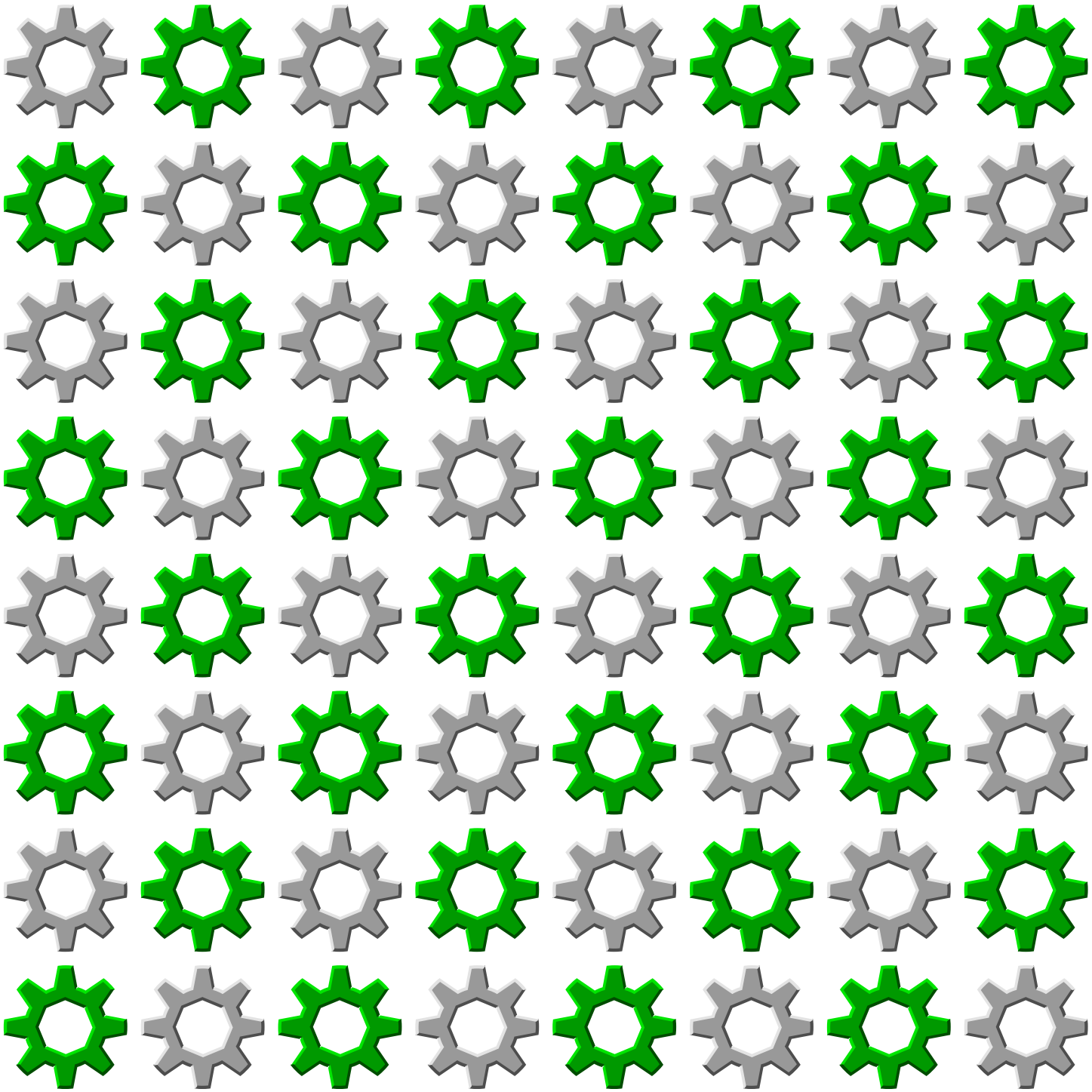
RISC OS DCI networking

Work In Progress



Gerph Publications

RISC OS DCI networking

Work In Progress

The authors have taken care to produce this document but make no warranty, express or implied as to the accuracy of the information presented here.

Contents

Network Device Drivers

Overview

1 History

See 'Document information' at the end of this document.

2 Unresolved Questions

There are no unresolved questions (at the moment).

3 Introduction

This document describes version 4 of the Device Control Interface ("DCI"), an interface between protocol modules and device driver modules in the RISC OS networking system.

3.1 Objectives

This new version is needed to overcome deficiencies/errors in the existing interface. The specific points it aims to address are:

- Full support for multiple protocol modules in a single system: older versions of the DCI notionally provided this support, but there were implicit features of the design which made support for more than one protocol module at any one time difficult.

- Support for multicast and promiscuous frame reception. Promiscuous reception is when an Ethernet interface receives **all** frames, regardless of their destination address; multicasting is a method used to transmit a single frame to multiple hosts simultaneously, it can be viewed as a form of limited broadcast: individual hosts on a network can choose whether or not they wish to receive multicast frames {The glossary of Internet terms in "Internetworking with TCP/IP" by Douglas Comer defines multicasting as "A technique that allows copies of a single [frame] to be passed to a selected subset of all possible destinations . . . . broadcast is a special form of multicast in which the subset of machines to receive a copy of a [frame] consists of the entire set."

- The need for improved data transfer rates compared to previous DCI versions, and the formal adoption of techniques already used to improve data throughput.

Backwards compatabilty

The radical changes and new features being introduced with this new version of the DCI, along with the inbuilt lack of flexibility in previous versions, combine to make any attempt at backwards compatibility impossible. With this lack of compatibility, care has been taken to ensure that there is no overlap between DCI 4 compliant modules, and other modules loaded on the same machine that implement earlier versions of the DCI, specifically:

1. In both old and new DCI versions, it is the responsibility of the protocol module to initialise the interface between itself and a device driver after either actively or passively learning of the device driver's presence; DCI 4 has replaced the active

(Service_FindNetworkDriver) and passive (Service_NetworkDriverStatus) service calls used by protocol modules with Service_EnumerateNetworkDrivers (on page 13) and Service_DCIDriverStatus (on page 14) respectively.

2. To prevent old device drivers getting confused by Service_ProtocolStatus, which no longer provides a sensible (as far as old DCI versions are concerned) value in R2, this service call has been made obsolete, and has been replaced by Service_DCIProtocolStatus (on page 17).

3.2 Principles of operation

The principle behind the interface is that protocol modules register a list of desirable frame types with network device drivers. When a device driver receives a frame, it passes it along to the protocol module that expressed an interest in the frame's type. Transmission is much simpler --- the protocol module passes the frame to be transmitted to the appropriate device driver. In both cases it is generally the recipient of the frame that assumes responsibility for the memory containing the frame (i.e. the protocol module for received packets, the device driver for transmitted packets).

Identifying Device Drivers

Device drivers are always identified by their "Driver Information Block", described on page 4. These Driver Information Blocks are used in a number of service calls, and are also given to protocol modules along with received frames.

There are fields within a Driver Information Block which uniquely identify each interface, but to prevent protocol modules having to make laborious (i.e. strcmp()) comparisons of these fields, device drivers should maintain a single, static, Driver Information Block for each interface it controls. In this way, protocol modules need only compare the address of Driver Information Blocks to identify an interface.

This scheme means that any use of the `*RMTidy` RISC OS command will kill any network stack on the machine --- this is not a great problem, since anyone who uses rmtidy in a modern RISC OS system is asking for all the trouble that they are about to receive. However, if a device driver module is re-initialised (via rmreinit), then the address of its Driver Information Block **will** change, therefore any protocol module's handler for the Service_DCIDriverStatus (on page 14) service call (page 6) cannot compare addresses, but must fall back to comparing those fields which uniquely identify an interface, i.e. dib_name & dib_unit.

3.3 Device Driver considerations

Important points for device driver writers to note are:

1. The DCI interface is optimised in various ways for Ethernet device drivers, specifically:

    1. Physical network addresses are 48-bit quantities.
    2. Protocol modules identify the physical network frames they wish to receive by the type of the frame.

    (For example, the Internet module claims frame types 0x800 (IP), 0x806 (ARP), and 0x8035 (RevARP). This is a 16-bit value transmitted as part of the Ethernet header.)

    Drivers for other types of network hardware will need to emulate an Ethernet driver at this interface by mapping "virtual Ethernet" values onto the real values meaningful to the network hardware.

2. At startup, driver modules must set the variable `Inet$EtherType` to the textual name of the controlled physical interface (e.g. "en", "ea"), with a suffix of `0'. This is for backwards compatibility with versions of Acorn's TCP/ IP Protocol Suite software already in the field. The textual name is the same string as the field dib_name in the Driver Information Block (see page 4 for a description of Driver Information Blocks). Note that this field variable reflects the **last** driver initialised, i.e. a driver will always set this field, regardless of whether or not it has been set previously.

4 Service Calls

As explained in the section on backwards compatibility (Section 3), the service calls defined in earlier versions of the DCI are all now obsolete. To summarise, these calls are

1. Service_ProtocolStatus
2. Service_FindNetworkDriver
3. Service_NetworkDriverStatus

The new service calls defined in DCI 4 are

1. Service_EnumerateNetworkDrivers (on page 13)
2. Service_DCIDriverStatus (on page 14)
3. Service_DCIFrameTypeFree (on page 16)
4. Service_DCIProtocolStatus (on page 17)

Note that the old, unnamed, service call 0x41200, which was never part of any formal DCI specification, but which used to be issued during finalisation of the Internet module, has now been officially replaced by Service_DCIProtocolStatus (on page 17).

4.1 Data Structures

Some user applications need to associate device drivers with the physical location of the network hardware, i.e. with which "slot" the hardware occupies. In order to support complex networking cards (e.g. one card with multiple, independent, interfaces), the concept of a slot is overloaded with a minor device number; the interpretation of this minor device number is device driver dependent.

Using C, a slot can be expressed as

```
struct slot
{
    unsigned int slotid:8,
    minor:8,
    pcmciaslot:5, /* must be zero if not a PCMCIA virtual slot */
    mbz:11; /* must be zero */
}
```

In this, and all other C code fragments, the standard Norcroft RISC OS compiler is assumed - with this example, this means that bitfields start at the least significant end of a word, i.e. `slotid` is bits 0-7, `minor` bits 8-15, and so on.

Bit(s)	Name	Meaning
0-7	slotid	8 bits: Physical location of the device
8-15	minor	8 bits: Minor field
16-20	pcmciaslot	5 bits: PCMCIA virtual slot, or 0 for non-PCMIA devices
21-31	mbz	Reserved, must be zero

Device drivers are free to interpret the minor field as they wish, but a typical use would be to discriminate multiple units on a single physical card. The pcmciaslot field is used to differentiate between cards in different PCMCIA slots (unfortunately, PCMCIA also uses the word "slot" to refer to the physical connection to a card). this field only has any significance when `slotid` is a PCMCIA virtual slot (see immediately below for a description of virtual slots).

As well as the physical expansion card slots, which now number from 0-8 with the introduction of the latest Acorn machines, there are also a number of "virtual" slots, i.e. network interfaces which don't use hardware in an expansion card slot. The list of physical and virtual slots can be summarised as

Value	Meaning
0-7	Physical expansion card slots
8	Risc PC network position
16-31	PCI slots
128	Parallel port
129	Serial port (e.g. PPP)
130	Econet socket
131	PCMCIA cards

Note:- there is only one PCMCIA virtual slot, this one virtual slot refers to PCMCIA hardware which may contain more than one physical PCMCIA slot; the pcmciaslot field within the slot number can be used by the device driver to differentiate between physical PCMCIA slots.

Driver Information Blocks

Device drivers identify themselves via a Driver Information Block, which can be expressed in C syntax as

```
struct dib
{
    unsigned int dib_swibase;
```

```
    char *dib_name;
    unsigned int dib_unit;
    unsigned char *dib_address;
    char *dib_module;
    char *dib_location;
    struct slot dib_slot;
    unsigned int dib_inquire;
};
```

The fields within this structure are:

Offset	Field	Contents
+0	dib_swibase	The base of the device driver's allocated SWI chunk.
+4	dib_name	A pointer to a short textual name unique to the driver (e.g. "en", "ppp"), and a terminating NULL.
+8	dib_unit	The unit number.
+12	dib_address	A pointer to a 6-byte character array which contains the hardware address of the interface.
+16	dib_module	A pointer to a string containing the title of the driver module (e.g. "Ether3").
+20	dib_location	A pointer to a string which attempts to describe the physical location of the interface. A typical string would be somewhere between 8 and 40 characters long and would be of the form "Network Expansion Slot", or "Expansion Slot 0, port #1" etc..
+24	dib_slot	The slot number for this unit.
+28	dib_inquire	A copy of the flags returned from the Inquire SWI (section 5.3).

Note that there is a subtle, but important, distinction between this definition of a Driver Information Block and its definition in previous versions of the DCI: the new definition has one Driver Information Block per unit, rather than one per device driver; if a device driver controls several units, then it must provide one struct dib per unit.

Units: a single device driver may control more than one physical network interface, either by driving multiple network cards, or by driving multiple interfaces on a single card. The driver is responsible for allocating a number to each interface under its control --- the first interface being unit 0, the second interface being unit 1, and so on. Any particular network connection can then be uniquely identified by its driver name and unit number, e.g. en0, ea2. If an interface is found to be faulty during any hardware check its driver may perform, the interface **must** still be assigned a unit number, and must still

appear in the enumerated list of device drivers.

Chained Driver Information blocks

The results from the Service_EnumerateNetworkDrivers (on page 13) service call are chained together into a linked list of Driver Information Blocks. The C structure used for this linked list is

```
struct chaindib
{
    struct chaindib *chd_next;
    struct dib *chd_dib;
};
```

Just in case the fields within this structure are not self-evident, they are:

Offset	Field	Contents
+0	chd_next	A pointer to the next entry in the linked list. The last entry in the list contains a NULL pointer.
+4	chd_dib	A pointer to the Driver Information Block for this entry in the linked list.

Protocol Information Blocks

In much the same way that device drivers are identified by their Driver Information Block, older versions of the DCI used to contain a Protocol Information Block which identified individual protocol modules. This Protocol Information Block is not needed in DCI 4, and has been removed.

4.2 Obsolete services

As already mentioned in section 4, the old, unnamed, service call 0x41200 will not be issued by DCI 4 compliant versions of the Internet module when it is terminating.

4.3 Device Driver Initialisation

When a device driver module initialises, it is expected to issue a Service_DCIDriverStatus (on page 14) service call to announce that it is starting; at the time this startup call is issued, the module must also be capable of handling SWIs raised by any protocol module interested in the device driver. Unfortunately, the RISC OS kernel does not recognise a module's SWIs until **after** its initialisation routine has returned, which means that driver modules must take explicit steps to allow SWIs to be caught before issuing the service call, i.e. they must either:

1. Install a handler on the unknown SWI software vector ("UKSWIV"), and check all unknown SWIs for an appropriate chunk number.
2. Setup a callback handler in the initialisation routine, and then issue the service call from within this callback handler.

Service_EnumerateNetworkDrivers (Service Call &9B)

List all active network drivers in the system

On entry

R0 = pointer to head of linked list of device drivers
R1 = &9B (reason code)

On exit

R0 = pointer to new head of linked list
R1 - R9 preserved

Use

This service call is used to obtain a list of all active network device drivers in the system. When the service call is issued, R0 is a NULL pointer; upon receipt of this call, a network device driver should chain Driver Information Blocks to the head of the list, one for each logical interface the driver controls. Section 4.1 describes the struct chaindib used to hold the linked list of Driver Information Blocks.

This service call should never be claimed.

Note: Struct chaindibs are transient objects: they should be allocated from RMA by the device drivers, and freed back into the RMA by the protocol module which issued the service call. The Driver Information Blocks referenced by the struct chaindibs must be static data, as explained in section 3.2.

Related APIs

None

Service_DCIDriverStatus (Service Call &9D)

Announce initialisation and finalisation of driver

On entry

R0 = pointer to Driver Information Block describing this driver
R1 = &9D (reason code)
R2 = Driver status:

Value	Meaning
0	Starting
1	Terminating

R3 = DCI version supported × 100

On exit

R0 - R9 preserved

Use

Service_DCIDriverStatus is issued by a network driver module during its initialisation (R2 = 0), and finalisation (R2 = 1) calls. If a network device driver controls multiple logical interfaces, then a separate service call must be issued for each interface the driver is responsible for.

Upon receipt of this service call from a driver that is starting up (i.e. R2 = 0), a protocol module should add the driver

to its list of known device drivers. If a service call is received from a driver that is terminating, the protocol module should scan its list of known device drivers for a Driver Information Block matching the one addressed by R0, removing it from the list if a match is found. A Driver Information Block is uniquely identified by its dib_name and dib_unit fields, therefore a comparison of these two fields is sufficient to prove a match.

The supported DCI version passed in R3 is in the same format as described for the DCIVersion SWI (Section 5.3), i.e. 405 decimal for this version.

When this call is issued while starting, device drivers should be able to receive SWIs raised by protocol modules; this means that the service call cannot be directly issued from a driver's initialisation routine --- see section 4.3 for an explanation of this feature, along with ways to overcome it.

When this call is issued during finalisation, protocol modules should not expect to be able to issue SWIs, therefore no special action to allow this is required of the driver module issuing the service call.

This service call should never be claimed.

Related services

Service_DCIProtocolStatus (on page 17)

Service_DCIFrameTypeFree (Service Call &9E)

Announce that a frame type has been freed by a driver

On entry

R0 = pointer to Driver Information Block describing this driver
R1 = &9E (reason code)
R2 = frame type being released
R3 = address level of former claim
R4 = error level of former claim

On exit

R0 preserved
R1 = 0 to claim the call, or preserved to pass it on.
R2 - R9 preserved

Use

This service call is issued by a device driver when a protocol module releases a claim it formerly had on a frame type (i.e. when the frame type becomes free for claiming by a different protocol module). This release may have been either explicit (the protocol module called the SWI DCIDriver_Filter (on page 30) SWI), or implicit (the protocol module issued a Service_DCIProtocolStatus (on page 17) service call).

If a protocol module wishes to claim the newly relinquished frame type for itself, it should use the Filter SWI to do so, and then claim the service call by setting R1 to 0.

Note: the concepts of frame types and the various filtering levels available are explained fully in section 6.2.

Related SWIs

SWI DCIDriver_Filter (on page 30)

Related services

Service_DCIDriverStatus (on page 14)

Service_DCIProtocolStatus (Service Call &9F)

Announce initialisation and finalisation of protocol

On entry

R0 = Protocol module's private word pointer
R1 = &9F (reason code)
R2 = Protocol status:

Value	Meaning
0	Starting
1	Terminating

R3 = DCI version supported × 100
R4 = Pointer to protocol module's title string

On exit

R0 - R9 preserved

Use

Service_DCIProtocolStatus is issued by a protocol module during its initialisation (R2 = 0), and finalisation (R2 = 1) calls. The private word pointer in R0 is the same as that supplied by the protocol module in the SWI DCIDriver_Filter (on page 30) SWI (see section 5.3).

The supported DCI version passed in R3 is in the same format as described for the SWI DCIDriver_DCIVersion (on page 23) SWI, i.e. 405 decimal for this version.

The title string pointed to by R4 should be identical to the title string in the protocol module's header. This string is not used anywhere else in the DCI --- it is intended for use by modules that rely on the protocol module, but which do not communicate with it via the DCI; these modules need to have the name of significant protocol modules built into them.

As with device drivers issuing Service_DCIDriverStatus (on page 14), protocol modules should already be capable of handling any SWIs at the time they issue this service call to announce that they are starting; the techniques described in section 4.3 are as equally valid for protocol modules as they are for device drivers.

When terminating, the protocol module which issued this service call must be prepared to handle receive events for all frame types it has

not explicitly relinquished until the service call returns; once the call has returned, device drivers should have deleted **all** references to the protocol module which issued the service call. If necessary, device drivers may enable interrupts while processing the service call, but they should return with the interrupt state preserved.

Device drivers must never claim this service call.

