NAME

ddi_ufm, **ddi_ufm_op_nimages**, **ddi_ufm_op_fill_image**, **ddi_ufm_op_nslots**, **ddi_ufm_op_fil_slot -** DDI upgradable firmware module entry points

SYNOPSIS

typedef struct ddi_ufm_handle ddi_ufm_handle_t
typedef struct ddi_ufm_ops ddi_ufm_ops_t

#include <sys/ddi_ufm.h>

int
ddi_ufm_op_nimages(*ddi_ufm_handle_t *uhp*, *void *drv_arg*, *uint_t *nimgp*);

int
ddi_ufm_op_fill_image(*ddi_ufm_handle_t *uhp*, *void *drv_arg*, *uint_t imgid*, *ddi_ufm_image_t *uip*);

int
ddi_ufm_op_nslots(*ddi_ufm_handle_t *uhp*, *void *drv_arg*, *uint_t *nslotsp*);

int
ddi_ufm_fill_slot(*ddi_ufm_handle_t *uhp*, *void *drv_arg*, *uint_t imgid*, *uint_t slotid*,
*ddi_ufm_slot_t *usp*);

INTERFACE LEVEL

Evolving - This interface is evolving still in illumos. API and ABI stability is not guaranteed.

PARAMETERS

uhp A handle corresponding to the device's UFM handle. This is the same value as returned in ddi_ufm_init(9F).

drv_arg This is a private value that the drier passed in when calling ddi_ufm_init(9F). *nimgp* A pointer that the driver should set with a number of images. *nslotp* A pointer that the driver should set with a number of slots.

imgid An integer indicating which image information is being requested for.

uip An opaque pointer that represents a UFM image.

slotid An integer indicating which slot information is being requested for.

usp An opaque pointer that represents a UFM slot.

DESCRIPTION

Upgradable firmware modules (UFM) are a potential component of many devices. These interfaces aim to provide a simple series of callbacks for a device driver to implement such that it is easy to report information and in the future, manipulate firmware modules.

UFM Background

UFMs may come in different flavors and styles ranging from a firmware blob, to an EEPROM image, to microcode, and more. Take for example a hard drive. While it contains a field replaceable unit (FRU), it also contains some amount of firmware that manages the drive and can be updated independently of replacing the drive.

The motherboard often has a UFM in the form of the BIOS or UEFI. The Lights out management controller on a system has a UFM, which is usually the entire system image. CPUs also have a UFM in the form of microcode.

An important property of a UFM is that it is a property of the device itself. For example, many WiFi device drivers are required to send a binary blob of firmware to the device after every reset. Because these images are not properties of the device and must be upgraded by either changing the device driver or related system files, we do not consider these UFMs.

There are also devices that have firmware which is a property of the device, but may not be upgradable from the running OS. This may be because the vendor doesn’t have tooling to upgrade the image or because the firmware image itself cannot be upgraded in the field at all. For example, a YubiKey has a firmware image that’s burned into it in the factory, but there is no way to change the firmware on it short of replacing the device in its entirety. However, because these images are a permanent part of the device, we also consider them a UFM.

Images and Slots

A device that supports UFMs is made up of one or more distinct firmware images. Each image has its own unique purpose. For example, a motherboard may have both a BIOS and a CPLD image, each of which has independent firmware revisions.

A given image may have a number of slots. A slot represents a particular version of the image. Only one slot can be active at a given time. Devices support slots such that a firmware image can be downloaded to the device without impacting the current device if it fails half-way through. The slot that’s currently in use is referred to as the *active* slot.

The various entry points are designed such that all a driver has to do is provide information about the image and its slots to the kernel, it does not have to wrangle with how that is marshalled to users and the appearance of those structures.

Registering with the UFM Subsystem

During a device driver’s attach(9E) entry point, a device driver should register with the UFM subsystem by filling out a UFM operations vector and then calling ddi_ufm_init(9F). The driver may pass in a value, usually a pointer to its soft state pointer, which it will then receive when its subsequent entry points are called.

Once the driver has finished initializing, it must call ddi_ufm_update(9F) to indicate that the driver is in a state where it’s ready to receive calls to the entry points.

