NAME

mac, **GLDv3** - MAC networking device driver overview

SYNOPSIS

#include <sys/mac_provider.h>
#include <sys/mac_ether.h>

INTERFACE LEVEL

illumos DDI specific

DESCRIPTION

The **MAC** framework provides a means for implementing high-performance networking device drivers. It is the successor to the GLD interfaces and is sometimes referred to as the GLDv3. The remainder of this manual introduces the aspects of writing devices drivers that leverage the MAC framework. While both the GLDv3 and MAC framework refer to the same thing, in this manual page we use the term the *MAC framework* to refer to the device driver interface.

MAC device drivers are character devices. They define the standard _init(9E), _fini(9E), and _info(9E) entry points to initialize the module, as well as dev_ops(9S) and cb_ops(9S) structures.

The main interface with MAC is through a series of callbacks defined in a mac_callbacks(9S) structure. These callbacks control all the aspects of the device. They range from sending data, getting and setting of properties, controlling mac address filters, and also managing promiscuous mode.

The MAC framework takes care of many aspects of the device driver's management. A device that uses the MAC framework does not have to worry about creating device nodes or implementing open(9E) or close(9E) routines. In addition, all of the work to interact with dlpi(7P) is taken care of automatically and transparently.

High-Level Design

At a high-level a device driver is chiefly concerned with two different operations:

1. Sending frames

2. Receiving frames

When sending frames, the MAC framework always calls functions registered in the mac_callbacks(9S) structure to have the driver transmit frames on hardware. When receiving frames, the driver will generally receive an interrupt which will cause it to check for incoming data and deliver it to the MAC framework.

Configuration of a device such as whether auto-negotiation should be enabled, the speeds that the device supports, the MTU (maximum transmission unit), and the generation of pause frames are all driven by properties. The functions to get, set, and obtain information about properties are defined through callback functions specified in the mac_callbacks(9S) structure. The full list of properties and a description of the relevant callbacks can be found in the *PROPERTIES* section.

The MAC framework is designed to take advantage of various modern features provided by hardware, such as checksumming, segmentation offload, and hardware filtering. The MAC framework assumes none of these advanced features are present and allows device drivers to negotiate them through a capability system. Drivers can declare that they support various capabilities by implementing the optional mc_getcapab(9E) entry point. Each capability has its associated entry points and structures to fill out. The capabilities are detailed in the *CAPABILITIES* section.

The following sections, describe the flow of a basic device driver. For advanced device drivers, the flow is generally the same. The primary distinction is in how frames are sent and received.

Initializing MAC Support

For a device to be used by the MAC framework, it must register with the framework and take specific actions during _init(9E), attach(9E), detach(9E), and _fini(9E).

All device drivers have to define a dev_ops(9S) structure which is pointed to by a modldrv(9S) structure and the corresponding NULL-terminated modlinkage(9S) structure. The dev_ops(9S) structure should have a cb_ops(9S) structure defined for it; however, it does not need to implement any of the standard cb_ops(9S) entry points.

Normally, in a driver's _init(9E) entry point, it passes its **modlinkage** structure directly to mod_install(9F). To properly register with MAC, the driver must call mac_init_ops(9F) before it calls mod_install(9F). If for some reason the mod_install(9F) function fails, then the driver must be removed by a call to mac_fini_ops(9F).

Conversely, in the driver's _fini(9F) routine, it should call mac_fini_ops(9F) after it successfully calls mod_remove(9F). For an example of how to use the mac_init_ops(9F) and mac_fini_ops(9F) functions, see the examples section in mac_init_ops(9F).

Registering with MAC

Every instance of a device should register separately with MAC. To register with MAC, a driver must allocate a mac_register(9S) structure, fill it in, and then call mac_register(9F). The **mac_register_t** structure contains information about the device and all of the required function pointers that will be used as callbacks by the framework.

These steps should all be taken during a device's attach(9E) entry point. It is recommended that the driver perform this sequence of steps after the device has finished its initialization of the chipset and interrupts, though interrupts should not be enabled at that point. After it calls mac_register(9F) it will start receiving callbacks from the MAC framework.

To allocate the registration structure, the driver should call mac_alloc(9F). Device drivers should generally always pass the symbol MAC_VERSION as the argument to mac_alloc(9F). Upon successful completion, the driver will receive a **mac_register_t** structure which it should fill in. The structure and its members are documented in mac_register(9S).

The mac_callbacks(9S) structure is not allocated as a part of the mac_register(9S) structure. In general, device drivers declare this statically. See the *MAC Callbacks* section for more information on how to fill it out.

Once the structure has been filled in, the driver should call mac_register(9F) to register itself with MAC. The handle that it uses to register with should be part of the driver's soft state. It will be used in various other support functions and callbacks.

If the call is successful, then the device driver should enable interrupts and finish any other initialization required. If the call to mac_register(9F) failed, then it should unwind its initialization and should return **DDI_FAILURE** from its attach(9E) routine.

MAC Callbacks

The MAC framework interacts with a device driver through a series of callbacks. These callbacks are described in their individual manual pages and the collection of callbacks is indicated in the mac_callbacks(9S) manual page. This section does not focus on the specific functions, but rather on interactions between them and the rest of the device driver framework.

A device driver should make no assumptions about when the various callbacks will be called and whether or not they will be called simultaneously. For example, a device driver may be asked to transmit data through a call to its mc_tx(9F) entry point while it is being asked to get a device property through a call to its mc_getprop(9F) entry point. As such, while some calls may be serialized to the device, such as setting properties, the device driver should always presume that all of its data needs to be protected with locks. While the device is holding locks, it is safe for it call the following MAC routines:

- mac_hcksum_get(9F)
- mac_hcksum_set(9F)
- mac_lso_get(9F)
- mac_maxsdu_update(9F)
- mac_prop_info_set_default_link_flowctrl(9F)
- mac_prop_info_set_default_str(9F)

- mac_prop_info_set_default_uint8(9F)
- mac_prop_info_set_default_uint32(9F)
- mac_prop_info_set_default_uint64(9F)
- mac_prop_info_set_perm(9F)
- mac_prop_info_set_range_uint32(9F)

Any other MAC related routines should not be called with locks held, such as mac_link_update(9F) or mac_rx(9F). Other routines in the DDI may be called while locks are held; however, device driver writers should be careful about calling blocking routines while locks are held or in interrupt context, even when it is legal to do so.

Receiving Data

A device driver will often receive data through the means of an interrupt or by being asked to poll for frames. When this occurs, zero or more frames, each with optional metadata, may be ready for the device driver to consume. Often each frame has a corresponding descriptor which has information about whether or not there were errors or whether or not the device successfully checksummed the packet.

During a single interrupt or poll request, a device driver should process a fixed number of frames. For each frame the device driver should:

1. Ensure that all of the DMA memory for the region is synchronized with the ddi_dma_sync(9F) function and the handle checked for errors if the device driver has enabled DMA error reporting as part of the Fault Management Architecture (FMA). If the driver does not rely on DMA, then it may skip this step. It is recommended that this is performed once per interrupt or poll for the entire region and not on a per-packet basis.

2. First check whether or not the frame has errors. If errors were detected, then the frame should not be sent to the operating system. It is recommended that devices keep kstats (see kstat_create(9S) for more information) and bump the counter whenever such an error is detected. If the device distinguishes between the types of errors, then separate kstats for each class of error are recommended. See the *STATISTICS* section for more information on the various error cases that should be considered.

3. Once the frame has been determined to be valid, the device driver should transform the frame into a mblk(9S). See the section *MBLKS AND DMA* for more information on how to transform and prepare a message block.

4. If the device supports hardware checksumming (see the *CAPABILITIES* section for more information on checksumming), then the device driver should set the corresponding checksumming information with a call to mac_hcksum_set(9F).

5. It should then append this new message block to the *end* of the message block chain, linking it to the **b_next** pointer. It is vitally important that all the frames be chained in the order that they were received. If the device driver mistakenly reorders frames, then it may cause performance impacts in the TCP stack and potentially impact application correctness.

Once all the frames have been processed and assembled, the device driver should deliver them to the rest of the operating system by calling mac_rx(9F). The device driver should try to give as many mblk_t structures to the system at once. It *should not* call mac_rx(9F) once for every assembled mblk_t.

The device driver must not hold any locks across the call to mac_rx(9F). When this function is called, received data will be pushed through the networking stack and some replies may be generated and given to the driver to send out.

It is not the device driver's responsibility to determine whether or not the system can keep up with a driver's delivery rate of frames. The rest of the networking stack will handle issues related to keeping up appropriately and ensure that kernel memory is not exhausted by packets that are not being processed.

If the device driver has negotiated the MAC_CAPAB_RINGS capability (discussed in mac_capab_rings(9E)) then it should call mac_rx_ring(9F) and not mac_rx(9F). A given interrupt may correspond to more than one ring that needs to be checked. In those cases, the driver should follow the above procedure independently for each ring. That means it will call mac_rx_ring(9F) once for each ring. When it is looking at the rings, the driver will need to make sure that the ring has not had interrupts disabled and is undergoing polling. This is discussed in greater detail in the mac_capab_rings(9E) and mri_poll(9E) manual pages.

Finally, the device driver should make sure that any other housekeeping activities required for the ring are taken care of such that more data can be received.

Transmitting Data and Back Pressure

A device driver will be asked to transmit a message block chain by having it's mc_tx(9E) entry point called. While the driver is processing the message blocks, it may run out of resources. For example, a transmit descriptor ring may become full. At that point, the device driver should return the remaining unprocessed frames. The act of returning frames indicates that the device has asserted flow control. Once this has been done, no additional calls will be made to the driver's transmit entry point and the back pressure will be propagated throughout the rest of the networking stack.

At some point in the future when resources have become available again, for example after an interrupt indicating that some portion of the transmit ring has been sent, then the device driver must notify the system that it can continue transmission. To do this, the driver should call mac_tx_update(9F). After

that point, the driver will receive calls to its mc_tx(9E) entry point again. As mentioned in the section on callbacks, the device driver should avoid holding any particular locks across the call to mac_tx_update(9F).

Interrupt Coalescing

For devices operating at higher data rates, interrupt coalescing is an important part of a well functioning device and may impact the performance of the device. Not all devices support interrupt coalescing. If interrupt coalescing is supported on the device, it is recommended that device driver writers provide private properties for their device to control the interrupt coalescing rate. This will make it much easier to perform experiments and observe the impact of different interrupt rates on the rest of the system.

MAC Address Filter Management

The MAC framework will attempt to use as many MAC address filters as a device has. To program a multicast address filter, the driver's mc_multicst(9E) entry point will be called. If the device driver runs out of filters, it should not take any special action and just return the appropriate error as documented in the corresponding manual pages for the entry points. The framework will ensure that the device is placed in promiscuous mode if it needs to.

If the hardware supports more than one unicast filters then the device driver should consider implementing the MAC_CAPAB_RINGS capability, which exposes a means for those multiple unicast MAC address filters to be used by the broader system. See mac_capab_rings(9E) for more information.

Receive Side Scaling

Receive side scaling is where a hardware device supports multiple, independent queues of frames that can be received. Each of these queues is generally associated with an independent interrupt and the hardware usually performs some form of hash across the queues. Hardware which supports this, should look at implementing the MAC_CAPAB_RINGS capability and see mac_capab_rings(9E) for more information.

Link Updates

It is the responsibility of the device driver to keep track of the data link's state. Many devices provide a means of receiving an interrupt when the state of the link changes. When such a change happens, the driver should update its internal data structures and then call mac_link_update(9F) to inform the MAC layer that this has occurred. If the device driver does not properly inform the system about link changes, then various features like link aggregations and other mechanisms that leverage the link state will not work correctly.

Link Speed and Auto-negotiation

Many networking devices support more than one possible speed that they can operate at. The selection of a speed is often performed through *auto-negotiation*, though some devices allow the user to control

what speeds are advertised and used.

Logically, there are two different sets of things that the device driver needs to keep track of while it's operating:

1. The supported speeds in hardware.

2. The enabled speeds from the user.

By default, when a link first comes up, the device driver should generally configure the link to support the common set of speeds and perform auto-negotiation.

A user can control what speeds a device advertises via auto-negotiation and whether or not it performs auto-negotiation at all by using a series of properties that have **_EN_** in the name. These are read/write properties and there is one for each speed supported in the operating system. For a full list of them, see the *PROPERTIES* section.

In addition to these properties, there is a corresponding set of properties with **_ADV_** in the name. These are similar to the **_EN_** family of properties, but they are read-only and indicate what the device has actually negotiated. While they are generally similar to the **_EN_** family of properties, they may change depending on power settings. See the **Ethernet Link Properties** section in dladm(1M) for more information.

It's worth discussing how these different values get used throughout the different entry points. The first entry point to consider is the mc_propinfo(9E) entry point. For a given speed, the driver should consult whether or not the hardware supports this speed. If it does, it should fill in the default value that the hardware takes and whether or not the property is writable. The properties should also be updated to indicate whether or not it is writable. This holds for both the **_EN_** and **_ADV_** family of properties.

The next entry point is mc_getprop(9E). Here, the device should first consult whether the given speed is supported. If it is not, then the driver should return ENOTSUP. If it does, then it should return the current value of the property.

The last property endpoint is the mc_setprop(9E) entry point. Here, the same logic applies. Before the driver considers whether or not the property is writable, it should first check whether or not it's a supported property. If it's not, then it should return ENOTSUP. Otherwise, it should proceed to check whether the property is writable, and if it is and a valid value, then it should update the property and restart the link's negotiation.

Finally, there is the mc_getstat(9E) entry point. Several of the statistics that are queried relate to auto-

negotiation and hardware capabilities. When a statistic relates to the hardware supporting a given speed, the **_EN_** properties should be ignored. The only thing that should be consulted is what the hardware itself supports. Otherwise, the statistics should look at what is currently being advertised by the device.

Unregistering from MAC

During a driver's detach(9E) routine, it should unregister the device instance from MAC by calling mac_unregister(9F) on the handle that it originally called it on. If the call to mac_unregister(9F) failed, then the device is likely still in use and the driver should fail the call to detach(9E).

Interacting with Devices

Administrators always interact with devices through the dladm(1M) command line interface. The state of devices such as whether the link is considered **up** or **down**, various link properties such as the **MTU**, **auto-negotiation** state, and **flow control** state, are all exposed. It is also the preferred way that these properties are set and configured.

While device tunables may be presented in a driver.conf(4) file, it is recommended instead to expose such things through dladm(1M) private properties, whether explicitly documented or not.

