NORSK DATA UNIX† (NDIX) - ND-5000/ND-100 Interface Specification

NDIX Development Group

Norsk Data Ltd
Benham Valence
Newbury
Berkshire
England

ABSTRACT

This document describes the software interface between the ND-5000 and the ND-100 processors for the purposes of the Norsk Data UNIX system implementation.

November 8, 1988

NORSK DATA UNIX† (NDIX) - ND-5000/ND-100 Interface Specification

NDIX Development Group

Norsk Data Ltd
Benham Valence
Newbury
Berkshire
England

1. Introduction

This document describes the software interface between the ND-5000 and the ND-100 processors for the purposes of the Norsk Data UNIX system (NDIX) implementation. The UNIX system is implemented using a hybrid approach in which the UNIX kernel runs in the ND-5000 processor and communicates with the ND-100 front-end processor. The front-end runs a the SINTRAN III operating system which performs all I/O operations on behalf of the ND-5000 processor. The NDIX kernel and user processes run as a single "process" in the ND-5000. Other SINTRAN processes may be run in the ND-5000 when NDIX is not running.

The basic approach adopted in the design of UNIX for the Norsk Data system is to regard the ND-100 as an intelligent I/O Processor which performs functions similar to those provided by I/O controllers of more conventional design. The model has been extended so that the ND-100 can be regarded as interrupting NDIX and vectoring it into an interrupt context. This approach allows the NDIX kernel to maintain full control over the operation of its user processes running in the ND-5000 processor.

Under SINTRAN III, the ND-5000 is regarded as a slave processor and the SINTRAN scheduler, which runs in the ND-100, has complete control over which processes run in the main processor. The NDIX kernel and user processes are regarded as one SINTRAN "process" within this model. SINTRAN plays no part in the choice of NDIX user process running at any time.

NDIX requests the performance of i/o operations by means of "FE" calls‡ sent to the ND-100. When an I/O, or other asynchronous operation is completed within the ND-100, it will stop NDIX, save its context and restart it in an interrupt context at a specific address. Interrupt contexts within NDIX will be grouped together by software into a number of specific priority levels and while an interrupt is being handled on a particular level no other interrupts will be allowed to occur on that, or a lower level. This mechanism, which allows the ND-100 to simulate a series of interrupt priority levels, is a common feature of other architectures on which UNIX has been implemented.

Since the UNIX kernel has a number of critical regions it will be necessary for it to request that specific levels be inhibited for a time. This will be achieved via an area in shared memory containing the current priority level.

---

† UNIX is a trademark of AT&T in the USA and other countries

‡ Documented thoroughly in the interface manual pages included in appendix 3 of this document.

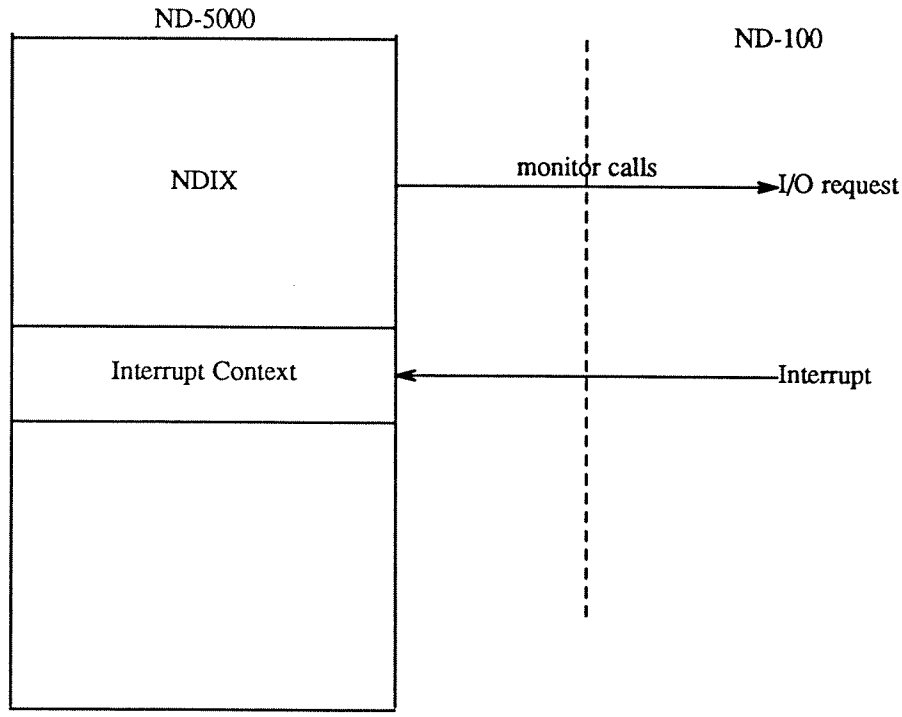


Figure 1. The ND-5000/ND-100 Interface.

2. Memory Configuration & Bootstrapping

When the ND-5000 system is bootstrapped into operation and a NDIX kernel is loaded into the ND-5000 memory the physical layout of the system data structures must be in a well defined state and they must be mapped onto the logical address space of the NDIX kernel at fixed addresses. This section defines the layout of physical memory which will be expected by NDIX, the initial logical address map of the NDIX kernel and the initial contents of the processor registers when the NDIX kernel is entered.

2.1. Physical Memory Configuration

The host computer for the NDIX implementation consists of an ND-5000 and an ND-100 processor sharing a common physical memory array of up to 32Mb. The hardware will be configured such that the bottom of physical memory is private to the ND-100 and is used by SINTRAN-III. All memory at ND-100 physical addresses above a certain physical address (PRIVATE) will be used by the ND-5000 and will map onto ND-5000 physical addresses starting at 0. Normally this private memory will be two megabytes, except in the case of the ND120/CX where all memory on the ND120 card will be private.

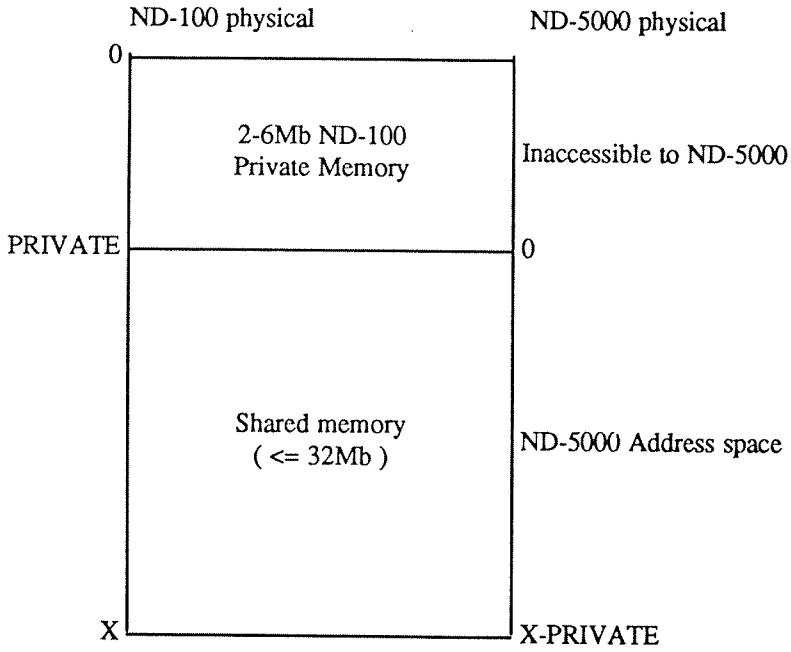


Figure 2.1 Physical Memory Allocation

The actual extent of the physical memory accessible to the ND-5000 will be passed to the NDIX kernel via the *feinit* monitor call when it begins execution; this monitor call is described in appendix 3.

Except when bootstrapping the ND-5000, debugging or performing specific functions, described below, which involve access to shared data structures, the ND-100 processor will *never* change the contents of the ND-5000 physical address space allocated to NDIX. Futhermore, all addresses passed to the ND-100 after an *feinit* call has been successfully executed will be ND-100 physical addresses. It is the responsibility of the ND-5000 program to relocate ND-5000 physical addresses accordingly before they are passed.

2.2. Physical Layout of Data Structures @ Bootime

When the NDIX kernel is being bootstrapped into the ND-5000 processor the following data structures will be established in the ND-5000 physical memory†.

† This layout is chosen to be as similar to previous versions of NDIX as possible, in an attempt to minimize NDIX kernel and debugger changes.

- A (possibly zero length) dynamically allocatable area of memory for use by either NDIX or other ND-5000 processes.
- The Physical Segment Table. Only the 2nd half of this table is used by NDIX, which should not be used by other ND5000 processes.
- A (possibly zero length) dynamically allocatable area of memory for use by either NDIX or other ND-5000 processes.

There then follows an area of memory dedicated to NDIX, the first address beyond this area being passed to NDIX in the *feinit* monitor call. If this area is not large enough to establish NDIX, then NDIX will terminate with an FE_EXIT monitor call, indicating how much contiguous memory is required.

The layout of this area is:

- The NDIX kernel data segment index page.
- The NDIX kernel text segment index page.
- The NDIX kernel stack segment index page.
- The PST index page.
- The NDIX Process Segment index page.
- The NDIX kernel data segment.
- Process Segment for the 256 domains, comprising the NDIX process.
- The NDIX kernel stack segment. The size of this segment is fixed at 8Kb.
- The NDIX kernel text segment.
- An area of memory to be allocated dynamically by the NDIX kernel, and mapped (dynamically) into several segments.

There is no easy formula for sizing this contiguous area, since its size is dependant in part on the size of the NDIX swap partitions. Therefore the size will be provided for the ND-100 in a file.‡

There then follows an area of memory which may be dynamically allocated to either NDIX or other ND-5000 processes.

There then follows an area of memory dedicated to NDIX, but whose use is shared between NDIX and the ND-100. The start address of this area is passed to NDIX in the *feinit* monitor call. This area consists of:

- The NDIX shared data segment index page.
- The NDIX shared data segment. The size of this segment is currently 32k.

There then follows an area of memory which may be dynamically allocated to either NDIX or other ND-5000 processes.

The layout of the ND5000 memory is illustrated in figure 2.2.

2.3. Loading the NDIX program

Modifications must be made to the ND-5000 monitor to set up the necessary data structures, specified in the previous section, in physical memory and to load a specified pair of SINTRAN-III PSEG & DSEG files containing an absolute NDIX program image into the ND-5000 memory.

‡ see § 8

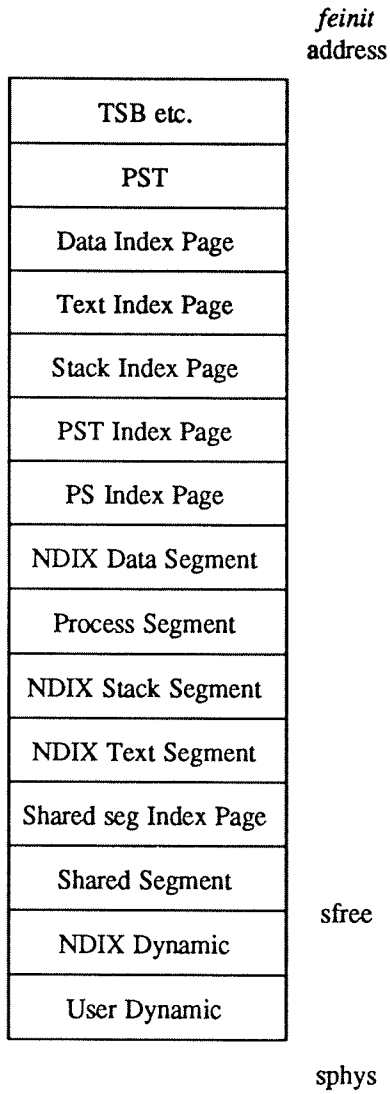


Figure 2.2 ND-5000 Physical Memory Layout

2.3.1. Initial PST entries

The following PST entries need setting to point to the relevant page tables (all segments loaded by the ND5000 monitor are small enough to be mapped by a single level of page tables).

