /*ARGSUSED2*/
1757 static int
1758 lofi_write(dev_t dev, struct uio *uio, struct cred *credp)
1759 {
1760         _NOTE(ARGUNUSED(credp));

1762         if (getminor(dev) == 0)
1763                 return (EINVAL);
1764         UIO_CHECK(uio);
1765         return (physio(lofi_strategy, NULL, dev, B_WRITE, minphys, uio));
1766 }

1768 static int
1769 lofi_urw(struct lofi_state *lsp, uint16_t fmode, diskaddr_t off, size_t size,
1770     intptr_t arg, int flag, cred_t *credp)
1771 {
1772         struct uio uio;
1773         iovec_t iov;

1775         /*
1776          * 1024 * 1024 apes cmlb_tg_max_efi_xfer as a reasonable max.
1777          */
1778         if (size == 0 || size > 1024 * 1024 ||
1779             (size % (1 << lsp->ls_lbshift)) != 0)
1780                 return (EINVAL);

1782         iov.iov_base = (void *)arg;
1783         iov.iov_len = size;
1784         uio.uio_iov = &iov;
1785         uio.uio_iovcnt = 1;
1786         uio.uio_loffset = off;
1787         uio.uio_segflg = (flag & FKIOCTL) ? UIO_SYSSPACE : UIO_USERSPACE;
1788         uio.uio_llimit = MAXOFFSET_T;
1789         uio.uio_resid = size;
1790         uio.uio_fmode = fmode;
1791         uio.uio_extflg = 0;

1793         return (fmode == FREAD ?
1794             lofi_read(lsp->ls_dev, &uio, credp) :
1795             lofi_write(lsp->ls_dev, &uio, credp));
1796 }

1798 /*ARGSUSED2*/
1799 static int
1800 lofi_aread(dev_t dev, struct aio_req *aio, struct cred *credp)
1801 {
1802         if (getminor(dev) == 0)
1803                 return (EINVAL);
1804         UIO_CHECK(aio->aio_uio);
1805         return (aphysio(lofi_strategy, anocancel, dev, B_READ, minphys, aio));
1806 }
______unchanged_portion_omitted_

3218 static int
3219 lofi_ioctl(dev_t dev, int cmd, intptr_t arg, int flag, cred_t *credp,
3220     int *rvalp)
3221 {
3222         int error;
3223         enum dkio_state dkstate;
3224         struct lofi_state *lsp;
3225         dk_efi_t user_efi;
3226         int id;

3228         id = LOFI_MINOR2ID(getminor(dev));

3230         /* lofi ioctls only apply to the master device */
```



```
3231         if (id == 0) {
3232                 struct lofi_ioctl *lip = (struct lofi_ioctl *)arg;

3234                 /*
3235                  * the query command only need read-access - i.e., normal
3236                  * users are allowed to do those on the ctl device as
3237                  * long as they can open it read-only.
3238                  */
3239                 switch (cmd) {
3240                 case LOFI_MAP_FILE:
3241                         if ((flag & FWRITE) == 0)
3242                                 return (EPERM);
3243                         return (lofi_map_file(dev, lip, 1, rvalp, credp, flag));
3244                 case LOFI_MAP_FILE_MINOR:
3245                         if ((flag & FWRITE) == 0)
3246                                 return (EPERM);
3247                         return (lofi_map_file(dev, lip, 0, rvalp, credp, flag));
3248                 case LOFI_UNMAP_FILE:
3249                         if ((flag & FWRITE) == 0)
3250                                 return (EPERM);
3251                         return (lofi_unmap_file(lip, 1, credp, flag));
3252                 case LOFI_UNMAP_FILE_MINOR:
3253                         if ((flag & FWRITE) == 0)
3254                                 return (EPERM);
3255                         return (lofi_unmap_file(lip, 0, credp, flag));
3256                 case LOFI_GET_FILENAME:
3257                         return (lofi_get_info(dev, lip, LOFI_GET_FILENAME,
3258                             credp, flag));
3259                 case LOFI_GET_MINOR:
3260                         return (lofi_get_info(dev, lip, LOFI_GET_MINOR,
3261                             credp, flag));

3263                 /*
3264                  * This API made limited sense when this value was fixed
3265                  * at LOFI_MAX_FILES.  However, its use to iterate
3266                  * across all possible devices in lofiadm means we don't
3267                  * want to return L_MAXMIN, but the highest
3268                  * *allocated* id.
3269                  */
3270                 case LOFI_GET_MAXMINOR:
3271                         id = 0;

3273                         mutex_enter(&lofi_lock);

3275                         for (lsp = list_head(&lofi_list); lsp != NULL;
3276                             lsp = list_next(&lofi_list, lsp)) {
3277                                 int i;
3278                                 if (lofi_access(lsp) != 0)
3279                                         continue;

3281                                 i = ddi_get_instance(lsp->ls_dip);
3282                                 if (i > id)
3283                                         id = i;
3284                         }

