Express C

Reference Guide

Version 3.0

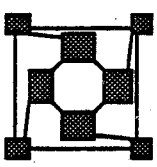
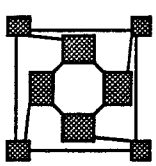


Table of Contents



System Commands

Tools providing services in support of *Express* applications

1 Executing *Express* commands in "non-windowing" operating systems

When running *Express* under operating systems with conventional line-oriented user interfaces such as UNIX, MS-DOS, VMS and similar, commands are executed by typing their names at the command line prompt.

Usage generally follows the conventional UNIX style with options being indicated by the '-' character, e.g.,

```
cubix -n 2 -t 120 toyland
```

The particular command line string '-?' provides a brief on-line summary of the options and purpose of a command. While this may help in jogging the memory this manual should be consulted for complete details.

On most machines you will need to add a new directory to the set which is searched when looking for executable programs. The exact mechanism for doing this varies from one machine to another and the details for individual operating systems can be found in the introductory guide to *Express* for that machine. If you find that messages such as

```
Command not found
```

or

```
No such file or directory
```

appear whenever you try to execute one of the *Express* commands then you should check that the appropriate directory really has been added to your search path. If this seems to be correct you should next try running the `excustom` program which will ensure that the *Express* installation is internally consistent.

In keeping with the conventional style all commands exit with status 0 upon successful termination and with non-zero values if errors occur.

2 Executing *Express* commands in "windowing" systems

In windowing systems such as MicroSoft Windows and the Macintosh, *Express* programs are usually executed by selecting icons from the screen. In most cases a dialog box will then be presented allowing the entry of parameters. In most cases the entries to be made have a one-to-one correspondence to the switches used in the line-oriented interfaces. Usually some mechanism is also provided to "`Abort`" or "`Cancel`" the program without executing any commands.

Note that only the line-oriented interfaces are completely documented in this reference. In most cases this causes few problems since the switches and "boxes" are obviously related to one another. In cases where confusion may arise the introductory guide to *Express* on your system should be consulted for more help.

3 Specifying numeric data in switches

Many of the parameters necessary to the commands listed in this section have numerical values - the number of processors to use, the number of bytes to display, the position at which to load the *Express* kernel, etc. In *most* cases these values can be entered with the usual C-style notation as either decimal, octal or hexadecimal values.

Consider the `exdump` command, for example. One of its arguments specifies the address from which data should be extracted - the '`-B`' switch. Typically one knows this value as some "hex" constant and would therefore use a command of the form

```
exdump -B 0x79000..........
```

Alternatively you could use either octal or decimal notation replacing this by

```
exdump -B 01710000........
```

or

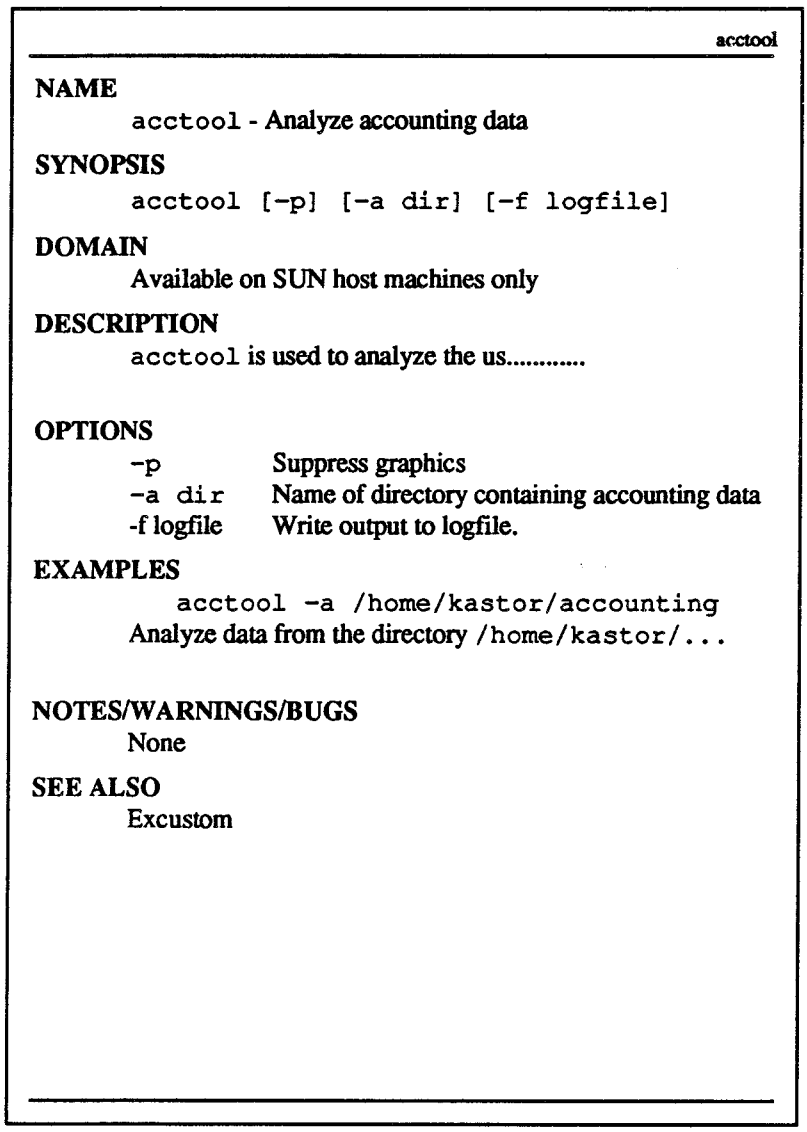
```
exdump -B 495616.......
```

to achieve the same effect. Similar remarks apply to most of the other *Express* commands - you can execute a *Cubix* program on 16 nodes with any of the command switches

```
cubix -n 16........
cubix -n 020........
cubix -n 0x10........
```

4 Manual Page Layout

The manual pages are, for better or worse, modeled after those often found in UNIX documentation which means that each manual page has several well-defined sections. The overall structure is shown below.



acctool

NAME

`acctool` - Analyze accounting data

SYNOPSIS

`acctool [-p] [-a dir] [-f logfile]`

DOMAIN

Available on SUN host machines only

DESCRIPTION

`acctool` is used to analyze the us.....

OPTIONS

`-p`	Suppress graphics
`-a dir`	Name of directory containing accounting data
-f logfile	Write output to logfile.

EXAMPLES

```
acctool -a /home/kastor/accounting
```

Analyze data from the directory `/home/kastor/...`

NOTES/WARNINGS/BUGS

None

SEE ALSO

Excustom

Header contains the name of the manual page which is usually the same as the command described.

The various sections and their contents are:

NAME Repeats the name associated with the manual page and a brief one-line description of the purpose of the associated routines

SYNOPSIS Summarizes the arguments used by the indicated command. Arguments enclosed in '[', ']' pairs are optional. If more than one command is described on a particular page then all are listed in this section

DOMAIN Describes the machines on which the command is available and any restrictions on when it may be used.

DESCRIPTION Describes the purpose of each command and lists the actions caused by its

most important arguments. This section is the most important reference material for each command.

OPTIONS This section lists all the supported arguments for each command and the actions caused by specifying them.

EXAMPLES Usually several examples are presented of the use of each function showing the most important arguments and switches.

NOTES If present this section contains useful information about oddities in the implementations of a particular command. It may also repeat important information from the **DESCRIPTION** section.

WARNINGS If the command has peculiar side effects or is "dangerous" in some way it will be noted in this (optional) section. Any non-intuitive behavior is also noted here.

BUGS Currently known bugs and/or unimplemented switches are noted in this (optional) section.

SEE ALSO Related commands and/or routines from the various *Express* libraries are noted in this section. Using this information is usually the quickest way to build a complete picture of the interaction between the various utilities.

NAME

acctool - Analyze parallel computer usage under *Express*

SYNOPSIS

```
acctool [-p] [-a account_dir] [-f logfile]
```

DOMAIN

This command is available on SUN host computers only.

DESCRIPTION

acctool is used to analyze accounting data previously obtained from *Express* programs.

If the accounting system has been enabled on a particular host every *Express* program writes an entry into a system data file whenever it allocates or deallocates nodes. Special entries are also assigned whenever the system crashes or is reinitialized. acctool analyzes this data in an interactive fashion displaying the usage of resources on a machine-by-machine basis.

Results are reported for all users, in hours, or on a single job basis for individual users, in seconds. Statistics are managed on a monthly basis with options to restrict attention to particular months or ranges of months.

The operation of the accounting system is controlled by the excustom command. One of this system's options is whether or not to enable the accounting system. If enabled a place must be indicated for the accounting information to be maintained.

OPTIONS

`-a account_dir`
: By default acctool looks in the current directory for the data files describing the system configuration and accounting data. This switch allows an alternative directory to be specified.

`-f logfile`
: All information provided by acctool appears on the display device. If this switch is given a "log file" will also be kept containing the identical information. (In the Sunview version of the program this effect is obtained by entering a name in the log file field of the control panel.)

`-p`
: By default acctool operates in the Sunview environment providing a simplified user interface. If Sunview is not supported on your system this switch enables a line-oriented interface in which the user is prompted to enter various options from the keyboard.

`-?`
: Print usage message.

EXAMPLES

The following command executes the profiling tool in a windowing environment and searches the directory /home/kastor/accounting for the necessary databases.

```
acctool -a /home/kastor/accounting
```

SEE ALSO

excustom.

NAME

`cnftool` - Configure Transputer systems.

SYNOPSIS

```
cnftool [-p] [-d]
```

DOMAIN

This command is available at the system prompt on the host processor.

DESCRIPTION

This command is used to configure or reconfigure a transputer network for use with *Express*. Two interfaces are available; with the '`-p`' switch a simple line oriented interface leads the user through the configuration process. Without this switch a menu driven utility allows the user to specify the physical transputer interconnect and also to add additional hosts to an existing system.

One of the features of the system is a "worm" program which can be used to detect the initial hardware configuration on statically wired systems. This program has a simple searching algorithm which examines the links on each node and attempts to find a node connected to each. As each link is examined and another node detected the program recursively examines other nodes which may be attached. Note that this can only be achieved if the system has "physical" rather than electrical connections. Hardware which has INMOS' link switch cannot be examined by this method since the links are initially disconnected on hardware reset.

Details of the use of this system can be found in the accompanying documentation, "Configuring Transputer Systems: `cnftool`".

OPTIONS

`-p`	By default `cnftool` supports a menu driven graphical interface. This switch enables a simpler, but more tedious, line interface for system configuration.
`-d`	Run silently - the system is configured in much the same way as with the '`-p`' switch except that a "general" network topology is also selected automatically. No user interaction is required unless the "worm" program fails to operate successfully.
`-?`	Print usage message.

SEE ALSO

"`CnfTool:` Configuring Transputer Systems"

NAME

`ctool` - Analyze Communication Profile

SYNOPSIS

```
ctool [-b nbins] [-p] [log_file_name]
```

DOMAIN

This command is available at the system prompt on the host processor.

DESCRIPTION

This command is used to examine and analyze the log file created with the communication profiler commands. The only argument is the name of the file containing the profile data which may be omitted if it has the default value "`cprof.out`".

If the "`-p`" switch is given this command presents a separate table on `stdout` from each node. The information contained in each table is:

- An identifier showing which node the following data is from.
- A summary of the calculation, communication and I/O times in the processor. In making this classification all inter-node and basic host-node communication comes under the heading "Communication" while genuine I/O requests such as calls to `read`, `write`, `printf`, `fopen`, etc. are counted as I/O.
- A summary of the time spent in, number of calls to and errors incurred in each communication function called by the processor. This information is use to give a quick breakdown of the total communication pattern. The "error" count is also a good place to look for obscure bugs. Each function makes some consistency checks on the supplied arguments and returns an error if they are inconsistent.
- A breakdown of the values returned by the communication functions. The return values are binned logarithmically - the column headed "8" indicates the frequency of return values in the inclusive range 8 thru 15. The exact interpretation of this data depends on the particular function being invoked but is usually related to the message length involved in the call. By default data from ten logarithmic bins is included in the output although the '`-b`' switch is provided to override this default.

One very important use of this system is the detection of programs which are sending too much data in their messages. These will show up very clearly in the histogram output.

This data appears on `stdout`.

If the `ctool` command is invoked without the "`-p`" switch then a graphical interface allows data to be presented in graphical form. The package is menu-driven and (hopefully) quite straightforward to use. A full list of the available options is presented in Subsection 3.4 of the *PM* manual.

OPTIONS

`-p` Suppress graphical output. The analysis results are presented in tabular form

on stream `stdout`.

`-b nbins` Specifies an alternate number of logarithmic bins to display when used in conjunction with the '`-p`' switch. (Default 10).

EXAMPLES

To examine the profile data in a file called "`phase3.prof`" execute the command

```
ctool phase3.prof
```

SEE ALSO

`cubix` in the *Express* documentation.

NAME

cubix - Host slave process for node programs.

SYNOPSIS

```
cubix [-?] [-n nodes] [-d doc] [-P] [-t time] [-S]
[-T plot_option] [-E custom_file] [-f file] [-fp]
[-mc|x|e] [-D] [-x] program [arg1] [arg2]...
```

DESCRIPTION

This command provides an interface between node applications and the host filesystem and operating system utilities. It is also responsible for node allocation and the communication of command line arguments to a node program.

This command, with the '-S' option, starts up a *Cubix* server process. Instead of loading a user application as is usually the case the server merely waits for I/O requests from any node in the parallel computer system.

While all standard I/O and graphics requests are available the stateless nature of the server may make its operation slightly strange in a multi-user environment. Routines which affect the state of the system such as chdir will have ramifications beyond those normally expected. In this case, for instance, a call which changes the active directory of the server for one user may invalidate requests to open files for another user.

OPTIONS

-n nodes	Allocate nodes processor for this process. Default 1.
-d doc	Alternative to -n switch. Specify size of processor group logarithmically in manner suitable for hypercubes (i.e., doc = 0 for 1 node, doc = 1 for two, doc = 2 for four, etc.) Default 0.
-P	Load the program into its processors but do not start it running. This option is useful in connection with the node debugger ndb.
-t time	Time out the process after the given number of seconds. This can be useful in detecting 'hung' programs. The default is no time out.
-T option	Specify a graphics option for programs that use *Plotix*.
-fp	Execute the program on attached "vector" nodes, if available.
-f file	Read the programs to be run and their arguments from the specified file. This option is used whenever different node programs are required or different arguments should be passed to different nodes. The file format is basically single lines containing a range of nodes, an executable program and an argument list. See the examples below.
-E custom_file	Directs cubix to use an alternative system customization file rather than the system default.
-m[x|c|e]	Enable the performance monitoring tools. The 'x', 'c' and 'e' characters

refer to the execution, communication and event driven profiling systems respectively and may be combined. For more details refer to the *PM* manual.

`-o start_node`
: Specifies which nodes are to be allocated to the program. An attempt wil be made to allocate consecutively numbered nodes starting at the indicated number. If this cannot be done the `cubix` command will terminate.

`-S`
: Enter server mode. (Used on multi-host systems.)

`arg1 arg2...`
: These arguments are passed to the node main program as the conventional runtime parameters `argc, argv`.

`-D`
: Enable system debugging. With this option set `cubix` prints a huge amount of data about the system as it runs. Should be of little interest to most users.

`-x`
: In windowing versions of *Express* such as the Macintosh or MicroSoft Windows this switch forces the *Express* kernel to be re-loaded before beginning the user program. It is essentially equivalent to running the `exinit` program from the shell.

`-?`
: Print usage message

EXAMPLES

```
cubix -n 4 noddy
```

Loads the program `noddy` into four processors. No arguments will be passed to `main()` other than the program name in `argv[0]`.

```
cubix -d 1 -t 120 -mce longjob 3.14 2.72
```

Loads the executable `longjob` into two nodes with a total execution time limit of two hours. Also passes two extra arguments to the node program. Finally enables both communication and event driven performance monitoring tools.

```
cubix -n4 -Tega plotter
```

Run the program `plotter` in four nodes and enable graphics output on an IBM Enhanced Graphics Adapter.

```
cubix -n 1 -P buggy
```

This sequence loads a single node with the user code buggy but halts execution before the users main routine. The job is run in background mode so that debugging can be carried out.

```
cubix -n 4 -f loadfile
```

This sequence allocates four nodes and then loads programs according to the instructions found in the file `loadfile`. Basically the format is single lines containing either a node number or a range of nodes followed by a program name and argument list. Blank lines are ignored and # introduces comments. Continuation lines, backslashes and quotes are processed in the conventional manner. As an example consider the following sample `loadfile`

```
# This is a command file specifying how node programs
# should be loaded into the cube.
     0-1 proga foo bar
     3 progb horse dog cow
     2 progc really\ one\ argument
```

Note that a range of nodes is indicated for `proga` and that the backslash symbol is used to concatenate tokens into a single argument - in the above case `progc` would have only two arguments the name `progc` and the string `really one argument`.

EXIT CODE

The `cubix` process exits to the shell with the same exit code as used in the call to `exit()` in the node program.

DIAGNOSTICS

Among the errors detected by `cubix` are requests for more nodes than are available and missing program files. After validating that the specified program is indeed an executable image it is loaded into the machine using the `exload` system call. This produces messages about the size of file to be loaded and a single 'b' character for each 1024 byte block loaded. A common situation is that in which the previous job crashed the node operating system in which case the loader will say `loading` some number of bytes but no 'b's appear, or many 'b's appear and the final 'E' but the program does nothing after the `Starting` message. This is usually a good time to run `exinit`.

Upon exit `cubix` reports the elapsed time divided between `user` and `system`. The latter is time spent performing system functions such as program loading and is always rather small. It is provided simply for compatibility with other systems running `cubix` applications.

NAME

`etool` - Analyze Event Profile

SYNOPSIS

```
etool [-p] [-t] [log_file_name]
```

DOMAIN

This command is available at the system prompt on the host processor.

DESCRIPTION

This command is used to examine and analyze the event log created with the `eprof` commands. The only argument is the name of the file containing the profile data which may be omitted if it has the default value "`eprof.out`".

This package is exclusively graphical and menu-driven. The most complete source of reference is the discussion in Section 4.4 of the *PM* manual.

OPTIONS

`-p`	Suppress graphical output. The analysis results are presented in tabular form on stream `stdout`.
`-t`	Display only the data from the "toggle" events.

EXAMPLES

To examine the profile data in a file called "`phase3.prof`" execute the command

```
etool phase3.prof
```

SEE ALSO

`cubix` in the *Express* documentation.

NAME

excustom - Reconfigure *Express*.

SYNOPSIS

```
excustom [-r] [-?] custom_file
```

DOMAIN

This command is available at the system prompt on the host processor.

DESCRIPTION

excustom is used to modify the system parameters which describe a particular implementation or version of the *Express* system. All system variables are maintained in a particular file called the "*Express* customization file" which is located in an operating system dependent location. By default excustom will modify this file although another may be indicated by the optional custom_file argument.

In non-windowing environments excustom prompts, in turn, for values of all important system parameters, offering defaults based on the values found in the current customization file. If you do not enter any input on a particular line the original value will be taken. In windowing systems a pop-up display is typically used to offer the current values of all parameters for modification. You can then change individual entries at will. An additional option restores all entries to "sensible" defaults which guarantee that *Express* will operate correctly. (This option is obtained in the non-windowing environment by specifying the '-r' switch when invoking excustom.)

The excustom tool typically asks only about top level information from which it derives all other related data using the "macro" mechanism discussed below. In some cases you may need to modify individual system parameters at a finer level of detail. This is achieved by simply editing the customization file with a suitable text editor or word processor. (You can find the name of the default customization file with the '-?' command line option.) The exact meaning of all system variables is shown in the accompanying "Excustom" manual.

In order for the customization file to take effect the system must be reloaded with the exinit command.

OPTIONS

-r	By default excustom prompts you with the current system parameters as obtained from the customization file. With this switch "sensible" defaults are used instead of the current values.
custom_file	This argument requests that the modification process be applied to the named file rather than the default system configuration file. This allows excustom to be used by applications which maintain their own customization systems.
-?	Print the name of the default system configuration file.

IMPLEMENTATION

The customization file is a line oriented ASCII file which contains definitions of important system variables, one to a line. Lines beginning with either '`;`' or '`#`' characters are treated as comments. Other lines take either of the symbolic forms

```
NAME=text
MACRO:=text
```

As is suggested by the above notation the former type are merely assignments to *Express* system parameters while the second define macros that may be further used in the customization file to simplify definitions of multiple related objects. A good example might be the default start-up information required by the debugger, `ndb`. As part of its configuration information it needs to know the location of the on-line help facility and also the system start-up file which contains the definitions of system commands. Since these are often in the same or related directories one might imagine two entries in the customization file of the type

```
NDB_HELPDIR=c:\parasoft\help
NDB_STARTUP=c:\parasoft\lib
```

These entries could, however, be replaced using the macro replacement facility with the lines

```
PARASOFT:=c:\parasoft
NDB_HELPDIR={PARASOFT}\help
NDB_STARTUP={PARASOFT}\lib
```

Notice that the value of the `PARASOFT` macro is indicated with the '`{`' and '`}`' characters.

While three lines may seem more complex than the original two the use of the `PARASOFT` macro means that the *Express* system can be moved from one directory to another by simply changing the macro rather than each line of the customization file.

SEE ALSO

`excustom` (subroutine), "Customizing *Express*".

NAME

exdump - Retrieve data from node RAM files

SYNOPSIS

```
exdump [-?] [-B base] [-d doc] [-l length] [-n nodes]
[-N node] [-N node-node] [-o file] [-t threshold]
[-p pid] [-s start] [device]
```

DOMAIN

This command is available at the command line prompt on the host processor.

DESCRIPTION

This command is used to retrieve the debugging information stored in the internal RAM file under *Cubix*. It can be used either as a post-mortem dump or while a process is still running. If set up correctly data can be retrieved after machine initialization with `exinit`. The `device` argument specifies which array the dump is to be taken from - in the current implementation this should be left to its machine dependent default.

By default the dump is assumed to contain ASCII data and continues until several consecutive unprintable characters are seen. An alternative is to dump in "binary mode" in which case data is just read from the node file and sent to `stdout`. In this case options are available to both control the amount of data printed and also redirect the output to a file - printing binary data to a terminal has rather detrimental effects on its behavior.

The detailed use of the RAM file and its manipulation are fully described in the accompanying *Cubix* documentation - "Programming Parallel Computers Without Programming Hosts".

OPTIONS

`-B` Dump binary data instead of ASCII. By default 16 Kbytes will be taken from each node.

`-b base` Defines the base address of the RAM file. Decimal, octal and hex constants are valid base values. Note that this option potentially interacts with the linker/locator and also the parameters used in the `ramfopen` call. Consult the Appendix discussing RAM files in the *Cubix* manual.

`-d doc` Dump data from 2^{doc} nodes. This is an alternative to the `-n` option designed for hypercube users.

`-l length` Specify amount of data to be dumped from each node. In the default ASCII mode less will be read if the data ends early.

`-n nodes` Specify number of nodes from which to dump data.

`-t threshold`
As currently implemented `exdump` is most useful for retrieving printable ASCII information. It continues reading data until `threshold` successive unprintable characters are seen and then moves onto the next node. The

default `threshold` is five.

`-N node` Read the RAM file from processor `node`.

`-N node1-node2`
Read RAM files from the inclusive range of nodes `node1-node2`.

`-o file` Redirect output to the named file. Default output is to `stdout`.

`-p pid` If the process whose file is to be examined is still active then its process ID should be specified and its RAM file will be read.

`-s start` Specify the physical node number from which the dump is to start. This is useful in cases where the program ran in high numbered nodes and you are dumping data after the program has stopped. Since the default allocation strategy is to allocate the lowest numbered nodes with the required size it is occasionally necessary to use this switch to "grab" the higher numbered processors.

`-?` Print usage message

EXAMPLES

```
exdump -d 1 -s 2
```

This command reads the RAM file from the default address in two nodes. The two processors will, if possible, be those starting at node two in the array. This form of the command is often used either after the node has "hung" in communication (and the nodes had to be reset with `exinit`) or when the process finished but with some error. Note that `exinit` normally initializes the contents of memory while loading *Express* so it is necessary to use the `excustom` facility to prevent this if we wish to preserve RAM file data.

```
exdump -n 1 -B 0x1000
```

Retrieves the data from a single node starting at address 1000 (hex). This form is used in conjunction with the ramfopen call in *Cubix*.

```
exdump -p 376 -n 4 -N2-3
```

This option retrieves the information currently contained in the RAM file of the process whose ***process ID*** is 376. Data will only be dumped from nodes 2 and 3 in the group allocated by the process.

```
exdump -b -B0x80001000 -o xdump.out -l 4096 -n2
```

Dump 4 Kbytes of data in binary mode from two nodes. Write the output to the file `xdump.out`.

NOTES

Numeric parameters may be specified in decimal, octal or hex using the usual C style notation: 123 is decimal, 0123 is octal and 0x123 is hex. Switch values may follow immediately after their switches or there may be intervening spaces: '`-B0x1000`' and '`-B 0x1000`' are both valid.

SEE ALSO

"*Cubix*: Programming Parallel Computers Without Programming Hosts."

`exinit` (command)

NAME

exinit - Reboot and reload *Express* kernel.

SYNOPSIS

```
exinit [-K] [-m] [custom_file]
```

DOMAIN

This command is available at the system prompt on the host processor.

DESCRIPTION

exinit must be executed before any routines may access the parallel machine - failure to do so will result in the failure of all attempts to allocate processors. It loads the *Express* kernel and starts it running in the node processors. It also performs any necessary hardware configuration on systems which support such abilities.

exinit attempts to check that no node processes are actually executing before resetting the hardware. If node programs are detected exinit will report and suggest the use of the '-K' switch. If this switch is supplied any node programs will be killed before the reset operation is performed.

It is important that all node processors be reset before loading the *Express* kernel since otherwise parts of the network may be inaccessible. Most hardware systems have intelligent reset lines so that several boards may be reset one by another. If you are connecting several hosts together the exreset command is available to reset a particular subset of the network. It does not, however, reload the *Express* kernel. This must be done with the exinit command from some other console.

The optional custom_file argument is used to specify an alternative customization file when downloading *Express*. This allows temporary modifications to the system parameters for testing and also allows custom applications to maintain their own customization information.

A very important point to note about exinit is that, by default, it destroys the contents of the node memory while loading *Express*. This behavior is normally quite reasonable with the exception that one may wish to preserve the contents of some RAM file for use with exdump or ndb. In this case the '-m' switch can be used to prevent memory initialization. Alternatively the excustom tool has an option which forces the kernel to be loaded without destroying memory by providing an explicit "start address". A good way to proceed, therefore, is to make an system customization file which contains the load address and then to re-load *Express* by telling exinit to use this file rather than the system default.

OPTIONS

-K	By default exinit aborts if any processes are still running in the parallel machine. This switch causes an attempt to be made to kill all such programs before resetting and reloading *Express*.
-m address	Load *Express* into the nodes at the indicated address without destroying

the contents of memory. This is useful in conjunction with the RAM file system for debugging after system crashes. The address used will depend on your hardware configuration.

`custom_file` Indicates that a system customization file other than the default should be used to load *Express*.

`-?` Print usage message.

EXAMPLES

```
exinit -m 0x80069000 -K
```

Reinitialize the machine by killing all currently executing processes and loading *Express* at the indicated address. The current contents of node memory will remain intact, except for the region near `0x80069000` which will be overwritten by the kernel.

SEE ALSO

`exstat`, "`CnfTool:` Configuring *Express*", "Using *Express* with Multiple Hosts", "`Excustom:` Customizing *Express*".

NAME

`exreset` - Reset a group of nodes.

SYNOPSIS

exreset

DOMAIN

This command is available at the system prompt on the host processor.

DESCRIPTION

This command is used to reset a set of boards without loading the *Express* kernel. If your hardware configuration is capable of supporting a tree-like reset path in which all nodes can be reset from a central "master" console this command will be totally unnecessary since `exinit` will be able to reset and load *Express* into all nodes.

If your hardware does not support this chaining of reset signals then you will need to partition the nodes into groups, each of which can be reset from a particular host. The `exreset` command will then perform this operation allowing a subsequent `exinit` to load *Express* into the entire network.

SEE ALSO

`exinit`, "`CnfTool:` Configuring *Express*".

NAME

`exstat` - Display node usage information.

SYNOPSIS

```
exstat [-l] [device]
```

DESCRIPTION

`exstat` is used to present statistics about the current node usage on the multiprocessor `device`. The default value will be either `/dev/transputer` or `/dev/ncube` depending on the hardware installed.

Without the `-l` switch only the number of allocated nodes and the total number of nodes are presented. The inclusion of this switch also provides information about which process is allocated which nodes, and which processes share which nodes.

Until the system has been initialized with a call to `exinit` this utility will return the rather disconcerting result that there are no nodes available.

OPTIONS

`-l`	Produce an extended (long and informative) listing which includes the process I. D. and physical node origin of all active processes.
`-?`	Print usage message.

SEE ALSO

exinit

NAME

ndb - Symbolic, source and assembly level debugger for parallel computers.

SYNOPSIS

```
ndb [-?] [-I incdir] [-p procid] [-d doc] [-n nodes] file
```

DESCRIPTION

ndb is an interactive symbolic debugger for use on parallel computers. Two styles of interface are available depending upon the particular hardware/compiler combination available.

The simplest interface is a source level debugger patterned after the UNIX utility dbx. At this level the user is able to examine code, set breakpoints and examine variables at the level of the original C or FORTRAN source code.

The lower level interface is designed for machine level debugging and is based on the conventional assembly level debugger adb. It allows for the examination of both data and assembly level code as well as the setting of run-time breakpoints. ndb incorporates a superset of adb commands which should facilitate its use.

ndb is always available at this second level and the commands associated with its use are described completely in this document. The availability of the source level interface is subject to compiler/hardware restrictions. The associated commands are described in this manual but may be unavailable in some implementations.

In order to effectively debug parallel programs a simple extension to the conventional debugger syntax has been made. This is the concept of a "set" of processors. Each command is executed on some group of nodes which can be defined and altered by the user. Several common groups of processors are predefined and user defined sets are also available.

COMMAND LINE OPTIONS

-d doc	Specify dimension of subcube to be debugged. Default is 0 (1 node).
-n nodes	Alternative to -d switch - specify the number of nodes rather than its logarithm.
-p procid	Debug a background process. This option is useful in conjunction with the -P switch to cubix which loads a program and stops it at its starting point. It is also used to perform post-mortem debugging on processes which are "hung".
-I dir	Specify a directory to be searched when looking for source code. By default only the current directory is searched. This switch may be repeated multiple times and the associated directories are searched in the order they are specified.
file	Specify the program whose symbol table is to be read. Unless this name is specified no symbol table entries will be available which significantly reduces the capabilities of ndb.

-?	Display information about supported run-time switches.

USAGE

An introductory guide to the debugger is available elsewhere so the following sections merely explain the syntax of the various commands. In nearly all cases the low level syntax is exactly as in the regular UNIX command `adb` while the source level command shares the same syntax as `dbx`.

1 Help

A certain amount of on-line help is available by entering the `help` command. Various topics may be selected for further perusal by entering

```
help topic
```

where `topic` is the name of the required subject. The syntax for a particular command can be found by using

```
help keyword
```

where ***keyword*** is the identifier whose usage is required.

2 Sets

Each command is executed on a collection of nodes called a "set". A 'current set' is maintained by `ndb` which is used unless overridden by prefixing a command with the `on` keyword. There are three verbs used in manipulating sets.

`on`	The following set specification is used for the command that follows it and then the current set reverts to its previous value.
`pick`	The current set is changed to the nodes listed.
`setdef`	Define a new set containing the specified nodes. The resulting set is assigned an index number which is returned and can be used in future set specifications.

The set specifications are built up from (comma separated) elements of the following types. (In the following the symbol # denotes a decimal, integer, constant).

#	A single node.
`node #`	A single node.
`# to #`	An inclusive range of nodes.
`# thru #`	An inclusive range of nodes.
`# - #`	An inclusive range of nodes.
`all`	All nodes in the subcube.
`even, odd`	Either the even or odd parity nodes defined by the number of bits set in the processor number.
`set #`	The set with index # as given by a previous `setdef` instruction. (Note that set numbers are indicated when using the `setdef`

command or with he "`show sets`" command.

`nof #` The "hypercube" neighbors of node #.

`neighbors` The "hypercube" neighbors of all the nodes listed so far.

`board n0;n1`
Specifies node `n1` on attached peripheral board `n0`. Ranges of nodes may also be given as `n1-n2`.

3 Displaying Source Code

The simplest command for displaying source code is `list`. With no parameters this prints out ten lines of program starting at the "current line". This latter is set implicitly during program execution by the `show state`, `where` and single stepping commands. It may be explicitly altered with

`file name` Set the "current line" to the first line of the named file. If no name is given display the current file.

`func name` Set the "current line" to the first line of the named function. With no parameters display the current function.

The `list` command has parameters itself which are either one or two integers separated by a comma. The various combinations of positive and negative values are used to indicate ranges of lines to display. A few examples should clarify the details.

`list 20` Display line 20 only.

`list 20,50` Display lines 20 through 50 inclusive.

`list -5` Display ten lines starting 5 before the current line. This option provides a "window" facility.

`list -5, 20`
Display 20 lines starting 5 before the current line.

`list function`
Display the first ten lines of the named function.

When source files are named, either explicitly in `file` commands or implicitly during program execution an internal directory search path is used to look for appropriate source files. Two methods are available for altering this path. When starting an `ndb` session the "`-I`" command line argument names a directory which should be appended to the current search list. Alternately the `use` command can be used - it is followed by a list of directories which replaces the current list. Thus

```
use . ../src ../lib ../tst
```

might be given to name several directories in which source files are to be found. The order of the entries in this list is important - directories are searched from left to right so possible name clashes may have to be considered. Of course, if `ndb` fails to find the correct version of a source file with its automatic search then the `file` command can be used with a full pathname to override `ndb`'s choice.

The `use` command with no parameters displays the current search path.

4 Stack Operations

The most useful command for finding the current status of the program is `where`. With no arguments this command displays the top 8 levels of stack activity naming subroutines called and displaying their arguments. If less than 8 subroutines have been called the list terminates with the system initialization routine. If more, or less than 8 levels are required then a numeric parameter may be given; `on all where 3` displays the top three function calls in all nodes.

It is important to note that these stack tracing operations require that several probing messages be sent to each node. As a result they may work incorrectly when the node program is actually executing since the stack may be significantly different each time leading to inconsistent results. If this is a problem then one of the single step commands can be used to effectively "stop" the nodes allowing `where` to succeed. The program can be later continued with the `cont` command.

Another useful function in connection with tracing subroutine calls is `isin`. One can say

```
on all isin main
```

to obtain a listing of the activity of all nodes which currently have the named routine in their stacktrace. This is a useful form of data reduction since it allows he user to immediately discover which nodes are in the wrong place.

`dump routine` displays the calling sequence and local variables of the named routine if it occurs in the stack backtrace. If no function is specified then that containing the current program counter is `dumped`.

Note that these commands may interact with the language specification flag discussed in context of the `ndbenv` command. Often different language compilers use different calling sequences that cannot be dynamically inferred from the actual code. In this case a stack trace may be invalid unless the language switch is set correctly. To change from C (the default) to Fortran, for example, one issues the command

```
ndbenv Fortran
```

Commands such as this are often best placed in the user's initialization file `.ndbinit`.

5 Displaying data

The simplest command for showing data values is `print` which takes a list of expressions and prints their values according to the variable types indicated by the program. One can, for example, type

```
print 1+2.5, my_struc
```

to which the response might be

```
1+2.5 = 3.5

my_struc = {
     i = 12
     f = 1.44
```

```
}
```

Some compilers do not preserve information about symbol types in which case you have to explicitly indicate in what manner you wish to see the data displayed. Occasionally you may also wish to override ndb's choice of formats for a printed variable.

Data display requests take the symbolic form

```
address, count / format
```

or

```
address, count ? format
```

The first form reads data from the nodes themselves while the second accesses the actual executable on disk. For this reason the second form is to be preferred when looking at assembly code while the first is most common for actual data.

Essentially the format field of the command is an instruction which explains how to print data. This command is repeated count times starting at address. The address and count fields may contain any valid expression (as explained below) while the format field contains any number of modifier characters which denote how a particular datum is to be formatted. The particular characters and their interpretation is as follows

D	32 bit integer
d	16 bit integer
O	32 bit octal
o	16 bit octal
X	32 bit hex
x	16 bit hex
U	32 bit unsigned
u	16 bit unsigned
b	8 bit octal
B	8 bit binary
c	8 bit ASCII
C	8 bit ASCII with interpretation of control characters
s	Null terminated string
S	Null terminated string with control character interpretation
i	Disassemble instruction
I	Source module and line number (No '.' increment)
a	Symbolic address (No '.' increment)
p	Symbolic address
Y	Date and time
t	Tab
r	Space
n	Carriage return
+	Increment '.' by current amount
-	Decrement '.' by current amount
^	Backup '.' by current amount

Each of these characters can be preceded by a repeat count. ndb maintains an idea of the current address on each node which is referred to by the special symbol '.'. Each of the formatting instructions (except those specifically mentioned) increments this quantity by the size of the object to be displayed.

Several other commands of this format are allowed and are denoted by the modifiers listed below

`v`	Dump data as both hex and ASCII. The `count` field denotes the number of 16 byte lines of data to show. A repeat count before the v character requests hex data values of that length (in bytes), i.e., `4v` requests a dump in 32 bit words.
`l value mask`	This command searches from the given `address` through `count` bytes looking for a value which, when "AND"ed with `mask` is equal to `value`. The default search length is 4096 bytes. Warning: this option is VERY slow
`L value mask`	Searches for a 32 bit match. See previous modifier.
`w value`	Write the specified 16 bit `value` at `address`
`W value`	Write the given 32 bit `value` at `address`.

6 Expressions

ndb recognizes most of the usual arithmetic operators in expressions. Symbolic constants are also recognized with or without the preceding '_' added by the C compiler or the conversion to upper case performed in Fortran. The scope rules for simple variable names is to look in the current function (as denoted by the register PC or the most recent `func` command) and then the external variable table. References to local variables in other than the present function can be made by specifying a full symbol name of the form

```
function`variable
```

> **NOTE** that certain keywords are reserved for the use of ndb and thus cannot be used as variable names. Since none of these words begin with an '_' character the variable with the same name can always be referenced by including the underbar.

The various legal expression elements are

`.`	The value of the current address.
`<name`	The value of the named register.
`(exp)`	The value of the enclosed expression
`name`	Address of the named symbol using the scope rule that the function denoted by the current program counter is searched first followed by the external variable table. (Can be overridden with the `func` command.)
`routine`name`	The value of the variable `name` in the given subroutine which

must be in the current stack backtrace.

The following are allowed operators in expressions

+	Binary addition
-	Either binary subtraction or unary minus
*	Either a pointer dereference or binary multiplication
%	Binary integer division
&	Binary AND operator
|	Binary OR operator
^	Binary XOR operator
~	Unary NOT operator
#	Round first operand to next multiple of second
<<	Left shift.
>>	Right shift
@	Pointer dereference

In addition to using expressions to specify addresses it is also possible to use `ndb` as a regular integer calculator. The values of expressions are printed by following the expression with '=' and a format specifier as indicated in the previous section on displaying data. Thus

```
0x1234 + 16*(1<<3^{}2 - 3) = X
```

prints an uninteresting 32 bit hex value.

7 The "`show`" command

Various special commands have been added to the usual syntax to take advantage of some special features of the parallel machine. These commands are all of the form

```
show something
```

where the `something` is chosen from the list (Other options may be available on your system, type "`help show`" for details.)

`breakpoints`	List active breakpoints.
`pregs`	Internal processor communication registers.
`queues`	Unread messages for this node.
`regs`	General processor registers, current instruction and source file location.
`sets`	User defined sets.
`state`	Process state, current instruction and source file location.
`times`	Idle and active times in this processor.

Note that only an initial substring of the names listed above is necessary to pick options so that `show st` is equivalent to `show state`.

8 Arrays

The simplest way to print out array elements is with the `print` command described previously. If you have to resort to the lower level formatting commands for some reason then array indices are indicated in a different way.

This syntax follows the C and Fortran notation with the addition that the user must specify the declared dimensions of the array as well as the indices required. In C, therefore, the syntax to print out the element `lattice[2][4][5]` from a 10x 10x 10 array as a 32 bit integer is

```
lattice[2;10][4;10][5;10]/D
```

where the values after the semi-colons are the declared dimensions of the array. In order to perform offset calculations `ndb` has to know the size of an individual array element. By default the value is taken to be 4, correct for integer and 32 bit floating point data types. If the data item is actually of a different size - e.g., a byte or a structure then this can be specified in braces after the array name. Thus to print out values from an array of sixteen byte structures one might use

```
complicated{16}[3;8][4;12]/ddfff
```

The notation for Fortran style arrays is similar with the array indices being paired up with dimensions via semi-colons. An example might be

```
array(3;4, 5;120)/f
```

`ndb` understands the difference in ordering between multi dimensional arrays in C and Fortran as well as the fact that Fortran array indices start at 1. It also nests array dimensions arbitrarily deeply.

9 High level job control

Several high level commands are available for running and controlling user programs. The first set are used to start up either the debugger or the user application

`run string` The command indicated by the "`string`" parameter is executed and the debugger attempts to attach to the resulting process as though started with an appropriate "`-p`" option. If no `string` is given then the previous `run` command in re-executed. I/O redirection is allowed with the usual '<', '>', '>&', '>>' syntax.

`debug program process`
This command can be used to name the program that is to be debugged after `ndb` has started. This is useful if more than one executable is loaded into the machine since it allows switching between symbol tables. The `process` argument is optional and specifies the process-ID number of the program that is to be debugged. The `program` argument can be the single character '-' in which case no symbol table will be loaded but a process-ID may still be given.

