Technische Universität München
Institut für Informatik
Lehrstuhl für Rechnertechnik und Rechnerorganisation
D-80290 München

Users' Guide
Version V1_1_4

Stefan Lamberts, Georg Stellner, and Thomas Ludwig
nxlib@informatik.tu-muenchen.de

February 15, 1995

Contents

1 Introduction

Commercial and scientific applications require much computational power today. To satisfy these needs supercomputer and multicomputer systems were developed. Especially the latter approach seems to be promising as distributed memory multiprocessors achieve good scalability. Recently, networks of workstations have also become popular to serve as a computational resource. Several message-passing libraries which support parallel applications on coupled workstations have become available [2] [1] [9] [3]. Most of these environments are also available for multicomputer systems, which allows for migrating applications from networks of workstations to multicomputer systems and vice versa.

The disadvantage of the above approaches is, that on multicomputer systems these environments introduce additional software overhead. The message-passing interfaces have to be mapped to calls of the basic message-passing interface offered by the operating system. Thus, applications waste some of the available computational power on those multicomputers. A different approach is to offer the programming interface of a multicomputer system, such as the Paragon on a network of workstations. Hence, on multicomputers no additional software overhead is necessary to run applications. On the workstations software overhead is introduced anyway, so there is no difference in offering a similar interface like a multicomputer system.

NXLib uses the latter approach to provide a message-passing interface which is compatible to the Paragon system. This facilitates using coupled workstations to develop parallel applications for Paragon systems. Apart from withdrawing development workload from Paragon systems also production runs of applications can be done on coupled workstations.

For more information on the implementation and experiences with NXLib see [8] [12] [10] [11].

2 Availability and Installation

2.1 Obtaining NXLib

The NXLib package is available via anonymous ftp and may be copied under the terms of the GNU software license agreement. Table 1 shows the information which is necessary to ftp NXLib. As the software and the User's Guide are installed in compressed format, make sure that you have set the `binary` mode for the ftp transmission.

Server:	ftpbode.informatik.tu-muenchen.de
	131.159.0.156
Login:	`anonymous`
Password:	your email address
Directory:	`NXLIB`
Files:	`NXLibV1_1_4.tar.Z`, `NXLibUgV1_1_4.ps.Z`

Table 1: Availability of NXLib via anonymous ftp

If you have any problems in using the ftp server you can send an email to nxlib@informatik.tu-muenchen.de with a detailed description of what is going wrong.

2.2 Unpacking

After the successful transmission of the files you will have to unpack them by the following commands:

```
$ uncompress NXLibV1_1_4.tar.Z
```

```
$ tar xvf NXLibV1_1_4.tar
```

This results in a directory called `NXLibV1_1_4` which has been created in the current working directory. It contains all source files required to install NXLib.

2.3 Installation procedure

Once you have successfully unpacked the source code and `NXLibV1_1_4` has been created, go to that directory:

```
$ cd NXLibV1_1_4
```

After that you must edit the `Makefile.config` and configure the apropriate architecture and the appropriate installation directory of the libraries and binaries. Therefore, you just have to choose one of the `ARCH` macros. Change the value of the `NXLIBDIR` marco if necessary. An installation can either be done in one's home directory (e.g. /home/joe_average/`NXLibV1_1_4`) or in the system area (e.g. /usr/local/`NXLibV1_1_4`). During the installation process the value of the installation directory will be inserted in several shell scripts which come along with NXLib. In addition the `CC` and `FC` macro has to be set to the C and Fortran compiler you want to use, e.g. `cc` or `gcc` and `f77`. Finally, if you want C++[1] support set the `CPLUSPLUS` to `YES` and set the `CXX`macro.

Additionally on Sun workstations running SunOS 4.1.x you have to specify the version of your Sun Fortran compiler and set the `SUNF77`macro accordingly. On IBM RS 6000 workstations you have to set the `IBMF77` macro respectively.

Depending on the compiler you want to use, you have to check the `NXLDLIBS` marco in the architecture makefiles, e.g. `Makefile.SUN_OS5`. This macro specifies additional linker flags which are needed to link NXLib applications and the daemon process. The current setting works on all but the Solaris version with the vendor's compiler. On Solaris it works with `gcc`.

If you want to debug the NXLib daemons or application programs using the `adbg` and `ddbg` scripts you can set the `DEBUGGER` macro in the `Makefile.config` to your favourite debugger[2].

Once the `Makefile.config` has been edited it may be necessary to adapt the `config.h` and `errno.h` in the `include` directory.

In `config.h` the macros

- `RSHPATH`
- `RSHNAME`
- `START_PORT`
- `END_PORT`

should be checked and modified if necessary. In the supplied `errno.h` file the pathname of the system `errno.h` file must be checked and modified if the location is different on your system.

After all files have been edited and the macros have been set correctly the final step of the installation is to compile and install the libraries and binaries. This is done by a simple `make`[3]:

```
$ make
```

During that the installation directory gets populated with several subdirectories and all necessary libraries and binaries to run NXLib.

[1] Geoffrey Furnish supplied the C++ front-end. Due to the lack of a C++ compiler (and the knowhow) the front-end could not be tested.

[2] See pages 33 and 36 in appendix B.

[3] On Alpha AXP systems `make` didn't work. Use `gmake`.

2.4 Currently supported platforms

Table 2 gives an overview over all supported platforms.

Vendor	Type	Operating system
Sun	SPARC	SunOS 4.1.x
HP	9000	HP UX 09.01
SGI		IRIX 5.1.1.1
IBM	RS 6000	AIX 2.3
DEC	Alpha AXP	OSF 1.2
Sun	SPARC	Solaris 2. (SunOS 5)
	PCs	Linux 1.1.59[4]

Table 2: Supported platforms

3 Using NXLib

3.1 User installation

A prerequisite to compile and run NXLib applications for every user is the following:

- add the installation directory to your path to have access to all NXLib commands, e.g. PATH=$PATH:/home/joe_average/`NXLibV1_1_4`/bin or PATH=$PATH:/usr/local/`NXLibV1_1_4`/bin for `sh` or `ksh`.

Optionally if more than one version of NXLib should be supported a further environment variable can be specified to switch between the different versions. This can be done with the following commands:

- place the installation directory in the `NXLIBPATH` environment variable, e.g. `export NXLIBPATH=/home/joe_average/NXLibV1_1_4` or `export NXLIBPATH=/usr/local/NXLibV1_1_4`. If this variable is not set the value which has been inserted into the scripts during the installation process is used as a default.

To avoid coherency problems the following lines should be placed in the `.profile`[5]:

```
$ grep NXLIB .profile
# set environment variable for NXLIB installation directory
export NXLIBPATH=/usr/local/dist/\NXLIBDIR
PATH=$PATH:$NXLIBPATH/bin
```

Each time you decide to switch between the different versions first the value of the `NXLIBPATH` must be changed and then the `.profile` must be executed. This method guarantees that always the versions of the binaries and libraries correspond.

The implementation of NXLib currently uses the Berkeley `rsh` service to create remote daemons which are necessary to run an application on several machines in a network. Therefore it is necessary that the `.rhosts` allows remote command execution on every machine which should take part in the computation of an application.

Upon the start of an NXLib application the environment of the local machine[6] is transfered to all other machines. Therefore, it is necessary that the `PATH` environment

[4] This is a preliminary version for Linux. It could not be tested and might not work correctly.
[5] For `csh` or `tcsh` the corresponding syntax has to be used to set the environment variables.
[6] The local machine is the machine were an NXLib application is started.

variable of the local machine contains those directories on the remote machines where the NXLib executables[7] are located.

3.2 The Paragon

The nodes on a Paragon system are interconnected in a mesh topology and are subdivided into three partitions: the I/O partition, the service partition and the compute partition. Figure 1 shows a typical configuration of a Paragon system.