PST INDEX	Segment
0†	Pageframe number of PST
4096	NDIX text
4097	NDIX data
4098	NDIX stack
4099	NDIX PST
4100	NDIX PS
4105	Shared segment

All other NDIX PST entries should be set to zero.

2.3.2. Initial Index Page Settings

- The NDIX Stack index page maps only 8Kb (4 physical pages).
- The NDIX Shared segment index page maps only 32Kb.

In all index pages the entries for pages beyond the allocated size of the segment should be zero.

For details of the sizes of the other segments see the next section.

2.3.3. Initial Segment Loading

Each segment should be loaded starting on a page boundary.

- The NDIX Text and Data segments are loaded from a pair of Sintran PSEG & DSEG files. The size of these segments is found from the Sintran files.
- The NDIX Stack segment is four pages long, all initialized to zero.
- The NDIX part of the PST is initialized as described above.
- In the Process Segment all domain information table entries for domains 1-255 should be initialised, by the ND-5000 monitor to zero.

  In domain zero, which describes the NDIX program itself, the value of all unused capability entries should be set to zero, so making them invalid; the domain call information, trap handling information and domain characteristics fields should also all be initialised to zero. The only field which should be set is byte 0xc8 (0310), the domain status byte. Bit zero of this byte should be set, indicating that privileged instructions are allowed in the NDIX domain.

  The capabilities used and their settings are‡:

  data

segment	value	meaning
0	0x9001	data segment, write permitted
1	0x9000	text segment, write permitted
2-5	0	invalid
6	0x3009	shared segment, write permitted, no-cache
7-26	0	invalid
27	0x9003	PST, write permitted
28	0x9004	Process Segment, write permitted
29	0x9002	stack (upage), write permitted
30-31	0	invalid

  instruction

segment	value	meaning
0	0x1000	text segment
1-30	0	invalid
31	0xc000	other machine, indirect

- Memory following the NDIX text segment does not need to be initialized, or indexed, but zeroing of this memory† may be considered advisable for secure systems.
- The first 2Kb of the shared segment is a bitmap covering physical memory allocation, one bit per page. The most significant bit in the first word corresponds to the first page in memory, the least significant in the last word to the last page in 32Mb of memory. A bit is set for each page of memory which may be dynamically allocated between NDIX and other ND5000 processes and is

† This entry is needed by the NDIX superstructure and is not usable as a valid PST entry.

‡ This currently does not match reality and will require updating

† The same consideration applies to any other memory transferred between NDIX and Sintran.

available for NDIX at startup.

If this bitmap does not indicate enough memory for NDIX user processes then NDIX will terminate with an *feexit* monitor call, indicating how much allocatable memory is required.

The remainder of this segment is used for the Xmsg command and response buffers (see § 9) the IPL record, (see § 4.2.1), the clock record (see § 5), and the sub-device descriptors (see § 4). This is depicted in figure 2.3.

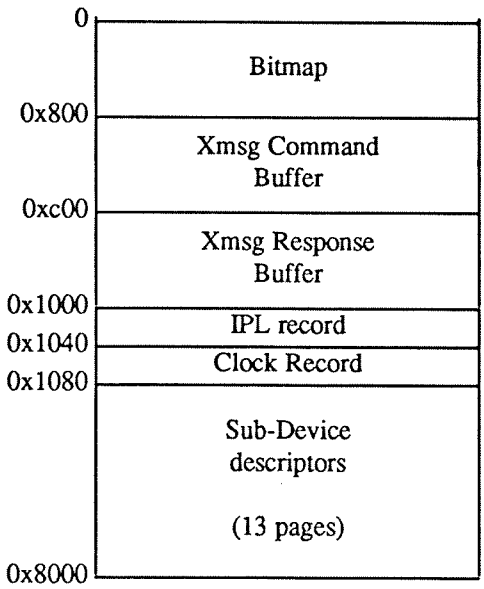


Figure 2.3 Shared segment layout

2.4. NDIX Kernel Logical Address Space

The NDIX kernel text and data domains (domain 0 of the NDIX process) must be partially established before the kernel can start running. This section outlines the initial environment, and provides the information used above to describe the initialization of the PST and PS for NDIX.

These mappings are changed by NDIX almost immediately it starts running. Do not assume they ever apply after booting, even if NDIX requests an immediate reboot. It is also unsafe to assume the contents of any of the segments survive unchanged after NDIX has started running.

2.4.1. The NDIX Text Domain

The NDIX kernel text domain logical address space is shown in figure 2.3.

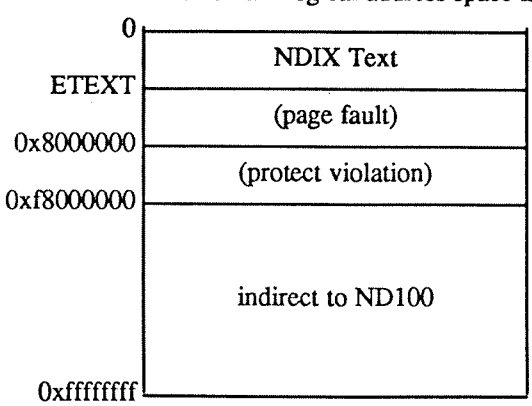


Figure 2.3 The NDIX Logical Text Domain

It is this logical to physical mapping that should established by the ND-5000 monitor/bootstrap, after it has loaded the NDIX program into physical memory in the manner described above.

In figure 2.3 the symbolic constant ETEXT is equated with the last valid address in the NDIX kernel text segment (rounded up to the next page boundary).

Accessing logical addresses between ETEXT and 0x7ffffff (the start of segment 1), will cause page faults.

Accessing logical addresses between 0x8000000 and 0xf7ffffff will cause a protection violation.

Specifically the logical segments of the NDIX text domain are used as follows:

logical segment 0

Mapped, onto the text segment of the NDIX program.

logical segments 1-30

Marked invalid in the program capability. Accessing these segments will cause a protection violation fault.

logical segment 31

Marked *indirect* and mapped onto the ND-100 domain (i.e. the *other machine* bit is set in the capability). This segment, which starts at kernel logical address 0xf8000000, is used to make monitor calls to the ND-100 (see below). An attempt to use an invalid monitor call number will raise an *instruction sequence error* trap.

2.4.2. The NDIX Data Domain

The NDIX kernel data domain logical address space is shown in figure 2.4. It is this logical to physical mapping that should established by the ND-5000 monitor/bootstrap, after it has loaded the NDIX program into physical memory in the manner described above.

In figure 2.4 the symbolic constant EDATA is equated with the last valid address in the NDIX kernel data segment (rounded up to the next page boundary).

Accessing logical addresses between EDATA and 0x7ffffff (the start of segment 1), will cause page faults.

Accessing logical addresses between (ETEXT + 0x800000) and 0xffffffff (the start of segment 2), will cause page faults.

Accessing logical addresses between 0x1000000 and 0x2fffffff will cause a protection violation.

Accessing logical addresses between 0x30008000 and 0x37ffffff (the start of segment 7), will cause page faults.

Accessing logical addresses between 0x3800000 and 0xd7ffffff will cause a protection violation.

Specifically the logical segments of the NDIX data domain are used as follows:

logical segment 0

Mapped, onto the data segment of the NDIX program.

logical segment 1

Mapped, onto the text segment of the NDIX program. This allows the NDIX program to change the contents of its text segment if required.

logical segments 2-5

Marked invalid in the data capability. Accessing these segments will cause a protection violation fault.

logical segment 6

Mapped to the physical pages containing the No-Cache segment. The segment starts at logical address 0x30000000 and ends at logical address 0x30007fff. Accessing logical addresses in the range 0x30008000 to 0x37ffffff will cause a page fault.

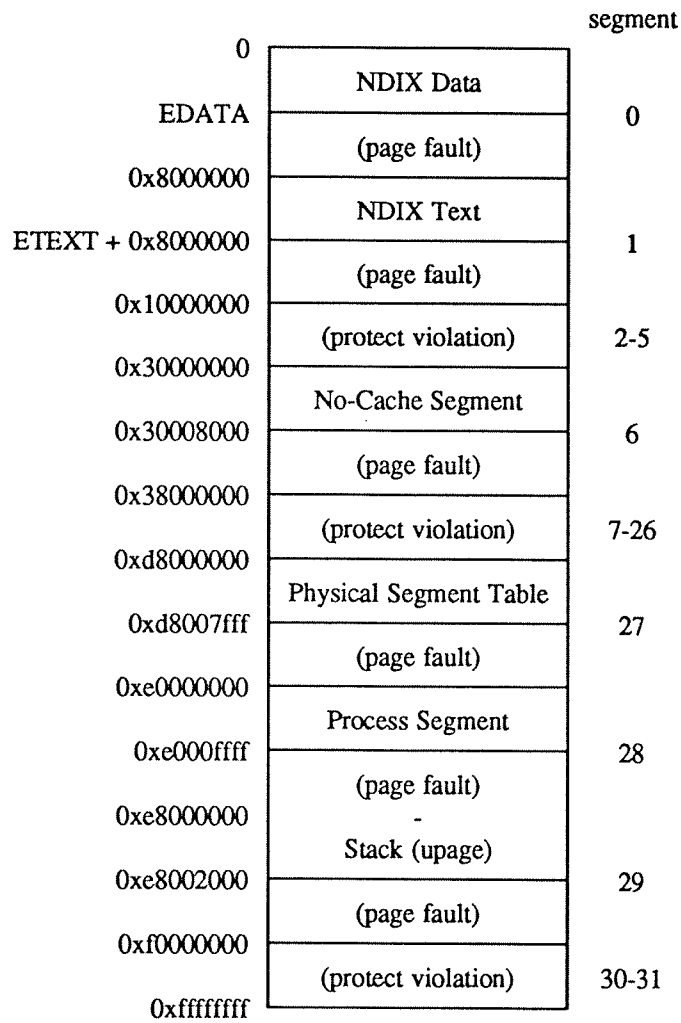


Figure 2.4 The NDIX Logical Data Domain

logical segments 7-26

Marked invalid in the data capability. Accessing these segments will cause a protection violation fault.

logical segment 27

Mapped to the physical pages containing the Physical Segment Table. The PST starts at logical address 0xd8000000 and ends at logical address 0xd8007fff. Accessing logical addresses in the range 0xd8008000 to 0xdfffffff will cause a page fault.

logical segment 28

Mapped to the physical pages containing the Process Segment of the NDIX process. The PS starts at logical address 0xe0000000 and ends at logical address 0xe000ffff. Accessing logical addresses in the range 0xe0010000 to 0xe7ffffff will cause a page fault.

logical segment 29

Mapped to the four stack pages. The logical address of the start of the stack segment is 0xe8000000. Accessing logical addresses in the range 0xe8002000 to 0xefffffff will cause a page fault.

logical segments 30 and 31

Marked invalid in their data capabilities. Accessing these segments will cause a protection violation.

2.5. Initial Register Contents

Once the NDIX program has been loaded into memory, and the memory management has been initialised we are ready to start execution of the NDIX kernel. Execution is commenced by loading the context block of the NDIX process with appropriate values and then starting the ND-5000 at location 4. In general the requirements of the NDIX kernel are similar to those of the SINTRAN-III swapper, however certain registers should be initialised as follows:

Register	Value
CED	0
CAD	0
PS	NDIX Process Segment

All other registers should be set in the same manner as is currently used for loading the SINTRAN-III swapper. The stack will subsequently be initialised by the NDIX kernel via an *init* instruction. The kernel is started at location 4 to avoid the possibility of an address zero trap when the kernel is executed.

3. Inter-Processor Monitor Calls

The basic structure of the ND-100/NDIX interface provides for an interaction based on monitor calls with data passed to and from NDIX via shared memory regions. The ND-100 has the power to stop, start and interrogate the ND-500(0) processor.

Programs running in the ND-500(0) can make monitor calls to other domains in the same machine, or to the ND-100, via CALL or CALLG instructions with a subroutine entrypoint which lies in an indirect segment. Within the NDIX environment system calls from user applications will be implemented via inter-domain monitor calls. The NDIX kernel itself uses monitor calls to communicate with the ND-100 front-end processor. In the case of monitor calls from NDIX to the ND-100 the capability of the indirect segment has bit 15 (the other machine bit) set to one.

