SAND 93–2618
Unlimited Distribution
Printed February 18, 1997

Distribution
Category
UC–705

# Intel NX compatibility under SUNMOS*

Kevin S. McCurley
Organization 1423
Sandia National Laboratories
Albuquerque, NM 87185
mccurley@cs.sandia.gov

November 3, 1993

**Abstract**

SUNMOS on the Intel Paragon™ provides partial compatibility with codes written to run under OSF/NX as well as NX on the Intel iPSC/860 and Delta. Some of the Intel NX library calls are not supported under SUNMOS, or behave differently from what might be expected. This document describes some of the limitations and differences between NX emulation under SUNMOS and true NX.

*WARNING: This document is subject to change. At present it describes a version of `clicompat.c` and `cligcompat.c` that has not been tested very extensively. It is recommended that you obtain the most up-to-date version possible of this document. A postscript version should be available by anonymous ftp from `cs.sandia.gov` in the directory `pub/sunmos/doc`. It is also included with the standard distribution of SUNMOS.

# 1 Introduction

This document is intended for people who are porting or tuning Intel NX applications for the Intel Paragon™ under SUNMOS [1]. A detailed discussion of the design goals of SUNMOS is beyond the scope of this document, but the overall goal is to provide high performance message passing and process service on nodes of Massively Parallel MIMD computers. Unlike OSF, SUNMOS is not intended to be a full-featured operating system.

Programmers who are porting codes from previous generations of Intel Supercomputers should be aware that each generation of hardware and software has introduced subtle differences in the way message passing is handled, and these differences can have substantial impact on the performance of applications. SUNMOS provides partial emulation for Intel NX-style message passing, and the purpose of this document is to describe some of the differences and limitations of this emulation. The basic documentation for Intel NX message passing applications remains:

- Paragon™ User's Guide,
- Paragon™ OSF/1 Fortan System Call Reference Manual,
- Paragon™ OSF/1 C System Call Reference Manual,

This document should be regarded as a supplement to these standard Intel manuals.

# 2 Processes

Since SUNMOS is a single-process/single-user operating system, the notion of a `ptype` does not exist. In SUNMOS, every node user process has process id of `_my_nid`. Routines that take an argument for process id will ignore that argument. This affects the following routines:

| | | | |
|---|---|---|---|
| `cprobex()` | `flushmsg()` | `irecvx()` | `nx_nfork()` |
| `crecvx()` | `hrecv()` | `isend()` | `nx_pri()` |
| `csend()` | `hrecvx()` | `isendrecv()` | `nx_waitall()` |
| `csendrecv()` | `hsend()` | `myptype()` | `setptype()` |
| `dclock()` | `hsendx()` | `nx_initve()` | |
| `flick()` | `iprobex()` | `nx_load()` | |

# 3 Partitions and the host/node model

There is currently no host/node programming model under SUNMOS, so the following calls are not supported: `myhost()`, `nx_initve()`, `nx_load()`, and `nx_nfork()`. In addition, SUNMOS does not use the notion of partitions, meaning that the following are not supported: `nx_chpart()`, `nx_mkpart()`, and `nx_rmpart()`.

[1] SUNMOS stands for Sandia UNM Operating System.

# 4 Message passing semantics and performance

## 4.1 Message buffering

Message buffering under SUNMOS is accomplished via a `comm` space that is set aside at load time. This is configurable by the user through a switch to the `yod` command. When a message arrives at a node, there are two possibilities. If a receive matching the characteristics of the message has already been posted by the user process, then the message is automatically deposited in the matching user buffer. Otherwise, space is allocated from the `comm` space, and the message is "buffered by the system". When too many messages arrive at a node without receives having been posted for them, it is possible to overflow the memory available in the `comm` space, and the application will currently fault. It is also possible that memory fragmentation may occur in the `comm` space, in which case there may not be a contiguous segment of memory large enough to accomodate the incoming message, causing a fault. When a message is sent on a node, a local copy may or not be made, depending on alignment of the send buffer (see below). Message buffering under SUNMOS therefore resembles the scheme used by VERTEX™ on the nCUBE-2, except for the fact that messages are not always copied into the `comm` space on the sending side under SUNMOS.

