NAME

mlDIIntro – introduction to ML di/dd interface.

DISCUSSION

This man page provides an introduction to the interface between the device independent (DI) and device dependent (DD) portions of the ML. This is intended for use by developers of ML modules to support new hardware. These functions are not intended for use by application developers (see **mlIntro**(3dm) for an introduction to the application programming interface).

FUNCTIONS EXPECTED BY THE DI/DD INTERFACE

The ML device-independent layer will interact with your device-dependent module by using three functional entry points. These functions must be provided by all ML modules:

ddInterrogate(3dm)
ddConnect(3dm)
ddDisconnect(3dm)

ddInterrogate(3dm) is called at initialization time for each device-independent module. It is an appropriate time for your module to interrogate the hardware and expose capabilities to the di layer.

ddConnect(3dm) is called the first time each process wishes to interact with a device advertised by your module. In that function, you must construct and return a function pointer table. The ML device-independent layer will use that table to map device-independent API calls into your module's device-dependent functions.

ddDisconnect(3dm) is called at exit time for each process which called **ddConnect**(3dm).

FUNCTIONS PROVIDED BY THE DI/DD INTERFACE

These functions are provided by the ML for use within your module.

New Physical Device

Your device-dependent module must use this function to advise the di layer of supported physical devices. This should be done inside your module's **ddInterrogate** routine. (That routine is called at initialization time by the DI layer). It should be called once for each physical device you wish to expose. (If two copies of the same device are installed, you must call this routine twice.)

mlDINewPhysicalDevice(3dm)

Utility functions

These utility functions provide a means to manipulate and interpret the id numbers which the ML system uses as tokens to uniquely identify system objects. Each token should be considered opaque and module should make use of these functions to create or decompose them.

mlDIextractIdType(3dm)
mlDIextractJackIndex(3dm)
mlDIextractPathIndex(3dm)
mlDIextractXcodeEngineIndex(3dm)
mlDIextractXcodePipeIndex(3dm)

mlDImakeJackId(3dm)
mlDImakePathId(3dm)
mlDImakeXcodeEngineId(3dm)
mlDImakeXcodePipeId(3dm)

mlDIparentIdOfDeviceId(3dm)
mlDIparentIdOfLogDevId(3dm)

mlDIisOpenId(3dm)
mlDIparseOpenOptions(3dm)
mlDIconvertOpenIdToStaticId(3dm)

Queue Functions

These queue functions are used to create, and interact with, the queue between the application and your device. These are user-level functions, but equivalent functionality may also be available via kernel-level calls in some implementations.

mlDIQueueCreate(3dm)
mlDIQueueDestroy(3dm)
mlDIQueuePushMessage(3dm)
mlDIQueueSendDeviceEvent(3dm)
mlDIQueueNextMessage(3dm)
mlDIQueueUpdateMessage(3dm)
mlDIQueueAdvanceMessages(3dm)
mlDIQueueAbortMessages(3dm)
mlDIQueueReturnEvent(3dm)
mlDIQueueReceiveMessage(3dm)
mlDIQueueGetSendCount(3dm)
mlDIQueueGetReceiveCount(3dm)
mlDIQueueGetReplyCount(3dm)
mlDIQueueGetEventCount(3dm)
mlDIQueueGetSendWaitable(3dm)
mlDIQueueGetReceiveWaitable(3dm)
mlDIQueueGetDeviceWaitable(3dm)

Message Functions

These routines are useful for interacting with messages on the queues between the application and your device. Unlike the previous functions, these routines are useful outside the di/dd interface. Hence they are part of the public ML API.

mlPvFind(3dm)
mlPvSizes(3dm)
mlPvCopy(3dm)

SEE ALSO

mlIntro(3dm), ddInterrogate(3dm), ddConnect(3dm), mlDINew(3dm), mlDIQueue(3dm), mlPvSizes(3dm), mlPvFind(3dm), mlPvCopy(3dm)

NAME

mlDINew: mlDINewPhysicalDevice – used to register ML devices.