The various UFM entry points may be called from an arbitrary kernel context. However, they will only ever be called from a single thread at a given time.

UFM operations vector

The UFM operations vector is a structure that has the following members:

```
typedef struct ddi_ufm_ops {
        int (*ddi_ufm_op_nimages)(ddi_ufm_handle_t *uhp, void *arg,
            uint_t *nimgp);
        int (*ddi_ufm_op_fill_image)(ddi_ufm_handle_t *uhp, void *arg,
            uint_t imgid, ddi_ufm_image_t *img);
        int (*ddi_ufm_op_nslots)(ddi_ufm_handle_t *uhp, void *arg,
            uint_t imgid, uint_t *nslots);
        int (*ddi_ufm_op_fill_slot)(ddi_ufm_handle_t *uhp, void *arg,
            int imgid, ddi_ufm_image_t *img, uint_t slotid,
            ddi_ufm_slot_t *slotp);
} ddi_ufm_ops_t;
```

Of the four members, only the **ddi_ufm_op_fill_image**() and **ddi_ufm_op_fill_slot**() are required. The other entry points are optional. If a device only has a single image, then there is no reason to implement the **ddi_ufm_op_nimages**() entry point. The system will assume that there is only a single image. The same holds true for the **ddi_ufm_op_nslots**() entry point. If only a single slot is supported, then the driver should not bother implementing this entry point.

Slots and images are numbered starting at zero. If a driver indicates support for multiple images or slots then the images or slots will be numbered sequentially going from 0 to the number of images or slots minus one. These values will be passed to the various entry points to indicate which image and slot the system is interested in. It is up to the driver to maintain a consistent view of the images and slots for a given UFM.

The members of this structure should be filled in the following ways:

ddi_ufm_op_nimages()

The **ddi_ufm_op_nimages**() entry point is an optional entry point that answers the question of how many different, distinct firmware images are present on the device. Once the driver determines how many are present, it should set the value in *nimgp* to the determined value.

It is legal for a device to pass in zero for this value, which indicates that there are none present.

Upon successful completion, the driver should return **0**. Otherwise, the driver should return the appropriate error number. For a full list of error numbers, see Intro(2). Common values are:

EIO	An error occurred while communicating with the device to determine the number of firmware images.

ddi_ufm_op_fill_image()

The **ddi_ufm_op_fill_image**() entry point is used to fill in information about a given image. The value in *imgid* is used to indicate which image the system is asking to fill information about. If the driver does not recognize the image ID in *imgid* then it should return an error.

The *ddi_ufm_image_t* structure passed in *uip* is opaque. To fill in information about the image, the driver should call the functions described in ddi_ufm_image(9F).

The driver should call the ddi_ufm_image_set_desc(9F) function to set a description of the image which indicates its purpose. This should be a human-readable string. The driver may also set any ancillary data that it deems may be useful with the ddi_ufm_image_set_misc(9F) function. This function takes an nvlist, allowing the driver to set arbitrary keys and values.

Once the driver has finished setting all of the information about the image then the driver should return **0**. Otherwise, the driver should return the appropriate error number. For a full list of error numbers, see Intro(2). Common values are:

EINVAL	The image indicated by *imgid* is unknown.
EIO	An error occurred talking to the device while trying to fill out firmware image information.
ENOMEM	The driver was unable to allocate memory while filling out image information.

ddi_ufm_op_nslots()

The **ddi_ufm_op_nslots**() entry point is an optional entry point that answers the question of how many different slots exist for the firmware image indicated by *imgid*. If *imgid* is an unknown image, then the driver should return an error.

Once the driver has determined the number of slots, it should update *nslotp* with the number.

A device driver should not set *nslotp* to zero as every firmware image is required to have a slot.

Upon successful completion, the driver should return **0**. Otherwise, the driver should return the appropriate error number. For a full list of error numbers, see Intro(2). Common values are:

EINVAL	The image indicated by *imgid* is unknown.
EIO	An error occurred while communicating with the device to determine the number of slots for the firmware image.

ddi_ufm_op_fill_slot()

The **ddi_ufm_op_fill_slot**() function is used to fill in information about a specific slot for a specific image. The value in *imgid* indicates the image the system wants slot information for and the value in *slotid* indicates which slot of that image the system is interested in. If the device driver does not recognize the value in either or *imgid* or *slotid*, then it should return an error.