Related services

Service_DCIDriverStatus (on page 14)

5 Device Driver SWIs

All network device drivers must provide a SWI call interface which protocol modules can use to

- send control commands
- pass data
- obtain information

to/from a device driver. Obviously, each device driver will have its own, unique, SWI chunk, so a protocol module must use the dib_swibase field from the Driver Information Block (see section 4.1) to determine the base of a driver's SWI chunk.

The SWI calls that a device driver must supply (and their offsets) are

SWI offset	SWI name
0	DCIVersion
1	Inquire
2	GetNetworkMTU
3	SetNetworkMTU
4	Transmit
5	Filter
6	Stats
7	MulticastRequest

Re-entrancy

All device driver SWIs are potentially re-entrant, i.e. the protocol modules are not expected to take any explicit action to prevent re-entrance. If re-entrancy is undesirable during the processing of a SWI, then the device driver should take explicit steps to prevent this, i.e. by disabling interrupts.

In order to minimise interrupt latency within the machine as a whole, device drivers should take steps to minimise the length of time during which interrupts are disabled.

Byte Sex

All data passed between the protocol modules and device drivers use the host byte sex, i.e. little-endian, whereas all data transmitted over the wire are (obviously) in network, or big-endian, byte sex. Device drivers are responsible for converting data to the appropriate sex.

5.1 Errors

All the SWI descriptions given later in this section ignore what will happen if the SWI needs to return an error. Obviously, there will be circumstances in which it is necessary to return an error from a SWI call, and the standard RISC OS mechanism is used, i.e. the device driver will set the V flag, and return with R0 pointing to an error block. Apart from the unavoidable corruption of R0, all other registers which are declared as being preserved by the SWI are still preserved, even when an error is returned.

Error numbers usually equate to Unix error numbers, as defined in the standard header file "errno.h"; these numbers are always less than 128, and are converted into offsets within the standard error block that has been defined for DCI 4 and Internet. This error block starts at &20E00, for example, the error EINVAL (invalid argument, defined as 22, &16) would be returned as error number &20E16. There are certain circumstances (e.g., the SWI DCIDriver_Transmit (on page 28) SWI indicating that transmission is blocked) where an appropriate Unix error number does not exist --- in this situation, a custom error number is defined specifically for this one error condition.

Standardised Errors

In an attempt to force some consistency, this sub-section defines some of the errors which various SWIs, and the circumstances in which these errors may be returned. This is not meant to be an exhaustive list, merely to cover all the errors explicitly mentioned in this document, plus some other common faults.

- Any SWI:

Error number	Unix error	Meaning
&20E16	EINVAL	Incorrect flags word in R0
&20E06	ENXIO	Invalid unit number supplied

- SWI DCIDriver_SetNetworkMTU (on page 27):

Error number	Unix error	Meaning
&20E19	ENOTTY	Illegal op for device

- SWI DCIDriver_Transmit (on page 28):

Error number	Unix error	Meaning
&20E86		Transmission is blocked
&20E32	ENETDOWN	Network hardware is down.
&20E28	EMSGSIZE	Frame length > network MTU.
&20E37	ENOBUFS	Not enough mbufs available.

- SWI DCIDriver_Filter (on page 30):

Error number	Unix error	Meaning
&20E87		Frame type already claimed
&20E16	EINVAL	Trying to claim illegal frame type
&20E16	EINVAL	Trying to release a non-existent claim.
&20E01	EPERM	Trying to free another protocol's claim.

- SWI DCIDriver_MulticastRequest (on page 33):

Error number	Unix error	Meaning
&20E16	EINVAL	Trying to claim illegal frame type
&20E16	EINVAL	Trying to release a non-existent claim.

5.2 Changes in DCI 4

The device driver SWI interface has been given an extensive overhaul for DCI 4: the complete break between this, and older versions of the DCI (as explained in the section about backwards compatibility 3.1) mean that **all** SWIs, even SWI DCIDriver_DCIVersion (on page 23) can be altered without worrying about the impact on non-DCI 4 modules within a machine.

The major changes made for DCI 4 are:

1. All SWIs now use R0 as a flag word: this provides an easy route to alter SWI functionality in any future versions of the DCI that prove to be necessary. All bits of this flag word should be set to zero, except where explicitly stated otherwise.
2. Several new SWIs have been added, i.e.

   - SWI DCIDriver_SetNetworkMTU (on page 27)
   - SWI DCIDriver_Inquire (on page 24)
   - SWI DCIDriver_Filter (on page 30)
   - SWI DCIDriver_Stats (on page 32)
   - SWI DCIDriver_MulticastRequest (on page 33) (as of

DCI 4.04)

3. SWI NetworkMTU has been renamed to SWI DCIDriver_GetNetworkMTU (on page 26), to differentiate it from the new SWI SWI DCIDriver_SetNetworkMTU (on page 27).
4. SWI NetworkIfSend has been renamed to SWI DCIDriver_Transmit (on page 28), mainly because it is less of a mouthful.
5. Some old SWIs have been deleted (see below).
6. Offsets of SWIs within a driver's SWI chunk have been changed to fill in the gaps left by the deleted SWIs; for example, SWI DCIDriver_DCIVersion (on page 23) has been moved from offset 4 to the more logical offset of 0.

Deleted SWI Details

As mentioned above, some of the SWIs from earlier versions of the DCI have been removed from DCI 4. These SWIs are:

SWI name	Meaning
NetworkIfStart	The decision on whether network hardware should be enabled lies with the device driver, not with a protocol module (consider the situation where one protocol module believes that the hardware should be enabled, while a different module is of the opinion that it should be disabled). As far as protocol modules are concerned, they can only reasonably expect the hardware to be enabled when they have declared an interest in one or more frame types; if they have no declared interests, then whether the hardware is enabled or not is of no significance to them.
NetworkIfUp	this SWI has been removed for the same reasons as NetworkIfStart (q.v.).
NetworkIfDown	this SWI has been removed for the same reasons as NetworkIfStart (q.v.).
TxEventRequired	DCI 4 no longer uses events to communicate between device drivers and protocol modules, therefore this SWI has become redundant.

DCIDriver_DCIVersion (SWI DCIDriver+&00)

Returns DCI version numbers supported by the device driver

On entry

R0 = Flags (all bits must be zero)

On exit

R0 preserved
R1 = Supported DCI version number (this version = 405 decimal)

Interrupts

Interrupts are undefined
Fast interrupts are undefined

Processor mode

Processor is in undefined mode

Re-entrancy

Not defined

Use

Returns DCI major and minor version numbers supported by the device driver. The supported DCI version number is calculated as (major version × 100) + minor version.

Note: earlier versions of the DCI only returned the major version, i.e. 1 or 2 (as opposed to 100 or 200).

There was no formal version 3 of the DCI.

DCIDriver_Inquire (SWI DCIDriver+&01)

Read the characteristics for the device driver

On entry

R0 = Flags (all bits must be zero)
R1 = Unit number

On exit

R0 - R1 preserved
R2 = Bitmap of supported features (see below)
R3 - R9 preserved

Interrupts

Interrupts are undefined
Fast interrupts are undefined

Processor mode

Processor is in undefined mode

Re-entrancy

Not defined

Use

This SWI is used to ascertain the characteristics of a device driver.
The flag bits within R2 are:

Bit(s)	Meaning
0	Multicast reception is supported
1	Promiscuous reception is supported
2	Interface receives its own transmitted packets
3	Station number required.
4	Interface can receive erroneous packets
5	Interface has a hardware address
6	Driver can alter interface's hardware address
7	Interface is a point to point link
8	Driver supplies standard statistics
9	Driver supplies extended statistics
10	This is a virtual interface
11	This virtual interface is software based
12	This interface can selectively receive multicast packets (ie SWI MulticastRequest is available)
13	This interface can IP checksum frames
14-31	Reserved, must be zero

Most of these flags are self-explanatory; "Station number required" (bit 3) is used by AUN software to find out whether the underlying network requires a fixed "pseudo-Econet" station number (i.e. set in CMOS RAM), or whether a dynamic station number allocation mechanism can be employed. For example, physical Econet requires a fixed station number and its driver should set bit 3 of the flags, but Ethernet does not, and any such driver should leave bit 3 clear.

The concept of virtual interfaces (bits 10 and 11) is explained in section 9.2.

The driver should only set bit 13 if it can checksum frames efficiently, either in hardware or while copying the data into mbufs. If it would involve an extra pass over memory, it should leave the checksum up to the protocol module. See section 6.3 for more details.

Some of these characteristisc are inter-related, specifically:

- If bit 5 (interface has a hardware address) is not set, then bit 6 (driver can alter hardware address) is ignored.
- A driver cannot supply extended statistics (bit 9), without also supplying standard statistics (bit 8).
- If bit 11 (virtual interface is software based) is set, then bit 10 (this is a virtual interface) should always be set as well.

DCIDriver_GetNetworkMTU (SWI DCIDriver+&02)

Return the MTU for the unit

On entry

R0 = Flags (all bits must be zero)
R1 = Unit number

On exit

R0 - R1 preserved
R2 = MTU
R3 - R9 preserved

Interrupts

Interrupts are undefined
Fast interrupts are undefined

Processor mode

Processor is in undefined mode

Re-entrancy

Not defined

Use

This SWI returns the MTU (Maximum Transmission Unit) for the unit specified in R0. Ethernet has a fixed MTU of 1500 bytes, other hardware layers (e.g. PPP) may have a variable MTU.

Note: this SWI has changed with respect to earlier versions of the DCI, in as much as

1. There is now the standard flags word in R0.
2. A unit number is now passed in R1.
3. Results are returned in R2.
4. A default return (R2 = 0), implying Ethernet MTU is no longer supported.

Related SWIs

SWI DCIDriver_SetNetworkMTU (on page 27)

DCIDriver_SetNetworkMTU (SWI DCIDriver+&03)

Set the MTU for the unit

On entry

R0 = Flags (all bits must be zero)
R1 = Unit number
R2 = MTU

On exit

R0 - R9 preserved

Interrupts

Interrupts are undefined
Fast interrupts are undefined

Processor mode

Processor is in undefined mode

Re-entrancy

Not defined

Use

For those device drivers that allow it (e.g. PPP), this SWI sets the Maximum Transmission Unit for the unit given in R1.

If the device driver has an immutable MTU, then it must still support this SWI, but return an error indicating an illegal

operation.

Note: protocol modules can only ever consider this MTU as a guideline - other protocol modules may set a

different MTU for the same logical unit.

Related SWIs

SWI DCIDriver_GetNetworkMTU (on page 26)

DCIDriver_Transmit (SWI DCIDriver+&04)

Request the driver send frames on the network

On entry

R0 = Flags
R1 = Unit number
R2 = Frame type
R3 = Pointer to mbuf chains containing data to transmit
R4 = (byte aligned) pointer to destination hardware address
R5 = (byte aligned) pointer to source hardware address (if applicable)

On exit

R0 - R9 preserved

Interrupts

Interrupts are undefined
Fast interrupts are undefined

Processor mode

Processor is in undefined mode

Re-entrancy

Not defined

Use

This SWI is a request from the protocol module for the device driver to send the packet addressed by R3 to the hardware address specified in R4. The "frame type" passed in R2 is something of a misnomer: the value given is copied into the last 2 bytes of an Ethernet frame header, i.e. it is the length field according to the IEEE 802.3 spec., and the frame type as far as Ethernet 2.0 is concerned.

If a previous frame is still being transmitted, the driver should queue the new request if possible, otherwise return an error indicating that transmission is blocked.

The flag bits within R0 are:

Bit(s)	Meaning
0	Clear: Use interface's own hardware address.
	Set: Use address given by R5 for source hardware address.
1	Clear: Device driver assumes ownership of memory resources
	Set: Protocol module retains ownership of memory resources
2-31	Reserved, must be zero

Regardless of who initially allocated the memory resources (i.e. mbuf chains) passed in R3, it is the new owner of these resources (i.e. the device driver if R0, bit1 = 0; the protocol module if R0, bit 1 = 1) that is responsible for returning these resources to the free pool when they are no longer needed.

This SWI uses the scheme, described in section 6.3 for linking several received mbuf chains, to pass multiple output chains to the device driver via asingle call to this SWI. Care must be taken to ensure that the flag bits in R0 are applicable to **all** , mbuf chains passed to the driver.

The data passed to the driver can be either "safe", or "unsafe" (section 8.3 explains the concept of unsafe data) --- if the device driver is given ownership of memory resources, and needs to keep these resources after the Transmit SWI has finished, then it must use the ensure_safe function of the memory manager (section 8.2) to obtain a safe copy of the data.

Note: the register numbers for this call have changed from earlier versions of the DCI, this change being made in an attempt to standardise register usage as far as possible.

DCIDriver_Filter (SWI DCIDriver+&05)

Register a request for network frames

On entry

R0 = Flags
R1 = Unit number
R2 = Frame type
R3 = Address level (for write)
R4 = Error level (for write)
R5 = Private word pointer
R6 = Address of handler routine for received frames

On exit

R0 - R9 preserved

Interrupts

Interrupts are undefined
Fast interrupts are undefined

Processor mode

Processor is in undefined mode

Re-entrancy

Not defined

Use

This SWI is the mechanism by which protocol modules inform device drivers which Ethernet frame types they would like to be passed. A full description of this interface is provided in section 6.2.

The flag bits within R0 are:

Bit(s)	Meaning	
0	Clear:	Claim a frame type.
	Set:	Release a previous claim on the frame type.
1	Clear:	Device drivers can pass unsafe mbuf chains to the receive handler
	Set:	Device drivers should ensure_safe mbuf chains before passing them to the receive handler.
2	Clear:	The protocol module wants all multicast frames (if indicated by R3)
	Set:	The protocol module will ask for specific multicast frames.
3	Clear:	IP checksum not required
	Set:	Device driver should place the IP checksum of received frames into the RxHdr.
4-31	Reserved, must be zero	

The private word pointer passed in R5 is the address of the protcol module's private word, which itself contains the address of the module's workspace.

This pointer is passed round in R0 by the protocol module in the Service_DCIProtocolStatus (on page 17) service call.

When a device driver receives a network frame of a type claimed by a protocol module, it will call the routine given in R6. Section 6.3 describes the parameters which must be passed to this received frame handler.

The concept of safe and unsafe data, as used in bit 1 of the flags is explained in section 8.3.

This SWI should return an error when:

- An illegal frame type is claimed.
- A frame type is already claimed. If a protocol module receives this error from a device driver, it can use Service_DCIFrameTypeFree (on page 16) to learn when the frame type is again free for claiming.
- An attempt is made to free a frame type which has not been previously claimed by the protocol module.

Related SWIs

SWI DCIDriver_MulticastRequest (on page 33)

DCIDriver_Stats
(SWI DCIDriver+&06)

Register a request for network frames

On entry

R0 = Flags
R1 = Unit number
R2 = Pointer to buffer for holding results

On exit

R0 - R9 preserved

Interrupts

Interrupts are undefined
Fast interrupts are undefined

Processor mode

Processor is in undefined mode

Re-entrancy

Not defined

Use

This SWI is the mechanism by which device drivers return statistics they have gathered while running. A full description of these statistics, including the structure copied into the the addressed by R2 is provided in section 7.

The flag bits within R0 are:

Bit(s)	Meaning
0	Clear: Return an indication of which statistics are gathered.
	Set: Return the statistics themselves.
1-31	Reserved, must be zero

The buffer addressed by R2 **must** be large enough to hold the full statistics structure, i.e. at least 100 bytes long; the driver is free to copy that many bytes into the buffer without thought for the consequences if the buffer is too small.

DCIDriver_MulticastRequest (SWI DCIDriver+&07)

Manage multicast addresses received by the driver

On entry

R0 = Flags
R1 = Unit number
R2 = Frame type
R3 = (byte aligned) pointer to multicast hardware (MAC) address
R4 = (word aligned) pointer to multicast logical address (eg pointer to IP address for frame type 0x800)
R5 = private word pointer
R6 = address of handler routine for received frames

On exit

R0 - R9 preserved

Interrupts

Interrupts are undefined
Fast interrupts are undefined

Processor mode

Processor is in undefined mode

Re-entrancy

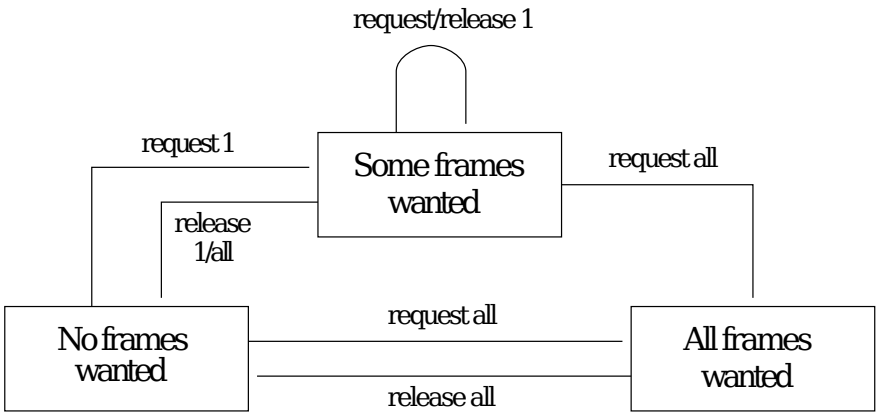
Not defined

Use

This SWI is the mechanism by which protocol modules specify which destination multicast addresses they wish to receive.

The flag bits within R0 are:

Bit(s)	Meaning	
0	Clear:	Request a multicast address
	Set:	Release a multicast address
1	Clear:	Requesting/releasing specific multicast address (as specified by R3,R4)
	Set:	Requesting/releasing all multicast addresses (R3 and R4 irrelevant) - these operations supersede specific multicast address operations (see state diagram below)
2-31	Reserved, must be zero	



How to interpret request/release calls when in a given state

If a protocol module calls the SWI DCIDriver_Filter (on page 30) SWI with bit 2 of R0 clear, then it will receive all multicast frames (if the address level is multicast or promiscuous). If, however, it sets bit 2 of R0, and the address level is multicast, then it will initially receive no frames (usually -- see below); to start to receive certain multicasts, it should call this SWI. R3 will point to a MAC address -- Ethernet drivers will use only this. Non-Ethernet drivers will probably need to know what logical address is being requested, as there may not be a one-to-one mapping between the logical and hardware multicast addresses for the specified frame type (as indeed there isn't for IP/Ethernet).

Contact Acorn for details of what to pass in R4 for specific frame types. ⚠ FIXME: Contact who now?

R1, R2, R5, and R6 must match the values passed into the Filter SWI so that the device driver can tell which filter this call is intended for.

It is not expected that the device driver will do software filtering of multicasts (beyond ensuring that specific and broadcast filters don't receive any multicasts); this is up to the protocol modules. The intention of this SWI is that it should be used to set up hardware filtering where possible; protocol modules may receive more

multicasts than they requested. For example, if one protocol module is using selective multicasts, while another, older protocol module isn't, the selective module will probably end up receiving all multicasts because the hardware filtering will have had to be switched off for the unselective protocol module.

This actually aids compatibility with DCI 4.03 driver modules -- a new protocol module need only set bit 2 of R0 when calling SWI DCIDriver_Filter (on page 30), then ignore any "SWI not known" errors from the MulticastRequest SWI. It will then work fine with older drivers.

Device drivers will need to track which filters are requesting which multicast addresses, so that when a filter is released or a protocol module dies all its multicast claims can be automatically removed. However, as specified above, there is no need to check whether a multicast filter has requested a specific multicast address before passing a received frame to it.

This SWI provides no function for filters with an address level other than ADDRLVL_MULTICAST, and if called for such a filter should return EINVAL.

Related SWIs

SWI DCIDriver_Filter (on page 30)

6 Received Frames

One of the major changes between DCI 4 and earlier DCI versions is the scheme used for handling received frames. The main changes introduced with this version are:

1. Support for multicast and promiscuous frames.
2. Improved handling of IEEE 802.3 format frames.
3. Protocol modules are informed of received frames with a direct call into a handler routine, rather than via an event.