Under NDIX the last segment of the kernel text domain (segment 31) will be used for making inter-processor monitor calls. The result of this mapping is that kernel virtual addresses in the range 0xF8000000 to 0xF9FFFFFF will map onto the ND-100 monitor calls 0 to 0x1FFFFFF.

All addresses passed between the ND-100 and NDIX, except some during the *feinit* monitor call, will be ND-100 physical addresses.

3.1. Making a Monitor Call

In order to make a monitor call to the ND-100 the NDIX kernel uses a *callg* instruction. This instruction takes as parameters: the address of the called routine, in this case an address in the indirect segment; a count of the number of arguments to be passed; and a series of operands specifying the addresses of the monitor call parameters. Monitor call parameters are always passed by address: it is not possible to call by value, though the contents of the parametric locations are actually available directly to the ND-100. As an example imagine that we wish to make a monitor call number 0x180 to the ND-100 with four parameters, then the following code should be executed:

```
callg 0xF8000180, 4, p1, p2, p3, p4
```

The reader should note that the low 21 bits of the address used are actually set to the number of the monitor call, not to the apparent address, in the called domain, of the handling routine vector. When the above instruction is executed the ND-100 is passed a packet of data which contains all the information included by the call instruction, plus an indication of the monitor call number (e.g. 0x180)†.

One of the parameters to this call will indicate whether it is to be executed synchronously or asynchronously. If the call is synchronous the NDIX kernel will stop and will only be restarted once the ND-100 has serviced the monitor call. Otherwise the NDIX kernel will continue processing and the response will cause an asynchronous "interrupt"‡.

In the NDIX environment a single monitor call is defined to request the ND-100 to perform I/O, and for communication between NDIX and SINTRAN. This call may have numerous different parameters. The basic format of the monitor call and associated parameters is defined in this section.

3.2. NDIX Monitor Call Numbers

Since all NDIX monitor calls are parameterised there need only be one ND-100 monitor call number, as passed in the data packet, reserved for communication between NDIX and SINTRAN-III; this monitor call will be number 0x180 (0600) and will be used exclusively for NDIX/SINTRAN-III communications. The exact nature of the monitor request being made from NDIX will be defined by the command field, passed as the second parameter in the data packet sent to the ND-100 (see figure 3.1).

3.3. NDIX Monitor Call Format

All NDIX monitor requests have the same format, which may be expressed in C as follows:

---

† The layout of individual monitor call parameters is described in appendix 3.

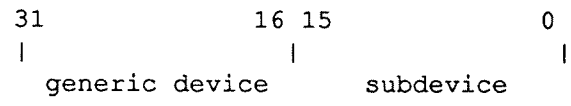
‡ See § 4

```
fecall(device, request, response, command)
long device, request;
char *response, *command;
```

This C call will be converted into a call instruction with four *call by address* parameters by a low level routine within the NDIX environment. All the addresses passed to the ND-100 will be 32 bit ND-100 physical addresses†, this means that they must be converted from ND-500(0) addresses. Also, all the addresses passed to the ND-100, and all the addresses stored in those locations which are used by SINTRAN-III must be aligned on an even byte address boundary; this is because the ND-100 is a word addressed machine. The detailed meaning of the parameters is as follows:

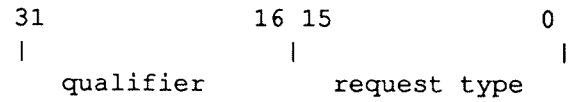
device

This is a 32 bit device code used by the ND-100 to determine with which I/O device, if any the request is associated. The top 16 bits indicate the number of the generic device type (disk, tape, terminal-in, terminal-out, clock, xmsg, message-device). The lower 16 bits indicate the subdevice number:



request

This is a 32 bit code specifying the type of request that is being made; it might specify a read request, or be a request to perform a control function. The lower 16 bits are used to indicate the request type, and the upper to indicate further options or qualifiers associated with the request. The most important qualifiers indicate whether a command is to be executed synchronously or asynchronously. The only other one currently used indicates a re-open on a currently open device.



response

This is a 32 bit word specifying the ND-100 physical address† of the response packet associated with the monitor request. The ND-100 fills in the response packet prior to restarting (for synchronous commands) or interrupting NDIX (asynchronous commands). See appendix 3 for details of specific response packet formats.

command

This is a 32 bit word specifying the ND-100 physical address† of a data packet which contains the parameters for this monitor call (see appendix 3 for details). For example a disk read request has a data field which contains the physical address where DMA is to begin, a count of the number of data bytes to be transferred and a logical block number, which specifies where on the disk device the transfer is to commence.

---

† Except the *feinit* monitor call which is made to learn the extent of the ND-100's private memory in the first place (see appendix 3).

† During the *feinit* monitor call, which is used to establish the physical position of the UNIX program, both the response and command packet addresses are passed as ND-500(0) logical data segment addresses. The ND-100 can determine their physical wherabouts because the data segment occupies contiguous pages starting at a known physical address (SDATA).

An Example

As an example, consider an asynchronous block device read request. The monitor call takes four parameters: the device number associated with the device on which we wish to perform the read; a request code, specifying an asynchronous read; the address of a response packet to be used by the ND-100 to return the completion status of the command; and the address of a packet containing the parameters to the read call. The monitor request is made in such a way that the parameters to the monitor call (two longwords and two addresses) all end up in the value fields of the monitor call packet delivered to the ND-100. The call may be expressed in C as follows:

```
struct read_rpk res_p;   /* response packet */
struct read_cpk com_p;  /* command packet */

fecall(DEVICE, FE_READ, &res_p, &com_p);
```

The fecall assembler routine will then be:

```
_fecall:
        ents $36

        dton b.28
        dton b.32

        callg 0xF8000180, $4, b.20, b.24, b.28, b.32

        ret
```

The dton instruction converts ND-500(0) addresses into ND-100 physical addresses. When the monitor call is made ND-100 will be passed a data packet containing the parameters of the call. If the monitor call is synchronous NDIX will stop execution while the request is processed, if it is asynchronous execution will continue uninterrupted. In the case of our example above, the packet will have the format illustrated in figure 3.1. For details of this packet and other possible formats, refer to appendix 3.

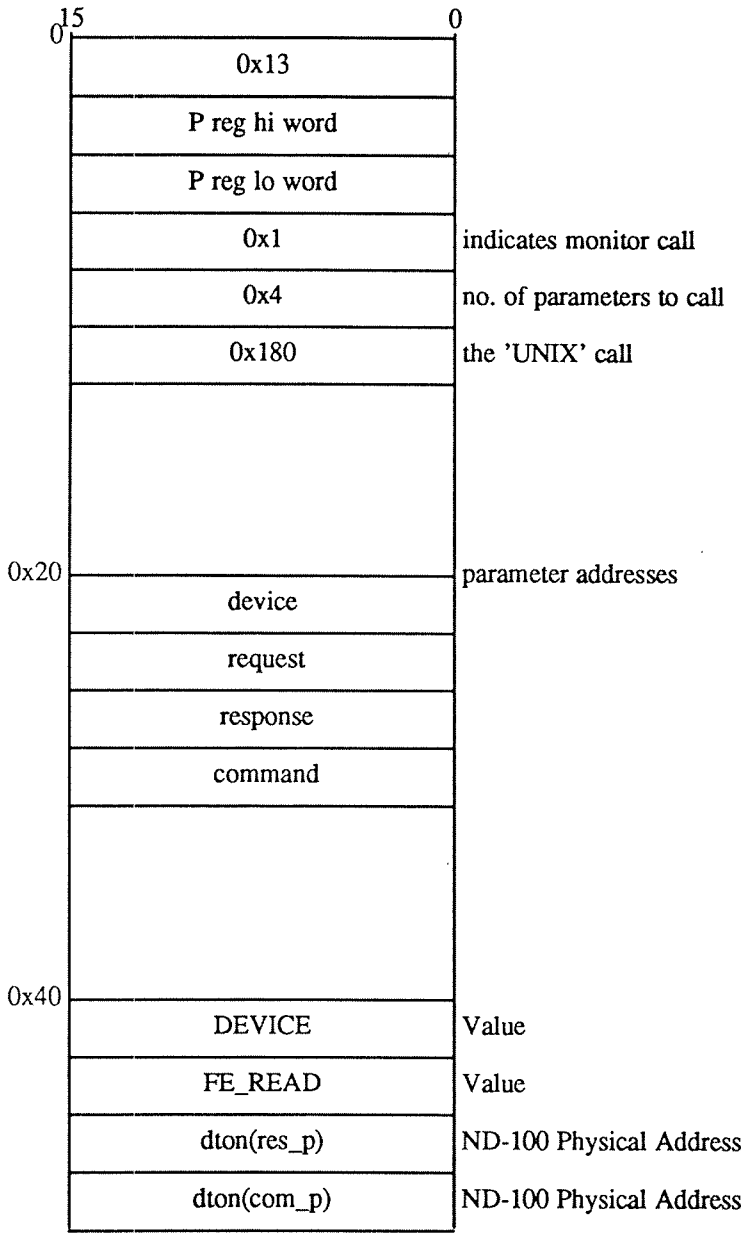


Figure 3.1 ND-100 Memory (16 bit words) following an NDIX monitor call.

3.3.1. Request Types

The nature of the actual monitor request, being made to the ND-100 is defined by the lower sixteen bits of the request parameter (parameter 2). The following are now defined:

value	mnemonic	meaning
0x1	FE_INIT	system initialisation
0x2	FE_IDEV	initialise a generic device
0x3	FE_OPEN	open a subdevice
0x4	FE_CLOS	close a subdevice
0x5	FE_READ	read a sub-device
0x6	FE_RCON	synchronous read from system console
0x7	FE_WRIT	write a sub-device
0x8	FE_WCON	synchronous write to system console

0x9	FE_DCTL	device control
0xA		unused
0xB	FE_EXIT	system shutdown
0xC		unused
0xD		unused
0xE	FE_ERRM	send error code to ND-100.

3.3.2. Request Qualifiers

Monitor requests can be divided into two categories: synchronous and asynchronous. Synchronous requests are generally those which take a short time to service or have no response associated with them; they either return to the caller only when they have completed or they never return at all. Asynchronous requests are associated with an interrupt context (see below) ; they return immediately to the caller, but the request may not be completed until sometime later. When an asynchronous request does complete an interrupt will be generated by the ND-100 into a context defined during initialisation†. Some monitor requests have both synchronous and asynchronous forms.

The above requests may be modified by qualifiers in the upper 16 bits of the request parameter, as follows:

value	mnemonic	meaning
0x0	QF_ASYNC	asynchronous
0x1	QF_SYNC	synchronous
0x2	QF_RE	applies on an open call only and indicates that a re-open is required. (i.e. allowed on already open terminal lines)

All the above monitor requests are defined in detail in appendix 3.

† During the *feinit* monitor call.

4. Inter-Processor Monitor Calls

The basic structure of the ND-100/NDIX interface provides for an interaction based on monitor calls with data passed to and from NDIX via shared memory regions. The ND-100 has the power to stop, start and interrogate the ND-500(0) processor.

Programs running in the ND-500(0) can make monitor calls to other domains in the same machine, or to the ND-100, via CALL or CALLG instructions with a subroutine entrypoint which lies in an indirect segment. Within the NDIX environment system calls from user applications will be implemented via inter-domain monitor calls. The NDIX kernel itself uses monitor calls to communicate with the ND-100 front-end processor. In the case of monitor calls from NDIX to the ND-100 the capability of the indirect segment has bit 15 (the other machine bit) set to one.

Under NDIX the last segment of the kernel text domain (segment 31) will be used for making inter-processor monitor calls. The result of this mapping is that kernel virtual addresses in the range 0xF8000000 to 0xF9FFFFFF will map onto the ND-100 monitor calls 0 to 0x1FFFFFF.

