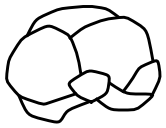


Contact

Research Computing
The Ohio State University
1224 Kinnear Road
Columbus, OH 43212

Consulting

Phone: (614) 292 8492
Internet: trollius@tbag.osc.edu

Copyright

This document is protected by copyright.
Author: GDB

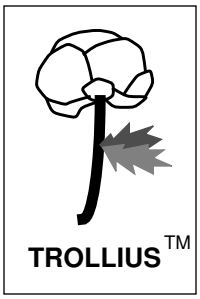
This document is based in part on the 2.0 edition of *Trollius Reference Manual for C Programmers*, dated July 6, 1990, Copyright 1990 The Ohio State University and Cornell Research Foundation.
Authors: RYL/MB

© Copyright 1992 The Ohio State University

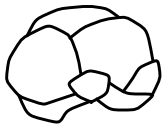
Trademarks

Trollius is a trademark of The Ohio State University and the Cornell Research Foundation.

UNIX is a trademark of AT&T.



Trollius Library Reference in C



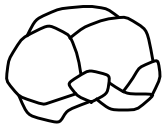
EOPTION	227	bad/unknown option
ENODERANGE	228	bad node range specification
EBADNODE	229	specified node does not exist
EBADNODESPEC	230	invalid mnemonic node specification
ENORDESCRIPTORS	232	out of route descriptors
ENOBFDESCRIPTORS	233	out of buffer descriptors
ENOBTDESCRIPTORS	234	out of boot descriptors
ENOFLDESCRIPTORS	235	out of flat descriptors
ENOPDESCRIPTORS	236	out of kenya descriptors
ENOKDESCRIPTORS	237	out of process descriptors
EBADMOSESREQ	238	bad moses request
EMAXBULLETS	239	can't kill datalink
EBADTAG	240	argv, load point, or boot image tag is invalid
EDELETE	241	delete from list failed
ENOBOOTSTRAP	242	no bootstrap present
EBADREC	243	invalid record format
EBADASCIINUMB	244	character string that is supposed to represent a number has invalid format
EHIGHWATER	245	booted system is too big

The Cubix file error codes are:

ECXREQ	260	badly synchronized Cubix request
ECXMEMBER	261	invalid Cubix member

UNIX error codes used in Trollius are:

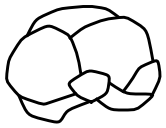
EINTR	4	interrupted system call
E2BIG	7	filename too long
ENOMEM	12	insufficient memory
EINVAL	22	invalid argument
EWOULDBLOCK	35	message sync would block



Appendix B: Error Codes

The Trollius error codes are:

ENOEVDESCRIPTORS	201	out of kernel message descriptors
ECIRCUITINUSE	202	virtual circuit in use
ENOEXDESCRIPTORS	203	out of cleanup function descriptors
EFULL	204	out of descriptors
ENOTPROCESS	205	given process ID does not identify a process attached to the kernel, or no process exists that satisfies the selection parameters
EBADREQUEST	206	invalid request
ENOTMALLOCED	207	memory block to be freed with free() was not obtained with malloc(), or the memory block to be freed with rfree() was not obtained with rmalloc(), or the memory block has already been freed
ESERVICE	209	service unavailable
EMAGIC	210	bad magic number
EBLOWN	211	blown stack
EIMPOSSIBLE	212	impossible, ergo a bug
EGARBAGE	214	garbage: for example, malloc()'s internal lists have been corrupted, probably due to a process writing to locations outside the allocated block
ENOKERNEL	215	no running kernel
ENOTATTACHED	216	process not attached to kernel
EPANIC	217	lost system integrity
EMSGLEN	218	message length exceeds maximum
EEOF	219	end of file
ECOLLIDE	220	conflict in registering identical multicast identifiers
ESHARE	223	can't find shared segment
EBADLINK	224	invalid link index
EACK	225	bad acknowledgment
EUSAGE	226	command line violates the rules as declared

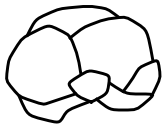


rspawn	spawn a remote copy of this process
rstate	report status of processes

Remote Level (libbuckeye)

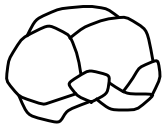
Several other remote level functions are provided in the Buckeye supplement to Trollius.

rledcc	carbon copy LED status to another node
rledop	set, clear, or reverse LEDs
rlestate	get status on LEDs
rtrfget	unload trace data into file
rtrsig	signal trace client to call rtrstore()
rtrstate	get trace data and client status from daemon
rtrsweep	free trace data memory in daemon
rtvclr	clear graphics frame buffer
rtvrect	plot rectangle in graphics frame buffer
rtvsnake	plot 1-D snake in graphics frame buffer
rtvstate	get status of remote graphics frame buffer
rtvtext	plot text in graphics frame buffer



Remote functions request a service from a Trollius server (daemon) located on any node.

bfprobe	check for arrival of network message
getnodes	get array of node identifiers from router
getsmart	block until router has a route table
rbfstate	report status of buffers
rbtadd	add object to boot daemon
rbtboot	boot objects into ITB node
rbtflush	remove objects from boot daemon
rbtspread	recursively boot objects into multiple ITB nodes
rbtstage	load boot objects from one boot daemon to another
rbtstate	get status on boot objects
rbuffer	set buffering parameters
rcask	report expected number of multireel messages
rcast	register a multicast identifier
rcasttype	register a multicast identifer
rcreate	create a process
rdoom	send a signal to a process, usually used to kill processes
rdrain	copy memory to a file
recho	send message to echo server
rflat	copy data to remote buffer identified by tag
rforget	same as rget(), but forgets the tag
rfstate	get status on remote file service
rget	get the real address rflat() attached to a given tag
rload	load a file
rloadgo	load and execute a program
rmelt	remove a multicast identifier
rmstate	report status of memory allocation on remote node
rpwait	wait for remote child process to complete
rremroute	remove Trollius routing information
rrnodewait	wait for nodeid to be defined
rroute	set complete Trollius routing information
rsetnet	set Trollius network information
rsetroute	set Trollius routing information



in	transputer channel input
ldpri	returns the priority the process is running in
ldtimer	reads the transputer on-chip timer
mint	minimum integer word (0x80000000)
move	block move
out	transputer channel output
resetch	reset a channel
runp	run a process
saveh	save high priority queue registers
savel	save low priority queue registers
setpri	set priority
sthb	store high priority back pointer
sthf	store high priority front pointer
stlb	store low priority back pointer
stlf	store low priority front pointer
stopp	stop process
sttimer	set the timer
tin	sleeps until transputer timer reaches some value
wReg	returns a pointer to the workspace of the calling function

These and other transputer level operations are also provided as in-line assembler macros. Include the header file <c.h>.

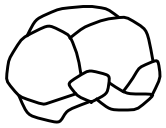
Remote Level (libt)

A few remote level functions are located in the T library.

rfree	low-level free memory block
rfreeall	low-level free all memory blocks
rmalloc	low-level allocate memory
rpattach	become a remote process
rpdetach	cease to be a remote process
rtrstore	store trace data with trace daemon

Remote Level (libtrillium)

Most of the Trollius low level functions are built into the ITB T library. Most of the remote level functions and some miscellaneous functions are found in the trillium library. Link this library to an ITB program by adding the -ltrillium option to the ITB compiler invocation.



getnotb	return the number of OTB nodes
getntype	return the number of nodes whose types have specified bits set to specified values
getorigin	return the origin node identifier
getrent	get route table information
getrentc	get route table information, and put in cache
getroute	given destination, find datalink to use
getrtype	get type of remote node
nrecv	network sub-level message receive
nsend	network sub-level message send
ntry_recv	non-blocking network sub-level message receive
ntry_send	non-blocking network sub-level send
precv	physical sub-level message receive (ITB)
psend	physical sub-level message send (ITB)
trecv	transport sub-level message receive
tsend	transport sub-level message send

Data Representation (libt)

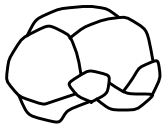
Different computers in the Trollius network may store the bytes that comprise a data type in different orders. The data representation functions are provided to convert between the local and standard Trollius representation. If no conversion is necessary, the functions will have no effect. Include the header file <t_types.h>.

ttol	converts integer representation from Trollius to local
ltot	converts integer representation from local to Trollius
ttols	converts short representation from Trollius to local
ltots	converts short representation from local to Trollius
ttolf	converts float representation from Trollius to local
ltotf	converts a float representation from local to Trollius
ttold	converts double representation from Trollius to local
ltotd	converts double representation from local to Trollius

Transputer Specific (libt)

Some transputer operations are useful for optimizations not possible within the confines of the C language. The following transputer machine level operations are implemented as functions in the ITB T library.

flick	moves the process to the back of the scheduling queue
gajw	general adjust workspace

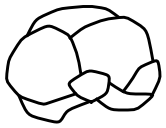


kexit	call cleanup functions and exit
_kexit	exit without calling cleanup functions
kinit	initialize data structures and call kattach()
kpar_Fe	create a parallel process, allocating it a stack and separate data space (ITB)
kpar_He	run a function as a parallel process (ITB)
kpar_Na	allocate stack space, create a parallel process (ITB)
kpause	wait until a signal is delivered
krecv	local message receive
ksend	local message send
ksigblock	block a given signal
ksignal	specify a signal handler for a signal
ksigretry	retry a given signal
ksigsetmask	set which signals to block
ksigsetretry	set which signals to retry
kstate	get state of a process from kernel's tables
ktrstart	enable trace generation and set TID
ktrstop	disable trace generation
ktry_recv	non-blocking local message receive
ktry_send	non-blocking local message send

Network Level (libt)

These functions access network level services. Include the header file <net.h>.

brecv	basic network message receive
bsend	basic network message send
drecv	datalink sub-level message receive
dsend	datalink sub-level message send
dtry_recv	non-blocking datalink sub-level message receive
dtry_send	non-blocking datalink sub-level message send
getnall	return total number of nodes
getncast	number of multicasts passing through the local node
getnet	get information on Trollius network
getnitb	return the number of ITB nodes
getnodeid	return the local node identifier
getnodetype	return the local node type



rfread	read data from an open file
async	change existing file access method to asynchronous
isasync	report if file is asynchronous
ismulti	report if file is multiple
issingl	report if file is singular
multi	change existing file access method to multiple
rforder	change the order in which files are serviced with the multiple access method
singl	change existing file access method to singular
rfwrite	write data to an open file

General OS Interface (libt)

All operating systems, including Trollius, provide standard interfaces to architecture independent services. Trollius adds some extra functions in this category. The following functions are provided ITB to take the place of the same functions provided by the OTB native operating system.

_exit	exit program without releasing resources
abort	abort program
calloc	allocate local memory for a group of elements
exit	release resources and exit program
free	release memory allocated with malloc
getpid	returns the process ID
malloc	allocate local memory
perror	print error message
realloc	reallocate local memory
sleep	suspend execution for an interval

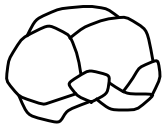
The following functions are provided on all nodes:

terror	print Trollius error message
ttime	read the current value of the system timer, in seconds

Kernel Level (libt)

The following functions access kernel level services.

atkexit	register cleanup function
kattach	attach process to the kernel
kcreate	create a new process (ITB)
kdetach	detach process from the kernel
kdoom	deliver a signal to a process



Appendix A: Library Organization

Trollius functions are organized into the T library, the Trillium library and the Buckeye library. The T library contains the essential services needed to support typical message passing programs. The Trillium and Buckeye libraries are remote service libraries, mostly supporting advanced programming. The OTB and ITB compilers are separate products from Trollius. A compiler is expected to provide a standard I/O library (containing formatting and buffering functions), a C intrinsic library of functions commonly used by C programmers, and a math library.