SYNOPSIS

#include <ML/ml_didd.h>

MLstatus MLAPI mlDINewPhysicalDevice(mlDISysCtxt mlDISysCtxt,
mlOSDriverCtxt pDriver,
const MLbyte* ddCookie,
MLint32 ddCookieSize);

PARAMETER

mlDISysContext Pointer to opaque structure used by the DI layer to store information related to this particular system. This was passed to the mlmodule in the **ddInterrogate**(3dm) call.

pDriver Pointer to opaque structure used by the DI layer to store information related to this particular driver. This was passed to the mlmodule in the **ddInterrogate**(3dm) call.

ddCookie Pointer to an array of information which your module would like stored by the device-independent layer. That information will be available to your module via the ddConnect(3dm) call. Note that information stored in the cookie will later be accessed by a different process, in a different address space, so it must only contain process-independent information.

ddCookieSize The number of bytes in the ddCookie array.

DESCRIPTION

This routine should be called from within a device-dependent module's **ddInterrogate**(3dm) routine. It is used to describe each physical ML device which the calling module wishes to advertise through the ML interface.

When **ddInterrogate**(3dm) is called, your module is passed opaque references to the system and your module. For each installed device, call **mlDINewPhysicalDevice**, passing it those references, along with the device location and a module-dependent "cookie". Later, whenever the device-independent layer needs to access that physical device, it will call your module's ddConnect(3dm) routine, passing it that private module-dependent cookie.

It is acceptable for a single module to call **mlDINewPhysicalDevice** multiple times (indicating that it supports more than one physical device).

DIAGNOSTICS

This function returns one of the following:

ML_STATUS_NO_ERROR
The object was successfully opened.

ML_STATUS_OUT_OF_MEMORY
There is insufficient memory to perform the operation

SEE ALSO

mlDIIntro(3dm), ddInterrogate(3dm), ddConnect(3dm).

NAME

mlDIQueue: mlDIQueueCreate, mlDIQueueDestroy, mlDIQueuePushMessage, mlDIQueueNextMessage, mlDIQueueUpdateMessage, mlDIQueueAdvanceMessage, mlDIQueueAbortMessages, mlDIQueueReturnEvent, mlDIQueueReceiveMessage, mlDIQueueGetSendCount, mlDIQueueGetReceiveCount, mlDIQueueGetReplyCount, mlDIQueueGetEventCount, mlDIQueueGetSendWaitable, mlDIQueueGetReceiveWaitable, mlDIQueueGetDeviceWaitable – queue functions used by device-dependent ML modules.

SYNOPSIS

#include <ML/ml_didd.h>

```
MLstatus MLAPI mlDIQueueSize(MLqueueOptions* pOpt, MLint32 *retSize);

MLstatus MLAPI mlDIQueueCreate(MLbyte* preallocSpace,
                    MLint32 preallocSize,
                    MLqueueOptions* pOpt,
                              MLqueueRec** pQueue);

MLstatus MLAPI mlDIQueueDestroy(MLqueueRec* pQueue);

MLstatus MLAPI mlDIQueuePushMessage(MLqueueRec* pQueue,
                                    MLint32 messageType,
                    MLpv* message,
                    MLbyte* ddData,
                    MLint32 ddDataSize,
                    MLint32 ddDataAlignment);

MLstatus MLAPI mlDIQueueSendDeviceEvent(MLqueueRec* pQueue,
                                    MLint32 messageType);

MLstatus MLAPI mlDIQueueNextMessage(MLqueueRec* pQueue,
                    MLqueueEntryRec** pEntry
                    MLint32 *messageType,
                              MLpv** message,
                    MLbyte** ddData,
                    MLint32* ddDataSize);

MLstatus MLAPI mlDIQueueUpdateMessage(MLqueueRec* pQueue,
                      MLqueueEntryRec* pEntry
                                MLint32 newMessageType);

MLstatus MLAPI mlDIQueueAdvanceMessages(MLqueueRec* pQueue);

MLstatus MLAPI mlDIQueueAbortMessages(MLqueueRec* pQueue);

MLstatus MLAPI mlDIQueueReturnEvent(MLqueueRec* pQueue,
                              MLint32 messageType,
                                    MLpv* message);

MLstatus MLAPI mlDIQueueReceiveMessage(MLqueueRec* pQueue,
                                       MLint32 messageType,
                                       MLpv** message,
                                       MLbyte** ddData,
                          MLint32* ddDataSize);
```

