Transputer 'C' Library Description

Version 91.1
11/15/91

Contents

Introduction

Overview

There are two general classes of library functions provided with the **Transputer Toolset**:

1. Standard "C" library routines. These include most of the functions in the proposed ANSI "C" standard, plus many of the routines which were common before it (in what is sometimes called "classic" "C").

2. Transputer specific routines. These allow access to the rich set of communication and parallel processing primitives which the Transputer microcode supports. Many of these functions are optionally supported by the compiler as "inline" functions to take full advantage of the instruction set.

The following two sections summarize both classes of functions, and provide general comments concerning use with the **Transputer Toolset**. After this, a section is devoted to a series of increasingly complex example programs. Finally, the last section of the manual contains detailed individual descriptions of the library functions (presented in alphabetic order).

Normal "C" Library Functions

Character Classification and Mapping

The following functions are provided:

isalnum	Test for alphanumeric "char"
isalpha	Test for alphabetic "char"
isascii	Test for ASCII "char"
iscntrl	Test for control "char"
isdigit	Test for decimal digit "char"
isgraph	Test for printable "char"
islower	Test for lower case "char"
isprint	Test for printable "char" and space
ispunct	Test for punctuation "char"
isspace	Test for whitespace "char"
isupper	Test for upper case "char"
isxdigit	Test for hexadecimal digit "char"
toascii	Convert "char" to ASCII (0x00-0x7F)
tolower	Convert upper case "char" to lower
toupper	Convert lower case "char" to upper

Numeric Conversion

The following functions are provided (floating point conversions are only available for the 32 bit Transputers):

atof	Convert string to "double"
atoi	Convert string to "int"
atol	Convert string to "long"
frexp/frexpf	Convert "double/float" to fraction & exp.
ldexp/ldexpf	Compose "double/float"
modf/modff	Decompose "double/float"
strtod	Convert string to "double"
strtol	Convert string to "long"
strtoul	Convert string to "unsigned long"

Math

The following functions are provided for 32 bit Transputers:

abs	Compute "int" absolute value
acos/acosf	Compute "double/float" arc-cosine
asin/asinf	Compute "double/float" arc-sine
atan/atanf	Compute "double/float" arc-tangent
atan2/atan2f	Compute "double/float" arc-tangent
cabs/cabsf	Compute "double/float" complex abs. value
ceil/ceilf	Compute "double/float" ceiling
cos/cosf	Compute "double/float" cosine
cosh/coshf	Compute "double/float" hyperbolic cosine
div	Compute "int" division/remainder
exp/expf	Compute "double/float" exponential
fabs/fabsf	Compute "double/float" absolute value
fft/fftf	Compute "double/float" FFT transform
floor/floorf	Compute "double/float" floor
fmod/fmodf	Compute "double/float" remainder
frand/frandf	Compute "double/float" random number
hypot/hypotf	Compute "double/float" Euclidean distance
ifft/ifftf	Compute "double/float" inverse FFT
labs	Compute "long" absolute value
ldiv	Compute "long int" division/remainder
log/logf	Compute "double/float" natural log
log10/log10f	Compute "double/float" log base 10
modf/modff	Compute truncated "double/float"
pow/powf	Compute "double/float" x^y
rand	Compute "int" random number
sin/sinf	Compute "double/float" sine
sinh/sinhf	Compute "double/float" hyberbolic sine
sqrt/sqrtf	Compute "double/float" square root
srand	Set "rand/frand/frandf" random seed
tan/tanf	Compute "double/float" tangent
tanh/tanhf	Compute "double/float" hyperbolic tangent

Floating point library functions come in two flavors:

1. The normal "double" precision routines. These take "double" arguments and return a "double" result.

1. A complementary set of "single" precision routines. These take "float" arguments and return a "float" result. They are differ from the "double" versions by having a "f" suffix added to the routine name (ANSI convention). Thus, both "sin" and "sinf" routines exist in the library.

In order to get correct results with the "sinf" function, the "math.h" include file must be used so that a ANSI prototype is in scope indicating that the argument to "sinf" will really be a "float" (instead of the normal default promotion to "double").

Some of the single precision routines are only provided for floating-point Transputers.

Emulated Math on Transputers

When compiling floating point code, the TCX "C" compiler has been optimized for use with Transputers with floating point hardware support. When used to generate floating point code for other types of 32 bit Transputers, TCX generates "emulation" instructions which correspond to the instructions found on floating point Transputers. These instructions are converted into function calls by TASM and invoke a simulation package included in the library. The simulation package can handle all the floating point code generated by TCX with one exception:

Floating point Transputers automatically save the floating point registers when a context switch occurs from low to high priority. Since there is no way the simulation package can tell when this occurs, it doesn't have a way of simulating the register save operation. Thus, if you use floating point code in both a low AND a high priority process whose execution may be interleaved, you must inform the simulation when to save the registers. This is accomplished by calling the "savefp" function from each high priority process which uses floating point PRIOR to any floating point code getting executed. You pass "savefp" a pointer to a structure of type "FPstate" (defined in "conc.h"), which will be used to store the current contents of the simulated floating point registers. Similarly, you must call "restorefp" with the same pointer just PRIOR to finishing the high priority process (after all floating point operations have been completed), to restore the previous values of the registers.