CAPABILITIES

Capabilities in the MAC Framework are optional features that a device supports which indicate various hardware features that the device supports. The two current capabilities that the system supports are related to being able to hardware perform large send offloads (LSO), often also known as TCP segmentation and the ability for hardware to calculate and verify the checksums present in IPv4, IPV6, and protocol headers such as TCP and UDP.

The MAC framework will query a device for support of a capability through the mc_getcapab(9E) function. Each capability has its own constant and may have corresponding data that goes along with it and a specific structure that the device is required to fill in. Note, the set of capabilities changes over time and there are also private capabilities in the system. Several of the capabilities are used in the implementation of the MAC framework. Others represent features that have not been stabilized and thus both API and binary compatibility for them is not guaranteed. It is important that the device driver handles unknown capabilities correctly. For more information, see mc_getcapab(9E).

The following capabilities are stable and defined in the system:

MAC_CAPAB_HCKSUM

The MAC_CAPAB_HCKSUM capability indicates to the system that the device driver supports some amount of checksumming. The specific data for this capability is a pointer to a **uint32_t**. To indicate no support for any kind of checksumming, the driver should either set this value to zero or simply return that it doesn't support the capability.

Note, the values that the driver declares in this capability indicate what it can do when it transmits data. If the driver can only verify checksums when receiving data, then it should not indicate that it supports this capability. The following set of flags may be combined through a bitwise inclusive OR:

HCKSUM_INET_PARTIAL
: This indicates that the hardware can calculate a partial checksum for both IPv4 and IPv6; however, it requires the pseudo-header checksum be calculated for it. The pseudo-header checksum will be available for the mblk_t when calling mac_hcksum_get(9F). Note this does not imply that the hardware is capable of calculating the IPv4 header checksum. That should be indicated with the HCKSUM_IPHDRCKSUM flag.

HCKSUM_INET_FULL_V4
: This indicates that the hardware will fully calculate the L4 checksum for outgoing IPv4 packets and does not require a pseudo-header checksum. Note this does not imply that the hardware is capable of calculating the IPv4 header checksum. That should be indicated with the HCKSUM_IPHDRCKSUM.

HCKSUM_INET_FULL_V6
: This indicates that the hardware will fully calculate the L4 checksum for outgoing IPv6 packets and does not require a pseudo-header checksum.

HCKSUM_IPHDRCKSUM
: This indicates that the hardware supports calculating the checksum for the IPv4 header itself.

When in a driver's transmit function, the driver will be processing a single frame. It should call mac_hcksum_get(9F) to see what checksum flags are set on it. Note that the flags that are set on it are different from the ones described above and are documented in its manual page. These flags indicate how the driver is expected to program the hardware and what checksumming is required. Not all frames will require hardware checksumming or will ask the hardware to checksum it.

If a driver supports offloading the receive checksum and verification, it should check to see what the hardware indicated was verified. The driver should then call mac_hcksum_set(9F). The flags used are different from the ones above and are discussed in detail in the mac_hcksum_set(9F) manual page. If there is no checksum information available or the driver does not support checksumming, then it should simply not call mac_hcksum_set(9F).

Note that the checksum flags should be set on the first mblk_t that makes up a given message. In other words, if multiple mblk_t structures are linked together by the **b_cont** member to describe a single frame, then it should only be called on the first mblk_t of that set. However, each distinct message

should have the checksum bits set on it, if applicable. In other words, each mblk_t that is linked together by the **b_next** pointer may have checksum flags set.

It is recommended that device drivers provide a private property or driver.conf(4) property to control whether or not checksumming is enabled for both rx and tx; however, the default disposition is recommended to be enabled for both. This way if hardware bugs are found in the checksumming implementation, they can be disabled without requiring software updates. The transmit property should be checked when determining how to reply to mc_getcapab(9E) and the receive property should be checked in the context of the receive function.

MAC_CAPAB_LSO

The MAC_CAPAB_LSO capability indicates that the driver supports various forms of large send offload (LSO). The private data is a pointer to a *mac_capab_lso_t* structure. At the moment, LSO support is limited to TCP inside of IPv4. This structure has the following members which are used to indicate various types of LSO support.

```
t_uscalar_t             lso_flags;
lso_basic_tcp_ivr4_t    lso_basic_tcp_ipv4;
```

The *lso_flags* member is used to indicate which members are valid and should be considered. Each flag represents a different form of LSO. The member should be set to the bitwise inclusive OR of the following values:

LSO_TX_BASIC_TCP_IPV4
: This indicates hardware support for performing TCP segmentation offloading over IPv4. When this flag is set, the *lso_basic_tcp_ipv4* member must be filled in.

The *lso_basic_tcp_ipv4* member is a structure with the following members:

```
t_uscalar_t     lso_max
```

The *lso_max* member should be set to the maximum size of the TCP data payload that can be offloaded to the hardware.

Like with checksumming, it is recommended that driver writers provide a means for disabling the support of LSO even if it is enabled by default. This deals with the case where issues that pop up for LSO may be worked around without requiring additional driver work.

The following capabilities are still evolving in the operating system. They are documented such that

device driver writers may experiment with them; however, if such drivers are not present inside the core operating system repository, they may be subject to API and ABI breakage.

MAC_CAPAB_RINGS

The **MAC_CAPAB_RINGS** capability is very important for implementing a high-performing device driver. Netowrking hardware structures the queues of packets to be sent and received into a ring. Each entry in this ring, has a descriptor, which describes the address and options for a packet which is going to be transmitted or received. While simple networking devices only have a single ring, many high-speed networking devices have support for many rings.

Rings are used for two important purposes. The first is receive side scaling (RSS), which is the ability to have the hardware hash the contents of a packet based on some of the protocol headers, and send it to one of several rings. These different rings may each have their own interrupt associated with them, allowing the card to receive traffic in parallel. Similar logic can be performed when sending traffic, to leverage multiple hardware resources, increasing capacity.

The second use of rings is to group them together and apply filtering rules. For example, if a packet matches a specific VLAN or MAC address, then it can be sent to a specific ring or a specific group of rings.

From the MAC framework's perspective, a driver can have one or more groups. A group consists of the following:

- One or more hardware rings.
- One or more MAC address or VLAN filters.

The details around how a device driver changes when rings are employed, the data structures that a driver must implement, and more are in a separate manual page. Please see mac_capab_rings(9E) for more information.

PROPERTIES

Properties in the MAC framework represent aspects of a link. These include things like the link's current state and MTU. Many of the properties in the system are focused around auto-negotiation and controlling what link speeds are advertised. Information about properties is covered by three different device entry points. The mc_propinfo(9E) entry point obtains metadata about the property. The mc_getprop(9E) entry point obtains the property. The mc_setprop(9E) entry point updates the property to a new value.

Many of the properties listed below are read-only. Each property indicates whether it's read-only or it's

read/write. However, driver writers may not implement the ability to set all writable properties. Many of these depend on the card itself. In particular, all properties that relate to auto-negotiation and are read/write may not be updated if the hardware in question does not support toggling what link speeds are auto-negotiated. While copper Ethernet often does not have this restriction, it often exists with various fiber standards and phys.

The following properties are the subset of MAC framework properties that driver writers should be aware of and handle. While other properties exist in the system, driver writers should always return an error when a property not listed below is encountered. See mc_getprop(9E) and mc_setprop(9E) for more information on how to handle them.

MAC_PROP_DUPLEX

Type: **link_duplex_t** | Permissions: **Read-Only**

The **MAC_PROP_DUPLEX** property is used to indicate whether or not the link is duplex. A duplex link may have traffic flowing in both directions at the same time. The **link_duplex_t** is an enumeration which may be set to any of the following values:

LINK_DUPLEX_UNKNOWN

The current state of the link is unknown. This may be because the link has not negotiated to a specific speed or it is down.

LINK_DUPLEX_HALF

The link is running at half duplex. Communication may travel in only one direction on the link at a given time.

LINK_DUPLEX_FULL

The link is running at full duplex. Communication may travel in both directions on the link simultaneously.

MAC_PROP_SPEED

Type: **uint64_t** | Permissions: **Read-Only**

The **MAC_PROP_SPEED** property stores the current link speed in bits per second. A link that is running at 100 MBit/s would store the value 100000000ULL. A link that is running at 40 Gbit/s would store the value 40000000000ULL.

MAC_PROP_STATUS

Type: **link_state_t** | Permissions: **Read-Only**

The **MAC_PROP_STATUS** property is used to indicate the current state of the link. It indicates whether the link is up or down. The **link_state_t** is an enumeration which may be set to any of the following values:

LINK_STATE_UNKNOWN
: The current state of the link is unknown. This may be because the driver's mc_start(9E) endpoint has not been called so it has not attempted to start the link.

LINK_STATE_DOWN
: The link is down. This may be because of a negotiation problem, a cable problem, or some other device specific issue.

LINK_STATE_UP
: The link is up. If auto-negotiation is in use, it should have completed. Traffic should be able to flow over the link, barring other issues.

MAC_PROP_AUTONEG

Type: **uint8_t** | Permissions: **Read/Write**

The **MAC_PROP_AUTONEG** property indicates whether or not the device is currently configured to perform auto-negotiation. A value of **0** indicates that auto-negotiation is disabled. A **non-zero** value indicates that auto-negotiation is enabled. Devices should generally default to enabling auto-negotiation.

When getting this property, the device driver should return the current state. When setting this property, if the device supports operating in the requested mode, then the device driver should reset the link to negotiate to the new speed after updating any internal registers.

MAC_PROP_MTU

Type: **uint32_t** | Permissions: **Read/Write**

The **MAC_PROP_MTU** property determines the maximum transmission unit (MTU). This indicates the maximum size packet that the device can transmit, ignoring its own headers. For an Ethernet device, this would exclude the size of the Ethernet header and any VLAN headers that would be placed. It is up to the driver to ensure that any MTU values that it accepts when adding in its margin and header sizes does not exceed its maximum frame size.

By default, drivers for Ethernet should initialize this value and the MTU to **1500**. When getting this property, the driver should return its current recorded MTU. When setting this property, the driver should first validate that it is within the device's valid range and then it must call

mac_maxsdu_update(9F). Note that the call may fail. If the call completes successfully, the driver should update the hardware with the new value of the MTU and perform any other work needed to handle it.

If the device does not support changing the MTU after the device's mc_start(9E) entry point has been called, then driver writers should return EBUSY.

MAC_PROP_FLOWCTRL

Type: **link_flowctrl_t** | Permissions: **Read/Write**

The **MAC_PROP_FLOWCTRL** property manages the configuration of pause frames as part of Ethernet flow control. Note, this only describes what this device will advertise. What is actually enabled may be different and is subject to the rules of auto-negotiation. The **link_flowctrl_t** is an enumeration that may be set to one of the following values:

LINK_FLOWCTRL_NONE

Flow control is disabled. No pause frames should be generated or honored.

LINK_FLOWCTRL_RX

The device can receive pause frames; however, it should not generate them.

LINK_FLOWCTRL_TX

The device can generate pause frames; however, it does not support receiving them.

LINK_FLOWCTRL_BI

The device supports both sending and receiving pause frames.

When getting this property, the device driver should return the way that it has configured the device, not what the device has actually negotiated. When setting the property, it should update the hardware and allow the link to potentially perform auto-negotiation again.

The remaining properties are all about various auto-negotiation link speeds. They fall into two different buckets: properties with **_ADV_** in the name and properties with **_EN_** in the name. For any given supported speed, there is one of each. The **_EN_** set of properties are read/write properties that control what should be advertised by the device. When these are retrieved, they should return the current value of the property. When they are set, they should change how the hardware advertises the specific speed and trigger any kind of link reset and auto-negotiation, if enabled, to occur.

The **_ADV_** set of properties are read-only properties. They are meant to reflect what has actually been negotiated. These may be different from the **_EN_** family of properties, especially when different power

management settings are at play.

See the *Link Speed and Auto-negotiation* section for more information.

The properties are ordered in increasing link speed:

MAC_PROP_ADV_10HDX_CAP
Type: **uint8_t** | Permissions: **Read-Only**

The **MAC_PROP_ADV_10HDX_CAP** property describes whether or not 10 Mbit/s half-duplex support is advertised.

MAC_PROP_EN_10HDX_CAP
Type: **uint8_t** | Permissions: **Read/Write**

The **MAC_PROP_EN_10HDX_CAP** property describes whether or not 10 Mbit/s half-duplex support is enabled.

MAC_PROP_ADV_10FDX_CAP
Type: **uint8_t** | Permissions: **Read-Only**

The **MAC_PROP_ADV_10FDX_CAP** property describes whether or not 10 Mbit/s full-duplex support is advertised.

MAC_PROP_EN_10FDX_CAP
Type: **uint8_t** | Permissions: **Read/Write**

The **MAC_PROP_EN_10FDX_CAP** property describes whether or not 10 Mbit/s full-duplex support is enabled.

MAC_PROP_ADV_100HDX_CAP
Type: **uint8_t** | Permissions: **Read-Only**

The **MAC_PROP_ADV_100HDX_CAP** property describes whether or not 100 Mbit/s half-duplex support is advertised.

MAC_PROP_EN_100HDX_CAP
Type: **uint8_t** | Permissions: **Read/Write**

The **MAC_PROP_EN_100HDX_CAP** property describes whether or not 100 Mbit/s half-

duplex support is enabled.

MAC_PROP_ADV_100FDX_CAP
Type: **uint8_t** | Permissions: **Read-Only**

The **MAC_PROP_ADV_100FDX_CAP** property describes whether or not 100 Mbit/s full-duplex support is advertised.

MAC_PROP_EN_100FDX_CAP
Type: **uint8_t** | Permissions: **Read/Write**

The **MAC_PROP_EN_100FDX_CAP** property describes whether or not 100 Mbit/s full-duplex support is enabled.

MAC_PROP_ADV_100T4_CAP
Type: **uint8_t** | Permissions: **Read-Only**

The **MAC_PROP_ADV_100T4_CAP** property describes whether or not 100 Mbit/s Ethernet using the 100BASE-T4 standard is advertised.

MAC_PROP_EN_100T4_CAP
Type: **uint8_t** | Permissions: **Read/Write**

The **MAC_PROP_ADV_100T4_CAP** property describes whether or not 100 Mbit/s Ethernet using the 100BASE-T4 standard is enabled.

MAC_PROP_ADV_1000HDX_CAP
Type: **uint8_t** | Permissions: **Read-Only**

The **MAC_PROP_ADV_1000HDX_CAP** property describes whether or not 1 Gbit/s half-duplex support is advertised.