`io` This command is used to redirect terminal input to the user program. Be default `ndb` reads all characters typed and interprets them as debugger commands. After this command all terminal input is sent to the user program. To issue further debugger commands use the keyboard interrupt sequence (usually CTRL-C) to return control to `ndb`.

`kill` Kill the program being debugged. Confirmation is requested.

As well as these functions commands are also available to control the execution of the user code at a finer level through breakpoints, single stepping etc. The commands are

`stop in name`
Insert a breakpoint at the first line of the named function.

`stop at number`
Insert a breakpoint at the indicated line in the current source file. Note that this command interacts with the `file` and `func` commands discussed earlier.

`stop variable`
Continue execution in single step mode and halt the program when the named variable changes value. This command executes rather slowly due to the interpretive nature of the processing required.

`cont` Continue execution from a breakpoint, or single step. This command interacts with the "`wait`" flag of the `ndbenv` command - by default the `ndb` prompt appears immediately allowing further commands to be entered. Alternately one can specify that `ndb` should continuously poll the nodes until another breakpoint is found or the application terminates before prompting again. This latter behavior is most common is sequential debuggers but slows down `ndb` somewhat as polling is an inherently slow process.

`step n` Single step the program over "n" lines of source code (default 1). If function calls are detected then the single stepping process enters each subroutine. If the current node "set" contains a single processor then this command will display the source lines as they are stepped past.

`next n` Single step over "n" lines of source code without entering any new functions. (Default 1 line). This option is similar to `step` but avoids the problems of having to single step through system functions etc. Source lines are displayed as processed if the current "set" contains a single node.

`status` Display the list of active breakpoints indicating which nodes they are present in, where they are placed and an index number used for deletions.

`delete n` Delete the breakpoint with index number "n" as determined from the `status` command.

Note that any of these commands may be prefixed by a "set" specification to allow different actions to be performed n distinct nodes. Thus to insert breakpoints in only the first three nodes one might use

```
on 0-2 stop at 23
```

10 Miscellaneous commands

Several miscellaneous commands are available to make debugging easier.

`sh string`
`! string` Any command line that begins with '!' or `sh` is executed by the shell.

`pwd` Show current directory.

`cd directory` Change to an new directory. This is occasionally useful for finding source files since the default search path starts with the "current directory".

`source file` Read `ndb` commands from the named source file. This is useful for performing repetitive tasks or for making data dumps. Consider also the `$>` command which redirects the output from the debugger. By default `ndb` attempts to find a file named `.ndbinit` in either your home directory or the current directory whenever started and reads initialization commands from it.

`alias s1 s2` Define a new command. Henceforth the command `s1` will be treated exactly like the command `s2`. The command

```
alias l list
```

for example, allows one to use the single character '`l`' instead of the `list` command. It is also possible to set up aliases with arguments and defaults using the UNIX C-shell syntax. The command

```
alias myuse use !:{1-.} !:{2-../src} !:3 !:4 !:5
```

defines a new command for setting the source code search path. 5 arguments are specified and the first two have defaults "`.`" and "`../src`" so that the simple command `myuse` can be issued without any arguments to set the search path to '`. ../src`' or arguments can be specified to set the path to other things.

`quit` Exit `ndb`. If the user program started within `ndb` a "`kill`" command will be given and you will be asked whether to terminate the program or not. If the program started outside of `ndb` it will be left alone.

11 The ndbenv command

This command defines the specific "environment" in which ndb is working. The currently implemented settings are the high level language being debugged, the "wait" state, the "repeat" mode and the "symbol match length". To see the options currently in effect type

```
ndbenv
```

which might yield

```
Language:                C
Wait mode:               FALSE
Autorepeat:              OFF
Symbol match length:     8192
```

Each of these options is explained below.

`Language` Certain features of ndb depend implicitly upon the high level language being debugged - for example array indexing and stack tracing. By default ndb is in the C mode suitable for the "C" language but may be switched over to Fortran with the command `ndbenv F`.

`Wait mode` This parameter controls the behavior of ndb upon receiving a `cont` or `run` command. By default the prompt immediately reappears and the user is able to enter further debugging commands while the node program continues to execute. If the wait state is set to `TRUE` with the command `ndbenv wait` then ndb continuously polls the nodes and only returns control to the user when all nodes have stopped at breakpoints or with some error. This mode can be turned off again with `ndbenv nowait`.

`Autorepeat` By default ndb repeats the last command entered whenever the user command is a single carriage return. This feature can be disabled with the command `ndbenv norep`.

`Symbol Match Length`
When translating memory address into variable names ndb uses a cutoff to avoid translating system memory addresses into user variable names - i.e., addresses further than this length above a known symbol will be translated into hex values rather than "`name+offset`". By default this cutoff is 32768 bytes. On occasions it may be necessary to increase this number so that large functions appear by name in stack traces rather than as hex numbers.

12 Assembly Level Debugging

In addition to the adb implementation effectively described in the next few sections the following commands are available for debugging at the machine code level.

`listi address` Display ten machine instructions from the given address. If none is given continue from the last address specified.

`stepi n` Similar to the `step` instruction but considers only the machine code. Encountered subroutines are entered and the

machine registers are displayed if the current node "set" contains only a single node.

`nexti n` Similar to `stepi` but passes over subroutine calls.

`stopi address` Place a breakpoint at the named address.

13 Assembly Level Job Control

Various commands are available which allow one to control the execution of a node program. They are all of the general form

```
arg1, arg2 : modifier string
```

in which `arg1` and `arg2` may be any general expression and the various modifiers are listed below. (Note that some cases do not require arguments in which case `arg1` and `arg2` can be omitted)

- `b` Set a breakpoint at address `arg1`. Note that only 8 breakpoints may be set in any node at one time so an attempt to set more will result in a request from `ndb` to delete an entry.
- `d` Delete the breakpoint at address `arg1`.
- `s` Step processor over a single machine instruction.
- `c` Continue as from a breakpoint.
- `C` Continue from breakpoint but instead of returning control to `ndb` immediately wait for the node specified as `arg1` to hit a breakpoint. If `arg1` is omitted wait for node 0.
- `k` Kill the process inside the machine.
- `K` Compare `arg2` bytes of memory starting at address `arg1` on all nodes in the active set.
- `r` Run the command specified by the `string` argument under the control of the debugger. I/O redirection is available with the usual constructions '>', '>>', '<' an, '>&'. Note the comment above on terminal input to the running process.
- `R` Run a command, as with the `r` specifier, above, but wait for the process specified in `arg1` to hit a breakpoint before returning control to `ndb`. If `arg1` is omitted, wait for node 0.

`ndb` leaves the debugger in control of the terminal even when continuing from breakpoints. This is contrary to conventional sequential debuggers which normally switch over to sending input to the debugged process whenever it is running. This distinction is made because of the distributed nature of parallel applications where it is not unusual to have some nodes running while looking at the state of others. If the running process requires terminal input the single command `io` switches control from `ndb` to the user process sending all further keyboard input to that process. To return control to the debugger use the interrupt sequence (usually `CTRL-C`).

14 Assembly Level System control

Various commands are available to control the way ndb interprets and outputs its results and to access some of the more machine specific requests. They all take the general form

```
arg1 $ modifier string
```

where arg1 is any legal ndb expression and the modifiers are as follows

- b List all active breakpoints. The notation for the nodes on which the breakpoint is active is essentially a bit mask with each bit (reading from left to right) denoting a single processor.
- c Traceback of all active C procedures together with their arguments interpreted as 32 bit hexadecimal constants.
- C As in the 'c' option above but prints out the values of all known local variables. Note that the appropriate compiler options must be used to compile information about local variables.
- d Set default base for numbers to 10.
- e Print out all external variables and their values interpreted as 32 bit hex constants.
- f Traceback of all active Fortran subroutine calls. No argument information is supplied by the compiler so the first few elements off the stack are interpreted as 32 bit hex constants.
- F Fortran traceback showing all local variables are 32 bit hex constants. Note that the appropriate compiler switch must be used to include information about local variables.
- m Print memory map of current program showing sizes of various data areas.
- n Show internal processor communication registers.
- o Set default base for input to octal
- q Quit from ndb. If you entered ndb via the -p command line option the node process is left alone. Otherwise it is killed.
- r Print general processor registers together with an interpretation of the current instruction and the source line/module information.
- s Set the maximum offset from the public symbol to arg1 for which ndb still interprets an address as being within that function.
- t Show a one line status summary for each processor showing the current state, program location and source file/line number information
- w Set the output page width to arg1.
- > Redirect output to the file named in string

WARNINGS

Error checking in ndb is rather primitive. Furthermore if an error is actually detected it will

quite probably be misdiagnosed. Certain words are reserved for use in commands and cannot, therefore, be used as variable names. The full list of reserved words is as follows: `on`, `setdef`, `pick`, `thru`, `to`, `set`, `node`, `even`, `odd`, `all`, `show`, `help`, `quit`, `io`, `neighbors`, `nof`.

Programs which put the nodes into strange states may also affect the debugger in odd ways.

DIAGNOSTICS

The prompt issued by `ndb` attempts to indicate the current set to which commands will be applied. Most variations are self-explanatory except the mysterious word `array` which indicates a node combination too complicated to figure out.

Syntactical errors on input generate many splendid messages, some of which might even complain about errors.

If no program is given on the command line a warning is issued about the lack of a symbol table.

Various out of memory errors produce both fatal and non-fatal diagnostics. Error recovery from these cases may or may not work.

Attempting to load a non-standard executable program will fail and produce a message suggesting corrective action.

BUGS

Printing non-floating point values with the `f` or `F` formats occasionally leads to core dumps. This sometimes happens even with legal floating point values under XENIX due to deficiencies in the run-time support.

The exact abilities of `ndb` depend a lot on the underlying operating system and hardware characteristics. As a result it is not possible to implement all features of `ndb` in all *Express* versions.

SEE ALSO

"NDB: A Guide to Parallel Debugging under *Express*."

NAME

`tcc` - Compile and link *Express* C and C++ programs for Transputers.

SYNOPSIS

```
tcc  [-B address] [-c] [-o outfile] [-Dname[=value]]
     [Idirname] [-Uname] [-E] [-g] [-dryrun] [-K] [-llibname]
     [-r] [-P] [-S] [-T0] [-T4] [-T8] [-e name] [-N] [-x] [-n]
     [@filelist] files...
```

DOMAIN

This command is available at the system prompt on the host processor.

DESCRIPTION

This command provides an interface to the Logical Systems C compiler useful for compiling programs to be run under *Express*. Filenames ending with the '`.c`' suffix are taken to be C source code and are compiled while those ending with '`.tal`' are treated as assembly code source and assembled. In both cases the resulting output files have the '`.trl`' suffix. Note that the preprocessor is run on assembler files by default allowing some of the advanced features of the Logical Systems assembler to be used.

After compiling all source files `tcc` proceeds, by default, to link the resulting object files into an executable program. If no '`-o`' switch is provided this will have the name `trans.tld`. By default libraries are searched which allow access to the *Express* communication routines only. The *Cubix* and *Plotix* subsystems are included with the `-lcubix` and `-lplotix` switches respectively. (It is important to note that programs linked with either of these switches *CANNOT* be executed with normal host programs in the "host-node" mode. Similarly programs compiled without one of these switches will not run with the `cubix` program.)

In addition to producing the executable image two additional files are (optionally) produced which have suffices '`.sym`' and '`.map`'. The former is used by the source level debugger `ndb` while the latter is of general interest - it contains information about the memory locations of program variables and which libraries and object files were searched.

By default all compilation/linking is performed for T800 transputers. Note that object files and libraries are not necessarily interchangeable between the two CPU types since instructions may be generated that are not supported on both types of hardware. The '`-T4`' switch is provided to force the generation of programs suitable for execution on T400 series systems. The '`-T0`' switch attempts to generate code independent of the transputer type by omitting instructions specific to only one model. Note that this switch does not support floating point operations.

OPTIONS

`-B address` Specify alternate load address for program. By default loading is performed at the beginning of "off-chip" memory. (See "`Excustom`" in the *"Express* users guide" for more information.)

`-c`	Compile only - do not proceed to link resulting object files.
`-dryrun`	Print the commands to be executed without actually performing any of them. This options implies both '`-x`' and '`-N`'.
`-Dname`	
`-Dname=value`	Define preprocessor symbol and optionally assign a value.
`-e name`	Specify an alternate entry point. By default the user program is entered through initialization routines required by *Express*.
`-E`	Run preprocessor only. Output is left in a file with the suffix '`.pp`'.
`-f2`	Use 32 bit floating point arithmetic for all "`double`" variables. (Default is 64 bit.) Other options are also available - see the Logical Systems documentation for more details.
`-g`	Include additional symbol table information for source level debugging. This switch adds additional code at entry and exit of *ALL* subroutine calls to enable stack-tracing which can significantly slow down execution.
`-K`	Disable stack tracing. Used to suppress stack tracing, even when '-g' switch is given.
`-Idirectory`	Add a directory to the path searched when looking for '`#include`' files.
`-lcubix`	Search the *Cubix* library for unresolved symbols in addition to that required by *Express*.
`-lplotix`	Search both *Cubix* and *Plotix* libraries for unresolved symbols.
`-n`	Execute the link phase of compilation on transputers rather than the host system. (Only available on some systems.)
`-N`	Keep all intermediate files. (Default is to delete them after use.)
`-o name`	Specify an alternate name for the executable program produced by the linker. Default is '`trans.tld`'.
`-P`	Run preprocessor only. Output remains in a file with the extension '`.pp`'.
`-r`	Generate "position-independent" code which can be relocated at runtime.
`-S`	Produce assembly code listing of C source program.
`-T4`	Compile for T400 series transputers.
`-T8`	Compile for T800 series transputers.
`-Uname`	Undefine a preprocessor symbol. Reverses the effect of '`-D`' switches or '`#define`' statements.
`-x`	Display all commands before executing them.
`@filename`	Take "`filename`" to be a file containing a list of source or object files to be compiled or linked, one name to a line.

-? Print usage message.

EXAMPLES

```
tcc -c hello.c
```

Compile, but do not link the C source file `hello.c`. The resulting object file will be called `hello.trl` and will be for the T800 series transputers.

```
tcc -o prog f1.trl f2.c f3.trl -lcubix
```

Compile file `f2.c` and proceed to link it with `f1.trl`, `f2.trl` and the *Cubix* libraries to make an executable program called `prog`. This executable will run on T800 transputers.

```
tcc -T4 -o prog4 g1.trl g2.c g3.trl -lcubix
```

This example is the same as the previous one but the resulting executable file, called `prog4` will run only on T400 series transputers. Notice that one cannot mix transputer types so the object files `g1.trl` and `g2.trl` must have also been produced with the `-T4` switch.

```
tcc -S -T4 foo.c
```

Generate an assembly code listing of the C source file `foo.c` suitable for a T400 transputer system.

```
tcc -Imyinc -DCUBIX -c noddy.c
```

Compile but do not link the C source code in the file `noddy.c` for a T800 series transputer system. Additionally define the `CUBIX` symbol and search the directory `myinc` when attempting to satisfy `#include` statements.

```
tcc -g -c noddy.c
```

Compile, but do not link, the file `noddy.c` for a T800 series transputer. Include both source line numbering information and also additional entry/exit subroutine calls to enable stack tracing. Note that the code resulting from this file will execute rather more slowly than would be the result if the '`-g`' switch were omitted.

```
tcc -o prog -g prog.bin subs.bin -lcubix
```

In this case the two named object files are linked together to produce an executable program called `prog`. In addition a symbol table called `prog.sym` will be created for use with the source level debugger, `ndb`.

MULTIPLE INPUT FILES

Some operating systems impose constraints on the length of a command line which preclude the linking of large programs with many input files using the standard `tcc` syntax. In this case `tcc` allows the list of filenames to be provided in a file and passed to the compiler using the '`@`' syntax. Consider, for example, a program made up of ten object files with names "`object0.trl`", "`object1.trl`" and so on up to "`object9.trl`". In this case we would create a file containing the ten lines

```
object0.trl
object1.trl
```

```
object2.trl
....
object9.trl
```

and save it with a name such as "`link.lst`". We could then invoke `tcc` with a command such as

```
tcc -o prog -g @link.lst -lcubix
```

to link the program with the *Cubix* libraries, build a symbol table for debugging and name the output file `prog`. Note that the suffix '`.lnk`' should not be used since `tcc` uses that name internally.

NAME

`tcc3l` - Compile and link *Express* C Transputer node programs.

SYNOPSIS

```
tcc3l [-B address] [-c] [-o outfile] [-Dname[=value]]
      [-Idirname] [-Uname] [-dryrun] [-i] [-g] [-llibname]
      [-T4] [-T8] [-x] [-N] [@filelist] files...
```

DOMAIN

This command is available at the system prompt on the host processor.

DESCRIPTION

This command provides an interface to the 3L C compiler useful for compiling programs to be run under *Express*. Filenames ending with the '`.c`' suffix are taken to be C source code and are compiled.The resulting output files have the '`.bin`' suffix.

After compiling all source files `tcc3l` proceeds, by default, to link the resulting object files into an executable program. If no '`-o`' switch is provided this will have the name `trans.tld`. By default libraries are searched which allow access to the *Express* communication routines only. The *Cubix* and *Plotix* subsystems are included with the `-lcubix` and `-lplotix` switches respectively. (It is important to note that programs linked with either of these switches *CANNOT* be executed with normal host programs in the "host-node" mode. Similarly programs compiled without one of these switches will not run with the `cubix` program.)

In addition to producing the executable image two additional files are (optionally) produced which have suffices '`.sym`' and '`.map`'. The former is used by the source level debugger `ndb` while the latter is of general interest - it contains information about the memory locations of program variables and which libraries and object files were searched. The map file also contains the error messages, if any, from the linker. If the program aborts with a message such as

```
Failed to find .b4 file
```

this usually indicates that the link process failed with some error which can be located by searching for the string "`ERROR`" in the "`.map`" file.

By default all compilation/linking is performed for T800 transputers. Note that object files and libraries are not necessarily interchangeable between the two CPU types since instructions may be generated that are not supported on both types of hardware. The `-T4` switch is provided to force the generation of programs suitable for execution on T400 series systems.

It is important to note that the 3L compilers/linkers execute directly on the first transputer in the attached network and destroy and *Express* programs executing there, including the *Express* kernel itself. It is necessary to re-load the system with the `exinit` command before running any program.

OPTIONS

`-B`	Specify alternate load address for program. By default loading is performed at the beginning of "off-chip" memory.
`-c`	Compile only - do not proceed to link resulting object files.
`-dryrun`	Print the commands to be executed without actually performing any of them. This options implies both '`-x`' and '`-N`'.
`-Dname` `-Dname=value`	Define preprocessor symbol and optionally assign a value.
`-g`	Include additional symbol table information for source level debugging. Used at link time to force the generation of a symbol table for use with the debugger, `ndb`.
`-i`	Do not search any of the built-in default directories for include files. Rely solely on the definition of the `3LCC_INC` environment variable.
`-Idirectory`	Add a directory to the path searched when looking for '`#include`' files.
`-lcubix`	Search the *Cubix* library for unresolved symbols in addition to that required by *Express*.
`-lplotix`	Search both *Cubix* and *Plotix* libraries for unresolved symbols.
`-N`	Keep all intermediate files instead of deleting them.
`-o name`	Specify an alternate name for the executable program produced by the linker. Default is '`trans.tld`'.
`-T4`	Compile for T400 series transputers.
`-T8`	Compile for T800 series transputers.
`-Uname`	Undefine a preprocessor symbol. Reverses the effect of '`-D`' switches or '`#define`' statements.
`-x`	Generate a listing of all command lines before they are executed. This option is useful if certain commands need to be run by hand.
`@filename`	Take "`filename`" to be a file containing a list of source or object files to be compiled or linked, one name to a line.
`-?`	Print usage message.

INCLUDE FILE PROCESSING

The rules regarding the searching for include files in the 3L compiler are quite tricky. On UNIX systems some attempt is made to locate system include files according to the customization information supplied when installing the system. While this method is usually effective it can lead to extremely long command lines which cannot be processed by the 3L compiler. To avoid this situation the '`-i`' switch should be given, which suppresses the default search completely. In this case only those directories specified in the `3LCC_INC` environment variable will be searched when looking for include files.

Under MS-DOS no attempt is made to locate default include file directories since the resulting command lines are nearly always too long for processing. In this case `tcc3l` will not execute unless the `3LCC_INC` variable is defined. If no such variable is found a suggestion will be made as to the correct assignment.

EXAMPLES

```
tcc3l -c hello.c
```

Compile, but do not link the C source file `hello.c`. The resulting object file will be called `hello.bin` and will be for the T800 series transputers.

```
tcc3l -o prog f1.bin f2.c f3.bin -lcubix
```

Compile file `f2.c` and proceed to link it with `f1.bin`, `f2.bin` and the *Cubix* libraries to make an executable program called `prog`. This executable will run on T800 transputers.

```
tcc3l -T4 -o prog4 g1.bin g2.c g3.bin -lcubix
```

This example is the same as the previous one but the resulting executable file, called `prog4` will run only on T400 series transputers. Notice that one cannot mix transputer types so the object files `g1.bin` and `g2.bin` must have also been produced with the `-T4` switch.

```
tcc3L -S -T4 foo.c
```

Generate an assembly code listing of the C source file `foo.c` suitable for a T400 transputer system.

```
tcc3l -Imyinc -DCUBIX -c noddy.c
```

Compile but do not link the C source code in the file `noddy.c` for a T800 series transputer system. Additionally define the `CUBIX` symbol and search the directory `myinc` when attempting to satisfy `#include` statements.

```
tcc3l -o prog -g prog.bin subs.bin -lcubix
```

In this case the two named object files are linked together to produce an executable program called prog. In addition a symbol table called prog.sym will be created for use with the source level debugger, ndb.

```
tcc3l -o prog -B0x80000ff8 prog.bin -lcubix
```

Link the named object file into a program called `prog` which will execute with its stack placed in the fast "on-chip" memory of the transputer, at address `0x80000ff8` As the program executes the stack will grow down from this address towards the bottom of memory.

MULTIPLE INPUT FILES

Some operating systems impose constraints on the length of a command line which preclude the linking of large programs with many input files using the standard `tcc3l` syntax. In this case `tcc3l` allows the list of filenames to be provided in a file and passed to the compiler using the '`@`' syntax. Consider, for example, a program made up of ten object files with names "`object0.bin`", "`object1.bin`" and so on up to "`object9.bin`".

and save it with a name such as "`link.lst`". We could then invoke `tcc3l` with a command such as

```
tcc3l -o prog -g @link.lst -lcubix
```

to link the program with the *Cubix* libraries, build a symbol table for debugging and name the output file `prog`. Note that the suffix '`.lnk`' should not be used since `tcc3l` uses that name internally.

DIAGNOSTICS

If the linking procedure fails for some reason a rather uninformative message similar to

```
Failed to find .b4 file
```

is often generated. In this case the "`.map`" file should be consulted for error messages. (A good way to do this is to search for the string "ERROR" with a text editor or similar.)

NAME

`tfc` - Compile and link *Express* FORTRAN Transputer node programs

SYNOPSIS

```
tfc  [-c] [-o outfile] [-g] [-llibname] [-T4] [-T8]
     [-dryrun] [-x] [-N] [@filelist] files...
```

DOMAIN

This command is available at the system prompt on the host processor.

DESCRIPTION

This command provides an interface to the 3L FORTRAN compiler useful for compiling programs to be run under *Express*. Filenames ending with the '`.f`' suffix are taken to be FORTRAN source code and are compiled. The resulting output files have the '`.bin`' suffix.

After compiling all source files `tfc` proceeds, by default, to link the resulting object files into an executable program. If no '`-o`' switch is provided this will have the name `trans.tld`. By default libraries are searched which allow access to the *Express* communication routines only. The *Cubix* and *Plotix* subsystems are included with the `-lcubix` and `-lplotix` switches respectively. (It is important to note that programs linked with either of these switches *CANNOT* be executed with normal host programs in the "host-node" mode. Similarly programs compiled without one of these switches will not run with the `cubix` program.)

In addition to producing the executable image two additional files are (optionally) produced which have suffices '`.sym`' and '`.map`'. The former is used by the source level debugger `ndb` while the latter is of general interest - it contains information about the memory locations of program variables and which libraries and object files were searched.

By default all compilation/linking is performed for T800 transputers. Note that object files and libraries are not necessarily interchangeable between the two CPU types since instructions may be generated that are not supported on both types of hardware. The `-T4` switch is provided to force the generation of programs suitable for execution on T400 series systems.

OPTIONS

`-c`	Compile only - do not proceed to link resulting object files.
`-dryrun`	Print the commands to be executed without actually performing any of them. This options implies both '`-x`' and '`-N`'.
`-g`	Include additional symbol table information for source level debugging. If specified at link time force the generation of the '`.sym`" file for debugging.
`-lcubix`	Search the *Cubix* library for unresolved symbols in addition to that required by *Express*.
`-lplotix`	Search both *Cubix* and *Plotix* libraries for unresolved symbols.

`-N`	Keep all intermediate files instead of deleting them.
`-o name`	Specify an alternate name for the executable program produced by the linker. Default is '`trans.tld`'.
`-T4`	Compile for T400 series transputers.
`-T8`	Compile for T800 series transputers.
`-x`	Print each command before executing it.
`@filename`	Take "`filename`" to be a file containing a list of source or object files to be compiled or linked, one name to a line.
`-?`	Print usage message.

EXAMPLES

```
tfc -c hello.f
```

Compile, but do not link the Fortran source file `hello.f`. The resulting object file will be called `hello.bin` and will be for the T800 series transputers.

```
tfc -o prog f1.bin f2.f f3.bin -lcubix
```

Compile file `f2.f` and proceed to link it with `f1.bin`, `f2.bin` and the *Cubix* libraries to make an executable program called `prog`. This executable will run on T800 transputers.

```
tfc -T4 -o prog4 g1.bin g2.f g3.bin -lcubix
```

This example is the same as the previous one but the resulting executable file, called `prog4` will run only on T400 series transputers. Notice that one cannot mix transputer types so the object files `g1.bin` and `g2.bin` must have also been produced with the `-T4` switch.

```
tfc -o prog -g prog.bin subs.bin -lcubix
```

In this case the two named object files are linked together to produce an executable program called `prog`. In addition a symbol table called `prog.sym` will be created for use with the source level debugger, ndb.

MULTIPLE INPUT FILES

Some operating systems impose constraints on the length of a command line which preclude the linking of large programs with many input files using the standard `tcc` syntax. In this case `tfc` allows the list of filenames to be provided in a file and passed to the compiler using the '`@`' syntax. Consider, for example, a program made up of ten object files with names "`object0.bin`", "`object1.bin`" and so on up to "`object9.bin`". In this case we would create a file containing the ten lines

```
object0.bin
object1.bin
object2.bin
....
object9.bin
```

and save it with a name such as "`link.lst`". We could then invoke `tfc` with a command such as

```
tfc -o prog -g @link.lst -lcubix
```

to link the program with the *Cubix* libraries, build a symbol table for debugging and name the output file `prog`. Note that the suffix '`.lnk`' should not be used since `tfc` uses that name internally.

DIAGNOSTICS

If the linking procedure fails for some reason a rather uninformative message similar to

```
Failed to find .b4 file
```

is often generated. In this case the "`.map`" file should be consulted for error messages. (A good way to do this is to search for the string "ERROR" with a text editor or similar.)

NAME

`xtool` - Analyze Execution Profile

SYNOPSIS

```
xtool program_name [log_file_name]
```

DOMAIN

This command is available at the system prompt on the host processor.

DESCRIPTION

This command is used to examine and analyze the log file created with the execution profiler, `xprof`, commands. The first argument is the name of the executable program to be profiled and the second is the name of the file containing the profile data. This may be omitted if it has the default value "`xprof.out`". Note that the execution profiler relies on data contained in a symbol table for correct functioning. This can usually be made by specifying the '`-g`' switch when linking the program - the same procedure as used for debugging with "`ndb`".

This command presents a separate table on `stdout` from each node. The information contained in each table is:

- An identifier showing which node the following data is from.
- A summary of the busy and idle time in each processor. In this regard we measure CPU time so that the only "idle" time is when the CPU is not actively executing the process such as when waiting for a message to arrive. All other classes of activity are counted as "busy". Note that this interpretation is different from that of `ctool` which distinguishes between calculation and communication time.
- A count of the number of profiling "misses". Since the buffer supplied to the profiling function `profil` may not be large enough to encapsulate the entire program it is possible that the execution profiler will "miss" occasionally - i.e., the program will be executing at an address which lies outside the region mapped by the `profil` call when it tries to log the profile event. In this case the "miss" counter is incremented. The ratio of hits to misses is presented to give a guide to the effectiveness of the profile obtained - a lot of misses means that the routines in the profile list may not, in fact, be the most heavily used.
- A profiling list containing the most heavily used 20 functions in the program. Each shows the fraction of the total profiling events that it corresponds to.

This data appears on `stdout`.

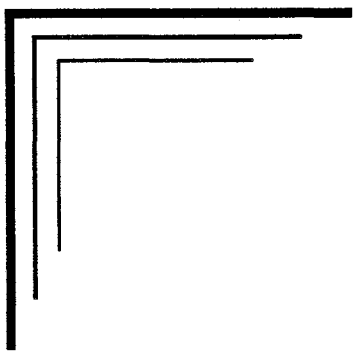
EXAMPLES

To examine the profile data in a file called "`phase3.prof`" created by the program `master` execute the command

```
xtool master phase3.prof
```

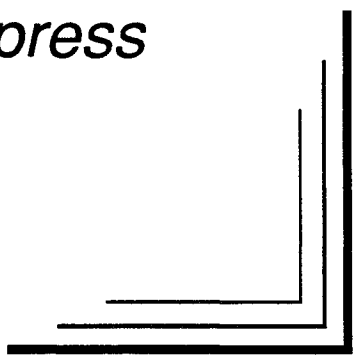
SEE ALSO

`cubix` in the *Express* documentation.



C runtime library

Library routines available to *Express* programs written in C



This (large) section of the manual is devoted to a listing of the contents of the subroutine library which is invoked by *Express* programs.

Since parallel processing is an inherently complex activity the capabilities of *Express* are correspondingly broad. This, in turn, leads to a very extensive set of functions which may appear daunting to those familiar with other parallel processing systems or totally unfamiliar with parallel computing. These users should not, however, be put off by the long list of routines given in this section since we have found that practical applications use only a small fraction of the available number. Unfortunately different programs tend to use different small subsets of the total list which makes predictions difficult.

As a help in selecting the appropriate functions we have tried to indicate routines with similar or related functionality in the **"SEE ALSO"** section at the end of each manual page. In conjunction with the full manual and the numerous **"EXAMPLES"** this should give a reasonable guide.

One of the most important pieces of information contained on each manual page is in the **"DOMAIN"** section. This paragraph tells you whether the routine in question is available to programs running on the "host" computer or to those running on the nodes of the parallel computer system. In the latter case there is also an indication of which library switch is required to gain access to the routine. Note that this information must be used in conjunction with that concerning the "Host-Node" and/or *Cubix* programming models.

If you are using the former style of computation then only routines shown as appearing in all node libraries may be called from your "node" programs. Routines shown as appearing in the *Cubix* or *Plotix* libraries cannot be called from such programs.

On the other hand, if you are using the *Cubix* model of computation you may freely call *nearly all* of the routines described in this manual being careful only to specify the *Plotix* libraries for the graphical routines. The exceptions in this case are those routines which specifically interface to similar routines in the host processor - since you will not be writing a program to run on the "host" you cannot call the corresponding routine there! Typical examples are the `cp/elt` combinations such as `cprofcp` and `cprofelt`. To achieve the effect of these routines in *Cubix* programs one would instead use `cprof_end`.

The information regarding which routines are available in which libraries and to which type of programming models they belong is summarized in a later section of this manual where we also show the correspondence between the various language variants of *Express*.

The various routines contained in *Express* can be classified according to their functionality in several broad categories. The following sections attempt to indicate some of the important features of each and also to supply, in a condensed form, some information about important *Express* parameters and the header files necessary to use them.

1 High Level Communication System

This section describes the communication system available to application programs under *Express*. Several levels of functionality are provided although some features are common to all. While one may use the system to send messages to specific destination processors by specifying their processor numbers one can also use the primitives in a "topology-independent" manner. The `exgrid` system allows problems to be specified in the domain of the user data structures and can be used to determine processor numbers automatically for use in the communication primitives.

Using this system it is possible to design applications that have absolutely no knowledge of the underlying hardware topology and which will, in fact, execute transparently on any hardware that supports *Express*. Similar routines are available to dynamically configure an application to the available processing resources at runtime.

Several concepts underpin the entire communication system and can be summarized as follows;

- All messages have "destinations". This merely specifies the node to which the message will be sent.
- All messages are "typed". As well as the information concerning what data is to be sent and to whom every message has a `type` field which allows receivers to distinguish between various messages.
- The message reception mechanism has an "acceptance" criterion. All read routines may specify source and type information which constrains the range of messages which may be read. This information may either limit attention to specific node/type combinations or various degrees of "dontcare" behavior may be specified using the `DONTCARE` macro.
- Both blocking and non-blocking read functions are supplied.
- Messages are "atomic". A single read operation corresponds to a single write operation. If the sender transmits more data than is read then the excess are discarded and may NOT be read with another read request. If less data are sent than were requested then the message is read and a return code indicates the discrepancy - another read request may not make up the difference unless another write request is also made.

On a more functional level the following generalities may also be observed:

- The "node" and "type" information associated with a message are always supplied in pointer form - even when this seems unnecessary. In read requests, for example, pointer values are required since a wildcard specification will be overwritten with the actual parameter value. In write requests, however, this is superfluous. The pointer standard is adopted for consistency.
- The general ordering of arguments is: what, how much, where and type - i.e., the first arguments specify what data is to be transmitted, the second how much, the third indicates to whom the data should be sent and the last argument denotes the type of the associated message. This standard leads to an obvious calling sequence for the simplest "read" and "write" operations

```
exread(buffer, length, &source, &type);
exwrite(buffer, length, &dest, &type);
```

(Note the pointer arguments for the last two). Some calls which both read and write data have the above sequence duplicated for both operations so the `exchange` function, for example, has the calling sequence

```
exchange(inbuffer, inlength, &source, &intype,
         outbuffer, outlength, &dest, &outtype);
```

2 Hardware Dependent Communication System

Express has been carefully designed to allow programs to be written which will execute transparently on a wide variety of different parallel architectures. As a result, none of the basic primitives make any reference to the underlying hardware configuration. On occasions, however, portability may be a less important goal than absolute performance on a particular piece of hardware. To support those users who have this type of constraint an *Express* subsystem has been provided with a "raw" interface to the communication hardware. Typically the use of these routines disables most of the higher level processing of which *Express* is capable such as the debugging and performance analysis tools. For this reason we suggest that its use be adopted as the final stage in the development of any parallel processing project after whatever bugs and/or performance questions have been resolved by using the full *Express* system.

3 Synchronization

One of the key concepts which underlies all of *Express* concerns interprocessor synchronization. In some sense this issue is the key to all of parallel processing - different algorithms can most often be classified not by the particular scientific or other field from which they arise but by the way in which they necessitate interprocessor synchronization. In *Express* we classify two types of behavior:

- Asynchronous
  "Asynchronous" system calls can be made in any node at any time regardless of the activities currently occurring in other nodes. One can consider that the node making the call is operating totally in isolation.
- Loosely synchronous
  A "loosely synchronous" system call can be perceived as a barrier to the further progress of the program. When one node makes a loosely synchronous call it waits for all other nodes to make the *same* system call (albeit with possibly different arguments). When all nodes have made the call every node proceeds. This concept might be classed "synchronous" but this is too restrictive - it is quite permissible for one node to make the "loosely synchronous" call far ahead of the other nodes. All nodes will, however, be synchronized *after* the call completes.

Note that these behaviors are not (usually) *states* of the system but are applied individually to different function calls. The function `exwrite`, for example, which sends an *Express* message may always be made asynchronously - i.e., any node may send a message at any time. Similarly any node may call `abort` to terminate a program at any time. On the other hand, `fmulti`, the system call which switches between file I/O modes must *always* be made "loosely synchronously".

Because the synchronization properties of a parallel program are often the key to its construction and optimization, the situation is actually more complex than just discussed.

The default state of *Express* is that every system call has an associated synchronization property. These states are listed in section 3 of this manual. Also available (in the *Cubix* library) is a global override function, `syncmode`, which switches all system calls to asynchronous mode.

At a slightly more useful level, each open file has its own synchronization property. This allows, for example, a program to have a global input stream for basic parameters, individually (and

asynchronously) accessed data files for operational data and error reporting, distributed (synchronized) files for output data, etc. In each of these modes different requirements are made by *Express* on what can and cannot be done to the files.

Even within the "asynchronous" functions there are different levels of behavior. The `exwrite` function mentioned earlier, for example, may be called at any time in a user program, but it does not return to its caller until a message has been transmitted to the receiving node. The analogous `exsend` system call also sends a message to another processor but returns immediately to its caller without waiting for the data to be transmitted. While both of these calls are "asynchronous" in the sense that the start of the operation may occur in any node at any time `exsend` is clearly "more" asynchronous than `exwrite` because the point at which the buffer containing the data which has been sent can be re-used is not known when the call returns.

This discussion may have convinced you that the topic of interprocessor synchronization is too complex to ever be understood. This is not, however, the case. While it is true that many of the elementary bugs in *Express* are caused by violations of some synchronization constraint they are remarkably easy to find and eliminate using tools like the *ParaSoft* debugger, `ndb`. Furthermore, the existence of these synchronization constraints tends to help rather than hinder the development process. Much care has gone into the I/O system, for example, to make the synchronization as natural as possible. Typically we find that the message

```
abort(-1)
```

(which is the response of the `cubix` program to a violation of a synchronization rule) is indicative of an error in the user application which might otherwise have gone unnoticed or else caused other problems to occur later on.

4 Decomposition Tools

This section describes the utilities used to automatically distribute problems among parallel processors. The *Express* manuals describe a set of communication primitives designed to allow "topology independent communication". Problems can be specified in their own natural domain - two dimensions for image processing applications and three for aircraft simulation, for example. The utilities in this section are then provided to assign the "processor numbers" used by the communication routines described in the previous section.

Also available is another utility which allows applications access to certain important runtime parameters. In conjunction with the other utilities this allows programs to be dynamically configured, at runtime, to the system on which they execute. This allows, for example, a program developed on a four processor system to be run on a 1000 node production machine by merely changing a single command line parameter.

5 Multitasking Support

Express supports a powerful *remote* multitasking facility which allows programs running on any processor in the system to initiate a "task" on another node of the parallel computer. This system is built around the `exhandle` function which associates a program segment with a particular message type. Upon the arrival of a registered message type the indicated program section is triggered as a separate task which is then free to pursue its own independent execution path.

In support of this multitasking facility is a set of semaphore operations designed to allow two or

more processes on a node to cooperatively update shared data. These routines use the data type EXSEM, defined in the standard header file `express.h`.

6 Processor Allocation and Control

This section describes in detail the control functions at the lowest levels of *Express*. They are used in "Host" programs to allocate groups of processors, load programs and start execution. Note that this section will not concern you if you intend to use the *Cubix* programming model since the `cubix` program takes care of the necessary steps automatically.

The unifying concept of this section is that of the ***processor group***. This is the fundamental unit of processor allocation - processors are allocated to processor groups which are then treated as a unit. When programs are to be loaded into processor groups the ***processor group index*** must be specified.

```
#include "express.h"

main()
{
     int pgindex;

/* Allocate four transputer nodes anywhere in array */

     if((pgindex=exopen("/dev/transputer", 4, DONTCARE))<0)
{
         fprintf(stderr, "Failed to allocate processors\n");
         exit(2);
     }

/* Load application, "noddy" into all processors */

     if(exload(pgindex, "noddy") < 0) {
         fprintf(stderr, "Failed to load application\n");
         exit(3);
     }
```

As well as allowing a single host process to allocate and manipulate more than one group of processors it is also possible for two or more users to simultaneously allocate and work with groups of processors. (Provided, of course, that the host operating system allows multitasking. This features is not, for example, supported under MS-DOS.) It is even possible for multiple host processes to share access to the same group of processors. This mechanism allows multiple, disjoint, front end processes such as a file serving utility and a complex graphical user interface to both have access to the same group of nodes. Routines are available to ensure that the processes do not interfere with each other.

All the routines in section return -1 to indicate errors. Some also write diagnostic messages and some cause immediate termination of the calling process. In any case the parallel machine should

remain intact and available for use by other applications and users.

7 I/O Services

The *Cubix* I/O library is available to programs using the *Cubix* programming model and associated server process. It provides a full set of C style I/O utilities as well as many extensions designed explicitly for parallel processing. Only the latter are fully documented here.

The following routines are provided from Section 2 of the UNIX programmers manual which is the best reference for the arguments and functionality of the routines.

_exit	creat	ftime	kill	pipe	sync
abort	dup	getgid	link	read	system
access	dup2	getpid	lseek	setgid	time
chdir	errno	getuid	mknod	setuid	umask
chmod	fcntl[4]	gtty	nice	stat	unlink
chown	fstat	ioctl	open[4]	stty	write
close					

(For the interpretation of the superscript notes, see below.)

Note that, by default, each routine must be called "loosely synchronously" in all processors with identical arguments. Note that the syncmode function can be used to relax this constraint.

The following is the list of supported functions from the Standard C library for C programs as specified in Chapter 4 of the draft ANSI standard (X3J11). Unless noted these functions may be called asynchronously in the processing nodes since they modify data structures local to the individual processors.