The principle of operation is that protocol modules register an interest in one or more frame types with a device driver, defining various filtering parameters in the process. When a device driver receives a network frame, it uses the frame type and filtering parameters to decide which (if any) protocol module should be passed the frame. Any one received frame can be passed to either one or no protocol modules, it is not possible for a single frame to be given to multiple protocol modules.

6.1 Frame Class --- Ethernet 2 and IEEE 802.3

All Ethernet frames have a 14 byte MAC header: 6 bytes of destination hardware address, 6 bytes of source hardware address, and 2 more bytes. Unfortunately, there are two competing "standards" which place a different interpretation on these last 2 bytes: Ethernet 2.0, which considers them as 16 bits of frame type, and IEEE 802.3 which treats them as 16 bits of frame length.

It is obviously not possible to refer to Ethernet 2.0 and IEEE 802.3 as different types of frame, so this document uses the term "class" to refer to the property of being either an Ethernet 2.0, or an IEEE 802.3 frame.

Since all Ethernet frames must be no more than 1500 bytes long, a device driver should assume that any received frame with an Ethernet 2.0 "frame type" of 0--1500 is an IEEE 802.3 frame, and that everything else is an Ethernet 2.0 frame. (Note that although Ethernet frames should be padded to a minimum length of 46 bytes, frame lengths < 46 are still legal values).

All IEEE 802.3 class frames should also conform to the IEEE 802.2 standard for Logical Link Control --- this latter standard defines a set of services to be supported, and provides a method to identify the type of an 802.3 class frame; the implementation of this IEEE 802.2 Logical Link Control layer cannot be the responsibility of any specific protocol module, and it would be inefficient to make each device driver responsible for the implementation, so DCI 4 caters for the scheme shown in figure 1.

Note: It cannot be protocol modules for two reasons:

- Protocol modules are frame type specific, whereas the standard services which an IEEE 802.2 implementor must provide are frame type independent.
- The software that implements the IEEE 802.2 layer will be expected to filter frame types, and pass them along to protocol modules; therefore, obviously, this software cannot be a standard protocol module itself.

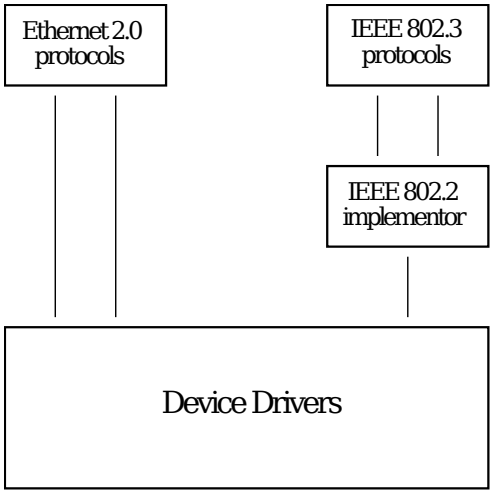


Figure 1: Filtering Ethernet 2.0 and IEEE 802.3 class frames

In this scheme, device drivers can differentiate between the two frame classes, and, furthermore, can distinguish Ethernet 2.0 frame types. However, no effort is made to ascertain frame types for IEEE 802.3 frames, and **all** frames of this class are passed to a pseudo-protocol module which implements the IEEE 802.2 Logical Link Control layer, and which provides a similar interface to DCI 4, allowing IEEE 802.3 protocol modules to claim specific frame types.

6.3 Frame Filtering

A protocol module uses SWI DCIDriver_Filter (on page 30) to identify a number of criteria which a received frame must match before being passed by the device driver to the protocol module; these criteria are

- Frame type
- Address level
- Error level

Only one protocol module is allowed to claim any given frame type, and when claimed, that frame type is **never** passed to any other protocol module. For example, if one protocol module has claimed a frame type with an address filter of specifically addressed packets only, then a second protocol module:

- cannot claim the same frame type with an address level of promiscuous.
- **can** claim all frame types not specifically registered, with (e.g.) an address level of multicast, but will **not** be passed any broadcast frames of the type claimed by the first protocol module (which will not receive the frame either, because the address level will filter out broadcast packets).

Frame Type

DCI 4 splits the 32-bit frame type into two 16-bit subfields --- the hi-order 16 bits specify the frame class and level, while the lo-order 16 bits provide the exact frame type (where significant).

Expressed in C format, the class/level subfield can take the following values:

```
#define FRMLVL_E2SPECIFIC 0x0001
#define FRMLVL_E2SINK 0x0002
#define FRMLVL_E2MONITOR 0x0003
#define FRMLVL_IEEE 0x0004
```

All other values for this subfield are illegal --- any attempt to use them in a SWI DCIDriver_Filter (on page 30) SWI should generate an error; similarly, if the hi-order subfield of the frame type is FRMLVL_E2SPECIFIC, then the lo-order subfield can take any value from 0x0000 -- 0xffff, otherwise it must be set to 0x0000, and any other value passed to Filter should be treated as an error.

The precise meanings of the class/level subfield values are:

Value	Level	Meaning
1	Specific	this is the standard frame level filter --- the protocol module is only passed Ethernet 2.0 frames whose type match that given in the lo-order, frame type subfield.
2	Sink	pass all Ethernet 2.0 frames that are not explicitly claimed by any protocol module.
3	Monitor	pass **all** Ethernet 2.0 frames to the protocol module.

For Ethernet 2.0 frames, the table below gives a summary of what frame levels are allowed on new claims, given the highest level of filtering currently active (monitor is considered higher than sink, and both of these levels are considered higher than normal)

Highest Current Level	New Levels Allowed
(Nothing)	Normal, Sink, Monitor
Normal	Normal, Sink
Sink	Normal
Monitor	(Nothing)

Address Level

The four levels of address level filtering can be expressed in C as

```
#define ADDRLVL_SPECIFIC 0
#define ADDRLVL_NORMAL 1
#define ADDRLVL_MULTICAST 2
#define ADDRLVL_PROMISCUOUS 3
```

These levels are:

Value	Level	Meaning
0	Specific	only pass frames addressed to the interface's specific hardware address.
1	Normal	only pass frames addressed to the interface's specific hardware address, and broadcast frames.
2	Multicast	pass all specifically addressed, broadcast and multicast frames. If bit 2 of R0 was set on entry to the SWI DCIDriver_Filter (on page 30) SWI, and the SWI DCIDriver_Inquire (on page 24) SWI returns with bit 12 set, then the driver should attempt to filter multicast frames -- see the SWI DCIDriver_MulticastRequest (on page 33) SWI for details. Otherwise, all multicast frames will be passed.
3	Promiscuous	pass all frames of the appropriate frame type, with no address matching at all.

Most Ethernet controllers can perform this address filtering at a hardware level, but, obviously, the hardware needs to be configured to the loosest level of filtering requested by any protocol module. In the situation where two protocol modules have specified two different levels of address filtering, the device driver must still filter out unwanted frames; protocol modules are **only** responsible for filtering out any unwanted subset of multicast frames.

Error Level

A device driver should provide two levels of error filtering, in C these are

```
#define ERRLVL_NO_ERRORS 0
#define ERRLVL_ERRORS 1
```

These levels are:

Value	Field	Meaning
0	ERRLVL_NO_ERRORS	only pass frames that are received error free.
1	ERRLVL_ERRORS	pass all frames, regardless of error state.

6.3 Received Frame Handlers

A major difference between DCI 4 and earlier versions of the DCI is the method used to notify protocol modules of the arrival of frames in which they have registered an interest. The main features of these receive handlers are

1. Device drivers call a direct entry point within the protocol module (earlier DCI versions used a receive event). The address of this direct entry point is passed to the device driver at the same time the frame type is claimed via the SWI DCIDriver_Filter (on page 30) SWI .
2. A device driver can pass several received frames to the protocol module with one call to the receive handler, rather than having to call the protocol module once per frame.
3. The protocol module becomes the new owner of all mbufs passed to its receive handler by device drivers: it is the protocol module that is responsible for freeing all resources once they are no longer needed.

Handler Details

On entry:

R0 = pointer to Driver Information Block describing the source interface

R1 = pointer to head of mbuf list of received frames

R12 = protocol module's private word pointer, i.e. value passed in R5 to SWI DCIDriver_Filter (on page 30) SWI

On exit:

All registers preserved.

Interrupt Status: Both interrupts and fast interrupts are enabled by the received frame handler.

Details of the exact structure of the mbuf list of received frames are given in the section below.

Each received frame has a header which can be described in terms of the following C structure:

```
struct rx_hdr
{
    void *rx_ptr;
    unsigned int rx_tag;
    unsigned char rx_src_addr[6], _spad[2];
    unsigned char rx_dst_addr[6], _dpad[2];
    unsigned int rx_frame_type;
    unsigned int rx_error_level;
    unsigned int rx_cksum;
}
```

The fields in this structure are:

Offset	Field	Contents
+0	rx_ptr	This field is for internal use by the receive handler, its value is undefined upon entry.
+4	rx_tag	This field is reserved for use by the IEEE 802.2 implementor, and must be set to zero by the device driver.
+8	rx_src_addr	The hardware source address of the frame. (Must be zeroed if hardware addresses not supported)
+14	_spad	Space filler to align the next field (dst_addr) on a word boundary. Must be zero filled.
+16	rx_dst_addr	The hardware destination address of the frame. (Must be zeroed if hardware addresses not supported)
+22	_dpad	Space filler to align the next field (frame_type) on a word boundary. Must be zero filled.
+24	rx_frame_type	The length (for IEEE 802.3), or the type (for Ethernet 2.0) of the received frame, i.e. the last 2 bytes of the frame's MAC header.
+28	rx_error_level	This field is zero if the frame was received with no errors, otherwise it contains a driver specific error code.
+32	rx_cksum	If bit 3 was set in the Filter call, this field must be filled in with the checksum of the complete frame. The checksum algorithm is that used by IP, ie the one's complement of the one's complement sum of all 16 bit words in the frame (padded with a zero-byte at the end if necessary to make a multiple of two bytes). Note that no knowledge of the IP packet format is required; in particular the complete frame should be checksummed, not just the length specified in the IP header. Furthermore, the frame need not even be an IP packet - other protocols may use the same checksum. The rx_cksum field is 32 bits wide only for ease of access - the most significant two bytes must be zero.

This frame header is passed in the first mbuf of each frame, the first byte of the frame data is in the second mbuf in the chain.

Note that DCI versions before 4.05 did not have the rx_cksum field; similarly later versions may have extra fields. Hence protocol modules must not fault unexpectedly long header mbufs; nor should they fault short headers, unless they were relying on the extra fields.

Mbuf Chaining

An mbuf contains two fields which point to the next mbuf in a linked list, specifically

field	Meaning
m_next	typically used to link mbufs in a chain.
m_list	typically used to link separate mbuf chains together.

When a device driver calls a protocol module's receive handler, it uses a single mbuf chain to hold each received frame, and can link several frames together for passing via the single call. Figure 2 shows how m_next and m_list are used to link chains of mbufs together into a list.

Note that, although this structure allows different frame types to be passed to the protocol module (because the first mbuf+ in each chain contains a struct rx_hdr which includes the frame_type field), the receive handler is only given a single Driver Information Block, and therefore **all** the frames passed in any one call to the handler must come from a single unit.

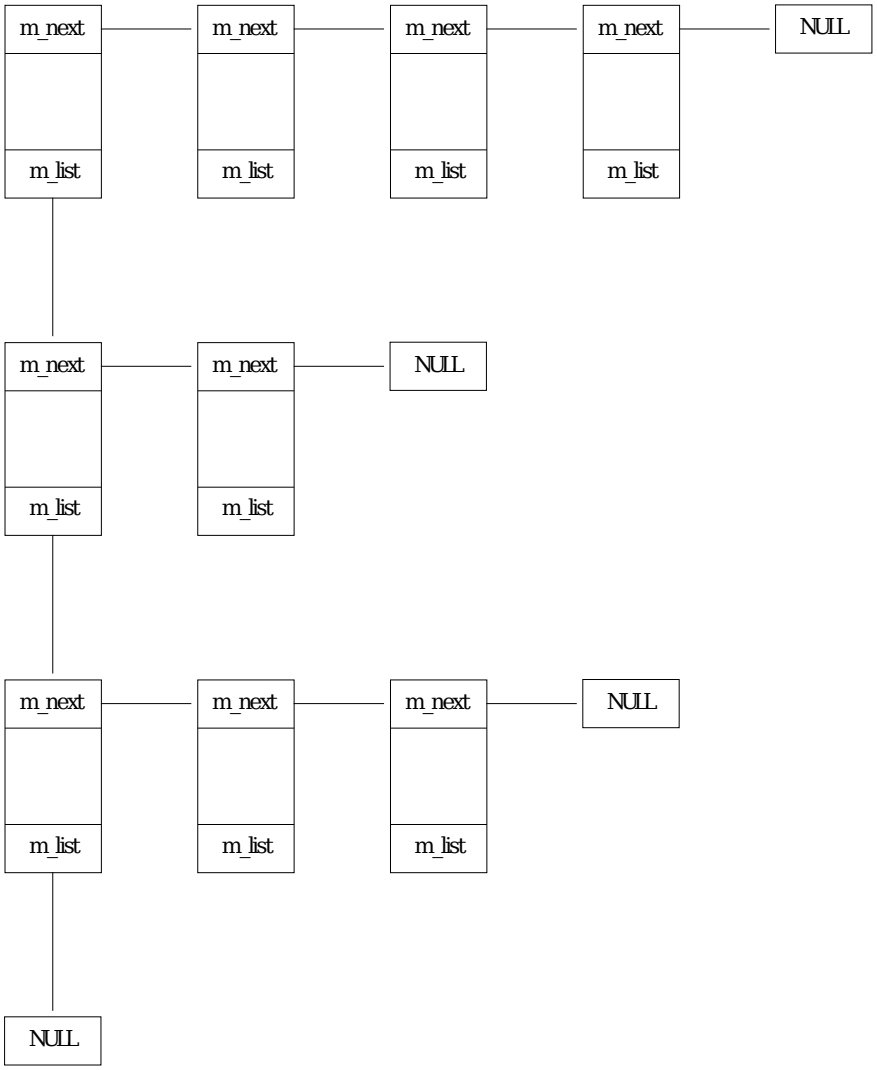


Figure 2: Linking mbuf chains

7 Statistics

7.1 Introduction

The SWI DCIDriver_Inquire (on page 24) SWI makes mention of device drivers supporting both standard, and extended statistics interfaces. This version of the DCI does not define an extended statistics interface, but it does define a standard stats. interface, and that is what this section is all about.

This document defines what it considers to be the definitive list of network parameters, and the driver maintains a subset of these (remembering that the whole set is a valid subset). An independent set of statistics is maintained for each unit that the driver controls.

The SWI DCIDriver_Stats (on page 32) serves two purposes:

1. It identifies which statistics the driver gathers for a particular unit.
2. It allows reading of the gathered statistics.

7.2 Data Structures

The statistics structure is written in C code as:

```
struct stats
{
    /* general information */
    unsigned char st_interface_type;
    unsigned char st_link_status;
    unsigned char st_link_polarity;
    unsigned char st_blank1;
    unsigned long st_link_failures;
    unsigned long st_network_collisions;

    /* transmit statistics */
    unsigned long st_collisions;
    unsigned long st_excess_collisions;
    unsigned long st_heartbeat_failures;
    unsigned long st_not_listening;
    /* unsigned long st_net_error; */ /* CJF: This is no longer par
    unsigned long st_tx_frames;
    unsigned long st_tx_bytes;
    unsigned long st_tx_general_errors;
    unsigned char st_last_dest_addr[8];

    /* receive statistics */
    unsigned long st_crc_failures;
    unsigned long st_frame_alignment_errors;
    unsigned long st_dropped_frames;
```

```
    unsigned long st_runt_frames;
    unsigned long st_overlong_frames;
    unsigned long st_jabbers;
    unsigned long st_late_events;
    unsigned long st_unwanted_frames;
    unsigned long st_rx_frames;
    unsigned long st_rx_bytes;
    unsigned long st_rx_general_errors;
    unsigned char st_last_src_addr[8];
};
```

The fields within this structure are:

Offset	Field	Contents
+0	st_interface_type	A single byte coding the specific hardware interface type. Values so far defined are:

Code	Interface type
1	10-base5
2	10-base2
3	10-baseT
4	Combination 10-base5 / 10-base2
5	Combination 10-base2 / 10-baseT
6	Reduced Squelch 10-baseT
7	Acorn Econet
8	Serial line
9	Parallel port
10	Combination 10-base5 / 10-base2 / 10-baseT
11	10-baseFX
12	100-baseTX
13	100-baseVG
14	100-baseT4
15	100-baseFX
16	ATM 25.6
17	ATM 155

Note that the use of 'combination' types is deprecated in favour of returning a specific value reflecting the interface's current configuration.

+1	st_link_status	A bitfield describing the current state of the interface; significant bits are:

Offset	Field	Contents

Bit(s) Meaning

0 Clear: Interface bad (i.e. self-test failed).

Set: Interface OK.

1 Clear: Interface is inactive.

Set: Interface is active.

2-3 Describe the currently configured receive level as follows:

Binary value	Meaning
00	Accept directly addressed frames only.
01	Accept directly addressed and broadcast frames only.
10	Accept direct, broadcast, and multicast frames.
11	Promiscuous mode, accept **all** frames.

4 Clear: Link is half duplex.

Set: Link is full duplex.

5-7 Reserved, must be zero

+2	st_link_polarity	A bitfield where:

Bit(s) Meaning

0 Set: Link polarity correct

Clear: Link polarity incorrect

1-7 Reserved, must be zero

+3	st_blank1	Unused, must be set to zero.

Offset	Field	Contents
+4	st_link_failures	Counts the number of times a good link went away.
+8	st_network_collisions	Counts the total number of collisions on the network.
+12	st_collisions	The number of times a collision has occured when trying to transmit a packet.
+16	st_excess_collisions	A count of excess transmit collisions.
+20	st_heartbeat_failures	The number of times the Signal Quality Error Test failed to detect a collision.
+24	st_not_listening	A count of the number of times when the remote station was not listening. (This statistic will usually be specific to Acorn Econet)
+28	st_tx_frames	The total number of frames transmitted since driver initialisation.
+32	st_tx_bytes	The total number of bytes transmitted since driver initialisation.
+36	st_tx_general_errors	A count of the number of non-specific network errors that occured during transmission.
+40	st_last_dest_addr	Hardware address of the last interface to which a frame was sent.
+68	st_jabbers	The number of times the interface was caught jabbering.
+76	st_unwanted_frames	The number of frames received, but not claimed by any protocol module.
+80	st_rx_frames	The total number of frames received since driver initialisation.
+84	st_rx_bytes	The total number of bytes received since driver initialisation.
+88	st_tx_general_errors	A count of the number of non-specific network errors that occured during frame reception.
+92	st_last_src_addr	Hardware address of the last interface from which a frame was received.

⚠ FIXME: The specification is missing a number of these descriptions - 44-64, 72.

Note that statistics are gathered for **all** frames received --- even if a driver subsequently decides that no protocol wants a given frame, that frame still appears in the relevant receive statistics (i.e. st_rx_bytes, st_last_src_addr etc.).

7.3 Statistics

The basic interface for reading statistics from a driver is the SWI DCIDriver_Stats (on page 32) SWI, outlined in section 5.3 There are two different forms of this SWI, selected by bit 0 of R0 --- the first form is used to determine which statistics are supported by the driver, while the second form is used to read the statistics.

To indicate which statistics it supports, a device driver returns a statistics structure with all bits in those fields it does support set to 1, and all bits in those fields it doesn't support set to 0. Those fields which are a variable length (i.e. st_last_dest_addr & st_last_src_addr) use the same mechanism to indicate which parts of the field are valid. For example, a standard Ethernet interface which uses 6-byte hardware addresses would return st_last_src_addr set to

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x00,

whereas a PPP driver which does not use hardware addresses, and therefore would not support this field would return it set to

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00.