UNIX Filesystem (libt)

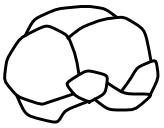
Many of the standard file I/O functions are supported. UNIX standard wrappers are also available. The functions open(), read(), etc., simply pass all their parameters to rfopen(), rfread(), etc.

`rfaccess`	determine accessibility of a file
`rfchdir`	change current directory
`rfclose`	close a file
`rfdup`	duplicate a file descriptor
`rfdup2`	duplicate a file descriptor to a specific handle
`rffstat`	get file status given a file descriptor
`rfftruncate`	set file descriptor to given length
`rfgetwd`	get pathname of current working directory
`rflseek`	adjust current location in a file
`rfmkdir`	make a new directory
`rfopen`	open a file for reading or writing
`rfread`	read data from an open file
`rfrmdir`	remove a directory
`rfstat`	get file status given a pathname
`rfsystem`	issue a shell command
`rftruncate`	set named file to a specific length
`rfunlink`	remove a file (or a link to a file)
`rfwrite`	write data to an open file

Cubix Filesystem (libt)

Trollius supports the following functions with the Cubix access method along with Cubix specific functions.

`rfclose`	close a file
`rflseek`	adjust current location in a file
`rfopen`	open a file



`in(chan, buf, len);`	Input a message.
`out(chan, buf, len);`	Output a message.
`resetch(chan);`	Reset a communication channel.
`_exit();`	Die right now.

Transputer Hardware Techniques

Native processes created by kpar functions that have attained no greater status in life will need to communicate using the underlying environment. On transputer based ITB nodes, this means utilizing the processor's unique instruction set. The header file <c.h> contains a fine set of macros to access hardware features of the local node not accessible from the C language.

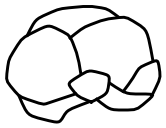
A transputer communication channel is nothing more than a word of memory. Allocate a word by defining a word length type (int) as a static variable. Before communicating on the channel, initialize it with the resetch() macro. It is best to do this in the parent process and let the children inherit it through the data segment.

```
resetch(int *chan);
in(int *chan, char *buf, int len);
out(int *chan, char *buf, int len);
```

The argument chan points to the communication channel and buf points to the message or the message buffer. The length of the message and message buffer, defined by the len argument, must be the same.

The transputer communication primitives can be used on hardware communication links in lieu of psend(). This further reduces the overhead of neighbour node message passing.

In some debugging circumstances, it may be useful to have a process die without any further action: no cleanup, no detaching, no status return. This is accomplished with the _exit() function.



`terror(message);`	Print an error message.
`ttime();`	Get value of system timer.

Miscellaneous Functions

Every process has the static system error variable, errno. It is set to the appropriate error code upon failure of any Trollius function. Once an error has been detected, a corresponding message can be printed on the standard output with the terror() function. Terror() is similar to the UNIX perror() function.

```
void terror(char *message);
```

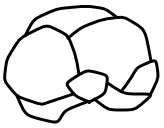
The caller supplied message is prefixed to the system error message.

Trollius error codes are listed in Appendix B.

The function ttime() returns the value of the local node's timer in seconds.

```
double ttime();
```

The precision of the timer depends on the underlying environment and/or the hardware of the local node. Ttime() is usually suitable for timing application performance.



`ktrstart(tid);`	Start tracing and define trace identifier.
`ktrstop();`	Stop tracing.

Trace Generation

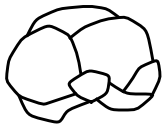
Traces are records of occurrences in the execution of a Trollius program. Assuming that a process has its trace master switch enabled (see *Process Runtime Flags*), kernel level functions explicitly control the generation of traces. Also, nsend() and nrecv() (see *Network Sub-level*) implicitly generate traces.

Besides the master boot-time switch, a program controllable runtime switch also regulates trace generation. Both the master switch and the runtime switch must be on for trace generation to occur. The ktrstart() and ktrstop() functions each generate a trace, and then turn on or off the runtime switch, respectively.

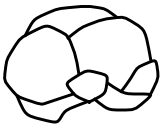
```
int ktrstart(int tid);
int ktrstop();
```

When a process is created with its master trace switch set in the on position, its runtime switch also begins in the on position. To immediately disable trace generation, ktrstop() must be called. Nsend() and nrecv() implicitly generate a trace if both switches are on and the synchronization event is positive, a user event.

All traces from the whole Trollius multicomputer are identified in post-processing and visualization steps by a trace identifier (tid). By default, a process generates traces with a tid equal to the nodeid on which the process is running. This is useful for debugging the entire parallel system on the node level. A process is free to change its tid to any value with the ktrstart() function. It is not necessary to call ktrstop() first in order to change the tid of a trace enabled process.



les_nled	number of LEDs at the node
les_shadow	current state of all LEDs
les_cc	current carbon copy nodeid



```
rledop(nodeid, op, bits);
                        Operate on remote LEDs.
rledcc(nodeid, cc);     Carbon copy LEDs to remote node.
rlestate(nodeid, state);
                        Report status of remote LEDs.
```

LED Control and Monitoring

A node may have an attached, programmable LED (Light Emitting Diode) peripheral. Trollius provides a remote service to control these LEDs, mostly for the purposes of demonstration applications, but also for debugging.

A LED is assumed to have two states, on and off. The rledop() function provides several operations that change the state of LEDs. The first argument is the remote nodeid controlling the LEDs. The second argument is the operation code and the last argument is a bit mask, with one bit controlling each LED on a node. A node may have a variable number of LEDs. If a node has N LEDs, the lower order N bits in the bit mask control them.

```
int rledop(int nodeid, int op, int bits);
```

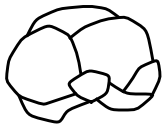
The following operations are provided by rledop():

LEDSET	Turn on all LEDs whose bits are set (1) in the bit mask. Do not change the state of LEDs whose bits are reset (0) in the bit mask.
LEDCLR	Turn off all LEDs whose bits are set.
LEDNOT	Toggle all LEDs whose bits are set.
LEDALL	Turn all LEDs whose bits are set. Turn off all LEDs whose bits are reset.

```
int rledcc(int nodeid, int cc);
int rlestate(int nodeid, struct lestate *state);
```

Rledcc() causes one node to reflect the state of all LEDs to a “carbon copy” node. The rlestate() function returns information on the configuration and status of a node’s LEDs. The principle argument is a pointer to an empty lestate structure, defined in <ledreq.h>, with the fields described below:

```
struct lestate {
     int            les_nled;
     int            les_shadow;
     int            les_cc;
}
```



The rtvsnake() function is similar to rtvrect(). A single starting pixel ordinate is given and the given number of pixels are plotted along the horizontal rows of the frame buffer.

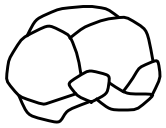
```
int rtvsnake(int nodeid, int x1, int len,
     char *colours);
```

The rtvtext() function views the screen in characters, not pixels, but otherwise operates much like rtvsnake(). At starting character coordinates rtvtext() plots the null terminated ascii character array across the horizontal character rows of the frame buffer. One colour is given for all characters.

```
int rtvtext(int nodeid, int x1, int y1,
     int colour, char *text);
```

The rtvclr() function sets all pixels in the frame buffer to the background colour (black).

```
int rtvclr(int nodeid);
```



```
rtvclr(nodeid);
```
Clear graphics frame.

```
rtvrect(nodeid, x1, y1, xlen, ylen, colours);
```
Plot rectangle of pixels.

```
rtvsnake(nodeid, x1, len, colours);
```
Plot snake of pixels.

```
rtvtext(nodeid, x1, y1, colour, text);
```
Plot text on graphics frame.

```
rtvstate(nodeid, ptvstate);
```
Get status of graphics frame.

Data Visualization

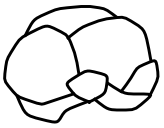
Trollius provides a simple interface to the graphical capabilities of a node, provided they exist. Before these functions can be used intelligently, some basic information must be learned about the graphics frame buffer. The rtvstate() function returns a tvstate structure.

```
int rtvstate(int nodeid, struct tvstate
     *ptvstate);
struct tvstate {
     int  tvs_xlen;
     int  tvs_ylen;
     int  tvs_ncolour;
     int  tvs_xtext;
     int  tvs_ytext;
};
```

tvs_xlen	width of frame buffer in pixels
tvs_ylen	height of frame buffer in pixels
tvs_ncolour	length of colour palette
tvs_xtext	width of frame buffer in characters
tvs_ytext	height of frame buffer in characters

The rtvrect() function plots a rectangle of pixels anywhere inside the frame buffer. The upper left pixel coordinates and the pixel dimensions of the rectangle are given. Each pixel is a byte (0 - 255) index into the colour map.

```
int rtvrect(int nodeid, int x1, int y1, int xlen,
     int ylen, unsigned char *colours);
```



`malloc(amount);`	Allocate memory from runtime heap.
`free(pmem);`	Free runtime heap allocated memory.
`rmalloc(nodeid, amount, region, align);`	Allocate memory from the system.
`rfree(nodeid, pmem);`	Free memory allocated from the system.

Memory Allocation

All available memory on an ITB node after booting is managed by a Trollius system process. It can manage several regions of memory that may have different characteristics (e.g.: performance). For transputer based ITB nodes the following memory region codes are defined in <machine/maconfig.h>.

`MAIN`	main memory
`ONCHIP`	transputer on-chip memory

The remote level function rmalloc() allocates aligned memory in the chosen region, providing a large enough block exists to satisfy the amount requested. Garbage collection is not performed so fragmentation is possible.

```
char *rmalloc(int nodeid, int amount, int region,
     int align);
```

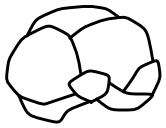
The memory block allocated begins at an address evenly divisible by the align argument. Unless you can figure out a good reason to allocate non-addressable memory on a remote node, always set the nodeid argument to LOCAL. Use rmalloc() and rfree() when infrequently allocating large memory blocks.

```
int rfree(int nodeid, char *pmem);
```

Memory allocated with rmalloc() must be freed with rfree().