3286                         mutex_exit(&lofi_lock);

3288                         error = ddi_copyout(&id, &lip->li_id,
3289                             sizeof (id), flag);
3290                         if (error)
3291                                 return (EFAULT);
3292                         return (0);

3294                 case LOFI_CHECK_COMPRESSED:
3295                         return (lofi_get_info(dev, lip, LOFI_CHECK_COMPRESSED,
3296                             credp, flag));
```

```
3297                 default:
3298                         return (EINVAL);
3299                 }
3300         }

3302         mutex_enter(&lofi_lock);
3303         lsp = ddi_get_soft_state(lofi_statep, id);
3304         if (lsp == NULL || lsp->ls_cleanup) {
3305                 mutex_exit(&lofi_lock);
3306                 return (ENXIO);
3307         }
3308         mutex_exit(&lofi_lock);

3310         if (ddi_prop_exists(DDI_DEV_T_ANY, lsp->ls_dip, DDI_PROP_DONTPASS,
3311             "labeled") == 1) {
3312                 error = cmlb_ioctl(lsp->ls_cmlbhandle, dev, cmd, arg, flag,
3313                     credp, rvalp, 0);
3314                 if (error != ENOTTY)
3315                         return (error);
3316         }

3318         /*
3319          * We explicitly allow DKIOCSTATE, but all other ioctls should fail with
3320          * EIO as if the device was no longer present.
3321          */
3322         if (lsp->ls_vp == NULL && cmd != DKIOCSTATE)
3323                 return (EIO);

3325         /* these are for faking out utilities like newfs */
3326         switch (cmd) {
3327         case DKIOCGMEDIAINFO:
3328         case DKIOCGMEDIAINFOEXT: {
3329                 struct dk_minfo_ext media_info;
3330                 int shift = lsp->ls_lbshift;
3331                 int size;

3333                 if (cmd == DKIOCGMEDIAINFOEXT) {
3334                         media_info.dki_pbsize = 1U << lsp->ls_pbshift;
3335                         size = sizeof (struct dk_minfo_ext);
3336                 } else {
3337                         size = sizeof (struct dk_minfo);
3338                 }

3340                 media_info.dki_media_type = DK_FIXED_DISK;
3341                 media_info.dki_lbsize = 1U << shift;
3342                 media_info.dki_capacity =
3343                     (lsp->ls_vp_size - lsp->ls_crypto_offset) >> shift;

3345                 if (ddi_copyout(&media_info, (void *)arg, size, flag))
3346                         return (EFAULT);
3347                 return (0);
3348         }
3349         case DKIOCREMOVABLE: {
3350                 int i = 0;
3351                 if (ddi_copyout(&i, (caddr_t)arg, sizeof (int), flag))
3352                         return (EFAULT);
3353                 return (0);
3354         }

3356         case DKIOCGVTOC: {
3357                 struct vtoc vt;
3358                 fake_disk_vtoc(lsp, &vt);

3360                 switch (ddi_model_convert_from(flag & FMODELS)) {
3361                 case DDI_MODEL_ILP32: {
3362                         struct vtoc32 vtoc32;
```



```
3364                         vtoctovtoc32(vt, vtoc32);
3365                         if (ddi_copyout(&vtoc32, (void *)arg,
3366                             sizeof (struct vtoc32), flag))
3367                                 return (EFAULT);
3368                         break;
3369                         }

3371                 case DDI_MODEL_NONE:
3372                         if (ddi_copyout(&vt, (void *)arg,
3373                             sizeof (struct vtoc), flag))
3374                                 return (EFAULT);
3375                         break;
3376                 }
3377                 return (0);
3378         }
3379         case DKIOCINFO: {
3380                 struct dk_cinfo ci;
3381                 fake_disk_info(dev, &ci);
3382                 if (ddi_copyout(&ci, (void *)arg, sizeof (ci), flag))
3383                         return (EFAULT);
3384                 return (0);
3385         }
3386         case DKIOCG_VIRTGEOM:
3387         case DKIOCG_PHYGEOM:
3388         case DKIOCGGEOM:
3389                 error = ddi_copyout(&lsp->ls_dkg, (void *)arg,
3390                     sizeof (struct dk_geom), flag);
3391                 if (error)
3392                         return (EFAULT);
3393                 return (0);
3394         case DKIOCSTATE:
3395                 /*
3396                  * Normally, lofi devices are always in the INSERTED state.  If
3397                  * a device is forcefully unmapped, then the device transitions
3398                  * to the DKIO_DEV_GONE state.
3399                  */
3400                 if (ddi_copyin((void *)arg, &dkstate, sizeof (dkstate),
3401                     flag) != 0)
3402                         return (EFAULT);