NaN	atexit	atof	atoi
exit[5]	fasync[1]	fclose[1]	feof
ferror	fflush[1]	fgetc[3]	fgetpos[1]
fgets[3]	fileno	finite	fmulti[1]
fprintf[2]	fputc[2]	fputs[2]	fread[3]
fscanf[3]	fseek[1]	fsetpos[1]	ftell[1]
fwrite[2]	getc[3]	getchar[3]	getenv
gets[3]	isalnum	isalpha	iscntrl
isdigit	isgraph	islower	ismulti
isprint	ispunct	isspace	isupper
isxdigit	malloc_avail	malloc_debug	malloc_print
malloc_verify	printf[2]	putc[2]	putchar[2]
puts[2]	rand	remove[1]	rename[1]
scanf[3]	setvbuf[4]	sprintf	sscanf
strerror	strtod	strtol	strtoul
tmpnam	ungetc[3]	vfprintf[2]	vfscanf[3]

NOTES:

1. These routines must be called "loosely synchronously" and with identical arguments in each node unless the stream argument is in `async` mode.
2. Must be called "loosely synchronously" and with identical arguments in each node unless the stream argument is in `async` or `multi` modes.
3. Must be called "loosely synchronously" in all processors unless the stream argument is in `async` mode. If the stream argument is in `multi` mode the arguments may differ from node to node but the function must still be called "loosely synchronously".
4. These functions have arguments in the *Cubix* I/O system which may not be supported on normal "host" systems and their use should be monitored carefully if program portability is to be maintained.
5. These functions must be called "loosely synchronously" and with identical arguments in every node unless the `syncmode` function has been used to initiate asynchronous processing.

8 Graphics

The *Plotix* library is supplied to allow both parallel programs running in the *Cubix* programming model and "host" programs access to device independent graphics in a portable manner. The library contains about twenty routines which are sufficient to cover the majority of graphical tasks while not being an implementation of any particular standard.

9 Header files, macros, variables, etc.

Central to the use of *Express* is the C header file "`express.h`" which should be included whenever *Express* functions are being used. This file defines a number of important parameters which have wide usage in the system.

`DONTCARE` This macro value is used to indicate that the source or type of an incoming message are of no consequence. Note that it is illegal to send a message with type `DONTCARE` even if you really don't care!

`PROCNUM_ORDER`
This macro value is used to indicate that the parallel I/O system should operate in a mode in which data from nodes is sent to and/or received from the host in order of increasing processor number. It is used in conjunction with the `forder`, `mread` and `mwrite` function calls.

`NONODE` This macro value is used by the `exgrid` functions to indicate that no node is attached to the user decomposition in the indicated direction. Such a case might arise, for example, when solving partial differential equations on a finite space - some nodes have no neighbors in some directions since they lie on the edge of the domain.

`NULLPTR` A macro value equivalent to the `NULL` pointer. Useful in a number of situations where *Express* would normally return information (in the

	`src` or `type` fields of a call to `exread` or the `sizes` argument to `exconcat`, for example) but the caller has no interest in the associated data.
`HOST`	An integer variable which contains the "processor number" used by node programs to send/receive data to the host. By default this is the machine that loaded the node program although it is possible to override this.
`HOSTMASK`	A macro value that can be used to create the "processor numbers" of any host within a multi-host system. When configured *Express* assigns small integer values to the various hosts present in the system and the "logical OR" of this value with `HOSTMASK` will create a processor number suitable for communication with the processor.
`ALLNODES`	A macro value used in the "global" communication routines (`excombine`, `exbroadcast`, `exconcat`, etc.) to indicate that a particular operation should be applied to all nodes in a parallel processing system. Never includes the host processor(s).
`ALLPROCS`	A macro value similar in use to `ALLNODES` except that it implicitly includes all host processors attached to the system.

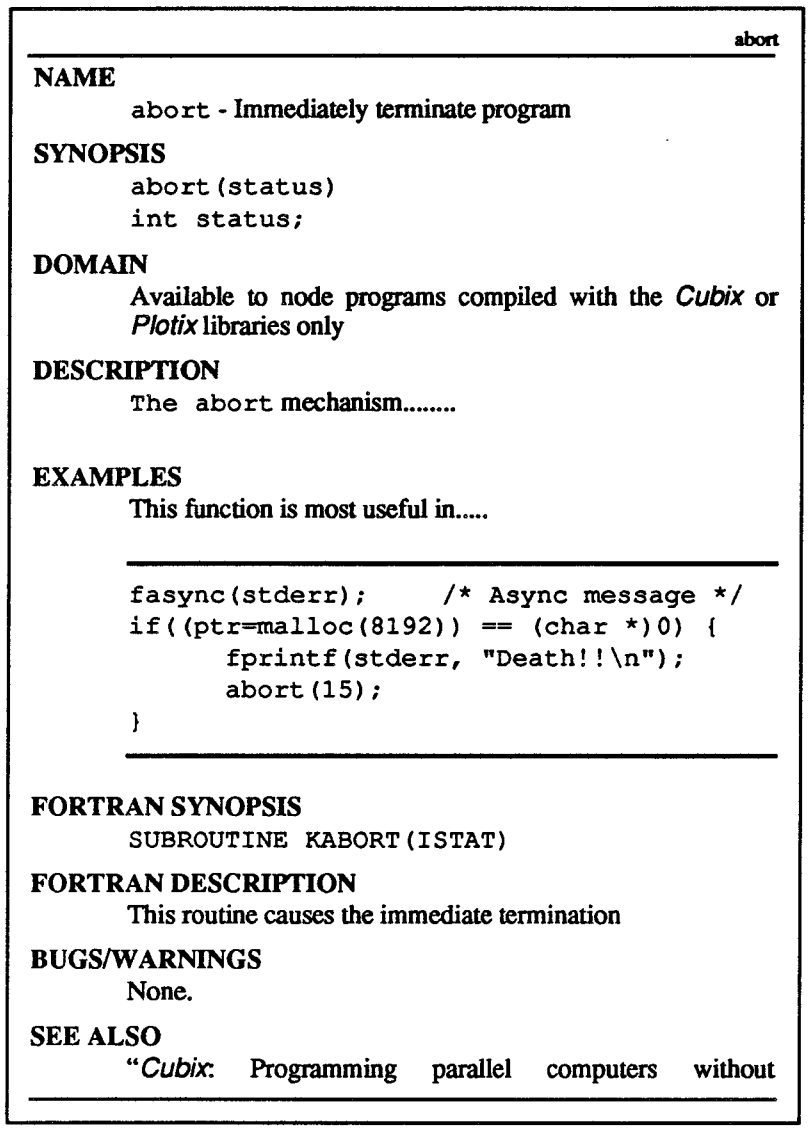
In addition to these variable/macro values `express.h` defines a number of important variable types.

`struct nodenv`	A data structure whose contents indicate the run-time parameters associated with the executing program, the number of nodes it is using etc. The values are obtained with the `exparam` function call.
`EXSEM`	The "semaphore" structure manipulated by the operations `exsemsig`, `exsemwait`, etc. to implement the mutual exclusion primitives important in multitasking operations.
`ETOGGLE`	The "toggle" structure used by the event profiler to measure CPU time and other statistics in indicated program segments.

10 Manual Page Layout

The manual pages are, for better or worse, modeled after those often found in UNIX documentation which means that each manual page has several well-defined sections. The overall structure is

shown below.



abort

NAME

`abort` - Immediately terminate program

SYNOPSIS

```
abort(status)
int status;
```

DOMAIN

Available to node programs compiled with the *Cubix* or *Plotix* libraries only

DESCRIPTION

The `abort` mechanism.....

EXAMPLES

This function is most useful in.....

```
fasync(stderr);     /* Async message */
if((ptr=malloc(8192)) == (char *)0) {
      fprintf(stderr, "Death!!\n");
      abort(15);
}
```

FORTRAN SYNOPSIS

```
SUBROUTINE KABORT(ISTAT)
```

FORTRAN DESCRIPTION

This routine causes the immediate termination

BUGS/WARNINGS

None.

SEE ALSO

"*Cubix*: Programming parallel computers without

Header contains the name of the manual page which is usually the same as the routine described.

The various sections and their contents are:

NAME	Repeats the name associated with the manual page and a brief one-line description of the purpose of the associated routines
SYNOPSIS	Summarizes the arguments used by the indicated routines. If more than one routines is described on a particular page then all are listed in this section
DOMAIN	Describes the libraries in which the routine is to be found and any restrictions on when it may be used.
DESCRIPTION	Describes the purpose of each routine and lists the actions caused by its most important arguments. This section is the most important reference material

for each command.

EXAMPLES One of more examples of the use of each routine are shown on each manual page. This section probably represents the best information on how the various arguments are put together in "real" examples. This section is also useful for demonstrating the order in which function calls should be made and which ones are necessary at which points in the execution of *Express* programs.

FORTRAN SYNOPSIS

Whenever a similar routine is available in FORTRAN its arguments and type are shown. In most cases the use of the arguments exactly parallel those of the equivalent C routine.

FORTRAN DESCRIPTION

If the arguments or usage of a FORTRAN function differ from those of the C counterpart this section attempts to explain the differences. Hopefully this information will be sufficient to allow C programmers to make the transition to FORTRAN but if not, the FORTRAN reference manual contains explicit examples and a more detailed discussion of the FORTRAN calling sequences.

WARNINGS If the routine has peculiar side effects or is "dangerous" in some way it will be noted in this (optional) section. Any non-intuitive behavior is also noted here.

BUGS Currently known bugs and/or unimplemented routines are noted in this (optional) section.

SEE ALSO Related commands and/or routines from the various *Express* libraries are noted in this section. Using this information is usually the quickest way to build a complete picture of the interaction between the various utilities.

NAME

`abort` - Immediately abort program

SYNOPSIS

```
abort(status)
int status;
```

DOMAIN

This routine may only be called in programs compiled with the *Cubix* or *Plotix* libraries.

DESCRIPTION

The `abort` function causes the immediate termination of the parallel program. A message is displayed on the host processor showing the processor number of the aborting node and also the `status` value supplied to the call. If examined with the debugger, `ndb`, the aborting nodes will appear to be at breakpoints.

EXAMPLE

This function is most useful for dealing with asynchronous problems which require termination of the program but which might not occur in every node. If the termination condition is known to occur in each node the regular system call, `exit` should probably be preferred. A common problem that often merits abort is the failure of the memory allocator malloc. Code such as the following prints a warning message and then halts processing via an abort call.

```
fasync(stderr);              /* Async message */

if((ptr=malloc(8192)) == (char *)0) {
        fprintf(stderr, "Out of memory !!\n");
        abort(15);
}
else {      /* Process new memory block */
```

FORTRAN SYNOPSIS

```
SUBROUTINE KABORT(ISTAT)
INTEGER ISTAT
```

NAME

aopen - Open a file asynchronously

SYNOPSIS

```
#include <fcntl.h>

fd = aopen(name, flags, mode)
char *name;
int flags;
int mode;

fd = open(name, flags, mode)
char *name;
int flags;
int mode;

fcntl(fd, cmd, arg)
int fd;
int cmd;
int arg;
```

DOMAIN

These routines may only be called in programs compiled with the *Cubix* or *Plotix* libraries. Host programs may use `open` and `fcntl` but not the `O_CBXASYNC` flag described below.

DESCRIPTION

The function `aopen` is used to open a file completely asynchronously for use on one node only. All subsequent calls to I/O primitives referring to the returned file descriptor will be executed only on the specific node, with no regard for synchronization. The `flags` and `mode` arguments are described in the programmers manual for the host computer under `open`.

The functions `open` and `fcntl` are also documented in the programmers manual for the host computer. *Cubix* provides for an additional flag bit, `O_CBXASYNC`, (which may also be used with `open`), which signifies that subsequent I/O operations on the file descriptor should be asynchronous. In contrast to `aopen`, `open` must be called synchronously on all processors. All processors receive the same file descriptor, but if the `O_CBXASYNC` bit was set in `flags`, they will subsequently access the file asynchronously.

This functionality is supported in the buffered I/O system through the extra "A" character in the call to `fopen`. The command

```
fp = fopen("/tmp/foobar", "wA");
```

opens the indicated file "asychrnously" - each node making this call will receive its own pointer into the file. Furthermore not all nodes need make the call together.

EXAMPLE

The following code can be used to modify the "synchronous" access mode for an individual file. We first acquire the current flag settings and then add in the special O_CBXASYNC flag.

```
#include <fcntl.h>/* Needed to define O_ flags */
{
     flags = fcntl(1, F_GETFL);
                /* Read current file settings */
     flags |= O_CBXASYNC;
                /* Add in the asynchronous "bit" */
     fcntl(1, F_SETFL, flags);
                /* Set new flags */
}
```

SEE ALSO

fmulti, forder, mread, mwrite

NAME

`aspect` - Inquire device aspect ratio.

SYNOPSIS

```
aspect(devx, devy)
double *devx, *devy;
```

DOMAIN

This routine may only be called in programs compiled with the *Plotix* libraries.

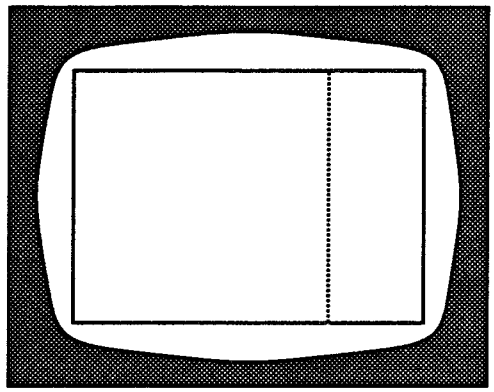
DESCRIPTION

This routine returns the "size" of the display surface. This concept is less than well-defined; it can either mean the number of pixels in each direction or the physical size of the display device. We choose to return the latter values since they seem to be more useful. In particular the sizes returned are the width and height of the display surface, in inches.

EXAMPLE

The following code shows the interaction between the `aspect` and `vport` calls which allows the user image to always appear in the largest square region of the display surface independent of the actual shape of the device screen. The `draw_square` routine draws a square in the user coordinate space. We show the output both on the default viewport and also after using `aspect` to make a square window.

```
double xfac, yfac, ratio;

draw_square();/* Draw square on default viewport */

aspect(&xfac,&yfac);
ratio = xfac/yfac;
if(ratio > 1.) vport(0.,0.,1./ratio,1.);
else vport(0.,0.,1.,1./ratio);

linemod(1);
draw_square();/* Now draw "real" square */
sendplot();
```



FORTRAN SYNOPSIS

```
SUBROUTINE KASPEC(XRES, YRES)
REAL*4 XRES, YRES
```

SEE ALSO

space, vport

NAME

box - Draw and fill rectangles.

SYNOPSIS

```
box(x0, y0, x1, y1, color, edge)
double x0, y0, x1, y1;
int color, edge;
```

DOMAIN

This routine may only be called in programs compiled with the *Cubix* or *Plotix* libraries.

DESCRIPTION

This routine provides a simple interface to the polygon drawing primitives for the common case of rectangular regions. A rectangle will be drawn whose bottom left corner has position `(x0, y0)` and whose top right corner is at `(x1, y1)`. The `color` argument indicates the manner in which the region should be filled. Positive values of `color` translate into solid colors in the same manner as the arguments to the line color primitive, `color`. Negative values yield device dependent shading patterns. If the `edge` argument is non-zero then the boundary of the region will be drawn in the color most recently specified in a call to the `color` function.

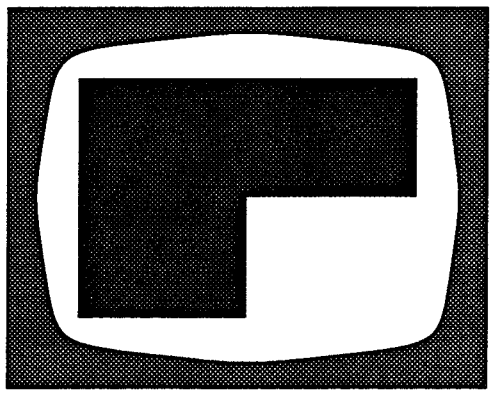
All coordinates are expressed relative to the most recent call to `space`.

Note that filling with `color = 0` and `edge = 0` results in a "selective erase" - specific areas of the screen can be erased.

EXAMPLE

The following code draws a simple box in the foreground color and then takes a "bite" out of it by drawing and filling in the background color.

```
space(0.,0.,10., 10.);/* Define coordinate system */

box(1.0, 1.0, 9.0, 9.0, 1, 1);
box(5.0, 1.0, 9.0, 5.0, 0, 0);

sendplot();
```



Note that this code achieves the same effect as that shown on the `panel` manual page but is much simpler. Also note that filling rectangular regions can often be done by hardware even in cases where no general hardware polygon fill is available. In these situations `box` will be significantly faster than the equivalent calls to either `polgn` or the `panel` routines.

FORTRAN SYNOPSIS

```
SUBROUTINE KBOX(X0,XY0,X1,Y1,ICOL,IEDGE)
REAL*4 X0, Y0, X1, Y1
INTEGER ICOL, IEDGE
```

SEE ALSO

`panel, color`

NAME

`callhost` - Interface to user host routines from *Cubix* program

SYNOPSIS

```
int callhost(func, odat,ocnt, idat,icnt, psent, pstat)
int func, ocnt, icnt, *psent, *pstat;
char *odat; *idat;

int starthost(func, odat, ocnt)
int func, ocnt;
char *odat;

int rethost(idat, icnt, psent, pstat)
int icnt, *psent, *pstat;
char *idat;
```

DOMAIN

These routines may only be called in programs compiled with the *Cubix* or *Plotix* libraries. Furthermore special software is necessary to re-link the part of the *Cubix* server which executes on the system host.

DESCRIPTION

These routines provide an interface between normal *Cubix* programs and user written routines which must run on the host computer. The simplest routine, `callhost`, causes a user written routine (denoted by the integer `func`, explained later) to begin execution. This routine can be passed up to 512 bytes of data from the buffer `odat` which it will receive as an argument. The exact number of bytes to be sent to the host routine is specified in the `ocnt` argument.

After processing the host routine is also allowed to send up to 512 bytes of information back to the node program which invoked it. This data will be stored at address `idat` in the node program. The `icnt` argument indicates the maximum number of bytes which should be copied to the node's `idat` buffer. If more are sent from the host they are ignored. In any case the variable pointed to by the `psent` argument will be initialized to the number of bytes which the host attempted to transfer, whether larger or smaller than `icnt`.

Finally the variable pointed to by the `pstat` argument will be set to the value returned by the host routine.

The 512 byte restriction is imposed by the fact that for reasons of speed the data transferred to/from the host routine is not copied to a "safe" user buffer but remains in the system buffer to which it was sent. If this poses too strict a restriction on the abilities of the node program then an alternative interface is provided by the `starthost` and `rethost` functions.

`starthost` is responsible for starting up the host routine and passing it up to 512 bytes of data in the same manner as indicated by the first three arguments to `callhost`. `rethost` performs the operation of the last four arguments to `callhost` which it

interprets in an identical fashion.The advantage of this interface is that between the calls to `starthost` and `rethost` in the node program the host and node codes are running in a mode identical to the "host-node" programming model and can, therefore, communicate data at will using the regular ***Express*** systems calls (`exread`, `exwrite`, `exbroadcast`, etc.)

Note, however, that this is a rather "double-edged" advantage. On the one hand it allows the host routine and the node program to communicate data at will avoiding the 512 byte restriction for data transactions in both directions. On the other hand the advantages of the ***Cubix*** programming model are suspended until the host routine terminates. As a result the node program *cannot* use any ***Cubix*** I/O or ***Plotix*** graphical commands until the call to `rethost` completes. Similarly the user will have to resolve potential byte swapping and alignment problems due to incompatible host and node CPU's which might otherwise have been covered up by the ***Cubix*** programming model. For these reasons, therefore, the interface through the `callhost` routine is to be preferred.

EXAMPLE

The following code segment is used to execute the host routine with index 3 in the host. A simple character string is sent to this routine and a small array of integers is returned.

```
    int tot, i, stat, hoststat, ngot;
    long indat[32];
    char *str = "Social security numbers";

    stat = callhost(3, str, strlen(str)+1,
                indat, sizeof(indat), &ngot, &hoststat);
/*
 * If either the host or nodes reported an error give up now.
 */
    if(stat < 0 || hoststat < 0) {
        fprintf(stderr,"Something went wrong: %d %d\n",
            stat, hoststat);
        return;
    }
/*
 * Figure out how many bytes we actually got. This is either
 * the number we asked for or the number sent, whichever is
 * smaller.
 */
    ngot = (ngot < sizeof(indat)) ? ngot : sizeof(indat);
/*
 * If everything seemed to be OK we can add up the numbers
 * returned by the host. Note that we might have to swap some
 * bytes here.....
 */
#ifdef SWAP
```

```
    _ex_swaw(indat, indat, ngot);
#endif
     tot = 0;
     for(i=0; i<ngot/sizeof(indat[0]); i++)
         tot += indat[i];
     }
```

It is important to note that the data buffer being transmitted to the host is sufficiently small to fit into the 512 byte restriction. This allows us to use the `callhost` interface. Further we take care to calculate correctly the amount of data returned to the node program and to (potentially) swap bytes. Since the data sent to the host is in the form of a character string we should not have to swap it's bytes. Were it any other data structure, however, the code shown within the `#ifdef SWAP` in the later part of this code might need to be added earlier too.

HOST INTERFACE

The previous sections described the interface to the system from the perspective of the node program. As well as incorporating one or more of the node system calls in your program you must also arrange for the host program to be linked with your host routines.

The linking of host routines into the `cubix` server process is controlled by the source file `userlink.c` supplied with the "*Cubix* user-link kit". This file contains an array of function pointers the indices of which correspond to the `func` argument passed to `callhost` or `starthost`. By default the top part of this file contains the following

```
static int user_no_op();

int (*user_funcs[])() = {
     user_no_op,
};
```

This code shows that a single host function is defined by default: `user_no_op`. This function doesn't actually do anything and is merely provided as a place holder to simplify the introduction of new user routines. If, for example, two additional user functions are required called, `search_DB` and `sort_DB` for example, we could modify the above part of the `userlink.c` file to read

```
extern int search_DB(), sort_DB();

int (*user_funcs[])() = {
     searchDB,
     sort_DB,
};
```

Notice that we elected to delete the `user_no_op` function and made the two new routines take indices 0 and 1. Also note that we changed the definition from `static` to `extern` since these routines are probably defined outside the `userlink.c` file.

Having initialized the data structures used by *Cubix* to find user host routines it remains only to discuss the calling sequence used when invoking them.

When a user host routine is called it is passed three arguments and is expected to return an integer value. The three arguments passed to the user routine are:

- A pointer to the buffer containing data sent to the host as the `odat` argument to `starthost` or `callhost`.
- The number of bytes contained in this buffer. This value will be the same as that specified as `ocnt` when calling the host routine from the nodes.
- A pointer to an integer which should be set to the number of bytes to be returned to the node program as `idat`. This data should be placed in the buffer pointed to by the first argument, overwriting whatever values were sent there from the nodes. The value written to this argument will be returned to the node program through the `psent` argument of `callhost` or `rethost`.

As an example, therefore, the skeleton of the `search_DB` function should be similar to

```
int search_DB(buffer, in_bytes, pout_bytes)
char *buffer;
int in_bytes;
int *pout_bytes;
{
     ........

     *pout_bytes = ...;
     return ...;
}
```

Notice that we finish the function by making sure that the `pout_bytes` argument is initialized. Finally we return a value which will be passed to the node program through the `pstat` argument.

DIAGNOSTICS

The node routines described here indicate error conditions by returning -1 and setting the external variable `errno` to a value indicating the source of the error. The possible error conditions are as follows:

`ETOOBIG` An attempt was made to either send too much data to the host or return too much to the nodes. The maximum amount of data that can be transmitted through the system invocation mechanism is 512 bytes.

`EBADPTR` The `func` argument indicated a function with an index outside those

defined in the host's function table.

It is important to note that if an error occurs in a call to `starthost` no call to `rethost` should be made.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KCALHO(FUNC, ODAT, OCNT,
                                IDAT, ICNT, PSENT, PSTAT)
INTEGER FUNC, ODAT(*), OCNT, IDAT(*), ICNT, PSENT, PSTAT

INTEGER FUNCTION KSTRHO(FUNC, ODAT, OCNT)
INTEGER FUNC, ODAT(*), OCNT

INTEGER FUNCTION KRETHO(IDAT, ICNT, PSENT, PSTAT)
INTEGER IDAT(*), ICNT, PSENT, PSTAT
```

SEE ALSO

syncmode.

NAME

`clipper` - Enable/Disable clipping.

SYNOPSIS

```
setclip(x0, y0, x1, y1)
double x0, y0, x1, y1;

endclip()
```

DOMAIN

This routine may only be called in programs compiled with the *Plotix* libraries.

DESCRIPTION

These two calls are used to enable and disable the clipping primitives. The `setclip` routine defines a two-dimensional clipping window relative to either the default user coordinate range or that set by the most recent call to `space`. Further lines, points, markers and polygons will be "clipped" relative to this window and portions lying outside the indicated range will be removed.

The `endclip` routine disables the clipper.

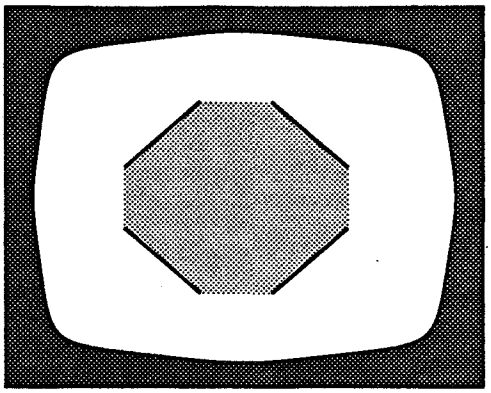
It is important to note that clipping is performed with respect to each "vport" and that the clipping window is specific to the active viewport at the time when `setclip` is invoked. Each call to `setvport` alters the clipping window to that associated with the particular "vport" selected.

Note that while clipping is typically expensive this process is supported on the nodes of a distributed machine rather than on the graphics device itself. As a result all clipping is performed in parallel leading to increased performance.

EXAMPLE

In the absence of the call to `setclip` the following code would draw a diamond shaped polygon on the display surface. After clipping only a portion of the figure remains.

```
space(0.,0.,4., 4.);/* Define coordinate system */
setclip(1., 1., 3., 3.);

initpanel(1, 1);
panelpoint(0., 2.);
panelpoint(2., 4.);
panelpoint(4., 2.);
panelpoint(2., 0.);
endpanel();

sendplot();
```



FORTRAN SYNOPSIS

```
SUBROUTINE KSETCL(X0, Y0, X1, Y1)
REAL*4 X0, Y0, X1, Y1

SUBROUTINE KENDCL
```

SEE ALSO

point, panel, move, marker, cont

NAME

`color` - Change color attribute of graphical objects.

SYNOPSIS

```
color(index)
int index;
```

DOMAIN

This routine may only be called in programs compiled with the *Plotix* libraries.

DESCRIPTION

This routine modifies the color used in drawing all subsequent lines and markers. The `index` argument is typically device-dependent but one can safely use the following on "color" devices.

```
0   Background color for device. ("Black")
1   Foreground color for device. ("White")
2   Red.
3   Green.
4   Blue.
5   Cyan.
6   Purple.
7   Yellow.
```

Monochrome devices, by default, support only two of these indices, 0 and 1.

The background color is often useful for selectively erasing previous symbols.

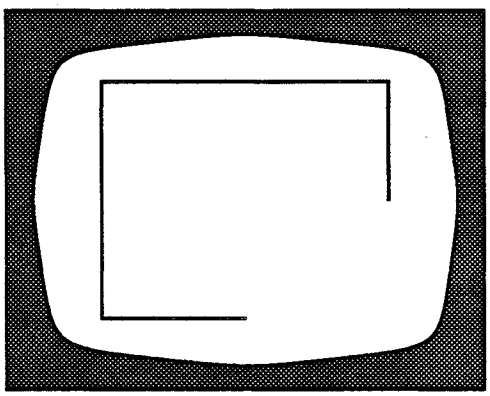
This function interacts with the `greyscale` and `rainbow` routines providing full color on devices capable of supporting such models.

EXAMPLE

The following code defines an 8 x 8 coordinate system and draws a simple box in the foreground color. It then overwrites the lower right hand corner of the box in the background color, erasing part of the image.

```
space(0.,0.,8., 8.);   /* Define coordinate system */

color(1);              /* Foreground color */
move(1.,1.);
cont(7.,1.);
cont(7.,7.);
cont(1.,7.);
cont(1.,1.);
```

```
color(0);                    /* Background color */
move(4.,1.);
cont(7.,1.);
cont(7.,4.);

sendplot();
```



FORTRAN SYNOPSIS

```
SUBROUTINE KCOLOR(ICOL)
INTEGER ICOL
```

SEE ALSO

cont, move, linemod, rainbow, greyscale

NAME

`console` - Redirect system calls.

SYNOPSIS

```
console_node(node)
int node;
```

DOMAIN

Available to node programs using the *Cubix* file server and linked with the *Cubix* or *Plotix* libraries.,

DESCRIPTION

This function is provided to support systems with more than one attached host. By default all *Cubix* system calls are directed to the processor which originally loaded and executed the user application. On occasion, however, it may be necessary to perform certain system tasks on other nodes in the system.

The `console_node` function has as its argument a processor number. All further (non-I/O) operating system requests will be directed to this node. To obtain suitable node numbers for use in this call we take the host identifier from the configuration utility, `cnftool`, and `OR` in the highest bit. If `cnftool` designated a particular host as "`H1`" then the appropriate node number to use is `0x8001`.

It is important to note that I/O related system requests are directed to the host which actually has the appropriate file, independent of the status of the `console_node` function.

EXAMPLE

Let us assume that three "host" processors are attached to our system. The first is the original system console which can be addressed through the value `HOST` from `express.h`. The others have the identifiers `H1` and `H2` as defined in the system configuration utility, `cnftool`. The following code executes a rather simple operation; it merely determines the "date" on each host in turn, returning overall control to the system console.

```
console_node(HOST);
system("date");          /* Main console */

console_node(0x8001);
system("date");          /* Host server1 */

console_node(0x8002);
system("date");          /* Host server2 */

console_node(HOST);      /* Switch back to main */
```

Note that different nodes are allowed to maintain distinct consoles with these calls although one must then use asynchronous requests to avoid deadlock.

FORTRAN SYNOPSIS

```
SUBROUTINE KCONND(NODE)
INTEGER NODE
```

SEE ALSO

`syncmode`, in "*Cubix*: Programming parallel computers without programming hosts." and "Using *Express* on systems with multiple hosts".

NAME

`cont` - Move and draw a line.

SYNOPSIS

```
cont(x, y)
double x, y;
```

DOMAIN

This routine may only be called in programs compiled with the *Plotix* libraries.

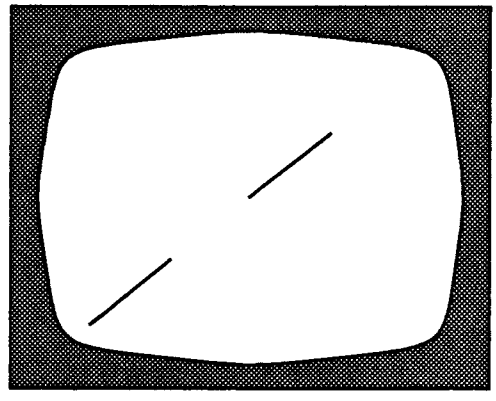
DESCRIPTION

Moves the current plotting position to (x,y) and draws a line in the current color from the previous plotting position. x and y are specified relative to the coordinate system defined by the most recent call to `space`.

EXAMPLE

The following code draws a broken diagonal line across the display surface.

```
space(0.,0.,4., 4.);/* Define coordinate system */

move(0.,0.);
cont(1.,1.);
move(2.,2.);
cont(3.,3.);

sendplot();
```



FORTRAN SYNOPSIS

```
SUBROUTINE KCONT(X0, Y0, X1, Y1)
REAL*4 X0, Y0, X1, Y1
```

SEE ALSO

move, color, linemod

NAME

`contour` - Contouring functions

SYNOPSIS

```
contour(func, gx, gy, levmin, levmax, nlev, panels)
double (*func)(), levmin, levmax;
int gx, gy, nlev, panels;

initlevel(func, gx, gy, level, panels)
double (*func)(), level;
int gx, gy, panels;

int getpoint(px, py)
double *px, *py;
```

DOMAIN

This routine may only be called in programs compiled with the *Plotix* libraries.

DESCRIPTION

The `contour` routine makes a two-dimensional contour map of the supplied function `func`. Rather than provide an array of values specifying the function to be contoured a function is supplied which will be called repeatedly with pairs of integer arguments representing the position at which a value is required. The range of values is specified by the `gx` and `gy` arguments; the user function will be called as `func(i,j)` with `i` and `j` in the ranges $0 \le$ `i` $<$ `gx` and $0 \le$ `j` $<$ `gy`.

Contours are drawn at `nlev` positions equally spaced between `levmin` and `levmax`. Optionally `levmin` and `levmax` can both be set to zero in which case suitable values are selected internally.

The final argument, `panels`, selects the type of contouring to be performed. If non-zero then the contours will be drawn as filled polygons while a zero value selects the more conventional style in which the contours are represented by lines. In the case of filled regions the lowest `nlev` indices of the color map will be used. The functions `rainbow` and `greyscale` can be used to re-map the appropriate color table entries.

Since this routine calls `usendplot` internally it must be called in all nodes together. Failure to do so will result in communication deadlock. Note, however, that no internode communication is done in performing the contouring. It is the responsibility of the user to distribute boundary values to processors that require them before calling the `contour` routine.

The contouring utility described here assumes that the data lie in a rectangular domain - i.e., that the mesh underlying the data is a Cartesian grid. In order to contour data specified in other coordinate systems, such as polar coordinates, the lower level `initlevel` and `getpoint` routines are available.

The former specifies a function to be contoured and a range of i and j values just as in

contour. The level argument selects the contouring level and the panels argument indicates the style of contouring to be performed. The interpretation of this value is not so straightforward as in the contour routine. Essentially the purpose is to control exactly what type of points are returned by the getpoint function. The allowed values and their interpretations are

panels = 0	All interior points are returned. The surrounding box is treated as a true rectangle and only the vertices are returned. This option is designed for simple line contouring of rectangular regions.
panels = 1	The contour map is cut into horizontal strips and coordinates are returned in such a way that the resulting polygonal regions are simply connected. The bounding box is treated as in option 0. Designed for color fill panels.
panels = 2	The interior points are treated as in option 1 but the boundary is also divided into many points which are returned individually. This option is designed for cases where the actual domain to be contoured is not rectangular and hence the boundary values need to be transformed in some manner.

The getpoint function is used, once a contour has been initialized, to return coordinates which lie on the contour. As well as returning an (x,y) coordinate pair under the supplied pointers the returned value indicates the nature of the returned point as follows

status = 0	This contour level is finished. Ignore returned coordinates.
status = 1	The coordinates are valid for the current contour.
status = 2	A segment of the current contour line is finished. Ignore the coordinates returned and call getpoint again in which case it will either return 0 indicating that no more points exist at this contour level or 1 indicating that another disjoint piece of the current contour exists.

A complete example of the use of these functions to contour a function supplied in polar coordinates is shown in the *Express* documentation.

EXAMPLE

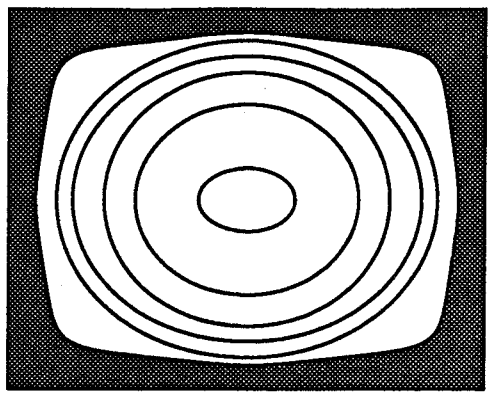
The following code demonstrates the elementary use of the contouring function.

```
double circles();
contour(circles, 10, 10, 0.0, 25.0, 6, 0);
```

The support routine which is invoked by the contouring package is defined as follows:

```
double circles(i,j)
int i,j;
```

```
{
      double retval, x0, y0;

      x0 = (double)(i-5);
      y0 = (double)(j-5);
      retval = x0*x0 + y0*y0;
      return retval;
}
```



FORTRAN SYNOPSIS

```
SUBROUTINE KCNTOR(FUNC,NX,NY,LEVMIN,LEVMAX,NLEV,IPAN)
REAL*4 FUNC, LEVMIN, LEVMAX
INTEGER NLEV, IPAN, NX, NY
EXTERNAL FUNC

SUBROUTINE KINITL(FUNC, NX, NY, LEVEL, IPAN)
REAL*4 FUNC, LEVEL
INTEGER NX, NY, IPAN
EXTERNAL FUNC

INTEGER FUNCTION KGETPT(X, Y)
REAL*4 X, Y
```

SEE ALSO

color, greyscale, rainbow.

NAME

cprof_on, cprof_off - Control communication profiler.

SYNOPSIS

```
cprof_on()

cprof_off()
```

DOMAIN

These routines may only be called from the nodes.

DESCRIPTION

cprof_on is used to enable and start the communication profiler. After this call all subsequent calls to the communication system result in entries being made in an internal log-file. cprof_off reverses this process - until a subsequent call to cprof_on no communication profiling will be performed.

For applications which have user programs running in the host computer the profiler is initially off and must be explicitly enabled with a call to cprof_on. For applications running in the *Cubix* environment the initial state of the profiler is controlled by a runtime switch in the cubix command. (See cprof_end).

The log of profiling information is written to the host file system with cprofcp or cprof_end.

EXAMPLE

The following code is a skeleton of that which might typically be used to control the communication profiler.

```
main()
{
/* Start off profiler */

     cprof_on();

/* Application Phase 1., profiler running */

          ...

/* Phase 1 complete, dump data with cprofcp/elt or
prof_end */

          ...

/* Application Phase 2., profiler turned off by previous
 * call to cprofelt or cprof_end
```

```
*/
        ...

/* Application phase 3., turn on profiler again */

    cprof_on();
        ...

/* Program over, dump data again and exit */

        ...
    exit(0);
}
```

Notice that we can selectively profile pieces of code. In this mode it makes sense to dump out the profile data independently to separate files for simplicity in later analysis.

FORTRAN SYNOPSIS

```
SUBROUTINE KCPON

SUBROUTINE KCPOFF
```

SEE ALSO

ctool (command), cprofcp, cprof_end

NAME

cprof_inq, cprof_end - Manipulate communication profiler under *Cubix*

SYNOPSIS

```
int cprof_inq()

cprof_end()
```

DOMAIN

These routines may only be called from the nodes.

DESCRIPTION

These routines provide a simple control interface to the communication profiler for applications running in the *Cubix* environment.

`cprof_inq` returns an integer value representing the state of the "-me" runtime switch on the `cubix` command line. This can be used to conveniently enable/disable the profiling system at runtime. Consider a typical command

```
cubix -n 4 toyland 1024 1024 <noddy.dat
```

Since no "-m" switch is present a call to `cprof_inq` will return zero. If we modify the above command to

```
cubix -mce -n 4 toyland 1024 1024 <noddy.dat
```

then the return value would be 1 since the character 'c' appears in the monitoring switch, "-m".

`cprof_end` is used to dump profiling data to the host computer file system. A file called "`cprof.out`" is created for later analysis with the `ctool` utility. In addition the profiler is disabled and its internal state reset to zero so that further profiling leads to distinct, non-overlapping data.

The operating system itself performs a check for the communication monitoring switch in the ***Cubix*** command and, if present, turns on the profiler with a call to `cprof_on`. It also arranges to call `cprof_end` at program termination with the `atexit` function. As a result a typical ***Cubix*** application need contain no explicit calls to the communication profiling routines - they are all made by the kernel. The only case in which such calls are needed is when more careful control is required over the profiler and the data it dumps.

EXAMPLE

The following code is a skeleton of that which might be used to control the communication profiler in a ***Cubix*** application.

```
main()
{
/* Start off profiler */
```

```
        if(cprof_inq())cprof_on();

    /* Application Phase 1., profiler running */

            ...

    /* Phase 1 complete, dump data with cprof_end,
    rename file */

        if(cprof_inq()) {
            cprof_end();
            rename("cprof.out", "phase1.cprof");
        }

    /* Application Phase 2., profiler off since cprof_end
    called */

            ...

    /* Application phase 3., turn on profiler again */

        if(cprof_inq()) cprof_on();
            ...

    /* Program over, dump data again and exit */

        if(cprof_inq()) {
            cprof_end();
            rename("cprof.out", "phase3.cprof");
        }
        exit(0);
    }
```

Notice that we can selectively profile pieces of code. In this mode it makes sense to dump out the profile data independently to separate files for simplicity in later analysis. The calls to `rename` in the above are necessary to prevent the second call to `cprof_inq` from overwriting the file created by the first call.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KCPINQ()

SUBROUTINE KCPEND
```

SEE ALSO

ctool (command), cprofcp, cprof

NAME

cprofcp, cprofelt - Dump communication profile data.

SYNOPSIS

```
cprofcp()

cprofelt(filename)
char *filename;
```

DOMAIN

cprofcp may only be called in the host processor and cprofelt may only be called in the nodes.

DESCRIPTION

These routines are used to dump the communication profiling data collected with the cprof functions. For each call to cprofelt on the nodes there must be a call to cprofcp in the host processor. The profiling data will be written to a file on the host with the name filename supplied in the node program.

Each call to cprofelt turns off the communication profiler and resets its internal counters so that further profiling starts from the zero state. This allows distinct communication profiles to be obtained for different regions of an application.

EXAMPLE

The following code is a skeleton of that which might typically be used to control the communication profile.

1. Host Program

```
main()
{
/* Allocate nodes, load programs */

        ...

/* Execute algorithm phase 1 and then dump data to
"phase1.cprof" */

      ...
     cprofcp();

/* Execute phase 2, profiler off */

        ...

/* Execute phase 3, profiler on, dump data to
```

```
"phase3.cprof" */

      ...
     cprofcp();
     exit(0);
}
```

2. Node Program

```
main()
{
/* Start off profiler */

     cprof_on();

/* Application Phase 1., profiler running */

          ...

/* Phase 1 complete, dump data with cprofcp/elt */

     cprofelt("phase1.cprof");

/* Application Phase 2., profiler off since cprofelt
called */

          ...

/* Application phase 3., turn on profiler again */

     cprof_on();
          ...

/* Program over, dump data again and exit */

     cprofelt("phase3.cprof");
     exit(0);
}
```

Notice that we can selectively profile pieces of code. In this mode it makes sense to dump out the profile data independently to separate files for simplicity in later analysis.