All addresses passed between the ND-100 and NDIX, except some during the *feinit* monitor call, will be ND-100 physical addresses.

4.1. Making a Monitor Call

In order to make a monitor call to the ND-100 the NDIX kernel uses a *callg* instruction. This instruction takes as parameters: the address of the called routine, in this case an address in the indirect segment; a count of the number of arguments to be passed; and a series of operands specifying the addresses of the monitor call parameters. Monitor call parameters are always passed by address: it is not possible to call by value, though the contents of the parametric locations are actually available directly to the ND-100. As an example imagine that we wish to make a monitor call number 0x180 to the ND-100 with four parameters, then the following code should be executed:

```
callg 0xF8000180, 4, p1, p2, p3, p4
```

The reader should note that the low 21 bits of the address used are actually set to the number of the monitor call, not to the apparent address, in the called domain, of the handling routine vector. When the above instruction is executed the ND-100 is passed a packet of data which contains all the information included by the call instruction, plus an indication of the monitor call number (e.g. 0x180)†.

One of the parameters to this call will indicate whether it is to be executed synchronously or asynchronously. If the call is synchronous the NDIX kernel will stop and will only be restarted once the ND-100 has serviced the monitor call. Otherwise the NDIX kernel will continue processing and the response will cause an asynchronous "interrupt"‡.

In the NDIX environment a single monitor call is defined to request the ND-100 to perform I/O, and for communication between NDIX and SINTRAN. This call may have numerous different parameters. The basic format of the monitor call and associated parameters is defined in this section.

4.2. NDIX Monitor Call Numbers

Since all NDIX monitor calls are parameterised there need only be one ND-100 monitor call number, as passed in the data packet, reserved for communication between NDIX and SINTRAN-III; this monitor call will be number 0x180 (0600) and will be used exclusively for NDIX/SINTRAN-III communications. The exact nature of the monitor request being made from NDIX will be defined by the command field, passed as the second parameter in the data packet sent to the ND-100 (see figure 3.1).

4.3. NDIX Monitor Call Format

All NDIX monitor requests have the same format, which may be expressed in C as follows:

---

† The layout of individual monitor call parameters is described in appendix 3.

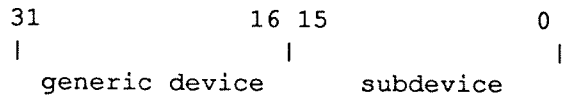
‡ See § 4

```
fecall(device, request, response, command)
long device, request;
char *response, *command;
```

This C call will be converted into a call instruction with four *call by address* parameters by a low level routine within the NDIX environment. All the addresses passed to the ND-100 will be 32 bit ND-100 physical addresses†, this means that they must be converted from ND-500(0) addresses. Also, all the addresses passed to the ND-100, and all the addresses stored in those locations which are used by SINTRAN-III must be aligned on an even byte address boundary; this is because the ND-100 is a word addressed machine. The detailed meaning of the parameters is as follows:

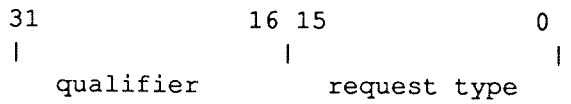
device

This is a 32 bit device code used by the ND-100 to determine with which I/O device, if any the request is associated. The top 16 bits indicate the number of the generic device type (disk, tape, terminal-in, terminal-out, clock, xmsg, message-device). The lower 16 bits indicate the subdevice number:



request

This is a 32 bit code specifying the type of request that is being made; it might specify a read request, or be a request to perform a control function. The lower 16 bits are used to indicate the request type, and the upper to indicate further options or qualifiers associated with the request. The most important qualifiers indicate whether a command is to be executed synchronously or asynchronously. The only other one currently used indicates a re-open on a currently open device.



response

This is a 32 bit word specifying the ND-100 physical address† of the response packet associated with the monitor request. The ND-100 fills in the response packet prior to restarting (for synchronous commands) or interrupting NDIX (asynchronous commands). See appendix 3 for details of specific response packet formats.

command

This is a 32 bit word specifying the ND-100 physical address† of a data packet which contains the parameters for this monitor call (see appendix 3 for details). For example a disk read request has a data field which contains the physical address where DMA is to begin, a count of the number of data bytes to be transferred and a logical block number, which specifies where on the disk device the transfer is to commence.

---

† Except the *feinit* monitor call which is made to learn the extent of the ND-100's private memory in the first place (see appendix 3).

† During the *feinit* monitor call, which is used to establish the physical position of the UNIX program, both the response and command packet addresses are passed as ND-500(0) logical data segment addresses. The ND-100 can determine their physical wherabouts because the data segment occupies contiguous pages starting at a known physical address (SDATA).

An Example

As an example, consider an asynchronous block device read request. The monitor call takes four parameters: the device number associated with the device on which we wish to perform the read; a request code, specifying an asynchronous read; the address of a response packet to be used by the ND-100 to return the completion status of the command; and the address of a packet containing the parameters to the read call. The monitor request is made in such a way that the parameters to the monitor call (two longwords and two addresses) all end up in the value fields of the monitor call packet delivered to the ND-100. The call may be expressed in C as follows:

```
struct read_rpk res_p;   /* response packet */
struct read_cpk com_p;  /* command packet */

fecall(DEVICE, FE_READ, &res_p, &com_p);
```

The fecall assembler routine will then be:

```
_fecall:
        ents $36

        dton b.28
        dton b.32

        callg 0xF8000180, $4, b.20, b.24, b.28, b.32

        ret
```

The dton instruction converts ND-500(0) addresses into ND-100 physical addresses. When the monitor call is made ND-100 will be passed a data packet containing the parameters of the call. If the monitor call is synchronous NDIX will stop execution while the request is processed, if it is asynchronous execution will continue uninterrupted. In the case of our example above, the packet will have the format illustrated in figure 3.1. For details of this packet and other possible formats, refer to appendix 3.
boxwid = 2.0 a = 0.3; b = 1.0 move right 1.0 down B1:box height a "0x13" box height a "P reg hi word" box height a "P reg lo word" B2:box height a "0x1" B3:box height a "0x4" B4:box height a "0x180" box height b B5: box height a "device" box height a " request" box height a " response" box height a " command" box height b B6: box height a "DEVICE" B7: box height a "FE_READ" B8: box height a "dton(res_p)" B9: box height a "dton(com_p)" "0 " at B1.nw rjust "15" at B1.nw above "0" at B1.ne above " indicates monitor call" at B2.e ljust " no. of parameters to call" at B3.e ljust " the 'UNIX' call" at B4.e ljust " parameter addresses" at B5.ne ljust "0x20 " at B5.nw rjust "0x40 " at B6.nw rjust " Value" at B6.e ljust " Value" at B7.e ljust " ND-100 Physical Address" at B8.e ljust " ND-100 Physical Address" at B9.e ljust Figure 3.1 ND-100 Memory (16 bit words) following an NDIX monitor call.

4.3.1. Request Types

The nature of the actual monitor request, being made to the ND-100 is defined by the lower sixteen bits of the request parameter (parameter 2). The following are now defined:

value	mnemonic	meaning
0x1	FE_INIT	system initialisation
0x2	FE_IDEV	initialise a generic device
0x3	FE_OPEN	open a subdevice
0x4	FE_CLOS	close a subdevice
0x5	FE_READ	read a sub-device
0x6	FE_RCON	synchronous read from system console
0x7	FE_WRIT	write a sub-device
0x8	FE_WCON	synchronous write to system console
0x9	FE_DCTL	device control
0xA		unused
0xB	FE_EXIT	system shutdown
0xC		unused
0xD		unused
0xE	FE_ERRM	send error code to ND-100.

4.3.2. Request Qualifiers

Monitor requests can be divided into two categories: synchronous and asynchronous. Synchronous requests are generally those which take a short time to service or have no response associated with them; they either return to the caller only when they have completed or they never return at all. Asynchronous requests are associated with an interrupt context (see below) ; they return immediately to the caller, but the request may not be completed until sometime later. When an asynchronous request does complete an interrupt will be generated by the ND-100 into a context defined during initialisation†. Some monitor requests have both synchronous and asynchronous forms.

The above requests may be modified by qualifiers in the upper 16 bits of the request parameter, as follows:

value	mnemonic	meaning
0x0	QF_ASYNC	asynchronous
0x1	QF_SYNC	synchronous
0x2	QF_RE	applies on an open call only and indicates that a re-open is required. (i.e. allowed on already open terminal lines)

All the above monitor requests are defined in detail in appendix 3.

† During the *feinit* monitor call.

5. Device Driving

Since the ND-500(0) has no I/O bus, and all the computer's peripherals are attached to the ND-100 and driven by its SINTRAN-III system; it is necessary for the NDIX system, running in the ND-500(0), to use the ND-100 as an I/O processor. All I/O requests made by NDIX drivers will be translated into monitor calls to the ND-100, using the format described above. This section discusses the special issues, relating to device driving, which arise out of the requirements of the NDIX device interface.

As explained in the introduction to this document, the NDIX kernel runs entirely in the ND-500(0) processor and makes use of the ND-100 front-end, running SINTRAN-III, as an intelligent I/O processor. Since the UNIX I/O system is designed around the assumption that it is driving hardware controllers, and that these controllers generate interrupts on a number of interrupt levels; NDIX, and the NDIX/ND-100 interface in particular needs to provide a simulation of this arrangement. Specifically this must include operation of NDIX at a number of interrupt priority levels (IPLs).

5.1. I/O requests

5.1.1. Asynchronous Requests

Most I/O requests (i.e. reads & writes), associated with devices attached to the ND-100 are designed to complete asynchronously. When the NDIX system makes a monitor call to start I/O the request will be accepted by the ND-100 without NDIX processing being interrupted. This is equivalent in functional terms to the NDIX driver starting an I/O request by setting an I/O controller's command register.

When the ND-100 eventually completes an I/O request a "sub-device descriptor" will be queued for NDIX in shared memory. If the IPL level set is lower than that associated with the interrupt the ND-100 will stop NDIX, wherever it happens to be executing, save its context in a specific location in ND-500(0) physical memory associated with the current IPL level†, and then restart it in a context which is dedicated to handling completion of I/O. NDIX will then be responsible for determining which I/O request has completed from information provided in a shared memory segment, and taking the relevant action (i.e calling the correct interrupt handling sub-routine). When the interrupt has been dealt with NDIX will check the interrupt queue for any further "interrupts" on the current IPL level or any level above that in which it was previously executing and will process those before reloading the saved context.

In summary, the sequence of events which will take place when a NDIX device driver makes an asynchronous I/O request are as follows:

- The NDIX kernel makes a monitor request, with the command qualifier bit zero clear, and continues processing.
- The data packet, associated with the monitor request is passed to the ND-500(0) driver, running in the ND-100.
- The ND-100 queues the I/O request inside SINTRAN-III.
- Time passes and eventually the ND-100 driver completes the requested I/O.
- The response packet for the completed command is updated by the ND-100.
- A "sub-device descriptor" is queued in shared memory.
- If the IPL level of this interrupt is higher than that currently set by NDIX then NDIX is stopped, and its context saved in an area of ND-500(0) physical memory, associated with its current IPL level. ND-500(0) memory, used in this way is called a context block (CXB).
- NDIX is restarted in the context specified for handling interrupts. At this point the ND-100 changes its idea of the interrupt priority level of NDIX to a new value associated with the currently interrupting device.

  If DMA is performed during the I/O request (i.e. is *fread* or *fwrite*), and the data buffers used for read/write transfers are not on a no-cache segment then a Dump_Dirty_Cache instruction must be performed by NDIX before the DMA transfer, and a Clear_Cache instruction after a read.

† For the layout of the context block see appendix 2.

- NDIX handles the interrupt and then processes all other queued interrupts with an IPL above that in which it was previously executing. These are queued by the ND100 in time within IPL level order. NDIX then reloads the saved context.

The above sequence is represented graphically in figure 4.1.