I am unable to completely understand the description of message buffering given in Chapter 2 of the Paragon™ User's Guide, and cannot therefore give a meaningful comparison to the OSF method of buffering and flow control. Intel has never supplied any public documentation to completely describe the flow control procedures used in any of their machines, but some description of the iPSC/860 and Delta machines was given by Rik Littlefield of Pacific Northwest Labs[2].

## 4.2 Message packetization and race conditions

One notable difference between SUNMOS and OSF/1 on the Paragon has to do with message packetization. SUNMOS currently does no packetization of messages, locking the entire channel until the message is completely sent. This allows SUNMOS to achieve much higher bandwidth, and eliminates the need for extraneous flow control messages from the system software (performance of SUNMOS message passing has been reported on previously[3]). An unfortunate side effect is that codes that suffer from race conditions in message passing may exhibit different behaviour under SUNMOS than they do under OSF/1, because small messages can effectively be delayed by large messages. This can be avoided by desiging the application to eliminate dependence on message timing.

## 4.3 Performance and message alignment

Under SUNMOS, a nonblocking message that is not aligned on an 8-byte boundary will have a local copy made before sending it (note: space for this copy is taken from the user's heap, *not* the `comm` space). For highest performance and conservation of memory, the programmer should therefore try to make sure that buffers are aligned on 8-byte boundaries. Note also that if an unaligned `isend()`

[2] Richard J. Littlefield, "Characterizing and Tuning Communications Performance on the Touchstone DELTA and iPSC/860 (extended abstract)". Proceedings of the 1992 Intel Supercomputer Users' Group Conference.

[3] David Greenberg, Barney Maccabe, Kevin S. McCurley, Rolf Riesen, and Stephen Wheat, "Communication on the Paragon". Proceedings of the 1993 Intel Supercomputer Users' Group Conference.

or `isendrecv()` is used, then the send message buffer can be immediately reused, since a copy is made anyway.

### 4.4 Broadcast

In the NX calls such as `csend()`, `isend()`, `csendrecv()`, and `isendrecv()`, the user can specify a destination node id of -1 to broadcast to all nodes except the calling node. Under NX, this is functionally equivalent to simply performing `numnodes()`-1 csend calls, but the implementation uses a store-and-forward protocol handled by the operating system to broadcast information via a tree. In SUNMOS, the broadcast is currently done by forwarding the message in a spanning tree, but the forwarding of the message is not guaranteed to be performed until *every* node either receives the messages using a blocking send, or a `msgid` is released from an asyncronous receive for the message. This restriction currently exists under OSF/NX as well.

### 4.5 Nonblocking send semantics

The functions `isend()` and `isendrecv()` behave differently under SUNMOS and NX. Under NX, the send buffer can apparently be reused as soon as the `isend()` or `isendrecv()` call returns, implying either that the call blocks until the message is gone from the node, or else a local copy is made before sending the message. Since it is faster under SUNMOS to send a large message than it is to copy it, there would seem to be no reason to have an asyncronous send call like `isend()`. Under SUNMOS, a send buffer for `isend()` should not be reused until the message transmission is completed, as indicated by the `msgdone()` or `msgwait()` calls.

Under SUNMOS, there are two possibilities for `isendrecv()`. If the send and receive buffers overlap, then a blocking send is followed by a nonblocking receive. If the buffers do not overlap, then a nonblocking receive is followed by a nonblocking send, and the two message ids are merged and returned to the user. For nonoverlapping buffers, a user should therefore check that the message has been sent using `msgwait()` or `msgdone()` before reusing the send buffer. This allows more aggressive overlapping of communication and computation under SUNMOS, but may result in improper behaviour of programs written to exploit this (peculiar) feature of NX.

### 4.6 Message cancellation

Intel NX provides several calls for canceling various kinds of message passing operations: `msgcancel()`, `msgignore()`, and `flushmsg()`. The primary goal of SUNMOS is to provide the highest level of performance possible, and this requires strict rules between user code and the SUNMOS kernel. As a consequence, there is very limited support for these functions, and they have a low priority for future support. See Section 12 for details. Note that `flushmsg()` is currently not implemented under OSF.

