SANA-II Network Device Driver Specification

Amiga Networking Group
Randell Jesup, Kenneth Dyke
Revised by Heinz Wrobel

Revision 3.1
15 December 2002

Warning

The information contained herein is subject to change without notice. Amiga specifically does not make any endorsement or representation with respect to the use, results, or performance of the information (including without limitation its capabilities, appropriateness, reliability, currentness or availability).

Disclaimer

This information is provided "As Is" without warranty of any kind, either express or implied. The entire risk as to the use of this information is assumed by the user. In no event will Amiga or its affiliated companies be liable for any damages, direct, indirect, incidental, special or consequential, resulting from any claim arising out of the information presented herein, even if it has been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of such implied warranties, so the above limitations may not apply.

Contents

1 SANA-II Network Device Driver Specification **5**

2 Driver Form **6**

3 Opening a SANA-II Device **6**

4 Buffer Management **8**
4.1 Packet filtering 10
4.2 Enhancements for better buffer management 10

5 Multicasting extensions **11**

6 Packet Type **12**
6.1 Ethernet Packet Types 12
6.2 ARCNET Frames 12

7 Addressing **13**

8 Hardware Type **14**

9 Errors **14**

10 Standard Commands **16**
10.1 Broadcast and Multicast 16
10.2 Stats 17
10.3 Configuration 20
10.4 On-line 21

11 Driver Installation **23**

12 Acknowledgments **23**

13 Unresolved Issues **23**

A SANA-II network device driver Autodocs **25**
A.1 sana2.device/AbortIO 25
A.2 sana2.device/CloseDevice 26
A.3 sana2.device/CMD_CLEAR 27
A.4 sana2.device/CMD_FLUSH 28
A.5 sana2.device/CMD_INVALID 29
A.6 sana2.device/CMD_READ 30
A.7 sana2.device/CMD_RESET 31
A.8 sana2.device/CMD_START 32
A.9 sana2.device/CMD_STOP 33

A.10 sana2.device/CMD_UPDATE 34
A.11 sana2.device/CMD_WRITE 35
A.12 sana2.device/OpenDevice 36
A.13 sana2.device/S2_ADDMULTICASTADDRESS 37
A.14 sana2.device/S2_ADDMULTICASTADDRESSES 38
A.15 sana2.device/S2_BROADCAST 39
A.16 sana2.device/S2_CONFIGINTERFACE 40
A.17 sana2.device/S2_DELMULTICASTADDRESS 41
A.18 sana2.device/S2_DELMULTICASTADDRESSES 42
A.19 sana2.device/S2_DEVICEQUERY 43
A.20 sana2.device/S2_GETGLOBALSTATS 44
A.21 sana2.device/S2_GETSPECIALSTATS 45
A.22 sana2.device/S2_GETSTATIONADDRESS 46
A.23 sana2.device/S2_GETTYPESTATS 47
A.24 sana2.device/S2_MULTICAST 48
A.25 sana2.device/S2_OFFLINE 49
A.26 sana2.device/S2_ONEVENT 50
A.27 sana2.device/S2_ONLINE 51
A.28 sana2.device/S2_READORPHAN 52
A.29 sana2.device/S2_TRACKTYPE 53
A.30 sana2.device/S2_UNTRACKTYPE 54

B Callback mechanism Autodocs **55**
B.1 CopyFromBuff 55
B.2 CopyToBuff 56
B.3 PacketFilter 57

C Ethernet description **58**

D ARCNET description **59**

E "sana.h" header file **60**

F "sana2specialstats.h" header file **65**

1 SANA-II Network Device Driver Specification

The "SANA-II Network Device Driver Specification" is a standard for an Amiga software interface between networking hardware and network protocol stacks (or for software tools such as network monitors). A network protocol stack is a layer of software that network applications use to address particular processes on remote machines and to send data reliably in spite of hardware errors. There are several common network protocol stacks including *TCP/IP*, *OSI*, *AppleTalk*, *DECNet* and *Novell*.

SANA-II device drivers are intended to allow multiple network protocol stacks running on the same machine to share one network device. For example, the TCP/IP and AppleTalk protocol stacks could both run on the same machine over one ethernet board. The device drivers are also intended to allow network protocol stacks to be written in a hardware-independent fashion so that a different version of each protocol stack doesn't have to be written for each networking hardware device.

The standard does not address the writing of network applications. Application writers must not use SANA-II Device Drivers directly. Network applications must use the API provided by the network protocol software the application supports. There is not an Amiga standard network API at the time of this writing, though there is the *AS225* TCP/IP package and its `socket.library` as well as other (third-party) packages.

To write a SANA-II device driver, you will need to be familiar with the specification documents for the hardware you are writing to and with the "SANA-II Network Device Driver Specification".

To write a network protocol stack which will use SANA-II device drivers, you should have general familiarity with common network hardware and must be very familiar with the "SANA-II Network Device Driver Specification" as well as the specification for the protocol you are developing. If you are creating a new protocol, you must obtain a protocol type number for any hardware on which your protocol will be used.

Amiga supports the SANA-II specification by providing drivers for the Amiga network hardware. We have an `a2065.device` (Ethernet) and intend to produce an `a2060.device` (ARCNET). We also try to examine review copies of third-party SANA-II networking hardware and software to try to make sure that they interoperate with our products.

This standard has undergone several drafts with long periods for comment from developers and the Amiga community at large. These drafts include a UseNet release which was also distributed on the Fish Disks in June, 1991 (as well as published in the '91 DevCon notes), and the November 7 Draft for Final Comment and Approval distributed via Bix, ADSP and UseNet. There were also several intermediate drafts with more limited distribution.

This version of the specification is final. Any new version of the standard

(i.e., to add new features) is planned to be backward compatible. No SANA-II device driver or software utilizing those drivers should be written to any earlier version of the specification.

Distribution of this version of the standard is unlimited. Anyone may write Amiga software which implements a SANA-II network device driver or which calls a SANA-II network device driver without restriction and may freely distribute such software that they have written.

It is important to try to test each SANA-II device driver against all software which uses SANA-II devices. Available example programs are valuable in initial testing. The Amiga Networking Group is interested in receiving evaluation and/or beta test copies of all Amiga networking hardware, SANA-II device drivers and software which uses SANA-II devices. However, we make no assurances regarding any testing which we may or may not perform with such evaluation copies.

2 Driver Form

SANA-II device drivers are Amiga Exec device drivers. They use an extended `IORequest` structure and a number of extended commands for tallying network statistics, sending broadcasts and multicasts, network addressing and the handling of unexpected packets. The *Amiga ROM Kernel Reference Manual: Devices* includes information on how to construct an Exec device.

3 Opening a SANA-II Device

As when opening any other Exec device, on the call to `OpenDevice()` a SANA-II device receives an `IORequest` structure which the device initializes for the opener's use. The opener must copy this structure if it desires to use multiple asynchronous requests. The SANA-II `IORequest` is defined as follows:

```
struct IOSana2Req
{
   struct IORequest ios2_Req;
   ULONG            ios2_WireError;
   ULONG            ios2_PacketType;
   UBYTE            ios2_SrcAddr[SANA2_MAX_ADDR_BYTES];
   UBYTE            ios2_DstAddr[SANA2_MAX_ADDR_BYTES];
   ULONG            ios2_DataLength;
   APTR             ios2_Data;
   APTR             ios2_StatData;
   APTR             ios2_BufferManagement;
```

};

ios2_Req A standard Exec device `IORequest`.

ios2_WireError A more specific device code which may be set when there is an `io_Error`. See `<devices/sana2.h>` on page 60 for the defined `WireError`s.

ios2_PacketType The type of packet requested. See the section on "Packet Types" on page 12.

ios2_SrcAddr The device fills in this field with the interface (network hardware) address of the source of the packet that satisfied a read command. The bytes used to hold the address will be left justified but the bit layout is dependent on the particular type of network.

ios2_DstAddr Before the device user sends a packet, it fills this with the interface destination address of the packet. On receives, the device fills this with the interface destination address. Other commands may use this field differently (see the "SANA-II network device driver Autodocs" on page 25). The bytes used to hold the address will be left justified but the bit layout is dependent on the particular type of network.

ios2_DataLength The device user initializes this field with the amount of data available in the Data buffer before passing the `IOSana2Req` to the device. The device fills in this field with the size of the packet data as it was sent on the wire. This does not include the header and trailer information. Depending on the network type and protocol type, the driver may have to calculate this value. This is generally used only for reads and writes (including broadcast and multicast).

ios2_Data A pointer to some abstract data structure containing packet data. *Drivers may not directly manipulate or examine anything pointed to by Data!* This is generally used only for reads and writes (including broadcast and multicast).

ios2_StatData Pointer to a structure in which to place a snapshot of device statistics. The data area must be long word aligned. This is only used on calls to the statistics commands.

ios2_BufferManagement The opener places a pointer to a tag list in this field before calling `OpenDevice()`. Functions pointed to in the tag list are called by the device when processing `IORequest`s from the opener. When returned from `OpenDevice()`, this field contains a pointer to driver-private information used to access these functions. See "Buffer Management" below for more details.

Note to implementors: *A SANA-II device must reject all open requests with a request structure that is too short, e.g. an* `IOStdReq`. *A simple check of the* `mn_Length` *field in the* `Message` *part of the request is needed to make sure that a device does not dereference invalid data due to a wrong device configuration.*

A SANA-II device may open if no buffer management tags are provided to make the configuration process and obtaining statistics easier. Buffer management tags with a NULL value must be treated as not specified. The device shall fail requests gracefully depending on the missing tags in this case. Any malfunction is not acceptable.

The flags used with the device on `OpenDevice()` are (`SANA2OPB_xxx`):

SANA2OPB_MINE Exclusive access to the unit requested.

SANA2OPB_PROM Promiscuous mode requested. Hardware which supports promiscuous mode allows all packets sent over the wire to be captured whether or not they are addressed to this node.

Note: *Promiscuous mode requires exclusive opening of the device.*

The flags used during I/O requests are (`SANA2IOB_xxx`):

SANA2IOB_RAW Raw packet read/write requested. Raw packets should include the entire data-link layer packet. Devices with the same hardware device number should have the same raw packet format.

SANA2IOB_BCAST Broadcast packet (received).

SANA2IOB_MCAST Multicast packet (received).

SANA2IOB_QUICK Quick IO requested.

4 Buffer Management

Unlike most other Exec Device drivers, SANA-II drivers have no internal buffers. Instead, they read/write to/from an abstract data structure allocated by the driver user. The driver accesses these buffers only via functions that the driver user provides to the driver. The driver user must provide two functions–one copies data to the abstract data structure and one copies data from the abstract data structure. The driver user can therefore choose the data structure used for buffer management by both the driver and driver user in order to have efficient memory and CPU usage overall.

The `IOSana2Req` contains a pointer to data and the length of said data. A driver is not allowed to make assumptions about how the data is stored. The driver cannot directly manipulate or examine the buffer in any manner.

The driver can only access the buffer by calling the functions provided by the driver user.