When returning the statistics, all multi-byte fields are returned with host byte ordering.

8 Memory Management

8.1 Introduction

In all versions of the DCI, data pass across the interface between protocol modules and device drivers in "mbufs". These are based upon the data structures originally developed for handling network data within BSD Unix kernels.

Mbufs within DCI 4 are noticeably different from their brethren, both those from BSD, and those from earlier versions of the DCI, the main distinctions being:

- Mbufs and the data they describe no longer occupy a contiguous piece of memory.
- It is no longer the responsibility of protocol modules to allocate and maintain pools of free memory --- DCI 4 introduces a single, centralised, memory manager module which all protocol and device driver modules claim memory from in the form of mbufs.
- The set of function calls and macros for manipulating mbufs (i.e. those operations defined in mbuf.c and mbuf.h) provided by the new memory manager module are completely changed from those used in earlier versions of the DCI. Any module being upgraded to DCI 4 will have to have all these calls changed to the new versions.

8.2 Memory Manager Module

Overview

Memory management for packet storage in earlier versions of the DCI is performed with mbufs. DCI 4 also uses mbuf based packet storage, but there are some differences. These differences are for the following reasons:

1. Correct design oversights in previous versions of the DCI.
2. Provide a more modular, and upgradeable, system.
3. Offer single mbuf arbiter with optimised routines available to all DCI4 components.

The memory manager, the arbiter module, is central to the DCI4 mbuf scheme. It performs most of the low level work associated with mbufs, as well as relieving both protocol and client modules of some tedium.

A complete specification of the memory manager is available separately; this document is designed to guide the reader conversant with "traditional" mbufs through using DCI4 mbufs.

Communication with the memory manager is centred around an mbctl structure. This is stored in the client's memory, and is mainly initialised by the memory manager to contain useful information, including the addresses of a number of routines within the memory manager for the client to call directly.

Direct entry points are designed to permit the easy inter-operation of assembler and APCS code (such as that generated by the NorCroft C compiler), and roughly obey APCS. A list of entry/exit characteristics follows (using APCS register naming convention):

1. a1 always points at an mbctl structure for all direct entry calls
2. the processor must be in supervisor mode
3. a1--a4 are the only parameter registers
4. a2--a4 and ip are corrupted by the call
5. a1 is either the call result or corrupted
6. other registers preserved by call
7. the processor flags are preserved by the call
8. no V set error convention (incompatible with APCS)
9. in general, an error results in a1=0 on exit
10. IRQ state preserved across call
11. IRQs may be disabled during calls
12. IRQs may be enabled during calls ONLY if specifically documented
13. FIQs assumed enabled on entry
14. FIQs preserved across calls

Currently, no direct entry point routine will enable interrupts if they are disabled on entry.

The header file mbuf.h provides some macros to manage interfacing with the memory manager routines.

These direct entry points provide access to allocator and free routines, along with a whole host of support routines. Rather than each protocol implementing it's own mbuf scheme, and each device driver having to choose the correct mbuf pool to allocate from, all protocols and all device drivers perform their allocations and frees via these direct entry points.

Structures

The new memory manager module uses two main data structures: the mbuf structure, each one of which describes a piece of atomically allocated memory, and struct mbctl, the control struture which describes the exact interface between the memory manager and one of its clients.

Note that, although a struct mbuf is recognisably similar to the structure used in "traditional" memory management schemes, there

are enough differences in the new structure to render it incompatible with the macros defined in the traditional versions of mbuf.h.

The new definition of an mbuf is shown below:

```
typedef struct mbuf
{
    struct mbuf *m_next; /* next mbuf in chain */
    struct mbuf *m_list; /* next mbuf in list (clients only) */
    ptrdiff_t m_off; /* current offset to data from mbuf itself */
    size_t m_len; /* current byte count */
    const ptrdiff_t m_inioff; /* original offset to data from mbuf
    const size_t m_inilen; /* original byte count (for underlying c
    unsigned char m_type; /* client use only */
    const unsigned char m_sys1; /* MBufManager use only */
    const unsigned char m_sys2; /* MBufManager use only */
    unsigned char m_flags /* client use only */
    struct pkthdr m_pkthdr; /* client use only */
} dci4_mbuf;
```

The MLEN macro value is no longer directly applicable --- each mbuf must have its maximum size checked individually. Likewise, reseting m_off now requires examining the `m_inioff` field of the mbuf - just setting it to zero is no longer good enough.

`m_act` has been renamed `m_list`.

`m_inilen` and `m_inioff` are provided to replace the MMINOFF and MMAXLEN macros, as the values are now dependent upon the particular mbuf in question.

`m_indir` has disappeared --- specific routines exist for determining if an mbuf chain contains unsafe data.

The mbuf structure has been separated from the underlying storage it describes. The underlying storage blocks may now be different sizes (128 and 1536 byte blocks are currently used).

Earlier versions of the DCI had big mbufs, but they were weakly defined and dtom did not work with them. DCI 4 corrects both these points --- indirect mbufs, which at times were not distinguishable from large mbufs, have been formalised into unsafe mbufs.

As an mbuf chain passes around the system, ownership of that chain is also transferred. Ownership brings with it the responsibility to free the mbuf chain (unless it is transferred to another component, although this is unlikely).

The other crucial structure in DCI 4 memory management is `mbctl`:

```
typedef struct mbctl
```

```
{
    /* reserved for MBufManager use in establishing context */
    int opaque; /* MBufManager use only */

    /* Client initialises before session is established */
    size_t mbcsize; /* size of mbctl structure from client */
    unsigned int mbcvers; /* client version of MBufManager* spec */
    unsigned long flags; /* */
    size_t advminubs; /* Advisory desired minimum underlying block
    size_t advmaxubs; /* Advisory desired maximum underlying block
    size_t mincontig; /* client required min ensure_contig value */
    unsigned long spare1; /* Must be set to zero on initialisation

    /* MBufManager initialises during session establishment */
    size_t minubs; /* Minimum underlying block size */
    size_t maxubs; /* Maximum underlying block size */
    size_t maxcontig; /* Maximum contiguify block size */
    unsigned long spare2; /* Reserved for future use */

    /* Allocation routines */
    struct mbuf * /* MBC_DEFAULT */
    (* alloc)
    (struct mbctl *, size_t bytes, void *ptr);

    struct mbuf * /* Parameter driven */
    (* alloc_g)
    (struct mbctl *, size_t bytes, void *ptr, unsigned long flags);

    struct mbuf * /* MBC_UNSAFE */
    (* alloc_u)
    (struct mbctl *, size_t bytes, void *ptr);

    struct mbuf * /* MBC_SINGLE */
    (* alloc_s)
    (struct mbctl *, size_t bytes, void *ptr);

    struct mbuf * /* MBC_CLEAR */
    (* alloc_c)
    (struct mbctl *, size_t bytes, void *ptr);

    /* Ensuring routines */
    struct mbuf *
    (* ensure_safe)
    (struct mbctl *, struct mbuf *mp);

    struct mbuf *
    (* ensure_contig)
    (struct mbctl *, struct mbuf *mp, size_t bytes);

    /* Freeing routines */
```

```
void
(* free)
(struct mbctl *, struct mbuf *mp);

void
(* freem)
(struct mbctl *, struct mbuf *mp);

void
(* dtom_free)
(struct mbctl *, struct mbuf *mp);

void
(* dtom_freem)
(struct mbctl *, struct mbuf *mp);

/* Support routines */
struct mbuf * /* No ownership transfer though */
(* dtom)
(struct mbctl *, void *ptr);

int /* Client retains mp ownership */
(* any_unsafe)
(struct mbctl *, struct mbuf *mp);

int /* Client retains mp ownership */
(* this_unsafe)
(struct mbctl *, struct mbuf *mp);

size_t /* Client retains mp ownership */
(* count_bytes)
(struct mbctl *, struct mbuf *mp);

struct mbuf * /* Client retains old, new ownership */
(* cat)
(struct mbctl *, struct mbuf *old, struct mbuf *new);

struct mbuf * /* Client retains mp ownership */
(* trim)
(struct mbctl *, struct mbuf *mp, int bytes, void *ptr);

struct mbuf * /* Client retains mp ownership */
(* copy)
(struct mbctl *, struct mbuf *mp, size_t off, size_t len);

struct mbuf * /* Client retains mp ownership */
(* copy_p)
(struct mbctl *, struct mbuf *mp, size_t off, size_t len);

struct mbuf * /* Client retains mp ownership */
```

```
    (* copy_u)
    (struct mbctl *, struct mbuf *mp, size_t off, size_t len);

    struct mbuf * /* Client retains mp ownership */
    (* import)
    (struct mbctl *, struct mbuf *mp, size_t bytes, void *ptr);

    struct mbuf * /* Client retains mp ownership */
    (* export)
    (struct mbctl *, struct mbuf *mp, size_t bytes, void *ptr);
} dci4_mbctl;
```

Some of the fields the client initialises are present to permit future versions of the memory manager to tune themselves as tightly as possible to the setup they are asked to support.

Note that dtom is no longer a macro. Don't worry --- it's efficient assembler, and it works with all sizes of mbuf the memory manager cares to use.

Using the memory manager module

Basic use of the memory manager is performed as follows

1. module loads and looks for memory manager
2. if memory manager is absent, module goes into a pre-active state, awaiting the arrival of the memory manager
3. once the memory manager is present, a "session" is opened with it
4. the device driver/protocol may now become active if it is pre-active (this might involve delaying arrival service calls until now)
5. the device driver/protocol uses direct entry points to communicate with the memory manager
6. the device driver/protocol is about to die --- it first closes the open session with the memory manager
7. the device driver/protocol can now die

Initialisation with the memory manager is performed with code something like:

```
static _kernel_oserror *open_mbuf_manager_session(void)
{
    _kernel_swi_regs r;
    memset(&mbctl, 0, sizeof(struct mbctl));
    mbctl.mbcsize = sizeof(struct mbctl);
    mbctl.mbcvers = MBUF_MANAGER_VERSION;
    mbctl.flags = 0;
    mbctl.advminubs = 0;
    mbctl.advmaxubs = 0;
```

```
    mbctl.mincontig = 0;
    mbctl.spare1 = 0;
    r.r[0] = (int) &mbctl;
    return(_kernel_swi(Mbuf_OpenSession, &r, &r));
}
```

Finalisation is performed with code something like:

```
static _kernel_oserror *close_mbuf_manager_session(void)
{
    _kernel_swi_regs r;
    r.r[0] = (int) &mbctl;
    return(_kernel_swi(Mbuf_CloseSession, &r, &r));
}
```

A quick summary of the available direct entry point routines:

Offset	Name	Contents
+48	alloc	standard allocator. Can import data, but cannot zero the underlying storage, force single mbuf allocation or allocate unsafe data.
+52	alloc_g	the allocator which can emulate the other allocator functions.
+56	alloc_u	allocate unsafe mbufs
+60	alloc_s	force allocation to a single mbuf
+64	alloc_c	clear the underlying storage after allocation.
+68	ensure_safe	Examines each mbuf and returns a modified mbuf chain if any mbufs are unsafe.
+72	ensure_contig	Ensures that a given region of the described data is contiguous in memory, to permit structures to be "cast over it".
+76	free	Frees a single mbuf
+80	freem	Frees an mbuf chain
+84	dtom_free	Performs a dtom operation and then a free operation on the result
+88	dtom_freem	Performs a dtom operation and then a freem operation on the result
+92	dtom	Transform a data pointer to the mbuf describing it
+96	any_unsafe	scan an mbuf chain for unsafe mbufs
+100	this_unsafe	determine whether an mbuf is safe or unsafe.
+104	count_bytes	return the number of bytes described by an mbuf chain
+108	cat	concatenate two mbuf chains together
+112	trim	adjust m _len and m _off values to remove data from an mbuf chain.
+116	copy	produce an mbuf chain containing a copy of the data described by an mbuf chain.
+120	copy_p	produce an mbuf chain containing a copy of the data described by an mbuf chain. The only difference between this routine and copy is that this routine assumes that the m_type, m_flags and m_pkthdr fields contain important data which should be preserved during the copy.
+124	copy_u	produce an unsafe copy of of the data described by an mbuf chain.
+128	import	import data from raw memory into an mbuf chain

Offset	Name	Contents
+132	export	export from from an mbuf chain into raw memory

So, a device driver might use the allocator as follows:

```
struct mbuf *mp = mbctl.alloc(&mbctl, packlen, NULL);
```

which allocates an mbuf chain of "packlen" bytes.

The entire chain might later be freed thus:

```
mbctl.freem(&mbctl, mp);
```

The reason all the direct entry point calls take a (struct mbctl *) value as there first parameter is to permit the MBufManager to establish a context within which it is operating (ie find its workspace!).

Finally, the memory manager supports the DCI4 statistics interface. This can be useful in fine tuning your DCI4 component.

8.3 Unsafe Data

The concept of indirect data mbufs was introduced in earlier versions of the DCI to eliminate the need for data to be copied where this is possible; this results in a significant improvement in frame rates. Typically, an mbuf is used to indirect to user data, rather than making a private copy of that data.

An important implication of this is that a protocol module cannot rely on the indirect pointers after any system call which uses them has returned to the user: if they need to keep the data after this time, then a private copy **must** be made of the data. Protocol modules should always know whether an mbuf chain is unsafe or not (since they create the chain in the first place); device drivers are informed via the flags in the SWI DCIDriver_Transmit (on page 28) SWI whether or not the passed mbuf chain is safe or not --- if they need to use any data from an unsafe mbuf chain after the SWI has returned, then they must make a copy of that chain.

9 Miscellanea

9.1 Network Card Self-Tests

All network cards should support at least one *-command, used to initiate a hardware self-test. This * -command should be of the form <*name*>test, where <*name*> is the driver name as supplied in the dib_name field of the driver information block

When invoked, the self-test command should, to the best of its ability, ascertain whether the network hardware is still functioning correctly, and print a short success/failure message. As part of this self-test, the drivers should, where possible, perform a live network test (this is because many network faults are due to cabling problems rather than hardware failures, and a live network test may be able to detect these problems).

9.2 Virtual Interfaces

When running some form of PC emulator under RISC OS, it is frequently desirable to run a second protocol stack within the emulator that is independant of a similar protocol stack running on the native OS (the classic example being a TCP/IP stack running with one Internet address under RISC OS, and a PC based TCP/IP stack running under the emulator with a different Internet address).

One hardware-based solution to this problem would be to simply have two Ethernet cards in the same machine, each dedicated to one of the competing protocol stacks; the obvious downside to this solution would be the expense --- any software-based solution that allowed one card to support two interfaces would be much cheaper.

The optional software solution supported by DCI 4 is the concept of 'virtual interfaces'. A virtual interface is where a driver creates a second unit for a physical interface, this unit having an Ethernet hardware address that is different from the hardware address for the first, "real" unit for the interface.

Exactly how this virtual interface is implemented is highly dependent upon the Ethernet controller chip used by the interface. Some controllers allow more than one hardware address to be specified for the one interface; this obviously makes the implemention of a virtual interface a relatively easy task. For controllers that do not allow more than one hardware address, the device driver will need to put the interface into promiscuous mode, and discard frames with unwanted hardware addresses under software control.

For any unit which is a virtual interface, bit 10 of the Inquire flags for that unit should be set; if the virtual interface uses software filtering of Ethernet hardware addresses, then bit 11 of these flags should

also be set.

Document information

Maintainer(s): Charles Ferguson <gerph@gerph.org>

History:

Revision	Date	Author	Changes
A	22 Jul 1994	Acorn/ ANT	● Initial draft revision, passed round for comment
B	12 Aug 1994	Acorn/ ANT	● Incorporated reviewers' comments. Added specification of MBufManager module.
C	16 Aug 1994	Acorn/ ANT	● Description of MBufManager module tidied up and clarified
D	09 Nov 1994	Acorn/ ANT	● Large number of changes introduced following formal review, and feedback from external reviewers. ● Device drivers can now be identified by the address of their Driver Information Block. A new subsection has been added to the introduction to explain this feature. ● The concept of a protocol handle has been removed. ● Register usage for service calls has been changed. ● Service calls Service_ProtocolDying, Service_FindNetworkDriver, and Service_NetworkDriverStatus are now obsolete. ● The new service calls Service_DCIProtocolDying, Service_DCIDriverStatus and Service_DCIFrameTypeFree have been added. ● All SWI calls now have a flags register. The gaps in the SWI chunk from earlier versions of the DCI have been closed. ● The Filter SWI has been heavily reworked: ● Frame type, and

			frame level have been merged into a single register; register numbers have been shuffled to fill the gap left by frame level. ● The read flag has been removed - it is no longer possible to read current filter levels. ● Releasing a filter is now achieved via a flag bit, rather than by specifying a frame level of FRMLVL_NONE. ● The ensure safe flag bit has been added. ● IEEE 802.3 frames are more complicated than originally thought, the concept of frame type for these frames has been extensively revised. ● The rx_frame_type field in struct rx_hdr now only contains the last 2 bytes of the MAC header. ● A completely new memory manager has been designed. ● The dib_swibase field has been moved to the top of a struct dib. ● Unsafe data are no longer flagged to the Transmit SWI. ● Added a small section on SWI re-entrancy, and re-enabling interrupts. ● Added new sections on acceptance tests, and on development test strategies.
E	03 Feb 1995	Acorn/ ANT	**DCI version 4.01** ● The official DCI version is now 4.01. ● Corrected some typos, then made some alterations and additions ● DCIProtocolDying has been renamed to Service_DCIProtocolStatus,

			and its register usage changed. ● Added a section on returning errors from SWI calls which defines some standard error numbers. ● Added the new SWI Stats. ● Created a whole new section describing the standard statistics interface. ● Added a couple of lines (in a new miscellanea section) about network card self-tests.
F	14 Mar 1995	Acorn/ ANT	**DCI version 4.02** ● This is now DCI version 4.02. ● Added a paragraph to the description of Service_DCIDriverStatus to define (by cross-reference) the format of the supported DCI version. ● Service_DCIFrameTypeFree has been changed to reflect the changes made to the Filter SWI for revision D, i.e. frame type and frame level have been merged into a single 32-bit register, R2; parameters that were in registers R4 & R5 have been moved into R3 & R4 respectively. ● Corrected some typos. ● Table of standardised errors (hopefully) made more explicit by adding a column detailing all the error numbers. ● Added a paragraph to the statistics section clarifying when statistics are gathered (i.e. for all frames). ● Added some new members to struct stats: st_tx_general_errors, st_unwanted_frames, st_rx_general_errors ● Added codes 9 & 10 for st_interface_type field in struct stats structure.

			● Removed field st_net_error from struct stats --- it has been made redundant by new field st_tx_general_errors. ● Field st_link_status in struct stats made into a bitfield.
G	10 Apr 1995	Acorn/ ANT	**DCI version 4.03** ● This is now version 4.03 of the DCI. ● The definition of a Driver Information Block has been extended to include a copy of the Inquiry flags. ● Added a couple of entry points missing from struct mbctl (copy_p and copy_u).
1.00	5 Sep 1995	Acorn/ ANT	● Wrote a new subsection on virtual interfaces, and added virtual interface flag bits to the Inquire SWI
1.01	14 Apr 1997	Acorn/ ANT	**DCI version 4.04** ● This is now version 4.04 of the DCI. ● SWI MulticastRequest added. ● Fixed some formatting errors in the Impression version of the specification.
1.02	14 Apr 1998	Acorn/ ANT	● Added extra flag bits for 100MB Ethernet.
1.02	16 Sep 1998	Acorn/ ANT	**Issue 2 released on ECO 4112** ● no change from version 1.02
1.03	26 Aug 1999	Acorn/ ANT	**DCI version 4.05** ● IP checksumming facility added. ● This is now version 4.05 of the DCI.
1.04CJF	08 Aug 2020	Gerph	**Restructure as PRM-in-XML** ● Exported from Impression, and converted to PRM-in-XML structured format. ● Formatting changes have been made to use structured data formats, with tables of structures in addition to the C structure definitions. ● Removed the st_net_error field from the statistics C definition; this field had been removed from the definitions in 4.02 (as seen in

			TCPIPLibs) but was still present within this document.
1.05	CJF	01 Nov 2022	Gerph	**Tidy up XML** ● Tidied up the XML and some of the structure to better match PRM-in-XML expectations and lint cleanly.