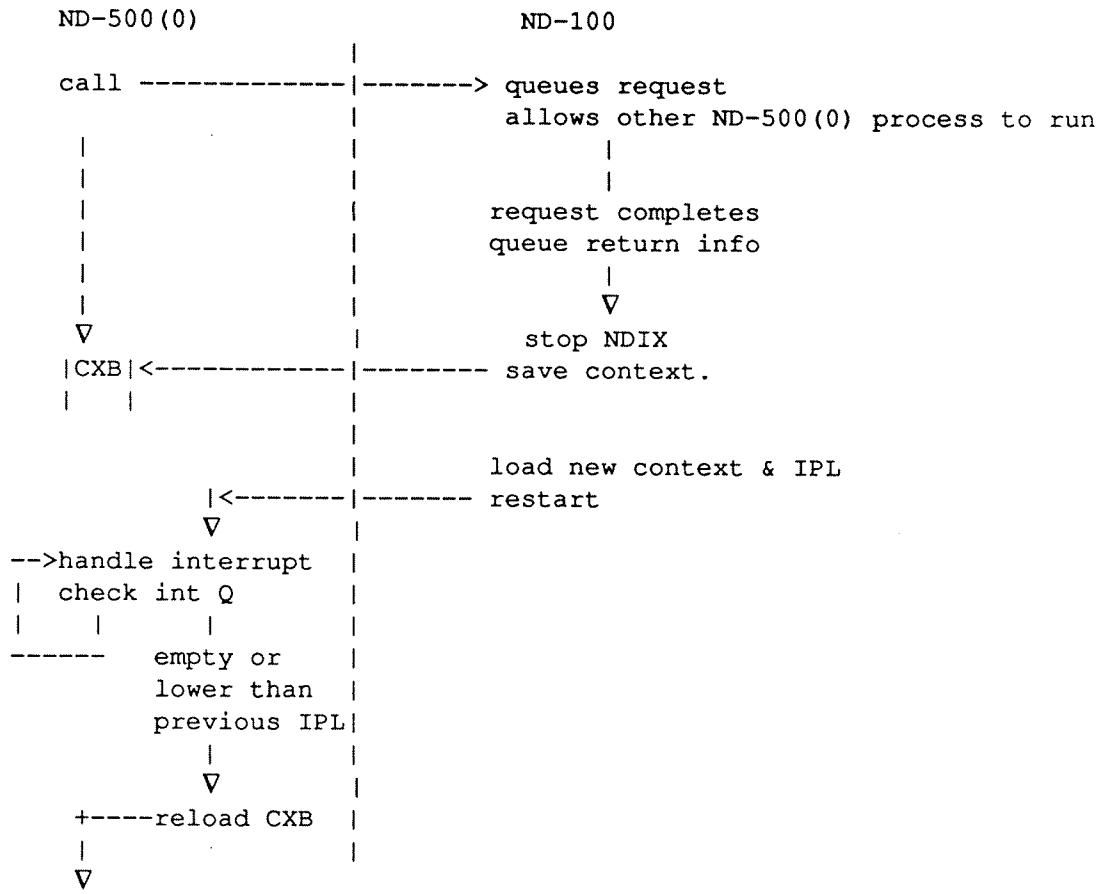


Figure 4.1 Asynchronous Device Driving

5.1.2. Synchronous Requests

When opening some devices, performing system initialisation, or when writing error messages to the system console the NDIX kernel waits for all I/O to complete before proceeding with further processing. For this reason the NDIX/ND-100 interface implements synchronous I/O requests, not associated with an interrupt context. When a synchronous request is made, NDIX is stopped until the I/O has been completed, at which point the response packet is filled by the ND-100, in exactly the same way as it would be for an asynchronous request. NDIX is then restarted at the location following the monitor call instruction.

In detail, the sequence of events which take place when an NDIX device driver makes a synchronous I/O request is as follows:

- The NDIX driver makes a monitor request, with the command qualifier bit zero set.
- NDIX stops and the data packet, associated with the monitor request is passed to the ND-100.
- The ND-100 queues the I/O request.
- Time passes and eventually the ND-100 driver completes the requested I/O. During this time no other asynchronous events can complete and NDIX remains stopped.
- The return status of the completed command is written, by the ND-100, into the response packet associated with the now completed request.

- NDIX is restarted at the instruction following the monitor call instruction, which initiated the I/O request.

The above sequence is represented graphically in figure 4.2.

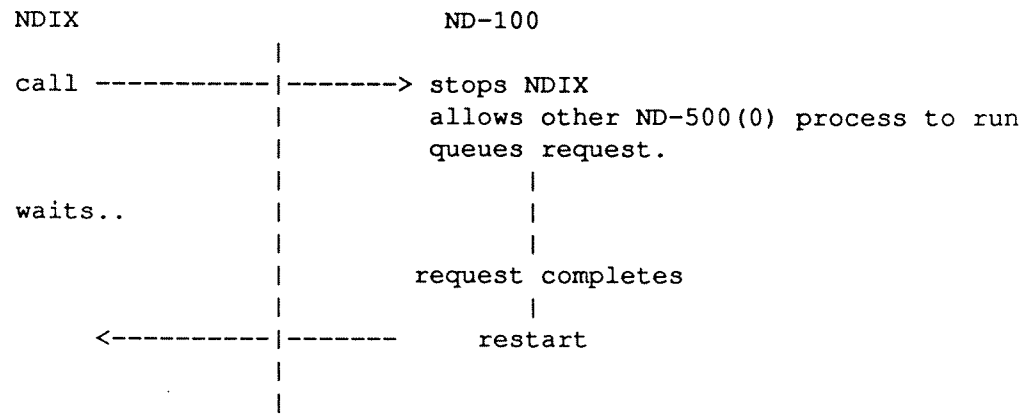


Figure 4.2 Synchronous Device Driving

For more details, see appendix 3.

5.1.3. Command and Response Packet Formats

All valid command packets start with a 16 bit operation code. The layout will only be specified if extra information is included.

Similarly all valid response packets start with a 16 bit completion code. If this is 0 the operation completed successfully. If not then it is an interface error code. For disks and tapes this code may indicate that a hardware error has occured and a hardware status word is always present following this, i.e:

name	offset	size	meaning
completion	0	2	0 command completed ok n May indicate an interface error, or a h/w fault (h/w status given in status)
status	2	2	drive hardware status

5.2. Interrupts

To prevent the interruption of NDIX when manipulating critical data structures and to prevent two interrupts from occuring simultaneously on the same generic device an interrupt priority scheme must be implemented. In order to achieve this the current NDIX interrupt priority level (IPL) is held in shared memory. Each generic device is associated with an IPL level at which it interrupts. This may be between 1-6. IPL level 0 indicates that all interrupts are enabled, 7 indicates that they are all inhibited. During servicing of an interrupt the IPL level is set to that associated with the generic device. An IPL mask is also held in shared memory indicating which interrupt levels are active but currently interrupted by a higher IPL.

Interrupts to NDIX from the ND-100 are vectored via an address passed in the *feinit* call. This call also specifies the addresses of the interrupt context blocks (CXBs). When an *feidev* monitor request (see Appendix 3) is made to initialise a specific generic device the device is associated with an IPL.

The exact sequence of actions which the ND-100 should perform when it interrupts NDIX is as follows:

- The IPL record area in shared memory is locked. This locks both the IPL data and the sub-device descriptor queue.
- The subdevice descriptor is placed on the interrupt queue.

- If the IPL at which NDIX is running is higher than the IPL of the interrupt the IPL record is unlocked and no more action is taken.
- Otherwise NDIX is stopped and its context saved in the context block associated with its currently executing level.
- The NDIX Interrupt Priority Level (IPL) is increased to that associated with the generic device, and the mask is updated if the previous level was non-zero. Saving the IPL allows the NDIX kernel to maintain a stack of IPLs when nested interrupts occur.
- The P register is loaded with the address passed in *feinit*. If the CED when the interrupt occurred was non-zero, the CAD is loaded with this value, otherwise, it is left unchanged. All other program registers should be left unchanged.
- NDIX is restarted in the new (kernel) context. (The IPL record is unlocked by this context).

5.2.1. Structure of the IPL Record

This is located in the same segment as the interrupt descriptors. There are 64 bytes reserved for this record of which 10 are currently used for the following fields:

Byte	Length	Meaning
0	4	Pointer to head of interrupt queue. ND-100 address.(Zero when queue is empty)
4	2	Current IPL
6	2	Bit mask indicating active interrupt levels
8	2	IPL lock, controls access to current IPL.

5.2.2. Structure of the Sub-device Descriptor

Sub-device descriptors are all located on the shared data segment. Each contains the following fields:

Byte	Length	Meaning
0	4	Link to next element in queue. ND-100 address. (Zero if no more queue elements)
4	2	Interrupt Priority Level of interrupting sub-device.
6	2	Address of SINTRAN datafield
8	2	Sintran logical device number/unit number
10	1	NDIX generic device number
12	1	NDIX sub-device (minor device) number
14	2	Flag word
16	2	Function code to NDIX server rt-program
18	4	Address of response packet. ND-100 address.

As an example of interrupt handling consider the following scenario:

- NDIX has requested I/O transfers on disk sub-devices 1 and 4, and also on terminal device 0x22. The IPL of NDIX is zero, and the IPLmask is zero.
- The I/O request on disk sub-device 0x1 completes and an interrupt descriptor is queued for NDIX. NDIX is stopped and its current context saved in context block 0. The IPL of NDIX is raised to 4 (the IPL associated with disks) the IPLmask remains as zero. NDIX is restarted in its interrupt context.
- NDIX accesses the interrupt descriptor, determines that this is a disk interrupt and dispatches the interrupt to the disk driver interrupt routine.
- While NDIX is servicing the interrupt the terminal I/O request completes.

- Using the rule of dispatching interrupts in order of decreasing IPL first and in the the order in which their respective commands were requested within each interrupt level, the interrupt associated with device 0x22 has priority as its associated IPL is 5. The interrupt descriptor is placed at the top of the queue and NDIX is stopped. Its context is saved in context block 4 and the IPL is raised to 5. Bit 4 in the IPL mask is set. NDIX is restarted in its interrupt context.
- NDIX accesses the interrupt descriptor, determines that this is a terminal interrupt and dispatches the interrupt to the terminal driver interrupt routine.
- The I/O request on disk sub-device 0x4 completes and an interrupt descriptor is queued for NDIX. The current IPL level is 5 so NDIX is not interrupted.
- NDIX completes servicing of the terminal interrupt. It checks the IPL level of the next queued interrupt and notes that this is lower than its current level. It checks the IPLmask and notes that the IPL of the saved context is not lower than the next queued interrupt. It clears bit 4 in the IPLmask, sets current IPL to 4, and reloads the context in context block 4.
- NDIX completes servicing the disk interrupt. It checks the IPL level of the next queued interrupt and notes that this is not lower than its current level so it dispatches the waiting disk interrupt.
- Eventually servicing of the second disk interrupt will complete. The interrupt queue is empty and the IPLmask is zero. NDIX set the current IPL to 0 and reloads the context saved in context block 0.

5.3. Device Initialisation and Control

When driving devices it is usually necessary to initialise them before any I/O is attempted and the NDIX system is no exception. Also, it is often required to perform operations, not specifically associated with any I/O. For instance, it may be required to write a file mark onto a magnetic tape, or to change the baud rate of a terminal line. The NDIX interface supports a number of monitor calls to perform these device initialisation and control functions.

Device Initialisation

> Before any I/O can be performed on a generic device an *feidev* call must be made. The main purpose of this call is to associate the device with an IPL level and to determine how many sub-devices are present. Following an *feidev* call each sub-device must be opened with an *feopen* call before any I/O is attempted.

Device Opening

> The *feopen* request is used to perform any sub-device specific initialisation and also to return configuration data, such as disk size to the NDIX kernel. It also mediates device use between NDIX and SINTRAN. Any device specific characteristics that may be altered by NDIX should be saved when the *feopen* is done. Once an *feopen* has been performed the device is used exclusively by NDIX until an *feclos* call is used.