MLint32 MLAPI mlDIQueueGetSendCount(MLqueueRec* pQueue);

MLint32 MLAPI mlDIQueueGetReceiveCount(MLqueueRec* pQueue);

MLint32 MLAPI mlDIQueueGetReplyCount(MLqueueRec* pQueue);

MLint32 MLAPI mlDIQueueGetEventCount(MLqueueRec* pQueue);

MLwaitable MLAPI mlIDQueueGetSendWaitable(MLqueueRec* pQueue);

MLwaitable MLAPI mlIDQueueGetReceiveWaitable(MLqueueRec* pQueue);

MLwaitable MLAPI mlIDQueueGetDeviceWaitable(MLqueueRec* pQueue);

PARAMETER

pOpts	Options from opening a new device. These include queue creation options.
size	The size of the entire queue area (in bytes). This is useful, if you wish to allocate your own memory for the queues.
preallocatedSpace	A pointer to preallocated space, this may be NULL if you wish the DI layer to allocate memory for the queue. If is is not null, then you must have previously allocated memory (use mlDIQueueComputeSize to determine how much). That memory must be aligned on an 8-byte boundary.
pQueue	Pointer to opaque structure used by the DI layer to store information related to this particular queue.
messageType	The type of the message, defined in mldefs.h
message	A ML message (array of MLpv's, where the last entry is ML_END).
ddData	Pointer to device-dependent data to be appended to the message. May be NULL if there is no device-dependent data.
ddDataSize	Size of device-dependent data in bytes. Must be 0 if there is no device-dependent data and non-zero otherwise.
ddDataAlignment	Alignment requirements for device-dependent data. A value of 1 indicates no alignment constraints, a value of 4 indicates 4-byte alignment, a value of 8, indicates 8-byte alignment is required.
pQueue	Pointer to opaque structure used by the DI layer to store information related to this particular queue entry.
newMessageType	This will be used to overwrite the existing messageType (the existing type was specified when it was first enqueued, this new type will be visible when the receiveMessage call is made).

DESCRIPTION

These routines should be called from within a module's device-dependent routines. (Pointers to those routines were passed to the DI layer by the module's **ddConnect**(3dm) routine.)

mlDIQueueCreate creates a ML queue for communication between an application and a device. That queue includes a send fifo (for message headers going to the device), a receive fifo (for message headers going back to the application), a message payload area (to hold the contents of messages in-flight) and an event area (to hold the contents of events in-flight). If the preallocatedSpace pointer is NULL, the memory for the queue is allocated automatically. If you prefer to allocate your own memory, then first call **mlDIQueueComputeSize** to find how much space is needed, allocate that much memory, and then call **mlDIQueueCreate**, passing in a pointer to your preallocated memory.

Note: preallocated space is not currently supported. The preallocatedSpace pointer should always be set to NULL.

mlDIQueueDestroy destroys the queue and frees any allocated resources. If memory was allocated in the **mlDIQueueCreate**, then it is released during this call. Note that if you manually allocated memory for the queue (passing in a non-NULL pointer to **mlDIQueueCreate**), then you are responsible for freeing that memory after **mlDIQueueDestroy** returns.

mlDIQueuePushMessage pushes a message from the application onto the queue (placing the body of the message in the message payload, and a header on the send fifo). In can optionally append device-dependent data with specified size and alignment. Use this in your sendControls and sendBuffers routines.

mlDIQueueSendDeviceEvent pushes a private device-dependent message onto the queue to the device. This event is consumed as the device advances the queue, and is not visible to the application. It is useful for private communication within your device-dependent module.