Before calling `OpenDevice()`, the driver user initializes `ios2_BufferManagement` to point to a list of tags (defined in `<devices/sana2.h>`) which include pointers to the buffer management functions required by the driver (defined below). The driver will fail to open if the driver user does not supply all of the required functions. If the device opens successfully, the driver sets `ios2_BufferManagement` to a value which this opener must use in all future calls to the driver. This "magic cookie" is used from then on to access these functions (a "magic cookie" is a value which one software entity passes to another but which is only meaningful to one of the software entities). The driver user may not use the "magic cookie" in any way–it is for the driver to do with as it wishes. The driver could in theory choose to just copy the tag list to driver-owned memory and then parse the list for every `IORequest`, but it is much more efficient for the driver to create some sort of table of functions and to point `ios2_BufferManagement` to that table.

Another recommendation for the "magic cookie" is to use it to maintain a seperate packet read queue for each device opener. This would allow multiple protocol stacks that all wish to receive the same packet type to work together without having to "know" about each other as *Envoy* and *AS225* do right now. What does multiple protocol stack support mean? Basically this means that each opener gets all the packets necessary. If a packet comes in that fills a request for more than one opener of the device, all of them will get a copy of the packet. This feature should never be left out of a device design. If it is missing, the usefulness of the device is severely limited.

The *SLIP* and *A2065* driver now do this, so it would be possible (for example) to run *Envoy*, *AS225* and the *AmiTCP* package together on the same hardware without conflicts.

In order to help system load, a new callback has been added to allow protocol stacks to reject packets that are known to not be useful. *Envoy*'s `nipc.library` (for example) could be modified to reject TCP packets (as it never uses them).

The specification currently defines three tags for the `OpenDevice()` `ios2_BufferManagement` tag list:

S2_CopyToBuff This is a pointer to a function which conforms to the `CopyToBuff` Autodoc (see page 56).

S2_CopyFromBuff This is a pointer to a function which conforms to the `CopyFromBuff` Autodoc (see page 55).

S2_PacketFilter *[optional]* This is a pointer to a standard `Hook` to be called before `S2_CopyToBuff` is done. See the `PacketFilter` Autodoc

on page 57 for more information.

4.1 Packet filtering

What does packet filtering do? With the original "SANA-II Network Device Driver Specification", a protocol stack could open a device and ask for certain packet types. It got all the packets that matched this type. As it turned out, this could be mighty inefficient if there were packets that the protocol stack did not use at all. These would go into read processing of the protocol stack and waste CPU time even though they could have been easily identified on arrival.

4.2 Enhancements for better buffer management

It has been observed that the standard buffer management callbacks may not be very efficient for certain types of hardware. They also do not allow driver DMA access. Therefore the original "SANA-II Network Device Driver Specification" has been enhanced to allow for more flexible buffer management. This enhancement is fully backwards compatible.

All the new features are completely optional and do not collide with existing features. They may be used only when the protocol stack asks for them on opening a driver.

The enhancements consist of several new tags that may be specified by a protocol stack on `OpenDevice()` to offer certain data transfer options. It is up to the device driver to choose which callbacks to use at what time. These tags are advisory only and may be ignored by the driver for any data buffer at any time:

```
#define S2_CopyToBuff16   (S2_Dummy + 4)
#define S2_CopyFromBuff16 (S2_Dummy + 5)
#define S2_CopyToBuff32   (S2_Dummy + 6)
#define S2_CopyFromBuff32 (S2_Dummy + 7)
```

These are optional callbacks presented to the device with the same calling interface as for `S2_CopyToBuff` or `S2_CopyFromBuff`, respectively. The difference to the original callbacks is the required and guaranteed transfer size and alignment for accessing the device's buffer for a single piece of a data of either 16 or 32 bits, a data word. The copy function called may only use 16/32 bit aligned read/write commands of 16/32 bits at once to transfer the data words, respectively. If the buffer data length is not a multiple of the required data word transfer size, the last data word transfer may contain garbage padding in either transfer direction.

The following tags have been added to support direct writes into hardware buffers that do not allow arbitrarily sized or aligned accesses:

```
#define S2_DMACopyToBuff32   (S2_Dummy + 8)
#define S2_DMACopyFromBuff32 (S2_Dummy + 9)
```

If the protocol stack wants to optionally enhance data transfer efficiency with DMA supporting devices, it may pass any of these optional tags to the device on `OpenDevice()`.

If the device driver supports DMA, it may call the respective callback with the abstract magic cookie `ios2_Data` in register A0. The callback may return NULL in D0. In this case, the driver may not use DMA for this buffer. Alternatively, the callback may return the address of the actual data buffer in D0, if it has these characteristics:

- The buffer is in contigous memory. Depending on the intended data direction, it shall be readable or writable.
- The buffer is aligned on a 32 bit boundary.
- The buffer size shall be a multiple of 32 bit and it is at least $\geq$ `ios2_DataLength`.

It is up to the driver to decide if it can use DMA for this buffer and it shall fall back to the standard CPU callbacks if necessary. The data transfer method actually used by the driver will not be known in advance by the protocol stack.

5 Multicasting extensions

SANA-II Rev 3 unfortunately only has a command that allows an application to specify a **single** multicast address. An application may issue multiple such commands, effectively accepting packets for more than one multicast address.

However because of memory and CPU time constraints this method cannot be used reasonably if an application needs to listen to a large **range** of multicast addresses (e.g. several thousand addresses or more), as is for instance the case with `mrouted`-like programs which need to forward **all** IP multicasting traffic between networks. This involves a multicast address range of 2^{23} addresses.

This section defines two additional commands which allow an application to listen to a whole, possibly very large, **range** of multicast addresses, overcoming the above-mentioned limitation in SANA-II Rev 3.

In addition to the existing commands `S2_ADDMULTICASTADDRESS` and `S2_DELMULTICASTADDRESS` two new commands `S2_ADDMULTICASTADDRESSES` and `S2_DELMULTICASTADDRESSES` are introduced. These commands enable and disable the reception of packets within a given address range.

6 Packet Type

Network frames always have a type field associated with them. These type fields vary in length, position and meaning by frame type (frame types generally correspond one-to-one with hardware types, but see "Ethernet Packet Types" below). The meanings of the type numbers are always carefully defined and every type number is registered with some official body. Do not use a type number which is not registered for any standard hardware you use or in a manner inconsistent with that registration.

The type field allows the SANA-II device driver to fulfill `CMD_READ`s based on the type of packet the driver user wants. Multiple protocols can therefore run over the same wire using the same driver without stepping on each other's toes.

Packet types are specified as a long word. Unfortunately, the type field means different things on different wires. Driver users must allow their software to be configured with a SANA-II device name, unit number and the type number(s) used by the protocol stack with each device. This way, if new hardware becomes available, a hardware manufacturer can supply a listing of type assignments to configure pre-existing software.

6.1 Ethernet Packet Types

Ethernet has a special problem with packet types. Two types of ethernet frames can be sent over the same wire–ethernet and 802.3. These frames differ in that the `Type` field of an ethernet frame is the `Length` field of an 802.3 frame. This creates a problem in that demultiplexing incoming packets can be cumbersome and inefficient, as well as requiring driver users to be aware of the frame type used.

All 802.3 frames have numbers less than 1500 in the `Type` field. The only frames with numbers less than 1500 in the type field are 802.3 frames. SANA-II ethernet drivers abnormally return packets contained in ethernet frames when the requested `Type` falls within the 802.3 range–if the `Type` requested is within the 802.3 range, the driver returns the next packet contained within an 802.3 frame, regardless of the type specified for the packet within the 802.3 frame. This requires that there be no more than one driver user requesting 802.3 packets and that it do its own interpretation of the frames.

6.2 ARCNET Frames

ARCNET also has a special problem with framing. ARCNET frames consist of a hardware header and a software header. The software header is in the data area of the hardware packet, and includes at least the protocol ID.

There are two types of software header. Old-style ARCNET software

headers consist entirely of a one or two byte protocol ID. New ARCNET software headers (defined in `RFC 1201` and in the paper "ARCNET Packet Header Definition Standard", Novell, Inc., 1989) include more information. They allow more efficient use of ARCNET through data link layer fragmentation and reassembly (ARCNET has a small Maximum Transmission Unit) and allow sending any size packet up to the MTU (rather than requiring that packets of size 253, 254 and 255 be padded to at least 256 bytes).

SANA-II device drivers for ARCNET should implement the old ARCNET packet headers. Driver users which wish to interoperate with platforms using the new software headers must add the new fields to the data to be sent and must process it for incoming data. A SANA-II driver which implemented the data link layer fragmentation internally (and advertised a large MTU) could be more efficient than requiring the driver user to do it. This would make driver writing more difficult and reduce interoperability, but if there is ever a demand for that extra performance, a new hardware type may be assigned by Amiga for SANA-II ARCNET device drivers which implement the new framing.

7 Addressing

In the SANA-II standard, network hardware addresses are stored in an array of n bytes. No meaning is ascribed by the standard to the contents of the array.

In case there exists a network which does not have an address field consisting of a number of bits not divisible by eight, add pad bits at the end of the bit stream. For example, if an address is ten bits long it will be stored like this:

```
98765432 10PPPPPP
BYTE 0   BYTE 1
```

Where the numerals are bit numbers and 'P' is a pad (ignored) bit.

Driver users which do not implement the bit shifting necessary to use a network with such addressing (if one exists) should at least check the number of significant bits in the address field (returned from the device's `S2_DEVICEQUERY` function) to make sure that it is evenly divisible by eight.

Driver users will map hardware addresses to protocol addresses in a protocol and hardware dependent manner, as described by the relevant standards (i.e., `RFC 826` for TCP/IP over Ethernet, `RFC 1201` or `RFC 1051` for TCP/IP over ARCNET). Some protocols will always use the same mapping on all hardware, but other protocols will have particular address mapping schemes for some particular hardware and a reasonable default for other (unknown) hardware.

Some SANA-II devices will have "hardware addresses" which aren't really hardware addresses. As an example, consider *PPP* (Point-to-Point Protocol). PPP is a standard for transmitting IP packets over a serial line. It uses IP addresses negotiated during the establishment of a connection. In a SANA-II driver implementation of PPP, the driver would negotiate the address at `S2_CONFIGINTERFACE`. Thus, the address in `SrcAddr` returned by the device on an `S2_CONFIGINTERFACE` (or in a subsequent `S2_GETSTATIONADDRESS`) will be a protocol address, not a true hardware address.

Note: *Some hardware always uses a ROM hardware address. Other hardware which has a ROM address or is configurable with DIP switches may be overridden by software. Some hardware always dynamically allocates a new hardware address at initialization. See "Configuration" on page 20 for details on how this is handled by driver writers and by driver users.*

8 Hardware Type

The `HardwareType` returned by the device's `S2_DEVICEQUERY` function is necessary for those protocols whose standards require different behavior on different hardware. It is also useful for determining appropriate packet type numbers to use with the device. The `HardwareType` values already issued for standard network hardware are the same as those in `RFC 1060` (assigned numbers). Hardware developers implementing networks without a SANA-II hardware number must contact Amiga to have a new hardware type number assigned. Driver users should all have reasonable defaults which can be used for hardware with which they are not familiar.

9 Errors

The SANA-II extended `IORequest` structure (`struct IOSana2Req`) includes both the `ios2_Error` and `ios2_WireError` fields. Driver users must always check `IOSana2Reqs` on return for an error in `ios2_Error`. `ios2_Error` will be zero if no error occurred, otherwise it will contain a value from `<exec/errors.h>` or `<devices/sana2.h>`. If there was an error, there may be more specific information in `ios2_WireError`. Drivers are required to fill in the `WireError` if there is an applicable error code.