FORTRAN SYNOPSIS

```
SUBROUTINE KCPCP
```

```
SUBROUTINE KCPELT(FNAME)
CHARACTER*80 FNAME
```

SEE ALSO

ctool (command), cprof, cprof_end

NAME

`display` - Redirect graphical output.

SYNOPSIS

```
display_node(node)
int node;
```

DOMAIN

Available only to node programs linked with the *Plotix* library.

DESCRIPTION

This function is provided to support systems with more than one attached display device. By default all *Plotix* system calls are directed to the processor which originally loaded and executed the user application. On occasion, however, it may be necessary to perform certain tasks on other nodes in the system.

The `display_node` function has as its argument a processor number. All further *Plotix* requests will be directed to this node. To obtain suitable node numbers for use in this call we take the host identifier from the configuration utility and `OR` in the highest bit. If `cnftool` designated a particular host as "`H1`" then the appropriate node number to use is `0x8001`.

EXAMPLE

Let us assume that three "host" processors are attached to our system. The first is the original system console which can be addressed through the value `HOST` from `express.h`. The others have the identifiers `H1` and `H2` as defined in the system configuration utility, `cnftool`. The following code sketches the most common usage.

```
display_node(HOST);
sendplot();     /* Send graphical data */

    ....            /* More PLOTIX calls ... */


display_node(0x8001);
usendplot();    /* Send more data */

    ....            /* More PLOTIX calls ... */


display_node(0x8002);
usendplot();/* Send more data */

display_node(HOST);/* Switch back to main */
```

It is important to remember that *Plotix* has no "retained" capability which means that the `sendplot` routines really do completely flush the graphical data. To plot the same image on all three display surfaces requires that it be drawn each time.

Note that different nodes are allowed to maintain distinct displays with these calls although one must then use asynchronous plotting requests (e.g., `asendplot`) to avoid deadlock.

FORTRAN SYNOPSIS

```
SUBROUTINE KDISND(NODE)
INTEGER NODE
```

SEE ALSO

`syncmode`, "*Cubix*: Programming parallel computers without programming hosts." and "Using *Express* on systems with multiple hosts".

NAME

dotext - Draw text with complex alignment.

SYNOPSIS

```
dotext(text, x, y, angle, hjust, vjust)
char *text;
double x, y;
int angle, hjust, vjust;
```

DOMAIN

This routine may only be called in programs compiled with the *Plotix* libraries.

DESCRIPTION

This routine draws the characters contained in the `text` string at the position (x,y). The text is rotated through `angle` degrees. The two "justification" parameters are used to position the string with respect to the indicated coordinates as follows

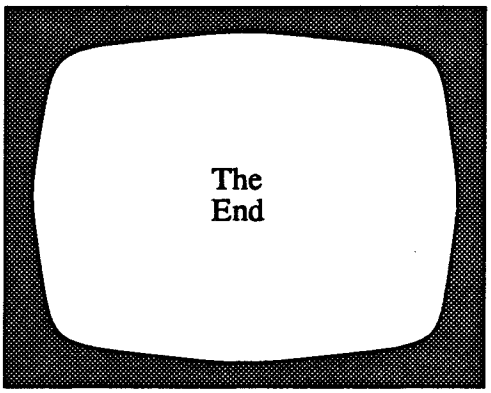
`hjust = -1`	Text is positioned entirely to the right of (x,y).
`hjust = 0`	The text is centered about (x,y).
`hjust = 1`	The text is placed entirely to the left of the indicated point.
`vjust = -1`	The text lies totally above (x,y).
`vjust = 0`	Text is centered vertically on (x,y).
`vjust = 1`	Text lies below (x,y).

Using various combinations of these parameters is it possible to align text in fairly arbitrary manners. Using the particular combination `hjust = vjust = 0` allows one to draw "markers" from the ASCII character set.

EXAMPLE

The following calls are used to position the phrase "`The End`" around a particular point.

```
dotext("The", x, y, 0, 0, -1);
dotext("End", x, y, 0, 0, 1);
```



WARNING

The current plotting position is undefined after this call. In order to perform reliable graphical operations `move` should be used before any further drawing is performed.

FORTRAN SYNOPSIS

```
SUBROUTINE KDOTEX(TEXT, X, Y, IANG, IHJUST, IVJUST)
CHARACTER*80 TEXT
REAL*4 X, Y
INTEGER IANG, IHJUST, IVJUST
```

SEE ALSO

`marker, label`

NAME

eprof_on, eprof_off, eprof_init, eprof_label, eprof_add - Event driven profiler.

SYNOPSIS

```
#include "express.h"

eprof_on()

eprof_off()

eprof_init(numlog, numlab)
int numlog, numlab;

eprof_label(index, title, format_str)
int index;
char *title, *format_str;

eprof_add(index, datum)
int index;
int datum;
```

DOMAIN

These routines may only be called from the nodes.

DESCRIPTION

These routines make up the interface to the user specified event driven profiling facility. eprof_on and eprof_off enable and disable the system respectively. While disabled no events are logged even if calls are made to eprof_add.

The routine eprof_init must be called before any of the other profiling calls. The arguments indicate the mount of space to reserve for "title" and "event" entries - each corresponds to a single call to the eprof_label and eprof_add functions. The special values DONTCARE may be given for either argument indicating that a system selected default should be used. The current overheads for log entries and labels are 12 and 68 bytes respectively.

eprof_add is the heart of the event system. It makes a new entry in the log file. Three items are logged; the event "index" and "datum" as given in the function call and the time at which the call is made. The index argument is used to differentiate between events at the highest level. This index corresponds to an optional title string in a call to eprof_label. The datum argument is used to identify events at the lowest level. This value is any 32-bit integer value which will be used in conjunction with the format_str argument in a call to eprof_label.

The function eprof_label is used to facilitate event recognition when the log-file is subsequently analyzed. Its use is optional. If no calls to eprof_label are made then

events will be identified by their "index" argument in the subsequent analysis and the datum value will be assumed to be an integer. Making a call such as

```
eprof_label(3, "After return from crunch_func",
            "Energy = %d");
```

builds in extra information. Together with the event "index" a legend will be presented which connects type 3 with the string "After return from crunch_func". Further, when the value of the datum argument is shown it will be formatted according to the format string - a typical result would be

```
Energy = 23
```

In applications which have a user written program running in the host computer the profiler is initially off and must be explicitly enabled with calls to eprof_init and eprof_on. In applications in the *Cubix* environment the initial state of the system is controlled by a runtime switch on the cubix command.

The log of profiling information is written to the host file system with eprofcp or eprof_end.

EXAMPLE

The following code is a skeleton of that which might typically be used to control the event profiler.

```
#include "express.h"

main()
{
     float Energy, resid, grind(), crunch();
     int iter, i;

/* Start profiler, make labels for indices 1-3, use
   default sizes */

     eprof_init(DONTCARE, DONTCARE);
     eprof_label(1, "Outer loop", "Iteration %d");
     eprof_label(2, "After crunch", "Energy = %d");
     eprof_label(3, "Inner loop", "resid = %d");
     eprof_on();

/* Start application code, then go into main loop */

          ...

     for(iter=0; iter<100; iter++) {
         eprof_add(1, iter);
         Energy = crunch(iter);
```

```
            eprof_add(2, (int)Energy);

            for(i=0; i<4; i++) {
                resid = grind(Energy);
                eprof_add(3, (int)resid);
            }
        }

    /* Program over, dump profile data and exit */

            ...
        exit(0);
    }
```

The insertion of events like these above can provide significant information about an application. The time between events 1 and 2, for example, indicates the duration of a call to the `crunch` function. Similar information is available about `grind` from events 2 and 3, averaged over the four calls per iteration. The auxiliary `datum` fields will show the interaction between the variables and the program execution rate. It may also show up bugs and/or unexpected behavior which could the key to understanding the failings of a particular parallelization scheme.

FORTRAN SYNOPSIS

```
SUBROUTINE KEPON

SUBROUTINE KEPOFF

SUBROUTINE KEPINI(LABBUF, LABSIZ, LOGBUF, LOGSIZ)
INTEGER LABBUF(*), LABSIZ, LOGBUF(*), LOGSIZ

SUBROUTINE KEPLAB(INDEX, TITLE, FORMAT)
INTEGER INDEX
CHARACTER*80 TITLE, FORMAT

SUBROUTINE KEPADD(INDEX, DATUM)
INTEGER INDEX
INTEGER*4 DATUM
```

FORTRAN DESCRIPTION

The Fortran equivalents of the above C functions operate in the obvious way with the exception of the `KEPLAB` and `KEPINI` subroutine. The former serves to initialize the event profiling system. The user provides two workspace buffers, `LABBUF` and `LOGBUF` for storing labels and log entries respectively. The size, in bytes, of each buffer is given by the following `SIZ` parameter. As a guide to appropriate sizes a label entry currently requires

68 bytes while a log entry needs 12.

The KEPLAB function assigns a label to a user "event". The format string which must be provided must be in the notation of the C function `printf`. While the details are complex one only needs to note that the string is printed as specified except that the special sequence "`%d`" is replaced by the value of the DATUM argument.

A suitable FORTRAN call which corresponds to that shown earlier in C is

```
CALL KEPLAB(3, 'After return from crunch_func',
          'Energy = %d')
```

SEE ALSO

`etool` (command), `eprofcp`, `eprof_end`

NAME

eprof_inq, eprof_end - Manipulate Event profile under *Cubix*

SYNOPSIS

```
int eprof_inq()

eprof_end()
```

DOMAIN

These routines may only be called from the nodes.

DESCRIPTION

These routines provide a simple control interface to the event profiler for applications running in the *Cubix* environment.

`eprof_inq` returns an integer value representing the state of the "-m" runtime switch on the `cubix` command line. This can be used to conveniently enable/disable the profiling system at runtime. Consider a typical command

```
cubix -n 4 toyland 1024 1024 <noddy.dat
```

Since no "-m" switch is present a call to `eprof_inq` will return zero. If we modify the above command to

```
cubix -mce -n 4 toyland 1024 1024 <noddy.dat
```

then the return value would be 1 since the character 'e' appears in the monitoring switch, "-m".

`eprof_end` is used to finally dump profiling data to the host computer file system. A file called "`eprof.out`" is created for later analysis with the `etool` utility. In addition the profiler is turned off and the internal state reset to its initial, zeroed, condition.

The operating system itself performs a check for the communication monitoring switch in the *Cubix* command and, if present, turns on the profiler with a call to `eprof_init` and `eprof_on`. The parameters passed to `eprof_init` are both `DONTCARE`. It also arranges to call `eprof_end` at program termination with the `atexit` function. As a result a typical *Cubix* application need only contain explicit calls to `eprof_add` and `eprof_label` - all control functions are performed by the kernel. The only case in which such calls are needed is when more careful control is required over the profiler and the data it dumps.

EXAMPLE

The following code is a skeleton of that which might typically be used to control the event profiler.

```
#include "express.h"
ETOGGLE mytog;
```

```
int eprof_is_on = 0;

main()
{
     eprof_init(DONTCARE, DONTCARE);
     eprof_label(1, "Outer Loop", "Iteration %d");
     eprof_toginit(mytog, "Timing inner loop");

     if(eprof_inq()) {
         eprof_is_on = 1;
     }
/*
 * Start first part of program using stuff initialized above.
 */
                .....

/* First phase is over. If the profiler was enabled, dump
 * the data to a file so that we can restart afresh ....
 */
     if(eprof_is_on) eprof_end();
/*
 * Start the second phases of the program with everything
 * reinitialized
 */
                ...
/*
 * When program finally finishes we can let the call to
 * "exit" take care of dumping any data that might be
 * left over.
 */
     exit(0);
}
```

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KEPINQ()

SUBROUTINE KEPEND
```

SEE ALSO

etool (command), eprofcp, eprof

NAME

eprofcp, eprofelt - Dump event log.

SYNOPSIS

```
eprofcp()

eprofelt(filename)
char *filename;
```

DOMAIN

eprofcp may only be called in the host processor while eprofelt may only be called in the nodes.

DESCRIPTION

These routines are used to dump the event profiling data collected with the eprof functions. For each call to eprofelt on the nodes there must be a call to eprofcp in the host processor. The profiling data will be written to a file on the host with the name filename supplied in the node program.

Each call to eprofelt turns off the profiler and resets its state so that future profiling commands begin with the system in its initial state.

EXAMPLE

The following code is a skeleton of that which might typically be used to control the event profiler.

1. Host Program

```
main()
{
/* Allocate node, load programs, etc. */

        ...

/* Dump out profile data to "eprof.out" */

    eprofcp();
    exit(0);
}
```

2. Node Program

```
#include "express.h"

main()
```

```
{
     float Energy, resid, grind(), crunch();
     int iter, i;

/* Start profiler, make labels for indices 1-3, use default
sizes */

     eprof_init(DONTCARE, DONTCARE);
     eprof_label(1, "Outer loop", "Iteration %d");
     eprof_label(2, "After crunch", "Energy = %f");
     eprof_label(3, "Inner loop", "resid = %f");
     eprof_on();

/* Compute, compute, compute ..... */

/* Program over, dump profile data and exit */

     eprofelt("eprof.out");
     exit(0);
}
```

Note that these functions may be called repeatedly - the only constraint is that each call to `eprofelt` in the nodes must have a corresponding call to `eprofcp` in the host.

FORTRAN SYNOPSIS

```
SUBROUTINE KEPCP

SUBROUTINE KEPELT(FNAME)
CHARACTER*80 FNAME
```

SEE ALSO

etool (command), eprofcp, eprof_end

NAME

`erase, aerase` - Clear the display surface.

SYNOPSIS

```
erase()

aerase()
```

DOMAIN

This routine may only be called in programs compiled with the *Plotix* libraries.

DESCRIPTION

These routines are used to clear the display surface. `erase` must be called at the same time in all nodes while `aerase` may be called asynchronously in any node at any time. Note that this latter option can cause rather unpredictable results unless used with some care since 1000 nodes all calling `aerase` makes the screen "blink" rather a lot!

It is important to note that neither of these routines flushes the graphics buffer to the output device. Rather they just reset the internal data structures to reflect empty buffers. All data that is required to appear, however briefly, on the display surface must be flushed explicitly with one of the `sendplot` routines before calling `erase`.

Hardcopy devices handle these functions in device specific ways. Some, for example, can only print on a single sheet at once and so the `erase` commands are handled by switching to a new output file. Eventually several files may be printed one at a time. Others, such as laser printers, merely switch to new pages.

EXAMPLE

The following is typical of the general use of the erase functions.

```
#include "express.h"
#include <stdio.h

{
      /* Start up graphics system */

     if(openpl(DONTCARE, (FILE *)0) < 0) {
          fprintf(stderr, "Failed to start up graphics\n");
          exit(1);
     }

/* Grind away .... graphics, graphics, graphics.... */

                ...

/* Finished with first image, erase and go again */
```

```
erase();
```

FORTRAN SYNOPSIS

```
SUBROUTINE KERASE
SUBROUTINE KAERAS
```

SEE ALSO

sendplot

NAME

eprof_toginit, eprof_toggle - Statistical analysis of code sections.

SYNOPSIS

```
#include "express.h"

eprof_toginit(togptr, label)
ETOGGLE *togptr;
char *label;

eprof_toggle(togptr)
ETOGGLE *togptr;
```

DOMAIN

These routines may only be called from the nodes.

DESCRIPTION

These routines allow selective analysis of particular sections of code. By surrounding code segments with calls to the eprof_toggle one can obtain statistics relating to the number of times the particular code section was called and the average and total times spent in these sections. The data is collected in exactly the same manner as the "event profiling" information obtained through calls to eprof_add. The same commands are available to dump the profiling data and/or rename the file containing it as are used by the other "eprof" utilities.

Each "toggle" data structure must be initialized with a call to eprof_toginit before it can be used for data collection. This function expects to be passed a pointer to a structure of type ETOGGLE, defined in "express.h", and a string that will later be used to identify the collected statistics when analyzed with etool.

The log of profiling information is written to the host file system with eprofcp or eprof_end.

EXAMPLE

The following example demonstrates the use of the "toggle" ideas.

```
#include "express.h"

ETOGGLE looptog, grindtog;

main()
{
     float Energy, grind_away();
     int iter, i;

/* Initialize toggle data structures. */
```

```
    eprof_toginit(&looptog, "Main iteration loop");
    eprof_toginit(&grindtog, "Calls to grind_away");

/* Start application code, then go into main loop */

        ...

    for(iter=0; iter<100; iter++) {
        eprof_toggle(&looptog);

            /* Other processing going on here.... */

                    ........


        for(i=0; i<4; i++) {
            eprof_toggle(&grindtog);
            grind_away(Energy, i);
            eprof_toggle(&grindtog);
        }
        eprof_toggle(&looptog);
    }
/*
 * Dump data to host for later analysis.....
 */
            .........

    exit(0);
}
```

The "toggle" data will be stored in a file with the name "`eprof.out`" (unless overridden by some other function call) together with the normal "event" data which may have also been collected with calls to `eprof_add`.

To analyze this data we execute the "`etool`" command with

```
etool -p -t
```

This combination of switches both suppresses the normal graphical output and also restricts

attention to the "toggle" data. The output for the above example might appear as follows

Node 0

Description	Total	#Calls	Avge.	Var.
Main iteration loop	478.32	100	4.78	.28
Calls to grind_away	363.96	400	0.91	.03

Node 1

Description	Total	#Calls	Avge.	Var.
Main iteration loop	478.32	100	4.78	.28
Calls to grind_away	363.96	400	0.91	.03

etc...

For each node is displayed the list of initialized toggles together with the number of times each code section was used, the total time elapsed in this section, the average time per call and the variance of these times. Using this information it is possible to build up a very accurate picture of the performance of a parallel program.

FORTRAN SYNOPSIS

```
SUBROUTINE KEPTGI(TOGGLE, TEXT)
INTEGER TOGGLE(16)
CHARACTER*80 TEXT

SUBROUTINE KEPTOG(TOGGLE)
INTEGER TOGGLE(16)
```

FORTRAN DESCRIPTION

The Fortran equivalents of the above C functions operate in the obvious way with the exception that no header file is available in FORTRAN to define the "toggle" data structure. Instead FORTRAN programs should use arrays of 16 integer values as the appropriate variables.

SEE ALSO

etool (command), eprofcp, eprof_end

NAME

exaccess - Share a processor group with another process

SYNOPSIS

```
#include "express.h"

int exaccess(device, pnodes)
char *device;
int *pnodes;
```

DOMAIN

Host processor only.

DESCRIPTION

This routine provides a "brute-force" mechanism by which a host program can obtain access to every node in the network irrespective of whether or not that node is currently executing a program - even if allocated to another user. This often useful for providing overall system monitoring or when only a single application is to run on the entire network.

The first argument specifies the particular parallel computer to which access is desired and is interpreted in the same manner as the corresponding argument to exopen. The last argument is returned to the caller containing the number of nodes in the system.

RETURN VALUE

The value returned by exaccess is the ***processor group index*** which must be used in future references to the shared processors.

If some error occurs or nodes are accessible to the host processor -1 is returned.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXACCS(DEVICE, NODES)
CHARACTER*80 DEVICE
INTEGER NODES
```

WARNINGS

Communicating with shared processor groups is complicated by interactions between source and type fields specified using the DONTCARE wildcard. This situation can be eased somewhat through the extype mechanisms which restrict the ranges indicated by wildcard values. It should further be noted that subsequent to this call the host must communicate with the processors using the node numbers indicated by cnftool rather than according to the logical mapping which results from exopen or exshare.

SEE ALSO

exopen, exshare, extype, "Cnftool: Configuring *Express*".

NAME

exbreak - Halt program at breakpoint

SYNOPSIS

```
exbreak()
```

DOMAIN

This routine may only be called in node programs.

DESCRIPTION

The exbreak function causes the program to halt as though it had encountered a breakpoint of the ytpe normally associated with the debugger, ndb. Examination of the process state with ndb will show the process to be in state Breakpoint.

FORTRAN SYNOPSIS

```
SUBROUTINE KXBREA
```

SEE ALSO

expause.

NAME

exbroadcast - Interprocessor broadcast.

SYNOPSIS

```
int exbroadcast(buffer, origin, nbytes, Nnodes,
                                        nodelist, ptype)
char *buffer;
int origin, nbytes, Nnodes, *nodelist, *ptype;
```

DOMAIN

exbroadcast may be called in both host and node processors.

DESCRIPTION

exbroadcast is used to perform broadcasting operations among the processors.

The broadcast starts from processor origin which attempts to broadcast the nbytes of data in the indicated buffer. The processors to which the broadcast will be sent are indicated by the Nnodes and nodelist arguments in the following way: nodelist is an array of processor numbers which should receive the broadcast message. Nnodes is the number of elements in the array. Further the special value Nnodes = ALLNODES indicates that the broadcast should go to all processors. In this case the value of nodelist is ignored - even the NULL pointer may be specified. The receiving nodes deposit the incoming data at buffer, up to a maximum of nbytes.

The broadcast operation carries a "type" field in common with all other communication primitives so that overlapping broadcasts may be distinguished. This parameter is supplied under the pointer argument ptype and may be any positive quantity - neither NULLPTR nor DONTCARE arguments may be used.

A call to exbroadcast in the originating node must have corresponding calls to exbroadcast in ***all target nodes***. A corresponding call in other nodes which are not target nodes is not necessary, but will be handled without error. All calls must specify the same values of the origin, Nnodes and nodelist arguments or communication deadlock will occur. A receiving node must specify nbytes greater than or equal to that specified in the originating node. When nodelist is not NULL, the contents arrays must be exactly identical in each processor. The origin may or may not appear in the nodelist, at the convenience of the calling routine. When no errors occur, the value returned is the number of bytes written by the originating node, or the number read by a receiving node.

EXAMPLE

In the following code we use the exgrid tools to find the processor number of the processor at the origin of a three dimensional processor decomposition. This processor then broadcasts a set of data values to all other nodes.

```
#include "express.h"
```

```
extern float datbuf[1024];

main()
{
     int nprocs[3], coord[3], corner, type=33;

/* Initiate a three-dimensional decomposition of eight
processors */

     nprocs[0] = nprocs[1] = nprocs[2] = 2;
     if(exgridinit(3, nprocs) < 0) {
         abort(1);
     }

/* Now find the processor in the (0,0,0) spot in the user
topology */

     coord[0] = coord[1] = coord[2] = 0;
     corner = exgridproc(coord);

     exbroadcast(datbuf, corner, 32*sizeof(float),
             ALLNODES, NULLPTR, &type);
```

Note that, since the broadcast operates totally within the node processors it is valid to use the `sizeof` operator. If the host included the host processor one would have to be more careful since variable types, especially `int` are often of different sizes on the two processors.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXBROD(BUFFER, ORIGIN, NBYTES,
           NNODES, NODEL, TYPE)
INTEGER BUFFER(*), ORIGIN, NBYTES, NNODES, NODEL(*),TYPE
```

DIAGNOSTICS

If any error occurs in `exbroadcast` -1 is returned. Possible sources of error are: an illegal buffer, a preposterous value of `nbytes` or invalid values of `origin`, `Nnodes` or `nodelist`. If no error occurs the number of bytes broadcast is returned in the originating processor and the number read in the receiving nodes. An error condition is also indicated in any node which reads less bytes than were originally transmitted by the originating processor. In this case `errno` is set to `ENOTREAD`, `errcnt` records the number of bytes actually sent and the unread bytes are discarded.

SEE ALSO

`excombine`, `exconcat`

NAME

exchan - Hardware dependent communication primitives.

SYNOPSIS

```
exchanon(chan)
int chan;

exchanoff(chan)
int chan;

int exchanrd(chan, buffer, nbytes)
int chan, nbytes;
char *buffer;

int exchanwt(chan, buffer, nbytes)
int chan, nbytes;
char *buffer;
```

DOMAIN

These routines are available only to node programs. Their availability is further subject to hardware restrictions on the system in use.

DESCRIPTION

These routines implement a message passing strategy which directly accesses the hardware present on the parallel processing system in use. As such their use is highly non-portable. Since, however, these routines have a very trivial syntax they can provide communication at the full speed of the underlying hardware. In most cases this affects the asymptotic communication rate only slightly but may reduce the start-up time (latency) by as much as an order of magnitude. They are most applicable, therefore, when the application needs to send many short messages.

Before attempting to use the message passing routines `exchanoff` must have been called for every channel on which the low level functions will be used. this function serves to disable the normal *Express* processing for that channel. Note that the user is responsible for ensuring that no internode communication traffic will be disrupted by the sudden removal of one of the message paths normally used by *Express*. In practice this usually means that the application should force a synchronization through some operation before disabling any of the communication channels. Note that while a channel is disabled none of the higher level *Express* functions may be used. In particular this means that the debugger, `ndb`, will be unable to operate.

`exchanon` performs the opposite function, causing *Express* to once again become active on the indicated channel. Again it is the user's responsibility to ensure that no *Express* messages are transmitted along channels that are still disabled.

The channel read function, `exchanrd`, reads `nbytes` bytes of data into the supplied buffer from the channel indicated by the `chan` argument. It will not return until exactly `nbytes` have been read. The node from which data is read depends on the interpretation

of the chan argument, which is hardware dependent.

Similarly the exchanwt function sends nbytes bytes of data into the channel indicated by the chan argument. The data to be transmitted is taken from the user supplied buffer. This function will not return until all data has been read by a corresponding call to exchanrd.

EXAMPLES

The following schematic code shows a typical sequence involving the exchan primitives. We assume that some routine, nearest_neighbor requires heavy internode message traffic between processors directly connected to each other in hardware. As such they are able to make use of the exchan functions.

```
/*
 * Assume that we can work here with the full Express system.
 */
                ..................
/*
 * For the next function we will disable Express.
 */
    exsync();       /* Force synchronization */
    for(i=0; i<nchans; i++) exchanoff(i);

    nearest_neighbor();
/*
 * Assume that this routine terminates fully synchronized
 * so that we can enable Express.
 */
    for(i=0; i<nchans; i++) exchanon(i);
/*
 * Proceed with Express functioning......
 */
```

Notice that we have used a variable nchans to indicate how many channels should be modified. The value of this variable is also somewhat machine dependent - on a transputer system it might be four for all the hardware links, for example, while on a hypercube it will usually be the base 2 logarithm on the number of nodes.

FORTRAN SYNOPSIS

```
SUBROUTINE KXCHON(CHAN)
INTEGER CHAN

SUBROUTINE KXCHOF(CHAN)
INTEGER CHAN
```

```
INTEGER FUNCTION KXCHRD(CHAN, BUFFER, NBYTES)
INTEGER CHAN, BUFFER(*), NBYTES

INTEGER FUNCTION KXCHWT(CHAN, BUFFER, NBYTES)
INTEGER CHAN, BUFFER(*), NBYTES
```

WARNING

These routines perform extremely hardware dependent operations and as such should be used with caution. The "nearest neighbor" communication model that they represent has however, been shown by a number of researchers to be adequate (if not optimal) for a wide class of algorithms. (An excellent reference is the book "Solving Problems on Concurrent Processors" by G.C.Fox *et al.*, published by Prentice-Hall, 1988.)

If these routines seem appropriate for your algorithm we suggest that the full *Express* routines be used during development, since this enables use of the other system tools such as the debugger, and then these routines be substituted in the final product.

SEE ALSO

`exread`, `exwrite`, `exsync`.

NAME

exchange, exvchange - Synchronous scalar/vector exchange primitive.

SYNOPSIS

```
int exchange(ibuf, ilen, isrc, itype, obuf, olen,
                                              odest, otype)
char *ibuf, *obuf;
int ilen, *isrc, *itype, olen, *odest, *otype;

int exvchange(ibuf, isize, ioff, iitems, isrc, itype,
                obuf, osize, ooff, oitems, odest, otype)
char *ibuf, *obuf;
int isize, ioff, iitems, *isrc, *itype,
osize, ooff, oitems, *odest, *otype;
```

DOMAIN

These functions may be called in either host or node processors.

DESCRIPTION

These functions are used to implement "synchronous" communication between two processors; a call to exchange in one processor will not return until the corresponding call has been made in the sending and receiving processors.

This function essentially performs a similar task to successive calls to exwrite and exread - i.e., data is first sent and then read from (possibly) different nodes. One advantage of this function are that its extra constraint (synchronicity) allows optimizations to be made for both speed and reliability. The former can be achieved because data transmission in the two directions can be overlapped while the latter is enhanced because low level "handshaking" can be performed to ensure that no intermediate buffers overflow. A second advantage is that the exchange of information can be considered to be simultaneous - the user is freed from any worry about which node should read first and which write. As a result these functions should be preferred to the analogous pair of exread/exwrite operations whenever the synchronous constraint can be met.

A further advantage is that the user is freed from any concern about "deadlock" conditions which can arise on certain types of hardware where it is important that reads and writes come in the correct order.

exchange causes olen bytes of data to be sent to the node denoted by odest in a message of type otype (which may *not* have the DONTCARE value). The data is taken from memory at obuf. It is not guaranteed that olen will be read by the reading processor; the actual number of bytes read depends on the number specified in the corresponding call to exchange. If no error occurs, the actual number of bytes written is returned to the calling program. The interpretation of the destination and type fields is exactly as in exwrite.

exchange also causes at most ilen bytes of data to be read from the source denoted by

isrc from a message of a type matching itype. The data is placed in memory at ibuf. It is not guaranteed that ilen bytes will be read; the actual number of bytes read depends on the number written by the transmitting processor. If no error occurs, the actual number of bytes read is returned to the calling program. The interpretation of the isrc and itype arguments is exactly as in exread.

A call to exchange must be complemented by calls to exchange in the processors denoted by odest and isrc in order to prevent communication deadlock. Similarly the message types in these processors must be compatible.

Note that the exchange of data is conceptually simultaneous - data is written to the output processor at the same time as it is received from the sender. This allows, for example, the buffer arguments to be identical. The kernel maintains the integrity of the data and handles any read/write synchronization problems.

The above discussion holds equally well for the exvchange function. The difference between the two is analogous to the difference between exread and exvread). While the former is used to transmit contiguous blocks of memory the latter is able to send messages made up of several disjoint memory areas.

The arguments to exvchange are interpreted in the same way as their counterparts in exvread. The message is specified by defining a number of "objects" to be sent. Each is of length size bytes and is separated from the next by offset bytes. In total items objects will be transmitted. This description applies to both the input and output arguments of exvchange.

EXAMPLE

Consider a simple model of a two-dimensional terminal screen. We assume that the data currently displayed is represented by an 80 x 24 array of characters. Using the exgrid and exchange primitives it is easy to construct routines which, for example, scroll the data in different directions when decomposed in parallel.

```
#include "express.h"    /* Defines nodenv structure */

#define HORIZ 0
#define VERT 1

/* The amount of the display in each node is found by
 * decomposing the 80 x 24 total over the processors
 */

char screen[20][12];/* The displayed data */

main()
{
     int nprocs[2];

     nprocs[HORIZ] = 4;
```

```
nprocs[VERT] = 2;

if(exgridinit(2, nprocs) < 0) {
    abort(-1);
}
```

The macros HORIZ and VERT are defined for our convenience and just serve to label the two axes on the screen. We assign four processors to the horizontal dimension and two to the vertical. (A more flexible assignment scheme is easily devised using the exparam and exgridsplit system calls to determine at runtime the number of processors available.)

Now consider a simple scrolling operation in which data is to be passed to the right. We need to figure out the processor numbers necessary to communicate in this direction using exgridnode.

```
struct nodenv nodedata;
int recpnum[2], perbc[2], type=12;

exparam(&nodedata);/* Get runtime parameters */

perbc[0] = perbc[1] = 0;/* Make non-periodic system */
exgridbc(perbc);

upnode = exgridnode(nodedata.procnum, VERT, 1);
downnode = exgridnode(nodedata.procnum, VERT, -1);
leftnode = exgridnode(nodedata.procnum, HORIZ, -1);
rightnode = exgridnode(nodedata.procnum, HORIZ, 1);
```

Note that we have made the additional step of dealing with the boundaries of the screen correctly. If a processor is on the extreme left edge of the display and it tries to communicate with a processor to its left then the value of leftnode will be correctly assigned the value NONODE which will, in turn, direct exchange to omit communication with this non-existent processor.

Now in order to "scroll" the data over to the left we merely use the following call to exchange.

```
exchange(screen, 12, &leftnode, &type,
        &screen[19][0], 12, &rightnode, &type);
```

Notice that at no point in these calculations did the topology of the hardware enter. Everything is specified in the user domain - i.e., screen coordinates, and exgrid and exchange do the rest. Notice the appearance of the "magic" number 12 in the above call. To arrive at this value we divided the height of the screen (24) by the number of processors

in that direction (2). We could do much better by using the exgridsize function which would also allow the possibility of changing the number of processors allocated to each dimension at runtime.

In order to scroll data vertically instead of horizontally we would just use the call

```
exvchange(screen, 1, 12, 20, &downnode, &type,
        &screen[0][11], 1, 12, 20, &upnode, &type);
```

The arguments here are arrived at in a similarly simple manner. If each processors piece of the display surface is 20 x 12 then we need to take every twelfth byte when we scroll upward. Also there are twenty bytes to transmit. (Again this can be made more flexible using the exgridsize function.)

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXCHAN(IBUF, ICNT, ISRC, ITYPE,
                        OBUF, OCNT, ODEST, OTYPE)
INTEGER IBUF(*), ICNT, ISRC, ITYPE
INTEGER OBUF(*), OCNT, ODEST, OTYPE

INTEGER FUNCTION KXVCHA(IBUF,ISZ,IOFF,IITMS,ISRC,ITYPE,
                    OBUF,OSZ,OOFF,OITMS,ODEST,OTYPE)
INTEGER IBUF(*), ISZ, IOFF, IITMS, ISRC, ITYPE
INTEGER OBUF(*), OSZ, OOFF, OITMS, ODEST, OTYPE
```

DIAGNOSTICS

If any error occurs in exchange or exvchange -1 is returned. Possible sources of error are: an illegal source or destination, an illegal buffer or a preposterous value of length, size, offset or item arguments. If no error occurs exchange returns the number of bytes read and exvchange the number of items read.

SEE ALSO

exread, exwrite, exvread, exvwrite, exgrid, exparam.

NAME

exclose - Deallocate processors.

SYNOPSIS

```
#include "express.h"

int exclose(pgind)
int pgind;
```

DOMAIN

Available to host programs only.

DESCRIPTION

This routine is used to terminate a connection between the host and a processor group.

This routine should be called at the end of an application's use of a processor group to ensure that system resources are correctly reset. The sole argument, pgind, is the ***Processor group index*** originally returned by the exopen call.

EXAMPLES

The following schematic code should be the general template of any host process which allocates and uses processor groups.

```
#include <stdio.h>
#include "express.h"

main(argc, argv)
int argc;
char *argv[];
{
     int pg1;

/* Allocate a processor group */

     pg1 = exopen("/dev/transputer", 4, DONTCARE)) < 0);

     if(pg1 < 0) {
         fprintf(stderr,
            "Failed to allocate processor group\n");
         exit(-1);
     }

/* Load progs, send/receive messages to processors */

          ...
```

```
/* Application finished. Clean up by deallocating
processors */

     exclose(pg1);

     exit(0);
}
```

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXCLOS(PGIND)
INTEGER PGIND
```

SEE ALSO

exopen, exshare, exinit (command).

NAME

excombine - Apply global operation to distributed data.

SYNOPSIS

```
int excombine(buffer, func, size, nitems, Nnodes,
                                        nodelist, ptype)
char *buffer;
int (*func)(), size, nitems, Nnodes, *nodelist, *ptype;
```

DOMAIN

excombine may be called in the node processors only.

DESCRIPTION

This function provides one of the most powerful facilities in *Express* - the ability to apply a global operation to a dataset distributed across the node of the parallel computer.

This routine is used to performing "combining" operations on data within the node processors. An example of such an operation is the sum of a set of values distributed over the nodes of the parallel machines. Other example combining functions are products, maximum and minimum functions.

nitems of data, each of size bytes and taken from buffer are individually combined across the specified node processors. The final vector of results will overwrite buffer in each node.

In the nodes the user-supplied combining function will be called for each of the nitems to be combined. In each case three arguments will be supplied to the routine. The first two are pointers to objects to be combined and the third is the size argument supplied in the call to excombine. The combining function's responsibility is to perform whatever operation is required and write the result at the address of the first operand. The value returned by the combining function is used to detect errors in the excombine routine. If a negative value is returned the current call to excombine is aborted and an error returned to its caller.

The nodes involved in the combining operation are specified by the Nnodes and nodelist arguments. The latter is an array of processor numbers listing those nodes on which the combining operation is to take place. Nnodes is the number of elements in this list. The special value Nnodes = ALLNODES is allowed and performs the combining operation on all processors. In this case the value of nodelist is ignored and may even be the NULL pointer.

The ptype argument is used to specify a "type" for the combine operation. This is used to distinguish between potentially overlapping communication operations. Any positive value is legal. (You may not use the DONTCARE or NULLPTR mechanisms for this argument.)

All processors involved in the combining operation must call excombine together with identical values for both Nnodes and nodelist - otherwise communication deadlock will occur.

EXAMPLE

The first example merely calculates the global sum of the components of a vector distributed over all processors. We assume that the values are supplied in an array, `values`, passed to this routine.

```
#include "express.h"

double get_sum(values, Npts)
doube values[];
int Npts;
{
     int i, type = 37;
     extern int sum_up();
     double result;

/* First compute the subtotal in each node. */

     result = 0.0;
     for(i=0; i<Npts; i++) result += values[i];

/* Now combine these values with the sum_up function */

     excombine(&result, sum_up, 8, 1,
            ALLNODES, NULLPTR, &type);
     return result;
}

int sum_up(p1, p2, size)
double *p1, *p2;
int size;
{
     *p1 += *p2;
     return 0;
}
```

Notice how the combining function replaces its first argument with the result of the combining operation and returns zero to indicate successful combination.

In the second example processors 0 thru 3 have obtained a vector of `Npts` floating point values, `myvec`. The purpose of the call is to calculate, for each array slot, the maximum value distributed over the nodes.

Note that we are assuming that more than four processors are involved in this calculation - if all nodes were taking part in the operation we would use the `ALLNODES` macro in the call to `excombine` instead of the `nodelist` argument.

```
find_max(myvec, Npts)
float myvec[];
int Npts;
{
     int i, Nnodes, nodelist[4], type=48;
     extern int maxflt();

/* Setup nodelist array to specify combining nodes */

     Nnodes = 4;
     for(i=0; i<Nnodes; i++) nodelist[i] = i;

/* Combine values with the maxflt function */

     excombine(myvec, maxflt, sizeof(float), Npts,
                    Nnodes, nodelist, &type);
}

int maxflt(p1, p2, size)
float *p1, *p2;
int size;
{
     if(*p2 > *p1) *p1 = *p2;
     return 0;
}
```

This example points out another important point. The purpose of the `nitems` field is to allow multiple data items to be combined ***and left separate*** - it is not to perform "on-node" combination operations before the global operation. This is the reason why we explicitly coded the subtotal operation in the first example. In this last example `Npts` values will be left in each node corresponding to the maximum value among all processors of `myvec[0]`, the maximum of `myvec[1]` and so on.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXCOMB(BUFFER, FUNC, SIZE, NITEMS,
                        NNODES, NODEL, TYPE)
INTEGER BUFFER(*), FUNC, SIZE, NITEMS
INTEGER NNODES, NODEL(*), TYPE
EXTERNAL FUNC
```

DIAGNOSTICS

If any error occurs in `excombine` -1 is returned. Possible sources of error are: an illegal `buffer`, preposterous values of `nitems` or `size` and the return of a negative value from the combining function. If no error occurs the number of items combined is returned.

WARNING

The combining function must be both commutative and associative in order to give results which are independent of the underlying hardware topology. If we denote the operation of the combining function on two elements A and B by A `op` B then the requirements can be written as

Commutativity:	A `op` B = B `op` A
Associativity:	(A `op` B) `op` C = A `op` (B `op` C)

Useful functions which satisfy these constraints are: addition, multiplication, maximum, minimum, logical AND, logical OR, logical XOR. Operations which do not satisfy the constraint are: subtraction (`3-1!=1-3`) and division (`4/2!=2/4`)

One particularly nasty problem which can be encountered is the use of this function with floating point values. Because of the manner in which truncation and rounding are applied to such values when performing arithmetic operations it cannot be guaranteed that every node will receive *exactly* the same answer when combining floating point numbers. While the difference is extremely small it can be sufficient to cause *Cubix* to abort with a violation of the "loosely synchronous" constraint if the resulting numbers are output in "singl" mode. No solution to this problem is known at this time.

BUGS

There is an implementation specific upper limit on the `size` of each individual item that can be combined. In most applications this should be an unimportant restriction. If the limit is exceeded `excombine` will return -1 and set `errno` to `ETOOBIG`.

SEE ALSO

`exbroadcast, exconcat.`

NAME

exconcat - Collect distributed data into each node.

SYNOPSIS

```
int exconcat(mybuf, mybytes, resbuf, ressize, sizes,
                                    Nnodes, nodelist, ptype)
char *mybuf, *resbuf;
int mybytes, ressize, *sizes;
int Nnodes, *nodelist, *ptype;
```

DOMAIN

exconcat may only be called in the node processors.

DESCRIPTION

This routine is used to collect and concatenate data in a set of node processors.

Each node contributes mybytes bytes of data from the array mybuf to be placed in each node's resbuf. The individual blocks of data are sorted into order of increasing processor number and placed in the resbuf buffer, separated by ressize bytes. If any node contributes more than ressize bytes to the global vector then the excess are discarded. If the pointer sizes is not the special value NULLPTR then the amount of data contributed by each processor is stored in the appropriate slot of the sizes array.