mlDIQueueNextMessage returns pointers to the oldest unread message on the application's send fifo. When your device is ready to work on a new message from the queue, call this function to look at the message (and any device-dependent data appended to it). This function has a side-effect of marking the message read, so the next call to **mlDIQueueNextMessage**, will return a pointer to the following message. It is acceptable to start working on later messages, before you have completed earlier ones, with the caveat that it must always appear to the application that messages were processed strictly in order.

mlDIQueueUpdateMessage marks a queue entry as having been processed and updates the message type. Use this when you're finished working on a message. Note that this does not return the completed message to the application, it merely marks it as completed so that it will be moved during a later call to **mlDIQueueAdvanceMessages**.

mlDIQueueAdvanceMessages looks at the fifo going to the device, removes any processed messages and pushes them onto the fifo going back to the application. Use this when you have processed one or more messages, and wish to return the results to the application.

mlDIQueueAbortMessages pops every remaining message header from the fifo going to the device, updates the message types to mark them as having been aborted, and pushes the result onto the fifo going back to the application. Use this when you need to abort a transfer.

mlDIQueueReturnEvent pushes an event from the device onto the queue (placing the body of the event in the event area, and a header on the application's receive fifo). Use this when, in the process of doing work, you notice an event for which the application has requested notification.

mlDIQueueReceiveMessage pops any old (previously-read) message, and returns the top (oldest) message on the application's receive queue. In addition, it returns a pointer to any device-dependent data which was appended to that message. Use this in your receiveMessage function.

mlDIQueueGetSendCount returns the number of items in the queue from application to device.

mlDIQueueGetReceiveCount returns the number of items (both replies and events) in the queue between device and application.

mlDIQueueGetReplyCount returns the number of replies in the queue between device and application.

mlDIQueueGetEventCount returns the number of events in the queue between device and application.

mlDIQueueGetSendWaitable returns a ML waitable (a file descriptor in Unix implementations). This

will fire whenever there are more than N slots free in the queue between application and device. It is used by applications which wish to enqueue messages in chunks. The setting for N is specified at open time via the ML_OPEN_SEND_SIGNAL_COUNT parameter.

mlDIQueueGetReceiveWaitable returns a waitable which will fire whenever there is an unread message on the queue from the device back to the application. Applications may wait on this, rather than polling for replies from the device.

mlDIQueueGetDeviceWaitable returns a waitable which will fire whenever there is an unread message on the queue from the application to the device. This is convenient for software devices which wish to be awaken whenever there is new data to be processed.

NOTES

The queuing code assumes a single application thread pushing and receiving messages, and a separate single device thread processing messages and generating replies. It has been carefully constructed so that no locking is necessary between reader and writer threads for that case. If you choose to have more than one application thread, or more than one device thread, then you must implement locking around all queueing functions.

The queueing functions described in this man page are user-level calls, but equivalent kernel-level calls may be available in some implementations. It is acceptable to allocate memory for the queue, map it into both user and kernel space, then access it using these functions in user space and the kernel functions in kernel space. Note that such an implementation is still subject to the threading constraint mentioned above.

SEE ALSO

mlDIIntro(3dm), ddInterrogate(3dm). ddConnect(3dm).

NAME

mlDIid: mlDIextractIdType, mlDIextractJackIndex, mlDIextractPathIndex, mlDIextractXcodeEngineIndex, mlDIextractXcodePipeIndex, mlDImakeJackId, mlDImakePathId, mlDImakeXcodeEngineId, mlDImakeXcodePipeId, mlDIparentIdOfDeviceId, mlDIparentIdOfLogDevId, mlDIisOpenId, mlDIconvertOpenIdToStaticId– id functions used by device-dependent ML modules.