Error codes are `#define`'d in the "defined errors" sections of the file `<devices/sana2.h>`:

`IOSana2Req` `S2io_Error` field (`S2ERR_xxx`):

S2ERR_NO_RESOURCES Insufficient resources available.

S2ERR_BAD_ARGUMENT Noticeably bad argument.

S2ERR_BAD_STATE Command inappropriate for current state.

S2ERR_BAD_ADDRESS Noticeably bad address.

S2ERR_MTU_EXCEEDED Write data too large.

S2ERR_NOT_SUPPORTED Command is not supported by this driver. This is similar to `IOERR_NOCMD` as defined in `<exec/errors.h>` but `S2ERR_NOT_SUPPORTED` indicates that the requested command is a valid SANA-II command and that the driver does not support it because the hardware is incapable of supporting it (e.g., `S2_MULTICAST`). Note that `IOERR_NOCMD` is still valid for reasons other than a lack of hardware support (i.e., commands which are no-ops in a SANA-II driver).

S2ERR_SOFTWARE Software error of some kind.

S2ERR_OUTOFSERVICE When a hardware device is taken off-line, any pending requests are returned with this error.

See also the standard errors in `<exec/errors.h>`.

`IOSana2Req` `S2io_WireError` field (`S2WERR_xxx`):

S2WERR_NOT_CONFIGURED Command requires unit to be configured.

S2WERR_UNIT_ONLINE Command requires that the unit be off-line.

S2WERR_UNIT_OFFLINE Command requires that the unit be on-line.

S2WERR_ALREADY_TRACKED Protocol is already being tracked.

S2WERR_NOT_TRACKED Protocol is not being tracked.

S2WERR_BUFF_ERROR Buffer management function returned an error.

S2WERR_SRC_ADDRESS Problem with the source address field.

S2WERR_DST_ADDRESS Problem with destination address field.

S2WERR_BAD_BROADCAST Problem with an attempt to broadcast.

S2WERR_BAD_MULTICAST Problem with an attempt to multicast.

S2WERR_MULTICAST_FULL Multicast address list full.

S2WERR_BAD_EVENT Event specified is unknown.

S2WERR_BAD_STATDATA The `ios2_StatData` pointer or the data it points to failed a sanity check.

S2WERR_IS_CONFIGURED Attempt to reconfigure the unit.

S2WERR_NULL_POINTER A NULL pointer was detected in one of the arguments. S2ERR_BAD_ARGUMENT should always be the S2ERR.

10 Standard Commands

See the "SANA-II network device driver Autodocs" on page 25 for full details on each of the SANA-II device commands. Extended commands are explained in the sections below.

Many of the Exec device standard commands are no-ops in SANA-II devices, but this may not always be the case. For example, CMD_RESET might someday be used for dynamically reconfiguring hardware. This should present no compatibility problems for properly written drivers.

10.1 Broadcast and Multicast

S2_ADDMULTICASTADDRESS S2_MULTICAST
S2_DELMULTICASTADDRESS S2_BROADCAST

Some hardware supports broadcast and/or multicast. A broadcast is a packet sent to all other machines. A multicast is a packet sent to a set of machines. Drivers for hardware which does not allow broadcast or multicast will return ios2_Error S2ERR_NOT_SUPPORTED as appropriate.

To send a broadcast, use S2_BROADCAST instead of CMD_WRITE. Broadcasts are received just like any other packets (using a CMD_READ for the appropriate packet type).

To send a multicast, use S2_MULTICAST instead of CMD_WRITE. The device keeps a list of addresses that want to receive multicasts. You add a receiver's address to this list by using S2_ADDMULTICASTADDRESS. The receiver then posts a CMD_READ for the type of packet to be received. Some SANA-II devices which support multicast may have a limit on the number of addresses that can simultaneously wait for packets. Always check for an S2WERR_MULTICAST_FULL error return when adding a multicast address.

Note that when the device adds a multicast address, it is usually added for all users of the device, not just the driver user which called S2_ADDMULTICASTADDRESS. In other words, received multicast packets will fill a read request of the appropriate type regardless of whether the requesting driver user is the same one which added the multicast address.

In general, driver users should not care how received packets were sent (normally or broadcast/multicast), only that it was received. If a driver user really must know, however, it can check for SANA2IOB_BCAST and/or SANA2IOB_MCAST in the ios2_Flags field.

Drivers should keep a count for the number of opens on a multicast address so that they don't actually remove it until it has been remove with

the S2_DELMULTICASTADDRESS command as many times as it has been added with the S2_ADDMULTICASTADDRESS command.

10.2 Stats

S2_TRACKTYPE	S2_GETTYPESTATS	S2_GETGLOBALSTATS
S2_UNTRACKTYPE	S2_GETSPECIALSTATS	S2_READORPHAN

There are many statistics which may be very important to someone trying to debug, tune or optimize a protocol stack, as well as to the end user who may need to tune parameters or investigate a problem. Some of these statistics can only be kept by the SANA-II driver, thus there are several required and optional statistics and commands for this purpose.

S2_TRACKTYPE tells the device driver to gather statistics for a particular packet type. S2_UNTRACKTYPE tells it to stop (keeping statistics by type causes the driver to use additional resources). S2_GETTYPESTATS returns any statistics accumulated by the driver for a type being tracked (stats are lost when a type is S2_UNTRACKTYPE'd). Drivers are required to implement the functionality of type tracking. The stats are returned in a struct Sana2PacketTypeStats:

```
struct Sana2PacketTypeStats
{
   ULONG PacketsSent;
   ULONG PacketsReceived;
   ULONG BytesSent;
   ULONG BytesReceived;
   ULONG PacketsDropped;
};
```

PacketsSent Number of packets of a particular type sent.

PacketsReceived Number of packets of a particular type that satisfied a read command.

BytesSent Number of bytes of data sent in packets of a particular type.

BytesReceived Number of bytes of data of a particular packet type that satisfied a read command.

PacketsDropped Number of packets of a particular type that were received while there were no pending reads of that packet type.

S2_GETGLOBALSTATS returns global statistics kept by the driver. Drivers are required to keep all applicable statistics. Since all are applicable to most hardware, most drivers will maintain all statistics. The stats are returned in a struct Sana2DeviceStats:

```
struct Sana2DeviceStats
{
   ULONG          PacketsReceived;
   ULONG          PacketsSent;
   ULONG          BadData;
   ULONG          Overruns;
   ULONG          UnknownTypesReceived;
   ULONG          Reconfigurations;
   struct timeval LastStart;
};
```

PacketsReceived Number of packets that this unit has received.

PacketsSent Number of packets that this unit has sent.

BadData Number of bad packets received (i.e., hardware CRC failed).

Overruns Number of packets dropped due to insufficient resources available in the network interface.

UnknownTypeReceived Number of packets received that had no pending read command with the appropriate packet type.

Reconfigurations Number of network reconfigurations since this unit was last configured.

LastStart The time when this unit last went on-line.

`S2_GETSPECIALSTATS` returns any special statistics kept by a particular driver. Each new wire type will have a set of documented, required statistics for that wire type and a standard set of optional statistics for that wire type (optional because they might not be available from all hardware). The data returned by `S2_GETSPECIALSTATS` will require wire-specific interpretation. See `<devices/sana2specialstats.h>` on page 65 for currently defined special statistics. The statistics are returned in the following structures:

```
struct Sana2SpecialStatRecord
{
   ULONG Type;
   ULONG Count;
   char *String;
};
```

Type Statistic identifier.

Count Statistic itself.

String An identifying, null-terminated string for the statistic. Should be plain ASCII with no formatting characters.

```
struct Sana2SpecialStatHeader
{
   ULONG  RecordCountMax;
   ULONG  RecordCountSupplied;
   struct Sana2SpecialStatRecord[RecordCountMax];
};
```

RecordCountMax There is space for this many records into which statistics may be placed.

RecordCountSupplied Number of statistic records supplied.

`S2_READORPHAN` is not, strictly speaking, a statistical function. It is a request to read any packet of a type for which there is no outstanding `CMD_READ`. `S2_READORPHAN` might be used in the same manner as many statistics, though, such as to determine what packet types are causing overruns, etc.

10.3 Configuration

`S2_DEVICEQUERY` `S2_CONFIGINTERFACE` `S2_GETSTATIONADDRESS`

The device driver needs to configure the hardware before using it. The driver user must know some network hardware parameters (hardware address and MTU, for example) when using it. These commands address those needs.

When a driver user is initialized, it should try to `S2_CONFIGINTERFACE` even though an interface can only be configured once and someone else may have done it. Before you call `S2_CONFIGINTERFACE`, first call `S2_GETSTATIONADDRESS` to determine the factory address (if any). Also provide for user-override of the factory address (that address may be optional and the user may need to override it). When `S2_CONFIGINTERFACE` returns, check the `ios2_SrcAddr` for the actual address the hardware has been configured with. This is because some hardware (or serial line standards such as PPP) always dynamically allocates an address at initialization.

Driver users will want to use `S2_DEVICEQUERY` to determine the MTU and other characteristics of the network. The structure returned from `S2_DEVICEQUERY` is defined as:

```
struct Sana2DeviceQuery
{
   ULONG SizeAvailable;
   ULONG SizeSupplied;
   ULONG DevQueryFormat;
   ULONG DeviceLevel;
   ULONG AddrFieldSize;
   ULONG MTU;
   ULONG BPS;
   ULONG HardwareType;
};
```

SizeAvailable Size, in bytes, of the space available in which to place device information. This includes both size fields.

SizeSupplied Size, in bytes, of the data supplied.

DevQueryFormat The format defined here is format 0.

DeviceLevel This spec defines level 0.

AddrFieldSize The number of bits in an interface address.

MTU Maximum Transmission Unit, the size, in bytes, of the maximum packet size, not including header and trailer information.

BPS Best guess at the raw line rate for this network in bits per second.

HardwareType Specifies the type of network hardware the driver controls.

10.4 On-line

`S2_ONLINE  S2_ONEVENT  S2_OFFLINE`

In order to run hardware tests on an otherwise live system, the `S2_OFFLINE` command allows the SANA-II device driver to be "turned off" until the tests are complete and an `S2_ONLINE` is sent to the driver. `S2_ONLINE` causes the interface to re-configure and re-initialize. Any packets destined for the hardware while the device is off-line will be lost. All pending and new requests to the driver shall be returned with `S2ERR_OUTOFSERVICE` when a device is off-line.

All driver users must understand that any I/O request may return with `S2ERR_OUTOFSERVICE` because the driver is off-line (any other program may call `S2_OFFLINE` to make it so). In such an event, the driver will usually want to wait until the unit comes back on-line (for the program which called `S2_OFFLINE` to call `S2_ONLINE`). It may do this by calling `S2_ONEVENT` to wait for `S2EVENT_ONLINE`. `S2_ONEVENT` allows the driver user to wait on various events.

A driver must track events, but may not distinguish between some types of events. Drivers return `S2_ONEVENT` with `S2ERR_NOT_SUPPORTED` and `S2WERR_BAD_EVENT` for unsupported events. One error may cause more than one event (see below). Errors which seem to have been caused by a malformed or unusual request should not generally trigger an event.

Event types (`S2EVENT_xxx`):

S2EVENT_ERROR Return when any error occurs.

S2EVENT_TX Return on any transmit error (always an error).

S2EVENT_RX Return on any receive error (always an error).

S2EVENT_ONLINE Return when unit goes on-line or return immediately if unit is already on-line (not an error).