MAC_PROP_EN_1000HDX_CAP
Type: **uint8_t** | Permissions: **Read/Write**

The **MAC_PROP_EN_1000HDX_CAP** property describes whether or not 1 Gbit/s half-duplex support is enabled.

MAC_PROP_ADV_1000FDX_CAP
Type: **uint8_t** | Permissions: **Read-Only**

The **MAC_PROP_ADV_1000FDX_CAP** property describes whether or not 1 Gbit/s full-duplex support is advertised.

MAC_PROP_EN_1000FDX_CAP
Type: **uint8_t** | Permissions: **Read/Write**

The **MAC_PROP_EN_1000FDX_CAP** property describes whether or not 1 Gbit/s full-duplex support is enabled.

MAC_PROP_ADV_2500FDX_CAP
Type: **uint8_t** | Permissions: **Read-Only**

The **MAC_PROP_ADV_2500FDX_CAP** property describes whether or not 2.5 Gbit/s full-duplex support is advertised.

MAC_PROP_EN_2500FDX_CAP
Type: **uint8_t** | Permissions: **Read/Write**

The **MAC_PROP_EN_2500FDX_CAP** property describes whether or not 2.5 Gbit/s full-duplex support is enabled.

MAC_PROP_ADV_5000FDX_CAP
Type: **uint8_t** | Permissions: **Read-Only**

The **MAC_PROP_ADV_5000FDX_CAP** property describes whether or not 5.0 Gbit/s full-duplex support is advertised.

MAC_PROP_EN_5000FDX_CAP
Type: **uint8_t** | Permissions: **Read/Write**

The **MAC_PROP_EN_5000FDX_CAP** property describes whether or not 5.0 Gbit/s full-duplex support is enabled.

MAC_PROP_ADV_10GFDX_CAP
Type: **uint8_t** | Permissions: **Read-Only**

The **MAC_PROP_ADV_10GFDX_CAP** property describes whether or not 10 Gbit/s full-duplex support is advertised.

MAC_PROP_EN_10GFDX_CAP

Type: **uint8_t** | Permissions: **Read/Write**

The **MAC_PROP_EN_10GFDX_CAP** property describes whether or not 10 Gbit/s full-duplex support is enabled.

MAC_PROP_ADV_40GFDX_CAP

Type: **uint8_t** | Permissions: **Read-Only**

The **MAC_PROP_ADV_40GFDX_CAP** property describes whether or not 40 Gbit/s full-duplex support is advertised.

MAC_PROP_EN_40GFDX_CAP

Type: **uint8_t** | Permissions: **Read/Write**

The **MAC_PROP_EN_40GFDX_CAP** property describes whether or not 40 Gbit/s full-duplex support is enabled.

MAC_PROP_ADV_100GFDX_CAP

Type: **uint8_t** | Permissions: **Read-Only**

The **MAC_PROP_ADV_100GFDX_CAP** property describes whether or not 100 Gbit/s full-duplex support is advertised.

MAC_PROP_EN_100GFDX_CAP

Type: **uint8_t** | Permissions: **Read/Write**

The **MAC_PROP_EN_100GFDX_CAP** property describes whether or not 100 Gbit/s full-duplex support is enabled.

Private Properties

In addition to the defined properties above, drivers are allowed to define private properties. These private properties are device-specific properties. All private properties share the same constant, **MAC_PROP_PRIVATE**. Properties are distinguished by a name, which is a character string. The list of such private properties is defined when registering with mac in the **m_priv_props** member of the mac_register(9S) structure.

The driver may define whatever semantics it wants for these private properties. They will not be listed when running dladm(1M), unless explicitly requested by name. All such properties should start with a leading underscore character and then consist of alphanumeric ASCII characters and additional underscores or hyphens.

Properties of type **MAC_PROP_PRIVATE** may show up in all three property related entry points: mc_propinfo(9E), mc_getprop(9E), and mc_setprop(9E). Device drivers should tell the different properties apart by using the strcmp(9F) function to compare it to the set of properties that it knows about. When encountering properties that it doesn't know, it should treat them like all other unknown properties.

STATISTICS

The MAC framework defines a couple different sets of statistics which are based on various standards for devices to implement. Statistics are retrieved through the mc_getstat(9E) entry point. There are both statistics that are required for all devices and then there is a separate set of Ethernet specific statistics. Not all devices will support every statistic. In many cases, several device registers will need to be combined to create the proper stat.

In general, if the device is not keeping track of these statistics, then it is recommended that the driver store these values as a **uint64_t** to ensure that overflow does not occur.

If a device does not support a specific statistic, then it is fine to return that it is not supported. The same should be used for unrecognized statistics. See mc_getstat(9E) for more information on the proper way to handle these.

General Device Statistics

The following statistics are based on MIB-II statistics from both RFC 1213 and RFC 1573.

MAC_STAT_IFSPEED

The device's current speed in bits per second.

MAC_STAT_MULTIRCV

The total number of received multicast packets.

MAC_STAT_BRDCSTRCV

The total number of received broadcast packets.

MAC_STAT_MULTIXMT

The total number of transmitted multicast packets.

MAC_STAT_BRDCSTXMT

The total number of received broadcast packets.

MAC_STAT_NORCVBUF

The total number of packets discarded by the hardware due to a lack of receive buffers.

MAC_STAT_IERRORS

The total number of errors detected on input.

MAC_STAT_UNKNOWNS

The total number of received packets that were discarded because they were of an unknown protocol.

MAC_STAT_NOXMTBUF

The total number of outgoing packets dropped due to a lack of transmit buffers.

MAC_STAT_OERRORS

The total number of outgoing packets that resulted in errors.

MAC_STAT_COLLISIONS

Total number of collisions encountered by the transmitter.

MAC_STAT_RBYTES

The total number of **bytes** received by the device, regardless of packet type.

MAC_STAT_IPACKETS

The total number of **packets** received by the device, regardless of packet type.

MAC_STAT_OBYTES

The total number of **bytes** transmitted by the device, regardless of packet type.

MAC_STAT_OPACKETS

The total number of **packets** sent by the device, regardless of packet type.

MAC_STAT_UNDERFLOWS

The total number of packets that were smaller than the minimum sized packet for the device and were therefore dropped.

MAC_STAT_OVERFLOWS

The total number of packets that were larger than the maximum sized packet for the device and were therefore dropped.

Ethernet Specific Statistics

The following statistics are specific to Ethernet devices. They refer to values from RFC 1643 and include various MII/GMII specific stats. Many of these are also defined in IEEE 802.3.

ETHER_STAT_ADV_CAP_1000FDX
Indicates that the device is advertising support for 1 Gbit/s full-duplex operation.

ETHER_STAT_ADV_CAP_1000HDX
Indicates that the device is advertising support for 1 Gbit/s half-duplex operation.

ETHER_STAT_ADV_CAP_100FDX
Indicates that the device is advertising support for 100 Mbit/s full-duplex operation.

ETHER_STAT_ADV_CAP_100GFDX
Indicates that the device is advertising support for 100 Gbit/s full-duplex operation.

ETHER_STAT_ADV_CAP_100HDX
Indicates that the device is advertising support for 100 Mbit/s half-duplex operation.

ETHER_STAT_ADV_CAP_100T4
Indicates that the device is advertising support for 100 Mbit/s 100BASE-T4 operation.

ETHER_STAT_ADV_CAP_10FDX
Indicates that the device is advertising support for 10 Mbit/s full-duplex operation.

ETHER_STAT_ADV_CAP_10GFDX
Indicates that the device is advertising support for 10 Gbit/s full-duplex operation.

ETHER_STAT_ADV_CAP_10HDX
Indicates that the device is advertising support for 10 Mbit/s half-duplex operation.

ETHER_STAT_ADV_CAP_2500FDX
Indicates that the device is advertising support for 2.5 Gbit/s full-duplex operation.

ETHER_STAT_ADV_CAP_40GFDX
Indicates that the device is advertising support for 40 Gbit/s full-duplex operation.

ETHER_STAT_ADV_CAP_5000FDX
Indicates that the device is advertising support for 5.0 Gbit/s full-duplex operation.

ETHER_STAT_ADV_CAP_ASMPAUSE
Indicates that the device is advertising support for receiving pause frames.

ETHER_STAT_ADV_CAP_AUTONEG

Indicates that the device is advertising support for auto-negotiation.

ETHER_STAT_ADV_CAP_PAUSE
Indicates that the device is advertising support for generating pause frames.

ETHER_STAT_ADV_REMFAULT
Indicates that the device is advertising support for detecting faults in the remote link peer.

ETHER_STAT_ALIGN_ERRORS
Indicates the number of times an alignment error was generated by the Ethernet device. This is a count of packets that were not an integral number of octets and failed the FCS check.

ETHER_STAT_CAP_1000FDX
Indicates the device supports 1 Gbit/s full-duplex operation.

ETHER_STAT_CAP_1000HDX
Indicates the device supports 1 Gbit/s half-duplex operation.

ETHER_STAT_CAP_100FDX
Indicates the device supports 100 Mbit/s full-duplex operation.

ETHER_STAT_CAP_100GFDX
Indicates the device supports 100 Gbit/s full-duplex operation.

ETHER_STAT_CAP_100HDX
Indicates the device supports 100 Mbit/s half-duplex operation.

ETHER_STAT_CAP_100T4
Indicates the device supports 100 Mbit/s 100BASE-T4 operation.

ETHER_STAT_CAP_10FDX
Indicates the device supports 10 Mbit/s full-duplex operation.

ETHER_STAT_CAP_10GFDX
Indicates the device supports 10 Gbit/s full-duplex operation.

ETHER_STAT_CAP_10HDX
Indicates the device supports 10 Mbit/s half-duplex operation.

ETHER_STAT_CAP_2500FDX

Indicates the device supports 2.5 Gbit/s full-duplex operation.

ETHER_STAT_CAP_40GFDX
Indicates the device supports 40 Gbit/s full-duplex operation.

ETHER_STAT_CAP_5000FDX
Indicates the device supports 5.0 Gbit/s full-duplex operation.

ETHER_STAT_CAP_ASMPAUSE
Indicates that the device supports the ability to receive pause frames.

ETHER_STAT_CAP_AUTONEG
Indicates that the device supports the ability to perform link auto-negotiation.

ETHER_STAT_CAP_PAUSE
Indicates that the device supports the ability to transmit pause frames.

ETHER_STAT_CAP_REMFAULT
Indicates that the device supports the ability of detecting a remote fault in a link peer.

ETHER_STAT_CARRIER_ERRORS
Indicates the number of times that the Ethernet carrier sense condition was lost or not asserted.

ETHER_STAT_DEFER_XMTS
Indicates the number of frames for which the device was unable to transmit the frame due to being busy and had to try again.

ETHER_STAT_EX_COLLISIONS
Indicates the number of frames that failed to send due to an excessive number of collisions.

ETHER_STAT_FCS_ERRORS
Indicates the number of times that a frame check sequence failed.

ETHER_STAT_FIRST_COLLISIONS
Indicates the number of times that a frame was eventually transmitted successfully, but only after a single collision.

ETHER_STAT_JABBER_ERRORS
Indicates the number of frames that were received that were both larger than the maximum packet size and failed the frame check sequence.

ETHER_STAT_LINK_ASMPAUSE

Indicates whether the link is currently configured to accept pause frames.

ETHER_STAT_LINK_AUTONEG

Indicates whether the current link state is a result of auto-negotiation.

ETHER_STAT_LINK_DUPLEX

Indicates the current duplex state of the link. The values used here should be the same as documented for **MAC_PROP_DUPLEX**.

ETHER_STAT_LINK_PAUSE

Indicates whether the link is currently configured to generate pause frames.

ETHER_STAT_LP_CAP_1000FDX

Indicates the remote device supports 1 Gbit/s full-duplex operation.

ETHER_STAT_LP_CAP_1000HDX

Indicates the remote device supports 1 Gbit/s half-duplex operation.

ETHER_STAT_LP_CAP_100FDX

Indicates the remote device supports 100 Mbit/s full-duplex operation.

ETHER_STAT_LP_CAP_100GFDX

Indicates the remote device supports 100 Gbit/s full-duplex operation.

ETHER_STAT_LP_CAP_100HDX

Indicates the remote device supports 100 Mbit/s half-duplex operation.

ETHER_STAT_LP_CAP_100T4

Indicates the remote device supports 100 Mbit/s 100BASE-T4 operation.

ETHER_STAT_LP_CAP_10FDX

Indicates the remote device supports 10 Mbit/s full-duplex operation.

ETHER_STAT_LP_CAP_10GFDX

Indicates the remote device supports 10 Gbit/s full-duplex operation.

ETHER_STAT_LP_CAP_10HDX

Indicates the remote device supports 10 Mbit/s half-duplex operation.

ETHER_STAT_LP_CAP_2500FDX

Indicates the remote device supports 2.5 Gbit/s full-duplex operation.

ETHER_STAT_LP_CAP_40GFDX

Indicates the remote device supports 40 Gbit/s full-duplex operation.

ETHER_STAT_LP_CAP_5000FDX

Indicates the remote device supports 5.0 Gbit/s full-duplex operation.

ETHER_STAT_LP_CAP_ASMPAUSE

Indicates that the remote device supports the ability to receive pause frames.

ETHER_STAT_LP_CAP_AUTONEG

Indicates that the remote device supports the ability to perform link auto-negotiation.

ETHER_STAT_LP_CAP_PAUSE

Indicates that the remote device supports the ability to transmit pause frames.

ETHER_STAT_LP_CAP_REMFAULT

Indicates that the remote device supports the ability of detecting a remote fault in a link peer.

ETHER_STAT_MACRCV_ERRORS

Indicates the number of times that the internal MAC layer encountered an error when attempting to receive and process a frame.

ETHER_STAT_MACXMT_ERRORS

Indicates the number of times that the internal MAC layer encountered an error when attempting to process and transmit a frame.

ETHER_STAT_MULTI_COLLISIONS

Indicates the number of times that a frame was eventually transmitted successfully, but only after more than one collision.

ETHER_STAT_SQE_ERRORS

Indicates the number of times that an SQE error occurred. The specific conditions for this error are documented in IEEE 802.3.

ETHER_STAT_TOOLONG_ERRORS

Indicates the number of frames that were received that were longer than the maximum frame size supported by the device.

ETHER_STAT_TOOSHORT_ERRORS

Indicates the number of frames that were received that were shorter than the minimum frame size supported by the device.