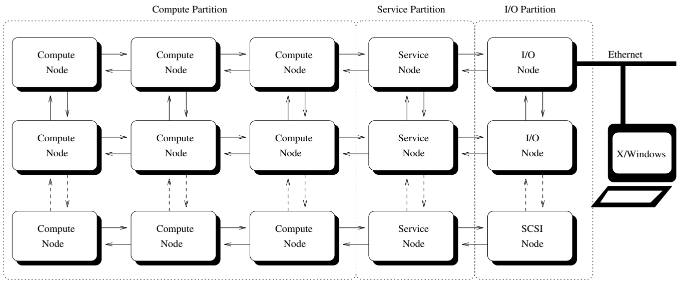

Figure 1: Different partitions in a Paragon system

Usually the largest partition in a configuration is the compute partition. Parallel user applications are executed on the nodes in this partition. In contrast to that, interactive processes, like shells, editors etc., are executed on the nodes in the service partition. Finally, the nodes in the I/O partition are used to connect I/O devices, like disks or local area networks, to the machine. In addition to the harware partitions it is also possible to create software partitions in the compute partition of a Paragon. Therefore, the Paragon operating systems offers calls to create, list and delete partitions.

Parallel applications on a Paragon consist of two parts: the controlling process on the service partition and the application processes on the compute partition. The application processes mainly do the computation and use Paragon NX message passing calls to exchange data. The controlling process is mainly used during the startup of the application and to redirect `stdin` and `stdout`.

A Paragon application is started on a partition. The partition can be selected with a commandline parameter (`-pn` *partition name*) which is specified upon the invocation of the application. Within that partition several nodes can be selected to start application processes. Again, command line paramters (`-sz` *size* or `-on` *node number*) are used to select the nodes.

During an application is running the processes can exchange data between each other with the NX message-passing calls. To address each other the pocesses use a node number and a *ptype*. Any process on a node can modify its own *ptype* during run-time. If a *ptype* value has been used from a process no other process can set its *ptype* to that value. Therefore, different processes on the same node cannot have the same *ptype* value and the pair consisting of a node number and a *ptype* can be used as an unique address. Upon the startup of an application the *ptype* value can be set with the `-pt` command line switch.

3.3 Creating a NXLib application

On Paragon systems the icc and if77 compiler are used to create parallel applications. The `-nx` and `-lnx` linker switch have to be used to link such an application. To minimize the effort to bring existing Paragon applications onto a network of workstations or vice versa a similar front end has been provided for the workstations. The `vpcc`[8], `vpCC`[9], and

[7] These are the NXLib binaries, such as the daemon, and the user programs.
[8] See page 39 in appendix B
[9] See page 40 in appendix B

`vpf77`[10] scripts can be used to compile and link parallel applications using NXLib calls for networks of workstations. The following example shows how a simple NXLib application which consists of one C source file is compiled and linked:

```
$ ls
NxlibSimpleDemo.c
$ vpcc -o NxlibSimpleDemo NxlibSimpleDemo.c -nx
$ ls
NxlibSimpleDemo             NxlibSimpleDemo.c
```

The above example uses the `-nx` switch to enable automatic startup, i.e. the processes to run the application are automatically created during the start up. If the `-lnx` switch was used to link the application the user would have to call the `nx_init` and `nx_loadve` routines in the controlling process of his application to start all application processes[11].

3.4 Partitions on workstations

As already introduced in section 3.2 the Paragon operating systems supports software partitions. In contrast to the Paragon where operating system commands are available to define, list or delete partitions NXLib does not support these commands. Instead files containing mapping tables are used to simulate these partitions. The names of the files may be used as a parameter to the `-pn` switch upon the invocation of an application. Figure 2 on page 6 shows how the mapping tables have to be specified.

Every *VirtualParagonNode* definition consists of at least a node number and a workstation name. The *NodeNumber* are positive integer numbers including zero. Contiguous intervals starting from zero must be used to define a partition. The *Workstation* is the symbolic name of a workstation. If a path is specified for a *VirtualParagonNode* the processes on this node will run in the specified directory. It is possible to use environment variables like `$PWD`. Beware that the environment variables of the workstation where you started the program are used. Additionally a *Login* can be specified if the login on that machine differs from the other one's. Finally, the *Arch* may be used to define the architecture of a machine. Currently this field may be specified but it is not yet used.

The following example shows a mapping table which defines a four node partition on three workstations:

```
# Comments may be used like in shell scripts
# NodeNumber Workstation                  Login Arch
  0          sun27
  1          sun08:$PWD                         % sun4
  2          sun66:/home/hoss/blabla      hoss  % sun4
  3          sun66                        lambo
```

As already explained in section 3.2 the `-sz` and `-on` switch may be used to select some of the nodes within a partition to run application processes. With the mapping table as the partition definition there is no change necessary to those switches.

---
[10] See page 41 in appendix B
[11] For a further discussion refer to [7]

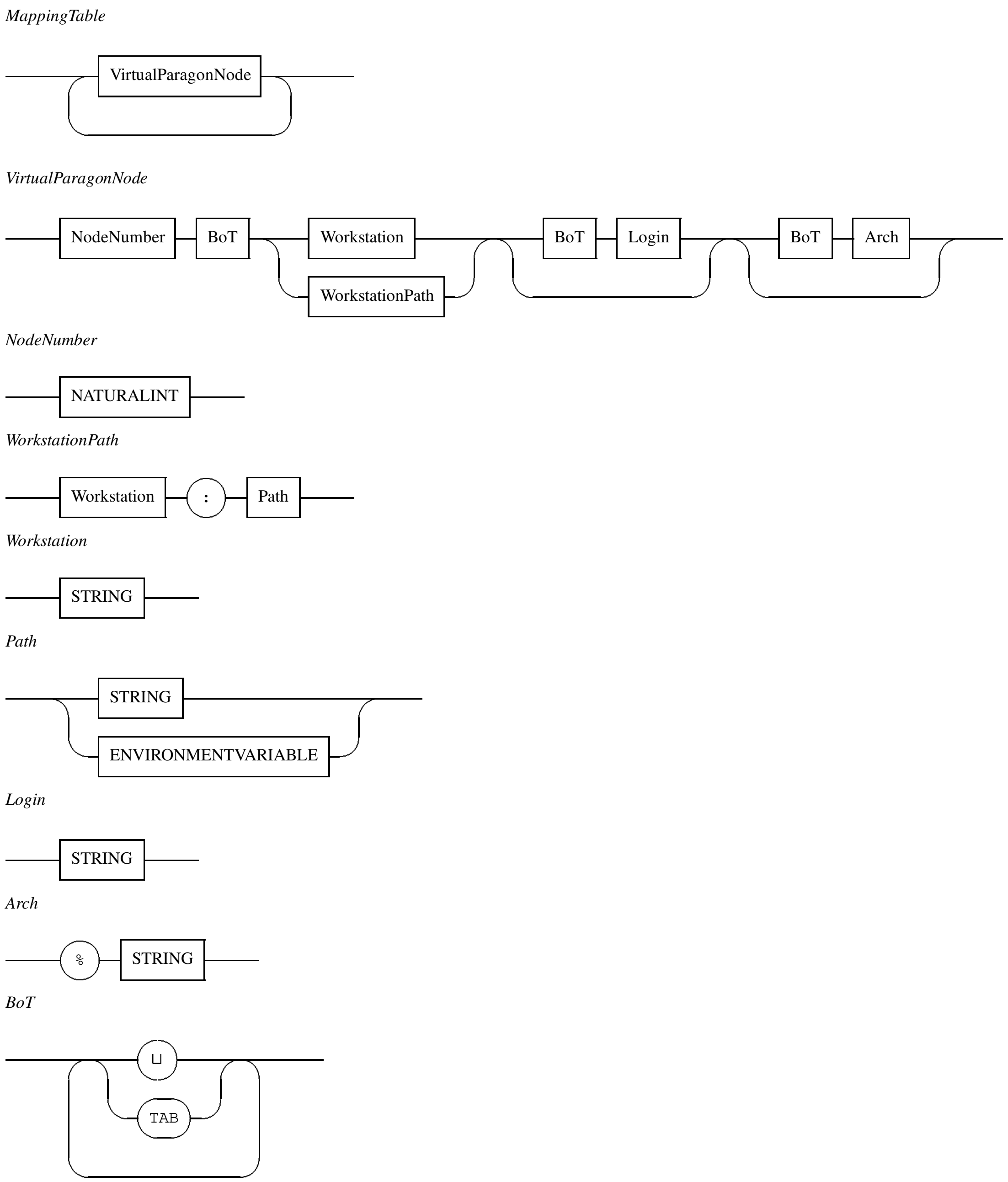

Figure 2: Syntax diagram for the mapping table

3.5 Starting a NXLib application

The invocation of an NXLib application is similar to the Paragon. It is sufficient to type the name of the application which has been created by the `vpcc`, `vpCC` or `vpf77` command.