S2EVENT_OFFLINE Return when unit goes off-line or return immediately if unit is already off-line (not an error.)

S2EVENT_BUFF Return on any buffer management function error (always an error).

S2EVENT_HARDWARE Return when any hardware error occurs (always an error, may be a `S2EVENT_TX` or `S2EVENT_RX`, too).

S2EVENT_SOFTWARE Return when any software error occurs (always an error, may be a `S2EVENT_TX` or `S2EVENT_RX`, too).

11 Driver Installation

The standard system location for SANA-II network device driver is in a directory called `"Networks"` which exists in the `"DEVS:"` directory.
Example:

```
DEVS:Networks/a2065.device
```

This is the official location for the drivers. It may be necessary for your install program/script to create this directory if it doesn't exist in a user's system.

12 Acknowledgments

Many people and companies have contributed to the "SANA-II Network Device Driver Specification". The original SANA-II Autodocs and includes were put together by Ray Brand, Perry Kivolowitz (ASDG) and Martin Hunt. Those original documents evolved to their current state and grew to include this document at the hands of Dale Larson and Greg Miller. Brian Jackson and John Orr provided valuable editing. Randell Jesup has provided sage advice on several occasions. The buffer management callback mechanism was his idea. Dale Luck (GfxBase) and Rick Spanbauer (Ameristar Technologies) have provided valuable comments throughout the process. Nicolas Benezan (ADONIS) provided many detailed and useful comments on weaknesses in late drafts of the specification. The enhancements for better buffer management, clarifications and notes for device implementors were added to the specification by Heinz Wrobel whilst consulting for Amiga Technologies GmbH, yielding revision 3.0 of the specification.

Thanks to all the above and the numerous others who have contributed with their comments, questions and discussions.

13 Unresolved Issues

Unfortunately, it isn't possible to completely isolate network protocols from the hardware they run on. Hardware types and addressing both remain somewhat hardware-dependent in spite of our efforts. See the "Packet Type" section on page 12 for an explanation of how packet types are handled and why protocols cannot be isolated from them. See the "Addressing" section on page 13 for an explanation of how addressing is handled any why protocols cannot be isolated from it.

Additionally, there are at least two cases where a hardware type has multiple framing methods in use (ethernet/802.3 and arcnet/(Novell) "ARCNET Packet Header Definition Standard"). In both cases, software which

must interoperate with other platforms on this hardware may need to be aware of the distinctions and may have to do extra processing in order to use the appropriate frame type. See the sections on "Ethernet Packet Types" on page 12 and on "ARCNET frames" on page 12 for more details.

Another feature that SANA-II currently lacks is any concept of dynamic addressing. Some hardware types such as LocalTalk or dialup SLIP/PPP connections may change their address on the fly. Currently there is no way for a device driver to report this event back to a protocol stack.

A SANA-II network device driver Autodocs

A.1 sana2.device/AbortIO

```
sana2.device/AbortIO                                    sana2.device/AbortIO

   NAME
        AbortIO -- Remove an existing device request.

   SYNOPSIS
        error = AbortIO(Sana2Req)
        D0              A1

        LONG AbortIO(struct IOSana2Req *);

   FUNCTION
        This is an exec.library call.

        This function aborts an ioRequest. If the request is active, it may or
        may not be aborted. If the request is queued it is removed. The
        request will be returned in the same way as if it had normally
        completed.  You must WaitIO() after AbortIO() for the request to
        return.

   INPUTS
        Sana2Req        - Sana2Req to be aborted.

   RESULTS
        error           - Zero if the request was aborted, non-zero otherwise.
                          io_Error in Sana2Req will be set to IOERR_ABORTED
                          if it was aborted.

   SEE ALSO
        exec.library/AbortIO(), exec.library/WaitIO()
```

A.2 sana2.device/CloseDevice

```
sana2.device/CloseDevice                                sana2.device/CloseDevice

   NAME
        CloseDevice -- Close the device.

   SYNOPSIS
        CloseDevice(Sana2Req)
                    A1

        void CloseDevice(struct IOSana2Req *);

   FUNCTION
        This function is called by exec.library CloseDevice().

        This function performs whatever cleanup is required at device closes.

        Note that all IORequests MUST be complete before closing. If any are
        pending, your program must AbortIO() then WaitIO() each outstanding
        IORequest to complete them.

   INPUTS
        Sana2Req        - Pointer to IOSana2Req initialized by OpenDevice().

   SEE ALSO
        exec.library/CloseDevice(), exec.library/OpenDevice()
```

A.3 sana2.device/CMD_CLEAR

```
sana2.device/CMD_CLEAR                                  sana2.device/CMD_CLEAR

   NAME
        Clear -- Clear internal network interface read buffers.

   FUNCTION
        There are no device internal buffers, so CMD_CLEAR does not apply to
        this class of device.

   IO REQUEST
        ios2_Command    - CMD_CLEAR.

   RESULTS
        ios2_Error      - IOERR_NOCMD.
```

A.4 sana2.device/CMD_FLUSH

```
sana2.device/CMD_FLUSH                                  sana2.device/CMD_FLUSH

   NAME
        Flush -- Clear all queued I/O requests for the SANA-II device.

   FUNCTION
        This command aborts all I/O requests in both the read and write
        request queues of the device.  All pending I/O requests are
        returned with an error message (IOERR_ABORTED).  CMD_FLUSH does not
        affect active requests.

   IO REQUEST
        ios2_Command    - CMD_FLUSH.

   RESULTS
        ios2_Error      - Zero if successful; non-zero otherwise.
```

A.5 sana2.device/CMD_INVALID

```
sana2.device/CMD_INVALID                              sana2.device/CMD_INVALID

   NAME
        Invalid -- Return with error IOERR_NOCMD.

   FUNCTION
        This command causes device driver to reply with an error IOERR_NOCMD
        as defined in <exec/errors.h> indicating the command is not supported.

   IO REQUEST
        ios2_Command    - CMD_INVALID.

   RESULTS
        ios2_Error      - IOERR_NOCMD.

   BUGS
        Not known to be useful.
```

A.6 sana2.device/CMD_READ

```
sana2.device/CMD_READ                                   sana2.device/CMD_READ

   NAME
        Read -- Get a packet from the network.

   FUNCTION
        Get the next packet available of the requested packet type. The data
        copied (via a call to the requestor-provided CopyToBuffer function)
        into ios2_Data is normally the Data Link Layer packet data only. If
        bit SANA2IOB_RAW is set in ios2_Flags, then the entire physical frame
        will be returned.

        Unlike most Exec devices, SANA-II device drivers do not have internal
        buffers.  If you wish to read data from a SANA-II device you should
        have multiple CMD_READ requests pending at any given time.  The
        functions provided by you the requestor will be used for any incoming
        packets of the type you've requested.  If no read requests are
        outstanding for a type which comes in and no read_orphan requests are
        outstanding, the packet will be lost.

   IO REQUEST
        ios2_Command    - CMD_READ
        ios2_Flags      - Supported flags are:
                            SANA2IOB_RAW
                            SANA2IOB_QUICK
        ios2_PacketType - Packet type desired.
        ios2_Data       - Abstract data structure to hold packet data.

   RESULTS
        ios2_Error      - Zero if successful; non-zero otherwise.
        ios2_WireError  - More specific error number.
        ios2_Flags      - The following flags may be returned:
                            SANA2IOB_RAW
                            SANA2IOB_BCAST
                            SANA2IOB_MCAST
        ios2_SrcAddr    - Source interface address of packet.
        ios2_DstAddr    - Destination interface address of packet.
        ios2_DataLength - Length of packet data.
        ios2_Data       - Abstract data structure which packet data is
                          contained in.

   NOTES
        The driver may not directly examine or modify anything pointed to by
        ios2_Data.  It *must* use the requester-provided functions to access
        this data.

   SEE ALSO
        S2_READORPHAN, CMD_WRITE, any_protocol/CopyToBuffer
```

A.7 sana2.device/CMD_RESET

```
sana2.device/CMD_RESET                                  sana2.device/CMD_RESET

   NAME
        Reset -- Reset the network interface to initialized state.

   FUNCTION
        Currently, SANA-II devices can only be configured once (with
        CMD_CONFIGINTERFACE) and cannot be re-configured, hence,
        CMD_RESET does not apply to this class of device.

   IO REQUEST
        ios2_Command    - CMD_RESET.

   RESULTS
        ios2_Error      - IOERR_NOCMD.
```

A.8 sana2.device/CMD_START

```
sana2.device/CMD_START                                  sana2.device/CMD_START

   NAME
        Start -- Restart device operation.

   FUNCTION
        There is no way for the driver to keep queuing requests without
        servicing them, so CMD_STOP does not apply to this class of device.
        S2_OFFLINE and S2_ONLINE do perform a similar function to CMD_STOP
        and CMD_START

   IO REQUEST
        ios2_Command    - CMD_START.

   RESULTS
        ios2_Error      - IOERR_NOCMD.

   SEE ALSO
        S2_ONLINE, S2_OFFLINE
```

A.9 sana2.device/CMD_STOP

```
sana2.device/CMD_STOP                                    sana2.device/CMD_STOP

   NAME
        Stop -- Pause device operation.

   FUNCTION
        There is no way for the driver to keep queuing requests without
        servicing them, so CMD_STOP does not apply to this class of device.
        S2_OFFLINE and S2_ONLINE do perform a similar function to CMD_STOP
        and CMD_START

   IO REQUEST
        ios2_Command    - CMD_STOP.

   RESULTS
        ios2_Error      - IOERR_NOCMD.

   NOTES

   SEE ALSO
        S2_ONLINE, S2_OFFLINE
```

A.10 sana2.device/CMD_UPDATE

```
sana2.device/CMD_UPDATE                                  sana2.device/CMD_UPDATE

   NAME
        Update -- Force packets out to device.

   FUNCTION
        Since there are no device internal buffers, CMD_UPDATE does not
        apply to this class of device.

   IO REQUEST
        ios2_Command    - CMD_UPDATE.

   RESULTS
        ios2_Error      - IOERR_NOCMD.
```

A.11 sana2.device/CMD_WRITE

```
sana2.device/CMD_WRITE                                  sana2.device/CMD_WRITE

   NAME
        Write -- Send packet to the network.

   FUNCTION
        This command causes the packet to be sent to the specified network
        interface. Normally, appropriate packet header and trailer information
        will be added to the packet data when it is sent.  If bit SANA2IOB_RAW
        is set in io_Flags, then the ios2_Data is assumed to contain an entire
        physical frame and will be sent (copied to the wire via
        CopyFromBuffer() unmodified.

        Note that the device should not check to see if the destination
        address is on the local hardware.  Network protocols should realize
        that the packet has a local destination long before it gets to a
        SANA-II driver.

   IO REQUEST
        ios2_Command    - CMD_WRITE.
        ios2_Flags      - Supported flags are:
                            SANA2IOB_RAW
                            SANA2IOB_QUICK
        ios2_PacketType - Packet type to send.
        ios2_DstAddr    - Destination interface address for this packet.
        ios2_DataLength - Length of the Data to be sent.
        ios2_Data       - Abstract data structure which packet data is
                          contained in.

   RESULTS
        ios2_Error      - Zero if successful; non-zero otherwise.
        ios2_WireError  - More specific error number.

   NOTES
        The driver may not directly examine or modify anything pointed to by
        ios2_Data.  It *must* use the requester-provided functions to access
        this data.

   SEE ALSO
        CMD_READ, S2_BROADCAST, S2_MULTICAST, any_protocol/CopyFromBuffer
```