3404                 mutex_enter(&lsp->ls_vp_lock);
3405                 while (((dkstate == DKIO_INSERTED && lsp->ls_vp != NULL) ||
3406                     (dkstate == DKIO_DEV_GONE && lsp->ls_vp == NULL)) &&
3407                     !lsp->ls_cleanup) {
3408                         /*
3409                          * By virtue of having the device open, we know that
3410                          * 'lsp' will remain valid when we return.
3411                          */
3412                         if (!cv_wait_sig(&lsp->ls_vp_cv, &lsp->ls_vp_lock)) {
3413                                 mutex_exit(&lsp->ls_vp_lock);
3414                                 return (EINTR);
3415                         }
3416                 }

3418                 dkstate = (!lsp->ls_cleanup && lsp->ls_vp != NULL ?
3419                     DKIO_INSERTED : DKIO_DEV_GONE);
3420                 mutex_exit(&lsp->ls_vp_lock);

3422                 if (ddi_copyout(&dkstate, (void *)arg,
3423                     sizeof (dkstate), flag) != 0)
3424                         return (EFAULT);
3425                 return (0);
3426         case USCSICMD: {
3427                 struct uscsi_cmd uscmd;
3428                 union scsi_cdb cdb;
```

```
3430                 if (uscsi_is_inquiry(arg, flag, &cdb, &uscmd) == 0) {
3431                         struct scsi_inquiry inq = {0};

3433                         lofi_create_inquiry(lsp, &inq);
3434                         if (ddi_copyout(&inq, uscmd.uscsi_bufaddr,
3435                             uscmd.uscsi_buflen, flag) != 0)
3436                                 return (EFAULT);
3437                         return (0);
3438                 } else if (cdb.scc_cmd == SCMD_READ_CAPACITY) {
3439                         struct scsi_capacity capacity;

3441                         capacity.capacity =
3442                             BE_32((lsp->ls_vp_size - lsp->ls_crypto_offset) >>
3443                             lsp->ls_lbshift);
3444                         capacity.lbasize = BE_32(1 << lsp->ls_lbshift);
3445                         if (ddi_copyout(&capacity, uscmd.uscsi_bufaddr,
3446                             uscmd.uscsi_buflen, flag) != 0)
3447                                 return (EFAULT);
3448                         return (0);
3449                 }

3451                 uscmd.uscsi_rqstatus = 0xff;
3452 #ifdef  _MULTI_DATAMODEL
3453                 switch (ddi_model_convert_from(flag & FMODELS)) {
3454                 case DDI_MODEL_ILP32: {
3455                         struct uscsi_cmd32 ucmd32;
3456                         uscsi_cmdtouscsi_cmd32((&uscmd), (&ucmd32));
3457                         if (ddi_copyout(&ucmd32, (void *)arg, sizeof (ucmd32),
3458                             flag) != 0)
3459                                 return (EFAULT);
3460                         break;
3461                 }
3462                 case DDI_MODEL_NONE:
3463                         if (ddi_copyout(&uscmd, (void *)arg, sizeof (uscmd),
3464                             flag) != 0)
3465                                 return (EFAULT);
3466                         break;
3467                 default:
3468                         return (EFAULT);
3469                 }
3470 #else
3471                 if (ddi_copyout(&uscmd, (void *)arg, sizeof (uscmd), flag) != 0)
3472                         return (EFAULT);
3473 #endif  /* _MULTI_DATAMODEL */
3474                 return (0);
3475         }

3477         case DKIOCGMBOOT:
3478                 return (lofi_urw(lsp, FREAD, 0, 1 << lsp->ls_lbshift,
3479                     arg, flag, credp));

3481         case DKIOCSMBOOT:
3482                 return (lofi_urw(lsp, FWRITE, 0, 1 << lsp->ls_lbshift,
3483                     arg, flag, credp));

3485         case DKIOCGETEFI:
3486                 if (ddi_copyin((void *)arg, &user_efi,
3487                     sizeof (dk_efi_t), flag) != 0)
3488                         return (EFAULT);

3490                 return (lofi_urw(lsp, FREAD,
3491                     user_efi.dki_lba * (1 << lsp->ls_lbshift),
3492                     user_efi.dki_length, (intptr_t)user_efi.dki_data,
3493                     flag, credp));
```



```
3495         case DKIOCSETEFI:
3496                 if (ddi_copyin((void *)arg, &user_efi,
3497                     sizeof (dk_efi_t), flag) != 0)
3498                         return (EFAULT);

3500                 return (lofi_urw(lsp, FWRITE,
3501                     user_efi.dki_lba * (1 << lsp->ls_lbshift),
3502                     user_efi.dki_length, (intptr_t)user_efi.dki_data,
3503                     flag, credp));

3505         default:
3506 #ifdef DEBUG
3507                 cmn_err(CE_WARN, "lofi_ioctl: %d is not implemented\n", cmd);
3508 #endif  /* DEBUG */
3509                 return (ENOTTY);
3510         }
3511 }
```

*unchanged_portion_omitted_*