Supported switches	`-pn`, `-pt`, `-sz`, `-on`, `\;`
Unsupported switches	`-pri`, `-pkt`, `-mbf`, `-mex`, `-mem`, `-sth`, `-sct`, `-gth`, `-plk`
Additional switches	`-tcpbuf`

Table 3: Supported and unsupported command line switches for NXLib applications

Table 3 list which Paragon command line switches are supported for NXLib applications[12]. Like on a Paragon the environment variables `NX_DFLT_PART` and `NX_DFLT_SIZE` are used if no partition name or partition size is specified on the command line of the application. If theese variables are not set the `.compute` partition file as a whole is used as the default[13]. The following examples show some possible invocations of NXLib applications:

```
$ ls -a
.compute                       My1stPartition
NxlibSimpleDemo                NxlibSimpleDemo.c
$ echo $NX_DFLT_PART

$ NxlibSimpleDemo
Hello from node 2!
Hello from node 0!
Hello from node 1!
$ NxlibSimpleDemo -pn My1stPartition
Hello from node 1!
Hello from node 0!
Hello from node 3!
Hello from node 2!
$ export NX_DFLT_PART=My1stPartition
$ echo $NX_DFLT_PART
My1stPartition
$ NxlibSimpleDemo
Hello from node 3!
Hello from node 2!
Hello from node 0!
Hello from node 1!
$ NxlibSimpleDemo -sz 1
Hello from node 0!
$
```

3.6 Accessing files

Files can currently be only accessed with standard library functions. The calls to support the parallel file system of a Paragon system are not yet supplied. File names are interpreted differently on the local and the remote machines. On the local machine the names are interpreted from the current working directory, i.e. the directory where the application is

[12] See page 34 in appendix B.
[13] Therefore a file with the name .compute must exist in the working directory.

started. On the remote machines the names are interpreted from the home directory if no path is specified in the partition file.

4 Commands Overview

The following sections provide an overview over the commands which are available within the NXLib environment. Calls which are additionally available will be explained in section 4.1. Finally section 4.2 will show differences between the Paragon environment and NXLib.

Appendix A will give a list of NX message-passing calls which are available within NXLib. In [5, 6, 4, 7] an intensive discussion is given. A reference to the commands is provided in appendix B.

4.1 Additional user commands

4.1.1 The compiler front-ends

The C, C++, and Fortran compilers on a Paragon use two special linker switches to generate parallel application. The -nx and -lnx switch both provide the NX library calls. The difference is that linking with the -nx switch allows for automatic application startup. In order to keep the changes which are necessary to migrate an application from a Paragon to a network of workstations minimal two compiler front-ends for the workstation compilers have been provided. Usually it is sufficient to substitute the `CC` the `CXX` or the `FC` macro in the application `Makefile` with the appropriate compiler. For a further discussion of the C compiler front-end `vpcc` refer to page 39, the C++ compiler front-end `vpCC` on page 40, and the Fortran compiler front-end `vpf77` on page 41 in appendix B.

4.1.2 The `nxdaemon`

The `nxdaemon` is used to model properties of a Paragon node on a workstation. It is therefore the basic unit of a virtual Paragon node. During the startup of an NXLib application the `nxdaemon` is started via a rsh command on the appropriate machine. For every node which was specified in the partition file exactly one `nxdaemon` is started. The daemons in turn start the application processes for that virtual node.

After the startup of an application the daemons are used for managing the application processes and to provide address information about other application processes. Signals which are sent to the controlling process are forwarded to all application processes if appropriate, e.g. a whole application can be stopped by sending a corresponding signal to the controlling process. As not all application processes are fully interconnected an initial communication request establishes an interconnection between application processes. These links are then used for the rest of the lifetime of the application. To get address information about non-local application processes the `nxdaemon` is responsible to gather the necessary information from remote `nxdaemons`. Again appendix B provides on page 37 a detailed discussion of the `nxdaemon`.

4.1.3 Reading `stdin` with `nxlstdin`

The `nxlstdin` shell script provides reading from stdin for NXLib applications. Input form stdin is limited to processes which run on the host where the application was started. Page 38 in appendix B provides more information about `nxlstdin`.

4.1.4 Debugging NXLib with `adbg` and `ddbg`

The `adbg` and `ddbg` shell scripts provide a simple front-end to invoke a debugger. Both commands connect a debugger to a running process. The `ddbg` is merely used by system programmers to debug the `nxdaemon` which was described in section 4.1.2. The `adbg` command allows for debugging application processes. The debugger which is used can be set during the installation of NXLib. As described in section 2.3 the `DEBUGGER` macro in the `Makefile` is used for this purpose. Page 33 and page 36 in appendix B provide more information about `adbg` and `ddbg` respectively.

4.2 Workstation specific changes and restrictions

Although great care has been taken to keep the changes to the Paragon environment as minimal as possible there are nevertheless several points were changes and restrictions were necessary. This section provides an overview on these points.

In contrast to the Paragon where a distributed operating system is used the workstations all have independent operating systems. Thus, the single system image which is provided on a Paragon is not fully available within NXLib, e.g. the process identifiers are not transparent to every node, gang scheduling and priorities are not supported. Modifications of the operating system on every machine would have been necessary, to implement these features.

Due to different hardware units, features like the partition management and the configuration of the buffers used for messages have been neglected or are available in a different manner. An implementation of these features would require similar hardware as on a Paragon.

Finally several facilities of a Paragon system were left out during the implementation as only limited man power was available. These are the reactive kernel interface, the iPSC/860 compatibility calls, support for parallel I/O and the nx_nfork call.

5 Troubleshooting

This section contains a few hints on what might be wrong if your application doesn't run as it should. It is very informal and should be seen as a "frequently asked questions" section.

Q: Application processes can not read from `stdin`. What can I do?

A: Use the `nxlstdin` shell script, which supports reading from `stdin` for local application processes. See page 38 in appendix B for more information about `nxlstdin`.

Q: If I try to start an application on a remote machine I get an error message like

```
ksh: nxdaemon: not found.
(node 2, ptype C): nxdaemon died before initialization
(node 2, ptype C) ./hello: allocator internal error
```

A: The nxdaemon executable can not be found in the `PATH` on the remote machine. Consider that `rsh` does not execute `.profile` or `.login`.

A workaround is to set the `NXLIBPATH` environment variable. If it is set the nxdaemon will be started with the full pathname.

Q: I use malloc() or related calls in my application program and sometimes I get memory errors.

A: NXLib uses dynamic memory allocation in its buffer layer. If a message arrives during a malloc call in the application the memory allocation might get corrupted.

Include <nxmalloc.h> which redefines malloc and related calls in order to turn off incoming messages during memory allocation.

Q: My file I/O doesn't work.

A: Beware that the `rsh` command executes your application in the home directory on the remote machine if no path is specified in the partition file. If your file is located in a different directory the `open` call will fail. Add a pathname in you partition file. This causes NXLib to change to the directory you specified.

Q: I specified a directory in the partition file but it doesn't work.

A: The algorithm to determine pathnames is not very sophisticated. You can use only simple pathnames. Even some special characters which are valid in pathname might not work. This is a bug in NXLib and should be corrected. Combinations of environment variables as parts of a pathname are not valid, too. A pathname may be either an environment variable or a pathname.