Device Close Down

> When a sub-device is no longer required an *feclos* call will be issued, this reverses the effect of the previous *feopen*, request. After an *feclos* call control of the device passes to SINTRAN, which should reset the device to its status before the *feopen*.

Device Control

> The *fedctl* call can be used to change the configuration of an already *feopen*'d sub-device, or to affect sub-device specific functions, such as writing tape marks, changing a terminal's configuration etc.

For details of how control monitor calls should be used on specific device types, see appendix 3.

5.4. Device Numbers & Types

NDIX and SINTRAN-III device numbers have a different format. Under NDIX each device is associated with two device numbers stored together as upper and lower bytes of a sixteen bit word. The upper byte, or *major device number* is associated with a particular device driver (i.e. a different piece of hardware).

The lower byte, or *minor device number* is used to specify a particular logical device on the specified major device. For example a magnetic tape unit might have major device 2 and the drives be addressed by specifying minor devices 0, 1, 2 etc. Under SINTRAN-III each minor device is associated with a distinct, and essentially random *logical unit number*. For example magnetic tape drives 0, 1 and 2 have logical unit numbers 32, 33 and 21 respectively.

All NDIX drivers make requests, via monitor calls, which specify the device on which I/O is to be performed in the form of a 32 bit longword. The high order 16 bits of the word are called the *generic device* and the low order 16 bits are called the sub-device. The generic device will be used to specify the type of device being addressed: disk, tape, terminal-input, terminal-output, clock, etc; this tells the ND-100 how to interpret the particular monitor request parameters.

On an *feidev* call the ND-100 returns information about the number of sub-devices that are available for a particular generic device type.

For disks and tapes this number will actually define the maximum number of devices present. Each minor device number will correspond to a known physical unit and some sub-devices within the possible range may not be present. This will be indicated by an error code in response to a subsequent *feopen.*

The maximum number of physical tape devices supported will be 16. The maximum number of hard disks will be 12, and floppy disks 4 (Floppy disks will always be minor device number 12-15). It is intended (in a future release) to separate hard and floppy disks generic devices. An *feopen* on any given sub-device may return further information (such as disk type/size, see § 4.4.2).

For terminals two numbers will be returned. The first will be a 32bit bitmap indicating the terminals physically present in banks of 4, starting with the most significant bit. i.e. if the most significant bit is set then terminal minor devices 1,2,3 & 4 are deemed to be present, if the least significant bit is set then minor devices 125,126,127 & 128 are present. Minor device 0 is used as a special logical device and refers to the logical NDIX console (see § 4.4.1). Minor device 129 is the hardware console and is considered to be always present. The second number refers to the total number (max 126) of remote terminals connected to the system.

The device number is used as to index into a table maintained by the ND-100. which tells the ND-100 which SINTRAN-III driver it should use to actually perform the I/O.

Certain device codes may be used when no real device is involved and the sub-device can be used to indicate other, request specific information.

Given below is a table of generic device numbers and the types of devices which are associated with them.

Type	Generic	
reserved	0	for trap handling
Disk	1	
Tape	2	
Terminal-in	3	
Terminal-out	4	
Clock	5	
reserved	6	(for floppy disks at a future date)
Xmsg	7	Communications link
Software Int	8	Used for loopback testing
Mdev	9	Used for SINTRAN-NDIX communication†

The following table summarises the valid combinations of generic devices with monitor requests, see appendix 3 for details. Also shown is the valid combinations of synchronous and asynchronous I/O modes.

	Generic Code									
	None	1	2	3	4	5	7	8	9	A/S
FE_INIT	X									S
FE_IDEV		X	X	X	X	X	X	X	X	S
FE_OPEN		X	X	X	X		X		X	A/S
FE_CLOS		X	X	X	X		X			A/S
FE_READ		X	X	X						A
FE_RCON				X						S
FE_WRIT		X	X		X					A
FE_WCON					X					S
FE_DCTL		X	X	X			X	X	X	A/S
FE_EXIT	X									S
FE_ERRM	Any									S

5.4.1. The NDIX Console

The device used as the NDIX console may vary and the information about which of the terminal minor devices is to be used as the NDIX console is passed in the *feinit* call. Internally NDIX should not support a login on /dev/console as requests on this device will be routed logically by the ND-100 to the specified terminal. This will prevent the system trying to run two getty processes on the same device.

The console terminal may be shared between NDIX and SINTRAN. Whilst a user is logged in to either system output from NDIX will always be allowed. The NDIX *fercon* may also always be used without error. In practise this call is only made once between loading a GENERIC kernel and its becoming

† see § 9

completely established (signalled by an *fedctl* on the message device). Therefore between a START-NDIX command and output of this *fedctl* control of the terminal should be given to NDIX.

5.4.2. Disk Codes

Figure 4.4 shows a list of codes returned from the *feopen* command and the corresponding disk types.

feopen type	NDIX name	disk
1	di70	CDC RSD 70Mb
2	di75	CDC SMD 75Mb
3	di140	Fujitsu M2322K 140Mb
4	di288	CDC SMD 288Mb
5	di310	CDC 9720 EMD 310Mb
6	di450	CDC FSD III 450Mb
7	di520	NEC D-2352-H 520Mb
8	di70x2	Shared Fujitsu M2322K 140Mb
9	di218	Shared CDC SMD 288Mb
10	di380	Shared CDC FSD III 450Mb
11	di240	Shared CDC 9720 EMD 310Mb
12	di70_450	Shared NEC D-2352-H 520Mb
128	difd	Floppy
255		SCSI type

Figure 4.4 Disk codes

5.5. Minor device numbers

- Hard disk unit numbers and names.

Controller	Unit	minor device numbers	NDIX disk
1	0	0 - 7	di0
1	1	8 - 15	di1
1	2	16 - 23	di2
1	3	24 - 31	di3
2	0	32 - 39	di4
2	1	40 - 47	di5
2	2	48 - 55	di6
2	3	56 - 63	di7
3	0	64 - 71	di8
3	1	72 - 79	di9
3	2	80 - 87	di10
3	3	88 - 95	di11

- Floppy disk unit numbers and names.

Controller	Unit	minor device numbers	NDIX disk
1	0	96 - 103	di12
1	1	104 - 111	di13
1	2	112 - 119	di14
1	3	120 - 127	di15

- Magnetic tape unit names and numbers.

  The device name is *mt<minor>* where minor is an 8 bit number derived as follows:

```
Bit 7          must be zero
Bit 6,5        controller number - 1
Bit 4,3        density 0 =  800 bpi
                       1 = 1600 bpi
                       2 = 6250 bpi
                       3 = 3200 bpi
Bit 2          no rewind on close
Bit 1,0        unit number
```

5.6. Scheduling Considerations within the ND-100

It should be clear that the relationship of the ND-500(0) and ND-100 processors when NDIX is running is markedly different from their relationship under SINTRAN III. The main difficulty which is likely to be encountered in the implementation of the ND-100/NDIX interface is the need to provide a driver in the ND-100 which implements a scheduling algorithm which differs significantly from that afforded by the current SINTRAN III timeslicer and swapper. The following points should be noted.

- All processing by NDIX takes place in one 'process'. The NDIX kernel occupies domain 0 of this process and a hierarchy of ND-500(0) domains, below this, run NDIX user tasks.
- Only the kernel domain will make monitor calls to the ND-100; as a result, and because the NDIX kernel must be allowed to choose which NDIX domain is active; control will, in almost all cases, be passed back to the NDIX kernel immediately without waiting for a monitor call to complete.
- The net result of the asynchronous completion of monitor calls from the NDIX is the need for the ND-100 driver to maintain a fairly lengthy queue of outstanding requests. The NDIX kernel will guarantee not to queue multiple simultaneous requests on the same sub-device, however it must be allowed to have one I/O request outstanding on multiple sub-devices at the same time†. This is the normal situation when using hardware controllers.

  NDIX must be allowed to have outstanding read and write requests on all terminal subdevices simultaneously; this is to allow the NDIX terminal interface to implement read-ahead, echo and program breaks, correctly. Furthermore the NDIX system must be allowed to execute synchronous, non-I/O commands at any time.
- During interrupt servicing the ND-100 must be inhibited from generating interrupts to the ND-500(0) NDIX kernel associated with interrupt levels lower than or equal to the level of the interrupt currently being serviced. This necessitates the maintenance of a queue of outstanding, but not yet serviced completion interrupts. When the IPL is eventually lowered again interrupts should be sent to the ND-500(0) in the same order that their respective I/O requests completed. This ensures that all interrupts will eventually get serviced.
- Even though the actual interrupt priority levels used by the ND-100 hardware for each device (especially the clock) may differ from those that NDIX expects the IPL scheme must be correctly simulated.

† Support for multiple, outstanding, per sub-device I/O requests may ultimately be needed, to increase the performance of NDIX.

6. The Clock Interface

The NDIX kernel requires frequent and accurate signals from hardware, specifying the passage of real-time. These signals are used to maintain the time of day clock, and to drive the timesharing system. Within the NDIX environment clock interrupts should be generated by the ND-100 at a frequency of 25Hz (i.e. half the line frequency of 50Hz), using the interrupt mechanism described previously.

The clock may be considered an I/O device. It is generic device 5 and has an interrupt priority level of 3. It is enabled, in the same manner as for any other I/O device, by the use of an *feidev* monitor call which passes the IPL level to the ND-100. The response to this should contain the current date and time in SINTRAN format. This must be correct GMT. Once the clock has been started by issuing the *feidev* monitor call the ND-100 will generate an interrupt, whenever the next clock period expires and subsequently at the end of each 40ms time interval.

The *feinit* call returns the address of a segment in shared memory containing a clock record at a known offset. This record should contain the number of clock ticks (in 20ms intervals) since bootup. If this value is updated correctly by the ND-100 every 20ms, even if the clock interrupts are currently inhibited, then the NDIX kernel will always be able to maintain a correct idea of the time. Normally the ND-100 will create a clock interrupt every 40ms. The ND-500(0) may inhibit pending clock interrupts from being issued by raising the IPL to level 3. When this is done the clock interrupt will be placed on the interrupt queue.

If a clock period expires before the ND-500(0) has lowered the IPL, or because a synchronous request has been generated, the ND-100 must keep a count of the number of periods that have actually elapsed by updating the tick count in the clock record. This allows the UNIX kernel to update its internal clock as necessary. Only 1 clock interrupt should be outstanding on the interrupt queue at any one time.

6.1. Clock record format

Byte	Length	Meaning
0	4	Clock ticks since bootup
4	4	cpu time used by NDIX, measured in clock ticks
8	4	NDIX run-Q bit-mask
12	1	NDIX in-kernel indicator
13	1	NDIX in-interrupt handler indicator

The NDIX run queue information is placed in the clock record to make it available for use by the SINTRAN scheduling algorithms. The bit mask indicates which of the 32 NDIX run queues is occupied. The highest priority queue corresponds to the least significant bit, the lowest to the most significant bit.

A total of 64 bytes is reserved for the clock record in the shared segment.

7. Traps in NDIX in the ND-500(0)

In previous versions of NDIX some non-fatal traps (page faults and protection violations) were automatically sent to the ND-100. This is no longer the case. For a description of the trap handling for these exceptions see "Trap & Interrupt handling in NDIX version C".

7.1. General Traps

Fatal traps which occur in any of the domains of the NDIX process (i.e in kernel or user processes) are automatically sent to the ND-100. The ND-100 receives a packet of data which describes the nature of the fault. This will occur for the following faults:

Trap handler missing
power fail
processor fault†
hardware fault
monitor-call error (pseudo-trap)

When a power fail occurs this will not be indicated to NDIX. A pause of up to half-an-hour can occur in processing. NDIX should be able to continue processing at any time within this period. (Check for problems)

When a "trap handler missing" fault or a monitor call error occurs under NDIX, or whenever a hardware fault occurs NDIX must be immediately restarted in a trap context by the ND-100 and some information describing the fault sent to the NDIX trap handler‡. This is achieved by associating the NDIX routine which handles such faults with a special reserved generic device number(0). NDIX will be interrupted in the same way as for device interrupts, except that the interrupt is not associated with any IPL. The context is saved in a special trap context area , and the information in a reserved data area. The addresses of the context area and the trap data area are passed to the ND-100 during *feinit*. Since the only action taken by NDIX at this point will be an attempt to flush the buffer cache, and write a crash image before calling *feexit* not much information needs to be transferred, except that it is more convenient for subsequent fault finding to write out this data with the crash image.