ETHER_STAT_TX_LATE_COLLISIONS

Indicates the number of times a collision was detected late on the device.

ETHER_STAT_XCVR_ADDR

Indicates the address of the MII/GMII receiver address.

ETHER_STAT_XCVR_ID

Indicates the id of the MII/GMII receiver address.

ETHER_STAT_XCVR_INUSE

Indicates what kind of receiver is in use. The following values may be used:

XCVR_UNDEFINED

The receiver type is undefined by the hardware.

XCVR_NONE

There is no receiver in use by the hardware.

XCVR_10

The receiver supports 10BASE-T operation.

XCVR_100T4

The receiver supports 100BASE-T4 operation.

XCVR_100X

The receiver supports 100BASE-TX operation.

XCVR_100T2

The receiver supports 100BASE-T2 operation.

XCVR_1000X

The receiver supports 1000BASE-X operation. This is used for all fiber receivers.

XCVR_1000T

The receiver supports 1000BASE-T operation. This is used for all copper receivers.

Device Specific kstats

In addition to the defined statistics above, if the device driver maintains additional statistics or the device provides additional statistics, it should create its own kstats through the kstat_create(9F) function to allow operators to observe them.

TX STALL DETECTION, DEVICE RESETS, AND FAULT MANAGEMENT

Device drivers are the first line of defense for dealing with broken devices and bugs in their firmware. While most devices will rarely fail, it is important that when designing and implementing the device driver that particular attention is paid in the design with respect to RAS (Reliability, Availability, and Serviceability). While everything described in this section is optional, it is highly recommended that all new device drivers follow these guidelines.

The Fault Management Architecture (FMA) provides facilities for detecting and reporting various classes of defects and faults. Specifically for networking device drivers, issues that should be detected and reported include:

- Device internal uncorrectable errors
- Device internal correctable errors
- PCI and PCI Express transport errors
- Device temperature alarms
- Device transmission stalls
- Device communication timeouts
- High invalid interrupts

All such errors fall into three primary categories:

1. Errors detected by the Fault Management Architecture
2. Errors detected by the device and indicated to the device driver
3. Errors detected by the device driver

Fault Management Setup and Teardown

Drivers should initialize support for the fault management framework by calling ddi_fm_init(9F) from

their attach(9E) routine. By registering with the fault management framework, a device driver is given the chance to detect and notice transport errors as well as report other errors that exist. While a device driver does not need to indicate that it is capable of all such capabilities described in ddi_fm_init(9F), we suggest that device drivers at least register the **DDI_FM_EREPORT_CAPABLE** so as to allow the driver to report issues that it detects.

If the driver registers with the fault management framework during its attach(9E) entry point, it must call ddi_fm_fini(9E) during its detach(9E) entry point.

Transport Errors

Many modern networking devices leverage PCI or PCI Express. As such, there are two primary ways that device drivers access data: they either memory map device registers and use routines like ddi_get8(9F) and ddi_put8(9F) or they use direct memory access (DMA). New device drivers should always enable checking of the transport layer by marking their support in the ddi_device_acc_attr_t(9S) structure and using routines like ddi_fm_acc_err_get(9F) and ddi_fm_dma_err_get(9F) to detect if errors have occurred.

Device Indicated Errors

Many devices have capabilities to announce to a device driver that a fatal correctable error or uncorrectable error has occurred. Other devices have the ability to indicate that various physical issues have occurred such as a fan failing or a temperature sensor having fired.

Drivers should wire themselves to receive notifications when these events occur. The means and capabilities will vary from device to device. For example, some devices will generate information about these notifications through special interrupts. Other devices may have a register that software can poll. In the cases where polling is required, driver writers should try not to poll too frequently and should generally only poll when the device is actively being used, e.g. between calls to the mc_start(9E) and mc_stop(9E) entry points.

Driver Transmit Stall Detection

One of the primary responsibilities of a hardened device driver is to perform transmit stall detection. The core idea behind tx stall detection is that the driver should record when it’s getting activity related to when data has been successfully transmitted. Most devices should be transmitting data on a regular basis as long as the link is up. If it is not, then this may indicate that the device is stuck and needs to be reset. At this time, the MAC framework does not provide any resources for performing these checks; however, polling on each individual transmit ring for the last completion time while something is actively being transmitted through the use of routines such as timeout(9F) may be a reasonable starting point.

Driver Command Timeout Detection

Each device is programmed in different ways. Some devices are programmed through asynchronous commands while others are programmed by writing directly to memory mapped registers. If a device receives asynchronous replies to commands, then the device driver should set reasonable timeouts for all such commands and plan on detecting them. If a timeout occurs, the driver should presume that there is an issue with the hardware and proceed to abort the command or reset the device.

Many devices do not have such a communication mechanism. However, whenever there is some activity where the device driver must wait, then it should be prepared for the fact that the device may never get back to it and react appropriately by performing some kind of device reset.

Reacting to Errors

When any of the above categories of errors has been triggered, the behavior that the device driver should take depends on the kind of error. If a fatal error, for example, a transport error, a transmit stall was detected, or the device indicated an uncorrectable error was detected, then it is important that the driver take the following steps:

1. Set a flag in the device driver's state that indicates that it has hit an error condition. When this error condition flag is asserted, transmitted packets should be accepted and dropped and actions that would require writing to the device state should fail with an error. This flag should remain until the device has been successfully restarted.

2. If the error was not a transport error that was indicated by the fault management architecture, e.g. a transport error that was detected, then the device driver should post an **ereport** indicating what has occurred with the ddi_fm_ereport_post(9F) function.

3. The device driver should indicate that the device's service was lost with a call to ddi_fm_service_impact(9F) using the symbol **DDI_SERVICE_LOST**.

4. At this point the device driver should issue a device reset through some device-specific means.

5. When the device reset has been completed, then the device driver should restore all of the programmed state to the device. This includes things like the current MTU, advertised auto-negotiation speeds, MAC address filters, and more.

6. Finally, when service has been restored, the device driver should call ddi_fm_service_impact(9F) using the symbol **DDI_SERVICE_RESTORED**.

When a non-fatal error occurs, then the device driver should submit an ereport and should optionally mark the device degraded using ddi_fm_service_impact(9F) with the **DDI_SERVICE_DEGRADED**

value depending on the nature of the problem that has occurred.

Device drivers should never make the decision to remove a device from service based on errors that have occurred nor should they panic the system. Rather, the device driver should always try to notify the operating system with various ereports and allow its policy decisions to occur. The decision to retire a device lies in the hands of the fault management architecture. It knows more about the operator's intent and the surrounding system's state than the device driver itself does and it will make the call to offline and retire the device if it is required.

Device Resets

When resetting a device, a device driver must exercise caution. If a device driver has not been written to plan for a device reset, then it may not correctly restore the device's state after such a reset. Such state should be stored in the instance's private state data as the MAC framework does not know about device resets and will not inform the device again about the expected, programmed state.

One wrinkle with device resets is that many networking cards show up as multiple PCI functions on a single device, for example, each port may show up as a separate function and thus have a separate instance of the device driver attached. When resetting a function, device driver writers should carefully read the device programming manuals and verify whether or not a reset impacts only the stalled function or if it impacts all function across the device.

If the only way to reset a given function is through the device, then this may require more coordination and work on the part of the device driver to ensure that all the other instances are correctly restored. In cases where this occurs, some devices offer ways of injecting interrupts onto those other functions to notify them that this is occurring.

MBLKS AND DMA

The networking stack manages framed data through the use of the mblk(9S) structure. The mblk allows for a single message to be made up of individual blocks. Each part is linked together through its **b_cont** member. However, it also allows for multiple messages to be chained together through the use of the **b_next** member. While the networking stack works with these structures, device drivers generally work with DMA regions. There are two different strategies that device drivers use for handling these two different cases: copying and binding.

Copying Data

The first way that device drivers handle interfacing between the two is by having two separate regions of memory. One part is memory which has been allocated for DMA through a call to ddi_dma_alloc(9F) and the other is memory associated with the memory block.

In this case, a driver will use bcopy(9F) to copy memory between the two distinct regions. When

transmitting a packet, it will copy the memory from the mblk_t to the DMA region. When receiving memory, it will allocate a mblk_t through the allocb(9F) routine, copy the memory across with bcopy(9F), and then increment the mblk_t's **w_ptr** structure.

If, when receiving, memory is not available for a new message block, then the frame should be skipped and effectively dropped. A kstat should be bumped when such an occasion occurs.

Binding Data

An alternative approach to copying data is to use DMA binding. When using DMA binding, the OS takes care of mapping between DMA memory and normal device memory. The exact process is a bit different between transmit and receive.

When transmitting a device driver has an mblk_t and needs to call the ddi_dma_addr_bind_handle(9F) function to bind it to an already existing DMA handle. At that point, it will receive various DMA cookies that it can use to obtain the addresses to program the device with for transmitting data. Once the transmit is done, the driver must then make sure to call freemsg(9F) to release the data. It must not call freemsg(9F) before it receives an interrupt from the device indicating that the data has been transmitted, otherwise it risks sending arbitrary kernel memory.

When receiving data, the device can perform a similar operation. First, it must bind the DMA memory into the kernel's virtual memory address space through a call to the ddi_dma_addr_bind_handle(9F) function if it has not already. Once it has, it must then call desballoc(9F) to try and create a new mblk_t which leverages the associated memory. It can then pass that mblk_t up to the stack.

Considerations

When deciding which of these options to use, there are many different considerations that must be made. The answer as to whether to bind memory or to copy data is not always simpler.

The first thing to remember is that DMA resources may be finite on a given platform. Consider the case of receiving data. A device driver that binds one of its receive descriptors may not get it back for quite some time as it may be used by the kernel until an application actually consumes it. Device drivers that try to bind memory for receive, often work with the constraint that they must be able to replace that DMA memory with another DMA descriptor. If they were not replaced, then eventually the device would not be able to receive additional data into the ring.

On the other hand, particularly for larger frames, copying every packet from one buffer to another can be a source of additional latency and memory waste in the system. For larger copies, the cost of copying may dwarf any potential cost of performing DMA binding.

For device driver authors that are unsure of what to do, they should first employ the copying method to

simplify the act of writing the device driver. The copying method is simpler and also allows the device driver author not to worry about allocated DMA memory that is still outstanding when it is asked to unload.

If device driver writers are worried about the cost, it is recommended to make the decision as to whether or not to copy or bind DMA data a separate private property for both transmitting and receiving. That private property should indicate the size of the received frame at which to switch from one format to the other. This way, data can be gathered to determine what the impact of each method is on a given platform.

SEE ALSO

dladm(1M), driver.conf(4), ieee802.3(5), dlpi(7P), _fini(9E), _info(9E), _init(9E), attach(9E), close(9E), detach(9E), mac_capab_rings(9E), mc_close(9E), mc_getcapab(9E), mc_getprop(9E), mc_getstat(9E), mc_multicst(9E), mc_open(9E), mc_propinfo(9E), mc_setpromisc(9E), mc_setprop(9E), mc_start(9E), mc_stop(9E), mc_tx(9E), mc_unicst(9E), open(9E), allocb(9F), bcopy(9F), ddi_dma_addr_bind_handle(9F), ddi_dma_alloc(9F), ddi_fm_acc_err_get(9F), ddi_fm_dma_err_get(9F), ddi_fm_ereport_post(9F), ddi_fm_fini(9F), ddi_fm_init(9F), ddi_fm_service_impact(9F), ddi_get8(9F), ddi_put8(9F), desballoc(9F), freemsg(9F), kstat_create(9F), mac_alloc(9F), mac_fini_ops(9F), mac_hcksum_get(9F), mac_hcksum_set(9F), mac_init_ops(9F), mac_link_update(9F), mac_lso_get(9F), mac_maxsdu_update(9F), mac_prop_info_set_default_link_flowctrl(9F), mac_prop_info_set_default_str(9F), mac_prop_info_set_default_uint32(9F), mac_prop_info_set_default_uint64(9F), mac_prop_info_set_default_uint8(9F), mac_prop_info_set_perm(9F), mac_prop_info_set_range_uint32(9F), mac_register(9F), mac_rx(9F), mac_unregister(9F), mc_getprop(9F), mc_tx(9F), mod_install(9F), mod_remove(9F), strcmp(9F), timeout(9F), cb_ops(9S), ddi_device_acc_attr_t(9S), dev_ops(9S), kstat_create(9S), mac_callbacks(9S), mac_register(9S), mblk(9S), modldrv(9S), modlinkage(9S)

McCloghrie, K. and Rose, M., *RFC 1213 Management Information Base for Network Management of, TCP/IP-based internets: MIB-II,* March 1991.

McCloghrie, K. and Kastenholz, F., *RFC 1573 Evolution of the Interfaces Group of MIB-II*, January 1994.

Kastenholz, F., *RFC 1643 Definitions of Managed Objects for the Ethernet-like, Interface Types*.

NAME

mac_capab_rings - MAC ring capability

SYNOPSIS

#include <sys/mac_provider.h>

typedef struct mac_capab_rings_s mac_capab_rings_t;

INTERFACE LEVEL

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

DESCRIPTION

The **MAC_CAPAB_RINGS** capability provides a means to for device drivers to take advantage of the additional resources offered by hardware. There are two primary concepts that this MAC capability relies on: rings and groups.

The *ring* is a logical construct, usually DMA memory, that maps to something that exists in hardware. The ring consists of a number of entries. Each entry in the ring is a descriptor which describes the location in memory of a packet to send or receive and attributes about that packet. These entries are then arranged in a fixed-size circular buffer called a ring that is generally shared between the operating system and the hardware with DMA-backed memory. Most NICs, regardless of their support for this capability, are using something like a ring under the hood. Some vendors may also call the ring a *queue*.

A collection of one or more rings is called a *group*. Each group usually has a collection of filters that can be associated with them. These filters are usually defined in terms of matching something like a MAC address, VLAN, or Ethertype, though more complex filters may exist in hardware. When a packet matches a filter, it will then be directed to those rings.

In the MAC framework, rings and groups are separated into categories based on their purpose: transmitting and receiving. While the MAC framework thinks of transmit and receive rings as different physical constructs, they may map to the same underlying resources in the hardware. The device driver may implement the MAC_CAPAB_RINGS capability for one of transmitting, receiving, or both.

Hardware mapping to Rings, and Groups

There are many different ways that hardware may map to this capability. Consider the following examples:

1. There is hardware that exists that supports a feature commonly called receive side scaling (RSS). With RSS, the hardware has multiple rings and it hashes all incoming traffic and directs that to a different ring. Rings are associated with different interrupts, allowing multiple rings to be

processed in parallel. This would map to a device that has a single group, but multiple rings inside of that group.