The *ddi_ufm_slot_t* structure passed in *usp* is opaque. To fill in information about the image the driver should call the functions described in ddi_ufm_slot(9F).

The driver should call the ddi_ufm_slot_set_version(9F) function to indicate the version of the UFM. The version is a device-specific character string. It should contain the current version of the UFM as a human can understand it and it should try to match the format used by device vendor.

The ddi_ufm_slot_set_attrs(9F) function should be used to set the attributes of the UFM slot. These attributes include the following enumeration values:

DDI_UFM_ATTR_READABLE
: This attribute indicates that the firmware image in the specified slot may be read, even if the device driver does not currently support such functionality.

DDI_UFM_ATTR_WRITEABLE
: This attributes indicates that the firmware image in the specified slot may be updated, even if the driver does not currently support such functionality.

The ddi_ufm_set_primary(9F) function should be used to indicate whether the specified slot is the primary slot. The primary slot is the one whose firmware revision is actively being used. If there is only one slot, then there is no need to call this function, it is always considered the primary.

Finally, if there are any device-specific key-value pairs that form useful, ancillary data, then the driver should assemble an nvlist and pass it to the ddi_ufm_set_misc(9F) function.

Once the driver has finished setting all of the information about the slot then the driver should return **0**. Otherwise, the driver should return the appropriate error number. For a full list of error numbers, see Intro(2). Common values are:

EINVAL	The image or slot indicated by *imgid* and *slotid* is unknown.
EIO	An error occurred talking to the device while trying to fill out firmware slot information.
ENOMEM	The driver was unable to allocate memory while filling out slot information.

Caching and Updates

The system will fetch firmware and slot information on an as-needed basis. Once it obtains some information, it may end up caching this information on behalf of the driver. Whenever the driver believes that something could have changed -- it need know that it has -- then the driver must call ddi_ufm_update(9F).

ioctl Integration

The UFM device driver interface will set properties on the device nodes to indicate that it supports various UFM operations. When the system sees this, then it will know that the driver supports various UFM ioctls. To facilitate this, there are two different functions that a driver can use in its ioctl(9E) routine.

The first routine is ddi_ufm_is_ioctl(9F) which determines whether or not a specified command is a UFM ioctl. If the command is a UFM ioctl, then the driver must call into the UFM subsystem to handle it. Otherwise, it can proceed to handle the ioctl normally.

To handle a UFM ioctl, a driver should call ddi_ufm_ioctl(9F) with all of the arguments that it received from the ioctl. Note, that this function will completely handle the ioctl and then return. The driver does not have to perform any individual checks or operations. The UFM subsystem will automatically handle any and all needed credentials checks. The driver should simply return the value that is returned by ddi_ufm_ioctl(9F).

Locking

All UFM operations on a single UFM handle will always be run serially. However, the device driver may still need to apply adequate locking to its structure members as other may be accessing the same data structure or trying to communicate with the device.

The driver must not hold any locks while calling either ddi_ufm_is_ioctl(9F) or ddi_ufm_ioctl(9F). A call to handle an ioctl may result in calling into a driver’s UFM entry points from another thread.

Unregistering from the UFM subsystem

When a device driver is detached, it should unregister from the UFM subsystem. To do so, the driver should call ddi_ufm_fini(9F). By the time this function returns, the driver is guaranteed that no UFM entry points will be called. However, if there are outstanding UFM related activity, the function will block until it is terminated.

CONTEXT

The various UFM entry points that a device driver must implement will always be called from **kernel** context.

SEE ALSO

Intro(2), attach(9E), ioctl(9E), ddi_ufm_fini(9F), ddi_ufm_image(9F), ddi_ufm_image_set_desc(9F), ddi_ufm_image_set_misc(9F), ddi_ufm_init(9F), ddi_ufm_ioctl(9F), ddi_ufm_is_ioctl(9F), ddi_ufm_set_misc(9F), ddi_ufm_set_primary(9F), ddi_ufm_slot(9F), ddi_ufm_slot_set_attrs(9F), ddi_ufm_slot_set_version(9F), ddi_ufm_update(9F)