Allocation from the system memory process is a high overhead procedure. If repeated allocations and de-allocations of small blocks of memory are required, the runtime heap allocation should be used. This procedure takes place entirely within the caller's process. The system memory process is only periodically consulted. The runtime functions are malloc() and free(). Memory allocated by malloc() must be de-allocated with free().

```
char *malloc(int amount);
int free(char *pmem);
```



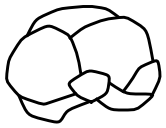
`select_value` remote process selection argument

`maxsize` number of entries in status array

Rdoom

The remote level equivalent of kdoom() is rdoom(). It delivers a signal to a remote process. The process selection capability means that several processes may receive signals. See *Monitoring and Controlling Kernel Processes* for more information on signals.

```
int rdoom(int nodeid, int select_mode,
     int select_value, int signum);
```



ps_reply Ignore this field.

ps_index This is essentially a remote level process identifier, distinct from the kernel level pid.

ps_kenya The pprocess structure contains remote level specific information and is shown below.

ps_kernel The kernel status structure was described in the previous section.

ps_name This is the filename of the load module.

```
struct pprocess {
     int             p_pid;
     int             p_stksize;
     char            *p_stack;
     char            *p_static;
     struct exec     *p_loadpt;
     int             p_argc;
     char            **p_argv;
     int             p_flags;
     int             p_nodeid;
     int             p_event;
};
```

p_pid kernel level process identifier

p_stksize stack size

p_stack stack base pointer

p_static static data segment pointer

p_loadpt load module pointer

p_argc number of command line argument strings

p_argv pointer to command line argument strings

p_flags runtime flags

p_nodeid parent process's nodeid

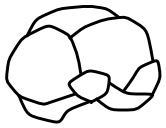
p_event parent process's synchronization event for exit status

```
int rstate(int nodeid, int select_mode,
     int select_value, struct pstate *ppstate,
     int maxsize);
```

The other arguments to rstate are:

nodeid nodeid of the target remote process

select_mode remote process selection mode



rstate(nodeid, select_mode, select_value, ppstate, maxsize);

Get status of a remote process.

rdoom(nodeid, select_mode, select_value, signum);

Deliver a signal to a remote process.

Monitoring and Controlling Remote Processes

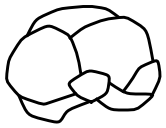
The remote process monitoring and control functions, rstate() and rdoom(), have an elaborate method of identifying target processes. They are not limited to affecting a single process per call identified by pid. They can affect several processes (on a single node) in one call identified by several different methods. Each function takes an argument that identifies the selection method. The selection method codes are defined in <preq.h>.

SELECT_PID	Select by pid. Obviously this specifies a single remote process. The value argument is the pid.
SELECT_INDEX	Select by index number. Again, this specifies a single remote process. The value argument is the index.
SELECT_BLKD	Select remote processes blocked on a specified event. The value argument is the event.
SELECT_LOADPT	Select remote processes with the specified load module address. This affects all remote processes running the same program. The value argument is the load module address.
SELECT_APPL	Select all user remote processes. The value argument is ignored.
SELECT_ALL	Select all remote processes. The value argument is ignored.

Rstate

The state of a remote process is returned by the rstate() function. The principle argument is a pointer to an empty pstate structure, defined in <preq.h> and shown below.

```
struct pstate {
     int               ps_reply;
     int               ps_index;
     struct pprocess   ps_kenya;
     struct kstate     ps_kernel;
     char              ps_name[PSMAXNAME];
};
```



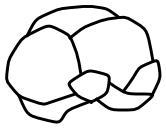
Some signals are used internally by the system. Some have useful default options and others are completely left to the user. The most useful signal is the one that obliges a kernel process to terminate itself. Signals are defined in <ksignal.h>.

SIGUDIE	terminate
SIGARREST	suspend execution
SIGRELEASE	resume execution
SIGA	user defined (default ignored)
SIGB	user defined (default ignored)

A kernel process can deliver a signal to another kernel process on the same node by calling the kdoom() function.

```
int kdoom(int pid, int signum);
```

The ksignal(), ksigblock() and ksigsetmask() functions operate the same as their UNIX counterparts. A kernel request function interrupted by a signal before completion is automatically retried. With the ksigretry() and ksigsetretry() functions, which operate similar to ksigblock() and ksigsetmask() respectively, the user can disable automatic kernel request retry and receive the EINTR error code (defined in <terror.h>) instead.



`kstate(pid, pkstate);`	Get status of kernel process.
`kdoom(pid, signum);`	Send signal to kernel process.
`ksignal(signum, func);`	Install a signal handler.
`ksigblock(mask);`	Block selected signals.
`ksigsetmask(mask);`	Set entire blocking mask.
`ksigretry(mask);`	Retry request after selected signals.
`ksigsetretry(mask);`	Set entire retry mask.
`sigmask(signum);`	Create signal mask.

Monitoring and Controlling Kernel Processes

The status of a kernel process is returned by the kstate() function. The principle argument is a pointer to a kstate structure, define in <kreq.h> and shown below.

```
struct kstate {
     int  ks_state;
     int  ks_priority;
     int  ks_event;
     int  ks_type;
     int  ks_signal;
};
```

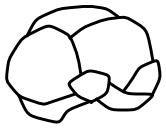
`ks_state`	The execution state of the process is defined in <kreq.h>. See the manual page for kstate().
`ks_priority`	The process priority was first given as the argument to kinit().
`ks_event`	If the execution state is blocked, this field holds the event.
`ks_type`	Likewise, this field holds the message type.
`ks_signal`	Any pending, but undelivered signals are stored here.

```
int kstate(int pid, struct kstate *pkstate);
```

Signals

Trollius provides a UNIX like signal package. The signals are different, but the support functions have UNIX names prefixed with the letter k. The key difference between UNIX and Trollius signals is that Trollius cannot guarantee immediate delivery of signals. It must wait for active processes to make a kernel request[1].

1. Immediate signal delivery is a simple feature in an executive kernel. Trollius is a server kernel. It does not completely control scheduling. Active processes are beyond its control.



```
int rpwait(int *nodeid, int *pid, int *status);
```

Rpattach

A Trollius process created by kpar_Fe() (see *Local Node Parallel Functions*) may become a remote function and enjoy all the control and monitoring features of the remote level. The rpattach() function takes a single argument, the nodeid, but don't consider any other value but LOCAL for this argument. The RTF_NOWAIT flag (see below) is automatically turned on with rpattach().

```
int rpattach(int nodeid);
```

Process Runtime Flags

Trollius process runtime flags are used mostly for internal system purposes but some are useful to the application programmer. They are defined in <kio.h>. When creating a process with rcreate() or rloadgo(), the following runtime flags provide added functionality:

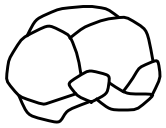
RTF_ONCHIP	On transputer based ITB nodes, place the stack of the new process in on-chip RAM.
RTF_NOWAIT	Without this flag, a message will be sent to the parent process upon the death of the child process. The parent should wait for this message with rpwait(). Use this flag if you don't want to be bothered with rpwait().
RTF_TRACE	Allow the process to generate execution traces.

Summary of Process Terminology

We have used a lot of adjectives to describe processes in the preceding two sections. All processes are created via the underlying environment. Who knows about a process, and hence, who can control and monitor it, is what distinguishes processes. Here is a summary of process terminology.

native	known only to the parent process on the local node; control and monitoring through the underlying environment
Trollius	(kernel) known to Trollius processes on the local node in addition to the parent process; control and monitoring through kernel level functions
remote	known to Trollius processes on all nodes in the multicomputer; control and monitoring through remote level functions; implies Trollius process

In the next two sections, we will look at control and monitoring at the kernel and remote levels.



argv	This is a pointer to the argv data structure. To meet the C standard, the argv structure consists of space for argc + 1 pointers followed by argc null terminated strings.
argvsize	The implementor of rloadgo() forgot to compute the size of the argv structure and the caller must provide it.
pid	same as rcreate()

```
int rloadgo(int src_nodeid, int dest_nodeid,
     int stksize, int rtflags, char *filename,
     int argc, char **argv, int argvsize,
     int *pid);
```

Rspawn

Rspawn() duplicates a process. The runtime flags are inherited from the original process along with a copy of the static data segment. If the process is spawned to the local node, the text (code) segment will be shared with the original process. Rspawn() requires the following arguments:

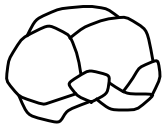
src_nodeid	The original process is running on this node.
select_mode	This code defines the identification method for the original process on the source node. See *Monitoring and Controlling Remote Processes*. The most common selection mode is SELECT_PID, defined in <preq.h>.
select_value	The meaning of this argument is defined by the previous argument. Usually, this argument is the pid of the original process.
dest_nodeid	Create the process on this node.
argc	same as rcreate()
argvtag	same as rcreate()
pid	same as rcreate()

```
int rspawn(int src_nodeid, int select_mode,
     int select_value, int dest_nodeid, int argc,
     int argvtag, int *pid);
```

Rpwait

By default, when a remote process dies, a message is sent to its parent containing the exit status, the value passed to kexit() by the child process. The rpwait() function is an easy way for the parent to wait for any of its children to die. Rpwait() requires the following arguments:

nodeid	The nodeid of the deceased child process is returned.
pid	The pid of the deceased child process is returned.
status	The exit status of the deceased child process is returned.



src_nodeid	Read the load module from a file on this node.
dest_nodeid	Load the load module on this node.
loadtag	Identify the load module on the destination node with this arbitrary integer.
filename	This is the name of the load module file.

```
int rload(int src_nodeid, int dest_nodeid,
     int loadtag, char *filename);
```

Rcreate

Rcreate() requires the following arguments:

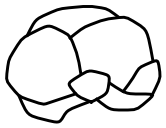
nodeid	Run the new process on this node.
loadtag	This is the load module tag selected for rload().
rtflags	Pass these runtime flags to the new process. See *Process Runtime Flags*, later in this section.
stksize	A stack of this size is required by the process.
argc	This is a count of command line argument strings. If this is set to zero, the argvtag argument will be ignored.
argvtag	This arbitrary integer tag identifies the argument strings resident on the destination node. While a file is loaded onto a remote node with rload(), a data structure is loaded onto a remote node with rflat() (see manual page for rflat()). See the manual page for rcreate() for information on how to build an argv structure.
pid	The pid of the new process is returned in this argument.

```
int rcreate(int nodeid, int loadtag, int flags,
     int stksize, int argc, int argvtag,
     int *pid);
```

Rloadgo

Rloadgo() is a wrapper for rload() and rcreate(). It also takes care of loading the argv data structure onto the destination node. It requires the following arguments:

src_nodeid	same as rload()
dest_nodeid	same as rload()
stksize	same as rcreate()
rtflags	same as rcreate()
filename	same as rload()
argc	same as rcreate()



```
rload(src_nodeid, dest_nodeid, loadtag, filename);
```
Load a load module onto an ITB node.

```
rcreate(nodeid, loadtag, rtflags, stksize, pid,
argc, argvtag);
```
Create process from load module.

```
rloadgo(src_nodeid, dest_nodeid, stksize, rtflags,
filename, argc, argv, argvsize, pid);
```
Load and create process from load module.

```
rspawn(src_nodeid, select_mode, select_value,
dest_nodeid, argc, argvtag, pid);
```
Spawn an existing process.

```
rpwait(nodeid, pid, status);
```
Wait for a child process to die.

```
rpattach(nodeid);
```
Become a remote process.

Remote Node Parallel Programs

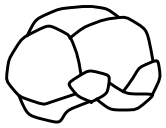
Trollius supplies functions that create parallel processes on remote ITB nodes from entire programs. A program that is prepared to become a process is called a load module and originally resides on a disk attached to a Trollius node. The steps to create a process are 1) load the load module onto the remote ITB node and 2) create the process. Creation involves allocating a stack. One function, rloadgo(), performs both steps. Rload() performs only the first step and rcreate() performs only the second step. Rspawn() duplicates an existing process.