MbufManager

Overview

0 Currently outstanding

C is acceptable as an illustrative language but is not ideally suited to a definition language. Language neutral versions of all structures, etc, need producing. Not all macros contained within the reference mbuf.h file are documented although their behaviour is readily determinable from this specification.

More should probably be said about unsafe data shadowing safe data.

The dci4 client interface contract appendix needs tightening up.

---

1 Conventions

'quoted strings' should be envisaged in an italic font.

"quoted strings" indicate phrases with specific technical interpretation, typically only when first introduced.

[bracketed text] is an aside to reviewers. Comments on such text is encouraged.

77 hyphen (-) characters denote table and figure delimitation.

---

2 Basics of operation

A "client" is a program (typically a module and in practise probably constrained to be a module) that uses the facilities of the "MbufManager".

Clients establish a session with the MbufManager on initialisation, use memory management facilities from the mbuf manager during their normal operation, and then close the session with the MbufManager just prior to their shutdown.

Memory is manipulated through the use of a descriptor structure, called an mbuf. An mbuf describes a number of contiguous bytes in memory.

The MbufManager imposes some restrictions, requirements and conventions upon the use of mbufs. A group of client typically implement a further interface contract of their own - the DCI4 client interface contract is such an interface (see Appendix).

3 MbufManager goals

- Hide any mechanics not necessary for client operation
- Provide single mbuf pool, rather than one per protocol
- Provide facilties suitable for device drivers and protocol modules
- Provide a balanced compromise between conflicting design goals
- Provide negligable long term fragmentation
- Provide efficient implementations of facilities offered
- Permit pre-allocation of packet storage space (DCI2 doesn't)
- Permit modular component upgrade path (DCI2 doesn't)

---

4 The mbuf structure and its uses

Mbufs are used to provide a descriptive structure layer that describe and dictate access to a conceptual block of memory.

In conventional memory management, the user manipulates a pointer to a block of memory. With mbuf memory management, the user manipulates a pointer to a descriptive structure (an "mbuf"), which itself provides the means to obtain a pointer to the block of memory that it "describes". Further, these structures are chained together to form a linked list, providing a form of scatter/gather memory description. The chain of mbufs describes a single conceptual block of memory, even though the actual memory used might well be scattered throughout real memory.

Thus, an extra layer of structuring is inserted between the user and the block of memory being manipulated when mbuf memory management is used.

By accessing a block of memory through an mbuf structure, it is possible to record size, type and some degree of linkage information. By manipulating the fields within an mbuf, it is possible to efficiently add and remove data from a described" block of memory in a variety of fashions.

Mbufs are allocated and freed by the MbufManager. Programs that requestallocation and freeing operations are called clients.

A client never has a "struct mbuf" itself, only pointers that at some stage came from the MbufManager. (This allows the size of an mbuf to grow later; in particular MbufManagers may differ in the amount of private data stored with the mbuf.)

The C definition of an mbuf structure:

```
struct ifnet;
struct pkthdr {
    int len; /* total packet length */
    struct ifnet *rcvif; /* receiving interface */
};
typedef struct mbuf {
    struct mbuf *m_next; /* next mbuf in chain */
    struct mbuf *m_list; /* next mbuf in list (clients only) */
    ptrdiff_t m_off; /* current offset to data from mbuf itself */
    size_t m_len; /* current byte count */
    const ptrdiff_t m_inioff; /* original offset to data from mbuf
    const size_t m_inilen; /* original byte count (for underlying c
    unsigned char m_type; /* client use only */
    const unsigned char m_sys1; /* MbufManager use only */
    const unsigned char m_sys2; /* MbufManager use only */
```

```
    unsigned char m_flags; /* client use only */
    struct pkthdr m_pkthdr; /* client use only */
} dci4_mbuf;

struct mbuf *m_next;
```

Although mbufs may be manipulated individually, they are almost always used as an mbuf chain. The 'm_next' field of the mbuf structure points at the next mbuf in an mbuf chain. There is no back pointer. Successive mbufs describe conceptually "later" bytes of memory, even if the underlying blocks of actual memory used to hold these bytes are not stored consecutively or "later" in memory. The end of an mbuf chain is indicated by the 'm_next' field containing the NULL pointer.

A client may allocate an mbuf chain for internal use or for communicating data to another (MbufManager) client. An mbuf chain may exist only for a small fraction of a second or it may be used and retained for weeks. It is the task of the MbufManager to ensure that such usage does not cause anything other than negligible memory fragmentation.

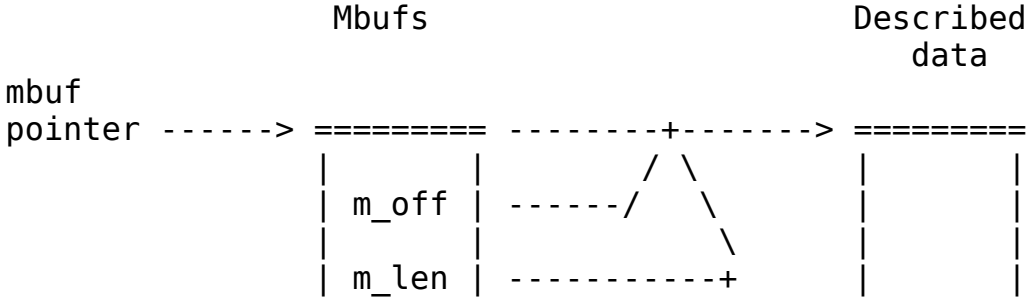
```
ptrdiff_t m_off;
```

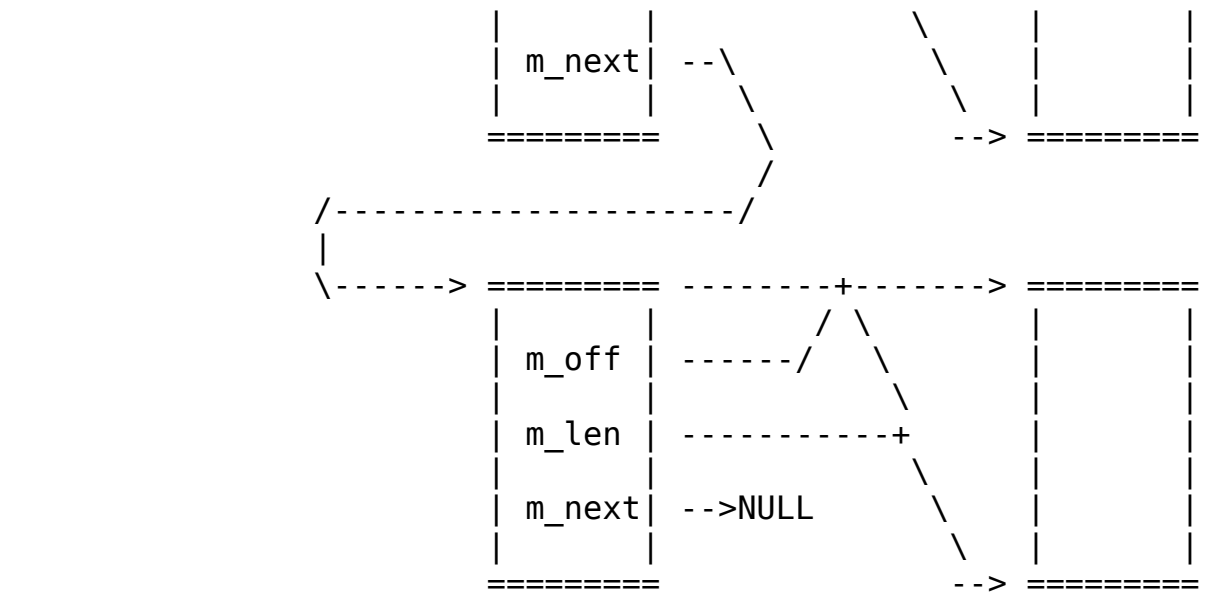
The allocation of an mbuf is always accompanied by the allocation of an additional block of memory (see the discussion on unsafe data later for the exceptions to this). It is this additional block of memory that the mbuf is said to "describe". Access to this described memory is through manipulation of the address of the mbuf itself and fields contained within the mbuf. The location of this memory, relative to the mbuf itself, is not defined, other than the 'm_off' field contains a suitable bias to access this memory.

```
size_t m_len;
```

The 'm_off' field is the bias to add to the address of the mbuf to obtain the address of the first byte of data described by that mbuf. The 'm_len' field specifies the number of bytes contained in the described data. This might be envisaged as follows:

Relationships between the address of an mbuf, and the 'm_off', 'm_next' and 'm_len' fields.





Notes:

Only three of the fields of an mbuf are indicated. This is for clarity purposes. The example illustrates an mbuf chain formed from two mbufs.

Example C code to flatten an mbuf chain into the single sequence of bytes that it conceptually describes:

```
void flatten_mbuf_chain(struct mbuf *mp, char *buffer)
{
    for ( ; mp != NULL; mp = mp->m_next)
    {
        memcpy(buffer, mtod(mp, char *), mp->m_len);
        buffer += mp->m_len;
    }
}
```

Notes:

As with most example programs, no thought has been given to the handling of exceptional circumstances.

This algorithm applies equally to safe and unsafe mbuf chains.

'mtod' is a macro, and adds the address of the first byte of the mbuf to the value of the 'm_off' field of the mbuf, yielding the address of the first byte of data described by the mbuf. It also performs a type cast. 'mtod' is supplied as a C macro for convenience.

Only byte alignment is required for the data described by the 'm_off' and 'm_len' fields. All clients must fully cope with non-word aligned addresses and lengths when manipulating the data described by an

mbuf, although optimisations for aligned data are encouraged, as is the generation of aligned data. An mbuf structure itself is always at least word aligned in memory.

When the MbufManager performs an allocation for a client and returns an mbuf chain to a client, that client is deemed to have taken "ownership" of that mbuf chain. The precise implications of ownership form part of the interface contract between clients of the MbufManager. For example, the DCI4 specification specifies an interface contract between compliant modules using mbufs. One of the prime responsibilities of ownership is to free the mbuf chain at some stage (or transfer ownership). Whenever the MbufManager returns an mbuf chain, it transfer ownership of this chain at the same time.

Whenever the MbufManager receives an mbuf chain it also takes ownership of that chain. A summary of ownership transfer for direct entry points is given later. Exceptions to these rules are details individually throughout the text.

In order to minimise long term fragmentation (and for various other implementation reasons), the sizes of the underlying memory blocks that may be allocated for an mbuf to describe are constrained to a number of sizes. This permits the situation where an mbuf describes only some of the underlying memory actually allocated.

```
const ptrdiff_t m_inioff;

const size_t m_inilen;
```

The 'm_inioff' and 'm_inilen' fields provide a description of the underlying block of memory in the same way as 'm_off' and 'm_len' fields (respectively) provide a description of the described block of memory (which resides somewhere within the underling block, although not necessarily always at the start or end of it). Any values of 'm_len' and 'm_off' are permitted provided that the described block of memory is fully contained within the underlying block described by 'm_inioff' and 'm_inilen' (but see the field validity table below).

```
struct mbuf *m_list;

unsigned char m_type;
```

The 'm_list' and 'm_type' fields are provided for the convenience of the client. The contents of these fields when an mbuf is passed from one client to another is part of the interface contract between clients. When a client owns an mbuf chain, it can set these fields to whatever values it requires.

If these fields are used, a client must explicitly initialise them. The

MbufManager never examines these fields.

```
const unsigned char m_sys1;

const unsigned char m_sys2;
```

The 'm_sys1' and 'm_sys2' fields are private fields maintained by the MbufManager. They should never be read or written by a client under any circumstances, even transiently. The MbufManager is entitled to asynchronously examine these three fields if it requires.

```
unsigned char m_flags;

struct pkthdr m_pkthdr;
```

The 'm_flags' field, and the 'm_pkthdr' field in version 0.15 or later of the MbufManager, are provided for the convenience of the client . The contents of these fields when an mbuf is passed from one client to another is part of the interface contract between clients. When a client owns an mbuf chain, it can set these fields to whatever values it requires.

If these fields are used, a client must explicitly initialise them. The MbufManager never examines these fields.

When an mbuf chain is freed, the storage required for the mbuf(s) comprising the chain and the underlying memory associated with each mbuf is placed backinto the free pool and becomes available for subsequent re-allocation. It is not possible to free only one of the mbuf and the underlying storage associated with that mbuf. This ensures that, as long as an mbuf chain is allocated, then the data it describes is also be correctly allocated.

---

5 Unsafe data

"Unsafe data" is a concept that breaks some of the rules just outlined. In particular, the memory described by an unsafe mbuf is not underlying memory allocated by the MbufManager in the fashion just described.

When an unsafe mbuf is allocated, the MbufManager does not allocate associated underlying storage. Rather, the mbuf is available for the client to set the 'm_off' and 'm_len' fields such that a portion of memory beyond the control of the MbufManager is described by the mbuf. The only requirement of the MbufManager on the 'm_off' and 'm_len' fields for an unsafe mbuf is that the data they describe must be valid whenever an MbufManager operation implicitly or explicitly accesses them. In practise, the only time when these fields may hold random values is when the mbuf (chain) is being freed or transiently during update within a client.

The MbufManager can tell whether an mbuf describes safe or unsafe data from examination of the mbuf. When an unsafe mbuf is freed, there is no freeing action performed on the associated data.

It is because there is no directly enforceable relationship (by the MbufManager) between the lifetime of an unsafe mbuf and the data it describes that the data is termed "unsafe". "Unsafe data" does not imply incorrect behaviour. The phrase is used as is a reminder of the additional constraints on the described data, and serves to encourage the programmer to take the necessary extra precautions.

Unsafe data is often used when it is not necessary to copy the data into a safe mbuf chain, eliminating a copy operation and increasing performance. A client may arrange its internal strategies to permit the use of unsafe data as an optimisation.

In order for the concept of unsafe data to be useful, some statement about the lifetime and validity of the described data must be possible. The client interface contract normally adds further detail to the requirements for unsafe mbufs.

Whenever an unsafe mbuf is supplied to the MbufManager in a context that the MbufManager may examine the data described, then the client pledges that this data will remain valid until that call into the MbufManager returns.

In practise there are two useful ways in which unsafe mbuf chains may be manipulated:

1. All use of the data is completed before the recipient (client or MbufManager) returns execution control to the supplying client. It is still theresponsibility of the recipient client to free

the mbuf chain.
2. The recipient client copies the unsafe data described into a safe mbuf chain before control returns back to the supplying client. Responsibilityfor freeing the unsafe mbuf chain still lies with the recipient client.

Ideally, all recipient clients would be capable of processing all mbuf chains they receive prior to returning control. Such clients could always use unsafe data in an efficient manner.

As a worst case fallback, whenever a client is supplied with an mbuf chain it always performs an 'ensure_safe' operation to ensure that the data is safe; this always entails data copying for unsafe mbuf chains.

In practise, "ensuring" (see the description of "ensuring" later) potentially unsafe mbufs chain to safe mbuf chains only when necessary is a reasonable compromise.

It is possible (indeed permitted) to allocate a safe mbuf and then generate a second reference to the same data with an unsafe mbuf. Data described in an mbuf chain may thus be "referenced" in almost the same way any other data may

be used in an unsafe mbuf. The difference is that an unsafe mbuf will be required for each describing mbuf in the chain, rather than a single unsafe mbuf to describe a single conceptual region of memory. Additional constraints are also imposed to ensure that the original mbuf chain is not freed before the unsafe mbuf chain. Freeing the safe mbuf chain after the return of the call where the unsafe mbuf chain is used will achieve correct operation.

In some traditional mbuf implementations, use is made of native memory management facilities to provide the ability to remap memory into "mbuf visible" regions, thus avoiding memory copying. This facility is not available in this specification. To some degree, unsafe data lessens this lack.

The 'm_inioff' and 'm_inilen' fields of an unsafe mbuf are initialised according to the allocation method used. See later for details.

Summary of mbuf field validity: client <=> mbuf manager

Field From mbuf To mbuf

===== ========= =======

m_next valid valid

m_list NULL invalid

m_off valid invalid

m_len valid invalid

m_inioff valid valid*

m_inilen valid valid*

m_type invalid invalid

m_sys1 opaque opaque

m_sys2 opaque opaque

m_sys3 opaque opaque

m_pkthdr invalid invalid

Notes:

Field: the name of a field within an mbuf structure.

From mbuf: an mbuf chain being passed from the MbufManager to a client.

To mbuf: an mbuf chain being passed from a client to the mbuf manager.

valid: the field meets the criteria stated within this document.

valid*: unsafe mbufs need not describe real memory in the underlying storage.

There is never any freeing of the underlying storage of an unsafe mbuf.

NULL: the NULL pointer.

invalid: any value may be present. No manipulation of such a value should ever be made. Either the field should be ignored or it should be initialised prior to use.

opaque: never read and never written by any client.

The 'm_inioff' and 'm_inilen' fields of a safe mbuf are never altered by a client - only read. The 'm_inioff' and 'm_inilen' fields of an unsafe mbuf may be altered by the client.

6 MbufManager sessions

The period of time when a client may allocate (and otherwise use) mbufs from the MbufManager is termed a "session". A session is initiated (opened) and terminated (closed) with SWI calls. A client must allocate and maintain an MbufManager control structure (an "mbctl" structure) for a duration encompassing a session (typically, the client has a static structure within it's data area for this purpose). A pointer to this structure is supplied to the MbufManager during both initialisation and termination calls and all direct entry points.

This structure contains, amongst other things, a set of function pointers. These function pointers provide direct entry points into individual routines within the MbufManager. They are initialised by the MbufManager during session initialisation and remain valid until session termination. All of the performance critical routines of the MbufManager (such as mbuf allocation and freeing) are accessed through these direct entry points, which incur considerably less overhead than SWI routines. These entry points are designed to permit the easy inter-operation of assembler and APCS code (such as that generated by the NorCroft C compiler), and roughly obey APCS. A list of entry/exit characteristics follows (using APCS register naming convention):

- a1 always points at an mbctl structure for all direct entry calls
- the processor must be in supervisor mode (but see MBC_USERMODE)
- a1-a4 are the only parameter registers
- a2-a4 and ip are corrupted by the call
- a1 is either the call result or corrupted
- other registers preserved by call
- the processor flags are preserved by the call
- no V set error convention (incompatible with APCS)
- in general, an error results in a1=0 on exit
- IRQ state preserved across call
- IRQs may be disable during calls
- IRQs may be enabled during calls ONLY if specifically documented
- FIQs assumed enabled on entry
- FIQs preserved across calls

Currently, no direct entry point routine will enable interrupts if they are disabled on entry.