If resbuf takes the special value NULLPTR then the concatenation operation is performed but no data are stored in the node memory.

The group of nodes participating in the concatenation operation is specified by the Nnodes and nodelist arguments. The latter is an array of processor numbers; Nnodes specifies the number of processors in the list. If Nnodes has the special value ALLNODES then the concatenation is performed by all nodes irrespective of the value of the nodelist argument which may even be the NULL pointer.

The ptype argument is used to specify a "type" for the concatenation operation which will distinguish it from other active communication. Any positive value may be supplied - you may not use the DONTCARE or NULLPTR mechanisms for this argument

All nodes involved in the concatenation operation must call exconcat together and with identical values for the Nnodes and nodelist arguments or communication deadlock will occur.

EXAMPLE

Consider a simple case with four processors which have buffers as follows

```
Processor 0: int mybuf[] = {12, 13, 14)
Processor 1: int mybuf[] = {32, 33, 34}
Processor 2: int mybuf[] = {52, 53, 54}
Processor 3: int mybuf[] = {72, 73, 74}
```

In the simplest case we can concatenate all four buffers with the code

```
exconcat(mybuf, 3*sizeof(int), ibuf, 3*sizeof(int),
            NULLPTR, ALLNODES, NULLPTR, &type);
```

which would result in each processor obtaining the following result in `ibuf`

```
ibuf[] = {12,13,14,32,33,34,52,53,54,72,73,74}
```

and the value returned by the call would be 12 times the size of an `int` in each node. Since the `sizes` argument is `NULLPTR` no attempt is made to store the number of bytes sent by each node.

Another simple case is obtained by sending different amounts of data from each processor. Consider the following code:

```
int sizes[4], type = 56;
struct nodenv nodedata;

exparam(&nodedata);
exconcat(mybuf, nodedata.procnum*sizeof(int), ibuf,
        3*sizeof(int),sizes, ALLNODES, NULLPTR, &type);
```

In this case the "result" buffers on each node would be

```
ibuf[]  = {0, 0, 0, 32, 0, 0, 52, 53, 0, 72, 73, 74}
sizes[] = {0, 4, 8, 12}
```

where we have assumed 4-byte integers. If we now change the arguments again by reducing the `ressize` parameter to 8 then the resulting buffers would be

```
ibuf[]  = {0, 0, 32, 0, 52, 53, 72, 73}
sizes[] = {0, 4, 8, 8}
```

In each node the call to `exconcat` would now return an error, setting `errcnt` to 12 and `errno` to `ENOTREAD` to reflect the fact that node 3 attempted to send more data than was to be read.

In the final example we perform the concatenation only in processors 0,1 and 3.

```
#include "express.h"        /* Defines NULLPTR, etc. */

do_conc(mybuf, ibuf)
int *mybuf, *ibuf;
{
    int Nnodes, nodelist[4], type = 12;

    Nnodes = 3;
    nodelist[0] = 0;
```

```
nodelist[1] = 1;
nodelist[2] = 3;

exconcat(mybuf, 3*sizeof(int), ibuf, 3*sizeof(int),
        NULLPTR, Nnodes, nodelist, &type);
```

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXCONC(MYBUF, MYSZ, RES, RESSIZ,
          SIZES, NNODES, NODEL, TYPE)
INTEGER MYBUF(*), MYSZ, RES(*), RESSIZ, SIZES(*)
INTEGER NNODES, NODEL(*), TYPE
```

DIAGNOSTICS

If any error occurs in `exconcat` -1 is returned. Possible sources of error are: illegal values of `mybuf` or `resbuf` and preposterous values of `mybytes` or `ressize`. If no error occurs the total number of bytes stored in memory is returned. An error condition can also be generated if the value of `ressize` on a node is smaller than the amount of data which is being sent by a processor (including the node itself). In this case `errno` is set to `ENOTREAD`, `errcnt` indicates the number of bytes which were actually sent and all excess bytes are discarded. If two or more processors both exceed the requested limit then the error conditions reflect the data sent by the offender with the highest processor number.

SEE ALSO

`exbroadcast, excombine`

NAME

excustom - Indicate an alternative system configuration file.

SYNOPSIS

```
int excustom(file_name)
char *file_name;
```

DOMAIN

Only available to host programs.

DESCRIPTION

excustom indicates that *Express* should use system configuration information from the named file rather than the system default. This allows applications to maintain their own customization programs independent of any other user or system requirements.

To complete the customization process the exinit command has an optional argument which names the customization file which should be used while loading *Express* into the transputer system. Similarly cubix has an additional '-E' switch allowing an alternative file to be named at runtime. In both cases the excustom function is invoked with the named file as argument.

The excustom call must be made before any other *Express* system calls.

RETURN VALUE

The returned value indicates whether or not the indicated customization file was found. Non-zero values indicate a failure to locate the named file.

EXAMPLES

The following code fragment could be used to allow a program the option of using an alternative customization file based on command line arguments.

```
#include "express.h"

main(argc, argv)
int argc;
char **argv;
{
     int pgind;
/*
 * If any command line argument is given use it as the
 * name of the EXPRESS customization file.
 */
     if(argc > 1) {
         if(excustom(argv[1]) != 0) {
             fprintf(stderr, "Problems with file: %s\n",
```

```
                                        argv[1]);
                exit(1);
            }
        }
/*
 * Carry on and use EXPRESS as usual.
 */
    if((pgind=exopen("/dev/transputer", 4, DONTCARE)) < 0) {
```

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXCUST(FILE)
CHARACTER*80 FILE
```

SEE ALSO

excustom (command).

NAME

execve - Overlay a node application with another program.

SYNOPSIS

```
execve(name, argp, envp)
char *name, **argp, **envp;

aexecve(name, argp, envp)
char *name, **argp, **envp;
```

DOMAIN

This routine may only be called in programs compiled with the *Cubix* or *Plotix* libraries.

DESCRIPTION

This routine overlays (and therefore terminates) the calling program by loading a new application from the file `name`. This routine immediately begins execution from its `main` entry point. Unless an error occurs the call to `execve` will not return; if an error occurs -1 will be returned and the external variable `errno` will indicate the appropriate error.

`argp` and `envp` are pointers to NULL terminated vectors of strings which will be passed to the new process through the `argv` and `environ` mechanism in the usual manner. Conventionally the first `argv` should be the name of the new program. In this manner the old process may communicate values to its successor.

By default `execve` causes the overlay to occur in all nodes. It must, therefore, be a loosely synchronous operation. If, however, the default *Cubix* mode is asynchronous then each node performs the overlaying operation independently. This is also the case for the `aexecve` system call.

Since memory is not re-initialized across calls to these routines it is possible to share large blocks of data in each node. To do this it is merely necessary for the data to be placed in a region of memory where none of the intended programs will overwrite it.

EXAMPLE

The following code section causes the program `pass2` to be loaded on top of the currently executing routine. The new program is passed a string argument which contains the address of the variable `pass1_var`.

```
#include <stdio.h>

int pass1_var;/* To be passed to the second phase */

exec_second_pass()
{
     char *args[3], tmpstr[128];
```

```
/* Build argument list for new program */

     args[0] = "pass2";
     sprintf(tmpstr, "%lx", (long)&pass1_var);
     args[1] = tmpstr;
     args[2] = (char *)0;

     if(execve("pass2", args, (char **)0) < 0) {
         fprintf(stderr, "Failed to exec second pass\n");
         return -1;
     }

/* NOTREACHED */
}
```

FORTRAN SYNOPSIS

```
SUBROUTINE KEXEC(PROGNM)
CHARACTER*80 PROGNM

SUBROUTINE KAEXEC(PROGNM)
CHARACTER*80 PROGNM
```

WARNINGS

This function is only available to programs running under *Cubix*. If you are running with a host program the same effect can be achieved by simply using the `exload` routines to download another node application.

BUGS

The current implementation does not respect the "close-on-exec" flags associated with open files; all files remain open across the `exec` call.

SEE ALSO

`exhandle`, `fcntl`.

NAME

exgrid - Automatic domain decomposition tools

SYNOPSIS

```
int exgridinit(griddim, nprocs)
int griddim, *nprocs;

exgridsplit(nodes, griddim, nsplit)
int nodes, griddim, *nsplit;

int exgridcoord(procnum, coord)
int procnum, *coord;

int exgridproc(coord)
int *coord;

int exgridsize(procnum, global, size, start)
int procnum, *global, *size, *start;

int exgridbc(perbc)
int *perbc;

int exgridnode(procnum, dir, dist)
int procnum, dir, dist;
```

DOMAIN

The exgrid routines may be called from both host and node programs.

DESCRIPTION

exgrid collectively refers to a set of utilities that perform automatic decompositions of user domains onto the underlying machine topology. A user specification for a problem domain which has the topology of a Cartesian grid in N dimensions is mapped onto the hardware topology and routines are available to enable processors (defined relative to the user topology) to communicate through the primitive system calls.

exgridinit is the routine which performs the elementary mapping and must be called before any of the other exgrid routines (except exgridsplit). The arguments are the number of dimensions in the user topology and the number of processors to be assigned to each dimension. If the requested topology is successfully mapped to the hardware zero is returned; otherwise the value returned is -1.

The function exgridsplit is used to divide up the nodes processors between the griddim dimensions in as even a way as possible consistent with the requirement that all processors be used. The number in each dimension will be returned in the array nsplit. A simple example would be that of two dimensional decompositions: for eight nodes we would obtain an 4 x 2 decomposition while nine processors yields 3 x 3.

Having set up the exgrid system in this way the other function calls are available to inquire about specific details of the decomposition. Particularly useful is information concerning where, in the user defined topology, a certain processor is to be found. The exgridcoord function call takes a processor number as argument and returns the coordinates in the Cartesian grid of this processor. The inverse transformation is provided by the function exgridproc which takes as arguments an array of coordinates and returns the processor number of the node at that position in the user grid.

The interface to the underlying communication structure is provided by the exgridnode and exgridbc functions. The arguments to the former are a processor number, a direction in the user grid and a distance. The returned value is a "node" suitable for use in calls such as exchange and exvchange which contains the necessary information for communication in that direction. The distance parameter specifies the offset from the current node in the direction indicated so that a value of +1 implies the next node along the positive axis while -1 indicates the next node in the negative direction. Magnitudes greater than 1 are also possible and correspond to multiple hops in the given direction.

The exgridbc function is provided to alter the boundary conditions at the edges of the user domain. By default exgridnode assumes that boundaries are connected periodically so that the processor to the "left" of the leftmost is the one on the extreme right hand edge. To suppress this feature one uses exgridbc. The sole argument is an array of integers, one for each dimension in the user domain. A non-zero value indicates that this dimension is to be considered periodic while a zero value causes exgridnode to return a NONODE at the boundary.

The last function in the exgrid collection, exgridsize, is used to distribute an array over the user grid. The first argument is again a processor number and the second is an array containing the global sizes of the array to be decomposed. After the call the third argument will be an array containing the number of entries in each dimension of the array which lie in the processor specified. The final argument will be an array containing the global index that corresponds to an index of zero in the local array.

A final point to note is that these routines are very useful in conjunction with the low level I/O primitives mread, mwrite, mread2d and mwrite2d which require arguments easily calculated by the exgrid functions.

EXAMPLE

As a simple example consider a problem involving two dimensional images to be executed on eight processors. A suitable call to initialize the system might be

```
#include "express.h"/* Defines nodenv structure */

#define HORIZ 0
#define VERT 1
struct nodenv nodedata;

start_decomp()
{
```

```
int nprocs[2];
int type = 33;/* Use same type for all messages */

exparam(&nodedata);
exgridsplit(nodedata.nprocs, 2, nprocs);
if(exgridinit(2, nprocs) < 0) {
     abort(-1);
}
```

The macros HORIZ and VERT are defined for our convenience and just serve to label the two axes in the grid. We assign processors to the horizontal and vertical directions of our grid by using exgridsplit to divide up the processors.

Now consider a simple scrolling operation in which data is to be passed to the right. We need to figure out the processor numbers of the appropriate nodes in order to communicate in this direction. The simple thing to do in this case is to use exgridnode to calculate the appropriate values. However, one must first consider the boundary values; What should happen when data is scrolled off the right hand edge of he display? The two options are to have it appear on the left hand edge, or to disappear completely. We adopt the latter approach which entails altering the default assumption of exgridnode that boundaries are periodic. The following code uses exgridbc to override this default and exgridnode to assign suitable processor values for the four directions we will be interested in. (We assume that the variables defined and initialized in the previous code segment are still available.)

```
int perbc[2];

perbc[0] = perbc[1] = 0;
exgridbc(perbc);
        /* Suppress periodic boundary conditions */

upnode = exgridnode(nodedata.procnum, VERT, 1);
downnode = exgridnode(nodedata.procnum, VERT, -1);
leftnode = exgridnode(nodedata.procnum, HORIZ, -1);
rightnode = exgridnode(nodedata.procnum, HORIZ, 1);
```

Now all the "nodes" are valid. If a processor is on the extreme left edge of the domain and it tries to communicate with a processor to its left then the value of leftnode has been correctly assigned the value NONODE which will, in turn, direct the communication system to omit communication with this non-existent processor. Note how simple it would be to adopt the alternative strategy and have data scroll off the right edge and re-appear on the left. We simply omit the call to exgridbc (or else change the zero values to ones) and the correct masks would be returned.

To show the actual use of these processor numbers assume that we wish to "scroll" 512 bytes along to the right. In each processor the data is to be taken from an array obuf and

the data coming in from the left is to be read into an array ibuf. The following call to exchange is all that is required

```
exchange(ibuf, 512, &leftnode, &type,
     obuf, 512, &rightnode, &type);
```

Notice that at no point in these calculations did the topology of the hardware enter. Everything is specified in the user domain - i.e., that of the image, and exgrid does the rest.

To demonstrate the use of the exgridsize function assume that the image to be "scrolled" is not 1024 bytes tall as was implicitly assumed in the previous code (We scrolled 512 bytes left in each processor and there are two processors in the vertical direction for a total of 1024 bytes.) Instead we will make the strange choice of an image which is 767 bytes high, and 1024 bytes wide. The exgridsize routine can then be used to tell us how many elements are in each processor through the following code

```
int global[2], sizes[2], start[2];

/* Decompose the array over the processor ring */

global[HORIZ] = 1024;
global[VERT] = 767;
exgridsize(nodedata.procnum, global, sizes, start);
```

At the completion of this call the values sizes[HORIZ] and sizes[VERT] contain the sizes of the subregions assigned to each processor. Further, the values start[HORIZ] and start[VERT] contain the horizontal and vertical index of the first byte that is stored in this processor. In the case described here every processor would have the value 256 for sizes[HORIZ] since the horizontal size is divided exactly by the number of processors in that direction. In the vertical direction, however, the division does not work out correctly and so the processors whose responsibility is the lower half of the display would have sizes[VERT] = 384 while those in the upper half would have 383. Similarly, the processors in the upper half have start[VERT] = 0 while those in the lower half have start[VERT] = 384. The modified call to exchange which scrolls the data to the right is

```
exchange(ibuf, sizes[VERT], &leftnode, &type,
                obuf, sizes[VERT], &rightnode, &type);
```

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXGDIN(GRIDIM, NUM)
INTEGER GRIDDIM, NUM(*)
```

```
INTEGER FUNCTION KXGDSP(NODES, GRIDIM, NSPLIT)
INTEGER NODES, GRIDIM, NSPLIT(*)

INTEGER FUNCTION KXGDCO(PROC, COORD)
INTEGER PROC, COORD(*)

INTEGER FUNCTION KXGDPR(COORD)
INTEGER COORD(*)

INTEGER FUNCTION KXGDSI(PROC, GLOBAL, SIZE, START)
INTEGER PROC, GLOBAL(*), LOCAL(*), SIZE(*), START(*)

INTEGER FUNCTION KXGDBC(PERBC)
INTEGER PERBC(*)

INTEGER FUNCTION KXGDNO(PROC, DIR, SIGN)
INTEGER PROC, DIR, SIGN
```

RETURN VALUE

If any error occurs in the `exgrid` routines they return -1. Particular errors include failing to call `exgridinit` before using the other functions and a failure of `exgridinit` to match the user requested topology onto that of the hardware.

SEE ALSO

`exparam, mread, mread2d, forder`

NAME

exhandle - Asynchronous message handler.

SYNOPSIS

```
int exhandle(function, psrc, ptype)
int (*function)(), *psrc, *ptype;
```

DOMAIN

exhandle may be called in the node processors only.

DESCRIPTION

This routine is used to initialize a "handler" for messages of certain types and sources. The idea is that whenever a message arrives that matches the psrc and ptype parameters the user-supplied procedure function is invoked to process the data. This process occurs immediately upon receipt of the message with as little overhead as possible and can be used to implement a totally asynchronous processing style in which messages can be handled transparently without the intervention of the main application code.

The function is invoked immediately a message has arrived in the internal node buffers with the following arguments

```
function(ptr, length, psrc, ptype)
    char *ptr;
    int length, *psrc, *ptype;
```

i.e., it looks just like a call to exread. Note however, that the supplied ptr argument actually points to a buffer within the *Express* kernel. If the application needs to keep the message for later processing memory must be allocated and the buffer copied. Otherwise the data becomes unavailable when the user function completes.

The src and type fields reflect the actual source and type of the message being handled in cases where "DONTCARE" values were originally supplied to the exhandle function.

The user supplied function must return an integer value to its caller. This value will determine the future behavior of the system; a negative value will terminate the association between the message source/type and the function while positive (and zero) values maintain the ***status quo***. In this way it is possible to have a message handler that is invoked only once, several times until a particular message arrives, or permanently.

EXAMPLE

The following example shows how this function can be used to implement a global, "read only" memory. A handler is set up which intercepts all messages of type MEMREAD and responds by sending back a message containing the memory requested. Obviously one could implement a writable shared memory in a similar manner although problems concerning mutual exclusion would probably have to be addressed.

```
#include "express.h"
```

```
#define MEMREAD(0x7001)/* Type for memory requests */
#define MEMORY (0x7002)/* Type for memory response */

struct memreq {
     int addr;/* Address to read from */
     int length;/* Number of bytes to read */
};

main()
{
     int mem_handler();
     int type, src;

/* Allow anyone to send memory requests */

     type = MEMREAD;
     src = DONTCARE;
     exhandle(mem_handler, &src, &type);
     exsync();

          ......
}

/* This is the function that fields requests for
 * memory. The first argument will point to a request
 * structure and the third to the requesting node.
 * NOTE: we return 0 so that the handler continues to
 * operate.
 */
int mem_handle(req, length, src, type)
struct memreq *req;
int length;
int *src, *type;
{
     int rtype = MEMORY;

     exwrite(req->addr, req->length, &src, &rtype);
     return 0;
}
```

Having set up this message handler we can access memory on another node by simply sending a message of type MEMREAD. Notice that the message handler sends back the data in a message of a different type that it read. This is an important point - if the routine adopted the simpler strategy of returning the same type message as it received then that

message would be trapped by the message handler on the original node and treated as a memory request. In this way an infinite chain of requests would be generated!

The following routine reads `length` bytes of memory from processor `node` and stores it in the specified `buffer`. The routine returns the number of bytes read.

```
read_memory(node, address, length, buffer)
int node, address, length;
char *buffer;
{
     struct memreq req;
     int stype = MEMREAD;
     int rtype = MEMORY;

     req.addr = address;
     req.length = length;
     exwrite(&req, sizeof(req), &node, &stype);
     return exread(buffer, length, &node, &rtype);
}
```

This function forms the basis of an extremely elegant multitasking system under *Express* which is discussed in more detail in the accompanying manual, "Multitasking under *Express*".

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXHAND(FUNC, SRC, TYPE)
INTEGER FUNC, SRC, TYPE
EXTERNAL FUNC
```

DIAGNOSTICS

If the kernel is unable to install the message handler -1 is returned. Otherwise the return value will be 0.

WARNING

The current implementation restricts the length of a message that can be sent to a handler to the size of the *Express* buffers as defined in `excustom`.

SEE ALSO

`exread`, `exreceive`

NAME

exload, exloadl, exloadv, exloadle, exloadve - Load a program.

SYNOPSIS

```
int exload(pgind, nodeprog)
int pgind;
char *nodeprog;

int exloadl(pgind, nodeprog,
          argv0, argv1, ..., argvN, NULLPTR)
int pgind;
char *nodeprog, *argv0, *argv1, ..., *argvN;

int exloadv(pgind, nodeprog, argv)
int pgind;
char *nodeprog, *argv[];

int exloadle(pgind, nodeprog,
          argv0, argv1, ..., argvN, NULLPTR, envp)
int pgind;
char *nodeprog, *argv0, *argv1, ..., *argvN, *envp[];

int exloadve(pgind, nodeprog, argv, envp)
int pgind;
char *nodeprog, *argv[], *envp[];
```

DOMAIN

The exload subroutines may only be called in the host computer.

DESCRIPTION

exload in all its forms loads the program nodeprog into a set of processors previously allocated with exopen. The pgind argument is the ***processor group index*** returned by the exopen call.

The exload package provides the simplest interface to allocating processors and loading application programs. A single application code is loaded into all processors and identical arguments are passed to each. The alternative exploadl (This routine has its own manual page.) is provided if either different arguments or different programs are to be loaded into different processors.

When a conventional C program is entered on a sequential computer it starts as though called with arguments:

```
main(argc, argv)
int argc;
```

```
char *argv[];
```

where `argc` counts arguments and `argv[]` is a vector of pointers to the character strings that make up the arguments. The `exload` routines allow host programs to initiate execution of node programs with arguments in a similar manner.

`exload` is the simplest of the loading routines. It simply loads the node program to all processors (in parallel) and begins execution. No arguments are passed to the node code.

`exloadl` allows the user to explicitly list the arguments that should be passed to the node program. This list must be terminated by the special pointer `NULLPTR`, defined in the header file `express.h`.

`exloadv` is useful when the pointers to the arguments have already been formed into a vector such as is the case with arguments passed to a host program. As always the vector must be terminated by a `NULLPTR` value (note that this is different from a pointer to a null string).

Traditionally the first argument, (`argv0` or `argv[0]`), is the name of the program being run, i.e., `nodeprog`, although the user is free to disregard this convention.

The routines `exloadle` and `exloadve` supply an environment string to the nodes as well as the command line arguments. An environment string is a vector of character strings much like `argv`. Usually, each string consists of a sequence of alphanumeric characters, optionally followed by '=' and another sequence of alphanumerics. The vector is terminated by a `NULLPTR`. In regular C programs (such as that running in the host processor), the environment is passed to the running program in the external variable

```
extern char **environ;
```

Similarly, the environment that is passed to the node program by the host processor can be found in the same external variable and queried with the system call `getenv`.

EXAMPLES

The following code loads a program (called `myprog`) into four processors. The main routine is called with `argc` set to 2, `argv[0]` set to the character string "`myprog`" and `argv[1]` set to the character string "`horse`." Note that the argument "`myprog`" appears twice in the argument list - the first time it is the name of the program to load into the machine while the second occurrence ensures that `argv[0]` in the node program contains the same string as is the conventional practice.

```
#include "express.h"

main()
{
     int pgind;

     if((pgind=exopen("/dev/transputer", 4, DONTCARE)) < 0) {
         fprintf(stderr, "Failed to allocate processors\n");
```

```
        exit(1);
    }
    exloadl(pgind, "myprog", "myprog", "horse", NULLPTR);

        ...
```

The following is an example of how to relay to the node program the same argument vector and environment that are received by the host program. The host program is called `mycpprog`, and looks something like:

```
#include "express.h"

extern char **environ;

main(cpargc, cpargv)
int cpargc;
char *cpargv[];
{
    int pgind;

    if((pgind=exopen("/dev/transputer",
                atoi(cpargv[1]),DONTCARE))<0) {
        fprintf(stderr, "Failed to allocate nodes\n");
        exit(1);
    }
    exloadve(pgind, cpargv[2], &cpargv[2], environ);

        ...
```

The node program, which is called `mynodeprog` looks like

```
#include "express.h"

extern char **environ;

main(nodeargc, nodeargv)
int nodeargc;
char *nodeargv[];
{
            ...
}
```

Finally, after the two programs are linked properly, the host program is executed with the

command

```
mycpprog 8 mynodeprog horse dog
```

which loads the node program into eight processors and passes the arguments "mynodeprog", "horse" and "dog" to the node program. The environment of the host process is also passed to the nodes.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXLOAD(PGIND, PRGNAM)
INTEGER PGIND
CHARACTER*80 PRGNAM
```

FORTRAN DESCRIPTION

KXLOAD may only be called from the host processor, and upon being called loads the named program into the group of nodes indicated by the PGIND argument. This routine is the equivalent of the C routine exload. Since Fortran programs do not have command line arguments the other C loading primitives have no Fortran partners.

DIAGNOSTICS

If any error occurs during loading -1 is returned. Possible sources of error are: an illegal value of pgind or the failure of the system to allocate a the correct number of processors. Errors are also returned if a routine fails to find an appropriate executable to load or if a communication error occurs during loading.

SEE ALSO

expload

NAME

exopen - Allocate a processor group.

SYNOPSIS

```
#include "express.h"

int exopen(device, nodes, start_node)
char *device;
int nodes, start_node;
```

DOMAIN

Only available to host programs.

DESCRIPTION

exopen allocates a processor group containing nodes processors in the device pointed to by the character string device.

The nodes argument indicates the number of nodes to be allocated and the last argument optionally requests a specific set of nodes within the parallel machine. The default value DONTCARE allows any group of nodes to be selected.

The exopen call must be used before attempting to access any processor group.

RETURN VALUE

The returned value is a ***processor group index*** which must be used in all further references to the allocated processors. In cases where no processor group of the appropriate size is available or some other hardware error occurs the value returned is -1.

EXAMPLES

The following code allocates a group of 4 processors anywhere in the parallel machine

```
#include <stdio.h>
#include "express.h"

main()
{
     int pgind;

          ...

     if((pgind=exopen("/dev/transputer", 4, DONTCARE)) < 0) {
         fprintf(stderr, "Failed to allocate processors\n");
         exit(1);
     }
```

If we wanted to allocate a particular set of processors in the machine then we could replace, for example, the DONTCARE argument in the exopen call:

```
exopen("/dev/transputer", 4, 8)
```

which attempts to allocate nodes 8 thru 11. In this case it is even more important that the value returned by exopen be checked since there is a larger chance of failure.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXOPEN(DEVICE, NNODES, STARTN)
CHARACTER*80 DEVICE
INTEGER NNODES, STARTN
```

WARNINGS

In current implementations the DEVICE parameter will be one of

/dev/transputer	Transputer based devices
/dev/ncube	NCUBE systems.
/dev/symult	Symult S2010, 68000 based nodes
/dev/symfpa	Symult S2010, Weitek VFPA nodes

Note that this list is not necessarily exhaustive. It is complete at the time of writing but may be extended at future dates.

Some systems are unable to support the start_node argument to this function.

SEE ALSO

exshare, exload, expload, exread, extest, extype, exwrite

NAME

exparam - Runtime parameters.

SYNOPSIS

```
#include "express.h"

struct nodenv {
     int procnum;
     int nprocs;
     int groupid;
     int taskid;
};

exparam(nodedata)
struct nodenv *nodedata;
```

DOMAIN

exparam may be called in either the host or node processors.

DESCRIPTION

This routine is used when an application program requires to know the details of its runtime environment. The information available and its correspondence to the structure elements defined above is

procnum	Processor number of the calling node. Nodes are numbered consecutively from (and including) 0.
nprocs	Number of processors allocated in this processor group.
groupid	Specifies the ***processor group index*** containing this node.
taskid	Specifies the ***process identifier*** of the process making the call.

The last two pieces of information are currently unused.

The use of this information and the exgrid utilities is the key to writing reconfigurable applications since they allow the program to adapt to different processor configurations at runtime.

EXAMPLE

Assume that we wish to use the exgrid tools to map the parallel machine to a two dimensional mesh of processors. The following code supplies the necessary parameters to the exgrid routine.

```
#include "express.h"/* Defines nodenv structure */

#define MESH 2/* Mesh has two dimensions */
main()
```

```
{
    struct nodenv nodedata;
    int nprocs[MESH];  /* No. procs in each dimension */

    exparam(&nodedata);     /* Get runtime parameters */
                            /* Divide up processors */
    exgridsplit(nodedata.nprocs, MESH, nprocs);

    if(exgridinit(MESH, nprocs) < 0) {
        abort(1);
    }
```

Note that we use the `exgridsplit` function to divide up the processors between the physical dimensions.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXPARA(ENV)
INTEGER ENV(4)
```

FORTRAN DESCRIPTION

The `KXPARA` call fills in the entries of the `ENV` array with runtime parameters describing the processor configuration. The correspondence between the array elements and the C-structure fields is as follows

```
ENV(1)    procnum
ENV(2)    nprocs
ENV(3)    groupid
ENV(4)    procid
```

SEE ALSO

`exgrid`

NAME

`expause` - Arrange for programs to be loaded "stopped".

SYNOPSIS

```
int expause()
```

DOMAIN

Only available to host programs.

DESCRIPTION

This routine is used to control the initial state of a program or programs being loaded into groups of processors. By default node programs start immediately. If `expause` is used before the appropriate `exload` call then the programs will halt at their first instruction after loading. This is useful when using the debugger, `ndb`, since it allows the user to control the entire course of execution by setting breakpoints etc.

EXAMPLE

Consider the case where debugging is occasionally required. The following code segment illustrates the use of `expause` to load programs in a stopped state if more than three runtime arguments are supplied to the host program. Otherwise programs will be loaded in the (default) running state.

```
#include "express.h"

main(argc, argv)
int argc;
char *argv[];
{
     int pgind;

/* Allocate 4 processors to run node program */

     if((pgind=exopen("/dev/transputer", 4, DONTCARE)) < 0) {
         fprintf(stderr, "Failed to allocate processors\n");
         exit(1);
     }

/* If user supplied more than three runtime args, load
 * stopped.
 */

     if(argc > 3) expause();

/* Finally load application program into nodes */
```

```
if(exload(pgind, "nodeprog") < 0) {
    fprintf(stderr, "Failed to download program\n");
    exit(2);
}
```

FORTRAN SYNOPSIS

```
SUBROUTINE KXPAUS
```

SEE ALSO

exopen, exshare, exinit (command).

NAME

exploaad, exargldl, exargldv, exenvld - Load a program into individual nodes.

SYNOPSIS

```
int exploaad(pgind, nodeprog, node)
int pgind, node;
char *nodeprog;

int exargldl(pgind, node,
                arg0, arg1, ..., argN, NULLPTR)
int pgind, node;
char *arg0, *arg1, ..., *argN;

int exargldv(pgind, node, argv)
int pgind, node;
char *argv[];

int exenvld(pgind, node, envp)
int pgind, node;
char *envp[];
```

DOMAIN

These routines may only be called in the host processor.

DESCRIPTION

These routines provide a complementary interface to the exload routines for loading programs into groups of processors. Instead of loading the entire array with a single node program these routines allow different applications to be loaded into individual nodes of the machine.

In each case a previous call to exopen must have allocated a set of processors into which we are attempting to load programs. The ***processor group index*** returned by this call must be supplied to the exploaad functions as the argument pgind.

Having allocated a group of nodes user applications are loaded with the exploaad primitive which loads the named code into the processor specified by the node argument. In this call and the ones for loading arguments the special value ALLNODES defined in express.h specifies that all processors are to be loaded with the same item.

Arguments and an environment string can also be passed to individual node programs using the exargld and exenvld primitives. The difference between the two forms of the exargld routine concerns whether the arguments are listed in the subroutine call or given as a vector of pointers. The description of the corresponding exload calls should clarify this point. These routines are optional and may be omitted if no arguments are to be passed.

Before execution of the node program can begin calls must be made to the exstart and exmain functions.

EXAMPLES

The following calls allocate, load and start a program in four processors

```
#include "express.h"

main()
{
     int pgind;

     if((pgind=exopen("/dev/transputer", 4, DONTCARE) < 0) {
         fprintf(stderr, "Failed to allocate processors\n");
         exit(1);
     }
     exload(pgind, "myprog", ALLNODES);
     exstart(pgind, ALLNODES);
     exmain(pgind, ALLNODES);
         ...
```

Note that the particular arguments chosen here make this code functionally equivalent to a call to `exload`.

In the following example we load the programs "`prog1`" into nodes 0 through 3 and "`prog2`" into nodes 4 through 15 of a sixteen processor group.

```
#include "express.h"

main()
{
     int pgind;

     if((pgind=exopen("/dev/ncube", 16, DONTCARE)) < 0) {
         fprintf(stderr, "Failed to allocate processors\n");
         exit(1);
     }
     for(i=0 ; i<4 ; i++) exload(pgind, "prog1", i);
     for(i=4 ; i<16; i++) exload(pgind, "prog2", i);

     exargldl(pgind, ALLNODES,
             "horse", "donkey", "cat", NULLPTR);
     exstart(pgind, ALLNODES);
     exmain(pgind, ALLNODES);
         ...
```

Notice that we also pass a list of arguments to the loaded programs - the same list to each

processor.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXPLOA(PGIND, PRGNAM, NODE)
INTEGER PGIND, NODE
CHARACTER*80 PRGNAM
```

FORTRAN DESCRIPTION

This routine loads the named program into the processor specified by the NODE argument. The special value NODE=IALNOD causes all processors to receive the same program. The IALNOD value is found in the XPRESS common block which is initialized by the call to KXINIT. The PGIND argument indicates the group of processors and is the value returned by a successful call to KXOPEN.

DIAGNOSTICS

exploa returns zero upon successful loading of the executable program. If the executable file is not found, or is invalid in some way the value -1 is returned.

SEE ALSO

exload, exstart.

NAME

exread - Read a message

SYNOPSIS

```
#include "express.h"

int exread(buf, length, psrc, ptype)
int length, *psrc, *ptype;
char *buf;
```

DOMAIN

exread is available to both host and node programs with identical calling sequences.

DESCRIPTION

This call is used to read messages in the *Express* system. This routine provides the simplest interface to the message system - a blocking read; the function only returns when a suitable message has been read.

The accepted message is read into the buffer pointed to by the buf argument and is truncated to size length bytes if necessary. The source and type of the message to be read can be specified by the pointer arguments psrc and ptype as described below.

This routine blocks until a message with suitable parameters has been received.

OPTIONS

Under *Express* messages have both destinations and types which are used by reading processes to distinguish between various available messages. A message will only be read if it matches, in both source and type, the parameters supplied in the read call. However, several options are available to allow the user extra flexibility. Both source and type fields are treated equivalently at this level so the following discussion applies equally to both.

*psrc = DONTCARE	A message will be read from any node. The particular node will be indicated by *modifying* the value under the pointer psrc. Note that the original DONTCARE value is overwritten by this call.
psrc = (char *)NULL	If the NULL pointer is specified then a message will be accepted from any of the nodes included in the above case but no attempt will be made to indicate where the message came from.
*psrc = number	Any positive numeric value will restrict attention to messages with that particular source.

These same considerations apply to the type field, ptype, except that the interpretation of the wildcard value, DONTCARE, is subject to modification through the extype system calls.

The special value HOST is used by nodes wishing to send messages to the host processor.

RETURN VALUE

The value returned is the length of the received message, after any necessary truncation has been performed. If some sort of hard error occurs then -1 is returned.

EXAMPLES

In the following examples we consider a case in which the following four messages have arrived on our node in the order given.

```
1. Source 1        Type 12        Length 32
2. Source HOST     Type 2         Length 512
3. Source 1        Type 15        Length 1024
4. Source 2        Type 0         Length 0
```

The simplest case is where both source and type are explicitly stated as in the call

```
int src=1, type=15;
stat = exread(buffer, 512, &src, &type);
```

In this case message three will be accepted for reading. Note, however, that the actual message is longer than the request length so only the first 512 bytes will be read and the rest discarded. The returned value, `stat` will be 512.

In the next case we give wildcard values to both arguments

```
stat = exread(buffer, 512, (char *)0, (char *)0);
```

This will result in the reading of the first message. Since its actual length is less than or equal to the length requested the entire message will be read and the returned value, `stat = 32`, the number of bytes read.

The next example uses the wildcard value, `DONTCARE`, to read a message but retain information about its source.

```
source = DONTCARE;
type = 0;
stat = exread(buffer, 512, &source, &type);
```

In this case the type is explicitly given and so message 4 will be read. The returned value will be 0, the length of the message read and the `source` variable will contain 2, the source of the message.

In the last example a wildcard value is given for the type field

```
source = 1;
type = DONTCARE;
```

```
stat = exread(buffer, 512, &source, &type);
```

In this case the source is given explicitly and the type allowed to take any value. With the parameters shown message 1 will be read and the value 12 stored in the `type` variable. 32 bytes will be copied into the user buffer and the same value returned as `stat`. Note that types are subject to extra processing through the `extype` commands. If we had specifically excluded type 12 from consideration then message 3 would have been read instead since it has the correct source and has not been excluded. If we had excluded both types 12 and 15 then the call to `exread` would block until a more suitable message arrived.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXREAD(BUFFER, LENGTH, SOURCE, TYPE)
INTEGER BUFFER(*), LENGTH, SOURCE, TYPE
```

WARNINGS

Types are restricted to be positive integers less than 16384. Other message types are reserved for use within the *Express* kernel.

A common "bug" introduced by the `DONTCARE` value is to write code similar to the following

```
node = DONTCARE;
type = 123;
for(i=0; i<nprocs; i++) {
    stat = exread(buf, sizeof(buf), &node, &type);
}
```

when trying to read a response from every node, in any order. The error here is that the first call to `exread` overwrites the `DONTCARE` value for `node` with the actual processor number of the node which responded. As a result the second call to `exread` specifies the *same* processor as sent the first message - probably not the intention of the program. The simple correction is to either use the `NULLPTR` value or else reset the `node` variable to `DONTCARE` at each loop iteration.

SEE ALSO

`exopen`, `exshare`, `extest`, `exwrite`, `extype`, `exgrid`.

NAME

exreadfd, exwritefd - Write a message to/from a file

SYNOPSIS

```
#include "express.h"

int exreadfd(fd, length, psrc, ptype)
int fd, *psrc, *ptype;
long length;

int exwritefd(fd, length, pdest, ptype)
int fd, *pdest, *ptype;
long length;
```

DOMAIN

These routines are only available on the host computer.

DESCRIPTION

These routines perform a very similar function to exread and exwrite with the exception that the appropriate message is not buffered but comes from or goes to a file. Another slight difference is that the length parameter is a long integer allowing file operations on large amounts of data.

exreadfd reads a message with the indicated source and type and places it into the file indicated by the file descriptor fd.

exwritefd writes a message to the specified destination. The message is taken from the file indicated by the file descriptor fd.

In both cases the interpretation of the src, dest and type arguments is exactly as in the corresponding call exread or exwrite.

Appropriate file descriptors for use with these functions are those obtained from the open, creat, etc. system calls.

RETURN VALUE

These functions return the number of bytes read (exreadfd) or written (exwritefd). If an error occurs in writing to or reading from the disk -1 is returned and the external variable errno reflects the problem.

EXAMPLES

The following code fragment sends 64 Kbyte blocks of data to each of 16 processors taken from the file "noddy.dat".

```
#include <fcntl.h>/* Defines O_RDONLY flag */
#include <stdio.h>
```

```
main()
{
      int fd, type=28;
      long length = 64*1024L;

/* Code here to allocate nodes and download programs
   etc.... */

                  ...

/* Open file and distribute data */

      if((fd=open("noddy.dat", O_RDONLY)) < 0) {
          fprintf(stderr,
              "Failed to open file for reading\n");
          exit(0);
      }
      for(node=0; node<16; node++) {
          if(exwritefd(fd, length, &node, &type) < 0) {
              fprintf(stderr, "File error: %d\n", errcnt);
              exit(2);
          }
      }
```

WARNINGS

These routines have long integer parameters for message length distinct from the standard `exread` and `exwrite` routines. This allows large files to be created but may cause a little confusion on machines where `int` and `long` are different sizes.

SEE ALSO

`exread`, `exwrite`

NAME

exreceive - Non-blocking read function.

SYNOPSIS

```
int exreceive(buffer, length, psrc, ptype, status)
int length, *psrc, *ptype, *status;
char *buffer;
```

DOMAIN

exreceive may be called in only the node processors.

DESCRIPTION

This function provides a non-blocking read function for *Express* messages. It is intended for use in applications such as "double-buffering" in which one wishes to process some data while waiting for another message to arrive.

When called it looks for a message in the buffers that matches the supplied psrc and ptype parameters. If such a message exists it is read as though by a normal call to exread and the value pointed to by status will contain the message length.

If no message exists which matches the requested parameters the value -1 is written under the status flag and the function immediately returns to its caller. When a message of the correct type and source subsequently arrives it will be read into memory at the address buffer and the length will be written under the status variable replacing the -1. The psrc and ptype variables will also be updated at that time to reflect the newly read message.

The interpretation of the first four arguments is exactly as in the corresponding call to exread. The last argument, status, is a mechanism by which one can poll for the arrival of the requested message; while negative, no message has been received.

EXAMPLE

The following example is a sketch of a typical "double-buffered" application. We assume that processor source is sending messages of type PROCESS which must be passed to the function grind_away for processing. When all messages for such treatment have been received a message of type FINISHED will be sent. We assume that each of the PROCESS messages will be of no more than 1024 bytes.