There are two major differences between these functions and the kernel level functions of the previous section.

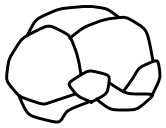
- Processes start execution at the load module entry point, not a user specified function. They automatically call kinit() before main(). The text (code) segment is shared only when rspawn() is used to duplicate a process on the local node.
- The child processes are indirectly created by the parent processes through a third system process. The system provides other remote services for control and monitoring of processes created by rcreate(), rloadgo() and rspawn(). We call them remote processes. Remote processes are easy to terminate; the parent does not have to wait for them to terminate.

Rload

Rload() requires the following arguments:



```
        khead.k_event = 11;
        khead.k_type = 0;
        khead.k_flags = 0;
        khead.k_length = 0;
        khead.k_msg = (char *) 0;
        (void) ksend(&khead);
/*
 * After telling the parent that I am going to
 * die, I must not do anything more than exit.
 */
        kexit(0);
}
```



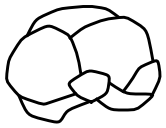
must advise the parent of their imminent exit through the underlying environment. Iron weight processes have the luxury of using kernel level message passing. Creating processes is easy enough, but terminating them will benefit from more support from future Trollius releases.

A parent that exits while child processes are running presents a major hazard. When a process exits, all memory gained through rmalloc() is automatically released. For a parent process, this includes memory allocated for the stack and possibly data segment of child processes. Yet the child processes are still running! The parent can avoid releasing memory by exiting with _kexit() instead of kexit() but the shared text (code) segment will still be released. In Trollius 2.2, the safest procedure is to ensure that parents outlive their children.

Kpar_Fe Example

We present an example of heavyweight process creation and destruction.

```
/*
 * Create a heavy weight process, wait for it to
 * die and then clean up.
 */
static void         child();
main()
{
     char           *childmem;
     int            pid;
     struct kmsg    khead;
     childmem = kpar_Fe(child, 4096, &pid, 0);
     khead.k_event = 11;
     khead.k_type = 0;
     khead.k_flags = 0;
     khead.k_length = 0;
     khead.k_msg = (char *) 0;
     (void) krecv(&khead);
     (void) rfree(LOCAL, childmem);
}
static void
child()
{
     struct kmsg    khead;
     (void) kinit(0);
```



returned child pid, the number of arguments to be passed to the main function and the arguments themselves. Each argument must be a data type represented by four bytes. Eight byte data types may be passed, but they count as two arguments.

The stack and the static data segments are allocated together within kpar_Fe() by rmalloc() (see *Memory Allocation*). Kpar_Fe() returns a pointer to the combined data/stack memory block. After the child process terminates, the parent process can free the memory with rfree(). The data segment begins as a copy of the parent's data segment so data can be passed to the child at creation, but not back to the parent at termination.

A process created by kpar_Fe() may attach to the kernel with kinit() and use any Trollius service. There is a way for a native process created by kpar_Fe() to become a remote process after becoming a Trollius process. See *Remote Node Parallel Programs* and see the rpattach() manual page.

Kpar_He

Kpar_He(), the lightweight function, sets up the arguments on the caller provided stack and creates the child process. The static data segment is shared with the parent process and data may be communicated through it. The limitation of a shared data segment is that the child process may not attach to the kernel and use Trollius services. A child process created by kpar_He() and its parent must synchronize through the underlying environment.

```
int kpar_He(void func(), char *pstack,
     int nparms, int parm1, ...);
```

The caller must allocate a large enough stack and pass the address immediately following the last word of the stack (the pointer to the first word of the stack plus the stack size) as the second argument to Kpar_He().

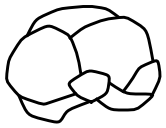
Kpar_Na

Kpar_Na(), the medium weight function, is the same as kpar_He() except it accepts the stack size and allocates the stack with rmalloc(), returning the pointer to the stack (similar to kpar_Fe()).

```
int kpar_Na(void func(), int stksize,
     int nparms, parm1, ...);
```

Kpar Weaknesses

A substantial weakness in the parallel thread package of Trollius 2.2 is termination and memory reclamation. In each of the three native process creation functions, the parent process is responsible for releasing the space allocated for the child's stack and possibly data segment. It is not easy to learn when the child process has terminated. Helium and sodium processes



kpar_He(func, pstack, nparms, parm1, parm2, ...);
Create a helium weight process.

kpar_Na(func, stksize, nparms, parm1, parm2, ...);
Create a sodium weight process.

kpar_Fe(func, stksize, pid, nparms, parm1, parm2, ...);
Create an iron weight process.

Local Node Parallel Functions

Trollius supplies three ITB functions that allow parallel execution of a function within a program on the local node. In order of increasing overhead and versatility they are kpar_He(), kpar_Na() and kpar_Fe(). They are considered kernel level functions (hence they start with a k), but they do not involve the kernel itself. Actual process creation is provided by the underlying environment[1]. A process created by a kpar function is called a native process.

The code for the child process is supplied as a C function in the parent process's program. The function can accept arguments, but cannot return a value. Other functions may be called by the child process.When the child process returns from its "main" function, it terminates. The parent and child process share the same text (code) segment; it is not copied.

Parallel processes within a node can create multiple simultaneous synchronization points that may improve robustness. Trollius itself is heavily process structured for the sake of clarity, flexibility and ease of maintenance. Parallel processes within a node do not impact performance because the processor must be time shared.

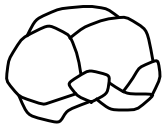
Kpar_Fe

Kpar_Fe(), the heavyweight function, creates a very independent native process capable of attaching to the kernel with kinit() and becoming a Trollius process. It is the recommended method of created long lived processes.

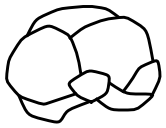
```
char *kpar_Fe(void func(), int stksize, int *pid,
     int nparms, int parm1, ...);
```

Kpar_Fe() accepts the name (address) of the main function to begin execution in the child process, the stack size in bytes, a pointer to the value

1. On transputer based ITB nodes, the underlying environment is provided in hardware. Process creation is a hardware instruction. Peer process destruction is very difficult.



```
main()
{
     int            fd;
     int            global_len;
     int            local_len;

/*
 * Open the file first with Cubix singl method.
 * This is not an error handling tutorial.
 */
     fd = open("data", O_RDONLY | O_SINGL, 0);

/*
 * Read the global (total) length of the array.
 */
     (void) read(fd, (char *) &global_len,
          sizeof(int));

/*
 * Switch to Cubix multiple method.
 */
     (void) multi(fd);

/*
 * Calculate the local length, allocate enough
 * space and read the local subset of the data.
 */
     local_len = global_len / getnitb();
     data = (float *) malloc((unsigned) (local_len
          * sizeof(float)));
     nread = read(fd, (char *) data,
          local_len * sizeof(float));

/*
 * Close the file
 */
     (void) close(fd);
}
```



right subset of data and in recomposing a data structure during write so that the data structure is not jumbled.

Without Cubix file access, an application often needs a controlling program to manage the parallel processes and filter I/O. Cubix can eliminate the need for a control program. The non-Cubix default access method is also called the asynchronous method (async). Without synchronization, a message written by N nodes appears N times on the terminal. A decomposed data structure written to a file appears in a random order.

The rforder() function changes the default order of client data transfer in the Cubix multiple method. Each client specifies a unique sequence number from 0 to N-1, where N is the number of clients (ITB nodes)[1].

```
int rforder(int fd, int sequence);
```

A file is opened with a Cubix access method by using one of the special open flags, O_SINGL or O_MULTI. The access method of a file can be changed with the functions listed below. In each case, one client process on each ITB node must call the same function. When going from asynchronous to a Cubix access method, all async file descriptors must match in pathname and non-Cubix open flags. More commonly, applications will need to switch a file between single and multiple access methods. See the manual page for cubix for more information.

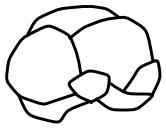
singl	Change file descriptor to single access method.
multi	Change file descriptor to multiple access method.
async	Change file descriptor to asynchronous access method.

We present a simple example program that reads the total size of a one dimensional array from a file, then reads its subset of the array based on the total number of ITB nodes.

Cubix Example

```
/*
 * Read and decompose a 1-D array of reals
 * across a 1-D array of ITB nodes.
 * Assume the array length decomposes evenly
 * across the number of ITB nodes.
 */
static float        *data;
```

1. Cubix places no restriction on the assignment of nodeids in the boot schema.



Some Trollius specific features of remote file access are controlled by additional flags in the flags argument of the open() function. These flags are listed below.

O_LOCK Lock the file descriptor into the remote file server's open descriptor cache. See the manual page rfunix().

O_SINGL Open the file using the single access method. See *Parallel File Access*.

O_MULTI Open the file using the multiple access method. See *Parallel File Access*.

O_REUSE Reuse existing open file descriptor for matching pathname and open flags if found. This is useful for asynchronous access to one open file with one file pointer.

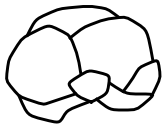
O_1WAY Write to the file without waiting for completion or return code. This greatly increases write performance but should only be used on a debugged application.

Parallel File Access

Trollius supports a loosely synchronous, parallel I/O access method based on Cal Tech's Cubix. One process on every ITB node must participate in Cubix access method functions. A file function is not completed until all client processes have called the same file function. There are two different Cubix access methods that solve two common file read/write problems in data parallel programming.

single All clients execute the same file function with the same amount of identical data. The data from only one (arbitrary) client is transferred. This is useful when all clients want to read a global value from a file, or write a global value to a file. It is especially convenient during I/O to a terminal. The user enters data once and it is read by all nodes. All nodes print a prompt or an error message and it appears once on the terminal.

multiple All clients execute the same file function with different amounts of different data. All the data from all the clients is transferred, but the order of transfer is strictly controlled. By default, the lowest value nodeid will transfer first and the sequence continues until the highest value nodeid transfers last. This is useful in decomposing a data structure during read so that the right nodes get the



`open(path, flags, mode);`	Open a file.
`close(fd);`	Close a file.
`read(fd, buffer, length);`	Read from a file.
`write(fd, buffer, length);`	Write to a file.
`lseek(fd, offset, whence);`	Change position in a file.
`access(path, mode);`	Check permissions of a file.
`mkdir(path, mode);`	Create directory.
`chdir(path);`	Change working directory.
`fstat(fd, buf);`	Get status on file descriptor.
`stat(path, buf);`	Get status on named file.
`dup(fd);`	Duplicate file descriptor.
`dup2(oldd, newd);`	Duplicate & place file descriptor.
`system(string);`	Issue a shell command.
`rmdir(path);`	Remove a directory.
`unlink(path);`	Remove a file.
`getwd(path);`	Get working directory.
`ftruncate(fd, length);`	Set length of file descriptor.
`truncate(path, length);`	Set length of named file.
`singl(fd);`	Switch to single access method.
`multi(fd);`	Switch to multiple access method.
`async(fd);`	Switch to async access method.
`rforder(fd, sequence);`	Change multiple node ordering.

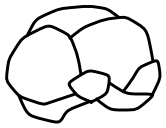
File Access

The OTB node in a Trollius multicomputer allows remote access to its file system via a standard UNIX interface. ITB nodes with a disk and a file system may also provide remote access in the same way. Trollius does not provide a file system, only remote access to a file system from any node.