SYNOPSIS

#include <ML/ml_didd.h>

MLint32 MLAPI mlDIextractIdType(MLint64 id);

MLint32 MLAPI mlDIextractJackIndex(MLint64 id);

MLint32 MLAPI mlDIextractPathIndex(MLint64 id);

MLint32 MLAPI mlDIextractXcodeEngineIndex(MLint64 id);

MLint32 MLAPI mlDIextractXcodePipeIndex(MLint64 id);

MLint64 MLAPI mlDImakeJackId(MLint64 deviceId, MLint32 jackIndex);

MLint64 MLAPI mlDImakePathId(MLint64 deviceId, MLint32 pathIndex);

MLint64 MLAPI mlDImakeXcodeEngineId(MLint64 deviceId, MLint32 xcodeEngineIndx);

MLint64 MLAPI mlDImakeXcodePipeId(MLint64 xcodeEngineId, MLint32 pipeIndex);

MLint64 MLAPI mlDIparentIdOfDeviceId(MLint64 deviceId);

MLint64 MLAPI mlDIparentIdOfLogDevId(MLint64 logDevId);

MLint32 MLAPI mlDIisOpenId(MLint64 candidateId);

MLint64 MLAPI mlDIconvertOpenIdToStaticId(MLopenid openId);

PARAMETER

id	A 64-bit ML id number.
deviceId	A 64-bit ML static id number for a ML device.
logDevId	A 64-bit ML static id number for a ML logical device.
xcodeEngineId	A 64-bit ML static id number for a ML transcoder.
jackIndex	An integer module-dependent static jack index.
pathIndex	An integer module-dependent static path index.
xcodeEngineIndex	An integer module-dependent static transcoder index.
pipeIndex	An integer module-dependent static transcode pipe index.
candidateId	A 64-bit ML id number (from either a previous call to getCapabilities, or a call to mlOpen)
openId	A 64-bit ML open id number (from a previous call to mlOpen)

DESCRIPTION

These routines should be called from within a device-dependent module's device-dependent routines. (Pointers to those routines were passed to the DI layer by the module's **ddConnect**(3dm) routine.)

Use these routines to manipulate ML id numbers in your module.

The make calls are useful for constructing id numbers when needed in the ddGetCapabilities call - they convert your module-dependent indexes into system-wide id numbers.

The extract calls are useful for interpreting id numbers passed to your device - they turn system-wide id numbers back into module-dependent indexes.

In addition there are calls to find the static id of parent devices, to distinguish open and static ids, and to convert between open and static ids.

SEE ALSO

mlDIIntro(3dm), ddInterrogate(3dm). ddConnect(3dm).

NAME

ddInterrogate – entry-point into ML device-dependent module.

SYNOPSIS

#include <ML/ml_didd.h>

MLstatus ddInterrogate(MLIsystemContext system,
MLImoduleContext module);

PARAMETER

system Opaque structure used by the DI layer to store information related to this particular system.

module Opaque structure used by the DI layer to store information related to this particular module.

DESCRIPTION

This routine is the first entry-point in every device-dependent module. It is expected that this routine should interrogate hardware on the system and then make a series of "call backs" that serve to register all the capabilities that your module intends to expose through the ML public API.

To register the capabilities of each physical device, make use of the callback function: **mlDINewPhysicalDevice**(3dm).

ddInterrogate(3dm) is passed two arguments. Both are opaque references constructed by the ML which it uses to manage the operation of this driver, its devices, and their capabilities. The opaque references given to **ddInterrogate**(3dm) should be passed in the call to **mlDINewPhysicalDevice**(3dm) without change.

The ML requires that **ddInterrogate**(3dm) register information that is independent of any process address space. Your **ddInterrogate**(3dm) function may only be called once, but the information it provides may be available to many different processes running at many different times. The ML device-independent layer coordinates this scheme, it is otherwise transparent to your driver.

ddInterrogate(3dm) must only register capabilities of the digital media device(s) it supports. It should not reserve or allocate resources. Those activities occur when **ddConnect**(3dm) is called.

In some implementations, **ddInterrogate**(3dm) may be called more than once. On each call, your module should expose all the devices it supports at that instant (by calling **mlDINewPhysicalDevice**(3dm)).

DIAGNOSTICS

This function returns one of the following:

ML_STATUS_NO_ERROR
The object was successfully opened.