```
#include "express.h"

#define PROCESS  (0x7001)   /* Type for grinding on */
#define FINISHED (0x7002)   /* Type for "done" */

char buffer[2][1024];/* For buffering data */

do_grind(node)
```

```
int node;       /* Source of data for processing */
{
     int done, type, this, next;
     int stat[2];

     done = 0;/* Not done yet */
     this = 0;/* Start using "slot" 0 */
     next = 1;

/* Get first buffer, blocking read this time */

     type = DONTCARE;
     stat[this] = exread(buffer[this], 1024, &node, &type);

     do {
         if(type != FINISHED) { /* Read another block */
             type = DONTCARE;
             exreceive(buffer[next], 1024,
                       &node, &type, stat+next);
         }
         else done = TRUE;
                 /* Finish after processing this block */

         grid_away(buffer[this], stat[this]);

/* If we're not done wait for next buffer */

         if(!done) {
             while(stat[next] < 0);
             next = (next + 1) % 2;
             this = (this + 1) % 2;
         }
     }
     while(done != TRUE);
}
```

There are several points to note in this code. We assume that we must process the buffer with the `FINISHED` type - this saves us a message since we can send valid data and still use the type field to convey the important information. We also save the length of the message we are going to process in the `stat` variable - this could be important in the `grind_away` function. Note that it would be dangerous to use a single variable here since it would get overwritten whenever the second buffer arrived - possibly before the call to `grind_away` had been passed the value. Finally note that we have to keep setting `type = DONTCARE` since its value is overwritten whenever a message comes. Failing to do this is quite a common error and would result in the failure to read the `FINISHED` message.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXRECV(BUFFER, LENGTH,
                        SRC, TYPE, STATUS)
INTEGER BUFFER(*), LENGTH, SRC, TYPE, STATUS
```

RETURN VALUE

This function returns zero unless insufficient memory is available to register the read function. In this case -1 is returned.

SEE ALSO

exread, exhandle

NAME

exsem - Various semaphore operations.

SYNOPSIS

```
#include <express.h>

EXSEM *exsemalloc()

void exsemfree(semptr)
EXSEM *semptr;

void exsemwait(semptr)
EXSEM *semptr;

void exsemsig(semptr)
EXSEM *semptr;
```

DOMAIN

These routines may only be called in node programs.

DESCRIPTION

These routines implement a semaphore mechanism essential to protect critical sections of code in a multitasking environment.

`exsemalloc` allocates a new semaphore and sets it so that the first call to `exsemwait` will not block. If no more semaphores are available a `NULL` pointer will be returned.

`exsemfree` returns a semaphore to the available pool for reuse.

Each call to `exsemwait` checks the status of the associated semaphore. If locked the calling task sleeps until another process unlocks the semaphore. While sleeping no CPU time is expended allowing other tasks to proceed.

The `exsemsig` call unlocks the indicated semaphore allowing other processes to enter a critical section of code.

EXAMPLE

The following code could be used to implement a global shared memory system for a distributed memory machine. We will assume that the data being accessed is such that only one process can be allowed access at any one time. This would be the case where, say, extended records are being written to memory in which case the integrity of any particular record is crucial. We would not, for example, allow two processes to both write records simultaneously since they may each write half leaving inconsistent data.

To implement these ideas we need to register a message handler which will field the read/write requests. For simplicity we will use only one handler for both purposes and let the data sent indicate the requested operation. For our own convenience we define a simple data structure to encode these requests.

The necessary message handler is as follows

```
#include <express.h>

#define MEM_READ     (97)    /* Request to read memory */
#define MEM_WRITE    (98)    /* Request to write memory */
#define MEM_ACCESS   (99)    /* Message type for mem access */
#define MEM_RESP     (100)   /* Message type for replies */
#define MEM_DATA     (101)   /* Data to be written */

extern EXSEM *mem_sem;       /* Memory protection semaphore */

struct mem_req {
     int code;       /* Which operation to perform */
     int address;    /* Memory address */
     int nbytes;     /* Number of bytes */
};

mem_access(ptr, length, src, type)
struct mem_req *ptr;
int length, *src, *type;
{
     int rtype;

     exsemwait(mem_sem);/* Wait for unique access */

     switch(ptr.code) {
     case MEM_READ:
         rtype = MEM_RESP;
         exwrite(mem_req.address, mem_req.nbytes,
                                  src, &rtype);
         break;
      case MEM_WRITE:
         rtype = MEM_DATA;
         exread(mem_req.address, mem_req.nbytes,
                                  src, &rtype);
         break;
     }
     exsemsig(mem_sem);
     return 0;
}
```

We have assumed in the above code that the call to `exhandle` which sets up this handler is made elsewhere. Similarly the `mem_sem` semaphore should be allocated before any use will be made of this routine.

To use these routines it is merely necessary to add the following calls. (Assume that the previous macros are defined here also.)

```
mem_read(ptr, length, node, address)
char *ptr;
int length, node;
long address;
{
     struct mem_req mem_task;
     int type = MEM_REQ, rtype = MEM_RESP;

     mem_task.code = MEM_READ;
     mem_task.address = address;
     mem_task.nbytes = length;

     exwrite(&mem_task, sizeof(mem_task), &node, &type);
     return exread(ptr, length, &node, &rtype);
}

mem_write(ptr, length, node, address)
char *ptr;
int length, node;
long address;
{
     struct mem_req mem_task;
     int type = MEM_REQ, rtype = MEM_RESP;

     mem_task.code = MEM_WRITE;
     mem_task.address = address;
     mem_task.nbytes = length;

     exwrite(&mem_task, sizeof(mem_task), &node, &type);
     return exwrite(ptr, length, &node, &rtype);
}
```

Notice that several potential improvements could be made to this code. In particular we could speed up the writing process by sending short amounts of data in the same message as invokes the `mem_access` handler. (`exhandle` can only deal with messages up to the system packet size defined in `excustom`, so any extra could be sent in a second message.) A further bottleneck is due to the fact that we have a single semaphore protecting a large memory space on each node. It might be more practical to have separate semaphores protecting disjoint areas of memory so that fewer processes would have to "sleep".

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXSEMA(ISEM)
```

```
INTEGER ISEM(2)

SUBROUTINE KXSEMW(ISEM)
INTEGER ISEM(2)

SUBROUTINE KXSEMS(ISEM)
INTEGER ISEM(2)
```

FORTRAN DESCRIPTION

The Fortran functions perform the same functions as their C counterparts with the exception that memory for the semaphore structure must be explicitly provided by the user. This should consist of two integers as shown in the above synopsis. Note that there is no need for a "free" function in this case.

SEE ALSO

`exsleep`, `exhandle`, `excustom` (command) and the discussion of *Express* buffering mechanisms in the `excustom` chapter of the users guide.

NAME

exsend - Non-blocking write function.

SYNOPSIS

```
int exsend(buffer, length, psrc, ptype, status)
int length, *psrc, *ptype, *status;
char *buffer;
```

DOMAIN

exsend may be called in only the node processors.

DESCRIPTION

This function provides a non-blocking write function for *Express* messages. It is intended for use in applications such as "double-buffering" in which one wishes to process some data while waiting for another message to arrive or be sent.

This routine provides a mechanism by which a node can transmit a message and then carry on processing regardless of whether or not the message has actually been sent. Upon return from the kernel the status variable is set to -1. When the message is finally processed this value will be changed to the number of bytes sent. Until this has happened the user should (probably) not alter the data in the message since it is unknown which bytes have been transmitted to the receiving node and which have yet to be sent.

The interpretation of the first four arguments is exactly as in the corresponding call to exwrite. The last argument, status, is a mechanism by which one can poll for the final dispatch of the requested message; while negative, the message has still to be sent.

EXAMPLE

The following example is a sketch of a typical "pipelined" application. We assume that processor source is sending messages of type PROCESS which must be passed to the function grind_away for processing, and then forwarded to node dest for further processing. When all messages for such treatment have been dealt with a message of type FINISHED will be sent. We assume that each of the PROCESS messages will be of no more than 1024 bytes.

```
#include "express.h"

#define PROCESS  (0x7001)   /* Type for grinding on */
#define FINISHED (0x7002)   /* Type for "done" */

char buffer[3][1024];       /* For buffering data */

pipeline(src, dest)
int src;        /* Source of data for processing */
int dest;       /* Destination of data */
{
```

```
        int done, this, next;
        int stat[3], type[3];

        done = 0;           /* Not done yet */
        last = -1;          /* Last buffer written */
        this = 0;           /* Start using "slot" 0 */
        next = 1;

/* Get first buffer, blocking read this time */

        type[this]=DONTCARE;
        stat[this]=exread(buffer[this],1024,&src,&type[this]);

        do {
            if(type[this] != FINISHED) {/* Read another block */
                type[next] = DONTCARE;
                if(last >= 0)
                    while(stat[last] < 0);
                exreceive(buffer[next], 1024,
                            &src, type+next, stat+next);
            }
            else done = TRUE;
                    /* Finish after processing this block */

            grid_away(buffer[this], stat[this]);
            exsend(buffer[this], stat[this], &dest, &type[this],
                    stat+this);

/* If we're not done wait for next buffer */

            if(!done) {
                while(stat[next] < 0);
                last = (last + 1) % 2;
                next = (next + 1) % 2;
                this = (this + 1) % 2;
            }
        }
        while(done != TRUE);
}
```

There are several points to note in this code. We assume that we must process the buffer with the `FINISHED` type - this saves us a message since we can send valid data and still use the type field to convey the important information. We also save the length of the message we are going to process in the `stat` variable - this could be important in the `grind_away` function. Note that it would be dangerous to use a single variable here since

it would get overwritten whenever the second buffer arrived - possibly before the call to `grind_away` had been passed the value. Finally note that we have to keep setting `type = DONTCARE` since its value is overwritten whenever a message comes. Failing to do this is quite a common error and would result in the failure to read the `FINISHED` message.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXSEND(BUFFER, LENGTH,
                        SRC, TYPE, STATUS)
INTEGER BUFFER(*), LENGTH, SRC, TYPE, STATUS
```

RETURN VALUE

This function returns zero unless insufficient memory is available to register the write function. In this case -1 is returned.

SEE ALSO

`exwrite, exhandle, exreceive`

NAME

exshare - Share a processor group with another process

SYNOPSIS

```
#include "express.h"

int exshare(device, pid, pnodes)
char *device;
int pid, *pnodes;

int expid(unix_ID)
int unix_ID;
```

DOMAIN

Host processor only.

DESCRIPTION

The exshare routine allows two or more host processes to share access to the same processor group. The first argument, device, specifies which array contains the processor group to be shared and is interpreted exactly as in the exopen call. The process ID of the process with which the processor group is to be shared must be specified by pid. Upon return the number of nodes in the shared processor group is written under the pointer pnodes.

The most reliable source of information about process ID's is provided by the exopen system call which reports the appropriate value. Similar information is often available from the exstat function. On UNIX machines the function expid is available whose argument is the UNIX process ID. The returned value is the *Express* process I.D, suitable for giving to the exshare function.

RETURN VALUE

The value returned by exshare is the ***processor group index*** which must be used in future references to the shared processors.

If the indicated process has terminated or is not using any processors itself the value -1 is returned.

EXAMPLE

The following code would be used if a second process wished to share the processor group currently assigned to the process with ***process-ID*** 349.

```
#include <stdio.h>
#include "express.h"

main()
```

```
{
    int Nnodes, msg_type, msg_src, pgind;

    if((pgind=exshare("/dev/transputer",
                  expid(349), &Nnodes)) < 0) {
        fprintf(stderr,
            "Failed to share nodes, job may have ended\n");
        exit(-1);
    }
    else fprintf(stderr, "Sharing %d nodes\n", Nnodes);

/* Successfully shared nodes, restrict wildcard message
 * types and start reading
 */
    exinctype(123, 125);

    msg_type = msg_src = DONTCARE;
    exread(buffer, 512, &msg_src, &msg_type);
```

Note that having successfully shared the nodes with process 349 we use the `extype` functions to restrict attention to the message types from 123 to 125. This allows us the freedom to use the wildcard `DONTCARE` values in reading without clashing with the process whose nodes we are sharing.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXSHAR(DEVICE, PROCID, NODES)
CHARACTER*80 DEVICE
INTEGER PROCID, NODES

INTEGER FUNCTION KXPID(IPID)
INTEGER IPID
```

WARNINGS

Communicating with shared processor groups is complicated by interactions between source and type fields specified using the `DONTCARE` wildcard. This situation can be eased somewhat through the `extype` mechanisms which restrict the ranges indicated by wildcard values.

SEE ALSO

`exopen`, `extype`.

NAME

exsleep - Pause process.

SYNOPSIS

```
exsleep(msecs)
int msecs;
```

DOMAIN

This routine may be called in any node program.

DESCRIPTION

This routine is used when a process needs to wait for an event without using CPU resources. The supplied argument is the minimum time to wait in microseconds. This routine should be used in multitasking applications where one task needs to wait for an event which will potentially be generated by another task on this node.

EXAMPLE

The following code makes use of the exsleep function to implement a global semaphore - i.e., a semaphore that can be used from any node. For definiteness we assume that the physical semaphore is located on node 0. In this node we should register the following function with a call to exhandle.

```
#include "express.h"

#define WAIT     (1)     /* Wait on global semaphore */
#define SIGNAL   (2)     /* Signal global semaphore */
#define OPEN     (3)     /* Global semaphore "open" */
#define CLOSED   (4)     /* Global semaphore "closed" */

#define SEM_REQ      (801)   /* Req for sem wait or signal */
#define SEM_RESP     (800)   /* Replies to callers */

extern EXSEM *gbl_sem;
extern int gbl_flag;

gbl_semaphore(msg, length, src, type)
int *msg, length, *src, *type;
{
     int resp, rtype = SEM_RESP;

     if(*msg == WAIT) {
         exsemwait(gbl_sem);
         if(gbl_flag == OPEN) {
             gbl_flag = CLOSED;
```

```
            resp = OPEN;
        }
        else resp = CLOSED;
        exsemsig(gbl_sem);
        exwrite(&resp, sizeof(resp), src, &rtype);
    }

    if(*msg == SIGNAL) {
        exsemwait(gbl_sem);
        gbl_flag = OPEN;
        exsemsig(gbl_sem);
    }

    return 1;
}
```

Note that we implement the global semaphore with a simple variable, `gbl_flag` to which access is restricted with the local semaphore, `gbl_sem`. If the semaphore is "locked" a message is sent back to the requesting node indicating that it should sleep. The code which implements the "signal" and "wait" requests for this global semaphore is shown below. For simplicity we assume that the same macros are defined as in the previous code segment.

```
gbl_signal()
{
    int req = SIGNAL, dest = 0, type = SEM_REQ;

    exwrite(&req, sizeof(req), &dest, &type);
}

gbl_wait()
{
    int req = WAIT, dest = 0, type = SEM_REQ;
    int rtype = SEM_RESP, status;

    status = CLOSED;
    while(status == CLOSED) {
        exwrite(&req, sizeof(req), &dest, &type);
        exread(&status, sizeof(status), &dest, &rtype);

        if(status == CLOSED) exsleep(10);
    }
}
```

The important point to note in this code is the call to `exsleep` in the last routine. This

allows other processes on a node to proceed even though the calling process is blocked waiting for the global semaphore.

FORTRAN SYNOPSIS

```
SUBROUTINE KXSLEE(MSECS)
INTEGER MSECS
```

SEE ALSO

exsem, exhandle.

NAME

exstart, exmain - Start execution of program

SYNOPSIS

```
#include "express.h"

exstart(pgind, node)
int pgind, node;

exmain(pgind, node)
int pgind, node;
```

DOMAIN

Available to host processes only.

DESCRIPTION

These routines begin execution of a program previously loaded into a node with the expload system call. Programs loaded with exload do not need to use these routines.

The two routines perform subtly different roles in starting up the user program. exstart is called after expload but before arguments are loaded with the exenvld or exargld routines. exmain is called after all arguments are loaded and actually starts up the user application.

The special value node = ALLNODES may be specified to either routine to perform the action on all allocated nodes.

EXAMPLE

The following example shows the correct use of exstart to begin execution of a job successfully loaded into the nodes.

```
#include <stdio.h>
#include "express.h"

main()
{
     int pgind;

     pgind = exopen("/dev/ncube", 4, DONTCARE);
     if(pgind < 0) {
         fprintf(stderr, "Failed to allocate processors\n");
         exit(1);
     }

/* Load program into processor group using index returned */
```

```
    if(exploadpgind, "noddy", ALLNODES) < 0) {
        fprintf(stderr, "Failed to load program\n");
        exit(2);
    }

/* Start application running */

    exstart(pgind, ALLNODES);
    exmain(pgind, ALLNODES);
```

Note that the calls to `exstart` and `exmain` can be used to explicitly control when a process begins executing. It may be important, for example, that certain actions be performed on the host before execution begins. In this case the "start" calls can be deferred until an appropriate time.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXSTAR(PGIND, NODES)
INTEGER PGIND, NODES

INTEGER FUNCTION KXMAIN(PGIND, NODES)
INTEGER PGIND, NODES
```

SEE ALSO

`exload`, `exploadexopen`

NAME

_ex_swab, _ex_swaw, _ex_swad - Byte swapping routines

SYNOPSIS

```
void _ex_swab(from, to, nbytes)
char *from, *to;
int nbytes;

void _ex_swaw(from, to, nbytes)
char *from, *to;
int nbytes;

void _ex_swad(from, to, nbytes)
char *from, *to;
int nbytes;
```

DOMAIN

These routines may be called in any *Express* program.

DESCRIPTION

An unfortunate problem with many parallel processing systems is that the host machines and node processors have different CPU types. It is often then the case that the binary representation of various data types is different. Typical examples are Sun workstations hosting transputer or NCUBE systems. The former has a Motorola CPU with the most significant byte of a word having the lowest memory address while the node processors store the least significant byte first.

To aid with these problems *Express* provides a set of byte swapping primitives for transforming data between "big" and "little" endian machines. Each routine has a pair of pointer arguments which denote the buffers from which data should be taken and into which it should be placed after swapping. These two pointers may be the same. The last argument, in each case, is the number of bytes in the buffer to be swapped. This should be a multiple of the size of each item being swapped.

The three routines each serve a different swapping style as follows:

_ex_swab	Swaps adjacent bytes in 2-byte quantities
_ex_swaw	Reverses the bytes in 4-byte quantities - i.e., the original order {0,1,2,3} becomes {3,2,1,0}.
_ex_swad	Reverses the bytes in 8-byte quantities - the original order {0,1,2,3,4,5,6,7} becomes {7,6,5,4,3,2,1,0}.

Note that these routines are sufficient to transform data items between Motorola byte ordered machines (Sun workstations, etc.) and INTEL byte ordered machines (NCUBE, transputers, etc.)

EXAMPLE

When necessary, byte swapping typically occurs in one of two places depending on the programming model in use.

In "Host-node" programs it is typical to have to swap all data items that are transmitted to or received from the nodes. The issue of which processor should perform the byte swapping is one of pure convenience - either the host or the nodes can swap the bytes. Often this decision is made according to who has to further use the data being swapped - the following code fragment represents a typical bug

```
/*
 * Byte swapping in a "host-node" program - INCORRECT
 */
#include "express.h"

iterate(ntimes)
     long ntimes;
{
     int i, type = 123;
#ifdef SWAP
     _ex_swaw(&ntimes, &ntimes, sizeof(ntimes));
#endif
     exbroadcast(&ntimes, HOST, sizeof(ntimes), ALLNODES,
                 NULLPTR, &type);
/*
 * This is a BUG ..... ntimes just had its bytes swapped!
 */
     for(i=0; i<ntimes; i++) {
             ....
     }
```

This code shows some typical features in a byte-swapping environment. We have a pre-processor symbol, `SWAP`, controlling the byte swapping routine - only if this is defined will the byte swapping primitives be called allowing the same source code to be used on all types of machines. The "bug" in the above code is that the host program swaps the bytes in the `ntimes` value and sends it to the nodes (correct) but then attempts to use the value in the following loop - *without swapping the bytes back.* As a result the loop will probably run for an extremely long time!

Among several possible "fixes" are:

- Adding another call to `_ex_swaw` after the call to `exbroadcast` to restore the `ntimes` variable to its proper state.
- Making a temporary variable, swapping `ntimes` into it and sending the temporary value to the nodes.

- Having the nodes do the byte swapping in this case.

Cubix programs should only have byte swapping problems when performing binary I/O. Regular text files should pose no problems since the internal protocols take care of all appropriate byte swapping. Arguments to system calls that will be sent to another host are also byte swapped automatically. For binary files, however, the problem remains and the only viable solution seems to be the insertion of many calls to the appropriate swapping routine.

FORTRAN SYNOPSIS

```
SUBROUTINE KXSWAB(FROM, TO, NBYTES)
INTEGER*2 FROM(*), TO(*)
INTEGER NBYTES

SUBROUTINE KXSWAW(FROM, TO, NBYTES)
INTEGER*4 FROM(*), TO(*)
INTEGER NBYTES

SUBROUTINE KXSWAD(FROM, TO, NBYTES)
REAL*8 FROM(*), TO(*)
INTEGER NBYTES
```

FORTRAN DESCRIPTION

These routines perform byte swapping in the same way as their C counterparts. Note that although the arguments above indicate that the `KXSWAW` routine expects `INTEGER*4` arrays `REAL*4` arrays are also acceptable.

SEE ALSO

`mread`, `mread2d`, `mwrite`.

NAME

exsync - Synchronization primitive

SYNOPSIS

```
int exsync()
```

DOMAIN

exsync may only be called from the nodes.

DESCRIPTION

This routine is used to implement synchronization points in applications. It is guaranteed that no processor will proceed past the call to exsync until all are ready to do so. Furthermore the processors emerge from the exsync calls on their respective nodes as synchronized as can be arranged.

A call to exsync in one processor must be complemented by a call to exsync in all other processors.

EXAMPLE

In the following code we assume that it is important that all processors be synchronized between two phases of an algorithm.

```
#include "express.h"

main()
{
/* PHASE 1 of application ..... */

             .....

/* Before beginning second phase make sure all processors
in sync */

     exsync();/* Block till called in all nodes */

/* PHASE 2 of application - all processors synchronized */

             .....
```

Another good place for this function is after installing message handlers with the exhandle system call. Synchronizing all processors is a good idea since it prevents any one processor sending a message to another which has yet to install its signal handler.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXSYNC()
```

DIAGNOSTICS

If an unrecoverable hardware error is detected `exsync` returns -1.

SEE ALSO

`exchange`

NAME

`extest` - Test for an incoming message, non-blocking

SYNOPSIS

```
#include "express.h"

int extest(psrc, ptype)
int *psrc, *ptype;
```

DOMAIN

`extest` is available to both host and node programs. The calling sequence is identical in both cases.

DESCRIPTION

This function looks for an incoming message in a non-blocking fashion. It is intended for use in implementing strategies which require non-blocking read capabilities. The pointer arguments `psrc` and `ptype` are interpreted just as in the `exread` call with the same wildcard interpretations.

The useful feature of the "test" function is that it returns immediately indicating by the return value whether or not a message currently exists which matches the supplied parameters. If no such message is found -1 is returned. Otherwise the return value is the length of the matching message.

EXAMPLES

In the following examples we consider a case in which the following four messages have arrived on our node in the order given.

```
1. Source 1        Type 12        Length 32
2. Source HOST     Type 2         Length 512
3. Source 1        Type 15        Length 1024
4. Source 2        Type 0         Length 0
```

The simplest case is where both source and type are explicitly stated as in the call

```
source = 1;
type = 15;
stat = extest(&source, &type);
```

In this case message three will be accepted. The returned value, `stat` will be 1024, the length of the acceptable message.

In the next case we give wildcard values to both arguments

```
stat = extest((char *)0, (char *)0);
```

This will result in the acceptance of the first message. The returned value, `stat` will be 32.

The next example uses the wildcard value, `DONTCARE`, to look for any message but retain information about its source.

```
source = DONTCARE;
type = 0;
stat = extest(&source, &type);
```

In this case the type is explicitly given and so message 4 will be matched. The returned value will be 0, the length of the message and the `source` variable will contain 2, the source of the message.

In the last example a wildcard value is given for the type field

```
source = 1;
type = DONTCARE;
stat = extest(&source, &type);
```

In this case the source is given explicitly and the type allowed to take any value. With the parameters shown message 1 will be accepted and the value 12 stored in the `type` variable. The value 32 will be returned. Note that types are subject to extra processing through the `extype` commands. If we had specifically excluded type 12 from consideration then message 3 would have been used instead since it has the correct source and has not been excluded. If we had excluded both types 12 and 15 then the call to `extest` would return -1 to indicate that no suitable message had yet arrived.

RETURN VALUE

The return value is the length of the matching message or -1 if no message can be found which fits the indicated parameters.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXTEST(SOURCE, TYPE)
INTEGER SOURCE, TYPE
```

SEE ALSO

`exopen`, `exread`, `extype`.

NAME

`extime, extick` - Time measurement

SYNOPSIS

```
#include "express.h"

long extime()

int extick()
```

DOMAIN

These functions are available to all node programs.

DESCRIPTION

`extime` returns the number of microseconds since a fixed reference point.

`extick` returns the number of hardware clock ticks since a fixed reference point.

Both routines are intended to be used for timing measurements. `extime` provides measurements in convenient units but suffers from the fact that its accuracy may depend on some "unknown" constant such as the hardware's clock speed. It may further require significantly longer than `extick` to return a result since one or more arithmetic operations will normally be required to convert the machine clock ticks to microseconds.

Note that the availability of a routine which returns time in microseconds should not be taken to imply the existence of hardware with this resolution. In most cases the hardware timers will have intervals of many microseconds.

FORTRAN SYNOPSIS

```
INTEGER*4 FUNCTION KXTIME()

INTEGER FUNCTION KXTICK()
```

NAME

exinctype, exexctype - Include or exclude certain message types in interpreting wildcards.

SYNOPSIS

```
#include "express.h"

int exinctype(lotype, hitype)
int lotype, hitype;

int exexctype(lotype, hitype)
int lotype, hitype;
```

DOMAIN

exinctype and exexctype are available in both host and node processors.

DESCRIPTION

These routines are used to modify the behavior of the "DONTCARE" wildcard value used in the ptype field of the calls exread, extest, etc. In particular the user can specify that certain types be excluded or included among those that match the "any type" condition.

exexctype specifies a low and high type value defining an (inclusive) range of types which should not be considered when processing the wildcard value. All the other types will remain acceptable.

exinctype specifies the low and high end of an (inclusive) range of types which can be accepted by the program. All other types of messages will be excluded.

These routines are of most use when two or more processes share the same processor group with the exshare call or when message handlers are being used (*cf.* exhandle). In this case the use of wildcards is dangerous, without previously calling these routines, since otherwise the recipient of any given message is unpredictable. Using these routines it is possible to allow one process access to only a restricted range of types while the other process can safely use all the other types and BOTH may still be permitted the use of wildcards.

EXAMPLES

In the following code we limit attention to types in the range 123 thru 125.

```
#include <stdio.h>
#include "express.h"

main()
{
     int msg_src, msg_type;
```

```
/* Code to allocate nodes and load programs */

            ...

/* Restrict attention to only a small range of message
   types */

     exinctype(123, 125);

/* Now read messages with these types */

     msg_src = DONTCARE;
     msg_type = DONTCARE;
     exread(buffer, 128, &msg_src, &msg_type);
```

After including only the specific types the wildcard values may be used freely but with their meanings restricted to a smaller range. In the above example the call to `exread` will only ever read messages whose types lie in the range 123-125.

As mentioned above this technique is most useful when two or more processes wish to share access to a particular set of nodes. If the above call had been made in one process then the other one might wish to make a call such as

```
exexctype(99, 125);
```

in which we explicitly delete the message type range 99-125 from consideration. (This would be useful if yet another process were sharing the same nodes and using types 99-100.) All other message types will remain valid.

The include/exclude mechanism can be turned off by supplying two `DONTCARE` arguments to the appropriate function.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXINCT(LOTYPE, HITYPE)
INTEGER LOTYPE, HITYPE

INTEGER FUNCTION KXEXCT(LOTYPE, HITYPE)
INTEGER LOTYPE, HITYPE
```

SEE ALSO

`exread`, `extest`, `exshare`

NAME

exvread, exvwrite - Vector read/write functions

SYNOPSIS

```
int exvread(buffer, size, offset, items, psrc, ptype)
char *buffer;
int size, offset, items, *psrc, *ptype;

int exvwrite(buffer, size, offset, items, pdest, ptype)
char *buffer;
int size, offset, items, *pdest, *ptype;
```

DOMAIN

exvread and exvwrite may be called in both host and node processors.

DESCRIPTION

These routines implement direct read and write functions. Additionally they allow non-contiguous blocks of data to be transmitted as a single message.

These functions correspond directly to exread and exwrite except in the interpretation of the actual bytes to be transmitted. In the exread function a single block of contiguous data is transmitted while the exvread functions allow messages to be built up from non-contiguous memory blocks.

The manner in which the blocks are specified to exvwrite is as follows: items objects, each of size bytes are taken starting from buffer. In addition each block is separated from the next by offset bytes.

The specification is similar for exvread except that objects are read into distinct memory blocks separated by offset bytes.

In all other regards the arguments to exvread and exvwrite perform just as they would in exread and exwrite.

EXAMPLE

The most useful application of these functions is to deal with multi-dimensional arrays in which we are required to pass data across a dimension in which the array data is not contiguous. (In C the last array dimension is the one that indexes contiguous memory locations while it is the first in Fortran). Consider an example in which we have a 10 x 10 array of values in each node corresponding to a two-dimensional image. The first dimension refers to the horizontal axis while the second refers to the vertical. (array[1][1] is thus near the bottom left-hand corner, for example). If we now consider a simple scrolling operation in which data is to be moved from left to right then we see that the data lies correctly (in C) and a suitable call to exread, for example, would be

```
exread(array, 10*sizeof(int), &leftnode, &type);
```

assuming that leftnode had been correctly assigned and a suitable call to exwrite had

been made in some processor. If the scroll were to be in the vertical direction, however, then exread is not appropriate; the operation can be coded as

```
exvread(array, sizeof(int), 10*sizeof(int), 10,
        &downnode, &type);
```

which specifies that each array element has the size of an int and that the total distance between elements array[i][j] and array[i+1][j] is 10 times the size of an individual element. Finally ten items should be transmitted. Notice that we can also use a call to exvread for the horizontal shift by merely changing the offset field in the above call from 10*sizeof(int) to sizeof(int). This allows the code to have a uniform structure for both axes.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION
          KXVREA(BUFFER, SIZE, OFFSET, ITEMS, SRC, TYPE)
INTEGER BUFFER(*), SIZE, OFFSET, ITEMS, SRC, TYPE

INTEGER FUNCTION
          KXVWRI(BUFFER, SIZE, OFFSET, ITEMS, DEST,TYPE)
INTEGER BUFFER(*), SIZE, OFFSET, ITEMS, DEST, TYPE
```

DIAGNOSTICS

If any error occurs in exvread or exvwrite -1 is returned. Possible sources of error are: an illegal source or destination, an illegal buffer or a preposterous value of size, offset or items. If no error occurs exvread returns the number of items read and exvwrite the number written. Note that it is an error to attempt to send a message of type DONTCARE with the exvwrite function.

SEE ALSO

exread, exwrite, exchange

NAME

exwrite - Write a message

SYNOPSIS

```
#include "express.h"

int exwrite(buf, length, pdest, ptype)
int length, *pdest, *ptype;
char *buf;
```

DOMAIN

exwrite is available to both host and node programs with identical calling sequences.

DESCRIPTION

This routine sends a message to the processor indicated by the pdest argument. The message will consist of length bytes taken from the supplied buf pointer. The message has the type specified by the ptype parameter. Both ptype and pdest are supplied as pointers for compatibility with the exread functions.

The special value DONTCARE is not allowed as either destination or type arguments to this function.

The special value HOST may be used to give the host processor as destination.

RETURN VALUE

exwrite returns the number of bytes written, or -1 upon unrecoverable errors.

EXAMPLES

The following code is used to send 15 bytes taken from the address my_buf to processor 12. The message will have type 99.

```
main()
{
     int dest = 12;
     int type = 99;

     exwrite(my_buf, 15, &dest, &type);

         ...
```

The next code sends a 128 byte message to the host processor. The message type will be 10.

```
#include "express.h"/* Defines HOST */
```

```
main()
{
     int dest = HOST;
     int type = 10;

     exwrite(data, 128, &dest, &type);

         ...
```

FORTRAN SYNOPSIS

```
SUBROUTINE KXWRIT(BUFFER, NBYTES, DEST, TYPE)
INTEGER BUFFER(*), NBYTES, DEST, TYPE
```

WARNINGS

Certain message types are restricted to the *Express* kernel. User message types must be less than 16384. It is illegal to transmit a message of type DONTCARE or to attempt to send a message to processor DONTCARE.

SEE ALSO

exopen, exread.

NAME

`fmulti, fsingl, ismulti, forder` - Parallel I/O characteristics of files.

SYNOPSIS

```
fmulti(fp)
FILE *fp;

fsingl(fp)
FILE *fp;

fasync(fp)
FILE *fp;

int ismulti(fp)
FILE *fp;

int isasync(fp)
FILE *fp;

forder(fp, order)
FILE *fp;
int order;
```

DOMAIN

These routines may only be called in programs compiled with the *Cubix* or *Plotix* libraries.

DESCRIPTION

These routines provide an interface to the parallel features of buffered file I/O - i.e., `printf`, `scanf`, `fopen` etc. As well as their usual characteristics *Cubix* "`FILE`"s are either in `singular`, `multiple` or `asynchronous` mode. This mode determines the exact behavior of read and write (or equivalently `printf` and `scanf`) calls on that file with regard to the distribution of data.

If a file is in singular mode then any read operation on it must be made loosely synchronously and exactly the same data is transferred to each node. Similarly, write operations must be made loosely synchronously and only node zero actually transmits any data to the file. This has the effect of allowing all nodes to apparently write but only one copy appears in the output file. In this mode it is an error if any node attempts to read or write different data from the others. This error normally causes internode communication to "hang" or `abort` with error status -1.

In multiple mode read requests are satisfied from the file independently. Thus each node can read its own data. Output requests can also be made independently with each node writing its own data to the file. Note that in multiple mode no implicit flushing of buffers is performed and it is the responsibility of the user to call `fflush` in order to cause data to appear in the indicated file. In connection with this point the routine `setvbuf` is

available to alter the size and behavior of the internal I/O buffers. This allows an application to perform more output between calls to `fflush`.

In asynchronous mode, I/O requests are handled independently on the processors on which they occur. No interprocessor synchronization is performed. Each processor maintains its own state variables recording the last byte it read or wrote in the file, and each request to read or write implicitly returns the file to that location before proceeding.

The routines `fmulti`, `fsingl` and `fasync` switch files between multiple, singular and asynchronous modes. `fmulti` puts a file into the multiple mode and `fsingl` restores a file to singular mode. `fasync` places a file in asynchronous mode. All three flush any data in the file's buffers prior to the call, and all must be made loosely synchronously in all nodes.

By default both input and output operations on "`multi`" mode files occur in order of increasing processor number - i.e., node 0 gets the first crack followed by node 1, node 2 and so on. The `forder` system call is available to alter this default. The first argument indicates the stream for which a new ordering is desired and the second is an integer in the range `0... nprocs-1`. Further "`multi`" mode operations on this file will result in the processor which specified `order=0` being first, followed by that which gave `order=1` etc. From this it should be obvious that the `order` parameters given in the call to `forder` must form a permutation of the set `{0,..., nprocs-1}` - i.e., each value must be specified exactly once in one of the nodes. Failure to observe this rule results in deadlock whenever I/O is attempted on the affected stream. (Examples of the use of this parameter in the lower-level `mread` and `mwrite` system calls can be found on the respective manual pages).

The routine `ismulti` returns 1 if its argument is in multiple mode and zero otherwise.

The routine `isasync` returns 1 if its argument is in asynchronous mode, and zero otherwise.

EXAMPLES

The following code segment demonstrates the effect of the fmulti call

```
#include <stdio.h>

main()
{
     printf("Hello world\n");
     fmulti(stdout);
     printf("This is one of the processors ...\n");
     fsingl(stdout);
     printf(" .. that's all for now folks !!\n");
     exit(0);
}
```

When executed on four processors this would produce the output

```
Hello world
This is one of the processors ...
This is one of the processors ...
This is one of the processors ...
This is one of the processors ...
.. that's all for now folks !!
```

showing that only one line of output results from each call to `printf` in single mode while each processor generates it own output while the file is in multi-mode.

Asynchronous mode typically arises in one of two situations. Either a code is truly asynchronous - it's behavior is too unpredictable in advance to allow use of the synchronous I/O modes or one might want to use this mode for reporting runtime errors that may only occur within a single node. A particularly good example is that of heap storage - when `malloc` fails to return more memory it may be appropriate to write out some diagnostic comment and yet it cannot be guaranteed that all nodes will fail at the same time, or even that all nodes will have to call `malloc` at the same time. The following code is an example of this situation

```
fasync(stderr);

if((ptr=malloc(8192)) == (char *)0) {
    fprintf(stderr, "Ran out of memory!!\n");
    abort(123);
}
```

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KMULTI(UNIT)
INTEGER UNIT

INTEGER FUNCTION KSINGL(UNIT)
INTEGER UNIT

INTEGER FUNCTION KASYNC(UNIT)
INTEGER UNIT

INTEGER FUNCTION ISASYN(UNIT)
INTEGER UNIT

INTEGER FUNCTION ISMULT(UNIT)
INTEGER UNIT

INTEGER FUNCTION KORDER(UNIT, ORDER)
INTEGER UNIT, ORDER
```

SEE ALSO

`fasync`, `forder`, `mread`, `mwrite`.

NAME

`fopen` - Open files for buffered I/O.

SYNOPSIS

```
#include <stdio.h>

FILE *fopen(name, mode)
char *name, *mode;
```

DOMAIN

This routine may only be called in programs compiled with the *Cubix* or *Plotix* libraries.

DESCRIPTION

This function serves the identical purpose to the standard C runtime library routine of the same name - to open a file for buffered I/O through such system calls as `fprintf`, `fscanf`, `fseek`, etc.

The normally acceptable values of the "`mode`" have their normal meanings with the addition that the "`t`" and "`b`" flags for "text" and "binary" files are understood on all systems irrespective of whether or not they are actually implemented on the host.

Calls to this routine must normally satisfy the "loosely synchronous" constraint in that all nodes must attempt to open the same file with the same "`mode`" argument. The exception to this rule is if the character '`A`' appears in the "`mode`" string. In this case every invoking node will attempt to open the file independent of all others. In this case not all nodes need make the system call and the arguments can be different in each calling node. This allows individual nodes to open different, or indeed no files. Note that the file, once open, is still in multi-mode even if it has been opened by only a single processor. To make effective use a call to "`fasync`" is required to switch to "multi"-mode.

EXAMPLE

The following code allows each node to open a file whose name contains its processor number.

```
#include <stdio.h>
#include <express.h>
struct nodenv nodedata;

main()
{
     FILE *fp;
     char name[80];
/*
 * First get our processor number, and make sure that errors
 * can be reported asynchronously.
 */
```

```
        fasync(stderr);
        exparam(&nodedata);
/*
 * Create the filename to use.
 */
        sprintf(name, "node%d", nodedata.procnum);
        if((fp=fopen(name, "wA")) == (FILE *)0) {
            fprintf(stderr,"Node %d fails to open file %s\n",
                nodedata.procnum, name);
            abort(nodedata.procnum);
        }
        fprintf(fp, "This is node %d\n", nodedata.procnum);
        printf("Done\n");
        exit(0);
}
```

Note how we switch the diagnostic stream `stderr` to "async" mode before attempting to open the files. This allows individual nodes to report errors if their file cannot be opened. If `stderr` were left in "single" mode and only a few of the nodes failed then the program would abort, violating the loosely synchronous constraint, rather than because of the failure of "`fopen`"!

After executing this program on `N` processors there should be a separate output file for each node such that the file with name "`node#`" will contain the text

```
This is node #
```

SEE ALSO

`aopen`, `fmulti`, `syncmode`.

NAME

gethost - Determine host specific characteristics

SYNOPSIS

```
void gethost(node, buffer, buflen)
int node;
char *buffer;
int buflen;
```

DOMAIN

This routine may only be called in programs compiled with the *Cubix* or *Plotix* libraries.

DESCRIPTION

This routine is used to determine host-specific characteristics. The current implementation is restricted to returning, in `buffer`, the name of the operating system running on the host whose node identified is `node`. Up to `buflen` characters of this information will be transferred to the indicated buffer, any extra will be discarded.

No attempt is made to differentiate between minor versions of operating systems, or between the various "unix-like" machines.

EXAMPLE

In the following code we determine the type of operating system running on our standard host in order to find the character used to separate components of filenames.

```
#include "express.h"/* Defines HOST */

char get_separator()
{
     char osbuf[32];

     gethost(HOST, osbuf, sizeof(osbuf));
     if(!strcmp(osbuf, "unix")) return '/';
     else if(!strcmp(osbuf, "dos")) return '\\';
     else if(!strcmp(osbuf, "macintosh")) return ':';
     else if(!strcmp(osbuf, "vms")) return '.';
     else {
         fprintf(stderr,"Don't recognize OS %s\n", osbuf);
         return '\0';
     }
}
```

FORTRAN SYNOPSIS

```
SUBROUTINE KGETHO(NODE, OSBUF)
```

```
INTEGER NODE
CHARACTER*80 OSBUF
```

SEE ALSO

syncmode

NAME

`gin, agin` - Graphical input operations

SYNOPSIS

```
int gin(pbutton, px, py)
int *pbutton;
double *px, *py;

int agin(pbutton, px, py)
int *pbutton;
double *px, *py;
```

DOMAIN

These routines may only be called in programs compiled with the *Plotix* libraries.

DESCRIPTION

These routines are used to perform graphical input operations usually termed "locator input". Upon execution a cursor appears on the screen and is positioned and triggered in a device specific manner. After triggering the `gin` calls return and the specific trigger and position are returned under the pointers supplied as arguments.

The `gin` routine must be called in each processor simultaneously while the `agin` function may be executed by any processor at any time. In this latter case it is the responsibility of the user to ensure that sufficient information is present to allow the operator to know which processor is requesting input. Further, no flushing is performed by these functions. It is up to the user to ensure that the display surface actually contains up-to-date data before requesting graphical input.

The coordinates returned to the user are expressed relative to those set up by the last call to `space` in each processor. Further a status value is returned to indicate the result of the `gin` operation. A negative value is returned by devices which are not capable of performing input. A zero return value implies that the `gin` operation completed successfully but that the cursor position was outside the window selected by the most recent call to `vport` in this processor. A positive return means that the coordinates selected lay within the processor window. This last mechanism can be used to select processors with a mouse, for example.

EXAMPLE

In the following we assign different halves of the display to two processors; node 0 gets the left half and node 1 the right. We then use the input routines to select one or the other node for some processing task.