2. There is hardware which may have only a single ring, but has support for multiple filters. A common example of this are 1 GbE devices. While the hardware only has one ring, it has support for multiple independent MAC address filters, each of which can be programmed with a single MAC address to receive traffic for. In this case, the driver would map that to a single group with a single ring. However, it would implement the ability to program several filters. While this may not seem useful at first, when virtual NICs are created on top of a physical NIC, the additional hardware filters will be used to avoid putting the device in promiscuous mode.

3. Finally, the third common class of device is one that has many rings, which may be placed into one of many different groups. Each group has its own filtering capabilities. In this world, the device driver would declare support for multiple groups, each of which has its own independent sets of rings.

Filters

The mac_group_info(9S) structure is used to define several different kinds of filters that the group might implement. There are three different classes of filters that exist:

MAC Address

A given frame matches a MAC Address filter if the receive address in the Ethernet Header match the specified MAC address.

VLAN A given frame matches a VLAN filter if it both has an 802.1Q VLAN tag and the specified VLAN matches the VLAN specified in the filter. If the frame's outer ethertype is not 0x8100, then the filter will not match.

MAC Address and VLAN

A given frame matches a MAC Address and VLAN filter if it matches both the specified MAC address and the specified VLAN. This is constructed as a logical and of the previous two filters. If only one of the two matches, then the frame does not match this filter.

Devices may support many different filter types. If the hardware resources are equal, drivers should prefer to implement the combined MAC Address and VLAN filter, rather than the separate filters.

The MAC framework assumes that the design of the filters when hardware is processing them follows the following rules:

1. When there are multiple filters of the same kind with different addresses, then the hardware will

accept a frame if it matches *ANY* of the specified filters. In other words, if there are two VLAN filters defined, one for VLAN 23 and one for VLAN 42, then if a frame has either VLAN 23 or VLAN 42, then it will be accepted for the group.

2. If multiple different classes of filters are defined, then the hardware should only accept a frame if it passes *ALL* of the filter classes. For example, if there is a MAC address filter and a separate VLAN filter, the hardware will only accept the frame if it passes both sets of filters.

3. If there are multiple different classes of filters and there are multiple filters present in each class, then the driver will accept a packet as long as it matches *ALL* filter classes. However, within a given filter class, it may match *ANY* of the filters.

Device drivers that support the combined MAC and VLAN exact match, should not implement support for the separate MAC and VLAN filters.

The following psuedocode summarizes the behavior for a device that supports independent MAC and VLAN filters. If the hardware only supports a single family of filters, then simply treat that in the psuedocode as though it is always true:

```
for each packet p:
   for each MAC filter m:
      if m matches p's mac:
         for each VLAN filter v:
            if v matches p's vlan:
               accept p for group
```

The following psuedocode summarizes the behavior for a device that supports a combined MAC address and VLAN filter:

```
for each packet p:
   for each filter f:
      if f.mac matches p's mac and f.vlan matches p's vlan:
         accept p for group
```

MAC Capability Structure

When the device driver's mc_getcapab(9E) function entry point is called with the capability requested set to MAC_CAPAB_RINGS, then the value of the capability structure is a pointer to a structure with the following members:

```
uint_t          mr_extensions;
```

```
uint_t          mr_flags;
mac_ring_type_t     mr_type;
mac_groupt_type_t   mr_group_type;
uint_t          mr_rnum;
uint_t          mr_gnum;
mac_get_ring_t      mr_rget;
mac_get_group_t     mr_gget;
```

If the devices supports the MAC_CAPAB_RINGS capability, then it should first check the *mr_type* member of the structure. This member has the following possible values:

MAC_RING_TYPE_RX
: Indicates that this group is for receive rings.

MAC_RING_TYPE_TX
: Indicates that this group is for transmit rings.

If neither of these values is specified, then the device driver must return B_FALSE from its mc_getcapab(9E) entry point. Once it has identified the type, it should fill in the capability structure based on the following rules:

mr_extensions The *mr_extensions* member is used to negotiate extensions between the MAC framework and the device driver. The MAC frameowrk will set the value of *mr_extensions* to include all of the currently known extensions. The driver should intersect this list with the set that the driver supports. At this time, no such features are defined and the driver should set this member to **0**.

mr_flags The *mr_flags* member is used to indicate additional capabilities of the ring capability. No such additional capabilities are defined at this time and so this member should be set to **0**.

mr_type The *mr_type* member is used to indicate whether this group is for transmit or receive rings. The *mr_type* member should not be modified by the device driver. It is set by the MAC framework when the driver's mc_getcapab(9E) entry point is called. As indicated above, the driver must check the value to determine which group this is referring to.

mr_group_type
: This member is used to indicate the group type. This should be set to MAC_GROUP_TYPE_STATIC, which indicates that there is a static mapping between which rings belong to which group. The number of rings per group may vary on the

group and can be set by the driver.

mr_rnum This indicates the total number of rings that are available from hardware. The number exposed may be less than the number supported in hardware. This is often due to receiving fewer resources such as interrupts.

mr_gnum This indicates the total number of groups that are available from hardware. The number exposed may be less than the number supported in hardware. This is often due to receiving fewer resources such as interrupts.

When working with transmit rings, this value may be zero. In this case, each ring is treated independently and separate groups for each transmit ring are not required.

mr_rget This member is a function that is responsible for filling in information about a group. Please see mr_rget(9E) for information on the function, its signature, and responsibilities.

mr_gget This member is a function that is responsible for filling in information about a group. Please see mr_gget(9E) for information on the function, its signature, and responsibilities.

DRIVER IMPLICATIONS

MAC Callback Entry Points

When a driver implements the MAC_CAPAB_RINGS capability, then it must not implement some of the traditional MAC callbacks. If the driver supports MAC_CAPAB_RINGS for receiving, then it must not implement the mc_unicst(9E) entry point. This is instead handled through the filters that were described earlier. The filter entry points are defined as part of the mac_group_info(9S) structure.

If the driver supports MAC_CAPAB_RINGS for transmitting, then it should not implement the mc_tx(9E) entry point, it will not be used. The MAC framework will instead use the mri_tx(9E) entry point that is part of the mac_ring_info(9S) structure.

Polling on rings

When the MAC_CAPAB_RINGS capability is implemented, then additional functionality for receiving becomes available. A receive ring has the ability to be polled. When the operating system desires to begin polling the ring, it will make a function call into the driver, asking it to receive packets from this ring. When receiving packets while polling, the process is generally identical to that described in the *Receiving Data* section of mac(9E). For more details, see mri_poll(9E).

When the MAC framework wants to enable polling, it will first turn off interrupts through the

mi_disable(9E) entry point on the driver. The driver must ensure that there is proper serialization between the interrupt enablement, interrupt disablement, the interrupt handler for that ring, and the mri_poll(9E) entry point. For more information on the locking, see the discussions in mri_poll(9E) and mi_disable(9E).

Updated callback functions

When using rings, two of the primary functions that were used change. First, the mac_rx(9F) function should be replaced with the mac_ring_rx(9F) function. Secondly, the mac_tx_update(9F) function should be replaced with the mac_tx_ring_update(9F) function.

Interrupt and Ring Mapping

Drivers often vary the number of rings that they expose based on the number of interrupts that exist. When a driver only supports a single group, there is often no reason to have more rings than interrupts. However, most hardware supports a means of having multiple rings tie to the same interrupt. Drivers then tie the rings in different groups to the same interrupts and therefore when an interrupt is triggered, iterate over all of the rings.

Filter Management

As part of general operation, the device driver will be asked to add various filters to groups. The MAC framework does not keep track of the assigned filters in such a way that after a device reset that they'll be given to the driver again. Therefore, it is recommended that the driver keep track of all filters it has assigned such that they can be reinstated after a device reset of some kind.

For more information, see the *TX STALL DETECTION, DEVICE RESETS, AND FAULT MANAGEMENT* section of mac(9E).

Broadcast, Multicast, and Promiscuous Mode

Rings and groups are currently designed to emphasize and enhance the receipt of filtered, unicast frames. This means that special handling is required when working with broadcast traffic, multicast traffic, and enabling promiscuous mode. This only applies to receive groups and rings.

By default, only the first group with index zero, sometimes called the default group, should ever be programmed to receive broadcast traffic. This group should always be programmed to receive broadcast traffic, the same way that the broader device is programmed to always receive broadcast traffic when the MAC_CAPAB_RINGS capability has not been negotiated.

When multicast addresses are assigned to the device through the mc_multicst(9E) entry point, those should also be assigned to the first group.

Similarly, when enabling promiscuous mode, the driver should only enable promiscuous traffic to be

received by the first group.

No other groups or rings should every receive broadcast, multicast, or promiscuous mode traffic.

SEE ALSO

mac(9E), mc_getcapab(9E), mc_multicst(9E), mc_tx(9E), mc_unicst(9E), mi_disable(9E), mr_gaddring(9E), mr_gget(9E), mr_gremring(9E), mr_rget(9E), mri_poll(9E), mac_ring_rx(9F), mac_rx(9F), mac_tx_ring_update(9F), mac_tx_update(9F), mac_group_info(9S)

NAME

mac_filter, **mgi_addmac**, **mgi_remmac**, **mgi_addvlan**, **mgi_remvlan**, **mgi_addmvf**, **mgi_remmvf** - add and remove filters from MAC groups

SYNOPSIS

#include <sys/mac_provider.h>

int
prefix_ring_add_mac(*mac_group_driver_t driver, const uint8_t *mac, uint_t flags*);

int
prefix_ring_rem_mac(*mac_group_driver_t driver, const uint8_t *mac, uint_t flags*);

int
prefix_ring_add_vlan(*mac_group_driver_t driver, uint16_t vlan, uint_t flags*);

int
prefix_ring_rem_vlan(*mac_group_driver_t driver, uint16_t vlan, uint_t flags*);

int
prefix_ring_add_macvlan(*mac_group_driver_t driver, const uint8_t *mac, uint16_t vlan, uint_t flags*);

int
prefix_ring_rem_macvlan(*mac_group_driver_t driver, const uint8_t *mac, uint16_t vlan, uint_t flags*);

INTERFACE LEVEL

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

PARAMETERS

driver A pointer to the ring's private data that was passed in via the *mgi_driver* member of the mac_group_info(9S) structure as part of the mr_gget(9E) entry point.

mac A pointer to an array of bytes that contains the unicast address to add or remove from a filter. It is guaranteed to be at least as long, in bytes, as the MAC plugin's address length. For Ethernet devices that length is six bytes, ETHERADDRL.

vlan The numeric value of the VLAN that should be added or removed from a filter.

flags This member is reserved for future use. Drivers should ensure that it is **0**.

DESCRIPTION

The **mac_filter**() family of entry points are used to add and remove various classes of filters from a device. For more information on the filters, see the *Filters* section of mac_capab_rings(9S).

The *driver* argument indicates which group the request to add or remove a filter is being requested on. The *flags* member contains values that modify the behavior of the filters. Currently this is reserved for future use. The driver must check the *flags* member. If any unknown or unsupported members values are present, then the driver should return ENOTSUP.

The filter addition operations, **mgi_addmac**(), **mgi_addvlan**(), and **mgi_addmvf**(), all instruct the system to add a filter to the specified group. The filter should not target any specific ring in the group. If multiple rings are present, then the driver should ensure that the hardware balances incoming traffic amongst all of the rings through a consistent hashing mechanism such as receive side scaling.

The **mgi_addmac**() entry point instructs the driver to add the MAC address specified in *mac* to the filter list for the group. The MAC address should always be a unicast MAC address; however, the driver is encouraged to verify that before adding it.

The **mgi_remmac**() should remove the MAC address specified in *mac* from the filter list for the group.

The **mgi_addvlan**() entry point instructs the driver to add the VLAN specified in *vlan* to the filter list for the group. The **mgi_remvlan**() entry point instructs the driver to remove the VLAN specified in *vlan* from the filter list for the group.

The **mgi_addmvl**() entry point instructs the driver to add a MAC address and VLAN tuple to the filter list for the group. This entry point should ensure that hardware only ever inserts values if it can match both the MAC address in *mac* and the VLAN in *vlan*. Importantly, it should only allow packets that match both the *mac* and *vlan*.

Stacking Filters

Multiple filters of the same class should always be treated as a logical-OR. The frame may match any of the filters in a given class to be accepted. Filters of different classes should always be treated as a logical-AND. The frame must match a filter in all programmed classes to be accepted. For more information, see the *Filters* section of mac_capab_rings(9S).

RETURN VALUES

Upon successful completion, the driver should ensure that the filter has been added or removed and return **0**. Otherwise, it should return the appropriate error number.

ERRORS

The device driver may return one of the following errors. While this list is not intended to be exhaustive, it is recommended to use one of these if possible.

EEXIST	The requested filter has already been programmed into the device.
EINVAL	The address *mac* is not a valid unicast address. The VLAN *vlan* is not a valid VLAN identifier.
EIO	The driver encountered a device or transport error while trying to update the device's state.
ENOENT	The driver was asked to remove a filter which was not currently programmed.
ENOTSUP	An unknown flag is present in the **flags**() argument.
ENOSPC	The driver has run out of available hardware filters.

SEE ALSO

mac(9E), mac_capab_rings(9E), mr_gget(9E), mac_group_info(9S)

NAME

mc_tx, **mri_tx -** transmit a message block chain

SYNOPSIS

#include <sys/mac_provider.h>

*mblk_t **
prefix_m_tx(*void *driver*, *mblk_t *mp_chain*);

*mblk_t **
prefix_ring_tx(*void *rh*, *mblk_t *mp_chain*);

INTERFACE LEVEL

illumos DDI specific

The **mri_tx**() entry point is **Evolving**. API and ABI stability is not guaranteed.

PARAMETERS

driver A pointer to the driver's private data that was passed in via the *m_pdata* member of the mac_register(9S) structure to the mac_register(9F) function.

rh A pointer to the ring's private data that was passed in via the *mri_driver* member of the mac_ring_info(9S) structure as part of the mr_rget(9E) entry point.

mp_chain A series of mblk(9S) structures that may have multiple independent packets linked together on their *b_next* member.

DESCRIPTION

The **mc_tx**() entry point is called when the system requires a device driver to transmit data. The device driver will receive a chain of message blocks. The *mp_chain* argument represents the first frame. The frame may be spread out across one or more mblk(9S) structures that are linked together by the *b_cont* member. There may be multiple frames, linked together by the *b_next* pointer of the mblk(9S).