Building upon the UNIX file functions, C programmers also have access to the C standard I/O library which provides for formatting and buffering. Neither level of I/O will be described here. File pathnames refer to files on the OTB node's operating system by default. However, a specific nodeid can be attached to a pathname with the following syntax:

```
nodeid:path
```

Each Trollius process may have a maximum of 32 simultaneously open file descriptors. Remember that all Trollius UNIX file functions involve message passing using the same links, buffers and other resources used by an application.



tireel from a multicast. As messages meet at forks in the spanning tree, they are combined into one message and sent on towards the top of the tree, now the destination node. The multireel messages are combined pair-wise until all messages arrive at a fork node and are combined. (The fork node itself may be a destination node.) The method by which two messages are combined into one message is defined by the user. The system supplies several simple built-in combining methods which are associated with a castid during the rcast() function. The reelfunc parameter to rcast() is set to one of the values listed below and defined in <castreq.h>.

CBN_ERR	combine error codes in first integer only
CBN_ADD	combine by adding all corresponding integers
CBN_MIN	combine by taking the minimum
CBN_MAX	combine by taking the maximum
CBN_AND	combine by applying a bitwise AND
CBN_OR	combine by applying a bitwise OR
CBN_XOR	combine by applying a bitwise XOR

A separate process can combine multireeled messages. In this case the reelfunc parameter to rcast() is set to an event used by the combining process. Combining can be done by a single application process, given the number of branches entering the local node. This important value is returned by the rcask() function. For a detailed discussion of multireel combining strategies, see the manual page for rcast().

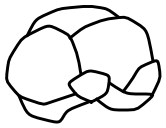
```
int rcask(int nodeid, int castid);
int rmelt(int nodeid, int castid);
int rrnodewait(int nodeid, int castid);
```

A multireel should not be attempted before the castid is defined. The rrnodewait() function can be used by the multireel source nodes to ensure that a castid is defined. When a castid is no longer needed, the spanning tree information can be freed with the rmelt() function.

Caveats

The 2.2 multireel service has caveats.

- Avoid multiple simultaneous multireels on the same castid and the same event.
- Reeled messages must all be the same length.
- Reeled messages may not use nh_data[6].



rcast(nodeid, castid, reelfunc, pcast, ncast);
Register a castid.

rmelt(nodeid, castid);
Remove a castid.

rcask(nodeid, castid);
Get number of branches at given node.

rrnodewait(nodeid, castid);
Wait for a castid to be defined.

Multicasting and Multireeling

Multicasting allows a message to be sent to a group of nodes, using a single call to the network sub-level function, nsend(). The reverse procedure, multireeling, receives a message from a group of nodes. The group of nodes representing the multicast destination must be registered under a cast identifier (castid). The castid can then be used instead of a nodeid in the nh_node field of the network message descriptor during a call to nsend().

The rcast() function handles castid registration.

```
int rcast(int nodeid, int castid, int reelfunc,
     int *pcast, int ncast);
```

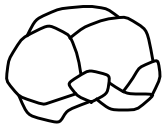
A castid defines not only the destination nodes, but also the source node. The first argument defines the source node. Though the registration can be initiated from any node, it is usually done from the multicast source node and the first argument may be given as LOCAL.

A castid exists within the domain of nodeids and must be unique among all nodeids in the Trollius multicomputer and all other castids. With this restriction, the user may choose any unused identifier.

The pcast argument points to an array of nodeids of length ncast. These are the destination nodes of the castid.

During registration, the system defines a spanning tree across the node network, beginning at the source node and reaching every destination node. The system remembers this spanning tree and uses it every time a message is sent to the castid from the source node. At each fork in the tree, the message is copied and sent to all branches.

Multireeling is more complicated. The leaves of the spanning tree become source nodes send a message to the castid. They must set the NREEL option in the nh_flags field of the network message descriptor to distinguish a mul-



bours only, and only after a network virtual circuit has been established between the two nodes (see *Network Virtual Circuits*). When physical sub-level communication is completed, the circuit must be terminated at the datalink or network sub-level.

```
int psend(struct nmsg *nhead);
int precv(struct nmsg *nhead);
```

nh_dl_event As with the datalink sub-level, the sending process uses this field to identify the output link. The receiving process does not use this field.

nh_node This field is not used at the physical sub-level.

nh_event The receiving process maintains this field with the value used to synchronize the virtual circuit. The sending process does not use this field.

nh_type This field is not used at the physical sub-level.

nh_flags This field is not used at the physical sub-level.

nh_length There is no protection against sender/receiver length mismatch at the physical sub-level.

Unlike the other network level functions, psend() and precv() do not handle the truncation that may be needed when the sending process specifies a length greater than that specified by the receiving process. If the processes do not specify equal lengths, they will likely hang.

Basic Network Communication

A special pair of functions provide a simplified interface to the network sub-level. Bsend() and brecv() are most appropriate for the novice Trollius programmer. These "basic" functions hide the event and the type synchronization points, leaving the programmer to concentrate on nodeids only. Only one process per node using bsend() and brecv() is allowed.

```
int bsend(struct nmsg *nhead);
int brecv(struct nmsg *nhead);
```

nh_dl_event Ignore this field.

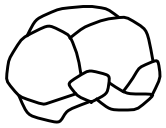
nh_node This is the destination node for bsend() and the source node for brecv().

nh_event Ignore this field

nh_type Ignore this field.

nh_flags The conversion flags indicate the data element type.

nh_length This is expressed as the number of data elements.



be changed with option flags in the nh_flags field of the sender's network message descriptor. The following flags, defined in <net.h>, each assume a data type and make the appropriate change in data representation. They will have no effect if no data conversion is needed.

DINTDATA	nh_data holds integers
DFLTDATA	nh_data holds floats
DDBLDATA	nh_data holds doubles
DRAWDATA	nh_data representation will not be changed
DINTMSG	nh_msg holds integers
DFLTMSG	nh_msg holds floats
DDBLMSG	nh_msg holds doubles
DRAWMSG	nh_msg representation will not be changed

If the data pouch or message body contain data of mixed types, you will need to convert each type separately. Trollius supplies macros that convert data between the local representation (regardless of what it is) and a standard (Trollius) representation. Data should be converted to Trollius representation before sending, and converted to local representation after receiving. You will need to use the macros appropriate to the given data type. The macros are defined in <t_types.h>.

ltot	converts integer representation from local to Trollius
ttol	converts integer representation from Trollius to local
ltots	converts short representation from local to Trollius
ttols	converts short representation from Trollius to local
ltotf	converts float representation from local to Trollius
ttolf	converts float representation from Trollius to local
ltotd	converts double representation from local to Trollius
ttold	converts double representation from Trollius to local

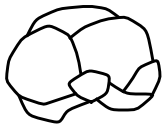
The syntax for all the above macros is of the form:

```
cvar = macro(var);
```

where cvar and var are of the appropriate data type for the given macro. Var is the original value and cvar is the value converted to the appropriate data representation.

Physical Sub-level

The physical sub-level functions psend() and precv() directly access hardware communication links. They may be used between ITB node neigh-

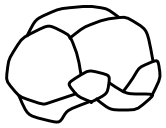


```
/*
 * The first message opens the virtual circuit.
 * We are separating this message from the
 * body messages for easy conversion to the
 * physical sub-level.
 */
     if (drecv(&incoming) < 0) {
          kexit(errno);
     }
/*
 * Receive the body messages.
 */
     for (i = 0; i < 8; ++i) {
          if (drecv(&incoming) < 0) {
                    kexit(errno);
          }
     }
/*
 * The last message closes the virtual circuit.
 */
     incoming.nh_flags = 0;
     if (drecv(&incoming) < 0) {
          kexit(errno);
     }
/*
 * Detach from Trollius kernel.
 */
     kexit(0);
}
```

Data Representation and Conversion

Message data, excepting text, must be converted to the proper machine representation and byte order when transmitted between nodes with different processors. The datalink sub-level can make the necessary conversions when a message crosses the link between differing nodes. The typical multicomputer may have a need for conversion between the OTB node and the ITB nodes. Data representation conversion is also available at the network and transport sub-levels.

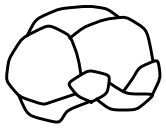
By default, the data pouch (nh_data) of the network message descriptor is assumed to hold integer data, and the data representation is converted to the appropriate integer byte ordering during network message passing. The message body (nh_msg) is not altered by default. The default behavior can



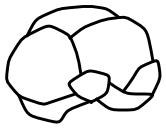
```
/*
 * The last message closes the virtual circuit.
 */
     outgoing.nh_flags = 0;
     if (dsend(&outgoing) < 0) {
          kexit(errno);
     }
/*
 * Detach from Trollius kernel.
 */
     kexit(0);
}
```

The receive program must match the send program exactly on KHOLD synchronization. A mistake here will cause the program to hang.

```
/*
 * Receive 10 messages using a virtual circuit.
 * The receiver does not specify the input link.
 */
#include <net.h>
extern int          errno;
main()
{
     int            i;
     struct nmsg    incoming;
     char           buffer[80];
/*
 * Attach to Trollius kernel.
 */
     if (kinit(0) < 0) {
          exit(errno);
     }
/*
 * Fill message descriptor.
 */
     incoming.nh_event = 11;
     incoming.nh_type = 0;
     incoming.nh_length = sizeof(buffer);
     incoming.nh_flags = KHOLD;
     incoming.nh_msg = buffer;
```



```
#include <net.h>
extern int          errno;
main()
{
     int            i;
     struct nmsg    outgoing;
     char           buffer[80];
/*
 * Attach to Trollius kernel.
 */
     if (kinit(0) < 0) {
          exit(errno);
     }
/*
 * Fill message descriptor.
 */
     outgoing.nh_dl_event = dl_event(1);
     outgoing.nh_node = LOCAL;
     outgoing.nh_event = 11;
     outgoing.nh_type = 0;
     outgoing.nh_length = sizeof(buffer);
     outgoing.nh_flags = KHOLD;
     outgoing.nh_msg = buffer;
/*
 * The first message opens the virtual circuit.
 * We are separating this message from the
 * body messages for easy conversion to the
 * physical sub-level.
 */
     if (dsend(&outgoing) < 0) {
          kexit(errno);
     }
/*
 * Send the body messages.
 */
     for (i = 0; i < 8; ++i) {
          if (dsend(&outgoing) < 0) {
                    kexit(errno);
          }
     }
```



network message buffering speeds up this problem by replacing the local memory with buffer workers as the critical resource[1]. A safe receiving process can receive any of its message classes at any time. A really safe receiving process buffers its messages internally. You can rely on Trollius buffers if your receiving process switches between only a few unique event/type combinations. Avoid using large numbers of event/type combinations or a deadlock will be your reward.

Network Virtual Circuits

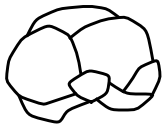
Network virtual circuits are a feature of the datalink sub-level that is also available to the network and transport sub-levels. A circuit can be created between any two processes in the multicomputer. The most common use is at the datalink sub-level between processes residing on neighbour nodes. The circuits offer substantially reduced overhead over several messages at the expense of multiple process access to the communication link. When a sending process creates a network virtual circuit to a neighbour node, the connecting communication output link is effectively reserved to the sending process. No other process can use the output link, including system processes. A receiving process reserves the communication input link in the same way. A circuit that goes beyond the neighbour reserves all the links along the route. Multiple simultaneous and not carefully planned remote circuits can quickly lead to deadlock. Concentrate on using neighbour circuits, but be careful to turn them off before the process exits, to allow the system to resume using the links.