Q: I tried to compile a Fortran program <u>without</u> a PROGRAM statement. That doesn't work.

A: `vpf77` requires a PROGRAM statement in your Fortran code in order to determine the entry point to the application program.

Q: My applications blocks sometimes in an exchange communication.

A: Set the `-tcpbuf` command line switch to a different value. NXLib uses the TCP buffer to avoid blocking in exchange messages. If this buffer is large enough the message will be buffered in the TCP layer.

Q: I'm using NXLib with C++. When I link the application I get an error message like

```
ld:
Unresolved:
main_nxlib_
```

A: Use the `vpCC` compiler front-end (See page 40 in appendix B). Make sure `CPLUSPLUS` was set to `YES` while NXLib was made.

Before Geoffery Furnish supplied the `vpCC` script Chris Elford had the following solution.

Replace your C++ main() with the following:

```
extern "C" main_nxlib_();
int main_nxlib_() ...
```

-Dmain=XXX simply changes main() into main_nxlib_() and does not give the routine external "C" linkage. We do this by hand in our C++/NXLib codes by making the substitution above.

This *should* fix your problem. However, you *may* run into the following problem with other link symbols. Our version of C++ wanted to compile main() itself because main() had to have an odd demangling style under our compiler. NXLib keeps its main() (which calls the application main_nxlib_()) We ended up copying the main_lnx.c, main_nx.c (the files that contain nxlib main()) and compiling them under C++. These two files are not contained in libnx.a but are separately linked by vpcc and vpf77 so you won't end up with duplicate symbols. I would not advise that you do this *unless* you run into problems with additional missing symbols.

References

[1] Ralph Butler and Ewing Lusk. *User' s Guide to the P4 Programming System*. Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439-4801, October 1992.

[2] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3.0 User' s Guide and Reference Manual*. Oak Ridge National Laboratory, Oak Ridge, TN 37831-8083, 1. edition, May 1993.

[3] Rolf Hempel. *The ANL/GMD Macro' s (PARMACS) in FORTRAN for Portable Parallel Programming using the Message Passing Model — User' s Guide and Reference Manual*. Gesellschaft für Mathematik und Datenverarbeitung mbH, Postfach 1240, W5205 St. Augustin, 5.1 edition, November 1991.

[4] Intel Supercomputer System's Division, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006. *Paragon OSF/1 C Commands Reference Manual*, 1 edition, April 1993. www: http://www.ccsf.caltech.edu/paragon/man.html.

[5] Intel Supercomputer System's Division, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006. *Paragon OSF/1 C System Calls Reference Manual*, 1 edition, April 1993. www: http://www.ccsf.caltech.edu/paragon/man.html.

[6] Intel Supercomputer System's Division, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006. *Paragon OSF/1 Fortran System Calls Reference Manual*, 1 edition, April 1993. www: http://www.ccsf.caltech.edu/paragon/man.html.

[7] Intel Supercomputer System's Division, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006. *Paragon OSF/1 User's Guide*, 1 edition, April 1993. www: http://www.ccsf.caltech.edu/paragon/man.html.

[8] Stefan Lamberts, Georg Stellner, Arndt Bode, and Thomas Ludwig. Paragon parallel programming environment on Sun workstations. In *Sun User Group Proceedings*, pages 87–98. Sun User Group, December 1993. ftp: ftpbode.informatik.tu-muenchen.de:NXLIB/PAPERS/SUG93.ps.Z.

[9] ParaSoft Corporation, 2500 E. Foothill Blvd., Pasadena CA 91107. *NetExpress — Introductory Guide*, 3.2 edition, 1992.

[10] G. Stellner, A. Bode, S. Lamberts, and T. Ludwig. Developing applications for multicomputer systems on workstation clusters. In *HPCN Europe, The International Conference and Exhibition on High-Performance Computing and Networking*, Munich, 1994. ftp: ftpbode.informatik.tu-muenchen.de:NXLIB/PAPERS/HPCN94.ps.Z.

[11] G. Stellner, A. Bode, S. Lamberts, and T. Ludwig. NXLib — a parallel programming environment for workstation clusters. In *PARLE'94, Parallel Architectures and Languages Europe*, Athens, 1994. ftp: ftpbode.informatik.tu-muenchen.de:NXLIB/PAPERS/Parle94.ps.Z.

[12] G. Stellner, M. Schumann, S. Lamberts, T. Ludwig, A. Bode, M. Kiehl, and R. Mehlhorn. Developing multicomputer applications on networks of workstations using NXLib. SFB-Bericht 342/17/93 A, Technische Universität München, 80290 München, December 1993.

A Summary of Commands and System Calls

This appendix summarizes the commands and library calls offered by NXLib. The complete syntax of each command and call is provided, along with a brief description of each. The C and Fortran versions of the calls are discussed in separate sections.

A.1 Command Summary

A.1.1 Compiling and Linking Applications

Commands for Compiling and Linking Applications

Command Synopsis	Description
vpcc -nx | -lnx [*switches*] *sourcefile...*	Compile a NXLib application written in C.
vpf77 -nx | -lnx [*switches*] *sourcefile...*	Compile a NXLib application written in Fortran.

A.1.2 Running Applications

Commands for Running Applications

Command Synopsis	Description
application [**-sz** *size*] [**-pt** *ptype*] [**-on** *nodespec*] [**-pn** *partition*] [**-tcpbuf** *buffersize*] [*application_args*] [\; *file* [**-pt** *ptype*] [**-on** *nodespec*] [*application_args*]] ...	Execute a NXLib application.

A.2 C System Call Summary

This section summarizes the C versions of the library calls.

A.2.1 Process Characteristics

C Calls for Process Characteristics

Synopsis	Description
long **mynode**(void);	Obtain the calling process' node number.
long **numnodes**(void);	Obtain the number of nodes allocated to the current application.
void **setptype**(long ptype);	Set the calling process' process type.
long **myptype**(void);	Obtain the calling process' process type.
long **myhost**(void);	Obtain the controlling process' node number.

A.2.2 Synchronous Send and Receive

C Calls for Synchronous Send and Receive

Synopsis	Description
void **csend**(long *type*, char **buf*, long *count*, long *node*, long *ptype*);	Send a message, waiting for completion.
void **crecv**(long *typesel*, char **buf*, long *count*);	Receive a message, waiting for completion.
long **csendrecv**(long *type*, char **sbuf*, long *scount*, long *node*, long *ptype*, long *typesel*, char **rbuf*, long *rcount*);	Send a message and post a receive for the reply. Wait for completion.
void **gsendx**(long *type*, char **buf*, long *count*, long *nodes[]*, long *nodecount*);	Send a message to a list of nodes, waiting for completion.

A.2.3 Asynchronous Send and Receive

C Calls for Asynchronous Send and Receive

Synopsis	Description
long **isend**(long *type*, char **buf*, long *count*, long *node*, long *ptype*);	Send a message without waiting for completion.
long **irecv**(long *typesel*, char **buf*, long *count*);	Receive a message without waiting for completion.
long **isendrecv**(long *type*, char **sbuf*, long *scount*, long *node*, long *ptype*, long *typesel*, char **rbuf*, long *rcount*);	Send a message and post a receive for the reply without waiting for completion.
long **msgdone**(long *mid*);	Determine whether a send or receive operation has completed.
void **msgwait**(long *mid*);	Wait for completion of a send or receive operation.
void **msgignore**(long *mid*);	Release a message ID as soon as a send or receive operation completes.
long **msgmerge**(long *mid1*, long *mid2*);	Merge two message IDs into a single ID that can be used to wait for completion of both operations.

A.2.4 Probing for Pending Messages

C Calls for Probing for Pending Messages

Synopsis	Description
void **cprobe**(long *typesel*);	Wait for a message of a selected type to arrive.
long **iprobe**(long *typesel*);	Determine whether a message of a selected type is pending.