A.12 sana2.device/OpenDevice

```
sana2.device/OpenDevice                                  sana2.device/OpenDevice

   NAME
        Open -- Request an opening of the network device.

   SYNOPSIS
        error = OpenDevice(unit, IOSana2Req, flags)
        D0                 D0    A1          D1

        BYTE OpenDevice(ULONG, struct IOSana2Req *, ULONG);

   FUNCTION
        This function is called by exec.library OpenDevice().

        This function performs whatever initialization is required per
        device open and initializes the Sana2Req for use by the
        device.

   INPUTS
        unit            - Device unit to open.
        Sana2Req        - Pointer to IOSana2Req structure to be initialized by
                          the sana2.device.
        flags           - Supported flags are:
                                SANA2OPB_MINE
                                SANA2OPB_PROM
        ios2_BufferManagement   - A pointer to a tag list containing
                                  pointers to buffer management functions.

   RESULTS
        error           - same as io_Error
        io_Error        - Zero if successful; non-zero otherwise.
        io_Device       - A pointer to whatever device will handle the calls
                          for this unit.  This pointer may be different
                          depending on what unit is requested.
        ios2_BufferManagement   - A pointer to device internal information
                                  used to call buffer management functions.

   NOTES
        A SANA-II device must reject all open requests with a request
        structure that is too short, e.g. an IOStdReq. A simple check of the
        mn_Length field in the Message part of the request is needed to make
        sure that a device does not dereference invalid data due to a wrong
        device configuration.

        A SANA-II device may open if no buffer management tags are provided to
        make the configuration process and obtaining statistics easier. Buffer
        management tags with a NULL value must be treated as not specified.
        The device shall fail requests gracefully depending on the missing
        tags in this case. Any malfunction is not acceptable.

   SEE ALSO
        exec.library/OpenDevice(), exec.library/CloseDevice()
```

A.13 sana2.device/S2_ADDMULTICASTADDRESS

```
sana2.device/S2_ADDMULTICASTADDRESS       sana2.device/S2_ADDMULTICASTADDRESS

   NAME
        AddMulticastAddress -- Enable an interface multicast address.

   FUNCTION
        This command causes the device driver to enable multicast packet
        reception for the requested address.

   IO REQUEST
        ios2_Command    - S2_ADDMULTICASTADDRESS.
        ios2_SrcAddr    - Multicast address to enable.

   RESULTS
        ios2_Error      - Zero if successful; non-zero otherwise.
        ios2_WireError  - More specific error number.

   NOTES
        Multicast addresses are added globally -- anyone using the device
        may receive packets as a result of any multicast address which has
        been added for the device.

        Since multicast addresses are not "bound" to a particular packet type,
        each enabled multicast address has an "enabled" count associated with
        it so that if two protocols add the same multicast address and later
        one removes it, it is still enabled until the second removes it.

   SEE ALSO
        S2_MULTICAST, S2_DELMULTICASTADDRESS
```

A.14 sana2.device/S2_ADDMULTICASTADDRESSES

```
sana2.device/S2_ADDMULTICASTADDRESSES     sana2.device/S2_ADDMULTICASTADDRESSES

   NAME
    AddMulticastAddresses -- Enable a range of interface multicast addresses.

   FUNCTION
    This command causes the device driver to enable multicast packet
    reception for the requested address range.

   IO REQUEST
    ios2_Command    - S2_ADDMULTICASTADDRESSES.
    ios2_SrcAddr    - Lowest Multicast address to enable.
    ios2_DstAddr    - Highest Multicast address to enable.

   RESULTS
    ios2_Error      - Zero if successful; non-zero otherwise.
    ios2_WireError  - More specific error number.

   NOTES
    Multicast address ranges are added globally -- anyone using the device
    may receive packets as a result of any multicast address range which
    has been added for the device.

    Since multicast address ranges are not "bound" to a particular packet
    type, each enabled multicast address range has an "enabled" count
    associated with it so that if two protocols add the same multicast
    address range and later one removes it, it is still enabled until
    the second removes it.

    An "address range" consists of all valid Multicast addresses between
    the specified addresses, i.e. ios2_SrcAddr <= address <= ios2_DstAddr.
    In this context "<=" has the intuitive meaning: it denotes a binary
    comparison, where each multicast address is treated as a binary
    number in Big Endian byte order.

    For some hardware types it is possible that a specified address range
    includes some multicast addresses and some unicast or broadcast
    addresses. In this case this command only enables the reception of
    packets sent to the multicast addresses defined by the address range,
    not the reception of packets sent to unicast or broadcast addresses
    within the same address range.

    This command is primarily designed to allow an application to receive
    packets sent to a potentially extremely large number of multicast
    addresses, i.e. it is imperative that a device driver does not
    implement this command by repeatedly executing S2_ADDMULTICASTADDRESS
    for each address in the range.

   SEE ALSO
    S2_MULTICAST, S2_DELMULTICASTADDRESSES, S2_ADDMULTICASTADDRESS,
```

A.15 sana2.device/S2_BROADCAST

```
sana2.device/S2_BROADCAST                          sana2.device/S2_BROADCAST

   NAME
        Broadcast -- Broadcast a packet on network.

   FUNCTION
        This command works the same as CMD_WRITE except that it also performs
        whatever special processing of the packet is required to do a
        broadcast send. The actual broadcast mechanism is neccessarily
        network/interface/device specific.

   IO REQUEST
        ios2_Command    - S2_BROADCAST.
        ios2_Flags      - Supported flags are:
                                SANA2IOB_RAW
                                SANA2IOB_QUICK
        ios2_PacketType - Packet type to send.
        ios2_DataLength - Length of the Data to be sent.
        ios2_Data       - Abstract data structure which packet data is
                          contained in.

   RESULTS
        ios2_DstAddr    - The contents of this field are to be
                          considered trash upon return of the IOReq.
        ios2_Error      - Zero if successful; non-zero otherwise.
                          This command can fail for many reasons and
                          is not supported by all networks and/or
                          network interfaces.
        ios2_WireError  - More specific error number.

   NOTES
        The DstAddr field may be trashed by the driver because this function
        may be implemented by filling DstAddr with a broadcast address and
        internally calling CMD_WRITE.

   SEE ALSO
        CMD_WRITE, S2_MULTICAST
```

A.16 sana2.device/S2_CONFIGINTERFACE

```
sana2.device/S2_CONFIGINTERFACE              sana2.device/S2_CONFIGINTERFACE

   NAME
        ConfigInterface -- Configure the network interface.

   FUNCTION
        This command causes the device driver to initialize the interface
        hardware and to set the network interface address to the address in
        ios2_SrcAddr. This command can only be executed once and, if
        successful, will leave the driver and network interface fully
        operational and the network interface in ios2_SrcAddr.

        To set the interface address to the factory address, the network
        management software must use GetStationAddress first and then call
        ConfigInterface with the result. If there is no factory address then
        the network software must pick an address to use.

        Until this command is executed the device will not listen for any
        packets on the hardware.

   IO REQUEST
        ios2_Command    - S2_CONFIGINTERFACE.
        ios2_Flags      - Supported flags are:
                                SANA2IOB_QUICK
        ios2_SrcAddr    - Address for this interface.

   RESULTS
        ios2_Error      - Zero if successful; non-zero otherwise.
        ios2_WireError  - More specific error number.
        ios2_SrcAddr    - Address of this interface as configured.

   NOTES
        Some networks have the interfaces choose a currently unused interface
        address each time the interface is initialized. The caller must check
        ios2_SrcAddr for the actual interface address after configuring the
        interface.

   SEE ALSO
        S2_GETSTATIONADDRESS
```

A.17 sana2.device/S2_DELMULTICASTADDRESS

```
sana2.device/S2_DELMULTICASTADDRESS       sana2.device/S2_DELMULTICASTADDRESS

   NAME
        DelMultiCastAddress -- Disable an interface multicast address.

   FUNCTION
        This command causes device driver to disable multicast packet
        reception for the requested address.

        It is an error to disable a multicast address that is not enabled.

   IO REQUEST
        ios2_Command    - S2_DELMULTICASTADDRESS
        ios2_SrcAddr    - Multicast address to disable.

   RESULTS
        ios2_Error      - Zero if successful; non-zero otherwise.
        ios2_WireError  - More specific error number.

   NOTES
        Multicast addresses are added globally -- anyone using the device
        may receive packets as a result of any multicast address which has
        been added for the device.

        Since multicast addresses are not "bound" to a particular packet type,
        each enabled multicast address has an "enabled" count associated with
        it so that if two protocols add the same multicast address and later
        one removes it, it is still enabled until the second removes it.

   SEE ALSO
        S2_ADDMULTICASTADDRESS
```

A.18 sana2.device/S2_DELMULTICASTADDRESSES

```
sana2.device/S2_DELMULTICASTADDRESSES       sana2.device/S2_DELMULTICASTADDRESSES

   NAME
    DelMultiCastAddresses -- Disable a range of interface multicast addresses.

   FUNCTION
    This command causes the device driver to disable multicast packet
    reception for the requested address range.

    It is an error to disable a multicast address range that is not enabled.

   IO REQUEST
    ios2_Command    - S2_DELMULTICASTADDRESSES
    ios2_SrcAddr    - Lowest Multicast address to disable.
    ios2_DstAddr    - Highest Multicast address to disable.

   RESULTS
    ios2_Error      - Zero if successful; non-zero otherwise.
    ios2_WireError  - More specific error number.

   NOTES
    Multicast address ranges are added globally -- anyone using the device
    may receive packets as a result of any multicast address range which
    has been added for the device.

    Since multicast address ranges are not "bound" to a particular packet
    type, each enabled multicast address range has an "enabled" count
    associated with it so that if two protocols add the same multicast
    address range and later one removes it, it is still enabled until
    the second removes it.

    The address range specified must exactly match an address range
    used earlier in an S2_ADDMULTICASTADDRESSES call. The device driver
    is not supposed to automatically merge consecutive address ranges,
    to automatically split up address ranges, or to perform any other
    types of "address range arithmetic" to support non-matching
    addition/deletion requests.

    It is an error to use S2_DELMULTICASTADDRESSES to disable a single
    address added earlier with S2_ADDMULTICASTADDRESS, or to use
    S2_DELMULTICASTADDRESS to disable an address range that consists of
    just a single address and was added earlier with
    S2_ADDMULTICASTADDRESSES.

   SEE ALSO
    S2_ADDMULTICASTADDRESSES
```

A.19 sana2.device/S2_DEVICEQUERY

NAME

DeviceQuery -- Return parameters for this network interface.

FUNCTION

This command causes the device driver to report information about the device. Up to SizeAvailable bytes of the information is copied into a buffer pointed to by ios2_StatData. The format of the data is as follows:

```
struct Sana2DeviceQuery
{
/*
** Standard information
*/
    ULONG SizeAvailble; /* bytes available */
    ULONG SizeSupplied; /* bytes supplied */
    LONG  DevQueryFormat;   /* this is type 0 */
    LONG  DeviceLevel;      /* this document is level 0 */

/*
** Common information
*/
    UWORD AddrFieldSize;    /* address size in bits */
    ULONG MTU;              /* maximum packet data size */
    LONG  bps;              /* line rate (bits/sec) */
    LONG  HardwareType;     /* what the wire is */

/*
** Format specific information
*/
};
```

The SizeAvailable specifies the number of bytes that the caller is prepared to accomodate, including the standard information fields.