### 4.7 Forced Message Types

The current version supports sending of forced message types from the range 1,073,741,824 to 1,999,999,999, but their behaviour is somewhat different than ordinary NX. Under NX, a forced message type is sent directly to user space on the receiving side if a receive has been posted, but is discarded if the receive has not been posted. Under SUNMOS, a forced message type is never discarded on the receiving end. If a program depends on the behaviour of NX to discard a message, then it may run out of communication space. If this is important for some applications, then it may be supported in the future.

### 4.8 Wildcarded receives

NX specifies that a wildcard type of -1 in a receive call should not match a system message type or a forced message type (e.g., a wildcard should not match a message type larger than 1,999,999,999). There is currently no support for this in SUNMOS, and a wildcard on the receiving side may in fact intercept such a message. If this happens, then an error message will be printed and the process will be aborted. Note that the `iprobe[x]()` and `cprobe[x]()` functions work correctly when given a -1 typesel.

### 4.9 Selective Receives via typesel

In NX, the user is able to specify a typesel less than -1 in order to wildcard over a small range of types. This is not supported under SUNMOS, and is unlikely to ever be supported.

### 4.10 Number of `msgids`

In NX, the number of `msgid`s available to the user from asyncronous message passing calls such as `isend()` and `irecv()` is limited (currently, 999 on the Paragon, and 5000 on the iPSC/860). Under SUNMOS, there is no limitation other than the amount of memory available through `malloc()`.

Users should be aware that having a large number of unsatisfied receives outstanding will have a negative impact on message latency since incoming messages will have to be matched against a list of posted receives. This could be fixed by using a hash table to match incoming messages to outstanding receives, but it would also require some overhead to maintain the hash table. We have chosen to implement SUNMOS without such hash tables, in the hopes that users will not have large numbers of outstanding receives.

A similar situation exists in the implementation of `iprobe()`, `iprobex()`, `cprobe()`, and `cprobex()`. These functions do a linear search on buffered messages, and are therefore most efficient when the list is short. If a user chooses to use the system to buffer large numbers of messages simultaneously, then the performance of these functions will degrade linearly.

### 4.11 `info...()` calls

The `infonode()`, `infocount()`, and `infotype()` calls are supported, but the access to the global array `msginfo` is not.

### 4.12 `hrecv()`, `hsend()`, etc.

SUNMOS currently does not support the message passing calls `hsend()`, `hrecv()`, etc., that rely on a handler. This should change very soon once the current implementation of signals undergoes more testing.

## 5 Underscore calls

In NX, every function call seems to have a corresponding function that has an underscore in front of the name. For example, there is a `_csend()` call that behaves similarly to the `csend()`. The difference between these two stems from their error handling; the underscore calls set `errno` upon discovering an error, and allow the user to recover from the error. The only underscore calls currently supported under SUNMOS are `_csend()`, `_crecv()`, `_isend()`, and `_irecv()`. Others can easily be added if needed. Note that the value of `errno` may not be defined in `errno.h`.

## 6 Global operations

The global operations are currently not optimized, but this will be improved greatly in future releases. In addition, `gcol()` and `gcolx()` currently requires `numnodes()` to be a power of two.

## 7 Include files

NX C programs written for Intel machines are supposed to include the following lines:

| Machine | include line |
|---|---|
| iPSC/860 NX | `#include <cube.h>` |
| Delta NX | `#include <mesh.h>` |
| Paragon™ OSF/1 | `#include <nx.h>` |

SUNMOS programs that are run on the Paragon™ should instead use the line

```
#include <icompat.h>
```

## 8 File I/O

File I/O is currently handled through communication with OSF nodes running the `yod` and `fyod` programs. Initial calls to open a file are sent to `yod`, which may then hand the request off to an `fyod` program running on another OSF node. `fyod` programs can be started to manage requests for a given directory. It responds to the node that made the request to open the file, and thereafter all communication for file I/O is handled by communication between the SUNMOS node and the `fyod` program. Ultimately this will provide better scalability, and will allow support for the Intel PFS scheme. Currently, I/O is handled in this way to access RAID disk systems under both Fortran and C, and future support is planned for PFS and possibly other file systems.