The rules for network virtual circuit creation and destruction are the same as for the kernel level (see *Kernel Virtual Circuits*). They are activated by the same option flag, KHOLD. Unlike the kernel level, network virtual circuits are unidirectional. Another difference is that network virtual circuits check and handle sender/receiver length mismatch.

In this subsection we present an example of datalink sub-level virtual circuit programming. The send and receive programs are close reflections of each other.

```
/*
 * Send 10 messages out link 1 using a virtual
 * circuit.
 */
```

1. The reward for this buffer process structure is that network message buffering remains outside the kernel and very flexible at runtime. The kernel remains ignorant of the network.



Let us take a closer look at the buffer process. There are really two types of processes at work. The buffer daemon collects all messages. The buffer daemon forwards messages to a limited number of buffer workers. Messages that cannot immediately be forwarded to a buffer worker are kept inside the buffer daemons internal pool of message buffers. Each buffer worker can only buffer one message. The buffer worker sends its message to the original destination, as if the buffer detour had not been made.

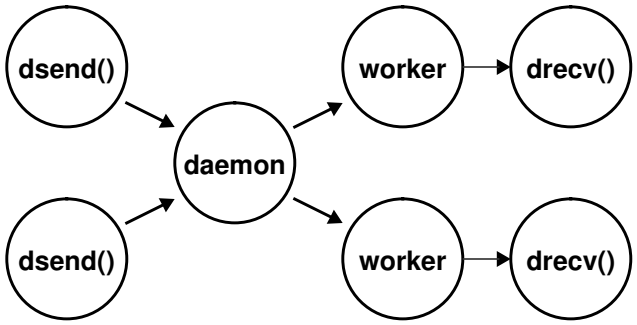
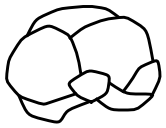


Figure 3: Buffer Daemon and Buffer Worker

Figure 3 illustrates the buffer daemon/worker structure. Notice that if a message passes through a buffer, it must pass through a worker to reach the receiving process. If the workers are full and cannot proceed, the message is stuck in the daemon's pool and the receiving process is permanently blocked. This is a deadlock.

It is the buffer daemon's policy to not have two workers busy with two messages of identical event and message type. This sets up a potential hazard for receiving processes that use several events and/or several types with restrictive matching. Such processes are effectively state machines. They synchronize on one event/type and then, perhaps dependent on the input data, synchronize on another event/type. This is a bad communication structure for the distributed memory architecture, but it is commonly used by application programmers. What happens if the local node's memory is filled by messages the receiver won't receive until another message event/type has arrived? The other message can't arrive because the memory is full. Trollius

1. An exception to this rule is the optimized datalink output process that runs on most ITB transputer links (dlo_t4). This process cannot hold a buffered message.



Network Message Buffers

Message buffering is a separate, modular feature of the network level directly accessed by the datalink sub-level but also accessible to the transport and network sub-levels. Without special option flags in the nh_flags field of the network message descriptor, buffers will be used when available and needed. Two option flags in nh_flags limit a message's buffer use.

NOBUF	The message will never use buffers.
BYBUF	The message will not use buffers on the local node.

If these options are left out of nh_flags, the message may use a buffer anywhere along the route to the destination node, though it is not required. Passing through a buffer constitutes a large overhead for message passing. If an application can guarantee that buffers will not be needed, the NOBUF option is recommended Only the sending functions use these options.

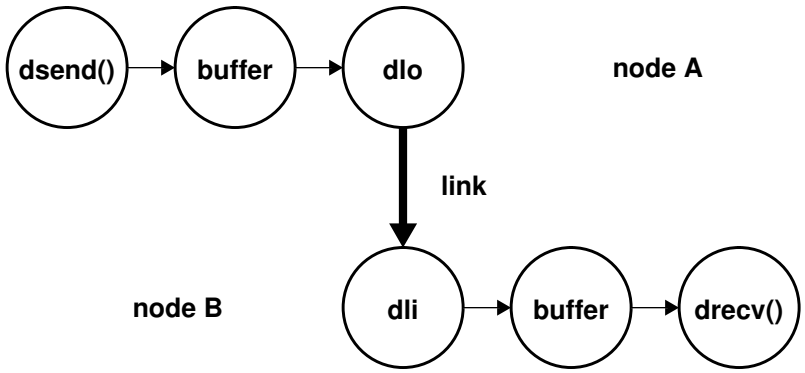
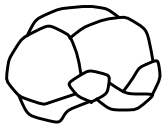


Figure 2: Network Message Buffering

The datalink sub-level message transfer, depicted in Figure 2, can involve buffers on the sending and receiving nodes. The BYBUF option locks out the first buffer opportunity in the dsend() to drecv() path but not the second. The NOBUF option locks out both buffer opportunities. Figure2 also shows the special Trollius system processes that own and operate communication links between nodes. They are called datalink processes and dlo is shorthand for datalink output; dli is shorthand for datalink input. The importance of these processes to the application programmer is that they each may hold one buffered message.[1]



cuits as part of the datalink sub-level, although they may also be accessed from the network and, in the case of buffering, the transport sub-levels.

The datalink sub-level functions are dsend() and drecv(). Drecv() is used in exactly the same way as nrecv(). To use dsend(), the sending process explicitly names the output link on the local node in the nh_dl_event field of the network message descriptor. There are several ways to obtain the correct link information.

- After using nsend() to send a message to a neighbour node, nh_dl_event contains the correct link information to use dsend().
- A link number, in the range 0 to L - 1, where L is the number of links on a node is converted by the macro dl_event() to the information needed by dsend(). The macro is defined in <net.h>.
- The nodeid of the neighbour node is converted by the function, getroute(), to the information needed by dsend(). The extra event parameter is returned by getroute() when the nodeid parameter is the local nodeid. This enables dsend() to send a message to a receiving process on the local node.

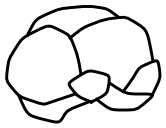
```
int dl_event(int link);
int getroute(int nodeid, int event);

int dsend(struct nmsg *nhead);
int drecv(struct nmsg *nhead);
int dtry_send(struct nmsg *nhead);
int dtry_recv(struct nmsg *nhead);
```

nh_dl_event This field must be set by the sending process using dsend(). It holds a value associated with the output link connecting the destination neighbour node. It can also hold the event of a local receiving process. This field is not used by drecv().

nh_node This field should be set to the nodeid of the destination neighbour node. It can also be set to LOCAL. This field is not used by drecv().

nh_flags KHOLD, NOBUF, BYBUF, DRAWDATA, DINTDATA, DFLTDATA, DDBLDATA, DRAWMSG, DINTMSG, DDBLMSG

nh_length There is no packetization at the datalink sub-level. Message length is limited to MAXNMSGLEN.



etization. In the 2.2 release, long message programmers must restrict their boot schema node identifiers to 24 bits and must not run more than 256 processes on a single node. This allows Trollius to express a unique process in the multicomputer as a single event and ensure, through a user transparent protocol, that packets from two senders synchronizing with one receiver on the same event/type do not get mixed. Expect this nodeid, process limit to be removed in a later release.

```
int nsend(struct nmsg *nhead);
int nrecv(struct nmsg *nhead);
int ntry_send(struct nmsg *nhead);
int ntry_recv(struct nmsg *nhead);
```

nh_dl_event This field contains information on the forwarding communication link after calling nsend(). It effectively prepares the network message descriptor to be used by the datalink sub-level. This field is ignored by nrecv().

nh_flags KHOLD, NOBUF, BYBUF, NREEL, DRAWDATA, DINTDATA, DFLTDATA, DDBLDATA, DRAWMSG, DINTMSG, DDBLMSG

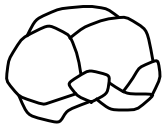
The non-blocking ntry_recv() function synchronizes very much like the ktry_recv() kernel level function. However, the ntry_send() function may behave quite differently due to the presence of message buffers between the sending and receiving processes. Ntry_send() can be successful even if the receiving process has not called nrecv() because the measure of success is only the ability to get rid of the message. If a free buffer exists, it stores the message and ntry_send() returns a 0.

The network sub-level is used by most applications and by most Trollius system programs. No program should attempt the use of the lower sub-levels without first demonstrating correct operation with this sub-level. If the overhead of this sub-level is acceptable, the user need not look any further.

Datalink Sub-level

Successful parallel applications rely on localized communication for performance and scalability. Much communication may be restricted to the immediate (nearest) neighbours of a node. If the packetization service of the network sub-level is not needed, the datalink functions dsend() and drecv() may be used for nearest neighbour message passing.

Much of the general functionality of the network level is provided by the datalink sub-level. In particular, we will discuss buffering and virtual cir-



Transport Sub-level

The transport sub-level functions tsend() and trecv() ensure process synchronization and flow control. A transport sub-level message will not be sent until the intended receiver has called trecv(). This prevents the sender from swamping the message buffers, or jamming the communication links, with a large number of messages. Synchronization is provided at the price of lower performance. Non-blocking functions are not available at the transport sub-level.

```
int tsend(struct nmsg *nhead);
int trecv(struct nmsg *nhead);
```

nh_dl_event Ignore this field.

nh_flags NOBUF, BYBUF, DRAWDATA, DINTDATA, DFLTDATA, DDBLDATA, DRAWMSG, DINTMSG, DDBLMSG

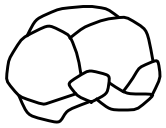
The best application of the transport sub-level is debugging communication. Virtual circuits (see *Network Virtual Circuits*) are not available at this sub-level.

Network Sub-level

The network sub-level is the most common form of Trollius message passing. It does not have the flow control of the transport sub-level, but it has everything else. The chief features are routing and long message packetization.

The network sub-level functions are nsend() and nrecv(). Nsend() can send a message to any node in the multicomputer. It uses the nh_node field to determine which of the local node's communication links to use in transmitting the message. If nh_node matches the local nodeid or nh_node holds the special nodeid, LOCAL, the message is sent to the local process matching the message event and type. This will happen eventually when the message arrives at the destination node. The nrecv() function does not distinguish between the source nodes of multiple incoming messages. The nh_node field is unused by nrecv() and the first message that matches the event and type is received.

The maximum length of a network message is defined by MAXNMSGLEN in <net.h> and is usually set at 4096 bytes. Longer messages are broken up into packets that respect this upper bound. The decomposition takes place inside nsend() and the recomposition is done by nrecv(). All the packets have the same node specified in the network message descriptor. Apart from the network sub-level runtime functions, Trollius is very ignorant of pack-



```
struct nmsg {
     int  nh_dl_event;   /* forwarding event */
     int  nh_node;       /* destination node */
     int  nh_event;      /* message event */
     int  nh_type;       /* message type */
     int  nh_length;     /* message length */
     int  nh_flags;      /* options */
     int  nh_data[8];    /* data pouch */
     char *nh_msg;       /* message buffer */
};
```

The network message descriptor is used generally in the same way by each sub-level. We will discuss the general use here and then point out peculiar usage during a discussion of each sub-level.

nh_dl_event This field is used by the sending process at the datalink and physical sub-levels to indicate the event of the forwarding process. Usually, this is the event of a datalink output process, the proprietor of a communication link.

nh_node The sending process places the destination nodeid into this field. The receiving process does not use this field. The receiving process cannot specify the sending node. Node filtering must be accomplished with the nh_event and/or nh_type fields. This is very important.

nh_event This is the synchronization event at the destination node. The usage is exactly the same as the kernel level.