A.2.5 Getting Information About Pending or Received Messages

C Calls for Getting Information About Pending or Received Messages

Synopsis	Description
long **infocount**(void);	Return size in bytes of a pending or received message.
long **infonode**(void);	Return node number of the node that sent a pending or received message.
long **infoptype**(void);	Return process type of the process that sent a pending or received message.
long **infotype**(void);	Return message type of a pending or received message.

A.2.6 Flushing and Canceling Messages

C Calls for Flushing and Canceling Messages

Synopsis	Description
void **flushmsg**(long *typesel*, long *nodesel*, long *ptypesel*);	Flush specified messages from the system.
void **msgcancel**(long *mid*);	Cancel an asynchronous send or receive operation.

A.2.7 Treating a Message as an Interrupt

C Calls for Treating a Message as an Interrupt

Synopsis	Description
void **hsend**(long *type*, char **buf*, long *count*, long *node*, long *ptype*, void (**handler*)());	Send a message and set up a handler procedure to be called when the send completes.
void **hrecv**(long *typesel*, char **buf*, long *count*, void (**handler*)());	Receive a message and set up a handler procedure to be called when the receive completes.
void **hsendrecv**(long *type*, char **sbuf*, long *scount*, long *node*, long *ptype*, long *typesel*, char **rbuf*, long *rcount*, void (**handler*)());	Send a message and post a receive for the reply. Set up a handler procedure to be called when the reply arrives.
long **masktrap**(long *state*);	Enable or disable interrupts for message handlers.
void **hsendx**(long *type*, char **buf*, long *count*, long *node*, long *ptype*, void (**xhandler*)(), long *hparam*);	Send a message and set up an extended handler procedure to be called with the value *hparam* when the reply arrives.

A.2.8 Extended Receive and Probe

C Calls for Extended Receive and Probe

Synopsis	Description
void **crecvx**(long *typesel*, char **buf*, long *count*, long *nodesel*, long *ptypesel*, long *info*[]);	Receive a message of a specified type from a specified sending node and process type, together with information about the message. Wait for completion.
long **irecvx**(long *typesel*, char **buf*, long *count*, long *nodesel*, long *ptypesel*, long *info*[]);	Receive a message of a specified type from a specified sending node and process type, together with information about the message. Do not wait for completion.
void **hrecvx**(long *typesel*, char **buf*, long *count*, long *nodesel*, long *ptypesel*, void (**xhandler*)(), long *hparam*);	Receive a message of a specified type from a specified sending node and process type. Set up an extended handler procedure to be called with information about the message and the value *hparam* when the receive completes.
void **cprobex**(long *typesel*, long *nodesel*, long *ptypesel*, long *info*[]);	Wait for a message of a specified type from a specified sending node and process type. Return information about the message.
long **iprobex**(long *typesel*, long *nodesel*, long *ptypesel*, long *info*[]);	Determine whether a message of a specified type from a specified sending node and process type is pending. If it is, return information about the message.

A.2.9 Global Operations

C Calls for Global Operations (1 of 2)

Synopsis	Description
void **gcol**(char *x*[], long *xlen*, char *y*[], long *ylen*, long **ncnt*);	Concatenation.
void **gcolx**(char *x*[], long *xlens*[], char *y*[]);	Concatenation for contributions of known length.
void **gdhigh**(double *x*[], long *n*, double *work*[]);	Vector double precision MAX.
void **gdlow**(double *x*[], long *n*, double *work*[]);	Vector double precision MIN.
void **gdprod**(double *x*[], long *n*, double *work*[]);	Vector double precision MULTIPLY.
void **gdsum**(double *x*[], long *n*, double *work*[]);	Vector double precision SUM.
void **giand**(long *x*[], long *n*, long *work*[]);	Vector integer bitwise AND.
void **gihigh**(long *x*[], long *n*, long *work*[]);	Vector integer MAX.
void **gilow**(long *x*[], long *n*, long *work*[]);	Vector integer MIN.

C Calls for Global Operations (2 of 2)

Synopsis	Description
void **gior**(long *x*[], long *n*, long *work*[]);	Vector integer bitwise OR.
void **giprod**(long *x*[], long *n*, long *work*[]);	Vector integer MULTIPLY.
void **gisum**(long *x*[], long *n*, long *work*[]);	Vector integer SUM.
void **gland**(long *x*[], long *n*, long *work*[]);	Vector logical AND.
void **glor**(long *x*[], long *n*, long *work*[]);	Vector logical inclusive OR.
void **gopf**(char *x*[], long *xlen*, char *work*[], long (**function*)());	Arbitrary commutative function.
void **gshigh**(float *x*[], long *n*, float *work*[]);	Vector real MAX.
void **gslow**(float *x*[], long *n*, float *work*[]);	Vector real MIN.
void **gsprod**(float *x*[], long *n*, float *work*[]);	Vector real MULTIPLY.
void **gssum**(float *x*[], long *n*, float *work*[]);	Vector real SUM.
void **gsync**(void);	Global synchronization.

A.2.10 Controlling Application Execution

C Calls for Controlling Application Execution

Synopsis	Description
long **nx_initve**(char **partition*, long *size*, char **account*, long **argc*, char **argv*[]);	Create a new application.
long **nx_load**(long *node_list*[], long *numnodes*, long *ptype*, long *pid_list*[], char **pathname*);	Execute a stored program on some or all nodes of an application.
long **nx_loadve**(long *node_list*[], long *numnodes*, long *ptype*, long *pid_list*[], char **pathname*, char **argv*[], char **envp*[]);	Execute a stored program on some or all nodes of an application, with specified argument list and environment.
long **nx_waitall**(void);	Wait for all application processes.

A.2.11 Handling Errors

C Calls for Handling Errors

Synopsis	Description
_call(...);	Special version of *call* that returns error value to caller.
void **nx_perror**(char **string*);	Print an error message corresponding to the current value of *errno*.

A.2.12 Miscellaneous Calls

Miscellaneous C Calls

Synopsis	Description
void **flick**(void);	No effect.
void **led**(long *state*);	No effect.
double **dclock**(void);	Return time in seconds since January 1st 1970.

A.2.13 Performing Extended Arithmetic

C Calls for Performing Extended Arithmetic

Synopsis	Description
esize_t **eadd**(esize_t *e1*, esize_t *e2*);	Add two extended numbers.
long **ecmp**(esize_t *e1*, esize_t *e2*);	Compare two extended numbers.
long **ediv**(esize_t *e*, long *n*);	Divide extended number by integer.
long **emod**(esize_t *e*, long *n*);	Give extended number modulo an integer.
esize_t **emul**(esize_t *e*, long *n*);	Multiply extended number by integer.
esize_t **esub**(esize_t *e1*, esize_t *e2*);	Subtract two extended numbers.
void **etos**(esize_t *e*, char **s*);	Convert extended number to string.
esize_t **stoe**(char **s*);	Convert string to extended number.

A.3 Fortran System Call Summary

This section summarizes the Fortran versions of the library calls.

A.3.1 Process Characteristics

Fortran Calls for Process Characteristics

Synopsis	Description
INTEGER FUNCTION **MYNODE**()	Obtain the calling process' node number.
INTEGER FUNCTION **NUMNODES**()	Obtain the number of nodes allocated to the current application.
SUBROUTINE **SETPTYPE**(ptype) INTEGER *ptype*	Set the calling process' process type.
INTEGER FUNCTION **MYPTYPE**()	Obtain the calling process' process type.
INTEGER FUNCTION **MYHOST**()	Obtain the controlling process' node number.