For each frame, the driver should allocate the required resources and prepare it for being transmitted on the wire. The driver may opt to copy those resources to a DMA buffer or it may bind them. For more information on these options, see the *MBLKS AND DMA* section of mac(9E).

As it processes each frame in the chain, if the device driver has advertised either of the MAC_CAPAB_HCKSUM or MAC_CAPAB_LSO flags, it must check whether either apply for the given frame using the mac_hcksum_get(9F) and mac_lso_get(9F) functions respectively. If either is

enabled for the given frame, the hardware must arrange for that to be taken care of.

For each frame that the device driver processes it is responsible for doing one of three things with it:

1. Transmit the frame.

2. Drop the frame by calling freemsg(9F) on the individual mblk_t.

3. Return the frames to indicate that resources are not available.

The device driver is in charge of the memory associated with *mp_chain*. If the device driver does not return the message blocks to the MAC framework, then it must call freemsg(9F) on the frames. If it does not, the memory associated with them will be leaked. When a frame is being transmitted, if the device driver performed DMA binding, it should not free the message block until after it is guaranteed that the frame has been transmitted. If the message block was copied to a DMA buffer, then it is allowed to call freemsg(9F) at any point.

In general, the device driver should not drop frames without transmitting them unless it has no other choice. Times when this happens may include the device driver being in a state where it can't transmit, an error was found in the frame while trying to establish the checksum or LSO state, or some other kind of error that represents an issue with the passed frame.

The device driver should not free the chain when it does not have enough resources. For example, if entries in a device's descriptor ring fill up, then it should not drop those frames and instead should return all of the frames that were not transmitted. This indicates to the stack that the device is full and that flow control should be asserted. Back pressure will be applied to the rest of the stack, allowing most systems to behave better.

Once a device driver has returned unprocessed frames from its **mc_tx**() entry point, then the device driver will not receive any additional calls to its **mc_tx**() entry point until it calls the mac_tx_update(9F) function to indicate that resources are available again. Note that because it is the device driver that is calling this function to indicate resources are available, it is very important that it only return frames in cases where the device driver itself will be notified that resources are available again. For example, when it receives an interrupt indicating that the data that it transmitted has been completed so it can use entries in its descriptor ring or other data structures again.

The device driver can obtain access to its soft state through the *driver* member. It should cast it to the appropriate structure. The device driver should employ any necessary locking to access the transmit related data structures. Note that the device driver should expect that it may have its transmit endpoints called into from other threads while it's servicing device interrupts related to them.

The **mri_tx**() entry point is similar to the **mc_tx**() entry point, except that it is used by device drivers that have negotiated the MAC_CAPAB_RINGS capability for transmitting. The driver should follow all of the same rules described earlier, except that it will access a ring-specific data structure through *rh* and when it needs to update that there is additional space available, it must use mac_tx_update_ring(9F) and not mac_tx_update(9F).

When the **mri_tx**() entry point is called, the ring that should be used has been specified. The driver must not attempt to use any other ring than the one specified by *rh* for any reason, including a lack of resources or an attempt to perform its own hashing.

CONTEXT

The **mc_tx**() entry point may be called from **kernel** or **interrupt** context.

RETURN VALUES

Upon successful completion, the device driver should return NULL. Otherwise, it should return all unprocessed message blocks and ensure that it calls either mac_tx_update(9F) or mac_tx_ring_update(9F) some time in the future.

SEE ALSO

mac(9E), mac_capab_rings(9E), mr_rget(9E), freemsg(9F), mac_hcksum_get(9F), mac_lso_get(9F), mac_register(9F), mac_tx_ring_update(9F), mac_tx_update(9F), mac_register(9S), mac_ring_info(9S), mblk(9S)

NAME

mc_unicst - set device unicast address

SYNOPSIS

#include <sys/mac_provider.h>

int
prefix_m_unicst(*void *driver, const uint8_t *mac*);

INTERFACE LEVEL

illumos DDI specific

PARAMETERS

driver A pointer to the driver's private data that was passed in via the **m_pdata** member of the mac_register(9S) structure to the mac_register(9F) function.

mac A pointer to an array of bytes that contains the new unicast address of the device. It is guaranteed to be at least a number of bytes long equal to the length of the MAC plugin's address length. For Ethernet devices that length is six bytes, **ETHERADDRL**.

DESCRIPTION

The **mc_unicst**() entry point is used by the MAC framework to indicate that the device driver should update the primary MAC address of the device. In the basic mode of operation, this entry point is required and the device has a single primary MAC address. If multiple MAC addresses are required, the device will be placed into promiscuous mode. This call should overwrite the existing MAC address that is programmed into the device.

As noted in the *PARAMETERS* section, the *mac* array is guaranteed to be at least as many bytes as is required to specify an address; however, it should be assumed to be no longer than that value.

The device driver can optionally assert that the address is in the valid form for a unicast address and then program the device. The device driver can access its device soft state by casting the *device* pointer to the appropriate structure. As this may be called while other operations are ongoing, the device driver should employ the appropriate locking while updating the data.

It is recommended that device drivers always maintain a copy of the current unicast address in its soft state so that way it can recover from various device reset and errors or handle requests to suspend and resume the device that may result in device registers being cleared.

Some devices support multiple MAC address filters. The **mc_unicst**() entry point only supports a single

MAC address. In this case, devices should only use a single MAC address and replace that MAC address. To enable the operating system to take advantage of multiple unicast MAC address filters, the driver should implement the MAC_CAPAB_RINGS capability. See mac_capab_rings(9E) for more information.

RETURN VALUES

Upon successful completion, the device driver should have updated its unicast filter and return **0**. Otherwise, the MAC address should remain unchanged and the driver should return an appropriate error number.

ERRORS

The device driver may return one of the following errors. While this list is not intended to be exhaustive, it is recommended to use one of these if possible.

EINVAL	The address *mac* is not a valid unicast address.
EIO	The driver encountered a device or transport error while trying to update the device's state.

SEE ALSO

mac(9E), mac_capab_rings(9E), mac_register(9F), mac_register(9S)

NAME

mgi_start, **mgi_stop** - Start and stop a MAC group

SYNOPSIS

#include <sys/mac_provider.h>

int
prefix_group_start(*mac_group_driver_t gh*);

void
prefix_group_stop(*mac_group_driver_t gh*);

INTERFACE LEVEL

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

PARAMETERS

driver A pointer to the ring's private data that was passed in via the *mgi_driver* member of the mac_group_info(9S) structure as part of the mr_gget(9E) entry point.

DESCRIPTION

The **mgi_start**() and **mgi_stop**() are optional entry points that allow a chance for the device driver to do anything that it needs to do to start and stop a group respectively. The group that is being operated on is identified by *gh*.

RETURN VALUES

Upon successful completion, the **mgi_start**() interface should return **0**. Otherwise, it should return the appropriate error number.

SEE ALSO

mac(9E), mac_capab_rings(9E), mr_gget(9E), mac_group_info(9S)

NAME

mi_enable, **mi_disable** - MAC interrupt enable and disable entry points

SYNOPSIS

```
#include <sys/mac_provider.h>

int
prefix_intr_enable(mac_intr_handle_t driver);

int
prefix_intr_disable(mac_intr_handle_t driver);
```

INTERFACE LEVEL

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

PARAMETERS

driver A pointer to the mac interrupt's private data that was passed in via the *mi_handle* member of the mac_intr(9S) structure.

DESCRIPTION

The **mi_enable**() and **mi_disable**() entry points are used by the MAC framework when it wishes to disable the generation of interrupts for the ring and poll on the it through the mri_poll(9E) entry point.

These entry points should enable and disable the generation of the interrupt for the ring that is represented *driver*. Generally, *driver* is part of a receive ring's *mri_intr* member in the mac_ring_info(9S) structure and refers to a specific ring. Importantly, this entry point is not asking to enable and disable the interrupt for all consumers of it. This should be implemented through device specific means such as writing to registers or sending control messages to enable or disable the generation of interrupts for the specified ring.

Drivers must not implement this in terms of the DDI interrupt functions such as ddi_intr_enable(9F) and ddi_intr_disable(9F).

When manipulating the device's control of interrupts, the driver should be careful to serialize these changes with the ongoing processing of interrupts through the interrupt handler and the mri_poll(9E) entry point. These should all be protected by the same mutex which is scoped to the ring itself when the ability to turn on and off interrupt generation may be manipulated on a per-ring basis. Failure to properly synchronize this may lead to the driver mistakenly delivering the same packet twice through both its interrupt handler and its mri_poll(9E) entry point.

RETURN VALUES

Upon successful completion, the **mi_enable**() and **mi_disable**() entry points should return **0**. Otherwise the appropriate error number should be returned.

SEE ALSO

mac(9E), mac_capab_rings(9E), mri_poll(9E), ddi_intr_disable(9F), ddi_intr_enable(9F), mac_intr(9S), mac_ring_info(9S)

NAME

mr_gget - fill in group information

SYNOPSIS

#include <sys/mac_provider.h>

int
prefix_fill_group(*void *driver, mac_ring_type_t rtype, const int group_index, mac_group_info_t *infop, mac_group_handle_t gh*);

INTERFACE LEVEL

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

PARAMETERS

driver A pointer to the driver's private data that was passed in via the *m_pdata* member of the mac_register(9S) structure to the mac_register(9F) function.

rtype A value indicating the type of ring that makes up the groups. Valid values include:

MAC_RING_TYPE_RX
The group is intended for use with receive rings.

MAC_RING_TYPE_TX
The group is intended for use with transmit rings.

group_index An integer value indicating the group that this ring belongs to. Groups are numbered starting from zero.

infop A pointer to an instance of a mac_group_info(9S) structure.

gh An opaque pointer to a group handle that can be used to identify this group.

DESCRIPTION

The **mr_gget**() entry point provides a means for the device driver to fill in information about a group. The driver must fill in information into the *infop* argument. For the list of fields and an explanation of how to fill them in, please see mac_group_info(9S).

The *rtype* argument describes whether this is a group of receive rings or a group of transmit rings. This is identified by the value in *rtype* which will be MAC_RING_TYPE_RX for a receive group and MAC_RING_TYPE_TX for a transmit group. The group information that is filled in varies between

transmit and receive groups. If separate entry points were not specified in the mac_capab_rings(9E) structure, then the driver must ensure that it checks this value and acts appropriately.

The *group_index* argument is used to uniquely identify a group. Groups are numbered starting at zero and end at one less then the number of groups specified in *mr_gnum* member of the *mac_capbab_rings_t* structure which is described in mac_capab_rings(9E).

After filling in the group information in *infop*, the driver should make sure to store the group handle in *gh* for future use.

CONTEXT

The **mr_gget**() entry point will be called in response to a driver calling the mac_register(9F) function and the driver has acknowledged that it supports the MAC_CAPAB_RINGS capability.

SEE ALSO

mac(9E), mac_capab_rings(9E), mac_register(9F), mac_group_info(9S), mac_register(9S)

NAME

mr_rget - fill in ring information

SYNOPSIS

#include <sys/mac_provider.h>

int
prefix_fill_ring(*void *driver, mac_ring_type_t rtype, const int group_index, const int ring_index, mac_ring_info_t *infop, mac_ring_handle_t rh*);

INTERFACE LEVEL

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

PARAMETERS

driver A pointer to the driver's private data that was passed in via the *m_pdata* member of the mac_register(9S) structure to the mac_register(9F) function.

group_index An integer value indicating the group that this ring belongs to. Groups are numbered starting from zero.

rtype A value indicating the type of ring. Valid values include:

MAC_RING_TYPE_RX
: The ring is a receive ring.

MAC_RING_TYPE_TX
: The ring is a transmit ring.

ring_index An integer indicating the index of the ring inside of the group. Ring indexes are numbered starting from zero. Each group has its own set of ring indexes.

infop A pointer to an instance of a mac_ring_info(9S) structure.

rh An opaque pointer to a ring handle that can be used to identify this ring.

DESCRIPTION

The **mr_rget**() entry point provides a means for the device driver to fill in information about a ring. The driver must fill in information into the *infop* argument. For the list of fields and an explanation of how to fill them in, please see mac_ring_info(9S).

The *rtype* argument describes whether this is a receive ring or transmit ring identified by a value of MAC_RING_TYPE_RX or MAC_RING_TYPE_TX respectively. The ring information that is filled in varies between transmit and receive rings. If separate entry points were not specified in the mac_capab_rings(9E) structure, then the driver must ensure that it checks this value.

The *group_index* and *ring_index* arguments are used to uniquely identify a ring. The number of groups that a driver supports is based on the values present in the *mr_gnum* member of the *mac_capbab_rings_t* structure which is described in mac_capab_rings(9E). The group index ranges from zero to the specified number of groups minus one. The number of rings in the group is determined based on the values specified in mac_group_info(9S) structure that is filled in during the mr_gget(9E) entry point. The numbering for each group is independent and always starts at zero. Based on the combination of group and ring index, the driver should be able to map that to a unique ring.

After filling out the ring structure in *infop*, the driver should make sure to store the ring handle in *rh* for future use. This is required for callbacks such as mac_rx_ring(9F) or mac_tx_ring_update(9F).

CONTEXT

The **mr_gget**() entry point will be called in response to a driver calling the mac_register(9F) function and the driver has acknowledged that it supports the MAC_CAPAB_RINGS capability.

SEE ALSO

mac(9E), mac_capab_rings(9E), mr_gget(9E), mac_register(9F), mac_rx_ring(9F), mac_tx_ring_update(9F), mac_group_info(9S), mac_register(9S), mac_ring_info(9S)

NAME

mri_poll - Poll a ring for network data

SYNOPSIS

#include <sys/mac_provider.h>

*mblk_t **
prefix_ring_poll(*void *driver, int poll_bytes*);

INTERFACE LEVEL

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

PARAMETERS

driver A pointer to the ring's private data that was passed in via the *mri_driver* member of the mac_ring_info(9S) structure as part of the mr_rget(9E) entry point.

poll_bytes The maximum number of bytes that the driver should poll in a given call.

DESCRIPTION

The **mri_poll**() entry point is called by the MAC framework when it wishes to have the driver check the ring specified by *driver* for available data.

The device driver should perform the same logic that it would when it's procesing an interrupt and as described in the *Receiving Data* section of mac(9E). The main difference is that instead of calling mac_ring_rx(9E), it should instead return that data. Also, while an interrupt may map to more than one ring, the driver should only process the ring indicated by *driver*.