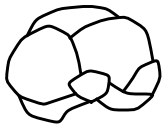
nh_type This is the synchronization type at the destination node. The usage is exactly the same as the kernel level.

nh_length The rules for message length and minimum transfer are the same as the kernel level. The difference is that the sender in not informed of the length received.

nh_flags This is a bit field of options, similar to the kernel level. Indeed, the kernel level KHOLD flag can be used.

nh_msg The usage is exactly the same as the kernel level.

The next subsections will discuss each network message passing sub-level, in order of decreasing functionality and overhead. This is also the order that would be employed in successive optimizations of an appropriate application program.



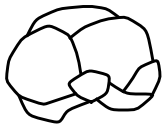
`tsend(&nhead);`	Send transport message.
`trecv(&nhead);`	Receive transport message.
`nsend(&nhead);`	Send network message.
`nrecv(&nhead);`	Receive network message.
`ntry_send(&nhead);`	Send non-blocking network message.
`ntry_recv(&nhead);`	Receive non-blocking network message.
`dsend(&nhead);`	Send datalink message.
`drecv(&nhead);`	Receive datalink message.
`dtry_send(&nhead);`	Send non-blocking datalink message.
`dtry_recv(&nhead);`	Receive non-blocking datalink message.
`psend(&nhead);`	Send physical message.
`precv(&nhead);`	Receive physical message

Network Message Passing

There are four sub-levels of message passing within the Trollius network level, representing different functionality/overhead compromises. They are, in order of increasing functionality and overhead, the physical, datalink, network and transport sub-levels. The primary feature of all these sub-levels is the ability to send a message to a remote (non-local) node. The ITB physical sub-level utilizes hardware links directly and efficiently. The datalink sub-level offers multiple process arbitration of a link, but has no built-in routing capabilities. The network sub-level adds routing and large message packetization to the functionality of the datalink sub-level. The transport sub-level manages message flow control; a message will not be transmitted until the receiving process is ready. The network sub-level functions are the most commonly used, most generic communication functions in Trollius.

The argument passed to all the network level functions is the address of a network message descriptor, labelled nmsg, which is an extension of the kernel level message structure used by ksend() and krecv(). The nmsg structure is defined in <net.h>, which must be included in all programs that call network level functions. The fields in the structure must be set prior to calling the message passing functions. Some fields will be altered upon return of a function.

Network message passing can be as simple or powerful and complex as the user desires. By sticking to the network sub-level and ignoring half the fields in the network message descriptor, the user will be dumb, happy and productive. Performance and fine control require more understanding of the operating system and even the hardware.

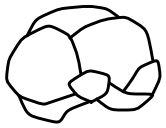


established is associated with the event[1] and is used for the passing of the **next** message with that event. The virtual circuit remains open until a message is passed on that event with the KHOLD bit of both k_flags fields reset to 0. While open, the ksend() and krecv() functions will use the virtual circuit in a manner transparent to the user. Virtual circuits do not support non-blocking synchronization.

A virtual circuit is bidirectional, and there can be up to 16 simultaneous virtual circuits for each process. The process can continue to send messages on non-circuit events; it will synchronize through the kernel for these events as usual. Events associated with virtual circuits must differ in the last four bits.

After the first message establishing the virtual circuit, there is no protection against length mismatch. The underlying environment may cause the processes to fail (crash or return an error) if the lengths differ. Above all, virtual circuits should always be used with caution.

1. The type is not used in identifying an existing virtual circuit. It is used, as normal, in synchronizing the establishment of a new virtual circuit.



```
struct kmsg {
     int  k_event;        /* message event */
     int  k_type;         /* message type */
     int  k_length;       /* message length */
     int  k_flags;        /* options */
     char *k_msg;         /* message buffer */
}
```

The fields of the kmsg structure must be filled before calling the message passing function. Some fields will be changed when the function returns.

The k_event field holds the event, a positive integer chosen by the user. The events specified by the sending and receiving processes must be equal. Negative events are reserved for system processes.

The k_type field is also used to match sending and receiving processes. This is an integer, used as a array of bits. If the type field is set to 0, the message can synchronize with any process matched on the event. If it is nonzero, a matching process must have at least one bit set in a corresponding position in its k_type field. In other words, the bitwise logical AND (*&*) of the two type fields must not be 0. This field is set to the sender's type after successfully returning from krecv() or ktry_recv().

The k_length field specifies the length, in bytes, of the message to be sent, or the size of the receiving buffer. If the sender and receiver specify different lengths, the smaller amount will be transferred. This field is set to the amount transferred after calling any message passing function that returns successfully (synchronizes without error).

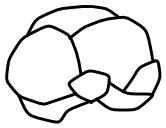
The k_flags field is normally set to 0. It can be set to KHOLD (defined in <kmsg.h>) to establish a virtual circuit (see *Kernel Virtual Circuits*).

The k_msg field is set to the address of the buffer holding the data to be sent, or the address into which the data is to be received.

Kernel Virtual Circuits

Two processes, using ksend() and krecv(), can establish a circuit that will remain in effect for the transmission of several messages. These messages (except for the first one) will synchronize directly with the process at the other end of the circuit, ignoring the event and type and bypassing the kernel, thus decreasing the overhead of message passing.

This is accomplished by setting the k_flags field in the kmsg structure to KHOLD in both the sending and receiving process. The virtual circuit thus



`ksend(&khead);`	Send a local message.
`krecv(&khead);`	Receive a local message.
`ktry_send(&khead);`	Try to send a local message.
`ktry_recv(&khead);`	Try to receive a local message.

Kernel Message Passing

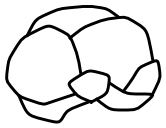
There are two levels of message passing in Trollius. The kernel, or local level allows communication between processes on the same node. The network level allows communication between processes on different nodes, as well as between processes on the same node. In fact, the network level is constructed solely with kernel level message passing. This section deals with the kernel level.

A process sending a message does not directly specify the pid of the receiving process. Instead, each process specifies an arbitrary 32 bit quantity called an event and an arbitrary 32 bit quantity called a message type. As an implicit, hidden part of the message passing procedure, each process sends a message (using the underlying environment) to the kernel. If the events and types “match”, the kernel sends reply messages to both parties. The reply message contains the information necessary for the sending process to send a message to the receiving process directly using the underlying environment. From the user’s point of view, two processes synchronize and a message is passed.

The kernel offers a non-blocking variation of the send and receive functions. Blocking functions do not return until synchronization and data transfer has occurred. Non-blocking functions return an error if synchronization cannot take place immediately. From these rules we can deduce that on the local node where only one process can run at any one time, two processes will never synchronize if they both use non-blocking functions. A non-blocking function that did not synchronize returns -1 and sets errno to EWOULDBLOCK.

The blocking functions are ksend() and krecv(). The non-blocking functions are ktry_send() and ktry_recv(). All four functions take a single parameter, a pointer to a kernel message descriptor, defined in <kmsg.h>

```
int ksend(struct kmsg *khead);
int krecv(struct kmsg *khead);
int ktry_send(struct kmsg *khead);
int ktry_recv(struct kmsg *head);
```



Node Types

The node type is a bit mask containing one or more attributes. The attributes are defined in <net.h> and are listed below.

NT_OTB	OTB node
NT_ITB	ITB node
NT_CAST	multicast node
NT_WASTE	node not available to user
NT_DISK	node with attached disk
NT_TUBE	node with attached video
NT_JONES	neighbour node
NT_BOOT	child node in boot tree

The type of any node is returned by the getrtype() function.

```
int getrtype(int nodeid);
```

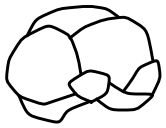
It is often useful to count the number of nodes matching certain node type attributes. The getntype() function requires the caller to give a node type and a node type mask indicating which bits in the node type are to be considered[1]. Other functions of the form getn*() are simply wrappers for getntype(). Getncast() counts all cast identifiers whose spanning tree passes through the local node (see *Multicasting and Multireeling*).

```
int getntype(int type, int mask);
int getnitb();
int getnotb();
int getnall();
int getncast();
```

Once you know the count of nodes matching a certain node type, you may wish to know the actual matching nodeids. A similar function, getnodes(), takes the same parameters as getntype() with the same usage in addition to a nodeid array pointer and size.

```
int getnodes(int *pnodes, int nnodes, int type,
     int mask);
```

1. Getntype(type, mask) counts a node type, nt, if ((nt & mask) == (type & mask)).



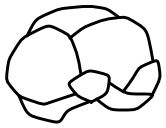
`getnodeid();`	Get the local nodeid.
`getnodetype();`	Get the local node type.
`getrtype(nodeid);`	Get the type of a remote node.
`getorigin();`	Get the origin OTB nodeid.
`getnitb();`	Get total number of ITB nodes.
`getnotb();`	Get total number of OTB nodes (always 1).
`getnall();`	Get total number of nodes.
`getncast();`	Get total number of castids at local node.
`getnet(&net);`	Get comprehensive network information.
`getntype(type, mask);`	Get total number of nodes of specific type.
`getnodes(pnodes, nnodes, type, mask);`	Get nodeids of specific type.
`ORIGIN`	getorigin()
`LOCAL`	special identifier for the local node

Network and Node Information

It is often necessary, especially for message passing, to identify specific nodes in the booted system. Identifiers for the nodes are established at boot time, based on information stored in a special file called a boot schema. Node identifiers may vary from schema to schema. The most commonly used nodeids are the local nodeid and the origin OTB nodeid. These two values, in addition to the local node type, form the net_info structure. The getnet() function fills this structure with information from the local route daemon.

```
int getnodeid();
int getorigin();
int getnodetype();
int getnet(struct net_info *net);

struct net_info {
     int  ni_nodeid;
     int  ni_nodetype;
     int  ni_origin;
}
```

The first three functions call getnet(), cache the information and return the appropriate field from the net_info structure. Subsequent calls simply return the value from the cache. The special nodeid LOCAL can be used to indicate the local node without knowing its actual nodeid. The macro ORIGIN is always replaced by getorigin(). These are defined in <net.h>.



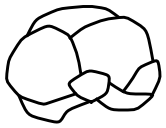
Controlled and Uncontrolled Scheduling

Kernel scheduling is a confusing topic that is best avoided by always attaching processes at priority 0. This puts the process into a category called uncontrolled. Scheduling uncontrolled processes is very simple. When a process is not blocked on a message, it receives an immediate reply from the kernel and then it is up to the underlying environment[1] to schedule the process. We have no control over the underlying scheduling policy and sometimes it is inadequate for the basic process structuring needs of an application; it may be inadequate for some Trollius system processes.

We occasionally need something more like real-time scheduling: a multi-priority environment where the highest priority unblocked process runs until it blocks (with no time-slicing). The Trollius kernel provides such scheduling to processes with non-zero priorities, called controlled processes. Only the highest priority unblocked controlled process receives a reply from the kernel and is allowed to be scheduled by the underlying environment. There may be other, lower priority unblocked processes (which would be released if they were uncontrolled) but they are not allowed to run. Eventually, the active controlled process will block and the kernel will release the next highest priority unblocked process. A caveat for real-time applications is that there is currently no pre-emption capability.