As such, standard UNIX file I/O is handled by the I/O nodes of the Paragon™. Achieving high rates of I/O requires the user to explicitly manage files on different physical devices. There is no support for CFS emulation, nor is there any access to non-blocking I/O calls. This means that the following are unsupported:

```
gopen()    iread()    iwrite()
iodone()   iomode()   setiomode()
```

## 9 mypart()

The function mypart() from the Delta is supported, with a few changes. First, mypart() returns a value equal to the number of nodes of the application. The C prototype is:

```
long mypart(long *rows, long *columns);
```

There are also cases where the values of *rows and *columns will be set to -1. This results from the fact that the yod utility attempts to allocate a rectangular partition, but if the -allocation strict flag is not used, it may allocate a partition that is not rectangular (and not on adjacent nodes). Note however that the allocator may choose a partition of the form

```
* * . *
* * . *
. . . .
* * . *
```

Such an allocation is considered to be rectangular, with dimensions $3 \times 3$, no matter whether the unallocated nodes (represented as dots) are OSF or SUNMOS nodes for another application. For such an allocation, myparts() works as you would expect.

## 10 64-bit integer support

NX includes support for a 64-bit integer data type, primarily for dealing with files in excess of 2 Gigabytes. No support for this is currently included or even planned in SUNMOS. This affects the following routines:

| | | | |
|---|---|---|---|
| eadd() | emod() | eseek() | etos() |
| ecmp() | emul() | esize() | |
| ediv() | esub() | estat() | |

## 11 List of fully supported calls

| | | | | |
|---|---|---|---|---|
| cprobe() | gdlow() | gisum() | gsendx() | msgmerge() |
| cread() | gilow() | gopf() | gsync() | msgwait() |
| cwrite() | gslow() | gssum() | infocount() | mynode() |
| dclock() | gdprod() | giand() | infonode() | numnodes() |
| gdhigh() | giprod() | gland() | infotype() | mypid() |
| gihigh() | gsprod() | gior() | iprobe() | nx_perror() |
| gshigh() | gdsum() | glor() | msgdone() | infopid() |

## 12 List of partially supported calls

| function | limitation |
|---|---|
| `cprobex()` | ptypesel is ignored |
| `crecv()` | ptypesel is ignored and wild cards may match system or forced message types. |
| `crecvx()` | ptypesel is ignored and wild cards may match system or forced message types. |
| `csend()` | ptype is ignored. Broadcast requires that everyone receive it. |
| `csendrecv()` | ptype and ptypesel are ignored and wild cards may match system or forced message types. |
| `flushmsg()` | Can only be called to flush messages in the system buffer on the node calling it. The method used is stupid - malloc space for it, receive it, and free up the space. Calling `flushmsg()` to flush messages from another node will produce an error message and kill the process. |
| `gcol()` | currently requires power of two nodes |
| `gcolx()` | currently requires power of two nodes |
| `hwclock()` | argument is an array of two ints |
| `infoptype()` | pid is always constant |
| `iprobex()` | ptypesel is ignored |
| `irecv()` | ptypesel is ignored and wild cards may match system or forced message types. |
| `irecvx()` | ptypesel is ignored and wild cards may match system or forced message types. |
| `isend()` | ptype is ignored. Broadcast requires that everyone receive it. |
| `isendrecv()` | ptype and ptypesel are ignored. Wild cards may match system and forced message types. Broadcast requires that everyone receive it. |
| `msgcancel()` | Can only be used for outgoing messages |
| `msgignore()` | For outgoing messages, it blocks until the message has been sent, and then frees up the `msgid`. For incoming messages that have already arrived and been buffered, it frees up any memory in comm space and releases the message id. For incoming messages that have not already been received by the system, it prints an error message and kills the process. |
| `myptype()` | pid is always constant |
| `nodedim()` | returns -1 if `numnodes()` is not a power of two |
| `mypart()` | see section 9 |