Drivers should exercise caution with the locking between the polling, interrupt disabling routines, and the interrupt handler. This lock is generally scoped to a receive ring and is used to synchronize the act of transitioning between polling and handling interrupts. That means that in addition to the **mri_poll**() entry point, the mi_enable(9E) and mi_disable(9E) entry points should synchronize on the same lock when transitioning the card. While the driver does not need to hold this lock across the entire interrupt handler, it should hold it when it is processing and trying to read frames for the specified ring in its interrupt handler.

The driver should limit the number of frames it collects based on the size value present in the *poll_bytes* argument.

RETURN VALUES

Upon successful completion, the device driver should return a message block chain of collected frames.

If no frames are available, then it should return NULL.

SEE ALSO

mac(9E), mac_capab_rings(9E), mac_ring_rx(9E), mi_disable(9E), mi_enable(9E), mr_rget(9E), mac_ring_info(9S)

NAME

mri_start, **mri_stop -** ring start and stop entry point

SYNOPSIS

#include <sys/mac_provider.h>

int
prefix_ring_start(*mac_ring_driver_t rh*, *uint64_t mr_gen*);

void
prefix_ring_stop(*mac_ring_driver_t rh*);

INTERFACE LEVEL

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

PARAMETERS

rh A pointer to the ring's private data that was passed in via the *mri_driver* member of the mac_ring_info(9S) structure as part of the mr_rget(9E) entry point.

mr_gen A 64-bit generation number.

DESCRIPTION

The **mri_start**() entry point is a required entry point that allows the driver a chance to take any action to start the ring in hardware. The ring is indicated by the driver's private data structure structure for the ring: *rh*. The driver should record the value of *mr_gen* in its private data structure. This value is used when receiving data as the argument to the mac_ring_rx(9F) function. For many drivers, the only action that is required is recording the generation number.

The **mri_stop**() entry point is an optional entry point that allows the driver a chance to take any actions to stop the ring in hardware. The ring is indicated by its private data structure *rh*.

RETURN VALUES

Upon successful completion, the device driver should return **0** from the **mri_start**() entry point. Otherwise, they should return anon-zero positive error number to indicate the error that occurred.

SEE ALSO

mac(9E), mac_capab_rings(9E), mr_rget(9E), mac_ring_rx(9F), mac_ring_info(9S)

NAME

mri_stat - Statistics collection entry point for rings

SYNOPSIS

#include <sys/mac_provider.h>

int
prefix_ring_stat(*mac_ring_driver_t rh, uint_t stat, uint64_t *val*);

INTERFACE LEVEL

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

PARAMETERS

rh A pointer to the ring's private data that was passed in via the *mri_driver* member of the mac_ring_info(9S) structure as part of the mr_rget(9E) entry point.

stat The numeric identifier of a statistic.

val A pointer to a 64-bit unsigned value in which the device driver should place statistic.

DESCRIPTION

The **mri_stat**() entry point is called by the MAC framework to get statistics that have been scoped to the ring, indicated by *rh*.

The set of statistics that the driver should check depends on the kind of ring that is in use. If the driver encounters an unknown statistic it should return ENOTSUP. All the statistics should be values that are scoped to the ring itself. This is in contrast to the normal mc_getstat(9E) entry point, which has statistics for the entire device. Other than the scoping, the statistics listed below have the same meaning as they do in the *STATISTICS* section of mac(9E). See mac(9E) for more detailed meanings of those statistics.

Receive rings should support the following statistics:

- MAC_STAT_IPACKETS
- MAC_STAT_RBYTES

Transmit rings should support the following statitics:

- MAC_STAT_OBYTES

- MAC_STAT_OPACKETS

EXAMPLES

The following example shows how a driver might structure its **mri_stat**() entry point.

```
#include <sys/mac_provider.h>

/*
 * Note, this example merely shows the structure of the function. For
 * the purpose of this example, we assume that we have a per-ring
 * structure which has members that indicate its stats and that it has a
 * lock which is used to serialize access to this data.
 */

static int
example_tx_ring_stat(mac_ring_driver_t rh, uint_t stat, uint64_t *val)
{
        example_tx_ring_t *etrp = arg;

        mutex_enter(&etrp->etrp_lock);
        switch (stat) {
        case MAC_STAT_OBYTES:
                *val = etrp->etrp_stats.eps_obytes;
                break;
        case MAC_STAT_OPACKETS:
                *val = etrp->etrp_stats.eps_opackets;
                break;
        default:
                mutex_exit(&etrp->etrp_lock);
                return (ENOTSUP);
        }
        mutex_exit(&etrp->etrp_lock);

        return (0);
}

static int
example_rx_ring_stat(mac_ring_driver_t rh, uint_t stat, uint64_t *val)
{
        example_rx_ring_t *errp = arg;
```

```
        mutex_enter(&errp->errp_lock);
        switch (stat) {
        case MAC_STAT_RBYTES:
                *val = errp->errp_stats.eps_ibytes;
                break;
        case MAC_STAT_IPACKETS:
                *val = errp->errp_stats.eps_ipackets;
                break;
        default:
                mutex_exit(&errp->errp_lock);
                return (ENOTSUP);
        }
        mutex_exit(&errp->errp_lock);

        return (0);
}
```

ERRORS

The device driver may return one of the following errors. While this list is not intended to be exhaustive, it is recommend to use one of these if possible.

ENOTSUP	The specified statistic is unknown, unsupported, or unimplemented.
EIO	A transport or DMA FM related error occurred while trying to sync data from the device.
ECANCELLED	The device is not currently in a state where it can currently service the request.

SEE ALSO

mac(9E), mac_capab_rings(9E), mc_getstat(9E), mr_rget(9E), mac_ring_info(9S)

NAME

mac_rx, **mac_ring_rx -** deliver frames from a driver to the system

SYNOPSIS

#include <sys/mac_provider.h>

void
mac_rx(*mac_handle_t mh, mac_resource_handle_t mrh, mblk_t *mp_chain*);

void
mac_rx_ring:(*mac_handle_t mh, mac_ring_handle_t mring, mblk_t *mp_chain, uint64_t mr_gen*);

INTERFACE LEVEL

illumos DDI specific

The **mac_rx_ring**() function point is **Evolving**. API and ABI stability is not guaranteed.

PARAMETERS

mh The MAC handle obtained from a call to mac_register(9F).

mrh A reserved parameter that should be passed as NULL.

mring A pointer to the ring handle that was passed to the driver in the mr_rget(9E) entry point.

mp_chain A series of one or more mblk(9S) structures chained together by their **b_next** member.

mr_gen The generation number for the current ring. The generation comes from the mri_start(9E) entry point.

DESCRIPTION

The **mac_rx**() function is used by device drivers to deliver frames that a device driver has received to the rest of the operating system. This will generally be called at the end of a device driver's interrupt handler after it is has converted all of the incoming data into a chain of mblk(9S) structures. For a full description of the process that the device driver should take as part of receiving data, see the *Receiving Data* section of mac(9E).

Device drivers should ensure that they are not holding any of their own locks when they call the **mac_rx**() function.

Device drivers should not call the **mac_rx**() function after each individual mblk_t is assembled. Rather,

the device driver should batch up as many frames as it is willing to process in a given interrupt or are available.

The **mac_rx_ring()** function is similar to the *mac_rx* function; however, it should be called by device drivers that have negotiated the MAC_CAPAB_RINGS capability and indicated that it supports receive groups. Device drivers that have negotiated this capability must not call the **mac_rx()** function, but use the **mac_rx_ring()** function instead. The driver should pass the ring handle in *mring* for the ring in question that it processed. If more than one ring was processed during an interrupt, then the driver must call **mac_ring_rx()** once for each ring and ensure that the *mr_gen* argument matches what was passed to the driver during the mri_start(9E) entry point.

When a driver supporting the MAC_CAPAB_RINGS capability is asked to poll via their mri_poll(9E) entry point, then the driver should not call the **mac_ring_rx()** function.

CONTEXT

The **mac_rx()** function can be called from **user**, **kernel**, or **interrupt** context.

SEE ALSO

mac(9E), mac_capab_rings(9E), mr_rget(9E), mri_poll(9E), mri_start(9E), mac_register(9F), mblk(9S)

NAME

mac_tx_update, **mac_tx_ring_update -** indicate that a device can transmit again

SYNOPSIS

#include <sys/mac_provider.h>

void
mac_tx_update(*mac_handle_t mh*);

void
mac_tx_ring_update(*mac_handle_t mh, mac_ring_handle_t mrh*);

INTERFACE LEVEL

illumos DDI specific

The **mac_tx_ring_update**() function point is **Evolving**. API and ABI stability is not guaranteed.

PARAMETERS

mh The MAC handle obtained from a call to mac_register(9F).

mrh The MAC ring handle obtained when the driver's ring entry point mr_rget(9E) was called.

DESCRIPTION

The **mac_tx_update**() function is used by device drivers to indicate that the device represented by the handle *mh* can transmit data again. It should only be called after the device driver has returned data from its mc_tx(9E) endpoint. For more information on when this should be called, see both mc_tx(9E) and the *Transmitting Data and Back Pressure* section of mac(9E).

Device drivers should not hold any of their own locks when calling into this function. See the *MAC Callbacks* section of mac(9E) for more information.

When a driver has negotiated the MAC_CAPAB_RINGS capability and indicated that it supports transmit groups, it must not use the **mac_tx_update**() function and should instead call the **mac_tx_ring_update**() function targeting a specific ring instead. The ring that is being updated is specified by the ring handle passed in the *mrh* argument. The ring should have previously returned frames from its mri_tx(9E) entry point to indicate that it was blocked.

In all other respects, the **mac_tx_ring_update**() function is similar to the **mac_tx_update**() function.

CONTEXT

The **mac_tx_update**() function may be called from **user**, **kernel**, or **interrupt** context.

SEE ALSO

mac(9E), mac_capab_rings(9E), mac_tx(9E), mc_tx(9E), mr_rget(9E), mri_tx(9E), mac_register(9F)

NAME

mac_callbacks, **mac_callbacks_t** - networking device driver entry points structure

SYNOPSIS

#include <sys/mac_provider.h>

INTERFACE LEVEL

illumos DDI specific

DESCRIPTION

The **mac_callbacks** structure is used by GLDv3 networking device drivers implementing and using the mac(9E) framework and interfaces.

The structure is normally allocated statically by drivers as a single global entry. A pointer to it is passed as the *m_callbacks* member of the *mac_register_t* structure.

TYPES

The following types define the function pointers in use in the *mac_register_t*.

```
typedef int          (*mac_getstat_t)(void *, uint_t, uint64_t *);
typedef    int       (*mac_start_t)(void *);
typedef void         (*mac_stop_t)(void *);
typedef int          (*mac_setpromisc_t)(void *, boolean_t);
typedef int          (*mac_multicst_t)(void *, boolean_t, const uint8_t *);
typedef int          (*mac_unicst_t)(void *, const uint8_t *);
typedef void         (*mac_ioctl_t)(void *, queue_t *, mblk_t *);
typedef void         (*mac_resources_t)(void *);
typedef mblk_t             *(*mac_tx_t)(void *, mblk_t *);
typedef    boolean_t(*mac_getcapab_t)(void *, mac_capab_t, void *);
typedef    int       (*mac_open_t)(void *);
typedef void         (*mac_close_t)(void *);
typedef    int       (*mac_set_prop_t)(void *, const char *, mac_prop_id_t,
                     uint_t, const void *);
typedef    int       (*mac_get_prop_t)(void *, const char *, mac_prop_id_t,
                     uint_t, void *);
typedef void         (*mac_prop_info_t)(void *, const char *, mac_prop_id_t,
                     mac_prop_info_handle_t);
```

STRUCTURE MEMBERS

```
uint_t          mc_callbacks; /* Denotes which callbacks are set */
```

```
mac_getstat_t  mc_getstat;     /* Get the value of a statistic */
mac_start_t    mc_start; /* Start the device */
mac_stop_t     mc_stop; /* Stop the device */
mac_setpromisc_t mc_setpromisc; /* Enable or disable promiscuous mode */
mac_multicst_t      mc_multicst;   /* Enable or disable a multicast addr */
mac_unicst_t   mc_unicst;      /* Set the unicast MAC address */
mac_tx_t mc_tx;           /* Transmit a packet */
void        *mc_reserved; /* Reserved, do not use */
mac_ioctl_t    mc_ioctl; /* Process an unknown ioctl */
mac_getcapab_t     mc_getcapab;  /* Get capability information */
mac_open_t    mc_open;/* Open the device */
mac_close_t    mc_close;       /* Close the device */
mac_set_prop_t     mc_setprop;    /* Set a device property */
mac_get_prop_t     mc_getprop;    /* Get a device property */
mac_prop_info_t    mc_propinfo;  /* Get property information */
```

The *mc_callbacks* member is used to denote which of a series of optional callbacks are present. This method allows additional members to be added to the *mac_callbacks_t* structure while maintaining ABI compatibility with existing modules. If a member is not mentioned below, then it is a part of the base version of the structure and device drivers do not need to set anything to indicate that it is present. The *mc_callbacks* member should be set to the bitwise inclusive OR of the following pre-processor values:

MC_IOCTL Indicates that the *mc_ioctl* structure member has been set.

MC_GETCAPAB
: Indicates that the *mc_getcapab* structure member has been set.

MC_OPEN Indicates that the *mc_open* structure member has been set.

MC_CLOSE Indicates that the *mc_close* structure member has been set.

MC_SETPROP
: Indicates that the *mc_setprop* structure member has been set.

MC_GETPROP
: Indicates that the *mc_getprop* structure member has been set.

MC_PROPINFO
: Indicates that the *mc_propinfo* structure member has been set.

MC_PROPERTIES

Indicates that the *mc_getprop*, *mc_propinfo*, and *mc_setprop* structure members have been set.

The *mc_getstat* function defines an entry point used to receive statistics about the device. A list of statistics that it is required to support is available in mac(9E). For more information on the requirements of the function, see mc_getstat(9E).

The *mc_start* member defines an entry point that is used to start the device. For more information on the requirements of the function, see mc_start(9E).

The *mc_stop* member defines an entry point that is used to stop the device. It is the opposite of the *mc_start* member. For more information on the requirements of the function, see mc_stop(9E).

The *mc_setpromisc* member is used to enable and disable promiscuous mode on the device. For more information on the requirements of the function, see mc_setpromisc(9E).

The *mc_multicst* member is used to enable or disable multicast addresses in the device's filters. For more information on the requirements of the function, see mc_multicst(9E).