A.3.2 Synchronous Send and Receive

Fortran Calls for Synchronous Send and Receive

Synopsis	Description
SUBROUTINE **CSEND**(type, buf, count, node, ptype) INTEGER *type* INTEGER *buf*(*) INTEGER *count* INTEGER *node* INTEGER *ptype*	Send a message, waiting for completion.
SUBROUTINE **CRECV**(typesel, buf, count) INTEGER *typesel* INTEGER *buf*(*) INTEGER *count*	Receive a message, waiting for completion.
INTEGER FUNCTION **CSENDRECV**(type, sbuf, scount, node, ptype, typesel, rbuf, rcount) INTEGER *type* INTEGER *sbuf*(*) INTEGER *scount* INTEGER *node* INTEGER *ptype* INTEGER *typesel* INTEGER *rbuf*(*) INTEGER *rcount*	Send a message and post a receive for the reply. Wait for completion.
SUBROUTINE **GSENDX**(type, buf, count, nodes, nodecount) INTEGER *type* INTEGER *buf*(*) INTEGER *count* INTEGER *nodes*(*) INTEGER *nodecount*	Send a message to a list of nodes, waiting for completion.

A.3.3 Asynchronous Send and Receive

Fortran Calls for Asynchronous Send and Receive

Synopsis	Description
INTEGER FUNCTION **ISEND**(type, buf, count, node, ptype) INTEGER *type* INTEGER *buf*(*) INTEGER *count* INTEGER *node* INTEGER *ptype*	Send a message without waiting for completion.
INTEGER FUNCTION **IRECV** (typesel, buf, count) INTEGER *typesel* INTEGER *buf*(*) INTEGER *count*	Receive a message without waiting for completion.
INTEGER FUNCTION **ISENDRECV**(type, sbuf, scount, node, ptype, typesel, rbuf, rcount) INTEGER *type* INTEGER *sbuf*(*) INTEGER *scount* INTEGER *node* INTEGER *ptype* INTEGER *typesel* INTEGER *rbuf*(*) INTEGER *rcount*	Send a message and post a receive for the reply without waiting for completion.
INTEGER FUNCTION **MSGDONE**(mid) INTEGER *mid*	Determine whether a send or receive operation has completed.
SUBROUTINE **MSGWAIT**(mid) INTEGER *mid*	Wait for completion of a send or receive operation.
SUBROUTINE **MSGIGNORE**(mid) INTEGER *mid*	Release a message ID as soon as a send or receive operation completes.
INTEGER FUNCTION **MSGMERGE**(mid1, mid2) INTEGER *mid1* INTEGER *mid2*	Merge two message IDs into a single ID that can be used to wait for completion of both operations.

A.3.4 Probing for Pending Messages

Fortran Calls for Probing for Pending Messages

Synopsis	Description
SUBROUTINE **CPROBE**(typesel) INTEGER *typesel*	Wait for a message of a selected type to arrive.
INTEGER FUNCTION **IPROBE** (typesel) INTEGER *typesel*	Determine whether a message of a selected type is pending.

A.3.5 Getting Information About Pending or Received Messages

Fortran Calls for Getting Information About Pending or Received Messages

Synopsis	Description
INTEGER FUNCTION **INFOCOUNT**()	Return size in bytes of a pending or received message.
INTEGER FUNCTION **INFONODE**()	Return node number of the node that sent a pending or received message.
INTEGER FUNCTION **INFOPTYPE**()	Return process type of the process that sent a pending or received message.
INTEGER FUNCTION **INFOTYPE**()	Return message type of a pending or received message.

A.3.6 Flushing and Canceling Messages

Fortran Calls for Flushing and Canceling Messages

Synopsis	Description
SUBROUTINE **FLUSHMSG**(typesel, nodesel, ptypesel) INTEGER *typesel* INTEGER *nodesel* INTEGER *ptypesel*	Flush specified messages from the system.
SUBROUTINE **MSGCANCEL**(mid) INTEGER *mid*	Cancel an asynchronous send or receive operation.

A.3.7 Treating a Message as an Interrupt

Fortran Calls for Treating a Message as an Interrupt (1 of 2)

Synopsis	Description
SUBROUTINE **HSEND**(type, buf, count, node, ptype, handler) INTEGER *type* INTEGER *buf*(*) INTEGER *count* INTEGER *node* INTEGER *ptype* EXTERNAL *handler*	Send a message and set up a handler procedure to be called when the send completes.
SUBROUTINE **HRECV**(typesel, buf, count, handler) INTEGER *typesel* INTEGER *buf*(*) INTEGER *count* EXTERNAL *handler*	Receive a message and set up a handler procedure to be called when the receive completes.
SUBROUTINE **HSENDRECV**(type, sbuf, scount, node, ptype, typesel, rbuf, rcount, handler) INTEGER *type* INTEGER *sbuf*(*) INTEGER *scount* INTEGER *node* INTEGER *ptype* INTEGER *typesel* INTEGER *rbuf*(*) INTEGER *rcount* EXTERNAL *handler*	Send a message and post a receive for the reply. Set up a handler procedure to be called when the reply arrives.
INTEGER FUNCTION **MASKTRAP**(state) INTEGER *state*	Enable or disable interrupts for message handlers.

Fortran Calls for Treating a Message as an Interrupt (2 of 2)

Synopsis	Description
SUBROUTINE **HSENDX**(type, buf, count, node, ptype, xhandler, hparam) INTEGER *type* INTEGER *buf*(*) INTEGER *count* INTEGER *node* INTEGER *ptype* EXTERNAL *xhandler* INTEGER *hparam*	Send a message and set up an extended handler procedure to be called with the value *hparam* when the reply arrives.

A.3.8 Extended Receive and Probe

Fortran Calls for Extended Receive and Probe (1 of 2)

Synopsis	Description
SUBROUTINE **CRECVX**(typesel, buf, count, nodesel, ptypesel, info) INTEGER *typesel* INTEGER *buf*(*) INTEGER *count* INTEGER *nodesel* INTEGER *ptypesel* INTEGER *info*(8)	Receive a message of a specified type from a specified sending node and process type, together with information about the message. Wait for completion.
INTEGER FUNCTION **IRECVX**(typesel, buf, count, nodesel, ptypesel, info) INTEGER *typesel* INTEGER *buf*(*) INTEGER *count* INTEGER *nodesel* INTEGER *ptypesel* INTEGER *info*(8)	Receive a message of a specified type from a specified sending node and process type, together with information about the message. Do not wait for completion.

Fortran Calls for Extended Receive and Probe (2 of 2)

Synopsis	Description
SUBROUTINE **HRECVX**(typesel, buf, count, nodesel, ptypesel, xhandler, hparam) INTEGER *typesel* INTEGER *buf*(*) INTEGER *count* INTEGER *nodesel* INTEGER *ptypesel* EXTERNAL *xhandler* INTEGER *hparam*	Receive a message of a specified type from a specified sending node and process type. Set up an extended handler procedure to be called with information about the message and the value *hparam* when the receive completes.
SUBROUTINE **CPROBEX**(typesel, nodesel, ptypesel, info) INTEGER *typesel* INTEGER *nodesel* INTEGER *ptypesel* INTEGER *info*(8)	Wait for a message of a specified type from a specified sending node and process type. Return information about the message.
INTEGER FUNCTION **IPROBEX**(typesel, nodesel, ptypesel, info) INTEGER *typesel* INTEGER *nodesel* INTEGER *ptypesel* INTEGER *info*(8)	Determine whether a message of a specified type from a specified sending node and process type is pending. If it is, return information about the message.

A.3.9 Global Operations

Fortran Calls for Global Operations (1 of 3)

Synopsis	Description
SUBROUTINE **GCOL**(x, xlen, y, ylen, ncnt) INTEGER *x*(*) INTEGER *xlen* INTEGER *y*(*) INTEGER *ylen* INTEGER *ncnt*	Concatenation.
SUBROUTINE **GCOLX**(x, xlens, y) INTEGER *x*(*) INTEGER *xlens*(*) INTEGER *y*(*)	Concatenation for contributions of known length.
SUBROUTINE **GDHIGH**(x, n, work) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Vector double precision MAX.
SUBROUTINE **GDLOW**(x, n, work) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Vector double precision MIN.
SUBROUTINE **GDPROD**(x, n, work) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Vector double precision MULTIPLY.
SUBROUTINE **GDSUM**(x, n, work) DOUBLE PRECISION *x*(*) INTEGER *n* DOUBLE PRECISION *work*(*)	Vector double precision SUM.
SUBROUTINE **GIAND**(x, n, work) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Vector integer bitwise AND.