```
#include "express.h"/* Defines nodenv structure */

do_gin()
{
```

```
    struct nodenv nodedata;
    double x,y;
    int stat, key;

/* Divide up the screen on the basis of processor number */

    exparam(&nodedata);    /* Get runtime parameters */

    if(nodedata.procnum == 0) vport(0., 0., .5, 1.);
    else vport(.5, 0., 1., 1.);

/* Now assign coordinates. Each processor's window will be
 * mapped individually to the unit square
 */
    space(0.,0.,1.,1.);

    do {
        if((stat = gin(&key, &x, &y)) < 0) break;
        if(stat != 0) grind_away(x, y);
    } while(key != 0);
```

Having set up the windows and coordinate systems we loop until the `key` parameter is returned as zero and the processor whose region we indicated with the mouse calls the `grind_away` function with the selected points as arguments. Note that we can perform similar operations on more processors by using the `exgrid` routines to set up and coordinate the distribution of processors to screen areas.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KGIN(KEY, X, Y)
INTEGER KEY
REAL*4 X, Y

INTEGER FUNCTION KAGIN(KEY, X, Y)
INTEGER KEY
REAL*4 X, Y
```

SEE ALSO

vport

NAME

`greyscale` - Change color attributes.

SYNOPSIS

```
greyscale(from to)
int from, to;
```

DOMAIN

This routine may only be called in programs compiled with the *Plotix* libraries.

DESCRIPTION

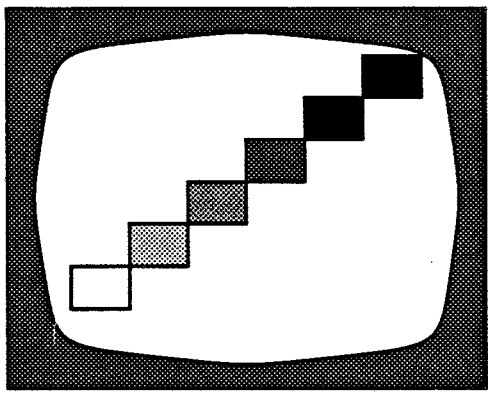
This routine changes the association of color indices to device colors used by *Plotix*. By default a limited color map is used which can be extended with the `greyscale` and `rainbow` function calls.

`greyscale` extends the *Plotix* color map by adding a smoothly varying grey-scale between the two selected values. The lower value will be white and the upper black. The number of distinct grey levels available is hardware dependent but in any case *Plotix* will map the indicated range in as smooth a manner as possible.

EXAMPLE

The following code draws a set of 6 boxes of varying grey shades along the diagonal of the screen image.

```
do_boxes()
{
     int i;
     double v;

     space(0., 0., 6., 6.);

     greyscale(10, 15);

     for(i=0; i<6; i++) {
         v = i;
         box(v, v, v+1., v+1., 10+i, 0);
     }
     sendplot();
}
```



FORTRAN SYNOPSIS

```
SUBROUTINE KGREYS(LO, HI)
INTEGER LO, HI
```

SEE ALSO

color, rainbow

NAME

`label` - Add text.

SYNOPSIS

```
label(text, x, y)
char *text;
double x, y;
```

DOMAIN

This routine may only be called in programs compiled with the *Plotix* libraries.

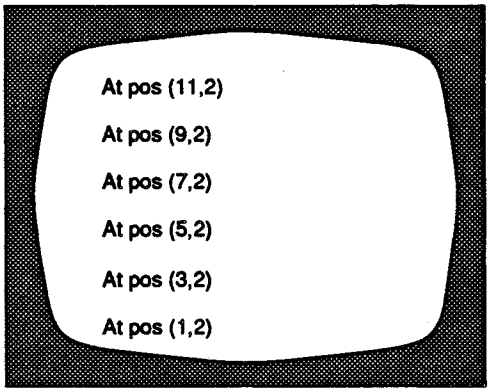
DESCRIPTION

This routine draws the characters contained in the `text` string at the position (x,y). The first character of the string is placed above and to the right of the indicated point. Other methods of justification can be obtained with the `dotext` function.

EXAMPLE

The following code defines a `12x12` coordinate system and writes a string at several positions on the screen.

```
do_labels()
{
     int i;
     char lab[32];

     space(0.,0.,12., 12.); /* Define coordinate system */

     for(i=1; i<12; i+= 2) {
         sprintf(lab, "At pos (%d, 2)", i);
         label(lab, 2.0, (double)i);
     }
     sendplot();
}
```



FORTRAN SYNOPSIS

```
SUBROUTINE KLABEL(TEXT, X, Y)
CHARACTER*80 TEXT
REAL*4 X, Y
```

WARNING

The current plotting position is undefined after this call. In order to perform reliable graphical operations `move` should be used before any further drawing is performed.

SEE ALSO

`dotext`, `marker`.

NAME

`linemod` - Modify drawing style for lines

SYNOPSIS

```
linemod(index)
int index;
```

DOMAIN

This routine may only be called in programs compiled with the *Plotix* libraries.

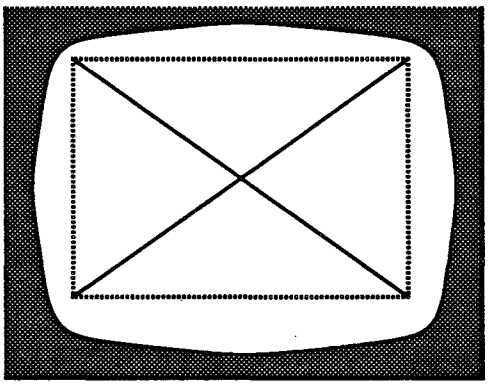
DESCRIPTION

Modifies the style in which all further lines are drawn. The `index` argument is an integer which specifies, in a device dependent manner, the actual linestyle to use. The value 0 will always create solid lines.

EXAMPLE

The following code defines a 10 x 10 coordinate system and draws a box with a dashed edge and a solid diagonal.

```
space(0.,0.,10., 10.);/* Define coordinate system */

linemod(1);
move(1.,1.);
cont(9.,1.);
cont(9.,9.);
cont(1.,9.);
cont(1.,1.);

linemod(0);

cont(9.,9.);
move(1.,9.);
cont(9.,1.);

sendplot();
```



FORTRAN SYNOPSIS

```
SUBROUTINE KLINEM(INDEX)
INTEGER INDEX
```

SEE ALSO

cont, color.

NAME

`malloc_debug` - Monitor behavior of memory allocator

SYNOPSIS

```
malloc_debug(level)
int level;

malloc_verify()

malloc_print(fp)
FILE *fp;

long malloc_avail()
```

DOMAIN

This routine may only be called in programs compiled with the *Cubix* or *Plotix* libraries.

DESCRIPTION

These functions are supplied to aid in diagnosing and finding errors associated with the standard memory allocation functions `malloc`, `free` etc. A common problem for C programmers occurs when a pointer not previously obtained from `malloc` is freed or when data in the internal structures controlled by the memory allocator is corrupted.

The function `malloc_debug` controls debugging within the memory allocation software by specification of the `level` argument interpreted in the following manner.

`level = 0`	No runtime debugging performed (The default)
`level = 1`	Abort if any problem is detected within the internal data structures. Only a minimal number of checks are performed and the impact on performance is slight.
`level = 2`	Check all internal data structures whenever one of the memory allocation routines is called. This option can be extremely slow.
`level = 3`	Same style of checking as in option 1 but additionally abort if a memory allocation routine would return the `NULL` pointer.
`level = 4`	Same style of checking as in option 2 but additionally abort if a memory allocation routine would return the `NULL` pointer.

The last two options are especially useful to application programmers who do not always check return codes from the memory management functions. In these cases an unexpected `NULL` return may cause grave effects in a program and the ability to have the system abort rather than corrupt data may be important.

The function `malloc_verify` may be called at any time to perform a single check of the memory allocator's internal data structures. It returns zero if all is well and non-zero if there is an inconsistency.

The function `malloc_print` prints a rather extensive map of the memory allocated by malloc on the stream pointed to by its argument.

The function `malloc_avail` returns the number of bytes as yet unallocated in memory. It does not account in any way for memory fragmentation, so it may not be possible to acquire all this memory in a single chunk.

EXAMPLE

The insertion of the call

```
malloc_debug(3);
```

at the beginning of a buggy program may be a life-saver. As well as looking for memory corruptions an abort will also occur if non memory is available for a particular operation. In serious circumstances increasing the `level` to 4 may also be justified although this may slow program execution significantly if a lot of memory allocation/freeing is being performed.

NAME

`marker` - Draw marker symbol.

SYNOPSIS

```
marker(symbol, x, y, size)
int symbol;
double x, y, size;
```

DOMAIN

This routine may only be called in programs compiled with the *Plotix* libraries.

DESCRIPTION

This routine draws a marker symbol at the position (x,y) expressed relative to the coordinate system most recently defined with the `space` function. The marker is drawn with the given `size`, expressed in the same units as the coordinates. The `symbol` argument is used to distinguish the various markers as follows

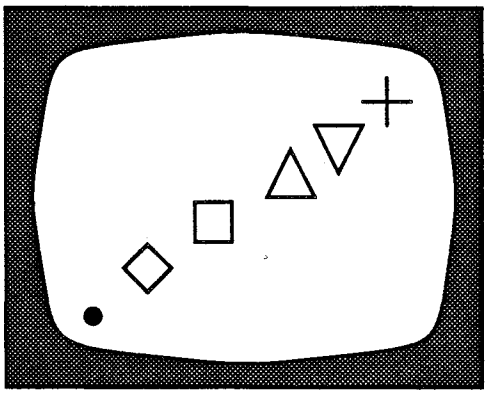
0	point
1	diamond
2	square
3	triangle
4	inverted triangle
5	cross
6	plus
7	star

Some attempt is made to compensate for the fact that "squares" look bigger than "triangles" - the `size` argument is not strictly interpreted as the height of the triangle, for example.

EXAMPLE

The following code defines an 9 x 9 coordinate system and draws different marker symbols along the diagonal.

```
do_markers()
{
     int i;

     space(0.,0.,8., 8.);   /* Define coordinate system */
     for(i=0; i<6; i++) {
          marker(i, (double)(i+1), (double)(i+1), .5);
     }
     sendplot();
}
```



FORTRAN SYNOPSIS

```
SUBROUTINE KMARKE(INDEX, X, Y, SIZE)
INTEGER INDEX
REAL*4 X, Y, SIZE
```

SEE ALSO

label, dotext.

NAME

move - Move without drawing.

SYNOPSIS

```
move(x, y)
double x, y;
```

DOMAIN

This routine may only be called in programs compiled with the *Plotix* libraries.

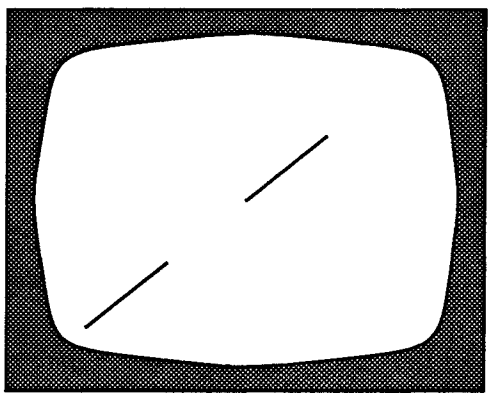
DESCRIPTION

Moves the current plotting position to (x,y). Nothing is drawn on the display surface. x and y are specified relative to the coordinate system defined by the most recent call to space.

EXAMPLE

The following code draws a broken diagonal line across the display surface.

```
space(0.,0.,4., 4.);/* Define coordinate system */

move(0.,0.);
cont(1.,1.);
move(2.,2.);
cont(3.,3.);
sendplot();
```



FORTRAN SYNOPSIS

```
SUBROUTINE KMOVE(X, Y)
```

```
REAL*4 X, Y
```

SEE ALSO

cont, color, linemod.

NAME

mread - Read independent data into each node.

SYNOPSIS

```
int mread(fd, buf, length, order)
int fd, length, order;
char *buf;
```

DOMAIN

This routine may only be called in programs compiled with the *Cubix* or *Plotix* libraries.

DESCRIPTION

mread reads data into the nodes from the file indicated by the ***file descriptor*** fd which should have been obtained by an earlier call to open. Independent data is read into each node; the length arguments need not all be the same.

The call to mread must be made loosely synchronously in all nodes.

The order argument determines in what order the data from the input file are to be placed in the nodes. The simplest case, obtained by setting order = PROCNUM_ORDER, is for the input to appear in order of increasing processor number so that node 0 receives the first block followed by node 1 and so on. Other cases are obtained by setting the value to be an integer between 0 and the number of processors. The node which specified order = 0 receives the first block and then the node which gave order = 1 and so on. Note that it is an error if a value between 0 and the number of processors is not specified in some node. This condition is indicated by mread returning -1 and setting the external variable errno to -1.

EXAMPLE

Suppose that we have decomposed our domain into a two dimensional mesh with NX and NY processors in the two dimensions. If we now want to read data blocks in the conventional manner for such a grid - i.e., along the rows, then the exgrid routines of *Express* can be used as follows

```
#include "express.h"

do_read(NX, NY)
int NX, NY;
{
     int ndim[2], coord[2];
     struct nodenv nodedata;

     exparam(&nodedata);    /* Get runtime parameters */

     ndim[0] = NX;
     ndim[1] = NY;
```

```
exgridinit(2, ndim);

exgridcoord(nodedata.procnum, coord);

mread(fd, buffer, length, coord[1]*NX + coord[0]);
```

This will order the input according to the row and column coordinates of the processor in the two dimensional mesh.

WARNING

Reading and writing binary files is complicated by the fact that the host and nodes of the parallel processing system may not have the same type of processor (CPU) and may not share the same byte ordering properties. An example might be a Sun workstation hosting a transputer or NCUBE machine. In this case the host processor is a Motorola based system which has the most significant byte at the lowest memory address while the nodes have the opposite ordering. To cover these cases *Express* provides a set of byte swapping primitives: `exswap`.

RETURN VALUE

`mread` returns the number of bytes read, or -1 upon unrecoverable errors. In the latter case the external variable `errno` is set to signify the exact error which occurred. In the special case where the error is due to incorrect specification of the `order` parameter `errno` will be set to -1. A return value of zero indicates an "end of file" condition.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KMREAD(UNIT, BUFFER, NBYTES, ORDER)
INTEGER UNIT, BUFFER(*), NBYTES, ORDER
```

SEE ALSO

`mwrite`, `mread2d`, `mwrite2d`, `exswap`.

NAME

`mread2d, mwrite2d` - Read/write two dimensional data sets.

SYNOPSIS

```
int mread2d(fd, buf, tot_cols, tot_rows, item_size,
                     col0, col1, row0, row1, skip_dist)
int fd, tot_cols, tot_rows, item_size;
int col0, col1, row0, row1;
int skip_dist;
char *buf;

int mwrite2d(fd, buf, tot_cols, tot_rows, item_size,
                    col0, col1, row0, row1, skip_dist)
int fd, tot_cols, tot_rows, item_size;
int col0, col1, row0, row1;
int skip_dist;
char *buf;
```

DOMAIN

This routine may only be called in programs compiled with the *Cubix* or *Plotix* libraries.

DESCRIPTION

These functions provide a primitive interface to a two-dimensional file access mode for *Cubix* programs. The basic idea is that data sets decomposed over a two dimensional array of processors can be read and written with a single function call.

Both functions operate on a file descriptor, `fd`, such as might be obtained through a call to `open`. Data is either read to or written from the array pointed to by `buf` and consists of some number of "items" each of size `item_size`. This concept is used instead of the more obvious "byte" notation so that the other arguments to these functions may be assigned as row or column indices.

The disk data set is treated as an array of `tot_rows` by `tot_cols` items of which a subset is to be read or written by each node. The particular piece of the global data set required by a given node is specified by the `row0`, `row1`, `col0` and `col1` arguments which are ***inclusive*** parameters indexed from zero - the specification

```
col0 = 0
col1 = 9
row0 = 0
row1 = 9
```

would access the 10 x 10 block in the upper left hand corner of the array.

The `skip_dist` parameter specifies the offset in the `buf` array between successive "row" entries again in "items". This can be used to leave a boundary strip around the edge of the data as is common in two dimensional decompositions and is illustrated in the example below.

EXAMPLE

Suppose we have a two dimensional array of integers of size NX by NY which we wish to decompose over the processors. The following code can be used to setup the decomposition with the exgrid functions.

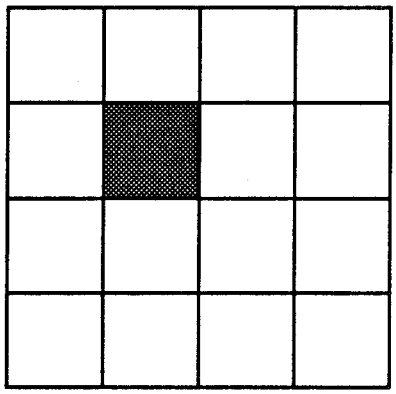
```
#include <express.h>

int ndim[2];
int gbl_size[2], lcl_size[2], lcl_start[2];
struct nodenv nodedata;

main()
{
/* Get runtime parameters, processor number etc.. */

     exparam(&nodedata);

/* Divide the nodes up among the two dimensions of the data
 * and initialize the exgrid system.
 */
     exgridsplit(cparm.nprocs, 2, ndim);
     exgridinit(2, ndim);

/* Figure out how much of the data fits in each node */

     gbl_size[0] = NX;
     gbl_size[1] = NY;
     exgridsize(cparm.procnum, gbl_size,lcl_size,lcl_start);
```

Notice how we use exgridsplit to evenly divide up the number of processors between the data dimensions and exgridsize to divide up the array between the processors. The parameters returned by exgridsize can be used to read in a two-dimensional data set as follows. (We assume that fd is a file descriptor corresponding to some previously opened file and that the global variables defined in the previous program fragment are still available.

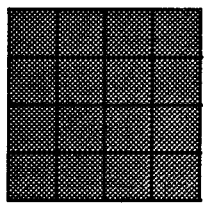
```
/*
 * Read data into nodes, no overlap allowed.
 */
read_data(fd)
int fd;
{
     mread2d(fd, data, NX, NY, sizeof(data[0]),
         lcl_start[0], lcl_start[0]+lcl_size[0]-1,
```

```
        lcl_start[1], lcl_start[1]+lcl_size[1]-1,
        lcl_size[0]);
    return;
}
```

This strategy uses the values returned by `exgridsize` to figure out exactly which data to request from the input data set. In this case each node gets a distinct piece of data, divided as evenly as possible between the processors but with no overlap and no space for any. The mapping is as shown in the following figure.



Processor decomposition



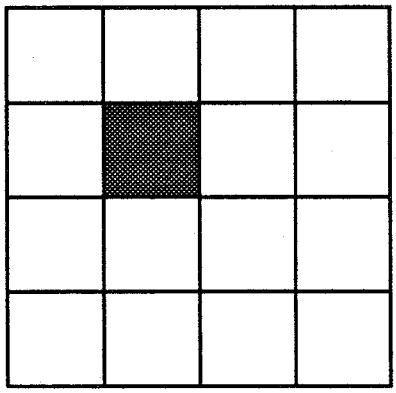
Data array

A common situation is that in which the input data set is required to be read into the center of a block which contains, around its edges, space for one or more entries from a neighbor node. This is a common situation in image processing, for example, where some local convolution is to be applied. To achieve this effect with the above parameters we change the call to `mread2d` as follows:

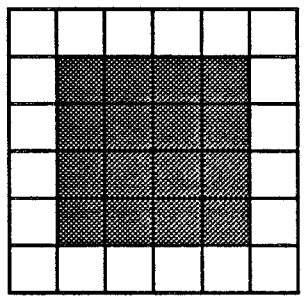
```
/*
 * Read data into nodes, overlap allowed but not performed.
 */
read_data(fd)
int fd;
{
    mread2d(fd, data+lc_size[0]+3, NX, NY, sizeof(data[0]),
        lcl_start[0], lcl_start[0]+lcl_size[0]-1,
        lcl_start[1], lcl_start[1]+lcl_size[1]-1,
        lcl_size[0]+2);
    return;
}
```

This call performs the mapping shown in the next figure. Note that the `skip_dist`

parameter has been modified to place a gap around each "row" of the data with one space at the beginning and one at the end. This would be suitable for a nearest neighbor interaction in which a single strip of data is required from each neighbor node.



Processor decomposition

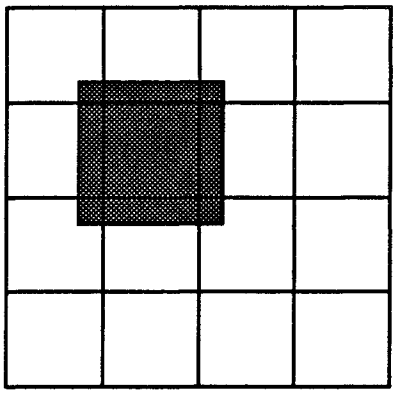


Data array

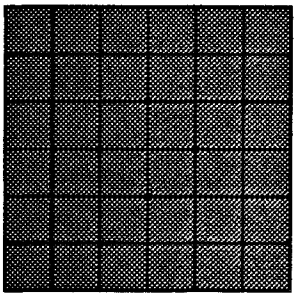
A last option which is interesting is one in which the data being read is overlapped at the time it is originally taken from the data set. This is merely a variation on the last call which provided space for the overlapped data but did not initialize it. The call required to read in overlapping data is as follows

```
/*
 * Read data into nodes with overlapping strip one "item"
 * wide.
 */
read_data(fd)
int fd;
{
     mread2d(fd, data, NX, NY, sizeof(data[0]),
         lcl_start[0]-1, lcl_start[0]+lcl_size[0],
         lcl_start[1]-1, lcl_start[1]+lcl_size[1],
         lcl_size[0]+2);
     return;
}
```

This mapping is shown in the next figure.



Processor decomposition



Data array

WARNING

Reading and writing binary files is complicated by the fact that the host and nodes of the parallel processing system may not have the same type of processor (CPU) and may not share the same byte ordering properties. An example might be a Sun workstation hosting a transputer or NCUBE machine. In this case the host processor is a Motorola based system which has the most significant byte at the lowest memory address while the nodes have the opposite ordering. To cover these cases *Express* provides a set of byte swapping primitives: `exswap`.

RETURN VALUE

`mread2d` returns the number of bytes read, or -1 upon unrecoverable errors. In the latter case the external variable `errno` is set to signify the exact error which occurred. Similarly `mwrite2d` returns the number of bytes written by the calling node or -1 upon disastrous errors.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KMRD2D(UNIT,BUFFER,TCOLS,TROWS,ISIZE,
                COL0, COL1, ROW0, ROW1, ISKIP)
INTEGER UNIT, BUFFER(*), TCOLS, TROWS, ISIZE
INTEGER COL0, COL1, ROW0, ROW1, ISKIP

INTEGER FUNCTION KMWT2D(UNIT,BUFFER,TCOLS,TROWS,ISIZE,
                COL0, COL1, ROW0, ROW1, ISKIP)
INTEGER UNIT, BUFFER(*), TCOLS, TROWS, ISIZE
INTEGER COL0, COL1, ROW0, ROW1, ISKIP
```

SEE ALSO

`mread, mwrite, exswap`

NAME

mwrite - Write independent data from each node.

SYNOPSIS

```
#include "express.h"

int mwrite(fd, buf, length, order)
int fd, length, order;
char *buf;
```

DOMAIN

This routine may only be called in programs compiled with the *Cubix* or *Plotix* libraries.

DESCRIPTION

mwrite writes data from the nodes to the file indicated by the ***file descriptor*** fd which should have been obtained by an earlier call to open. Independent data is written from each node; the length arguments need not all be the same.

The call to mwrite must be made loosely synchronously in all nodes.

The order argument determines in what order the data from the various nodes are to be placed in the output file. The simplest case, obtained by setting order = PROCNUM_ORDER, is for the output to appear in order of increasing processor number. Other cases are obtained by setting the value to be an integer between 0 and the number of processors. First in the output appears the data from the node which specified order = 0 then that from the node with order = 1 and so on. Note that it is an error if a value between 0 and the number of processors is not specified in some node. This condition is indicated by mwrite returning -1 and setting the external variable errno to -1.

EXAMPLE

Suppose that we have decomposed our domain into a two dimensional mesh with NX and NY processors in the two dimensions. If we now want to write out data blocks in the conventional manner for such a grid - i.e., along the rows then the exgrid(4) routines of *Express* can be used as follows

```
#include "express.h"

do_write(NX, NY)
int NX, NY;
{
     int ndim[2], coord[2];
     struct nodenv nodedata;

     exparam(&nodedata);    /* Get runtime parameters */

     ndim[0] = NX;
```

```
ndim[1] = NY;

exgridinit(2, ndim);

exgridcoord(nodedata.procnum, coord);

mwrite(fd, buffer, length, coord[1]*NX + coord[0]);
```

This will order the output according to the blocks in the two dimensional grid.

WARNING

Reading and writing binary files is complicated by the fact that the host and nodes of the parallel processing system may not have the same type of processor (CPU) and may not share the same byte ordering properties. An example might be a Sun workstation hosting a transputer or NCUBE machine. In this case the host processor is a Motorola based system which has the most significant byte at the lowest memory address while the nodes have the opposite ordering. To cover these cases *Express* provides a set of byte swapping primitives: `exswap`.

RETURN VALUE

`mwrite` returns the number of bytes written, or -1 upon unrecoverable errors. In the latter case the external variable `errno` is set to signify the exact error which occurred. In the special case where the error is due to incorrect specification of the `order` parameter `errno` will be set to -1.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KMWRIT(UNIT, BUFFER, NBYTES, ORDER)
INTEGER UNIT, BUFFER(*), NBYTES, ORDER
```

SEE ALSO

`mread, mread2d, exswap`

NAME

`openpl, aopenpl, closepl` - Begin and terminate graphics system.

SYNOPSIS

```
#include <stdio.h>
#include "express.h"

int openpl(bufsize, fp)
int bufsize;
FILE *fp;

int aopenpl(bufsize, fp)
int bufsize;
FILE *fp;

closepl()
```

DOMAIN

This routine may only be called in programs compiled with the *Plotix* libraries.

DESCRIPTION

These routines initialize and terminate the graphics system.

One of the `openpl` functions must be the first *Plotix* function called in any graphics application. The first argument denotes the size of the internal buffer to be allocated for storing graphical information between calls to the `sendplot` functions. The special value `DONTCARE` can be used to allocate a default sized buffer. As a guide to appropriate sizes a call to `move` or `cont` requires 5 bytes. The second argument describes the disposition of the graphical output. If a non-zero value is given then graphical data is written in metafile format to the indicated stream. The NULL value is used to perform on-line graphics - output is sent directly to the selected device.

`aopenpl` performs the same function as `openpl` but asynchronously - that is any node may call this routine independent of the others with no synchronization constraints.

`openpl` returns a status code indicating the success or failure of the setup procedures. Negative values indicate errors and it is unwise to proceed if an error condition exists since terminals, for example, may be sent into strange states.

The last graphical routine to be called by an application should be `closepl`. This serves to close any open files and reset interactive devices to their normal states.

Both `openpl` and `closepl` must be called loosely synchronously in all nodes, while `aopenpl` may be called independently at any time by any node.

EXAMPLE

The following skeleton code should provide the basis for all graphics applications

```
#include <stdio.h>
#include "express.h"    /* Defines DONTCARE */

main()
{
     if(openpl(DONTCARE, (FILE *)NULL) < 0) {
          fprintf(stderr,
             "Failed to initialize graphics system\n");
         exit(1);
     }

/* Application code .......... */

              ...

/* Application finished, clear up graphics system */

     closepl();
     exit(0);
}
```

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KOPENP(BUFFER, ISIZE)
INTEGER BUFFER(*), ISIZE

INTEGER FUNCTION KAOPEN(BUFFER, ISIZE)
INTEGER BUFFER(*), ISIZE

SUBROUTINE KCLOSP
```

FORTRAN DESCRIPTION

Since FORTRAN programs do not generally have access to the dynamic memory management function of C the calling sequence of the KOPENP and KAOPEN calls differs slightly from the C equivalents. The FORTRAN functions require that a buffer be passed directly to the system together with its length. No FILE argument is required

SEE ALSO

sendplot.

NAME

panel - Draw and fill polygons.

SYNOPSIS

```
initpanel(color, edge)
int color, edge;

panelpoint(x, y)
double x, y;

endpanel()

polgn(npts, xpts, ypts, color, edge)
int npts, color, edge;
double *xpts, *ypts;
```

DOMAIN

These routines may only be called in programs compiled with the *Plotix* libraries.

DESCRIPTION

These routines are used to draw and fill polygonal regions of the display surface. `polgn`, the most straightforward of the routines takes two arrays each containing `npts` values as the x and y coordinates of the vertices of the polygon to be drawn. The coordinates need not close - the first and last points are joined by the system. The resulting polygon will be filled according to the `color` argument and will have its outline drawn in the current color if `edge` is non-zero.

Positive values of `color` translate into solid colors in the same manner as the arguments to the line color primitive, `color`. Negative values yield device dependent shading patterns.

All coordinates are expressed relative to the most recent call to `space`.

An alternative interface to the polygon routines is provided by `initpanel`, `panelpoint` and `endpanel`. The first routine initializes the system so that the following polygon will be drawn and filled according to the `color` and `edge` arguments, interpreted as above. This routine must be called to initialize each polygon. Successive calls to `panelpoint` then add vertices to the current polygon and the figure is closed and filled by the `endpanel` call. This interface is often superior to `polgn` since it does not have the memory overhead of storing points in arrays.

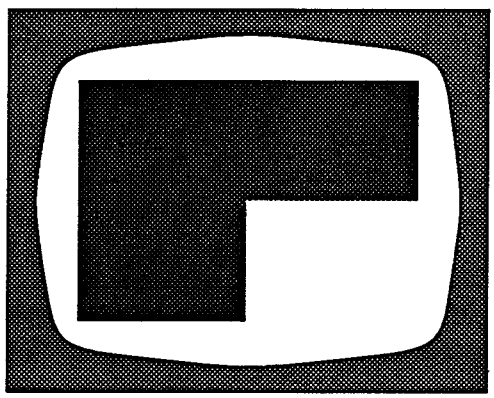
Note that filling with `color = 0` and `edge = 0` results in a "selective erase" - specific areas of the screen can be erased.

EXAMPLE

The following code draws a simple box in the foreground color using the `polgn` primitive and then takes a "bite" out of it with the alternate routines by drawing and filling in the

background color.

```
broken_square()
{
     double xpts[4], ypts[4];

     space(0.,0.,10., 10.); /* Define coordinate system */

     xpts[0] = 1.0; ypts[0] = 1.0;
     xpts[1] = 9.0; ypts[1] = 1.0;
     xpts[2] = 9.0; ypts[2] = 9.0;
     xpts[3] = 1.0; ypts[3] = 9.0;

     polgn(4, xpts, ypts, 1, 1);
                          /* Draw edge and fill */

     initpanel(0, 0);       /* Fill background color */

     panelpoint(5., 1.);
     panelpoint(9., 1.);
     panelpoint(9., 5.);
     panelpoint(5., 5.);
     endpanel();    /* Erase part of square */

     sendplot();
}
```



FORTRAN SYNOPSIS

```
SUBROUTINE KINITP(ICOL, IEDGE)
INTEGER ICOL, IEDGE

SUBROUTINE KPANLP(X, Y)
```

```
REAL*4 X, Y

SUBROUTINE KENDPA

SUBROUTINE KPOLGN(NPTS, XPTS, YPTS, ICOL, IEDGE)
INTEGER NPTS, ICOL, IEDGE
REAL*4 XPTS(*), YPTS(*)
```

SEE ALSO

box, color

NAME

`plothwm` - Analyze usage of system buffers.

SYNOPSIS

```
int plothwm()
```

DOMAIN

This routine may only be called in programs compiled with the *Plotix* libraries.

DESCRIPTION

Graphics commands are buffered internally on each node until flushed by one of the `sendplot` commands. This necessitates assigning a fixed size buffer for graphics. In order to "tune" the size of this buffer and ensure that neither graphics gets lost nor too much memory is devoted to this system the function `plothwm` returns the "high water mark" from the graphics system - i.e., the maximum number of bytes that were present between any two calls to the `sendplot` primitives. Using this function allows the user to exactly determine system memory requirements.

EXAMPLE

Assuming that the default system buffer size (8192 bytes) is currently in use the following code might be used to warn of impending overflows.

```
usendplot();    /* Make display "current" */

fasync(stderr);
if(plothwm() > 8000) {
    fprintf(stderr,
        "WARNING: graphics buffer tight !!\n");
}
```

Notice that we use the asynchronous *Cubix* mode for the warning message since it is not guaranteed that all processors will have filled their buffers to the same extent.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KPLOTH
```

SEE ALSO

`openpl`, `sendplot`

NAME

`getplxopt`, `setplxopt` - Manipulate hardware dependencies in *Plotix* programs.

SYNOPSIS

```
int getplxopt(option, pvalue)
char *option;
void *pvalue;

int setplxopt(option, value)
char *option;
long value;
```

DOMAIN

These routines may only be called in programs compiled with the *Plotix* libraries.

DESCRIPTION

Plotix attempts to provide device-independent graphical capabilities to *Express* programs. Due to the simple nature of the underlying graphics model this can usually be achieved to a large degree. To deal with those cases where either more capabilities are available or where more information is required about a particular *Plotix* implementation these two functions are provided.

`getplxopt` accesses the value of some internal property described by the character string `option` and stores it under the supplied pointer variable. The particular values of `option` supported on any particular device vary according to device capabilities and are listed in the device specific section of the *Plotix* chapter of the User's Guide. If the indicated property is not supported on the device in use -1 is returned.

The opposite function is provided to set internal state of some *Plotix* system with `setplxopt`. This routine takes a character representation of the required property and a single 32-bit value to which the indicated property will be set. If the named property is not supported on the device in use -1 will be returned.

When successful both routines return 0.

EXAMPLE

The following code segment initializes a *Plotix* system and also attempts to perform the following three tasks:

- Inquire how many distinct colors are available for drawing lines.
- Request output in "landscape" rather than the default "portrait" mode.
- Install a named "redraw" function which will be used in windowing versions of *Plotix* to repaint the screen under certain well-defined circumstances.

Note that any one of these requests may fail because the device currently in use may not be able to support them. In the code segment below we imagine that the calling program is able to deal with such failures without having to tell the user. In other situations we could look

for a 0 return value from the calls to indicate failure and issue diagnostics.

```
#include <stdio.h>

extern int my_repaint();

start_graph(pncols)
long *pncols;
{
     setplxopt("redraw", my_repaint);
     setplxopt("landscape", 0);

     if(openpl(8192, (FILE *)0) < 0) {
         fprintf(stderr,
             "ERROR: Failed to initialize graphics\n");
         exit(-1);
     }
     if(getplxopt("nlcolors", pncols) < 0) {
         fprintf(stderr,
             "Information unavailable, assume monochrome\n");
         *pncols = 2;
     }
```

We make the calls to `setplxopt` before the call to `openpl` while the call to `getplxopt` follows it. This is common practice - in many *Plotix* implementations the call to `openpl` is responsible for setting up a lot of the default behavior of the system and so it makes sense to make our preferences known before starting the system. This is one of the few cases in which `openpl` should *not* be the first call made to *Plotix*. Similarly we wait until after the device has been initialized before asking how many colors are available. This allows for systems which must be initialized before they can know how many colors are available.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KPXGOP(OPTION, VALUE)
CHARACTER *80 OPTION
INTEGER VALUE

INTEGER FUNCTION KPXSOP(OPTION, VALUE)
CHARCTER*80 OPTION
INTEGER VALUE
```

SEE ALSO

`openpl`

NAME

`profil` - Low level execution profiler

SYNOPSIS

```
profil(buffer, buflen, start, scale)
char *buffer;
void *start;
int buflen, scale;
```

DOMAIN

`profil` may only be called in the nodes.

DESCRIPTION

This routine serves to initialize the execution profiler. Every few milliseconds the program counter of the user application is examined and a histogram entry in the memory area denoted by `buffer` is incremented. The size of the histogram is `buflen` bytes and its first bin starts at the address specified as `start` - normally the name of some program subroutine.

In order to decide which histogram entry to increment a "mapping function" is applied to the program counter discovered by the system. First `start` is subtracted and then the result is multiplied by `scale` and divided by 0x10000 - i.e., the complete mapping is

```
bin_number = (PC - start)*scale/0x10000
```

The overall effect of the `scale` parameter is to map groups of adjacent program locations into the same histogram bin. The value `scale = 0x10000` maps every program location into a separate histogram bin, `scale = 0x8000` maps each pair of locations into a single bin, `scale = 0x4000` every group of four, and so on.

Using combinations of the `buflen`, `start` and `scale` parameters it is possible to allocate various memory ranges to be profiled. Note that no errors are incurred if the range is not large enough resulting in a calculated `bin_number` which is out of the histogram range. In this case a special "misses" counter is incremented. This latter feature also provides some diagnostic information concerning the success of the profiling attempt - if an incorrect profiling range is selected most of the histogram entries will be in the "miss" bin allowing easy diagnosis.

`profil` does not enable the profiler. An explicit call to `xprof_on` must be made to begin gathering profile data.

EXAMPLE

The following code is a skeleton of that which might typically be used to control the execution profile.

```
#define PROFSIZE (8192)     /* Size of profiler buffer */
#define PROFSCALE (0x2000)  /* Map eight bytes per bin */
```

```
int profbuf[PROFSIZE];

extern int myfunc();     /* Low address for profiling */
                         /* Found from the compiler map */
main()
{
/* Start off profiler */

     profil(profbuf, sizeof(profbuf), myfunc, PROFSCALE);
     xprof_on();

/* Application Phase 1., profiler running */

          ...
```

The choice of the `start` argument is most conveniently made in conjunction with the "linker map" provided by the compiler. This usually contains a list of the addresses of all the functions in an application and can be used to find the minimum.

FORTRAN SYNOPSIS

```
SUBROUTINE KPROFI(PRBUF, PRLEN, START, SCALE)
INTEGER PRBUF(*), PRLEN, START, SCALE
EXTERNAL START
```

SEE ALSO

`xtool` (command), `xprof`.

NAME

`rainbow` - Change color attributes.

SYNOPSIS

```
rainbow(from to)
int from, to;
```

DOMAIN

This routine may only be called in programs compiled with the *Plotix* libraries.

DESCRIPTION

This routine changes the association of color indices to device colors used by *Plotix*. By default a limited color map is used which can be extended with the `greyscale` and `rainbow` function calls.

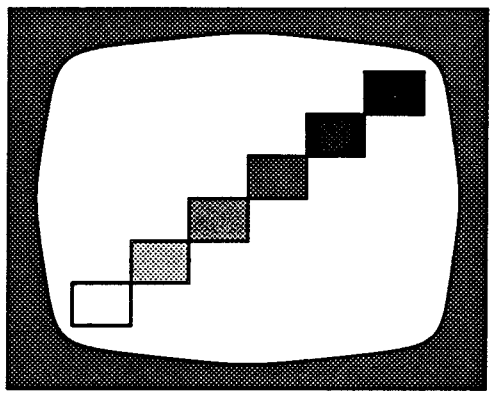
`rainbow` extends the *Plotix* color map by adding a smoothly varying color spectrum between the indicated values. The "rainbow" starts with red and varies, with full saturation and value, through the different hues; red, magenta, blue, yellow, cyan and back to red. The number of distinct colors available is hardware dependent but in any case *Plotix* will map the indicated range in as smooth a manner as possible.

On devices incapable of providing color output this function is treated exactly as a call to `greyscale`.

EXAMPLE

The following code draws a set of 6 boxes of varying colors along the diagonal of the screen image. Since the manual is printed on a monochrome device the output is exactly as if the call to `rainbow` were replaced with one to `greyscale`.

```
make_rainbow()
{
      int i;
      double v;

      space(0., 0., 6., 6.);

      rainbow(10, 15);
      for(i=0; i<6; i++) {
          v = i;
          box(v, v, v+1.0, v+1.0, 10+i, 0);
      }
      sendplot();
}
```



FORTRAN SYNOPSIS

```
SUBROUTINE KRAINB(LO, HI)
INTEGER LO, HI
```

SEE ALSO

color, greyscale

NAME

ramfopen - Create a RAM file.

SYNOPSIS

```
#include <stdio.h>

FILE *ramfopen(address, length)
char *address;
int length;
```

DOMAIN

This routine may only be called in programs compiled with the *Cubix* or *Plotix* libraries.

DESCRIPTION

The ramfopen function creates an "in-memory" or RAM file which may subsequently be accessed in an equivalent manner to any file opened with the fopen system call - i.e., the functions fprintf, fscanf, fread, etc. may be used. The significant difference between this FILE and others is that no data is ever transferred to the host file system, instead the data is merely retained on the node making the I/O calls.

This facility is useful for maintaining temporary scratch data and/or debugging. In the latter case it has been found to be extremely powerful for debugging asynchronous or real-time applications where the overhead of writing to a file on the host processor causes too much distortion in the behavior of the algorithm being debugged. The use of the RAM file together with the exdump command and the advanced options of excustom which prevent *Express* from overwriting memory when initialized make this a very powerful tool.

The address argument is the actual memory location which should be used for the RAM file. This is typically a statically declared array or a physical address in the memory of the particular processor in use. The RAM file will be truncated after length bytes have been placed in it. Further data will "wrap" in the file, overwriting the contents from the beginning.