The *mc_unicst* member is used to set the primary unicast MAC address of the device. For more information on the requirements of the function, see mc_unicst(9E).

The *mc_tx* member is used to transmit a single message on the wire. For more information on the requirements of the function, see mc_tx(9E).

The *mc_ioctl* member is used to process device specific ioctls. The MAC framework does not define any ioctls that devices should handle; however, there may be private ioctls for this device. This entry point is optional. For it to be considered, the MC_IOCTL value must be present in the *mc_callbacks* member. For more information on the requirements of the function, see mc_ioctl(9E).

The *mc_getcapab* member is used to determine device capabilities. Each capability has its own data and semantics associated with it. A list of capabilities is provided in mac(9E). This entry point is optional. For it to be used, the MC_GETCAPAB value must be present in the *mc_callbacks* member. For more information on the requirements of the function, see mc_getcapab(9E).

The *mc_open* member is used to provide specific actions to take when the device is opened. Note that most device drivers will not have a need to implement this. It is not required for this function to be implemented for this device to be used with dlpi(7P). This entry point is optional. For it to be used, the MC_OPEN value must be present in the *mc_callbacks* member. For more information on the

requirements of the function, see mc_open(9E).

The *mc_close* member is used to provide specific actions to take when the device is closed. Note that most device drivers will not have a need to implement this. It is not required for this function to be implemented for this device to be used with dlpi(7P). This entry point is optional. For it to be used, the MC_CLOSE value must be present in the *mc_callbacks* member. For more information on the requirements of the function, see mc_close(9E).

The *mc_getprop* member is used to get the current value of a property from the device. A list of properties, their sizes, and their interpretation is available in mac(9E). This entry point is optional. For it to be used, the MC_GETPROP value must be present in the *mc_callbacks* member. For more information on the requirements of the function, see mc_getprop(9E).

The *mc_setprop* member is used to set the value of a device property. A list of properties, their sizes, and their interpretation is available in mac(9E). This entry point is optional. For it to be used, the MC_SETPROP value must be present in the *mc_callbacks* member. For more information on the requirements of the function, see mc_setprop(9E).

The *mc_propinfo* member is used to obtain metadata about a property such as its default value, whether or not it is writable, and more. A list of properties, their sizes, and their interpretation is available in mac(9E). This entry point is optional. For it to be used, the MC_PROPINFO value must be present in the *mc_callbacks* member. For more information on the requirements of the function, see mc_propinfo(9E).

Required Members

Many members in the structure are optional; however, the following members must be set or a call to mac_register(9F) will fail.

- *mc_getstat*
- *mc_start*
- *mc_stop*
- *mc_setpromisc*
- *mc_multicst*
- *mc_tx*

- *mc_unicst*

Devices which implement the MAC_CAPAB_RINGS capability for receive rings must not implement the *mc_unicst* entry point. Devices which implement the MAC_CAPAB_RINGS capability for transmit rings must not implement the *mc_tx* entry points. For more information about the capability, please see mac_capab_rings(9E).

Generally, a device that implements one of *mc_getprop*, *mc_setprop*, or *mc_propinfo* will want to implement all three endpoints to ensure that the property is fully integrated into user land utilities such as dladm(1M).

SEE ALSO

dladm(1M), dlpi(7P), mac(9E), mac_capab_rings(9E), mc_close(9E), mc_getcapab(9E), mc_getprop(9E), mc_getstat(9E), mc_ioctl(9E), mc_multicst(9E), mc_open(9E), mc_propinfo(9E), mc_setpromisc(9E), mc_setprop(9E), mc_start(9E), mc_stop(9E), mc_tx(9E), mc_unicst(9E), mac_register(9F), mac_register(9S)

NAME

mac_group_info, **mac_group_info_t** - MAC group information structure

SYNOPSIS

#include <sys/mac_provider.h>

INTERFACE LEVEL

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

DESCRIPTION

The *mac_group_info_t* structure is used by the MAC framework as part of the MAC_CAPAB_RINGS capability. For background on the MAC framework, please see mac(9E).

When a device driver declares that it supports the MAC_CAPAB_RINGS capability and fills out the capability structure as described in mac_capab_rings(9E), it indicates that it supports a number of transmit and receive groups. For each group that it indicates, its mr_gget(9E) entry point will be called, during which it will have to fill out the *mac_group_info_t* structure described here.

TYPES

The following types define the function pointers in use in the *mac_group_info_t* structure.

```
typedef int (*mac_group_start_t)(mac_group_driver_t);
typedef void (*mac_group_stop_t)(mac_group_driver_t);
typedef int (*mac_add_mac_addr_t)(mac_group_driver_t,
    const uint8_t *mac, uint_t flags)
typedef int (*mac_rem_mac_addr_t)(mac_group_driver_t,
    const uint8_t *mac, uint_t flags)
typedef int (*mac_add_vlan_t)(mac_group_driver_t, uint16_t vlan,
     uint_t flags)
typedef int (*mac_rem_vlan_t)(mac_group_driver_t, uint16_t vlan,
     uint_t flags)
typedef int (*mac_add_mv_filter_t)(mac_group_driver_t,
    const uint8_t *mac, uint16_t vlan, uint_t flags);
typedef int (*mac_rem_mv_filter_t)(mac_group_driver_t,
    const uint8_t *mac, uint16_t vlan, uint_t flags);
```

STRUCTURE MEMBERS

```
uint_t              mgi_extensions;
uint_t              mgi_flags;
mac_group_driver_t  mgi_driver;
```

```
mac_group_start_t       mgi_start;
mac_group_start_t       mgi_stop;
uint_t                  mgi_count;
mac_add_mac_addr_t      mgi_addmac;
mac_rem_mac_addr_t      mgi_remmac;
mac_add_vlan_t          mgi_addvlan;
mac_rem_vlan_t          mgi_remvlan;
mac_add_mv_filter_t     mgi_addmvf;
mac_rem_mv_filter_t     mgi_remmvf;
```

The *mgi_extensions* member is used to negotiate extensions between the MAC framework and the device driver. The MAC framework will set the value of *mgi_extensions* to include all of the currently known extensions. The driver should intersect this list with the set that the driver supports. At this time, no such features are defined and the driver should set this member to **0**.

The *mgi_flags* member is used to indicate various additional capabilities supported by the group. None are currently defined and the driver should set this to **0**.

The *mgi_driver* member should be set by the driver to a driver-specific value that represents the data structure that corresponds to this group. The driver will receive this value in all of the callback functions that are defined in this structure.

The *mgi_start* member is an optional entry point. If the driver needs to take a specific action before it the group is used, then it should set this to a function. For more information, see mgi_start(9E).

The *mgi_stop* member is an optional entry point. If the driver needs to take a specific action when the group is being stopped, then it should set this to a function. For more information, see mgi_stop(9E).

The *mgi_count* member should be set to a count of the number of rings that are present in this group. When the group type is MAC_GROUP_TYPE_STATIC, then the value in *mgi_count* represents the fixed number of rings available to the group.

The *mgi_addmac* member is an optional entry point and should be set to a function that can add a MAC address filter to the group in hardware. For more information, see mgi_addmac(9E). This member only has meaning for a receive group, transmit groups should set this to NULL.

The *mgi_remmac* member is an optional entry point and should be set to a function that can remove a MAC address filter from a group in hardware. If the *mgi_addmac* member is a valid pointer, then this entry point must be as well. For more information, see mgi_remmac(9E). This member only has meaning for a receive group, transmit groups should set this to NULL.

The *mgi_addvlan* member is an optional entry point and should be set to a function that can add a VLAN filter to the group in hardware. For more information, see mgi_addvlan(9E). This member only has meaning for a receive group, transmit groups should set this to NULL.

The *mgi_remvlan* member is an optional entry point and should be set to a function that can remove a VLAN filter from a group in hardware. If the *mgi_addvlan* member is a valid pointer, then this entry point must be as well. For more information, see mgi_remvlan(9E). This member only has meaning for a receive group, transmit groups should set this to NULL.

The *mgi_addmvf* member is an optional entry point and should be set to a function that can add a MAC Address and VLAN tuple filter to the group in hardware. For more information, see mgi_addmvf(9E). This member only has meaning for a receive group, transmit groups should set this to NULL.

The *mgi_remmvf* member is an optional entry point and should be set to a function that can remove a MAC address and VLAN tuple filter from the group in hardware. If the *mgi_addmvf* member is a valid pointer, then this entry point must be as well. For more information, see mgi_remmvf(9E). This member only has meaning for a receive group, transmit groups should set this to NULL.

Required Members

All of the non-function pointers described in this manual are required members for both transmit and receive groups. The *mgi_start* and *mgi_stop* members are optional for both transmit and receive groups.

For transmit groups, all of the filter entry points must be set to NULL.

Receive groups must have some way to set a MAC address filter. This means that one of the MAC address related functions must be set. The driver must implement either *mgi_addmac* and *mgi_remmac* or *mgi_addmvf* and *mgi_remmvf*.

SEE ALSO

mac(9E), mac_capab_rings(9E), mgi_addmac(9E), mgi_addmvf(9E), mgi_addvlan(9E), mgi_remmac(9E), mgi_remmvf(9E), mgi_remvlan(9E), mgi_start(9E), mgi_stop(9E), mr_gget(9E)

NAME

mac_intr, **mac_intr_t** - MAC interrupt information

SYNOPSIS

#include <sys/mac_provider.h>

INTERFACE STABILITY

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

DESCRIPTION

The *mac_intr_t* structure is used by the MAC framework as part of the MAC_CAPAB_RINGS capability. For more background on the MAC framework, please see mac(9E).

The *mac_intr_t* structure is used to describe an interrupt and additional capabilities around it. The structure is usually used as part of another mac(9E) related structure such as the *mri_intr* member of the mac_ring_info(9S) structure.

This structure is used to describe the interrupt backing a given broader mac structure such as a ring. However, the functions present on it as they relate to an interrupt refer more to the broader structure's ability to generate the interrupt.

TYPES

Tye following types define the function pointers in use in the *mac_intr_t* structure.

```
typedef int     (*mac_intr_enable_t)(mac_intr_handle_t);
typedef int     (*mac_intr_disable_t)(mac_intr_handle_t);
```

STRUCTURE MEMBERS

```
mac_intr_handle_t       mi_handle;
mac_intr_enable_t       mi_enable;
mac_intr_disable_t      mi_disable;
ddi_intr_handle_t       mi_ddi_handle;
```

The *mi_handle* member should be set to a driver-specific value that will be passed back to the driver in various callback functions.

The *mi_enable* member is a required entry point for receive rings and optional for transmit rings. It should be set to a function which enables interrupts for the ring. For more information, see mi_enable(9E).

The *mi_disable* member is a required entry point for receive rings and an optional entry point for transmit rings. It should be set to a function which disables interrupts for the ring. For more information, see mi_disable(9E).

The *mi_ddi_handle* member should be set to the interrupt handle that corresponds to the ring. the interrupt handle will have come from ddi_intr_alloc(9F). This member should only be set if the interrupt is a MSI or MSI-X interrupt.

SEE ALSO

mac(9E), mac_capab_rings(9E), mi_disable(9E), mi_enable(9E), ddi_intr_alloc(9F), mac_ring_inf(9S)

NAME

mac_ring_info, **mac_ring_info_t** - MAC ring information structure

SYNOPSIS

#include <sys/mac_provider.h>

INTERFACE STABILITY

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

DESCRIPTION

The *mac_ring_info_t* structure is used by the MAC framework as part of the MAC_CAPAB_RINGS capability. For more background on the MAC framework, please see mac(9E).

When a device driver declares that it supports the MAC_CAPAB_RINGS cpability and fills out the structure as described in mac_capab_rings(9E), it indicates that it supports a number of rings for transmitting and receiving.

TYPES

The following types define the function pointers in use in the *mac_ring_info_t* structure.

```
typedef int     (*mac_ring_start_t)(mac_ring_driver_t, uint64_t);
typedef void    (*mac_ring_stop_t)(mac_ring_driver_t);

typedef mblk_t  *(*mac_ring_send_t)(mac_ring_driver_t, mblk_t *);
typedef mblk_t  *(*mac_ring_poll_t)(mac_ring_driver_t, int);

typedef int     (*mac_ring_stat_t)(mac_ring_driver_t, uint_t, uint64_t *);
```

STRUCTURE MEMBERS

```
uint_t                  mri_extensions;
uint_t                mri_flags;
mac_ring_driver_t  mri_driver;
mac_ring_start_t       mri_start;
mac_ring_stop_t        mri_stop;
mac_intr_t             mri_intr;
mac_ring_send_t    mri_tx;
mac_ring_poll_t     mri_poll;
mac_ring_stat_t        mri_stat;
```

The *mri_extensions* member is used to negotiate extensions between the MAC framework and the

device driver. The MAC framework will set the value of *mri_extensions* to include all of the currently known extensions. The driver should intersect this list with the set that the driver supports. At this time, no such features are defined and the driver should set this member to **0**.

The *mri_flags* member is used to indicate various additional capabilities supported by the ring. Currently, no flags are defined and the device driver should set it to **0**.

The *mri_driver* member should be set to a driver-specific value that represents the data structure that corresponds to the ring. The driver will receive this value in all of the callback functions that are defined in this structure.

The *mri_start* member is a required entry point that is used to start the ring. While the device driver may not need to do any work with hardware to start the use of the ring, it must record the ring's generation number. For more information, see mri_start(9E).

The *mri_stop* member is an optional entry point that will be called when the ring is being stopped. For more information, see mri_stop(9E).

The *mri_intr* member contains information about the interrupt associated with the ring. For more information on filling it out, see mac_intr(9S).

The *mri_tx* member should only be set on transmit rings. It must not be set on receive rings. The *mri_tx* member should be set to a function that will transmit a given frame on the specified ring. For more information, see mri_tx(9E).

The *mri_poll* member should only be set on receive rings. It must not be set on transmit rings. The *mri_poll* member should be set to a function which will poll the specified ring. For more information, see mri_poll(9E).

The *mri_stat* member should be set to a function which will retrieve statistics about the specified ring. For more information, see mri_stat(9E).

Required Members

All non-function members are required. The *mri_intr* member must be a properly filled out as per mac_intr(9S).

For transmit rings, the *mri_tx* member is required.

For receive rings, the *mri_poll* member is required.

SEE ALSO

mac(9E), mac_capab_rings(9E), mri_poll(9E), mri_start(9E), mri_stat(9E), mri_stop(9E), mri_tx(9E), mac_intr(9S)