SizeSupplied is the number of bytes actually supplied, including the standard information fields, which will not exceed SizeAvailable.

<devices/sana2.h> includes constants for these values. If your hardware does not have a number assigned to it, you must contact Amiga to get a hardware number.

IO REQUEST

ios2_Command - S2_DEVICEQUERY.
ios2_StatData - Pointer to Sana2DeviceQuery structure to fill in.

RESULTS

ios2_Error - Zero if successful; non-zero otherwise.
ios2_WireError - More specific error number.

A.20 sana2.device/S2_GETGLOBALSTATS

```
sana2.device/S2_GETGLOBALSTATS                sana2.device/S2_GETGLOBALSTATS

   NAME
        GetGlobalStats -- Get interface accumulated statistics.

   FUNCTION
        This command causes the device driver to retrieve various global
        runtime statistics for this network interface. The format of the
        data returned is as follows:

            struct Sana2DeviceStats
            {
                ULONG PacketsReceived;
                ULONG PacketsSent;
                ULONG BadData;
                ULONG Overruns;
                ULONG UnknownTypesReceived;
                ULONG Reconfigurations;
                timeval LastStart;
            };

   IO REQUEST
        ios2_Command    - S2_GETGLOBALSTATS.
        ios2_StatData   - Pointer to Sana2DeviceStats structure to fill.

   RESULTS
        ios2_Error      - Zero if successful; non-zero otherwise.
        ios2_WireError  - More specific error number.

   SEE ALSO
        S2_GETSPECIALSTATS
```

A.21 sana2.device/S2_GETSPECIALSTATS

```
sana2.device/S2_GETSPECIALSTATS              sana2.device/S2_GETSPECIALSTATS

   NAME
        GetSpecialStats -- Get network type specific statistics.

   FUNCTION
        This function returns statistics which are specific to the type of
        network medium this driver controls. For example, this command could
        return statistics common to all Ethernets which are not common to all
        network mediums in general.

        The supplied Sana2SpecialStatData structure is given below:

            struct Sana2SpecialStatData
            {
                ULONG RecordCountMax;
                ULONG RecordCountSupplied;
                struct Sana2StatRecord[RecordCountMax];
            };

        The format of the data returned is:

            struct Sana2StatRecord
            {
                ULONG Type;     /* Amiga registered */
                LONG Count;     /* the stat itself */
                char *String;   /* null terminated */
            };

        The RecordCountMax field specifies the number of records that the
        caller is prepared to accomodate.

        RecordCountSupplied is the number of record actually supplied which
        will not exceed RecordCountMax.

   IO REQUEST
        ios2_Command    - S2_GETSPECIALSTATS.
        ios2_StatData   - Pointer to a Sana2SpecialStatData structure to fill.
                          RecordCountMax must be initialized.

   RESULTS
        ios2_Error      - Zero if successful; non-zero otherwise.
        ios2_WireError  - More specific error number.

   NOTES
        Amiga will maintain registered statistic Types.

   SEE ALSO
        S2_GETGLOBALSTATS, <devices/sana2specialstats.h>
```

A.22 sana2.device/S2_GETSTATIONADDRESS

```
sana2.device/S2_GETSTATIONADDRESS          sana2.device/S2_GETSTATIONADDRESS

   NAME
        GetStationAddress -- Get default and interface address.

   FUNCTION
        This command causes the device driver to copy the current interface
        address into ios2_SrcAddr, and to copy the factory default station
        address (if any) into ios2_DstAddr.

   IO REQUEST
        ios2_Command    - S2_GETSTATIONADDRESS.

   RESULTS
        ios2_Error      - Zero if successful; non-zero otherwise.
        ios2_WireError  - More specific error number.
        ios2_SrcAddr    - Current interface address.
        ios2_DstAddr    - Default interface address (if any).

   SEE ALSO
        S2_CONFIGINTERFACE
```

A.23 sana2.device/S2_GETTYPESTATS

```
sana2.device/S2_GETTYPESTATS                    sana2.device/S2_GETTYPESTATS

   NAME
        GetTypeStats -- Get accumulated type specific statistics.

   FUNCTION
        This command causes the device driver to retrieve various packet type
        specific runtime statistics for this network interface. The format of
        the data returned is as follows:

            struct Sana2TypeStatData
            {
                LONG PacketsSent;
                LONG PacketsReceived;
                LONG BytesSent;
                LONG BytesReceived;
                LONG PacketsDropped;
            };

   IO REQUEST
        ios2_Command    - S2_GETTYPESTATS.
        ios2_PacketType - Packet type of interest.
        ios2_StatData   - Pointer to TypeStatData structure to fill in.

   RESULTS
        ios2_Error      - Zero if successful; non-zero otherwise.
        ios2_WireError  - More specific error number.

   NOTES
        Statistics for a particular packet type are only available while that
        packet type is being ``tracked''.

   SEE ALSO
        S2_TRACKTYPE, S2_UNTRACKTYPE
```

A.24 sana2.device/S2_MULTICAST

```
sana2.device/S2_MULTICAST                          sana2.device/S2_MULTICAST

   NAME
        Multicast -- Multicast a packet on network.

   FUNCTION
        This command works the same as CMD_WRITE except that it also performs
        whatever special processing of the packet is required to do a
        multicast send. The actual multicast mechanism is neccessarily
        network/interface/device specific.

   IO REQUEST
        ios2_Command    - S2_MULTICAST.
        ios2_Flags      - Supported flags are:
                                SANA2IOB_RAW
                                SANA2IOB_QUICK
        ios2_PacketType - Packet type to send.
        ios2_DstAddr    - Destination interface address for this packet.
        ios2_DataLength - Length of the Data to be sent.
        ios2_Data       - Abstract data structure which packet data is
                          contained in.

   RESULTS
        ios2_Error      - Zero if successful; non-zero otherwise.
                          This command can fail for many reasons and
                          is not supported by all networks and/or
                          network interfaces.
        ios2_WireError  - More specific error number.

   NOTES
        The address supplied in ios2_DstAddr will be sanity checked (if
        possible) by the driver. If the supplied address fails this sanity
        check, the multicast request will fail immediately with ios2_Error
        set to S2WERR_BAD_MULTICAST.

        Another Amiga will not receive a multicast packet unless it has had
        the particular multicast address being used S2_ADDMULTICASTADRESS'd.

   SEE ALSO
        CMD_WRITE, S2_BROADCAST, S2_ADDMULTICASTADDRESS
```

A.25 sana2.device/S2_OFFLINE

```
sana2.device/S2_OFFLINE                              sana2.device/S2_OFFLINE

   NAME
        Offline -- Remove interface from service.

   FUNCTION
        This command removes a network interface from service.

   IO REQUEST
        ios2_Command    - S2_OFFLINE.

   RESULTS
        ios2_Error      - Zero if successful; non-zero otherwise.
        ios2_WireError  - More specific error number.

   NOTES
        Aborts all pending reads and writes with ios2_Error set to
        S2ERR_OUTOFSERVICE.

        While the interface is offline, all read, writes and any other
        command that touches interface hardware will be rejected with
        ios2_Error set to S2ERR_OUTOFSERVICE.

        This command is intended to permit a network interface to be
        tested on an otherwise live system.

   SEE ALSO
        S2_ONLINE
```

A.26 sana2.device/S2_ONEVENT

```
sana2.device/S2_ONEVENT                                 sana2.device/S2_ONEVENT

   NAME
        OnEvent -- Return when specified event occures.

   FUNCTION
        This command returns when a particular event condition has occured
        on the network or this network interface.

   IO REQUEST
        ios2_Command    - S2_ONEVENT.
        ios2_Flags      - Supported flags are:
                                SANA2IOB_QUICK
        ios2_WireError  - Mask of event(s) to wait for
                          (from <devices/sana2.h>).

   RESULTS
        ios2_Error      - Zero if successful; non-zero otherwise.
        ios2_WireError  - Mask of events that occured.

   NOTES
        If this device driver does not understand the specified event
        condition(s) then the command returns immediately with
        ios2_Req.io_Error set to S2_ERR_NOT_SUPPORTED and ios2_WireError
        S2WERR_BAD_EVENT.  A successful return will have ios2_Error set to
        zero ios2_WireError set to the event number.

        All pending requests for a particular event will be returned when
        that event occurs.

        All event types that cover a particular condition are returned when
        that condition occures. For instance, if an error is returned by
        a buffer management function during receive processing, events of
        types S2EVENT_ERROR, S2EVENT_RX and S2EVENT_BUFF would be returned if
        pending.

        Types ONLINE and OFFLINE return immediately if the device is
        already in the state to be waited for.
```

A.27 sana2.device/S2_ONLINE

```
sana2.device/S2_ONLINE                                  sana2.device/S2_ONLINE

   NAME
        Online -- Put a network interface back in service.

   FUNCTION
        This command places an offline network interface back into service.

   IO REQUEST
        ios2_Command    - S2_ONLINE.

   RESULTS
        ios2_Error      - Zero if successful; non-zero otherwise.
        ios2_WireError  - More specific error number.

   NOTES
        This command is responsible for putting the network interface
        hardware back into a known state (as close as possible to the
        state before S2_OFFLINE) and resets the unit global and special
        statistics.

   SEE ALSO
        S2_OFFLINE
```

A.28 sana2.device/S2_READORPHAN

```
sana2.device/S2_READORPHAN                    sana2.device/S2_READORPHAN

   NAME
        ReadOrphan -- Get a packet for which there is no reader.

   FUNCTION
        Get the next packet available that does not satisfy any then-pending
        CMD_READ requests. The data returned in the ios2_Data structure is
        normally the Data Link Layer packet type field and the packet data. If
        bit SANA2IOB_RAW is set in ios2_Flags, then the entire Data Link Layer
        packet, including both header and trailer information, will be
        returned.

   IO REQUEST
        ios2_Command    - CMD_READORPHAN.
        ios2_Flags      - Supported flags are:
                              SANA2IOB_RAW
                              SANA2IOB_QUICK
        ios2_DataLength - Length of the Data to be sent.
        ios2_Data       - Abstract data structure which packet data is
                          contained in.

   RESULTS
        ios2_Error      - Zero if successful; non-zero otherwise.
        ios2_WireError  - More specific error number.
        ios2_Flags      - The following flags may be returned:
                              SANA2IOB_RAW
                              SANA2IOB_BCAST
                              SANA2IOB_MCAST
        ios2_SrcAddr    - Source interface address of packet.
        ios2_DstAddr    - Destination interface address of packet.
        ios2_DataLength - Length of the Data to be sent.
        ios2_Data       - Abstract data structure which packet data is
                          contained in.

   NOTES
        This is intended for debugging and management tools.  Protocols should
        not use this.

        As with 802.3 packets on an ethernet, to determine which protocol
        family the returned packet belongs to you may have to specify
        SANA2IOB_RAW to get the entire data link layer wrapper (which is where
        the protocol type may be kept). Notice this necessarily means that
        this cannot be done in a network interface independent fashion.  The
        driver will, however, fill in the PacketType field to the best of its
        ability.

   SEE ALSO
        CMD_READ, CMD_WRITE
```