ML_STATUS_OUT_OF_MEMORY
There is insufficient memory to perform the operation

SEE ALSO

mlDIIntro(3dm), ddConnect(3dm), mlDINew(3dm).

NAME

ddConnect: ddDisconnect – entry-points into ML device-dependent module.

SYNOPSIS

#include <ML/ml_didd.h>

MLstatus
MLstatus ddConnect(MLbyte *ddCookie,
MLint64 staticDeviceId,
MLphysicalDeviceOps *pOps,
MLbyte retddDevPriv);**
MLstatus ddDisconnect(MLbyte* ddDevPriv);

PARAMETER

ddCookie Pointer to an array of information which your module requested stored when it called **mlDINewPhysicalDevice**(3dm). This information should be treated as read-only by your module and is only guaranteed to be valid for the duration of this function call.

staticDeviceID 64-bit ID number representing the physical device to which the calling application wishes to connect. This static id number is assigned by the device-independent layer.

pOps This should be filled in by your module to indicate device-dependent functional entry points. In addition, there is space in this structure for your module to indicate the version of the ML di/dd interface to which it was written.

retddDevPriv Pointer to your module's private process-dependent data. Use this to store information you may later need while processing application requests.

DESCRIPTION

These routines are a required entry-points in every device-dependent mlmodule.

ddConnect(3dm) will be called in every process that accesses a physical device exposed by your device-dependent module. **ddConnect**(3dm) establishes the connection between the process independent description of a physical device and process specific addresses for entry points and private memory structures. It does this by means of the process-independent cookie that your module gave when it registered each new physical device.

ddDisconnect(3dm) is called at exit time for each process which called **ddConnect**(3dm). It is a convenient place for you to free any memory allocated during the connect call (thereby eliminating a reported memory leak which debugging tools may flag). Note that **ddDisconnect**(3dm) is not guaranteed to be called if an application terminates abnormally.

The **ddConnect**(3dm) routine should construct and return a table of function pointers to device-dependent routines for the specified logical device. That table should include pointers for the following functions:

MLstatus (*ddGetCapabilities)(MLbyte* ddDevPriv,
MLint64 staticObjectId,
MLpv capabilities);**

MLstatus (*ddPvGetCapabilities)(MLbyte* ddDevPriv,
MLint64 staticObjectId,
MLint64 paramId,
MLpv capabilities);**

MLstatus (*ddOpen)(MLbyte* ddDevPriv
MLint64 staticObjectId,

MLopenid openObjectId
MLpv* openOptions
MLbyte retddOpenPriv);**

MLstatus (*ddSetControls)(MLbyte* ddOpenPriv,
MLopenid openObjectId,
MLpv *controls);

MLstatus (*ddGetControls)(MLbyte* ddOpenPriv,
MLopenid openObjectId,
MLpv *controls);

MLstatus (*ddSendControls)(MLbyte* ddOpenPriv,
MLopenid openObjectId,
MLpv *controls);

MLstatus (*ddSendBuffers)(MLbyte* ddOpenPriv,
MLopenid openObjectId,
MLpv *buffers);

MLstatus (*ddReceiveMessage)(MLbyte* ddOpenPriv,
MLopenid openObjectId,
MLint32 *retMsgType,
MLpv **retReply);

MLstatus (*ddXcodeWork)(MLbyte* ddOpenPriv,
MLopenid openObjectId);

MLstatus (*ddClose)(MLbyte* ddOpenPriv,
MLopenid openObjectId);

DIAGNOSTICS

This function returns one of the following:

ML_STATUS_NO_ERROR
: The object was successfully opened.

ML_STATUS_OUT_OF_MEMORY
: There is insufficient memory to perform the operation

ML_STATUS_DEVICE_UNAVAILABLE
: The physical device is not available. Note that this error should only be returned if the actual hardware is unavailable (for example, if it was unplugged). If the device is merely busy, that should not preclude connecting to it.

SEE ALSO

mlDIIntro(3dm), ddInterrogate(3dm), mlDIQueue(3dm).