Unfortunately, there is an opportunity for further complication. When a controlled process synchronizes with an uncontrolled process, we have learned through experience that it is highly desirable to impede the uncontrolled process until the controlled process is released and then blocks. In other words, the uncontrolled process becomes temporarily controlled. See the manual page for kinit() for further discussion of this topic.

1. The underlying environment is a lower level kernel providing some form of multi-process scheduling and interprocess communication. Transputers provide this in hardware. Other ITB microprocessors run a software sub-kernel. On Trollius OTB nodes, UNIX provides the scheduling and IPC. It is important to understand that the Trollius kernel is a server and not an executive. It is scheduled along with its clients by the underlying environment.



`kinit(priority);`	Initialize data structures and kattach().
`kattach(priority);`	Attach to kernel.
`kdetach(0);`	Detach from kernel.
`kexit(errno);`	Call clean functions, kdetach() and terminate.
`_kexit(errno);`	Kdetach() and terminate.
`atkexit(func, arg);`	Register cleanup function at kexit().

Initialization and Termination

Before Trollius functions may be called, a process must initialize Trollius related data structures and attach to the Trollius kernel. Both of these functions are accomplished by kinit(). The lone parameter is the priority, which will be discussed in detail later in this section. In most cases, the priority should be set to 0.

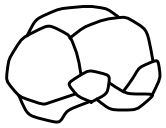
```
int kinit(int priority);
int kattach(0);
```

Most ITB programs call kinit(0) prior to entering main(). If priority 0 is satisfactory, ITB programs need not call kinit(), though it is recommended that they do, for portability. Kinit() calls kattach() to register with the kernel. After kattach() other kernel requests may be made. Kdetach() says good-bye to the kernel, but the program continues to run. A process on a Trollius node need not be a Trollius process (though ITB processes automatically start out that way). A process can kdetach() at any time, go about Trollius independent business, kattach() again and call Trollius functions. Passing 0 to kdetach() means “detach myself”. An ITB process must begin with kinit() if the process was started with kpar_Fe() (see *Local Node Parallel Functions*).

Trollius processes can terminate by calling kexit(). Before the process dies, it will call kdetach(). The parameter passed to kexit() is an error code with 0 indicating that no error occurred. ITB programs will call kexit(0) after returning from main().

```
int kdetach(int pid);
int kexit(int errno);
int _kexit(int errno);
int atkexit(int (*func)(), char *arg);
```

Before detaching from the kernel, kexit() will call all the user and system registered clean-up functions in the reverse order of registration. Registration is done with the atkexit() function. If the process wishes to exit without calling clean-up functions, it can do so with _kexit().



- in the state of a process blocked on sending a message
- in the state of a process blocked on receiving a message
- in a network (remote node) message

Equal events synchronize. The event has an associate, subordinate object called the type which is found wherever the event is found. It is the secondary synchronization point between two local processes. It is irrelevant if the event does not synchronize. It is irrelevant if it is set to zero. Non-zero types that share a common one bit (bitwise AND is non-zero) synchronize.

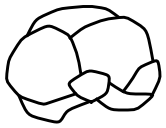
C Programs under Trollius

A C program that will run under Trollius is the same as a serial C program in terms of variable types, statements, control flow, and functions. Program structure is virtually the same. A C program running under Trollius is simply a C program that makes use of the special Trollius functions for message passing, process creation, etc.

Writing an individual Trollius program is thus very similar to writing a serial program. An extra level of complexity is added, naturally, in designing processes that will run in parallel. Programmers must deal with decomposition, data alignment, synchronization and load balancing. These topics are beyond the scope of this reference. We can state, however, that Trollius can be used to implement many different models of parallel programming from data domain decomposition to neural networks.

Return Values and Error Reporting

Unless stated otherwise, Trollius functions return 0 upon successful execution and -1 if an error was encountered. In the latter case only, the global system variable errno is set to the appropriate error code. Trollius error codes are listed in *Appendix B: Error Codes*.



When running Trollius, we refer to ITB or OTB computers as nodes. Very often we make no node type distinction when describing functionality - because there is none. Unless the node type is specified, assume the function applies generally to any node.

Figure 1 represents the broad Trollius multicomputer.

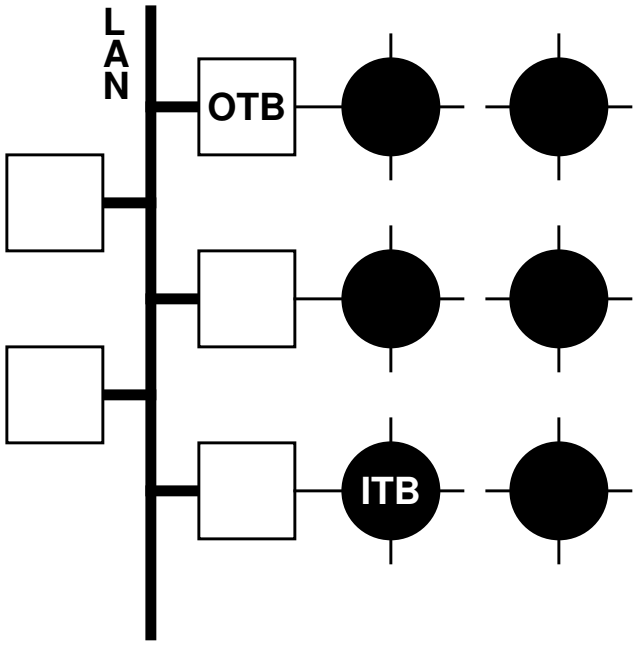


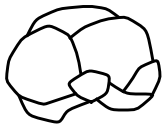
Figure 1: Trollius Multicomputer Architecture

Basic Objects

Trollius does not restrict the interconnection topology of nodes. A globally unique 32 bit identifier is given to each node (nodeid), chosen by the user. Many Trollius functions take the nodeid as their first parameter. The node is the only global object in Trollius. The two other important objects are purely local to each node. They are the process and the event.

Every Trollius node has a multi-tasking, message passing local environment upon which the rest of the operating system is built. There is a small kernel at each node that manages and synchronizes any number of local processes. Each process has a locally unique 32 bit identifier (pid). Many Trollius functions, though not message passing, operate on pids.

The event is an arbitrary 32 bit quantity and the key synchronization point for two local processes. You will see an event in three places.



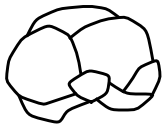
Introduction

Trollius presents a broad view of the multicomputer architecture. It recognizes that any computer reachable via communication link(s) is a potential participant in a message passing application. The software presents a uniform programming environment across general purpose computers (e.g.: workstations) and dedicated highly integrated computer ensembles. The latter is usually singled out and tagged as the "parallel machine" and is packaged into an impressive looking box. In Trollius, we refer to these computers as Inside The Box computers or ITB. We recognize the following characteristics of ITB computers:

- They are dedicated to single users.
- All incoming and outgoing links are dedicated to single users.
- They are great in number.
- They are going to be used for large parallel computational problems and not the common applications that UNIX workstations do well.
- On average, they have few peripherals.
- They will run Trollius natively.

Since Trollius is not going to provide all the services of a general purpose operating system, the ITB computers must be hosted by a computer with such an operating system. The host computer will have one or more communication links to one or more ITB computers. As indicated above, other general purpose computers may participate in a multicomputer if the communication links exist such that all computers of both types are reachable (directly or indirectly by hopping through intermediate computers). In most cases, general purpose computers, be they workstations or mainframes, are reachable via a Local Area Network (LAN). In Trollius, we refer to general purpose computers as Outside The Box computers or OTB. We recognize the following characteristics of OTB computers:

- They are shared among multiple users.
- Links to other OTB computers are shared among multiple users.
- They may control peripherals needed by a parallel application.
- They will run Trollius as an application of the native, general purpose operating system.



Rloadgo 39
Rspawn 40
Rpwait 40
Rpattach 41
Process Runtime Flags 41
Summary of Process Terminology 41
Monitoring and Controlling Kernel Processes 42
Signals 42
Monitoring and Controlling Remote Processes 44
Rstate 44
Rdoom 46
Memory Allocation 47
Data Visualization 48
LED Control and Monitoring 50
Trace Generation 52
Miscellaneous Functions 53
Transputer Hardware Techniques 54
Appendix A: Library Organization 55
UNIX Filesystem (libt) 55
Cubix Filesystem (libt) 55
General OS Interface (libt) 56
Kernel Level (libt) 56
Network Level (libt) 57
Data Representation (libt) 58
Transputer Specific (libt) 58
Remote Level (libt) 59
Remote Level (libtrillium) 59
Remote Level (libbuckeye) 61
Appendix B: Error Codes 62

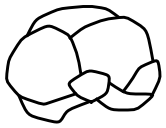


Table of Contents

How to Use This Document 2

Introduction 5

- Basic Objects 6
- C Programs under Trollius 7
- Return Values and Error Reporting 7

Initialization and Termination 8

- Controlled and Uncontrolled Scheduling 9

Network and Node Information 10

- Node Types 11

Kernel Message Passing 12

- Kernel Virtual Circuits 13

Network Message Passing 15

- Transport Sub-level 17
- Network Sub-level 17
- Datalink Sub-level 18
- Network Message Buffers 20
- Network Virtual Circuits 22
- Data Representation and Conversion 25
- Physical Sub-level 26
- Basic Network Communication 27

Multicasting and Multireeling 28

- Caveats 29

File Access 30

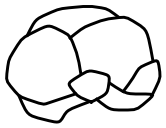
- Parallel File Access 31
- Cubix Example 32

Local Node Parallel Functions 34

- Kpar_Fe 34
- Kpar_He 35
- Kpar_Na 35
- Kpar Weaknesses 35
- Kpar_Fe Example 36

Remote Node Parallel Programs 38

- Rload 38
- Rcreate 39



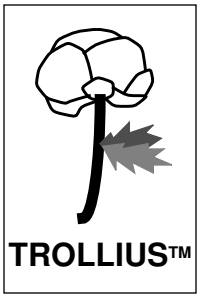
How to Use This Document

This document is a Trollius programmer's reference describing all user level functions. Extra emphasis is placed on the critical aspects of multicomputer parallel programming: message passing, process management and file access.

This document is user oriented. It does not give much insight into how the system is implemented. It does not detail every option and capability of every function. The manual pages cover all the functions in great detail and are meant to supplement this document.

We have kept the font and syntax conventions to a minimum.

`code`	This font is used for things you type on the keyboard or see printed on the screen. We use it in code sections and tables but not in the main text.
Section	Italics are used to cross reference another section in the document or another document.



Trollius Library Reference in C

Trollius is a programming environment for message passing MIMD parallel computers (multicomputers). This document covers all aspects of Trollius C programming for application programmers. It is not a complete reference for system programmers. New users are referred to the *Trollius Tutorial Introduction in C*. *Trollius Command Reference* describes how to operate a Trollius multicomputer. A version of this document is available for Fortran programmers.

Ohio Supercomputer Center

Research Computing
The Ohio State University

Document Series 2/2 **March 29, 1992**