A.29 sana2.device/S2_TRACKTYPE

```
sana2.device/S2_TRACKTYPE                          sana2.device/S2_TRACKTYPE

   NAME
        TrackType -- Accumulate statistics about a packet type.

   FUNCTION
        This command causes the device driver to accumulate statistics about
        a particular packet type. Packet type statistics, for the particular
        packet type, are zeroed by this command.

   IO REQUEST
        ios2_Command    - S2_TRACKTYPE.
        ios2_PacketType - Packet type of interest.

   RESULTS
        ios2_Error      - Zero if successful; non-zero otherwise.
        ios2_WireError  - More specific error number.

   SEE ALSO
        S2_UNTRACKTYPE, S2_GETTYPESTATS
```

A.30 sana2.device/S2_UNTRACKTYPE

```
sana2.device/S2_UNTRACKTYPE                    sana2.device/S2_UNTRACKTYPE

   NAME
        UntrackType -- End statistics about a packet type.

   FUNCTION
        This command causes the device driver to stop accumulating
        statistics about a particular packet type.

   IO REQUEST
        ios2_Command    - S2_UNTRACKTYPE.
        ios2_PacketType - Packet type of interest.

   RESULTS
        ios2_Error      - Zero if successful; non-zero otherwise.
        ios2_WireError  - More specific error number.

   SEE ALSO
        S2_TRACKTYPE, S2_GETTYPESTATS
```

B Callback mechanism Autodocs

B.1 CopyFromBuff

```
CopyFromBuff                                                    CopyFromBuff

   NAME
        CopyFromBuff -- Copy n bytes from an abstract data structure.

   SYNOPSIS
        success = CopyFromBuff(to, from, n)
        d0                     a0  a1    d0

        BOOL CopyToBuff(VOID *, VOID *, ULONG);

   FUNCTION
        This function copies 'n' bytes of data in the abstract data structure
        pointed to by 'from' into the contigous memory pointed to by 'to'.
        'to' must contain at least 'n' bytes of usable memory or innocent
        memory will be overwritten.

   INPUTS
        to              - pointer to contiguous memory to copy to.
        from            - pointer to abstract structure to copy from.
        n               - number of bytes to copy.

   RESULT
        success         - TRUE if operation was successful, else FALSE.

   NOTES
        This function must be callable from interrupts.  In particular, this
        means that this function may not directly or indirectly call any
        system memory functions (since those functions rely on Forbid() to
        protect themselves) and that  you must not compile this function
        with stack checking enabled.  See the RKM:Libraries Exec:Interrupts
        chapter for more details on what is legal in a routine called from
        an interrupt handler.

        'C' programmers should not compile with stack checking (option '-v'
        in SAS) and should geta4() or __saveds.
```

B.2 CopyToBuff

```
CopyToBuff                                                    CopyToBuff

   NAME
        CopyToBuff -- Copy n bytes to an abstract data structure.

   SYNOPSIS
        success = CopyToBuff(to, from, n)
        d0                   a0  a1    d0

        BOOL CopyToBuff(VOID *, VOID *, ULONG);

   FUNCTION
        This function first does any initialization and/or allocation
        required to prepare the abstract data structure pointed at by 'to'
        to be filled with 'n' bytes of data from 'from'.  It then executes
        the copy operation.

        If, for example, there is not enough memory available to prepare
        the abstract data structure, the call is failed and FALSE is returned.

        The buffer management scheme should be such that any memory needed
        to fulfill CopyToBuff() calls is already allocated from the system
        before the call to CopyToBuff() is made.

   INPUTS
        to              - pointer to abstract structure to copy to.
        from            - pointer to contiguous memory to copy from.
        n               - number of bytes to copy.

   RESULT
        success         - TRUE if operation was successful, else FALSE.

   NOTES
        This function must be callable from interrupts.  In particular, this
        means that this function may not directly or indirectly call any
        system memory functions (since those functions rely on Forbid() to
        protect themselves) and that you must not compile this function
        with stack checking enabled.  See the RKM:Libraries Exec:Interrupts
        chapter for more details on what is legal in a routine called from
        an interrupt handler.

        'C' programmers should not compile with stack checking (option '-v'
        in SAS) and should geta4() or __saveds.
```

B.3 PacketFilter

```
PacketFilter                                                    PacketFilter

   NAME
        PacketFilter -- Perform filtering operation on CMD_READ's.

   SYNOPSIS
        keep = PacketFilter(hook, ios2, data)
        d0                  a0    a2    a1

        BOOL PacketFilter(struct Hook *, struct IOSana2Req *, APTR);

   FUNCTION
        This function (if supplied by a protocol stack) may be used to
        reject packets before they are copied into a protocol stack's
        internal buffers.

        The IOSana2Req structure should be set up to look (almost) exactly
        as it would if it was successfully returned for the current packet.
        Specifically, the fields that should be set up correctly are:

        ios2->ios2_DataLength
        ios2->ios2_SrcAddr
        ios2->ios2_DstAddr

       The "data" pointer must point to the beginning of the packet data
        that is stored in contiguous memory.  The data should NOT include
        any hardware specific headers (unless of course the CMD_READ
        request wanted RAW packets).

   INPUTS
        hook            - pointer to the Hook originally supplied during
                          OpenDevice().
        ios2            - The IOSana2Req CMD_READ request that will be used
                          (the "object" of the Hook call).
        data            - The packet data (the "message" of the Hook call).

   RESULT
        success         - TRUE if the driver should provide the packet to
                          the protocol stack, FALSE if the packet should be
                          ignored.

   NOTES
        This function must be callable from interrupts.  In particular, this
        means that this function may not directly or indirectly call any
        system memory functions (since those functions rely on Forbid() to
        protect themselves) and that  you must not compile this function
        with stack checking enabled.  See the RKM:Libraries Exec:Interrupts
        chapter for more details on what is legal in a routine called from
        an interrupt handler.

        'C' programmers should not compile with stack checking (option '-v'
        in SAS) and should geta4() or __saveds.

        What does packet filtering do? With the original ``SANA-II Network
        Device Driver Specification'', a protocol stack could open a device
        and ask for certain packet types. It got all the packets that matched
        this type. As it turned out, this could be mighty inefficient if there
        were packets that the protocol stack did not use at all. These would
        go into read processing of the protocol stack and waste CPU time even
        though they could have been easily identified on arrival.
```

C Ethernet description

```
#define S2WireType_Ethernet             1

ios2_DataLength:
        valid ethernet packets have 64 to 1500 bytes of data.

Address format:
        Ethernet addresses consist of 47 bits of address information and
        a 1 bit multicast flag. The standard for expressing ethernet
        addresses is as 6 bytes (octets) in the order in which the bytes
        are transmitted with the low-order bits in a byte transmitted
        first. The multicast flag bit is the least-significant bit of the
        first byte.

        Ethernet addresses in a Sana2IOReq occupy the first 6 bytes of
        an address field in transmission order with the low-order bits in
        a byte transmitted first.

Station Address:
        Each ethernet board must have a unique ethernet hardware address.
        Drivers will override any attempt to set the address to anything
        other than the ROM address.

Raw reads and writes:
        6 bytes of destination address,
        6 bytes of source address,
        2 bytes of type,
        64 to 1500 bytes of data
        (followed by 4 byte CRC value covering all of the above
         which is hardware generated and checked, hence not included
         in even raw packets)

Multicast:      Supported

Broadcast:      Supported

Promiscuous:    Supported

Packet Type Numbers for Ethernet are assigned by:

        Xerox Corporation
        Xerox Systems Institute
        475 Oakmead Parkway, Sunnyvale, CA 94086
        Attn: Ms. Fonda Pallone
        (408) 737-4652


Some Common Packet Type Numbers:

      decimal  Hex        Description
      -------  ---        -----------
         000   0000-05DC  IEEE 802.3 Length Field
        2048   0800       TCP/IP -- IP
        2054   0806       TCP/IP -- ARP
       32821   8035       TCP/IP -- RARP
       32923   809B       Appletalk
       33011   80F3       AppleTalk AARP (Kinetics)
       33100   814C       SNMP
       33079   8137-8138  Novell, Inc.
```

D ARCNET description

```
S2WireType_Arcnet       7

ios2_DataLength:
        506 byte MTU (because of the possibility of two byte Types).
        Packets of size 254, 255, or 256 bytes are padded to 257 bytes before
        transmition.

Station Address:
        ARCNET hardware may have addresses set with jumpers, DIP switches
        or software.  Different drivers may therefore behave differently
        with S2_CONFIGINTERFACE.

        Hardware addresses should be assigned by users from highest to lowest
        because there is some efficiency gained in the token passing scheme
        this way.  For example, on a three node network, hardware numbers 254,
        253 and 252 should be used rather than 1, 2 and 3.

Raw reads and writes:
        Short Packets (1-253 bytes)
                Destination Address             (1 byte)
                Source Address                  (1 byte)
                Count (256-N-Type length)       (1 byte)
                Padding                         (to byte number Count)
                Type                            (1 or 2 bytes)
                Data                            (N bytes)

        Long Packets (257-506 bytes)
                Destination Address             (1 byte)
                Source Address                  (1 byte)
                zero                            (1 byte)
                Count (512-N-Type length)       (1 byte)
                Padding                         (to byte number Count)
                Type                            (1 or 2 bytes)
                Data                            (N bytes)


Multicast:      Not Supported

Broadcast:      Supported

Promiscuous:    Generally Not Supported

Packet Type Numbers for ARCNET are assigned by:
        Datapoint Corporation

Some Common Packet Type Numbers

        decimal  hex    description
        -------  ---    -----------
        221      DD     AppleTalk
        240      F0     TCP/IP -- IP   (RFC 1051)
        241      F1     TCP/IP -- ARP  (RFC 1051)
        212      F0     TCP/IP    IP   (RFC 1201, proposed)
        213      F1     TCP/IP -- ARP  (RFC 1201, proposed)
        214      D6     TCP/IP -- RARP (RFC 1201, proposed)
        247      F7     Banyan Vines
        250      FA     Novell IPX
```