C definition of an MbufManager control structure:

```
typedef struct mbctl
{
    /* reserved for MbufManager use in establishing context */
```

```
int opaque; /* MbufManager use only */
/* Client initialises before session is established */
size_t mbcsize; /* size of mbctl structure from client */
unsigned int mbcvers; /* client version of MbufManager spec */
unsigned long flags; /* */
size_t advminubs; /* Advisory desired minimum underlying block
size_t advmaxubs; /* Advisory desired maximum underlying block
size_t mincontig; /* client required min ensure_contig value */
unsigned long spare1; /* Must be set to zero on initialisation
/* MbufManager initialises during session establishment */
size_t minubs; /* Minimum underlying block size */
size_t maxubs; /* Maximum underlying block size */
size_t maxcontig; /* Maximum contiguify block size */
unsigned long spare2; /* Reserved for future use */
/* Allocation routines */
struct mbuf * /* MBC_DEFAULT */
(* alloc)
(struct mbctl *, size_t bytes, void *ptr);
struct mbuf * /* Parameter driven */
(* alloc_g)
(struct mbctl *, size_t bytes, void *ptr, unsigned long flags);
struct mbuf * /* MBC_UNSAFE */
(* alloc_u)
(struct mbctl *, size_t bytes, void *ptr);
struct mbuf * /* MBC_SINGLE */
(* alloc_s)
(struct mbctl *, size_t bytes, void *ptr);
struct mbuf * /* MBC_CLEAR */
(* alloc_c)
(struct mbctl *, size_t bytes, void *ptr);
/* Ensuring routines */
struct mbuf *
(* ensure_safe)
(struct mbctl *, struct mbuf *mp);
struct mbuf *
(* ensure_contig)
(struct mbctl *, struct mbuf *mp, size_t bytes);
/* Freeing routines */
void
(* free)
(struct mbctl *, struct mbuf *mp);
void
(* freem)
(struct mbctl *, struct mbuf *mp);
void
(* dtom_free)
(struct mbctl *, struct mbuf *mp);
void
(* dtom_freem)
(struct mbctl *, struct mbuf *mp);
```

```
    /* Support routines */
    struct mbuf * /* No ownership transfer though */
    (* dtom)
    (struct mbctl *, void *ptr);
    int /* Client retains mp ownership */
    (* any_unsafe)
    (struct mbctl *, struct mbuf *mp);
    int /* Client retains mp ownership */
    (* this_unsafe)
    (struct mbctl *, struct mbuf *mp);
    size_t /* Client retains mp ownership */
    (* count_bytes)
    (struct mbctl *, struct mbuf *mp);
    struct mbuf * /* Client retains old, new ownership */
    (* cat)
    (struct mbctl *, struct mbuf *old, struct mbuf *new);
    struct mbuf * /* Client retains mp ownership */
    (* trim)
    (struct mbctl *, struct mbuf *mp, int bytes, void *ptr);
    struct mbuf * /* Client retains mp ownership */
    (* copy)
    (struct mbctl *, struct mbuf *mp, size_t off, size_t len);
    struct mbuf * /* Client retains mp ownership */
    (* copy_p)
    (struct mbctl *, struct mbuf *mp, size_t off, size_t len);
    struct mbuf * /* Client retains mp ownership */
    (* copy_u)
    (struct mbctl *, struct mbuf *mp, size_t off, size_t len);
    struct mbuf * /* Client retains mp ownership */
    (* import)
    (struct mbctl *, struct mbuf *mp, size_t bytes, void *ptr);
    struct mbuf * /* Client retains mp ownership */
    (* export)
    (struct mbctl *, struct mbuf *mp, size_t bytes, void *ptr);
} dci4_mbctl;
```

Prior to establishing a session, the client initialises the following fields of the mbctl structure:

- size_t mbcsize;
- unsigned int mbcvers;
- unsigned long flags;
- size_t advminubs;
- size_t advmaxubs;
- size_t mincontig;
- unsigned long spare1;

The values a client initialises these fields to are defined as follows:

mbcsize: The size of the mbctl structure. This is the size of the

structure understood by the compiler/assembler at compilation time. Future versions of this specification may add other fields. In C, one might use "sizeof(struct mbctl)".

mbcvers: The version of the MbufManager specification that the client is implemented against. This is the major version times one hundred plus the minor version. Minor version number changes indicate bug fixes and the possible introduction of small and upwardly compatible changes. Major revision number changes indicate major and possibly not entirely backwardly compatible changes.

flags: This bitset supplies various pieces of information to the MbufManager. See the description of the SWI Mbuf_OpenSession (on page 101) SWI later on for details of suitable values to enter in this field.

advminubs: Advisory minimum underlying block size. This value advises the MbufManager of the smallest underlying block size that the client thinks appropriate for its requirements. Traditional mbuf clients might well use the orignal value of MLEN (112 in most cases) for this value. If no particular value seems appropriate, a client should set this field to zero.

advmaxubs: Advisory maximum underlying block size. This value advises the MbufManager of the largest underlying block size that the client thinks appropriate for its requirements. An ethernet device driver client might well use the ethernet MTU (maximum transmission unit) value of 1500. If no particular value seems appropriate, a client should set this field to zero.

mincontig: This specifies the maximum size the client will ever specify to the "contiguify" routine. If the MbufManager can never meet the value specified, it will refuse to open the session. If no particular value seems appropriate, a client should set this field to zero.

spare1: This field must be initialised to zero.

The contents of all other fields of the mbctl structure are irrelevant at the start of session initiation.

The next stage of session initiation is the issuing of an SWI Mbuf_OpenSession (on page 101) SWI call, supplying the address of this mbctl structure to the MbufManager as a parameter. If the session requested can be support by the MbufManager then it will initialise all other fields of the mbctl structure before returning. If the session cannot be supported, then no fields of the mbctl structure will be modified by the MbufManager and an error will be returned.

If no error is returned, the session is established and the client may use the direct entry points now available.

A session is terminated with the SWI Mbuf_CloseSession (on page 103) SWI call, supplying it the address of the same mbctl structure used to establish the connection.

int opaque;

The 'opaque' field is for the use of the MbufManager. It is initialised during session establishment. It must never be read or written by a client during a session.

All the direct entry points take a fixed first parameter of the address ofthe mbctl structure used to establish the session. This permits the MbufManager to establish any necessary context.

7 Allocation routines

```
struct mbuf * /* Parameter driven */
(* alloc_g)
(struct mbctl *, size_t bytes, void *ptr, unsigned long flags);
```

There is one general purpose allocation routine (alloc_g), and a number of more specialised allocation routines. All of these are accessed through the direct entry addresses contained in the initialised mbctl structure. The particular values the MbufManager supplies for the addresses of the direct entry point routines are chosen to be as optimal as possible for the clients indicated requirements. The functionality of these specific routines may be accessed through the general purpose routine; they are provided solely for performance reasons.

A successful allocation returns a chain of mbufs that satisfy all the criteria of the allocation. An unsuccessful allocation returns the NULL pointer. A NULL pointer indicates either a lack of some resource (typically mbufs or underlying storage) or a set of criteria that cannot be satisifed. If, for whatever reason, an allocation is constrained to a single mbuf, then the 'm_next' field of that mbuf will always be zero. In other words, whatever the entry flags may suggest, effectively, an mbuf chain is always returned.

The 'flags' bitset provides a list of constraints and deviations that are to be applied to an allocation.

The default allocation has all bits of the 'flags' bitset clear. In particular, the default allocation is for safe data. The defined bits are as follows:

MBC_DEFAULT 0x00000000ul

Bit(s)	Name	Meaning
0	MBC_UNSAFE	
1	MBC_SINGLE	
2	MBC_CLEAR	
2-31		Reserved, must be zero

Allocation consists of up to three internal phases:

- allocation
- clearing
- copying

Roughly speaking; the allocation phase always happens, the clearing phase happens when the MBC_CLEAR bit is set, and the copying phase happens when 'ptr' is not the NULL pointer.

If the allocation phase fails the clearing and copying phases are always skipped and the NULL pointer returned. The clearing and copying phases are not capable of failing (merely not happening).

8 The allocation phase

Table of different allocation options

MBC_UNSAFE bytes MBC_SINGLE type

0 0 ? 1

0 \= 0 0 2

0 \= 0 1 3

1 0 ? 4

1 \= 0 ? 5

Notes:

Column headings:

MBC_UNSAFE: the value of the MBC_UNSAFE bit

bytes: the value of the 'bytes' parameter

MBC_SINGLE: the value of the MBC_SINGLE bit

type: reference to detailed description

Column contents:

0: equal to zero, or flag clear

1: flag set

\=0: not equal to zero

?: any value (0 or 1 for bits)

1) MBC_UNSAFE = 0, bytes = 0, MBC_SINGLE = ?

The first available mbuf is chosen (so the setting of MBC_SINGLE is irrelevant). The actual size of the described data returned is unknown in advance, other than it is equal to or larger than the minimum underlying block size of the MbufManager (the 'minubs' field of the mbctl structure). 'm_len' and 'm_off' are set to reflect the underlying block (ie 'm_len' =

'm_inilen' and 'm_off' = 'm_inioff', respectively). Should a clearing or copying phase occur, then the value used for 'bytes' will be the value of 'm_len' in the newly allocated mbuf. Such copying might be the

start of a variable sized mbuf chain building algorithm.

m_next: NULL - always only one mbuf

m_list: NULL

m_off: describes underlying block

m_len: size of underlying block

m_inioff: describes underlying block

m_inilen: size of underlying block

2) MBC_UNSAFE = 0, bytes \= 0, MBC_SINGLE = 0

A chain of an arbitary number of mbufs is allocated, with a total described data size of 'bytes' bytes.

m_next: chain of mbufs returned

m_list: NULL

m_off: describes allocated memory

m_len: summed over the chain, gives 'bytes'

m_inioff: describes underlying block

m_inilen: size of underlying block

3) MBC_UNSAFE = 0, bytes \= 0, MBC_SINGLE = 1

Precisely one mbuf is allocated to describe the required number of bytes. It is possible for such allocations to fail due to not being able to locate an mbuf and underlying block with sufficient size.

m_next: NULL - always only one mbuf

m_list: NULL

m_off: describes allocated memory

m_len: 'bytes'

m_inioff: describes underlying block

m_inilen: size of underlying block

4) MBC_UNSAFE = 1, bytes = 0, MBC_SINGLE = ?

A single unsafe mbuf is allocated and set to describe no data. The value of 'ptr' is irrelevant. The MBC_CLEAR flag will be forced clear. 'm_off' and 'm_inioff' will describe the same value as the NULL pointer, and 'm_len' and 'm_inilen' will be zero. This is the only circumstance in which an mbuf (anywhere in an allocated chain) is allocated with zero in the 'm_len' field and returned directly from an allocation routine. (Note the anomoly for the 'copy' routine when asked to duplicate zero bytes.) The data described by an unsafe mbuf is not suitable for 'dtom', unless the data described is a "shadow" or "reference" to some previously allocated safe data, in which case 'dtom' will return the mbuf pointer for the original, safe, mbuf.

m_next: NULL - always only one mbuf

m_list: NULL

m_off: describes the NULL pointer

m_len: zero

m_inioff: describes the NULL pointer

m_inilen: zero

5) MBC_UNSAFE = 1, bytes \= 0, MBC_SINGLE = ?

A single unsafe mbuf is allocated. The 'm_len' field is set to the value of 'bytes'. 'm_off' is set to describe 'ptr', whatever the value of 'ptr'. This means suppling 'ptr' as the NULL pointer will cause the unsafe mbuf returned to have a non-zero byte count but for the data described to occur from address zero onwards. This is only useful when 'm_off' is later initialised to describe real memory. The data described by an unsafe mbuf is not suitable for 'dtom', unless the data described is a "shadow" or "reference" to some previously allocated safe data, in which case 'dtom' will return the mbuf pointer for the original, safe, mbuf.

m_next: NULL - always only one mbuf

m_list: NULL

m_off: describes 'ptr'

m_len: 'bytes'

m_inioff: describes 'ptr'

m_inilen: 'bytes'

9 The clearing phase

The clearing phase will set to zero all the underlying bytes in the allocated mbuf chain. The number of bytes zeroed is directly independent of the 'bytes' and 'ptr' values ('bytes' as zero indirectly dictates the number of bytes allocated). The clearing phase only occurs if the following conditions are all met:

1. The allocation phase succeeded
2. The MBC_CLEAR bit is set *
3. The MBC_UNSAFE bit is clear

*: The MBC_CLEAR bit can be cleared during the allocation phase. This clearing overrides any value the bit may have had on entry to the allocation routine and prevents the clearing phase from occurring.

---

10 The copying phase

The copying phase copies data from 'ptr' into the described data. The number of bytes copied is the number of bytes described by the mbuf chain. This is the value of 'bytes' supplied to the allocation routine if 'bytes' was non-zero, and the underlying block size if 'bytes' was zero. The copying phase may be viewed as importing data into an mbuf chain from "raw" memory.

A client cannot determine if copied bytes were cleared during the clearing phase or not. The copying phase only occurs if the following conditions are all met:

1. The allocation phase succeeded
2. The MBC_UNSAFE bit is clear
3. 'ptr' is not the NULL pointer

Summary of allocator routines and implicit or explicit 'flags' settings

Allocator Control over flags

alloc MBC_DEFAULT (0)

alloc_g parameter to the call

alloc_s MBC_SINGLE

alloc_u MBC_UNSAFE

alloc_c MBC_CLEAR

11 Freeing mbufs

```
void
(* free)
(struct mbctl *, struct mbuf *mp);
void
(* freem)
(struct mbctl *, struct mbuf *mp);
void
(* dtom_free)
(struct mbctl *, struct mbuf *mp);
void
(* dtom_freem)
(struct mbctl *, struct mbuf *mp);
```

Once an mbuf chain or an individual mbuf is finished with, it is freed and its resources become available for re-allocation. Variants of the free call are available that free either just a single mbuf (without examining the 'm_next' field of the mbuf supplied) or the entire chain it describes. The routine that frees a single mbuf is called 'free' and the routine that potentially frees multiple mbufs is called 'freem'.

Additionally, routines that perform the equivalent of a 'dtom' call followed by a freeing call are provided.

No action is performed by a free call if supplied the NULL pointer.

Summary of mbuf and mbuf chain freeing routines:

free Free single mbuf (ignores 'm_next' field)

freem Free entire mbuf chain (uses 'm_next' field)

dtom_free Performs 'dtom' action then behaves the same as 'free'

dtom_freem Performs 'dtom' action then behaves the same as 'freem'

---

12 Support and ensuring routines

```
struct mbuf *
(* dtom)
(struct mbctl *, void *ptr);
```

Under the right circumstances, it is possible to perform a transformation from the address of any byte described by an mbuf to the address of the mbuf describing that byte. This transformation is performed with the 'dtom' routine. The presence of a 'm_next' field, and the lack of a hypothetical 'm_prev' field means that 'dtom' provides access to a portion of the conceptually described data, starting with the first byte described by the mbuf that describes the supplied address, and extending to the end of the mbuf chain. The client cannot directly determine if the mbuf returned by 'dtom' is the first mbuf in a chain or an mbuf part-way along a chain.

The required circumstances for 'dtom' to operate correctly are that the described data is safe.

Applying 'dtom' to an unsuitable address will return the NULL pointer. The NULL pointer is always an unsuitable address for 'dtom'.

The 'dtom' and 'mtod' transformations are not entirely symmetrical. 'dtom' will always return the address of the mbuf owning the underlying storage referenced, indepedent of the number of unsafe mbufs also referencing that storage.

The transformation performed by 'dtom' is necessarily based on the address supplied; this includes whether the address is within the region(s) of memory controlled by the MbufManager or not. For this reason, if a safe mbuf chain has been "shadowed" with an unsafe mbuf chain, then 'dtom' will always return the original safe mbuf. Further, freeing the chain returned by 'dtom' will free the original, safe mbuf chain, leaving the unsafe mbuf chain describing (through reference) now unknown data (this certainly warrants the description "unsafe data" and is one of the reasons for the 'ensure_safe' routine, although in this particular case it would be far too late to make the call to 'ensure_safe'). It is the clients responsibility to ensure that such problems do not occur (typically through appropriate interface contracts between clients).

```
struct mbuf *
(* ensure_contig)
(struct mbctl *, struct mbuf *mp, size_t bytes);
```

For a protocol client, the removal of protocol layers (headers or trailers) when a packet passes up a protocol stack is often made easier if all of the bytes constituting a particular header level are contiguous. (This permits the protocol to conceptually overlay the

received packet with a structure describing the protocol header.) The 'ensure_contig' routine is used to ensure such "contiguousness" requirements are met by an mbuf chain, and "contiguifies" the specified number of bytes at the head of the mbuf chain supplied. Described data that is contiguified is also always at least word aligned for the first byte. This helps with the overlaying of word orientated structures.

```
struct mbuf *
(* ensure_safe)
(struct mbctl *, struct mbuf *mp);
```

If safe data is required and the safeness of an mbuf chain is uncertain, then the 'ensure_safe' routine may be used to ensure that data in the returned mbuf chain is safe.

In both cases (that is, 'ensure_contig' and 'ensure_safe'), "ensure" is used as a technical term meaning:

IF

the data (mbuf chain) meets the required condition

THEN

return the data unmodified

ELSE

return some data that does meet the required condition

FI

The process of generating data that does meet the required condition involves allocating one or more mbufs with appropriate constraints (equivalent to MBC_SINGLE, and MBC_DEFAULT allocation constraints) and replacing existing mbufs in the chain with these new mbufs. Any existing mbuf that is replaced is automatically freed.

Any of the ensure routines may fail if they have to perform allocations and there is a lack of resource. If this happens, then the entire mbuf chain supplied is freed and the NULL pointer is returned.

If the ensure operation does not fail, then the returned mbuf chain will meet the desired criteria. Whether an ensure operation fails or succeeds, the client must use the returned mbuf chain. There is an ownership transfer to the MbufManager whilst the ensure operation is performed and then another ownership transfer back to the client of the resulting mbuf chain that meets the criteria - these two mbuf chains may happen to be the same, but the loss and regain of ownership means a client cannot tell.

If the mbuf chain meets the indicated criteria on entry, then the ensure routine cannot fail and will always return the supplied pointer without modification.

Under some circumstances the 'ensure_contig' routine may be able to avoid an allocation by moving data around within the existing mbuf chain (this requires some underlying bytes not described by the mbuf chain itself). If these circumstances apply, they cannot generate a failure condition themselves. Thus, if only shuffling is required, then 'ensure_contig' cannot fail, but if shuffling and allocation are required, then 'ensure_contig' can fail through lack of resources for the allocation.

Note that an mbuf chain may contain a mixture of mbufs, each with its own characteristics. For example, an individual mbuf may contain safe or unsafe data, it may meet some contiguity requirement or not and there may or may not be underlying bytes described. An mbuf chain may be composed of any mixture of such mbufs. The ensure routines arrange that all necessary mbufs within a chain meet the desired criteria.

'ensure_contig' cannot operate on unsafe mbufs and will fail (ie freethe supplied chain and return NULL) if asked to do so.

'ensure_safe' returns an mbuf chain where every byte described by it is known to be safe.

'ensure_contig' returns an mbuf chain where the first 'N' bytes are known to be contiguous.

```
int
(* any_unsafe)
(struct mbctl *, struct mbuf *mp);
int
(* this_unsafe)
(struct mbctl *, struct mbuf *mp);
```

A client may determine if an mbuf chain has any unsafe data in it with the 'any_unsafe' routine. This returns 0 for either no unsafe data (ie all data safe) or if supplied the NULL pointer. It returns 1 if the mbuf chain supplied contains unsafe data. The 'this_unsafe' returns the same values but only examines the mbuf supplied - that is, it does not follow the 'm_next' field.

```
size_t
(* count_bytes)
(struct mbctl *, struct mbuf *mp);
```

The number of bytes described by an mbuf chain may be quickly determined with the 'count_bytes' routine. If supplied the NULL

pointer, then 0 is returned.

```
struct mbuf *
(* cat)
(struct mbctl *, struct mbuf *old, struct mbuf *new);
```

One mbuf chain may be appended to the end of another mbuf chain with the 'cat' routine. The mbuf chain that gets appended to is the first mbuf parameter ('old'). The mbuf chain to be appended is the second mbuf parameter ('new'). Note that there is no ownership transfer of the second mbuf parameter. If 'old' is the NULL pointer, the 'new' is returned without examination. If 'old' is not the NULL pointer and 'new' is the NULL pointer then 'old' is returned without any modifications made to it.

```
struct mbuf *
(* trim)
(struct mbctl *, struct mbuf *mp, int bytes, void *ptr);
```

The 'trim' routine is used to remove bytes from either the head or the tail of an mbuf chain. It may optionally copy the bytes described to another piece of memory (performing a "flattening" operation in the process). All trimming is performed by adjusting 'm_off' and 'm_len'. No mbufs are removed (unlinked and freed) from the either the head or the tail of the chain. If 'bytes' is greater than zero, then it specifies the number of bytes to remove from the head of the mbuf chain. If 'bytes' is zero then no alterations are performed and no data is copied. If 'bytes' is less than zero, then the absolute value of 'bytes' is the number of bytes to remove from the tail of the mbuf chain.