EXAMPLE

The following code shows an important use of the RAM files as debugging aids. We assume that an asynchronous application is being developed around the exhandle routine. By inserting very simple code in the interrupt handling routine we can later examine the timing of the interrupts while controlling exactly how much we affect the real-time behavior of the algorithm.

```
#include <stdio.h>
#include "express.h"

FILE *ramfp, *ramfopen();
```

```
main()
{
     int src = DONTCARE, type = DONTCARE;
     int my_intr();
/*
 * Initialize the RAM file to some machine dependent physical
 * address. Print a cheery message.
 */
     ramfp = ramfopen((char *)0x80069000, 4096);
     fprintf(ramfp, "Here is the RAM file....!!\n");
/*
 * Set up the interrupt handler and ensure that no races
 * occur by forcing the nodes to synchronize.
 */
     exhandle(my_intr, &src, &type);
     exsync();
/*
 * Start program.........
 */
    for(i=0; i<1024; i++) {
        fprintf(ramfp, "This is iteration %d\n", i);
        grid_away();
     }

        ............
```

Now we set up the interrupt handling routine as follows:

```
int my_intr(buffer, len, psrc, ptype)
char *buffer;
int len, *psrc, *ptype;
{
     fputc('+', ramfp);      /* Short and sweet !! */
     process_interrupt();
     return 0;
}
```

Notice that the main routine, whose progress we might understand contains calls to `fprintf`, a relatively slow function while the interrupt handler uses the much faster `fputc`. This means that the timing of the interrupts is affected relatively little by the extra overhead of the I/O to the RAM file. Of course, we could add more information during the course of the interrupt processing but this may distort the performance of the application enough to actually make the bug disappear!

To retrieve the data from the file we typically wait for the program to finish, or crash and then use the `exdump` utility. If the machine crashed badly we may have to reload *Express* again before `exdump` will work. In this case we should take care to select a physical address at which to load the kernel, using the `excustom` tool so that the contents of our RAM file are preserved. (Be careful to choose an address which will not clash with the one chosen for the RAM file.)

Assuming that *Express* either survived the program or was reloaded successfully we can now retrieve the RAM file data with the command

```
exdump -B0x80069000 -n 4
```

In our case this may print something along the lines of

```
Here is the RAM file....!!
This is iteration 1
Thi++s is+ iteration 2
+This ++is+ iter+ation 3
```

and so on. Note that the asynchronous behavior of the program is at once apparent - we can see the interruption of even the call to `fprintf` in the main loop. Of course, there may not be enough information in this display to find the cause of a very mysterious problem but the amount of data dumped into the RAM file can be increased by simple I/O calls.

FORTRAN SYNOPSIS

This utility is currently unavailable in FORTRAN due to limitations in the I/O models of current compilers. We hope to make it available in the near future.

SEE ALSO

`exdump` (command), `exinit` (command), `excustom` (command).

NAME

`sendplot` - Flush graphical data to display surface.

SYNOPSIS

```
sendplot()

usendplot()

asendplot()
```

DOMAIN

This routine may only be called in programs compiled with the *Plotix* libraries.

DESCRIPTION

In the implementation of *Plotix* for parallel computers output is "buffered". This means that each `move`, `cont`, `panelpoint`, etc. command merely stores its parameters in an area of memory rather than immediately attempting to draw the associated object. This strategy is dictated by the fact that typical parallel computers have large computing power but little I/O bandwidth. As a result it makes no sense to send lots of small messages about graphical objects to the device since this would result in spending all ones time communicating rather than computing. Instead we store up a large number of objects and then send them all at once.

This method introduces the "flushing" concept to the graphical system. No data actually appears on the display surface until one of the three `sendplot` commands is executed. The differences between the three commands are typified by the following observations of common situations

`sendplot` All the nodes have been simultaneously drawing the same part of an image. This situation is quite common - it costs nothing to duplicate the same sequential effort in all processors. All nodes make the call to `sendplot` together but the data is only flushed to the display once.

`usendplot` The nodes have been working separately on their own pieces of the image and are now ready to ship it out to the display. All processors call `usendplot` together and the set of objects from each node appear in order of increasing processor number.

`asendplot` The nodes are working totally independently and asynchronously. A particular node wishes to send some data to the display and has no way of knowing the status of the other processors. Any node may call `asendplot` at any time.

The effect of these calls is to empty the buffer on the calling node ready for more graphical objects.

The buffer size required for graphical objects varies quite significantly from application to application. In some codes it may be possible and efficient to call the `sendplot` functions

quite regularly and so only a small buffer is required. Others may operate for long periods without flushing data and, as a result, need large buffers. The size of the graphics buffer is set in the call to `openpl`.

EXAMPLE

The following code segment illustrates one of the less obvious bugs possible under *Plotix*. We use the system calls to draw a "menu" and then accept selections from it with gin.

```
/* Use of PLOTIX - WRONG !! */

     space(0.,0.,4.,4.);

/* Draw simple menu on left hand edge of display */

     label("QUIT", .1, .5);     /* Option 0 */
     label("ITERATE", .1, 1.5); /* Option 1 */
     label("RESET", .1, 2.5);   /* Option 2 */
     label("OUTPUT", .1, 3.5); /* Option 3 */

/* Use gin to get user option from menu */

     gin(&key, &x, &y);
     option = (int)y;
```

The error here is that data is not flushed before the call to `gin`. As a result the user is asked to make a selection from an invisible menu. Not very friendly. The solution is, however, very straightforward; insert a call to `sendplot` before the call to `gin`. Note that this illustrates another aspect of the flushing commands - since all processors have been drawing the same thing and we only want to see one copy of it on the display the appropriate flushing function is `sendplot`.

FORTRAN SYNOPSIS

```
SUBROUTINE KSENDP

SUBROUTINE KASEND

SUBROUTINE KUSEND
```

SEE ALSO

`openpl`

NAME

`setvbuf` - Modify buffering character of file

SYNOPSIS

```
#include <stdio.h>

int setvbuf(fp, buffer, mode, size)
FILE *fp;
char *buffer;
int mode, size;
```

DOMAIN

This routine may only be called, *with the arguments discussed here*, in programs compiled with the *Cubix* or *Plotix* libraries. This function is normally also available to programs running on the host processor but with other arguments.

DESCRIPTION

This function is used to control the buffering characteristics of a file stream. Several types of buffering exist, indicated by the different values of the `mode` argument.

`_IOFBF`	Full buffering - the system allocates a fixed size buffer and automatically flushes whenever it becomes full.
`_IOLBF`	Line buffering - the system allocates a fixed size buffer and flushes it at the same points as in the full-buffered case and additionally whenever a carriage return is output and whenever input is requested from any other stream.
`_IONBF`	No buffering - each character is flushed to the file as soon as it is written.
`_IOCBF`	Circular buffering - a fixed size buffer is allocated an no automatic flushing performed. Whenever the buffer fills data is overwritten from the beginning.
`_IOEBF`	Extensible buffering - an initial buffer is allocated and extended whenever it fills. No automatic flushing is performed.

The last two buffering modes are *Cubix* extensions to the standard I/O library and are used to support `fmulti`. By default multi-mode files are assigned the _IOEBF mode while single mode files have `_IOLBF` if they are attached to terminals or `_IOFBF` for disk files.

While the default buffering modes should suffice it is occasionally necessary to switch to another option or to alter the buffer sizes assigned by the system.

The `setvbuf` call performs these functions on the file denoted by the argument `fp`. If the `buf` argument in non-NULL then it will be used in place of the system assigned buffer. The size should be given as the last argument. If the `buf` is `(char *) 0` then a buffer will be assigned by the system whenever necessary - it's size will be that of the `size` argument.

Note that the characteristics set by the `setvbuf` call are distinct for the "single" and

"multi" modes. Calling `setvbuf` for a file in single mode has no effect on that file once it is switched over to multi-mode.

`setvbuf` may only be called while the file is in an "idle" state - i.e., immediately after opening or a call to `fflush` but before data is read or written to the file.

EXAMPLE

One of the most common uses of `setvbuf` is to reclaim storage used in "extensible" mode. If a large file has been processed then the system has probably allocated a large internal buffer for the data. By default this is not returned when the data is flushed but may be reclaimed with the call

```
setvbuf(fp, (char *)0, _IOEBF, 1024);
```

which makes the system free up any buffer currently associated with the file `fp` and assign a new one of size 1024 whenever it becomes necessary.

SEE ALSO

`fmulti`

NAME

space - Define user coordinate system.

SYNOPSIS

```
space(lowx, lowy, highx, highy)
double lowx, lowy, highx, highy;

ortho_space(lowx, lowy, highx, highy, justify)
double lowx, lowy, highx, highy;
int justify;
```

DOMAIN

This routine may only be called in programs compiled with the *Plotix* libraries.

DESCRIPTION

These routines define a coordinate system to be mapped onto the current window. By default all plotting commands take place in a coordinate system which has (0., 0.) at its lower left corner and (1., 1.) at the upper right. After this call all future plotting commands, including the input request commands, will operate in the new coordinate system.

While the `space` routine covers the entire viewport with the selected coordinate range the `ortho_space` function can be used to preserve the aspect ratio of the indicated coordinate system. A mapping is created so that objects will actually appear with the correct shape independent of the specific characteristics of a particular output device - circles will be circular not elliptical.

Since a correctly normalized region may not completely fill the current viewport the `justify` parameter is used to indicate exactly where the region should be place. The value -1 implies that the new region should be placed either to the left or at the bottom of the viewport while +1 indicates the right or top. A zero value centers the region within the viewport.

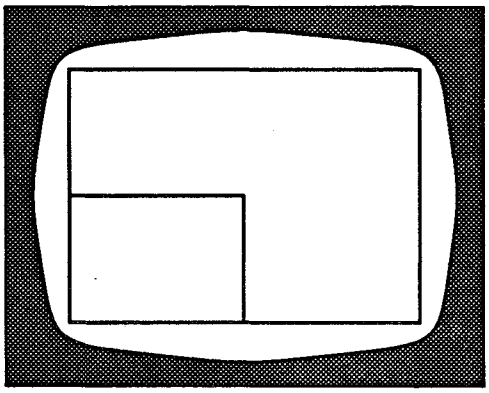
EXAMPLE

This example shows the effect of `space` transformations on simple objects. The routine `mybox` draws a unit square on the screen.

```
mybox()
{
      move(0., 0.);
      cont(1., 0.);
      cont(1., 1.);
      cont(0., 1.);
      cont(0., 0.);
}
```

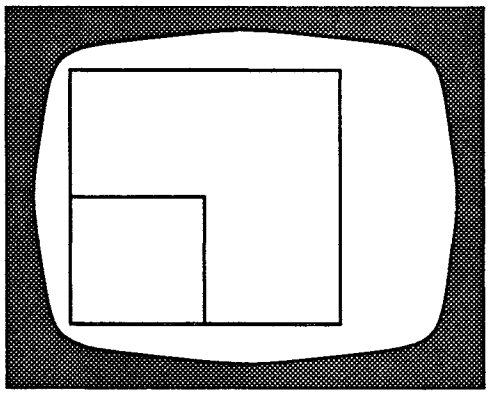
To see the effect of the space call consider the following sequence

```
mybox();        /* Default coords ==> full screen */
space(0., 0., 2., 2.);
mybox();        /* After space ==> quadrant only */

sendplot();
```



As can be seen the resulting "square" is not! To correct this we could instead use the ortho_space function as shown below. Note that we chose the justification that the used area should be to the left of the viewport.

```
mybox();    /* Default coords ==> "full" screen */
ortho_space(0., 0., 2., 2., -1);
mybox();    /* After space ==> quadrant only */

sendplot();
```



The *Plotix* manual contains a complete example program in which the exgrid routines are used to map processors to their own individual windows on the display surface and space is used to map each individual processor region to its own coordinate range. Note

that it is possible to have different coordinate ranges in separate processors.

FORTRAN SYNOPSIS

```
SUBROUTINE KSPACE(X0, Y0, X1, Y1)
REAL*4 X0, Y0, X1, Y1

SUBROUTINE KORTHO(X0, Y0, X1, Y1)
REAL*4 X0, Y0, X1, Y1
```

SEE ALSO

vport

NAME

syncmode - Specify synchronous or asynchronous system calls

SYNOPSIS

```
int syncmode(flag)
int flag;
```

DOMAIN

This routine may only be called in programs compiled with the *Cubix* or *Plotix* libraries.

DESCRIPTION

This call provides a system override which controls the overall synchronous or asynchronous behavior of all non I/O *Cubix* system calls.

By default the system is in "synchronous mode" which means that all function calls must be made loosely synchronously. Furthermore each node must address its system requests to the same system console. (Note, however, that asynchronous I/O is still supported in this mode on a file by file basis).

Calling `syncmode` with a zero argument places the system in asynchronous mode. All further operating system requests are made on a node by node, first come-first served, basis. In this mode any node may address any host processor with impunity but the responsibility for maintaining "sensible" ordering lies with the user. It is important to note, however, that I/O requests will occur with the synchronization implied by the individual file modes, and not by the status of the `syncmode` function.

The value returned by this call is the previous value of the synchronization flag - it can be used in subsequent calls to `syncmode` to restore the mode altered by the call.

EXAMPLE

The asynchronous mode is rather difficult to control not the least because the inherent asynchronicity introduced into applications make them harder to debug. It can, however, be useful in system with multiple consoles, each under the control of a different group of nodes. In the following example we suppose that nodes with even processor numbers should communicate with host processor "`H1`" (in the naming system used by `cnftool`) while those with odd processor numbers remain connected to the main system console.

```
try_async()
{
     struct nodenv nodedata;

     exparam(&nodedata);        /* Get processor numbers */

     if((nodepata.procnum % 2) != 0) {
         console_node(0x8001);
     }
```

```
    syncmode(0);                    /* Go "asynchronous" */

/* Nodes 0 and 1 execute a (hypothetical) command
 * called "reboot" on their respective hosts
 */
    if(nodedata.procnum < 2)
        system("reboot");
}
```

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KCBXSY(MODE)
INTEGER MODE
```

SEE ALSO

console_node.

NAME

vport - Specify area of display to hold image.

SYNOPSIS

```
int vport(lowx, lowy, highx, highy)
double lowx, lowy, highx, highy;

setvport(window)
int window;
```

DOMAIN

This routine may only be called in programs compiled with the *Plotix* libraries.

DESCRIPTION

This routine allocates a certain area of the display surface to an image. The supplied parameters are expressed as fractions of the total view surface so that the default 0.0<x<1.0 and 0.0<y<1.0 is the entire display. By selecting smaller regions in x and y it is possible to confine an image to a smaller region of the display. This is useful if the final image is required to have a certain aspect ratio or in parallel processing applications where each processor is to be assigned a piece of the view surface.

Plotix allows several viewports to be present on the same display. Each is indicated by a number returned by the corresponding call to vport and is selected by a call to setvport. Note that each viewport or window has its own coordinate range specified by a call to space and that clipping is performed independently in each window. Further, since the call to vport selects the new viewport a call to space to set up a coordinate system must come after the corresponding call to vport.

EXAMPLE

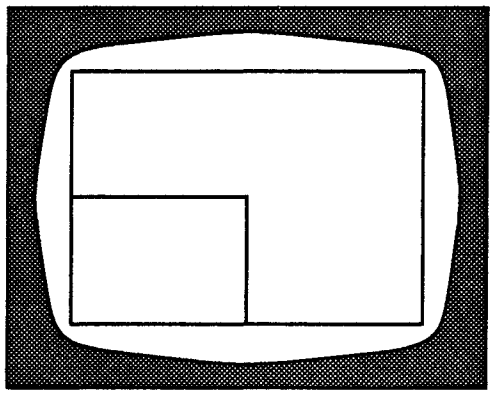
This example shows the effect of vport transformations on simple objects. The routine mybox draws a unit square on the screen.

```
mybox()
{
      move(0., 0.);
      cont(1., 0.);
      cont(1., 1.);
      cont(0., 1.);
      cont(0., 0.);
}
```

To see the effect of the vport call consider the following sequence

```
      mybox();/* Default coords ==> full screen */
```

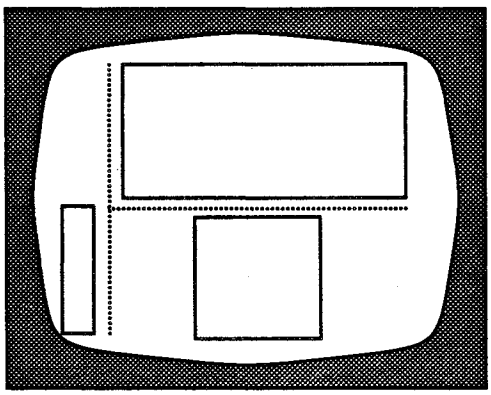
```
     vport(0., 0., .5, .5);
     mybox();/* After vport ==> quadrant only */

     sendplot();
```



To see the effect of multiple viewports consider the following code segment. We create three windows. The left window has a call to space which means that the "box" fills only the bottom part of the viewport. The second window has no call to space so its coordinate range will have the usual default. The last window uses ortho_space to make a viewport with the correct aspect ratio - the square actually comes out square!

```
setup_vports()
{
     int left_win, top_win, low_win;

/* Left window, scaling range (0,0) --> (1,2) */

     left_win = vport(0.0, 0.0, 0.2, 1.0);
     space(0.0, 0.0, 1.0, 2.0);

/* Top window, default scaling range (0,0) --> (1,1) */

     top_win = vport(0.2, 0.5, 1.0, 1.0);

/* Lower window, scaled (0,0) --> (1,1), correct
 * aspect ratio
 */
     low_win = vport(0.2, 0.0, 1.0, 0.5);
     ortho_space(0.0, 0.0, 1.0, 1.0, 0);

/* Set up windows, draw the squares ..... */
```

```
        setvport(left_win);
        mybox();

        setvport(top_win);
        mybox();

        setvport(low_win);
        mybox();

        sendplot();
}
```



FORTRAN SYNOPSIS

```
INTEGER FUNCTION KVPORT(X0, Y0, X1, Y1)
REAL*4 X0, Y0, X1, Y1

SUBROUTINE KSETVP(IVPORT)
INTEGER IVPORT
```

SEE ALSO

space

NAME

xprof_on, xprof_off - Control execution profiler.

SYNOPSIS

```
xprof_on()

xprof_off()
```

DOMAIN

These routines may only be called from the nodes.

DESCRIPTION

xprof_on is used to enable and start the execution profiler which must have been previously initialized with a call to profil. Subsequently a periodically scheduled event occurs which causes the program counter of the user application to be "logged" in an internal structure. xprof_off reverses this process - until a subsequent call to xprof_on no execution profiling will be performed.

The profiler is initially off and must be explicitly enabled with calls to profil and xprof_on.

The log of profiling information is written to the host file system with xprofcp or xprof_end.

EXAMPLE

The following code is a skeleton of that which might typically be used to control the execution profiler.

```
#define PROFSIZE (8192)
#define PROFSCALE (0x1fff)
int profbuf[PROFSIZE];

extern int myfunc();          /* Low address for profiling */
                              /* Found from the compiler map */
main()
{
/* Start off profiler */

     profil(profbuf, sizeof(profbuf), myfunc, PROFSCALE);
     xprof_on();

/* Application Phase 1., profiler running */

          ...

/* Phase 1 complete, dump data with xprofcp/elt or
```

```
 * xprof_end
 */
        ...

/* Application Phase 2., profiler off since data dumped */

        ...

/* Application phase 3., turn on profiler again */

    xprof_on();
        ...

/* Program over, dump data again and exit */

        ...
    exit(0);
}
```

Notice that we can selectively profile pieces of code. In this mode it makes sense to dump out the profile data independently to separate files for simplicity in later analysis.

FORTRAN SYNOPSIS

```
SUBROUTINE KXPON

SUBROUTINE KXPOFF
```

SEE ALSO

xtool, profil, xprofcp, xprof_end.

NAME

`xprof_inq, xprof_end` - Manipulate execution profiler under *Cubix*

SYNOPSIS

```
int xprof_inq()

xprof_end()
```

DOMAIN

These routines may only be called from the nodes in programs compiled with either the *Cubix* or *Plotix* libraries.

DESCRIPTION

These routines provide a simple control interface to the execution profiler for applications running in the *Cubix* environment.

`xprof_inq` returns an integer value representing the state of the "-m" runtime switch on the `cubix` command line. This can be used to conveniently enable/disable the profiling system at runtime. Consider a typical command

```
cubix -n 4 toyland 1024 1024 <noddy.dat
```

Since no "-m" switch is present a call to `xprof_inq` will return zero. If we modify the above command to

```
cubix -mcxe -n 4 toyland 1024 1024 <noddy.dat
```

then the return value would be 1 since the character 'x' appears in the monitoring switch, "-m".

`xprof_end` is used to finally dump profiling data to the host computer file system. A file called "`xprof.out`" is created for later analysis with the `xtool` utility. In addition the profiler is disabled and its initial state reset to zero. This allows distinct phases of an application to be profiled totally independently.

EXAMPLE

The following code is a skeleton of that which might typically be used to control the execution profiler.

```
#define PROFSIZE (8192)
#define PROFSCALE (0x1fff)
int profbuf[PROFSIZE];

extern int myfunc();          /* Low address for profiling */
                              /* Found from the compiler map */
main()
{
/* Start off profiler */
```

```
    if(xprof_inq()) {
        profil(profbuf, sizeof(profbuf), myfunc, PROFSCALE);
        xprof_on();
    }

/* Application Phase 1., profiler running */

        ...

/* Phase 1 complete, dump data with xprof_end, rename
 * output file.
 */

    if(xprof_inq()) {
        xprof_end();
        rename("xprof.out", "phase1.xprof");
    }

/* Application Phase 2., profiler off since xprofelt
 * called
 */

        ...

/* Application phase 3., turn on profiler again */

    if(xprof_inq()) xprof_on();
        ...

/* Program over, dump data again and exit */

    if(xprof_inq()) {
        xprof_end();
        rename("xprof.out", "phase3.xprof");
    }
    exit(0);
}
```

Notice that we can selectively profile pieces of code. In this mode it makes sense to dump out the profile data independently to separate files for simplicity in later analysis. Note that the output files are preserved with the `rename` function calls. Without this the second call to `cprof_end` would overwrite the file produced by the first call.

FORTRAN SYNOPSIS

```
INTEGER FUNCTION KXPINQ()

SUBROUTINE KXPEND
```

SEE ALSO

xtool, profil, xprofcp, xprof.

NAME

xprofcp, xprofelt - Dump execution profile data.

SYNOPSIS

```
xprofcp()

xprofelt(filename)
char *filename;
```

DOMAIN

xprofcp may only be called in the host processor while xprofelt may only be called in the nodes.

DESCRIPTION

These routines are used to dump the execution profile data collected with the xprof functions. For each call to xprofelt on the nodes there must be a call to xprofcp in the host processor. The profiling data will be written to a file on the host with the name filename supplied in the node program.

In addition to dumping out the profile data xprofelt also turns off the profiler and resets its internal state so that further invocations of the execution profiler will begin from the zero state and hence be totally independent.

EXAMPLE

The following code is a skeleton of that which might typically be used to control the execution profiler.

1. Host Program

```
main()
{
/* Allocate nodes, load programs */

        ...

/* Execute algorithm phase 1 and then dump data to
 * "phase1.xprof"
 */
             ...

    xprofcp();

/* Execute phase 2, profiler off */

        ...
```

```
/* Execute phase 3, profiler on, dump data to
 * "phase3.cprof"
 */
                ...

     xprofcp();
     exit(0);
}
```

2. Node Program

```
#define PROFSIZE (8192)
#define PROFSCALE (0x1fff)
int profbuf[PROFSIZE];

extern int myfunc();          /* Low address for profiling */
                              /* Found from the compiler map */
main()
{
/* Start off profiler */

     profil(profbuf, sizeof(profbuf), myfunc, PROFSCALE);
     xprof_on();

/* Application Phase 1., profiler running */

          ...

/* Phase 1 complete, dump data with xprofcp/elt */

     xprofelt("phase1.xprof");

/* Application Phase 2., profiler off since xprofelt
 * called
 */
          ...

/* Application phase 3., turn on profiler again */

     xprof_on();
          ...

/* Program over, dump data again and exit */

     xprofelt("phase3.xprof");
```

```
        exit(0);
}
```

FORTRAN SYNOPSIS

```
SUBROUTINE KXPCP

SUBROUTINE KXPELT(FNAME)
CHARACTER*80 FNAME
```

SEE ALSO

xtool, profil, xprof, xprof_end.

Classification of routines

A listing of the *Express* routines, broken down by functionality

This section lists the various functions and routines available to *Express* programs grouped according to functionality. While no exact division between routines is possible this information may serve as a useful guide to "related" functions.

User Commands

		(Man page)
acctool	Analyze accounting data	acctool
cnftool	Configure Transputer systems	cnftool
ctool	Analyze communication profile data	ctool
cubix	Download and execute *Cubix* programs, I/O server	cubix
etool	analyze event profile data and "toggles"	etool
excustom	Modify *Express* system parameters	excustom
exdump	Retrieve data from RAM files	exdump
exinit	Reboot and reload *Express* kernel	exinit
exreset	Reset transputer system	exinit
exstat	Display node usage information	exstat
ndb	Source level debugger	ndb
xtool	Analyze execution profile data	xtool

Compilers

		(Man page)
ncc	C compiler and linker for NCUBE	ncc
nf77	FORTRAN compiler and linker for NCUBE	nf77
symcc	C compiler and linker for SYMULT	symcc
symf77	FORTRAN compiler and linker for SYMULT	symf77
tcc	Logical Systems C compiler and linker for transputers	tcc
tcc3L	3L C compiler and linker for transputers	tcc3L
tfc	3L FORTRAN compiler and linker for transputers	tfc

Processor Allocation and Control

		(Man page)
exargldl	Load an argument list into a single node	exploud
exargldv	Load an argument vector into a single node	exploud
exclose	Deallocate processor group	exclose
exenvldl	Load an environment vector into a single node	exploud
exload	Load program into all nodes	exload
exloadl	Load program into all nodes with argument list	exload
exloadle	Load program with argument list and environment	exload
exloadv	Load program into all nodes with argument vector	exload
exloadve	Load program with argument vector and environment	exload
exmain	Start execution of main program	exstart

exopen	Allocate a group of processors	exopen
expause	Arrange for program to be loaded "stopped"	expause
expid	Translate UNIX process ID to *Express* process ID	exshare
expload	Load a program into a single node	expload
exshare	Share a processor group between multiple host programs	exshare
exstart	Start execution of a node program	exstart

Basic Communication System

		(Man page)
exexctype	Define meaning of "read/write" wildcards	extype
exinctype	Define meaning of "read/write" wildcards	extype
exread	Read a message	exread
exreadfd	Read a message and place contents into file	exreadfd
extest	Test for an incoming message - non-blocking	extest
exvread	Read a *vector* message	exvread
exvwrite	Send a *vector* message	exvread
exwrite	Send a message	exwrite
exwritefd	Send a message, data taken from a file	exreadfd

"Global" Communication System

		(Man page)
exbroadcast	Interprocessor broadcast	exbroadcast
exchange	Synchronous multi-node data exchange	exchange
excombine	Apply user supplied operation to distributed data set	excombine
exconcat	Transfer distributed data to local memory	exconcat
exsync	Synchronize processors	exsync
exvchange	Synchronous multi-node *vector* exchange	exchange

Asynchronous Communication System

		(Man page)
exhandle	Install asynchronous message handler	exhandle
exreceive	Read a message - non-blocking	exreceive
exsend	Send a message - non-blocking	exsend

Hardware Dependent Communication System

		(Man page)
exchanon	Re-enable *Express* processing on a channel	exchan
exchanoff	Disable *Express* processing on a channel	exchan
exchanrd	Read bytes from disabled channel	exchan
exchanwt	Write bytes to disabled channel	exchan

Decomposition Tools

		(Man page)
exgridbc	Define boundary conditions on user domain	exgrid
exgridcoord	Determine position in user domain	exgrid
exgridinit	Initialize decomposition system	exgrid
exgridnode	Determine communication parameters from user domain	exgrid
exgridproc	Map user domain coordinates to processor number	exgrid
exgridsize	Distribute data among processors	exgrid
exgridsplit	Distribute processors on user domain	exgrid
exparam	Determine run-time configuration	exparam

Customization

		(Man page)
excustom	Modify *Express* system parameters	excustom

I/O

		(Man page)
abort	Immediately terminate node program	abort
aopen	Asynchronously open a file	aopen
fasync	Switch file I/O mode to "async"	fmulti
fcntl	Modify the synchronous/asynchronous file character	fcntl
fmulti	Switch file I/O mode to "multi"	fmulti
fsingl	Switch file I/O mode to "single"	fmulti
isasync	Inquire file I/O mode	fmulti
ismulti	Inquire file I/O mode	fmulti
mread	Read independent data into nodes	mread
mread2d	Read two-dimensional data set into nodes	mread2d
mwrite	Write independent data from node	mwrite
mwrite2d	Write two-dimensional data set into nodes	mread2d
setvbuf	Assign buffering modes to files	setvbuf
syncmode	Assign overall synchronous/asynchronous mode	syncmode

Debugging

		(Man page)
abort	Immediately terminate node program	abort
exbreak	Halt program at breakpoint	exbreak
expause	Load node program "stopped" at a breakpoint	expause
malloc_debug	Internal consistency checking for malloc/free	malloc_debug
ramfopen	Create in-memory RAM file	ramfopen

Multi-Host systems (Man page)

Multitasking (Man page)

Graphics (Man page)

marker	Draw marker symbol	marker
move	Move current position without drawing	move
openpl	Initialize Plotix	openpl
ortho_space	Define user coordinate range	space
panelpoint	Define point in polygon	panel
plothwm	Monitor graphics buffer usage	plothwm
polgn	Draw polygon	panel
rainbow	Modify color look-up table, create HSV table	rainbow
sendplot	Flush graphical data to display surface synchronously	sendplot
setclip	Enable clipping against rectangular region	clipper
setplxopt	Indicate hardware dependent option	plxopt
setvport	Switch between "windows"	vport
space	Define user coordinate range	space
vport	Define a region of the display as a "window"	vport
usendplot	Flush independent data to display synchronously	sendplot

Performance Analysis (Man page)

cprof_end	Terminate communication profiler and dump data	cprof_inq
cprof_inq	Inquire setting of runtime '-mc' switch	cprof_inq
cprof_on	Enable communication profiler	cprof
cprof_off	Disable communication profiler	cprof
cprofcp	Receive communication profile data in host processor	cprofcp
cprofelt	Send communication profile data to host processor	cprofcp
eprof_add	Indicate a "user" event	eprof
eprof_end	Terminate event profiler and dump data	eprof_inq
eprof_init	Initialize memory for event profiler	eprof
eprof_inq	Inquire setting of runtime '-me' switch	eprof_inq
eprof_label	Assign a label to a user specified "event"	eprof
eprof_on	Enable event profiler	eprof
eprof_off	Disable event profiler	eprof
eprof_toggle	Enable/disable timing for a region of source code	etoggle
eprof_toginit	Initialize memory for a "toggle"	etoggle
eprofcp	Receive event profile data in host processor	eprofcp
eprofelt	Send event profile data to host processor	eprofcp
profil	Assign memory for execution profiler	profil
xprof_end	Terminate execution profiler and dump data	xprof_inq
xprof_inq	Inquire setting of runtime '-mx' switch	xprof_inq
xprof_on	Enable execution profiler	xprof
xprof_off	Disable execution profiler	xprof

`xprofcp`	Receive execution profile data in host processor	`xprofcp`
`xprofelt`	Send execution profile data to host processor	`xprofcp`

Host Interface Utilities

		(Man page)
`callhost`	Call host routine from *Cubix* node program	`callhost`
`gethost`	Inquire host capabilities	`gethost`
`rethost`	Return from host routine in *Cubix* node programs	`callhost`
`starthost`	Start host routine in *Cubix* node program	`callhost`

Utility Routines

		(Man page)
`_ex_swab`	Reverse bytes in 16-bit quantities	`exswap`
`_ex_swad`	Reverse bytes in 64-bit quantities	`exswap`
`_ex_swaw`	Reverse bytes in 32-bit quantities	`exswap`
`extick`	Measure time in hardware "ticks"	`extime`
`extime`	Measure time in microseconds	`extime`

Library Availability

The correspondence between C and FORTRAN libraries and the synchronization properties of *Express* functions

1 Correspondence between C and FORTRAN

The first two columns of the following table list the equivalent C and FORTRAN routines. A blank entry indicates that no such routine exists.

2 Synchronization Rules

The third column of the table indicates the synchronization modes associated with each function. The various codes are:

a	These routines may be called with no regard to any synchronization constraints - any node may make such a call at any time.
ls, all	These routines must be made "loosely synchronously" in *all processors*. When a node calls one of these routines it will halt until all other nodes have called the same routine. Arguments may or may not be different in each node according to the particular function involved.
ls, group	These routines must be made "loosely synchronously" in all participating processors. Typically this means that two processors will be involved in some transaction in which case the first to arrive will halt until the others arrive at the synchronization point.
mode	The synchronization requirements of these calls depend on the global synchronization state of the system, as modified with the `syncmode` or `KCBXSY` system calls. If the global synchronization mode is "on" (the default) then these routines behave as though their synchronization constraint were "ls, all". If the global state is "off" they behave as "a".

3 Libraries and Programming Models

The last column in the table indicates the libraries and/or programming model combinations which support the named routines. These latter are coded as follows:

h	Routine is available to programs running on the host processor, linked with the *Express* library.
n	Available to programs running on the parallel computer nodes in the "Host-Node" programming style. Such programs should NOT be linked with either the *Cubix* or *Plotix* libraries.
c	These routines are part of the *Cubix* I/O library and may only be linked with programs using the *Cubix* programming model. Usually a compiler switch is available to indicate this programming model and the associated libraries.
p	These routines are to be found in the *Plotix* library which can be linked to programs running under the *Cubix* programming model. In some cases a compiler switch is available which links both the *Cubix* and *Plotix* libraries. If this is not so on your system the *Cubix* switch should be used supplemented by the pathname of the *Plotix* library.

4 NOTES

(i) While no corresponding routine is available in FORTRAN the effect can be achieved by modifying the parameters to an OPEN statement. See the section on "open file modes" in the *Cubix* chapter for more details.

(ii) These calls may be made asynchronously but they have no subsequent effect on the objects they access. The graphical open functions, for example, may be made asynchronously but the mode in which data is flushed to the output device is still determined by the flushing function used. Similarly a file opened with one of the asynchronous "open" functions still has as its default the "singl" access mode.

(iii) These functions *can* be called asynchronously be will usually be used in a mode similar to "ls, group". Once invoked they leave the affected nodes in a state which will almost certainly function in a an unpredictable manner until the corresponding action has been performed on other members of the group.

(iv) These functions can be called asynchronously but should be used with extreme care when so doing. Because of their nature it is easy to introduce "race conditions" when using these routines asynchronously. In most cases it is easy (and safer) to force a synchronization after using one of these routines.

(v) The synchronization behavior of these routines depends upon the "mode" of the associated file. For "singl" and "multi" mode files the constraint is "ls, group" while it becomes "a" for "async" mode files.

C	FORTRAN	Synchronization	Library
	KXEXIT	ls, all	n
	KXINIT	ls, all	h,n,c,p
_ex_swab	KXSWAB	a	h,n,c,p
_ex_swad	KXSWAD	a	h,n,c,p
_ex_swaw	KXSWAW	a	h,n,c,p
abort	KABORT	a	c,p
aerase	KAERAS	a	p
aexecve	KAEXEC	a	c,p
agin	KAGIN	a	p
aopen	(i)	a, (ii)	c,p
aopenpl	KAOPEN	a, (ii)	p
asendplot	KASEND	a,	p
aspect	KASPEC	a,	p
box	KBOX	a,	p
callhost	KCALHO	mode	c,p
closepl	KCLOSP	ls, all	p
color	KCOLOR	a	p
console_node	KCONND	a, (ii)	c,p
cont	KCONT	a	p
contour	KCNTOR	ls, all	p
cprof_end	KCPEND	ls, all	c,p
cprof_inq	KCPINQ	mode	c,p
cprof_off	KCPOFF	a	n,c,p
cprof_on	KCPON	a	n,c,p
cprofcp	KCPCP		h
cprofelt	KCPELT	ls, all	n
display_node	KDISND	a, (ii)	p
dotext	KDOTEX	a	p
endpanel	KENDP	a	p
eprof_add	KEPADD	a	n,c,p
eprof_end	KEPEND	ls, all	c,p
eprof_init	KEPINI	ls, all	n,c,p
eprof_inq	KEPINQ	mode	c,p
eprof_label	KEPLAB	a	n,c,p
eprof_off	KEPOFF	a	n,c,p
eprof_on	KEPON	a	n,c,p
eprof_toggle	KEPTOG	a	n,c,p
eprof_toginit	KEPTGI	a	n,c,p
eprofcp	KEPCP		h
eprofelt	KEPELT	ls, all	n
erase	KERASE	ls, all	p
exaccess	KXACCS		h
exargldl			h
exargldv			h

C	FORTRAN	Synchronization	Library
exbreak	KXBREA	a	n,c,p
exbroadcast	KXBROD	ls, group	h,n,c,p
exchange	KXCHAN	ls, group	h,n,c,p
exchanoff	KXCHOF	a, (iii)	n,c,p
exchanon	KXCHON	a, (iii)	n,c,p
exchanrd	KXCHRD	ls, group	n,c,p
exchanwt	KXCHWT	ls, group	n,c,p
exclose	KXCLOS		h
excombine	KXCOMB	ls, group	n,c,p
exconcat	KXCONC	ls, group	n,c,p
excustom	KXCUST		h
execve	KEXEC	mode	c,p
exenvld			h
exexctype	KXEXCT	a, (iv)	h,n,c,p
exgridbc	KXGDBC	a	h,n,c,p
exgridcoord	KXGDCO	a	h,n,c,p
exgridinit	KXGDIN	a	h,n,c,p
exgridnode	KXGDNO	a	h,n,c,p
exgridproc	KXGDPR	a	h,n,c,p
exgridsize	KXGDSI	a	h,n,c,p
exgridsplit	KXGDSP	a	h,n,c,p
exhandle	KXHAND	a, (iv)	n,c,p
exinctype	KXINCT	a, (iv)	h,n,c,p
exload	KXLOAD		h
exloadl			h
exloadle			h
exloadv			h
exloadve			h
exmain	KXMAIN		h
exopen	KXOPEN		h
exparam	KXPARA	a	h,n,c,p
expause	KXPAUS		h
expid	KXPID		h
expload	KXPLOA		h
exread	KXREAD	a	h,n,c,p
exreadfd			h
exreceive	KXRECV	a	n,c,p
exsemalloc	KXSEMI	a	n,c,p
exsemfree		a	n,c,p
exsemsig	KXSEMS	a	n,c,p
exsemwait	KXSEMW	a	n,c,p
exsend	KXSEND	a	n,c,p
exshare	KXSHAR		h
exsleep	KXSLEE	a	n,c,p

C	FORTRAN	Synchronization	Library
exstart	KXSTAR		h
exsync	KXSYNC	ls, all	n,c,p
extest	KXTEST	a	h,n,c,p
extick	KXTIME	a	n,c,p
extime	KXTIME	a	n,c,p
exvchange	KXVCHA	ls, group	h,n,c,p
exvread	KXVREA	a	h,n,c,p
exvwrite	KXVWRI	a	h,n,c,p
exwrite	KXWRIT	a	h,n,c,p
exwritefd			h
fasync	KASYNC	ls, all	c,p
fmulti	KMULTI	ls,all	c,p
forder	KORDER	a, (ii)	c,p
fsingl	KSINGL	ls, all	c,p
gethost	KGETHO	mode	c,p
getplxopt	KPXGOP	mode	p
getpoint	KGETPT	a	p
gin	KGIN	mode	p
greyscale	KGREYS	a	p
initlevel	KINITL	a	p
initpanel	KINITP	a	p
isasync	KISASY	a	c,p
ismulti	KISMUL	a	c,p
label	KLABEL	a	p
linemod	KLINEM	a	p
malloc_avail		a	c,p
malloc_debug		a	c,p
malloc_print		ls, group (v)	c,p
malloc_verify		a	c,p
marker	KMARKE	a	p
move	KMOVE	a	p
mread	KMREAD	ls, all	c,p
mread2d	KMRD2D	ls,all	c,p
mwrite	KMWRIT	ls,all	c,p
mwrite2d	KMWT2D	ls,all	c,p
openpl	KOPENP	mode	p
ortho_space	KORTHO	a	p
panelpoint	KPANLP	a	p
plothwm	KPLOTH	a	p
polgn	KPOLGN	a	p
profil	KPROFI	a	n,c,p
rainbow	KRAINB	a	p
ramfopen	(i)	a	c,p
rethost	KRETHO	mode	c,p

C	FORTRAN	Synchronization	Library
sendplot	KSENDP	mode	p
setclip	KSETCL	a	p
setplxopt	KPXSOP	mode	p
setvbuf		a, (ii)	c,p
space	KSPACE	a	p
starthost	KSTRHO	mode	c,p
syncmode	KCBXSY	a, (ii)	c,p
usendplot	KUSEND	ls, all	p
vport	KVPORT	a	p
xprof_end	KXPEND	ls, all	c,p
xprof_inq	KXPINQ	mode	c,p
xprof_off	KXPOFF	a	n,c,p
xprof_on	KXPON	a	n,c,p
xprofcp	KXPCP		h
xprofelt	KXPELT	ls, all	n

Index of Routines

An alphabetical listing of routines, variables, commands and macros

Index to Routines

This index contains an alphabetical list of the various subroutines, macros and variables which may be of use to *Express* programs. Each routine has an indication of the page on which its definition and arguments can be found.

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

X

Index

General index to *Express* and the examples from the text

General Index

This index is the general reference for all the topics discussed in this manual. It lists not only the various functions/routines but also the examples and other points of note. Index entries referring solely to subroutines have their page numbers in typewriter font: exwrite `178`, for example.

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W