## 13 List of unsupported calls

The following list is of unsupported OSF and NX library calls, but does not include all of the library functions that are possible from OSF/NX. A quick check revealed at least 1,680 possible library calls under OSF!

| | | | |
|---|---|---|---|
| `eadd()` | `forflush()` | `isnand()` | `nx_mkpart()` |
| `ecmp()` | `gopen()` | `isnanf()` | `nx_mkpart_rect()` |
| `ediv()` | `hrecv()` | `iwrite()` | `nx_mkpart_map()` |
| `emod()` | `hrecvx()` | `led()` | `nx_nfork()` |
| `emul()` | `hsend()` | `load()` | `nx_pri()` |
| `esub()` | `hsendx()` | `lsize()` | `nx_rmpart()` |
| `eseek()` | `hsendrecv()` | `masktrap()` | `nx_waitall()` |
| `esize()` | `iodone()` | `mclock()` | `restrictvol()` |
| `estat()` | `iomode()` | `myhost()` | `setiomode()` |
| `etos()` | `iowait()` | `nx_chpart()` | `setptype()` |
| `festat()` | `iread()` | `nx_initve()` | `stoe()` |
| `forceflush()` | `iseof()` | `nx_load()` | `waitall()` |
| `flick()` | `isnan()` | `nx_loadve()` | |

## 14 List of Untested Functions

In general we believe that all standard UNIX I/O calls are emulated from SUNMOS nodes (as described in recent editions of K&R). In addition, the following functions are apparently included in the libraries that SUNMOS links, but have not as yet been tested.

| | |
|---|---|
| `fpgetround()` | `fpgetsticky()` |
| `fpsetround()` | `fpsetsticky()` |
| `fpgetmask()` | `lseek()` |
| `fpsetmask()` | |

## 15 Extensions

SUNMOS provides the rather strange ability to run codes with a combination of both nCUBE and Intel NX message passing calls, but this is not recommended. The following additional calls are currently supported on the Paragon:

| function name | purpose |
|---|---|
| `pmem_size()` | currently returns 16 or 32 to represent the number of megabytes of physical memory on the node. |
| `_myphysnode()` | the physical node number. This starts at 0 in the upper left corner of the machine, and counts across the machine to the end, and then returns to the second row at the left. Empty node bays in the machine are counted. |

## 16 Bugs

Please report bugs or comments to `sunmos-support@cs.sandia.gov`.

## Internal Distribution

| | |
|---|---|
| 1400 | E. H. Barsis |
| 1402 | S. J. Dosanjh |
| 1403 | G. S. Davidson |
| 1404 | J. A. Ang |
| 1421 | W. J. Camp |
| 1421 | Daniel Barnette |
| 1421 | David Gardner |
| 1421 | Grant Heffelfinger |
| 1421 | Scott Hutchinson |
| 1421 | Steve Plimpton |
| 1421 | Mark Sears |
| 1421 | John Shadid |
| 1421 | Julie Swisshelm |
| 1422 | R. C. Allen |
| 1422 | Bruce Hendrickson |
| 1422 | David Womble |
| 1423 | E. F. Brickell |
| 1423 | David Greenberg |
| 1424 | A. L. Hale |
| 1424 | Robert Benner |
| 1424 | Pat Crossno |
| 1424 | Carl Diegert |
| 1424 | John van Dyke |
| 1424 | Gabi Istrail |
| 1424 | Robert Leland |
| 1424 | Barney Maccabe |
| 1424 | Mike Proicou |
| 1424 | Rolf Riesen |
| 1424 | Mack Stallcup |
| 1424 | Courtenay Vaughan |
| 1424 | Stephen Wheat |
| 1425 | J. H. Biffle |
| 1431 | J. M. McGlaun |
| 1431 | Allen Robinson |
| 1432 | P. Yarrington |
| 1432 | Huei Eliot Fang |
| 1433 | P. L. Stanton |
| 1434 | D. R. Martinez |