It would be more convenient, and possibly safer, for NDIX merely to flush its buffer cache and call *feexit*. The *feexit* would contain information as to where the crash image should be placed (i.e. disk subdevice and block offset). The ND-100 could then write the trap information out with the crash image to the specified place. In these circumstances there would be no need to transfer the trap context and data to NDIX.

7.2. Trap Data

The layout of the current trap data packet is as shown in fig 6.2, the layout of a context block is shown in appendix 2.

Monitor call errors will be indicated by a "trap number" of 42. The erroneous generic device and subdevice numbers passed to the ND-100 will be placed in the first word of the data array in the trap data block.

† HKD to check cause of this

‡ It is currently undecided whether processor faults should always be vectored to NDIX or only when it is using the cpu.

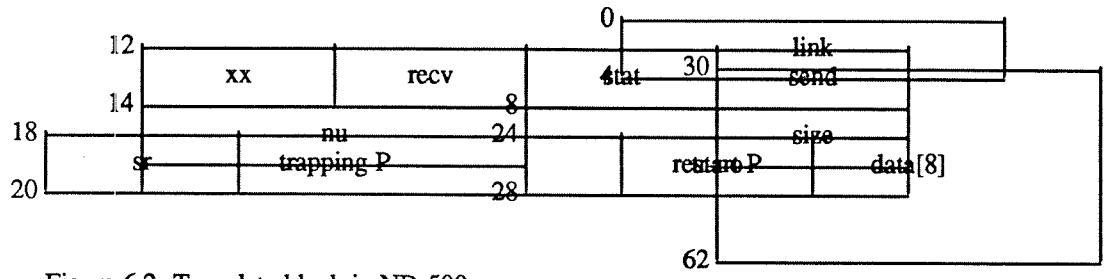


Figure 6.2 Trap data block in ND-500

8. SINTRAN/NDIX Communication

Since SINTRAN and NDIX share cpu time and memory in the ND-500(0) a method of communicating between the two operating systems is required.

A generic device (number 9) will be used for communication between Sintran and NDIX. This will have 2 subdevices. Subdevice 0 will be used for passing information for scheduling, and subdevice 1 will be used for communication about the transfer of pages between Sintran and NDIX.

8.1. Memory Sharing

Information about which physical pages are used by which operating system is passed in the form of a bit-map. This is 2048 bytes long, and each bit corresponds to a page in memory. The maximum amount of memory that can be shared in this way is therefore 16k pages or 32Mbytes. The bit-map is always passed via a shared data segment. The address of an area in memory containing the index page for this segment and the segment itself is passed to NDIX in the *feinit* call. At that time the bitmap indicates which shareable pages are available for use by NDIX (other than the kernel pages which have already been loaded and are not shareable).

After the the bit-map information has been saved an *feidev* will be issued for the message device. This will be followed by an asynchronous *fedctl* request on subdevice 1. The request will complete if Sintran requires to give or take memory from NDIX. The output of this first *fedctl* indicates to Sintran that NDIX has established itself successfully.

There will be two operator commands affecting memory allocation; Give-NDIX-memory and Take-NDIX-memory. Both commands will be passed through the interrupt-handling mechanism and cause a wake-up on the NDIX page daemon. When the page daemon has actioned the command it will send a completion message to Sintran in the form of another *fedctl* request on the message device subdevice 1. The request will not complete until Sintran requires a further transfer of pages.

The Take-NDIX-memory response will specify the number of pages required. NDIX will page out as required, and create a bit map to indicate the pages released before informing Sintran. If NDIX cannot supply all the pages requested by Sintran it should page out as many as possible and then return (in the *fedctl* command packet) the number of pages actually supplied. Sintran should not access the bit map until the reply is received. On a Give-NDIX-Memory the bit-map will indicate the pages newly available to NDIX. NDIX will check the new map (a panic may occur if it is inconsistent) update its coremap and indicate completion to Sintran. Again Sintran should not access the bit map until the reply is received.

8.2. Scheduling

NDIX must be able to inform Sintran when it is in idle loop so its priority can be lowered. The message device, subdevice 0 will be used for passing this information. A synchronous *fedctl* will be sent when NDIX is in the idle loop. The response will only be made if an interrupt from another device is about to occur.

9. Starting and Stopping NDIX

This section describes the startup and shutdown of NDIX. It's intention is to outline the flow of control, structures and utilities required.

9.1. Loading from Sintran.

The NDIX kernel is loaded† into ND-500(0) memory by the Sintran command:

@LOAD-NDIX,<FILENAME>,<CONSOLE>

and starts running at location 4, domain 0. The actual kernel loaded is in the Sintran files:

FILENAME:DSEG
FILENAME:PSEG

which appear to be standard ND500(0) data and program segments. The memory allocation for the NDIX system is derived from a "sidefile":

FILENAME:INIT

which contains (inter alia) the minimum size of the contiguous area and the minimum number of allocatable pages.

9.2. Initial NDIX activity

The NDIX kernel will issue an *feinit* call, and the response packet is filled in by the ND-100, obtaining the *howto* and *rootdev* fields from the sidefile. There are three main conditions which can cause the boot to fail after this point:

- The NDIX kernel attempts to find its root device, either from *rootdev*, or by operator intervention (when RB_GENERIC set in *howto*) at the NDIX console‡. If *rootdev* does not specify a valid root device, the the NDIX kernel will issue an *feexit* with RB_BOOT in *s3_how* unset, indicating that no further action can be attempted. If the operator does not specify a valid root device, the NDIX kernel will reprompt.
- If the contiguous memory allocated to NDIX is not enough, the NDIX kernel will issue an *feexit* with *s3_how* set to RB_CONTIG and RB_RESET. The command packet will be built from the response packet to the *feinit*, except that *contigno* will have been modified.
- If the dynamically allocatable memory allocated to NDIX is not enough, the NDIX kernel will issue an *feexit* with *s3_how* set to RB_SCATTER and RB_RESET. The command packet will be built from the response packet to the *feinit*, except that *pageno* will have been modified.

The NDIX kernel will then examine the flag RB_SINGLE in *howto*, and modify the flags to */etc/init* accordingly so that *init* will bring NDIX to either single or multi-user state.

9.2.1. Sintran activity if boot fails

If NDIX fails to establish itself, and is still able to communicate with the ND100, then it will issue an *feexit* monitor call.

If RB_BOOT in *s3_how* is unset, then NDIX should be halted with an error message (this may or may not be in addition to the error message passed in *feexit*) indicating that the kernel image (or its associated data in the sidefile) on Sintran is not usable.

Sintran's indication that NDIX is successfully established is an *fedctl* on the message device - this should be interpreted as the kernel image (and sidefile) are usable, and future RB_HALT settings will not indicate a Sintran error condition.

---

† This loading is described in detail in § 2.
‡ See § 4.4.1

Otherwise, either RB_CONTIG or RB_SCATTER will be set, together with RB_RESET, and the sidefile should be modified to include the new value, and the system rebooted using the information contained in the *feexit* command packet.

9.3. Normal NDIX termination

A normal NDIX termination is caused by a call to *reboot(2)* from one the utilities *halt(8)* or *reboot(8)*†. Normal NDIX shutdowns can be grouped into 3 categories:

9.3.1. Total Shutdown without Reboot

This is caused by a call to *halt(8)*, which will issue one of the following system calls, dependant on the *-n* flag:

```
reboot(0, (char *)0, (dev_t)0, (dev_t)0);
reboot(RB_NOSYNC, (char *)0, (dev_t)0, (dev_t)0);
```

The NDIX kernel will issue an *feexit* with RB_BOOT in *s3_how* unset.

9.3.2. Shutdown with Reboot of the Same System

This is caused by a call to *reboot(8)*, which will issue the following system call:

```
reboot(RB_BOOT|RB_SAME, (char *)0, rootdev, condev);
```

The command *reboot(8)* will need modifying for new arguments *-s*, *-r rootdev* and *-c condev*. The flags will have the following effects on the system call:

-n The first argument to the system call will have RB_NOSYNC additionally set.

-s The first argument to the system call will have RB_SINGLE additionally set.

-r The third argument to the system call will be set to indicate the root device. If this flag is not present, then the first argument is aditionally set to RB_SAMEROOT.

-c The fourth argument to the system call will be set to indicate the NDIX console. If this flag is not present, then the first argument is aditionally set to RB_SAMECON.

The reboot system call leads to the issue of an *feexit* command packet with RB_BOOT set in *s3_how* and default values for the remaining fields as follows:

howto rootdev condev	value received in the *feinit* call
contigno pageno bootme	value received in the *feinit* call
crshdev crshst crshend	not set in this release

Aditionally, *s3_how* has RB_RESET set if any of RB_SINGLE, RB_SAMEROOT or RB_SAMECON are set in *howto*.

The additional bits set in the first argument will have the following effects on the *feexit* command packet:

RB_NOSYNC	None.
RB_SINGLE	*howto* in the command packet is set to include RB_SINGLE.
RB_SAMEROOT	The third argument to the system call is copied to *rootdev* in the command packet.
RB_SAMECON	The fourth argument to the system call is copied to *condev* in the command packet.

† Other utilities call these utilities, however *halt(8)* and *reboot(8)* are the only ones which invoke the system call.

9.3.3. Shutdown with Reboot of a New System.

This is caused by a call to the new utility *newboot(8)*,

newboot [-g] [-s] [-r rootdev] [-c condev] filename

which will issue the following system call:

reboot(RB_BOOT|RB_NEW, filename, rootdev, condev);

The flags are defined as follows:

-g The first argument to the system call will have RB_GENERIC additionally set.

-s The first argument to the system call will have RB_SINGLE additionally set.

The second argument is passed unchanged to the system call.

If the flag *-r* is present, then *rootdev* is passed as the third argument, otherwise the first argument will have RB_SAMEROOT additionally set.

If the flag *-c* is present, then *condev* is passed as the fourth argument, otherwise the first argument will have RB_SAMECON additionally set.

NDIX will issue an *feexit* call with *s3_how* set to RB_BOOT|RB_RESET, and *howto* reflecting the RB_GENERIC and RB_SINGLE options.

If RB_SAMEROOT was set in the call to *reboot(2)* then *rootdev* is set to the value passed in the *feinit* response packet, otherwise it is set to the third argument.

If RB_SAMECON was set in the call to *reboot(2)* then *condev* is set to the value passed in the *feinit* response packet, otherwise it is set to the fourth argument.

The *feinit* response packet is also used for the values of *contigno* and *pageno*. The setting of *bootme* is taken from the second argument to *reboot(2)*.

9.3.4. Sintran Activity on Normal NDIX Termination

If RB_BOOT in *s3_how* is unset NDIX memory may be reclaimed for other use. NDIX will only be restarted by a new *LOAD-NDIX* command.

If RB_BOOT is set in *s3_howto*, the NDIX driver should reboot NDIX, from the file named in *bootme*. Before rebooting NDIX however, the sidefile must be updated if RB_RESET was set in *s3_how*. Note that the sidefile might not exist at this point.

9.4. Abnormal NDIX termination

When the NDIX kernel terminates abnormally it will behave as if the system call

reboot(RB_BOOT, (char *)0, (dev_t)0, (dev_t)0);

had been issued.

9.4.1. Sintran Activity on Abnormal NDIX Termination

Is covered in §1.5 above, since Sintran is not informed that the termination was not intentional.

9.5. Data Structures

9.5.1. The Sidefile

Each NDIX system has five variables required for booting and memory sizing. They are:

howto	the mode the kernel is booted in
rootdev	the root device
condev	the NDIX console device

contigno	size of the contiguous memory area
pageno	number of dynamically allocated pages

These variables need preserving over complete system shutdowns and are associated with a given pair of Sintran files. This document assumes these variables are located in a file, however, the exact implementation may be decided by the driver.