Fortran Calls for Global Operations (2 of 3)

Synopsis	Description
SUBROUTINE **GIHIGH**(x, n, work) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Vector integer MAX.
SUBROUTINE **GILOW**(x, n, work) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Vector integer MIN.
SUBROUTINE **GIOR**(x, n, work) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Vector integer bitwise OR.
SUBROUTINE **GIPROD**(x, n, work) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Vector integer MULTIPLY.
SUBROUTINE **GISUM**(x, n, work) INTEGER *x*(*) INTEGER *n* INTEGER *work*(*)	Vector integer SUM.
SUBROUTINE **GLAND**(x, n, work) LOGICAL *x*(*) INTEGER *n* LOGICAL *work*(*)	Vector logical AND.
SUBROUTINE **GLOR**(x, n, work) LOGICAL *x*(*) INTEGER *n* LOGICAL *work*(*)	Vector logical inclusive OR.
SUBROUTINE **GOPF**(x, xlen, work, function) INTEGER *x*(*) INTEGER *xlen* INTEGER *work*(*) EXTERNAL *function*	Arbitrary commutative function.

Fortran Calls for Global Operations (3 of 3)

Synopsis	Description
SUBROUTINE **GSHIGH**(x, n, work) REAL *x*(*) INTEGER *n* REAL *work*(*)	Vector real MAX.
SUBROUTINE **GSLOW**(x, n, work) REAL *x*(*) INTEGER *n* REAL *work*(*)	Vector real MIN.
SUBROUTINE **GSPROD**(x, n, work) REAL *x*(*) INTEGER *n* REAL *work*(*)	Vector real MULTIPLY.
SUBROUTINE **GSSUM**(x, n, work) REAL *x*(*) INTEGER *n* REAL *work*(*)	Vector real SUM.
SUBROUTINE **GSYNC**()	Global synchronization.

A.3.10 Controlling Application Execution

Fortran Calls for Controlling Application Execution

Synopsis	Description
INTEGER FUNCTION **NX_INITVE**(partition, size, account, argc, argv) CHARACTER *partition**(*) INTEGER *size* CHARACTER *account**(*) INTEGER *argc* INTEGER *argv*	Create a new application.
INTEGER FUNCTION **NX_LOAD**(node_list, numnodes, ptype, pid_list, pathname) INTEGER *node_list*(*) INTEGER *numnodes* INTEGER *ptype* INTEGER *pid_list*(*) CHARACTER *pathname**(*)	Execute a stored program on some or all nodes of an application.
INTEGER FUNCTION **NX_LOADVE**(node_list, numnodes, ptype, pid_list, pathname, argv, envp) INTEGER *node_list*(*) INTEGER *numnodes* INTEGER *ptype* INTEGER *pid_list*(*) CHARACTER *pathname**(*) INTEGER *argv* INTEGER *envp*	Execute a stored program on some or all nodes of an application, with specified argument list and environment.
SUBROUTINE **NX_WAITALL**()	Wait for all application processes.

A.3.11 Handling Errors

Fortran Calls for Handling Errors

Synopsis	Description
SUBROUTINE **NX_PERROR**(string) CHARACTER *string**(*)	Print an error message corresponding to the current value of *errno*

A.3.12 Miscellaneous Calls

Miscellaneous Fortran Calls

Synopsis	Description
SUBROUTINE **FLICK**()	No effect.
SUBROUTINE **LED**(state) INTEGER *state*	No effect.
DOUBLE PRECISION FUNCTION **DCLOCK**()	Return time in seconds since January 1st 1970.

A.3.13 Performing Extended Arithmetic

Fortran Calls for Performing Extended Arithmetic

Synopsis	Description
SUBROUTINE **EADD**(e1, e2, eresult) INTEGER *e1*(2) INTEGER *e2*(2) INTEGER *eresult*(2)	Add two extended numbers.
INTEGER FUNCTION **ECMP**(e1, e2) INTEGER *e1*(2) INTEGER *e2*(2)	Compare two extended numbers.
SUBROUTINE **EDIV**(e, n, result) INTEGER *e*(2) INTEGER *n* INTEGER *result*	Divide extended number by integer.
SUBROUTINE **EMOD**(e, n, result) INTEGER *e*(2) INTEGER *n* INTEGER *result*	Give extended number modulo an integer
SUBROUTINE **EMUL**(e, n, eresult) INTEGER *e*(2) INTEGER *n* INTEGER *eresult*(2)	Multiply extended number by integer.
SUBROUTINE **ESUB**(e1, e2, eresult) INTEGER *e1*(2) INTEGER *e2*(2) INTEGER *eresult*(2)	Subtract two extended numbers.
SUBROUTINE **ETOS**(e, s) INTEGER *e*(2) CHARACTER *s*(*)	Convert extended number to string.
SUBROUTINE **STOE**(s, e) CHARACTER *s*(*) INTEGER *e*(2)	Convert string to extended number.

B Reference Manual

NAME adbg(1)

adbg — connect a debugger to all local application processes

SYNOPSIS

adbg application [debuggeroptions]

DESCRIPTION

The adbg command starts a debugger for all application processes on the machine where the adbg command is invoked. Therefore, the commands scans the process list with the ps(1) command and retrieves the PIDs of all processes with the command name application. After that for every process a debugger is started which connects to the corresponding PID. Optionally the command passes the debuggeroptions to every debugger which is invoked.

CONFIGURATION

During the installation process of NXLib the debugger which is used may be selected by setting the `DEBUGGER` macro in the `Makefile` to any debugger like adb(1), dbx(1) or xdbx(1).

SEE ALSO

adb(1), dbx(1), ddbg(1), ps(1), xdbx(1)

BUGS

The debugger which is used must be capable to attach to a running process. adbg was tested only on Suns with SunOS 4.1.x using xdbx.

NAME *application*(1)

application — start an NXLib application

SYNOPSIS

application [-on nodespec] [-pn partition] [-pt ptype] [-sz size] [-tcpbuf buffersize] [appl_args]
[\; file [-on ptype] [-pt ptype] [appl_args]]

DESCRIPTION

Start a parallel application.

Note that the directory where file can be found must be part of the **local** `PATH` environment variable, even if file even on remote machines. This is because the complete environment will be exported to remote machines.

OPTIONS

-on nodespec Loads the application on the nodes specified by the nodespec argument. The default loads the application on all application's nodes. The nodespec must be one of the following node specifiers:

- **x** Node whose number is x.
- **x..y** Range of nodes form number x to y, inclusive.
- **n** Letter n specifies the last node of the partition.
- **nspec[,nspec]...** List of nodes for the application. Each nspec argument is a node specifier. You can list nodes using any combination of the node specifiers x, x..y, or n. Do not put spaces in this list.

The number you use with the -on switch are node numbers within the application. The range of node numbers is form 0 to one less than the number of nodes allocated to the application.

-pn partition Specifies the partition file partition which is used. If partition is a relative pathname it is accessed relative to the working directory. If no partition is specified on the command line the value of the environment variable `NX_DFLT_PART` is used as the partition. If the environment variable `NX_DFLT_PART` is not set either the .compute partition is used as the default partition.

The partition file specifies the workstations, on which the application will be run. Every line in the partition file specifies a virtual node with the following syntax:

```
nodenumber machinenname[:pathname] [login] [% arch]
```

Both the node number and machine name are required whereas the login and arch specification are optional.

-pt ptype Sets the process type of each process in the application. The ptype argument must be an integer greater than or equal 0. The default is 0.

-sz size Sets the number of nodes to allocate to the application. The size argument must be an integer ranging from 1 to the number of nodes in the partition. The default is the value of the `NX_DFLT_SIZE` environment variable, or the number of nodes in the partition if the `NX_DFLT_SIZE` variable is not set.