E “sana.h” header file

```
#ifndef SANA2_SANA2DEVICE_H
#define SANA2_SANA2DEVICE_H 1
/*
**      $VER: sana2.h 50.1 (15.12.2002)
**      Includes Release 50.1
**
**      Structure definitions for SANA-II devices.
**
**      (C) Copyright 1991-2002 Amiga, Inc.
**              All Rights Reserved
*/


#ifndef EXEC_TYPES_H
#include <exec/types.h>
#endif

#ifndef EXEC_PORTS_H
#include <exec/ports.h>
#endif

#ifndef EXEC_IO_H
#include <exec/io.h>
#endif

#ifndef EXEC_ERRORS_H
#include <exec/errors.h>
#endif

#ifndef DEVICES_TIMER_H
#include <devices/timer.h>
#endif

#ifndef UTILITY_TAGITEM_H
#include <utility/tagitem.h>
#endif


#define SANA2_MAX_ADDR_BITS     (128)
#define SANA2_MAX_ADDR_BYTES    ((SANA2_MAX_ADDR_BITS+7)/8)

struct IOSana2Req
{
        struct IORequest ios2_Req;
        ULONG ios2_WireError;           /* wire type specific error     */
        ULONG ios2_PacketType;          /* packet type                  */
        UBYTE ios2_SrcAddr[SANA2_MAX_ADDR_BYTES];       /* source addr  */
        UBYTE ios2_DstAddr[SANA2_MAX_ADDR_BYTES];       /* dest address */
        ULONG ios2_DataLength;          /* length of packet data        */
        VOID *ios2_Data;                /* packet data                  */
        VOID *ios2_StatData;            /* statistics data pointer      */
        VOID *ios2_BufferManagement;    /* see SANA-II OpenDevice adoc  */
};


/*
** defines for the io_Flags field
*/
#define SANA2IOB_RAW    (7)             /* raw packet I/O requested     */
#define SANA2IOF_RAW    (1<<SANA2IOB_RAW)
```

```
#define SANA2IOB_BCAST  (6)             /* broadcast packet (received)  */
#define SANA2IOF_BCAST  (1<<SANA2IOB_BCAST)

#define SANA2IOB_MCAST  (5)             /* multicast packet (received)  */
#define SANA2IOF_MCAST  (1<<SANA2IOB_MCAST)

#define SANA2IOB_QUICK  (IOB_QUICK)     /* quick IO requested (0)       */
#define SANA2IOF_QUICK  (IOF_QUICK)


/*
** defines for OpenDevice() flags
*/
#define SANA2OPB_MINE   (0)             /* exclusive access requested   */
#define SANA2OPF_MINE   (1<<SANA2OPB_MINE)

#define SANA2OPB_PROM   (1)             /* promiscuous mode requested   */
#define SANA2OPF_PROM   (1<<SANA2OPB_PROM)


/*
** defines for OpenDevice() tags
*/
#define S2_Dummy             (TAG_USER + 0xB0000)

#define S2_CopyToBuff        (S2_Dummy + 1)
#define S2_CopyFromBuff      (S2_Dummy + 2)
#define S2_PacketFilter      (S2_Dummy + 3)
#define S2_CopyToBuff16      (S2_Dummy + 4)
#define S2_CopyFromBuff16    (S2_Dummy + 5)
#define S2_CopyToBuff32      (S2_Dummy + 6)
#define S2_CopyFromBuff32    (S2_Dummy + 7)
#define S2_DMACopyToBuff32   (S2_Dummy + 8)
#define S2_DMACopyFromBuff32 (S2_Dummy + 9)

struct Sana2DeviceQuery
{
/*
** Standard information
*/
        ULONG   SizeAvailable;          /* bytes available              */
        ULONG   SizeSupplied;           /* bytes supplied               */
        ULONG   DevQueryFormat;         /* this is type 0               */
        ULONG   DeviceLevel;            /* this document is level 0     */
/*
** Common information
*/
        UWORD   AddrFieldSize;          /* address size in bits         */
        ULONG   MTU;                    /* maximum packet data size     */
        ULONG   BPS;                    /* line rate (bits/sec)         */
        ULONG   HardwareType;           /* what the wire is             */
/*
** Format specific information
*/
};


/*
** defined Hardware types
**
**  If your hardware type isn't listed below, contact Amiga to get a new
```

```
**  type number added for your hardware.
*/
#define S2WireType_Ethernet             1
#define S2WireType_IEEE802              6
#define S2WireType_Arcnet               7
#define S2WireType_LocalTalk            11
#define S2WireType_DyLAN                12

#define S2WireType_AmokNet              200     /* Amiga Floppy Port hardware */

#define S2WireType_Liana                202     /* Village Tronic parallel port hw */

#define S2WireType_PPP                  253
#define S2WireType_SLIP                 254
#define S2WireType_CSLIP                255     /* Compressed SLIP */

#define S2WireType_PLIP                 420     /* SLIP over a parallel port */

struct Sana2PacketTypeStats
{
        ULONG PacketsSent;              /* transmitted count            */
        ULONG PacketsReceived;          /* received count               */
        ULONG BytesSent;                /* bytes transmitted count      */
        ULONG BytesReceived;            /* bytes received count         */
        ULONG PacketsDropped;           /* packets dropped count        */
};


struct Sana2SpecialStatRecord
{
        ULONG Type;                     /* statistic identifier         */
        ULONG Count;                    /* the statistic                */
        char *String;                   /* statistic name               */
};


struct Sana2SpecialStatHeader
{
        ULONG RecordCountMax;           /* room available               */
        ULONG RecordCountSupplied;      /* number supplied              */
        /* struct Sana2SpecialStatRecord[RecordCountMax]; */
};


struct Sana2DeviceStats
{
        ULONG PacketsReceived;          /* received count               */
        ULONG PacketsSent;              /* transmitted count            */
        ULONG BadData;                  /* bad packets received         */
        ULONG Overruns;                 /* hardware miss count          */
        ULONG Unused;                   /* Unused field                 */
        ULONG UnknownTypesReceived;     /* orphan count                 */
        ULONG Reconfigurations;         /* network reconfigurations     */
        struct timeval LastStart;       /* time of last online          */
};


/*
** Device Commands
*/
#define S2_START                (CMD_NONSTD)
```

```
#define S2_DEVICEQUERY          (S2_START+ 0)
#define S2_GETSTATIONADDRESS    (S2_START+ 1)
#define S2_CONFIGINTERFACE      (S2_START+ 2)
#define S2_ADDMULTICASTADDRESS  (S2_START+ 5)
#define S2_DELMULTICASTADDRESS  (S2_START+ 6)
#define S2_MULTICAST            (S2_START+ 7)
#define S2_BROADCAST            (S2_START+ 8)
#define S2_TRACKTYPE            (S2_START+ 9)
#define S2_UNTRACKTYPE          (S2_START+10)
#define S2_GETTYPESTATS         (S2_START+11)
#define S2_GETSPECIALSTATS      (S2_START+12)
#define S2_GETGLOBALSTATS       (S2_START+13)
#define S2_ONEVENT              (S2_START+14)
#define S2_READORPHAN           (S2_START+15)
#define S2_ONLINE               (S2_START+16)
#define S2_OFFLINE              (S2_START+17)

#define S2_END                  (S2_START+18)

/*
** Multicast address range extensions
*/
#define S2_ADDMULTICASTADDRESSES 0xC000
#define S2_DELMULTICASTADDRESSES 0xC001

/*
** defined errors for io_Error  (see also <exec/errors.h>)
*/
#define S2ERR_NO_ERROR          0       /* peachy-keen                  */
#define S2ERR_NO_RESOURCES      1       /* resource allocation failure  */
#define S2ERR_BAD_ARGUMENT      3       /* garbage somewhere            */
#define S2ERR_BAD_STATE         4       /* inappropriate state          */
#define S2ERR_BAD_ADDRESS       5       /* who?                         */
#define S2ERR_MTU_EXCEEDED      6       /* too much to chew             */
#define S2ERR_NOT_SUPPORTED     8       /* hardware can't support cmd   */
#define S2ERR_SOFTWARE          9       /* software error detected      */
#define S2ERR_OUTOFSERVICE      10      /* driver is OFFLINE            */
#define S2ERR_TX_FAILURE        11      /* Transmission attempt failed  */
/*
** From <exec/errors.h>
**
**      IOERR_OPENFAIL   (-1) * device/unit failed to open *
**      IOERR_ABORTED    (-2) * request terminated early [after AbortIO()] *
**      IOERR_NOCMD      (-3) * command not supported by device *
**      IOERR_BADLENGTH  (-4) * not a valid length (usually IO_LENGTH) *
**      IOERR_BADADDRESS (-5) * invalid address (misaligned or bad range) *
**      IOERR_UNITBUSY   (-6) * device opens ok, but requested unit is busy *
**      IOERR_SELFTEST   (-7) * hardware failed self-test *
*/

/*
** defined errors for ios2_WireError
*/
#define S2WERR_GENERIC_ERROR    0       /* no specific info available   */
#define S2WERR_NOT_CONFIGURED   1       /* unit not configured          */
#define S2WERR_UNIT_ONLINE      2       /* unit is currently online     */
#define S2WERR_UNIT_OFFLINE     3       /* unit is currently offline    */
#define S2WERR_ALREADY_TRACKED  4       /* protocol already tracked     */
#define S2WERR_NOT_TRACKED      5       /* protocol not tracked         */
#define S2WERR_BUFF_ERROR       6       /* buff mgt func returned error */
#define S2WERR_SRC_ADDRESS      7       /* source address problem       */
#define S2WERR_DST_ADDRESS      8       /* destination address problem  */
```

```
#define S2WERR_BAD_BROADCAST    9       /* broadcast address problem    */
#define S2WERR_BAD_MULTICAST    10      /* multicast address problem    */
#define S2WERR_MULTICAST_FULL   11      /* multicast address list full  */
#define S2WERR_BAD_EVENT        12      /* unsupported event class      */
#define S2WERR_BAD_STATDATA     13      /* statdata failed sanity check */
#define S2WERR_IS_CONFIGURED    15      /* attempt to config twice      */
#define S2WERR_NULL_POINTER     16      /* null pointer detected        */
#define S2WERR_TOO_MANY_RETRIES 17      /* tx failed - too many retries */
#define S2WERR_RCVREL_HDW_ERR   18      /* Driver fixable HW error      */


/*
** defined events
*/
#define S2EVENT_ERROR           (1L<<0) /* error catch all              */
#define S2EVENT_TX              (1L<<1) /* transmitter error catch all  */
#define S2EVENT_RX              (1L<<2) /* receiver error catch all     */
#define S2EVENT_ONLINE          (1L<<3) /* unit is in service           */
#define S2EVENT_OFFLINE         (1L<<4) /* unit is not in service       */
#define S2EVENT_BUFF            (1L<<5) /* buff mgt function error      */
#define S2EVENT_HARDWARE        (1L<<6) /* hardware error catch all     */
#define S2EVENT_SOFTWARE        (1L<<7) /* software error catch all     */

#endif  /* SANA2_SANA2DEVICE_H */
```

F “sana2specialstats.h” header file

```
#ifndef SANA2_SANA2SPECIALSTATS_H
#define SANA2_SANA2SPECIALSTATS_H 1
/*
**      $VER: sana2specialstats.h 50.1 (15.12.2002)
**      Includes Release 50.1
**
**      Defined IDs for SANA-II special statistics.
**
**      (C) Copyright 1991-2002 Amiga, Inc.
**              All Rights Reserved
*/

#ifndef SANA2_SANA2DEVICE_H
#include <devices/sana2.h>
#endif  /* !SANA2_SANA2DEVICE_H */

/*
** The SANA-II special statistic identifier is an unsigned 32 number.
** The upper 16 bits identify the network wire type to which the
** statistic applies and the lower 16 bits identify the particular
** statistic.
**
** If you desire to add a new statistic identifier, contact Amiga.
*/

/*
** defined ethernet special statistics
*/

#define S2SS_ETHERNET_BADMULTICAST      ((((S2WireType_Ethernet)&0xffff)<<16)|0x0000)
/*
** This count will record the number of times a received packet tripped
** the hardware's multicast filtering mechanism but was not actually in
** the current multicast table.
*/

#define S2SS_ETHERNET_RETRIES           ((((S2WireType_Ethernet)&0xffff)<<16)|0x0001)
/*
** This count records the total number of retries which have resulted
** from transmissions on this board.
*/

#define S2SS_ETHERNET_FIFO_UNDERRUNS    ((((S2WireType_Ethernet)&0xffff)<<16)|0x0002)
/*
** This count records an error condition which indicates that the host
** computer did not feed the network interface card at a high enough rate.
*/

#endif  /* SANA2_SANA2SPECIALSTATS_H */
```