9.5.2. Reboot Flags

The following flags are used to indicate the type of boot or halt required. First in the variable *s3_how* (in the *feexit* command packet) which indicates the actions required by Sintran.

RB_BOOT	0x1	Reboot NDIX
RB_RESET	0x2	Reset the sidefile
RB_CONTIG	0x4	New value of *contigno*
RB_SCATTER	0x8	New value of *pageno*
RB_CRASH	0x10	Take crash dump (not implemented)

The variable *howto* (in the sidefile, and sent by *feexit* and *feopen*) has the following values:

RB_GENERIC	0x20	This is a Generic Kernel
RB_SINGLE	0x40	Only boot to single user

The following flags are (additionally) used in the first parameter to *reboot(2)*.

RB_NOSYNC	0x80	Do not *sync(2)* before halt
RB_SAME	0x100	Rebooting from same kernel
RB_NEW	0x200	Rebooting from new kernel
RB_SAMEROOT	0x400	Use same root after reboot
RB_SAMECON	0x800	Use same console after reboot

The kernel uses internally:

RB_PANIC	0x1000	System is panicing

The following flags are reserved for future use:

RB_ASKNAME	0x2000	Ask for kernel filename
RB_INITNAME	0x4000	Name given for /etc/init

Please note these values conflict with those currently in *<sys/reboot.h>*, care will be needed when these values become standard.

9.5.3. Data Common to *feinit* and *feexit*

The following fields are common to both the *feinit* response packet and the *feexit* command packet.

howto	mode to reboot NDIX in
rootdev	NDIX root device
contigno	number of contiguous pages needed by NDIX
pageno	number of pages allocated in bitmap
bootme	Sintran file containing kernel

9.5.4. Types of *condev* and *rootdev*

These variables are passed to utilities as filenames, and to the system call as *dev_t* (both major and minor parts).

The values passed to/from the NDIX driver and stored in the sidefile are the 16 bit *minor device* number.

10. Xmsg Implementation

Communication between the Xmsg driver in NDIX and the ND-100 will be through 2 ring buffers. One for commands from NDIX to the ND-100, and another for responses from the ND-100 to NDIX. This minimises the number of monitor calls required, and the number of interrupts queued for NDIX.

10.1. Initialisation

This will use an *feidev* call, similar to that for any other generic device. It is used to determine whether Xmsg is running in the ND-100. The ND-100 also returns the addresses of the Command and response buffers.

10.2. Operation

Two *fedctl* calls will be implemented; XKICK which acts as a "wakeup" on the ND-100 to indicate that a command has been placed in an empty ring buffer, and XWBUF which indicates that the command buffer is full and an interrupt is required when a command has been actioned and there is room for NDIX to place another command in it. Both these calls should be asynchronous.

The normal method of operation when inserting data into a ring buffer will follow the sequence:

- If the command buffer is full send an *fedctl*(XWBUF) message to the ND-100. Sleep until a response is received indicating that there is sufficient free space in the command buffer.
- Transfer command data to buffer. Update write pointer.
- If the buffer was empty before the command data was transferred (i.e. Readptr = Writeptr) send an *fedctl*(XKICK) message to the ND-100.

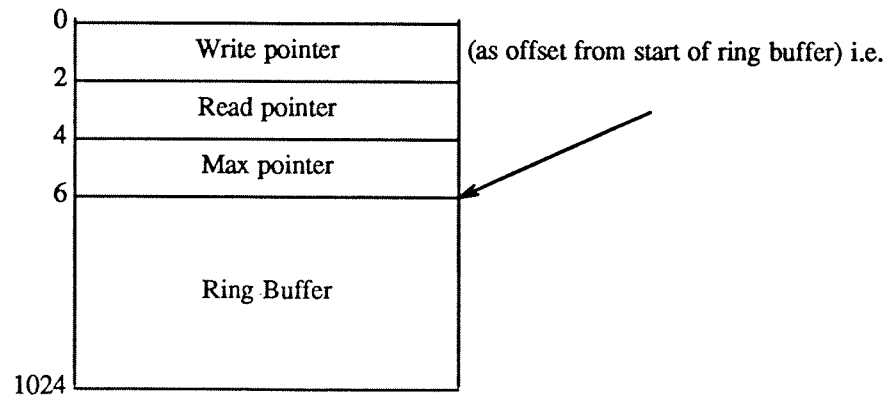
Occasionally an interrupt will be queued for an Xmsg device. The action that should be taken by NDIX at this point is as follows:

- Check the receive buffer for any incoming packets. Read and take the necessary action.
- Wakeup any task that is waiting to input data to the command buffer.

10.3. Data Areas

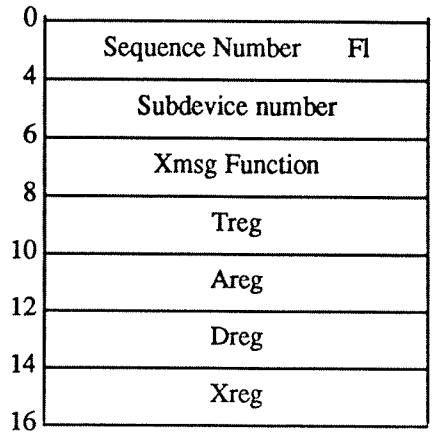
The data buffers, command buffer and response buffer should all be situated on segments with no-cache capability. The command and response buffers are situated on the shared data segment information about which is passed in the *feinit* call. All data buffers used by a subdevice must reside within the same physical memory bank, the address of which is passed during *feopen*. The data buffers are situated on the NDIX no-cache segment†.

10.3.1. Ring Buffers



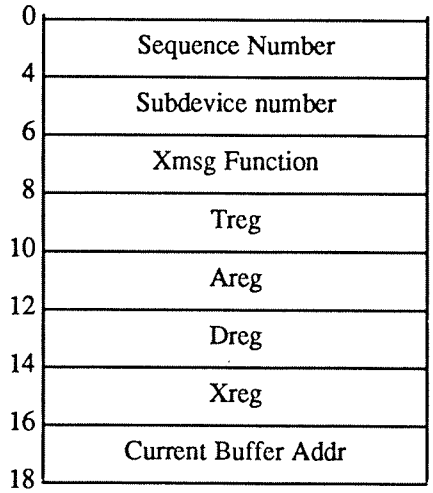
† For the layout of this see appendix 1.

10.3.2. Command Packet



The least-significant bit in the sequence number is a flag which indicates whether an interrupt should be generated when this command has been actioned, if it is set an interrupt is required.

10.3.3. Response Packet



10.4. Xmsg Commands Supported by NDIX†

† AMSA to provide this.

11. Error Handling

The *feerrm* monitor call has been defined to allow UNIX to send a single error code to the ND-100, when it detects a non-fatal error condition or inconsistency. This call is always synchronous.

The error codes which can be sent in this way are:

Code	Meaning
1	Unexpected interrupt on this device
2	Interrupt received for unknown device

Error codes are passed from the ND-100 to NDIX in the monitor call response packet. The following codes are defined:

Code	Meaning
0150740	Generic Device number out of range
0150741	Generic group not-initialised
0150742	Illegal sub-device number
0150743	Sub-device not open
0150744	Generic device already initialised
0150745	Illegal request type (monitor call number)
0150746	Xmsg not installed or not running
0150747	sub-device already open
0150750	Not legal NDIX terminal
0150751	Fecall must be synchronous
0150752	Fecall must be asynchronous
0150753	Illegal parameter value in fecall
0150754	Illegal operation code in fecall
0150755	SIII Driver error
0150756	Error from mass storage device in Sintran
0150757	Error in i/o detected by Sysmon
0150760	Byte count not modulo hardware block size
0150761	Byte count not even
0150762	Carrier lost

APPENDIX 1. Pre-defined Addresses†

Shared data segment

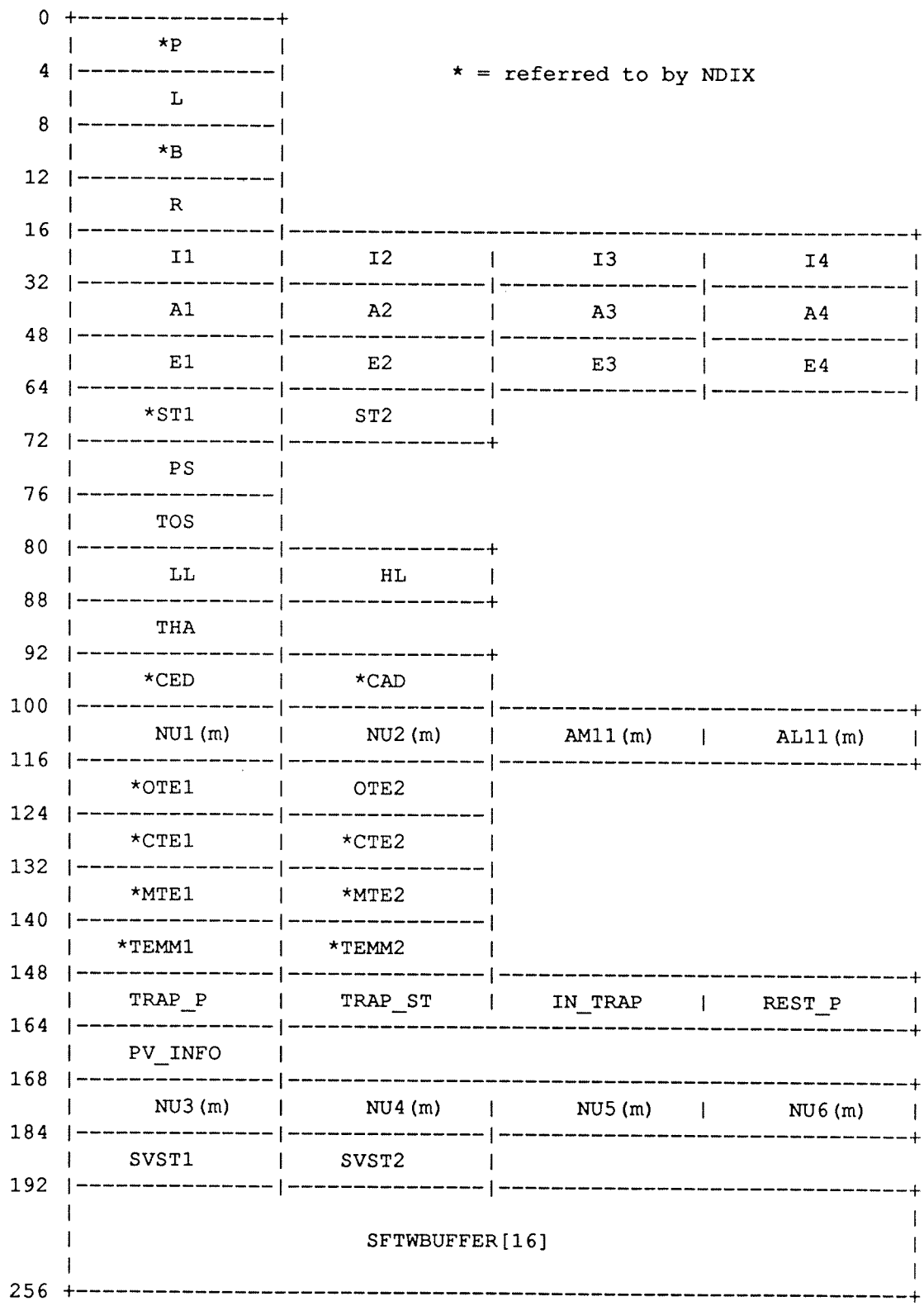
Name	Address
Bitmap	
IPL record	
Clock record	
Xmsg command buffer	
Xmsg response buffer	

NDIX no-cache segment

Name	Address
Buffer cache	
raw i/o buffers	
xmsg data buffers	

† To be completed

APPENDIX 2. Context block (AS DEF in cxb.h - CHECK VALIDITY)



APPENDIX 3. Monitor Call Details

Table of Contents