-tcpbuf buffersize The buffersize argument must be a positive integer. It specifies the size of the TCP send and receive buffer used. The default value is set in NXLib's `include/config.h` file and is usually 32768. This value produces the best performance for most applications. In order to tune your application or if the application blocks in exchange communications this parameter can be modified.

appl_args Additional arguments specific to the application.

file Loads file onto some or all of the same nodes as *application*. The file must be a compiled executable file. The -pt and -on switches following file specify the process type and nodes for file; the appl_args following file specify additional application-specific arguments for file. All switches specified before the escaped semicolon (`\;`) apply both to the *application* and the file.
The escaped semicolon (`\;`) before file argument must be preceded and followed by a space or tab. Otherwise it will be considered part of the preceding or following argument.
Note that the directory where file can be found must be part of the **local** `PATH` environment variable, even if file even on remote machines. This is because the complete environment will be exported to remote machines.

ENVIRONMENT

The environment variable `NXLIBPATH` is used to select an alternate installation directory. The environment variable `NX_DFLT_PART` is used to select a partition if no command line switch -pn is given. The environment variable `NX_DFLT_SIZE` is used to specify the number of nodes to allocate to the application if no command line switch -sz is given.
To insure proper operation of an NXLib application the directory where the *application* binary can be found must be part of the `PATH` environment variable. The complete environment of the machine where the NXLib application is started, is also established on the remote machines. Therefore, if the *application* binary is installed in different directories on the different machines these directories must also be part of the local `PATH`.

NAME ddbg(1)

ddbg — connect a debugger to all local `nxdaemon` processes

SYNOPSIS

ddbg [debuggeroptions]

DESCRIPTION

The ddbg command starts a debugger for the `nxdaemon` on the machine where the ddbg command is invoked. Therefore, the commands scans the process list with the ps(1) command and retrieves the PID of the `nxdaemon`. After that a debugger is started which connects to the corresponding PID. Optionally the command passes the debuggeroptions to the debugger which is invoked.

CONFIGURATION

During the installation process of NXLib the debugger which is used may be selected by setting the `DEBUGGER` macro in the `Makefile` to any debugger like adb(1), dbx(1) or xdbx(1).

SEE ALSO

adb(1), adbg(1), dbx(1), nxdaemon(1), ps(1), xdbx(1)

BUGS

The debugger which is used must be capable to attach to a running process. ddbg was tested only on Suns with SunOS 4.1.x using xdbx.

NAME nxdaemon(1)

nxdaemon — NXLib process management daemon

SYNOPSIS

nxdaemon daemonparameters

DESCRIPTION

During the startup phase of a NXLib *application*(1) for every node a nxdaemon process is created on the appropriate machine via an rsh(1C) call. After that the nxdaemon creates the necessary application processes for that virtual Paragon node. The daemonparameters are used to initialize the nxdaemon and may change in later NXLib versions. During run-time the nxdaemon is used for process management and address resolution to establish connections between application processes.

NOTE

As this is a system command users should refrain from invoking it from the command line. Usually an invocation from the command line will produce either hanging nxdaemons or cores.

ENVIRONMENT

The environment variable `NXLIBPATH` is used to select an alternate installation directory. If you use `NXLIBPATH` make sure that this directory exists on every machine which is specified in the partition file.
To insure proper operation of an NXLib application the directory where the nxdaemon binary can be found must be part of the `PATH` environment variable of each machine which is specified in the partition file.

SEE ALSO

application(1)

NAME nxlstdin(1)

nxlstdin — Support reading from stdin

SYNOPSIS

nxlstdin application and arguments

DESCRIPTION

The nxlstdin shell skript provides reading from stdin for NXLib applications. Input form stdin is limited to processes which run on the host where the application was started.

Input must be terminated by a EOF which usually is the Ctrl-D key.

SIGTSTP (Ctrl-Z) is disabled

BUGS

If more than one process reads from stdin it is undefined whether one or more processes will read and which of the processes will get the input.

SEE ALSO

application(1)

NAME vpcc(1)

vpcc — compile NXLib C applications

SYNOPSIS

vpcc [-V] [-nx|-lnx] [compileroptions]

DESCRIPTION

The vpcc shell script provides an easy way to compile and link NXLib applications. The vpcc can be called similarly to the Paragon icc(1) compiler. Depending on the compileroptions the necessary compile and link operations are carried out.

OPTIONS

-lnx no automatic startup is performed. The programmer is responsible to start the parallel application on the specified partition and nodes from the controlling process.

-nx used to link applications which automatically start on the partition and number of nodes which were specified on the command line of the application. The controlling process performs the startup.
If neither -nx nor -lnx are specified the standard C compiler is used and no NXLib application will be produced.

-V turns on verbose command processing, i.e. the script echoes some commands which it executes to stdout.

compileroptions Additions arguments which are passed to the compiler. Make sure the compiler can interpret the arguments.

CONFIGURATION

During installation the `Makefile` macros `NXLIBDIR` and `CC` are inserted into the shell script. Their values specify the default installation path and the C compiler which is used to compile and link applications.

ENVIRONMENT

The environment variable `NXLIBPATH` is used to select an alternate installation directory.

SEE ALSO

icc(1), vpf77(1)

BUGS

Currently all compileroptions must be switches which can be interpreted by the local C compiler. For later versions an automatic conversion of the Intel Paragon switches to corresponding switches will be integrated.

NAME vpCC(1)

vpCC — compile NXLib C++ applications

SYNOPSIS

vpCC [-V] [-nx|-lnx] [compileroptions]

DESCRIPTION

The vpCC shell script provides an easy way to compile and link NXLib applications. Depending on the compileroptions the necessary compile and link operations are carried out.

OPTIONS

-lnx no automatic startup is performed. The programmer is responsible to start the parallel application on the specified partition and nodes from the controlling process.

-nx used to link applications which automatically start on the partition and number of nodes which were specified on the command line of the application. The controlling process performs the startup.
If neither -nx nor -lnx are specified the standard CC compiler is used and no NXLib application will be produced.

-V turns on verbose command processing, i.e. the script echoes some commands which it executes to stdout.

compileroptions Additions arguments which are passed to the compiler. Make sure the compiler can interpret the arguments.

CONFIGURATION

During installation the `Makefile` macros `NXLIBDIR` and `CXX` are inserted into the shell script. Their values specify the default installation path and the CC compiler which is used to compile and link applications.

ENVIRONMENT

The environment variable `NXLIBPATH` is used to select an alternate installation directory.

SEE ALSO

icc(1), vpf77(1)

BUGS

All compileroptions must be switches which can be interpreted by the local C compiler. The skript is not tested with various different C++ installations.

DISCALMER

Geoffrey Furnish supplied this shell script in the hope that it will be useful for other NXLib users.

NAME vpf77(1)

vpf77 — compile NXLib Fortran applications

SYNOPSIS

vpf77 [-V] [-nx|-lnx] [compileroptions]

DESCRIPTION

The vpf77 shell script provides an easy way to compile and link NXLib applications. The vpf77 can be called similarly to the Paragon if77(1) compiler. Depending on the compileroptions the necessary compile and link operations are carried out.

OPTIONS

-lnx no automatic startup is performed. The programmer is responsible to start the parallel application on the specified partition and nodes from the controlling process.

-nx used to link applications which automatically start on the partition and number of nodes which were specified on the command line of the application. The controlling process performs the startup.
If neither -nx nor -lnx are specified the standard Fortran compiler is used and no NXLib application will be produced.

-V turns on verbose command processing, i.e. the script echoes some commands which it executes to stdout.

compileroptions Additions arguments which are passed to the compiler. Make sure the compiler can interpret the arguments.

CONFIGURATION

During installation the `Makefile` macros `NXLIBDIR` and `FLIBDIR` are inserted into the shell script. Their values specify the default installation path and the location of the Fortran libraries.

ENVIRONMENT

The environment variable `NXLIBPATH` is used to select an alternate installation directory.

SEE ALSO

icc(1), vpcc(1)

BUGS

Currently all compileroptions must be switches which can be interpreted by the local Fortran compiler. For later versions an automatic conversion of the Intel Paragon switches to corresponding switches will be integrated.