If the NULL pointer is supplied for the mbuf chain, no operations are performed. If 'ptr' is not the NULL pointer, then any bytes "trimmed" from the mbuf chain will be copied into the supplied area of memory. If 'ptr' is the NULL pointer, then no data copying is performed and only a trimming operation occurs. The magic value M_COPYALL may be used for the 'bytes' parameter to indicate the entire mbuf chain. This is only useful if a copy is also being performed, although it will always correctly set the mbuf chain to describe zero bytes. If the number of bytes to trim (after tail adjustment

if applicable) is greater than the number of bytes described by the mbuf chain, then behaviour is as if M_COPYALL was supplied for a trim byte count.

```
struct mbuf *
(* copy)
(struct mbctl *, struct mbuf *mp, size_t off, size_t len);
struct mbuf *
(* copy_p)
(struct mbctl *, struct mbuf *mp, size_t off, size_t len);
```

```
struct mbuf *
(* copy_u)
(struct mbctl *, struct mbuf *mp, size_t off, size_t len);
```

The 'copy' routine is used to duplicate a portion of an mbuf chain. An mbuf chain is always allocated, even if it eventually describes zero bytes. This distinguishes successful allocations that required no data from unsuccessful allocations and makes the behaviour of the 'len' parameter more orthogonal. The returned mbuf chain will have a minimum byte count of zero and a maximum byte count equal to the byte count of the supplied mbuf chain. The portion

of the mbuf chain copied is the intersecting region between the described data of the supplied mbuf chain and the region starting 'off' bytes into the supplied chain and continuing for 'len' bytes. The first byte described by an mbuf chain is byte 0.

The 'alloc' allocator routine is used. This means the returned mbuf chain will be safe, may have any number of mbufs, is directly suitable for the 'dtom' routine and any unused underlying storage may hold random values. If a lack of resources occur, the NULL pointer is returned. If 'mp' is the NULL pointer, then the NULL pointer is returned. If 'len' holds the magic value M_COPYALL, then all remaining bytes in the mbuf chain from 'off' onwards are copied. M_COPYALL has the value 0x7f000000, in hexadecimal in C notation. The mbuf chain supplied will not be altered.

The 'copy_p' routine behaves as 'copy' does, except that the m_type, m_flags and m_pkthdr fields are assumed to contain significant information. This prevents two small mbufs with different m_type values from being merged into a single larger mbuf. This does not prevent a single mbuf being copied into more than one mbuf; this would replicate the m_type field. If the usage of m_type is more sophisticated than the simple 'tagging' discussed above, then it is likely than the client will require a custom copying routine.

The 'm_copy_u' routine produces an unsafe copy of an mbuf chain. This means no new underlying storage will be allocated. The chain returned will have the same number of mbufs as that supplied. The m_type, m_flags and m_pkthdr fields are replicated from the old chain to the new chain. The 'alloc_u' routine is used to perform the new allocations.

```
struct mbuf *
(* import)
(struct mbctl *, struct mbuf *mp, size_t bytes, void *ptr);
```

Data may be copied from raw memory into an mbuf chain with the 'import' routine. Copying starts at 'ptr' for reading and the first byte described by the mbuf chain for writing. Copying proceeds until

either the entire mbuf chain has been filled or 'bytes' bytes have been read. The magic value M_COPYALL may be used to indicate that the entire mbuf chain should be filled. The return value is the mbuf chain supplied. If either the mbuf chain or 'ptr' is the NULL pointer then no operation is performed.

```
struct mbuf *
(* export)
(struct mbctl *, struct mbuf *mp, size_t bytes, void *ptr);
```

Data may be copied from an mbuf chain into raw memory with the 'export' routine. Copying starts at the first byte described by the mbuf chain for reading and 'ptr' for writing. Copying proceeds until either the entire mbuf

chain has been copied or 'bytes' bytes have been written to raw memory. The magic value M_COPYALL may be used to indicate that the entire mbuf chain should be copied to raw memory. The return value is the mbuf chain supplied. If either the mbuf chain or 'ptr' is the NULL pointer then no operation is performed.

Summary of mbuf chain ownership transfer for direct entry points

Routine Category

alloc Fresh

alloc_g Fresh

alloc_u Fresh

alloc_s Fresh

alloc_c Fresh

ensure_safe Release and gain

ensure_contig Release and gain

free Release

freem Release

dtom_free Release

dtom_freem Release

dtom No transfer

any_unsafe No transfer

this_unsafe No transfer

count_bytes No transfer

cat No transfer

trim No transfer

copy Fresh *

copy_p Fresh *

copy_u Fresh *

import No transfer

export No transfer

Notes:

Fresh: A new mbuf chain is generated and the client receives ownership of this new chain when it receives the chain itself.

Fresh *: Ownership of the supplied mbuf chain remains with the caller.

Release: Ownership of the mbuf chain supplied is passed to the MbufManager.

Release and gain: Ownership of the mbuf chain supplied is passed to the MbufManager. Ownership of the returned chain is passed to the client at the same time as the mbuf chain itself.

No transfer: No ownership transfer occurs (and hence no linkage changes are performed), but the MbufManager does expect a valid mbuf chain that remains static during the period of the call.

---

13 SWI Entry points

All SWIs defined here obey the RISC OS convention of indicating success by returning with the V flag clear and failure by returning with the V flag set and r0 pointing at a standard RISC OS error block. For convenience, this is omitted from the definition of each SWI individually. All SWIs defined here also obey the standard convention regarding interrupts, unless otherwise specified. That is: IRQ interrupt state is preserved across the SWI, although it may be enabled during the SWI. FIQ interrupt state is assumed enabled and not altered. This is described as the normal behaviour in the text below.

Mbuf_OpenSession (SWI &4A580)

Establishes a session with the MbufManager

On entry

R0 = Address of an 'mbctl' structure

On exit

R0 - R9 preserved

Interrupts

Interrupts are preserved, but may be enabled during call
Fast interrupts are preserved

Processor mode

Processor is in undefined mode

Re-entrancy

Not defined

Use

This SWI is used by clients to establish a session with the MbufManager. This informs the MbufManager of the clients mbuf requirements, and informs the client of the direct entry point addresses into the MbufManager that are appropriate for its mbuf requirements. A certain amount of validation of the proposed session is performed by the MbufManager. This may result in modifications of behaviour within the MbufManager and it may also result in a refusal to accept a session, with an error being returned in the normal fashion.

The flags field of the mbctl structure provides additional information to the MbufManager about the required session. The only flag with a defined meaning at present is the MBC_USERMODE flag.

The MBC_USERMODE flag may be specified in the flags field to request that the direct entry points be suitable for user mode calling. If this is not specified, then the direct entry points must be called in supervisor mode. If MBC_USERMODE is specified, then the direct entry points supplied must be called in user mode. This permits

normal user mode applications to interact with the MbufManager. Care must be taken with unsafe data to ensure that the memory described is valid when the user mode application is not the current application, and hence might not have its memory currently 'mapped in'.

All other bits in the flags bitset should be zero.

MBC_USERMODE

0 - This client requires supervisor mode direct entry points.

1 - This client requires user mode direct entry points.

Errors:

As normal.

"MbufManager unsuitable for client"

Notes:

The addresess of the routines supplied for the direct entry points may vary according to the requirements indicated by a client. In some circumstances, the MbufManager is able to supply a "null" routine: ie an immediate return.

Further details of the fields and their uses is found elsewhere in this document.

Mbuf_CloseSession (SWI &4A581)

Terminates a session with the MbufManager

On entry

R0 = Address of an 'mbctl' structure previously supplied

On exit

R0 - R9 preserved

Interrupts

Interrupts are preserved, but may be enabled during call
Fast interrupts are preserved

Processor mode

Processor is in undefined mode

Re-entrancy

Not defined

Use

This SWI is uses to terminate a session that has previously been successfully created with the Mbuf_OpenSession SWI.

Errors:

As normal.

"No such session"

Notes:

This SWI is used to terminate a session. The address supplied must be the same as that supplied to Mbuf_Init when the session was created. Whether an error is returned or not, the client must consider the session closed after issuing this SWI.

Mbuf_Memory (SWI &4A582)

Controls maximum memory usage

On entry

R0 = New limit to use in bytes, or 0 to read current limit

On exit

R0 = Approximate limit active when SWI issued
R1 - R9 preserved

Interrupts

Interrupts are preserved, but may be disabled during call
Fast interrupts are preserved

Processor mode

Processor is in undefined mode

Re-entrancy

Not defined

Use

Provides a means to limit the maximum amount of memory that may be claimed by the MbufManager for mbuf and underlying storage.

Errors:

As normal.

Notes:

If zero is supplied as the new desired limit, then the limit is not altered and only an examination is performed. Limits are specified in bytes. They are approximate figures only (due to the underlying allocations being performed in granularities larger than a single byte). They are normally within one about kilobyte of the actual value. A new, larger limit does not cause more memory to be automatically claimed. The MbufManager may attempt to dynamically maintain an appropriately sized free pool. This limit provides a ceiling to any dynamic fluctuations. A user interface might

well use a granularity of four kilobytes.

Mbuf_Statistic (SWI &4A583)

Returns statistics for the MbufManager

On entry

None

On exit

None

Interrupts

Interrupts are undefined
Fast interrupts are undefined

Processor mode

Processor is in undefined mode

Re-entrancy

Not defined

Use

This SWI provides an entry point that conforms to the DCI4 Statistic Interface. Please refer to that document for further details.

⚠ FIXME: Provide links to the Statistics documentation

⚠ FIXME: Provide at least the register call information

Mbuf_Control (SWI &4A584)

General purpose control interface for MbufManager

On entry

R0 = Reason code:

Value	Meaning
0	SWI MBuf_Control 0 (on page 108)

On exit

R1 - R9 = Dependant on reason code

Interrupts

Interrupts are preserved
Fast interrupts are preserved

Processor mode

Processor is in undefined mode

Re-entrancy

Not defined

Use

This is a general purpose control interface to the MbufManager. Different implementations may implement different control calls.

Notes:

Issuing this SWI with a reason code of 0 is a good method of checking for the presence of the MbufManager.

Mbuf_Control 0 Version (SWI &4A584)

Read version number of the MbufManager

On entry

R0 = 0 (reason code)

On exit

R0 = MbufManager version × 100
R1 - R9 preserved

Interrupts

Interrupts are preserved
Fast interrupts are preserved

Processor mode

Processor is in undefined mode

Re-entrancy

Not defined

Use

This SWI reason is used to read the version of the MbufManager.

14 Service calls

Service_MbufManagerStatus (Service Call &A2)

MbufManager state change notifications

On entry

R1 = Reason code (&A2)
R2 = Sub-reason code :

Value	Meaning
0	MbufManager has started (on page 110)
1	MbufManager is shutting down (on page 111)
2	MbufManager wishes to reclaim buffers (on page 112)

On exit

R1 preserved

Use

The MbufManager issues service calls to notify clients and potential clients of desired and actual state changes. The service call used is Service_MbufManagerStatus (service call number 0xa2, in C). A reason code is passed in r0 to indicate the reason for the service call.

Related services

Service_MbufManagerStatus 0 (on page 110)
Service_MbufManagerStatus 1 (on page 111)
Service_MbufManagerStatus 2 (on page 112)

Service_MbufManagerStatus 0 Started (Service Call &A2)

MbufManager has started

On entry

R1 = reason code (&8A)
R2 = Sub-reason code (0)

On exit

R1 preserved

Use

MbufManager has started and is now available for use. It is possible to issue SWIs to the MbufManager as soon as this service call has been seen (it is issued from a callback to ensure this is possible).

Related services

Service_MbufManagerStatus 1 (on page 111)

Service_MbufManagerStatus 1 Stopping (Service Call &A2)

MbufManager is shutting down

On entry

R1 = reason code (&8A)
R2 = Sub-reason code (1)

On exit

R1 preserved

Use

The MbufManager is finishing. There are no open sessions if this reason code is used. The MbufManager will refuse to die if there are any open sessions.

Related services

Service_MbufManagerStatus 0 (on page 110)

Service_MbufManagerStatus 2 Scavenge (Service Call &A2)

MbufManager wishes to reclaim buffers

On entry

R1 = reason code (&8A)
R2 = Sub-reason code (2)

On exit

R1 preserved

Use

This reason code is used to indicate that the MbufManager is running short of allocatable memory and any clients with allocated data that may be easily recreated (such as cached data) should release this memory (ideally) before returning from the service call.

Related APIs

None

15 The life and death of an MbufManager

The components necessary to form a working DCI4 environment may be loaded in a number of different orders. To permit sensible, defined behaviour for all of these orders, the MbufManager and all device drivers provide mechanisms to announce their arrival and departure, and for their presence to be determined through a polled action.

The MbufManager announces its arrival with the Service_MbufManagerStatus service call with a reason code of MbufManagerStatus_Started (on page 110). An client that cannot detect the presence of the MbufManager when it (the client) loads (via an SWI Mbuf_Control (on page 107) SWI call, for example) should place itself in a 'pre-active' state and await this service call. The MbufManager will respond to SWIs when the service call is issued.

Any attempt to kill the MbufManager when there are open sessions will be refused.

If the MbufManager is requested to die and there are not open sessions, it will issue the Service_MbufManagerStatus 1 (on page 111) reason code in a Service_MbufManagerStatus service call to notify potential clients of this.

Clients that can remain inactive without an open session with the MbufManager should do so, to give the user more flexibility should it be necessary to upgrade the MbufManager. Either way, killing all modules with open sessions should always permit the MbufManager to be killed.

16 Glossary

A "NULL pointer" is a pointer with all bits clear. It is never a valid address of examination or modification for virtually all programs under virtually all circumstances.

A linked list of mbufs constructed with the 'm_next' field is referred to as a "chain of mbufs".

A linked list of mbufs constructed with the 'm_list' field is referred to as a "list of mbufs" or a "list of mbuf chains".

A chain and a list may consist of just one mbuf. The ends of the chain and list are indicated by the NULL pointer.

Lists of chains are constructed, but not vice versa (certainly within this specification).

A1 Appendix - DCI4 MbufManager client interface contract

Ownership of an mbuf chain grants permission to examine and modify the described data, alter the order of the chain, etc. It also brings the responsibility to either pass ownership to another client or to ensure that the mbuf chain is (eventually) freed.

In short, ownership permits useful things to be done with an mbuf chain, and carries with it the responsibility to free the data.

When a client obtains a pointer to an mbuf chain from outside itself (ie from another client or from the MbufManager), it is deemed to have taken ownership of that chain. When it supplies that pointer to another client or the MbufManager, it has lost ownership. An mbuf chain is never owned by more than one client.

The presence of asynchronous execution mechanisms (such as interrupts) requires a more precise definition. During ownership transfer, there is a transient period where the relinquishing client has called the recipient client, but the call has not yet returned. Depending upon the precise definition, during this period of time, the mbuf chain could be viewed as being owned by zero, one or two clients. The definition for this specification is that the mbuf chain is owned by zero clients.

This requires the relinquishing client to take whatever steps are necessary to ensure that it cannot continue to access an mbuf chain before calling the recipient client. An example of such steps might be to remove the mbuf chain pointer from a list of such pointers examined by an interrupt routine of the relinquishing client or to set a semaphore of some form.

The mbuf pointer supplied to the call that transfers ownership should be the only copy of that pointer value that the relinquishing client has.

In short, once ownership transfer is committed to, the original owner has already lost ownership.

[Is this adequate. Does it supply the necessary degree of precision?]

Whenever one client passes an mbuf chain that describes unsafe data (an "unsafe mbuf chain") to another client, it pledges that the data will remain valid until the recipient client returns through the thread of control that supplied the unsafe mbuf chain.

If the recipient client were to retain the supplied mbuf chain beyond this point in time, then it might describe invalid data (it is possible

that exceptions will arise if an attempt is made to access the described data, for example).

A device driver never supplies a protocol module an unsafe mbuf chain (this is a DCI4 protocol to device driver restriction only).

[I dislike this restriction, but it is necessary for the sweeping 'dtom' statement to apply.]

When a device driver supplies a received packet chain to a protocol module, the m_type field of the first mbuf holds MT_HEADER (2) and all the other m_type fields hold MT_DATA (1).

When a protocol module supplies an mbuf chain to a device driver, the m_type field of all described mbufs is invalid and must not influence the behaviour of the device driver.

Summary of mbuf field validity: dci4 client <=> dci4 client

Field From protocol From device driver

m_next valid valid

m_list valid* valid*

m_off valid valid

m_len valid valid

m_inioff valid valid

m_inilen valid valid

m_type invalid valid

m_sys1 opaque opaque

m_sys2 opaque opaque

m_flags invalid invalid

m_pkthdr invalid invalid

Notes:

Field: the name of a field within an mbuf structure.

From protocol: an mbuf chain for transmission

From device driver: a newly received mbuf chain for protocol processing.

From protocol: an mbuf chain being passed from a protocol to a device driver for transmission.

valid: the field meets the criteria stated within this document.

valid*: This is a list of mbuf chains. A protocol can avoid fragmenting datagrams down to the device driver mtu by using a list of unsafe mbufs to shadow the real data. The device driver uses 'ensure_safe' if it needs to retain an mbuf beyond the transmit call returning.

invalid: any value may be present. No manipulation of such a value should ever be made. Either the field should be ignored or it should be initialised prior to use.

opaque: never read and never written by any client.

The 'm_inioff' and 'm_inilen' fields of a safe mbuf are never altered by a client - only read. The 'm_inioff' and 'm_inilen' fields of an unsafe mbuf may be altered by the client.

A2 Appendix - Supplementary clarifications

There are no supplementary clarifications known to be required at present.

Document information

Maintainer(s): Charles Ferguson <gerph@gerph.org>

History:

Revision	Date	Author	Changes
0.01 Issue 1	14 Sep 1994	Borris	● A draft specification containing most necessary details and only a few contradictions.
0.01 Issue 2	15 Sep 1994	Borris	● An internally coherent document that specifies something that can useful be implemented. A example "mbuf.h" header file illustrates how C clients might be started. This file also contains a set of wrappers to make interfacing existing traditional mbuf source code easier (such as the internet source code).
0.02 Issue 1	21 Sep 1994	Borris	● All safe mbufs are now dtom'able. ● Better specified direct entry point register conventions. Basically, these entry points are now suitable for APCS routines - a2-a4 trashed on exit and maybe a1. ● Rewrote the description of allocation to remove abiguities and increase understandability. ● Added a table summarising owner transfer for the direct entry points.
0.02 Issue 2	Sep 1994	Borris	● Minor corrections. Stricter definitions for most of the support routines. ● Fixed duplicate maxcontig field definition in mbctl.
0.99 Issue 1	1994	Borris	● Removed the restriction that the copying phase of an allocation only occurred if there was a data import to be performed (ie ptr != NULL). ● Rename maxcontig_c to mincontig and maxcontig_m

			to maxcontig in struct mbctl. ● mbctl.trim returns the mbuf chain it was given for convenience. ● m_type field is initialised to the value one for each mbuf allocated. [this has subsequently been taken out again - see below] ● Clarified m_list field after allocation.
1.00 Issue 1	14 Nov 1994	Borris	● Service_MbufManagerStatus (0xa2) defined and added. ● MBC_USERMODE flag for getting user mode direct entry points added. ● Details added in the OpenSession SWI documentation. ● Added section on life and death of an MbufManager to detail startup interactions (the mmintro document that provides an introduction of DCI2 to DCI4 mbuf usage has also been expanded similarly). ● Definitions for MBC_DONTWAIT and MBC_DTOMABLE added, although these flags are not currently used (they probably will be one day, though). ● Reduce currently outstanding section correspondingly. ● Extended legal stuff at head of document. ● m_type from allocation is undefined - device drivers are expected to initialise it before supplying received data to protocols though. Field means nothing when data moves from protocol to device driver. ● m_copy variants called m_copy_p (preserve m_type information) and m_copy_u

			(make an unsafe copy) have been added and changes to reflect this made to header files and structure definitions. ● Added MT_HEADER and MT_DATA related comments to the DCI4 protocol/device driver interface contract section. ● Corrections to dci4 protocol/ client contract summary table.
1.00 Issue 2	Nov 1994	Borris	● Removed comment about ensure_contig being able to do its job at the tail end of an mbuf chain as well as the head. Added comment about unsafe mbufs not being suitable for ensure_contig to operate on.
1.01 Issue 3	Apr 1997	KBracey	● Added information about m_pkthdr and m_flags fields.
1.02CJF	08 Aug 2020	Gerph	**Restructure as PRM-in-XML** ● Formatting changes have been made to use structured data formats, with tables of structures in addition to the C structure definitions.
1.03CJF	01 Nov 2022	Gerph	**Tidy up XML** ● Tidied up the XML and some of the structure to better match PRM-in-XML expectations and lint cleanly.

EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSER. NOT INTENDED FOR USE IN LIFE CRITICAL APPLICATIONS.

DCI Statistics

Introduction

The maintenance of statistics such as "packets transmitted" for a network interface can greatly assist the system administrator (or user) in a variety of circumstances.

Traditionally, support for such statistics gathering has been irregular and generally limited to direct screen printing. This specification provides a more controlled method of presenting statistics to the user, and permits a single generic display program to cope with all compliant modules.

The following characterise the intentions of this interface specification:

- **Plus points**
  - Simplicity of implementation
  - Single generic statistics display program
  - No essential central information that needs frequent updating
  - Information is for presentation to humans primarily
  - Atomic operations without any latching mechanisms
  - Well-behaved with multiple gatherers
  - Well-behaved if a supplier unexpectedly disappears
  - Virtually stateless communication
- **Minus points**
  - Not re-entrant - callbacks or user mode code only
  - Restrictive string model
  - Polled interface, rather than event driven
  - Only simple flat structure to statistics
  - Only really suitable for human display

Technical Details

Basics of operation

A piece of software that wishes to offer statistics is termed a "supplier". A piece of software that obtains statistics from a supplier and presents them to the user is termed a "gatherer", and the operation it performs is often referred to as "enumerating the statistics". There are typically more suppliers than gatherers in a system.

As part of its normal operations, a supplier maintains a set of statistics. A gatherer, typically upon user invocation, enumerates the available suppliers, enumerates the statistics supplied by each supplier and then presents this information to the user. More dynamic presentation is also possible. A supplier is passive (other than its normal operations), whilst the gatherer actively seeks out the information to be presented.

The first stage of enumeration (determining what suppliers are present) is performed with a service call (Service_StatisticEnumerate (on page 139)), and the second stage (determining what statistics are available and their values) is performed through a SWI whose SWI number is obtained during the first stage of enumeration (SWI StatisticsProvider_Statistics (on page 140)). This SWI number is used for all stages of operation apart from enumeration itself.

Each statistic has a number of different pieces of information associated with it, forming its description, and a value. The description provides the necessary information to access and manipulate the value, including the number of bytes of storage required for the value, it's type, format and desired presentation. The data types supported are boolean, integer and string. An unused data type also provides for padding.

Descriptions and statistics are read in ranges from a lower statistic number to an upper statistic number, inclusive. The first statistic of any supplier is zero. The highest statistic number of a supplier is obtained when the SWI number for a supplier is obtained. A range is always processed atomically by the supplier, ensuring consistent values and presentation to a gatherer when so required. A supplier indicates the volatility of its statistics and typically attempts to group statistics into ranges of the same volatility.

Enumerating suppliers

The list of available suppliers is determined with a single service call, Service_StatisticEnumerate (on page 139).

On receiving this service, each supplier allocates a small structure

from the RMA, initializes it and links it into the linked list whose head is pointed to by R0. Whether the supplier adds an entry at the head or tail of the list does not matter, although it is recommended that the entries be added to the head for simplicity. If a gatherer requires any specific order, then it should explicitly sort the suppliers list itself. A gatherer should not cache SWI values across invocations of suppliers, as a module might choose a dynamic SWI numbering scheme if it offers multiple effective suppliers.

Statistics Provider Structure

In C, this structure is defined as follows:

```
typedef struct spctl
{
    struct spctl       *next;           /* Next structure in list *
    unsigned int        i_version;      /* Interface version */
    unsigned long       features;       /* Combination of SF_ value
    unsigned int        swinumber;      /* The SWI SA_DESCRIBE/SA_R
    unsigned int        max_stat;       /* Highest stat number (inc
    unsigned int        type;           /* Acorn assigned supplier
    unsigned int        s_version;      /* Supplier version */
    char               *module;         /* Module name (short one)
    char               *title;          /* Title string - short */
    char               *description;    /* Descriptive string - lon
    unsigned char       reset[8];       /* Unique for each invocati
} dci4_spctl;
```

Offset	Name	Contents
+0	next	This field is used to construct the linked list of suppliers for the enumeration operation. A value of zero indicates the end of the list.
+4	i_version	The DCI4 Statistic Interface version that the supplier is implemented against.
+8	features	A bitset of flags defining optional aspects of a supplier. No features are currently defined and the the value must be set to 0.
+12	swinumber	The SWI number through which all other communication with the supplier is performed.
+16	max_stat	The highest statistic number, inclusive, that the supplier provides. This implies a supplier must always supply at least one statistic.
+20	type	This field is available to provide some classification of suppliers. It is the only centrally administered resource in this specification, and its use is optional. The currently defined values are as follows:

Value	Name	Meaning
0	SPT_GENERAL_SUPPLIER	Use this if no other suitable type
1	SPT_NETWORK_PROTOCOL	A DCI4 protocol module
2	SPT_NETWORK_DRIVER	A DCI4 device driver module
3	SPT_MBUF_MANAGER	The DCI4 mbuf manager module

+24	s_version	This is the version number of the supplier module itself (as opposed to the version number of an interface it conforms to). It is provided for the convenience of the user.
+28	module	This is the supplier's module title (as opposed to its help string).
+32	title	This is a short descriptive string to identify the supplier to the user. The gatherer only contracts to print at least the first twenty characters of this string.

Offset	Name	Contents
+36	description	This is a longer description of the supplier that should convey it's purpose to the user. The gatherer only contracts to print at least the first fifty characters of this string.
+40	reset	This 8 byte field is initialised with a unique value each time the supplier is initialised. It permits the gatherer to spot a supplier that has just re-initialised. The time and date at which the supplier initialises are recommended values to write here. The gatherer performs an 8 byte equality test and otherwise assumes no further format information about these bytes.

When a gatherer has finished interacting with a supplier, it should free the allocated structures back into the RMA.

Describing and obtaining statistics

Apart from the enumeration phase, all communication with a supplier is performed through the single SWI number obtained during the enumeration phase (described as SWI StatisticsProvider_Statistics (on page 140)).

SA_DESCRIBE (0) obtains descriptions of statistics, whilst SA_READ (1) obtains the actual values of statistics. It is necessary for first obtain the description of a statistic before reading it, as the description is the only way the gatherer can obtain the size of the statistic value and know how much buffer space to allocate.

In essence, both these actions enumerate information about a range of statistics into the buffer supplied. This is performed atomically, where necessary, by the supplier. In both cases, the operation starts from the first statistic and proceeds in steps of one and stops once the last statistic has been processed. Output is placed into the buffer supplied. If this buffer is not big enough for the requested range of statistics, then processing will stop when it is no longer possible to write all the necessary information for a statistic (i.e. if the buffer supplied is not big enough, the number of unused bytes will be less than the size of a statistic description). Return values of the number of statistics processed and the number of bytes of the buffer used are returned. The buffer must be word aligned in memory.

Statistic Description Structure

Each statistic is described with a fixed sized buffer, described in C as follows:

```
typedef struct stdesc
```

```
{
    unsigned int            type;           /* ST_ series */
    unsigned int            format;         /* SxF_ series */
    unsigned int            presentation;   /* SxP_ series */
    unsigned int            size;           /* Measured in bytes */
    unsigned int            volatility;     /* SV_ series */
    char                   *name;           /* String is static */
    unsigned int            name_tag;       /* See specification */
    unsigned int            spare;          /* Unused. Always zero
} dci4_stdesc;
```

Offset	Name	Contents
+0	type	This defines the type of statistic. Possible values are:

Offset	Name	Contents		
		Value	**Name**	**Meaning**
		0	ST_UNUSED	ST_UNUSED is used by a supplier to reserve a statistic number. Such a statistic always has a value occupying zero bytes, and hence can always be enumerated, no matter how many bytes are left in the called supplied buffer. A statistic with a type of ST_UNUSED should always be skipped over when printing and enumerating - none of the fields beyond the type and size fields are valid for unused statistics.
		1	ST_BOOLEAN	ST_BOOLEAN values are single bit boolean values. A variety of different presentation methods may be selected by the supplier in order to maximise coherence between the description and the presentation of a boolean value. Only the least significant bit of a boolean type has a defined value.
		2	ST_STRING	ST_STRING values are string values. Currently, strings are defined using a zero byte as a terminator character {future versions may well introduce counted length strings as

Offset	Name	Contents		
		Value	**Name**	**Meaning**
				well). A supplier is required to be able to place an upper length value on all strings so that sufficient space may be allocated within the caller supplied buffer for the SA_READ operation.
		3	ST_INTEGER8	ST_INTEGER8 is an 8-bit (byte) quantity.
		4	ST_INTEGER16	ST_INTEGER8 is an 16-bit (half-word) quantity.
		5	ST_INTEGER32	ST_INTEGER8 is an 32-bit (word) quantity.
		6	ST_INTEGER64	ST_INTEGER8 is an 64-bit (double-word) quantity.
		7	ST_ADDRESS	ST_ADDRESS values are addresses given in a protocol-specific format. A variety of different forms of address are possible. The presentation formats available for this type are presently undefined. This type of statistic is only supported by 1.01 of the DCI statistics format.
		8	ST_TIME	ST_TIME values are time values given in a variety of formats. The presentation methods presented allow for times to be specified in a number of ways. This type of statistic is only supported by

Offset	Name	Contents

Value	Name	Meaning
		1.01 of the DCI statistics format.

Offset	Name	Contents
+4	format	This field contains a value indicating the format of the value. The range of values is specific to each type (all the integer types share a common set of formats and presentations for convenience). Depending upon the type in question, this value may be an enumeration, a bitset of flags or a combination of both (none of the currently defined formats mix enumeration and flag bitsets). See below for the meanings of the formats,
+8	presentation	This field dictates how the supplier would like the value to be presented. A gatherer may choose to ignore this value, but only at the risk of presenting the user with incoherent information. See below for the meaning of the presentations.
+12	size	This is the number of bytes the statistic wishes reserved in the buffer for an SA_READ operation. It is always a multiple of four bytes. It is sometimes larger than the actual value necessary.
+16	volatility	This field indicates how volatile a statistic is. The three possible values are:

Value	Name	Meaning
0	SV_STATIC	Constant per invocation
1	SV_VARIABLE	Unlikely to have changed in 5 minutes
2	SV_VOLATILE	Can change very rapidly

Offset	Name	Contents
+20	name	This fields points at the name of the statistic. It is a short descriptive string that the gatherer contracts to display at least the first twenty characters of. The string itself is static, zero byte terminated and contained with the supplier module.
+24	name_tag	This value, together with the information contained in the dci4_spctl structure and a ISV/IHV supplied data file permit the gatherer

Offset	Name	Contents
		to obtain a description of a statistic. The precise mechanism has yet to be determined. Must be zero.
+28	spare	This field is reserved for future use and must be zero.

The format and presentation vary with the statistic types.

Statistic type	Name	Format and presentation
0	ST_UNUSED	There is no format or presentation defined for the ST_UNUSED type.
1	ST_BOOLEAN	Format:

Value	Name	Meaning
0	SBF_NORMAL	The value supplied is 0 if boolean 'false' and any other value for boolean 'true'.
1	SBF_INVERTED	The boolean state is inverted, 0 if boolean 'tru and any other value for boolean 'false'.

Presentation:

Value	Name	Meaning
0	SBP_ON_OFF	The strings 'on' and 'off' will be used.
1	SBP_YES_NO	The strings 'yes' an 'no' will be used.
2	SBP_TRUE_FALSE	The strings 'true' a 'false' will be used.
3	SBP_ALWAYS_NEVER	The strings 'always and 'never' will be used.
4	SBP_ONE_ZERO	The strings '1' and will be used.

2 ST_STRING Format:

Value	Name	Meaning
0	SSF_ZEROTERM	This is the only defined format, indicating that t string is zero-terminated

Presentation:

Value	Name	Meaning
0	SSP_LITERAL	The string should be presented as supplied.

3-6 ST_INTEGER# Format:

Statistic type	Name	Format and presentation

Bit(s)	Name	Meaning
0	SIF_UNSIGNED	Set: The value is unsigned Clear: The value is sign
1	SIF_BIGENDIAN	Set: The value is supplied as big-endian Clear: The value is supplied as little-endian
2-31		Reserved, must be zero

Presentation:

Value	Name	Meaning
0	SIP_HEXADECIMAL	The value should be presented as a hexadecimal number
1	SIP_DECIMAL	The value should be presented as a decim number.
2	SIP_DOTTED	The value should be presented separating every octet with a '.' character. This was intended for use for addresses; it has bee superseded by the ST_ADDRESS type.

7 ST_ADDRESS Format:

Value	Name	Meaning
0	SAF_ETHERNET	The value is a 6 byte MA address.
1	SAF_IP	The value is a 4 byte IP address.
2	SAF_ECONET	The value is a 2 byte (net.station) address.

Presentation:

Value	Name	Meaning
0	SAP_NORMAL	There is only this presentation type.

Statistic type	Name	Format and presentation
7	ST_TIME	Format:

Value	Name	Meaning
0	STF_5BYTE	The value is a standard bytes format (8 bytes, top 3 bytes ignored), measured in centiseconds since 00: 1st January 1900.
1	STF_CTIME	The value is a C time() value, measured in seconds since 00:00 1s January 1970.
2	STF_MONOTONIC	The value is a centisecond value, use by OS_ReadMonotonicTi measured in seconds since the system starte

Presentation:

Value	Name	Meaning
0	STP_ABSOLUTE	The values are absolute and should be presented time values from their reference base.
1	STP_RELATIVE	The values are relative to the current time, and the reference base should be ignored. For example, a value of 250 in format STF_MONOTONIC might be presented as '2.50 seconds'.

Ensuring statistic atomicity

The supplier is required to perform each read request atomically. This means a gatherer can rely upon a read operation producing consistent statistics, but it cannot be certain quite when in time these statistics apply to: in some cases, statistics are virtually always out of date by the time the gatherer obtains and uses them.

The simplistic approach to ensuring an atomic read operation is to perform the entire action with interrupts disabled. This, however, has

an undesirable effect on interrupt latency for suppliers with many or complex statistics and is not recommended.

The suggested method is a three buffer scheme. One buffer holds the accumulated statistics, the second holds the increments (deltas) being generated "live", and the third is either entirely full of zero values, or there is an active read operation in progress and it holds a previous set of increments that are currently being added (merged) into the accumulated values.

Two pointers are maintained. One for the "live" buffer and the other for the "merge" buffer. The operation that is necessary to perform atomically is the swapping of these two pointer values.

The overall structure suggested is as follows:

```
start thread of control (ie SWI)

    enter mutex (eg disable interrupts)

        temp := buffer pointer 1
        buffer pointer 1 := buffer pointer 2
        buffer pointer 2 := temp

    exit mutex (and ensure interrupts enabled)

    add values from buffer pointer 2 into the accumulated values

    zero the memory described by buffer pointer 2

end thread of control (ie SWI)
```

This maximises the usefulness of the non re-entrancy requirement and requires ONLY the swapping of the buffer pointers to be performed atomically. All other operations can proceed with interrupts enabled. This minimal critical region is the prime motivation behind this scheme.

Handling dynamic statistics

Some applications of the statistic interface will require a single RISC OS module to behave as multiple suppliers. For example, a device driver module that controls multiple interfaces might well wish to provide an overall set of statistics, and then an instance of a set of statistics per interface controlled.

A separate SWI number is required for each 'effective supplier'. This is not normally a problem, as very few modules approach the limit of the 64 separate SWIs allocated. Gatherers are expected to obtain SWI numbers through the enumeration method supplied, and should

be capable of handling dynamic SWI number assignment. For example, a device driver may choose SWIs 63, 62 and 61 for units 0, 1 and 2, respectively. If a gatherer were to attempt to 'cache' these SWI numbers across invocations of the device driver, then it might not correctly cater for the addition or removal of interfaces.

Services

Service_StatisticEnumerate (Service Call &A1)

Enumerate providers of extended DCI statistics

On entry

R0 = Pointer to the head of the enumeration chain
R1 = Reason code (&A1)

On exit

R0 = New pointer to the head of the enumeration chain containing all the statistic providers

Use

This service call is used to enumerate the statistic providers which are currently active. Each supplier should allocate a block in the RMA containing the Statistics Provider Structure (on page 125) and fill in the details of its provider. It should then update the `next` pointer to point to the supplied head of the chain, and return the new pointer in R1. It must not claim the service.

Statistics gatherers will use this information to decide what to display, and call the supplied SWI number as appropriate. Initially the statistics provider will supply the pointer in R0 as 0 when the service is issued.

Related SWIs

SWI StatisticsProvider_Statistics (on page 140)

SWI calls

StatisticsProvider_Statistics (SWI StatisticsProvider+&0)

Obtain statistics from a statistics provider

On entry

R0 = SA_DESCRIBE (0) or SA_READ (1) depending on the operation required
R1 = First statistic, inclusive
R2 = Last statistic, inclusive
R3 = First byte of buffer in memory
R4 = Number of bytes in buffer

On exit

R0 - R4 preserved
R5 = Number of statistics processed
R6 = Number of bytes of buffer used

Interrupts

Interrupts are undefined
Fast interrupts are undefined

Processor mode

Processor is in undefined mode

Re-entrancy

Not defined

Use

This SWI is used to obtain a description of the statistics provided by a supplier, or the actual values of those statitics. It is expected that it be called at least twice, once to read the descriptions of the statistics, and then future times to read the values of those statistics.

When the operation requested is SA_DESCRIBE (0), the statistics buffer supplied in R1 should be filled with structures as described in Statistic Description Structure (on page 127).

When the operation requested is SA_READ (1), the statistics buffer

supplied in R1 should be filled with the values corresponding to those that would be described if SA_DESCRIBE (0) had been requested. Each statistics has a length as defined by the Statistic Description Structure (on page 127).

If the buffer is filled before all statistics are written, the SWI should exit with as much data as would fit into the buffer written and appropriate values in R5 and R6 on exit. Statistics gatherers may recognise this condition and provide extended buffers.

Related services

Service_StatisticEnumerate (on page 139)

Document information

Maintainer(s): Charles Ferguson <gerph@gerph.org>

History: Revision	Date	Author	Changes
1	23 Sep 1994	Borris	● Tried to get the basic ideas into a coherent document.
2	25 Sep 1994	Borris	● Changed precise details of register updating on return from some of the swi operations.
3	Oct 1994	Borris	● Listened to feedback from reviewers. ● Adjust underlying model and streamlined most aspects. ● Added details of suggested scheme for atomic sampling. ● This issue not widely distributed.
4	7 Nov 1994	Borris	● Tided up and checked consistency with reference header file. ● Added correct service call number. ● Added comments about effective suppliers and dynamic SWI number possibilities. ● name_tag as being outstanding added.
5	28 Apr 2003	Gerph	● Added ST_TIME and ST_ADDRESS types.
6	08 Aug 2020	Gerph	**Restructure as PRM-in-XML** ● Exported from Impression, and converted to PRM-in-XML structured format. ● Restructured document to match PRM structure and style.

- Added note that the features field is not currently defined and should be 0.

Index (SWIs)

Index (SWIs by number)

Index (Services)

Index (Services by number)