The simulation package has been carefully constructed such that the context switches which occur as a result of the timeslicing of low priority processes can't cause corruption of the simulated registers. Thus, you don't need to call "savefp" and "restorefp" if all of your floating point is in processes with the same priority.

Heap Management

The following functions are supported:

addfree	Add memory to heap
calloc	Allocate and init memory from heap
cfree	Return a block of memory to heap
free	Return a block of memory to heap
malloc	Allocate memory from heap
realloc	Change size of allocated memory

Because of the possibility of destroying the "C" heap, the dynamic memory routines have the appropriate critical sections protected (via "ProcToHigh"/"ProcToLow" function calls). This allows low and high priority processes to coexist without heap concurrency battles.

Heap Placement and Size

The location and size of the heap can be configured in the following two fashions:

1. By default, the heap is set to begin immediately after the highest addressed code or data in the Transputer "C" program. This placement is performed automatically by TLNK since the starting label is defined in a high numbered load module (#254, see the TLNK documentation for more information about this facility). The default heap size is set to 128K bytes (32K bytes on 16 bit Transputers). You may change the size setting by modifying the contents of the "env.c" library file, or you may explicitly set the heap end address by modifying the "_heapend" pointer (defined in "env.c"), to be whatever the desired ending address is. This pointer should be modified prior to any use of the "C" heap and should be set to a word aligned address (bottom two bits should be zero for a 32 bit Transputer). If you don't wish to use this heap allocation method, simply set "_heapend" to MostNeg (0x80000000 if 32 bit Transputer, 0x8000 if 16 bit), and no heap memory will be allocated from this area.

Note that you must NOT set "_heapend" to MostNeg if you are planning on using **Software Virtual Channels**. The initialization performed by the "_vcmain" or "_ns_vcmain" entry point functions requires a small amount of heap to be available prior to the user-provided "main" function being called.

2. In addition to the above automatic source of heap memory, you may manually add a region of memory to the heap using the "addfree" function. This function may be used in place of the above facility (if you've set "_heapend" to MostNeg), or both techniques may be used in combination. You may also add more than one additional region using multiple calls to "addfree", but each region thus added is disjoint (even if they are adjacent), and they may not be later combined into a single, larger, region.

String Manipulation

The following functions are supported:

bcmp	Compare memory regions for equality
bcopy	Copy memory region (unsafe if overlap)
bzero	Initialize memory region to zero
index	Find first occurrence of "char" in string
memccpy	Copy memory region to a certain "char"
memchr	Search memory region for certain "char"
memcmp	Compare memory regions
memcpy	Copy memory region (unsafe if overlap)
memmove	Copy memory region (safe if overlap)
memset	Set memory region to "char" value
rindex	Find last occurrence of "char" in string
strcat	String concatenation
strchr	Find first occurrence of "char" in string
strcmp	Compare strings
strcpy	Copy string
strcspn	Find length of string not in "char" set
strerror	Map "errno" error # to error message
strlen	Find string length
strncat	String concatenation with length
strncmp	Compare strings with length
strncpy	Copy string with length
strpbrk	Find first "char" from set in string
strrchr	Find last occurrence of "char" in string
strspn	Find length of "char" set in string
strstr	Find substring in string
strtok	Split string into tokens

Input/Output

The following functions are supported:

clearerr	Clear stream error & EOF flags
close	Close file
creat	Create new file
dup	Duplicate file handle
dup2	Duplicate file handle
exit	Close files and exit program
fclose	Close stream
fcloseall	Close all streams
fdopen	Convert handle to stream
feof	Check for EOF on stream
ferror	Check for error on stream
fflush	Flush stream buffer
fgetc	Read character from stream
fgets	Read line from stream
fileno	Convert stream to handle
fopen	Open stream
fprintf	Formatted write to stream
fputc	Write character to stream
fputs	Write line to stream
fread	Block read from stream
freopen	Redirect stream
fscanf	Formatted read from stream
fseek	Change read/write position in stream
ftell	Report read/write position in stream
fwrite	Block write to stream
getc	Read character from stream
getch	Read character from console without echo
getchar	Read character from "stdin" stream
getche	Read character from console with echo
gets	Read line from "stdin" stream
kbhit	Check if console key pressed
lseek	Change read/write position in file
open	Open file
printf	Formatted write to "stdout" stream
putc	Write character to stream
putchar	Write character to "stdout" stream
puts	Write line to "stdout" stream
read	Block read from file
remove	Delete file
rename	Rename file
rewind	Rewind stream
scanf	Formatted read from "stdin" stream
sprintf	Formatted write to memory
sscanf	Formatted read from memory
tmpfile	Create temporary file
tmpnam	Create unique file name
ungetc	Push back character to stream
unlink	Delete file
vfprintf	Variable arg. formatted write to stream

vprintf	Variable arg. formatted write to "stdout"
vsprintf	Variable arg. formatted write to memory
write	Block write to file
_ns_exit	Non-server replacement for "exit"
_ns_main	Non-server replacement for "_main"
_ns_printf	Non-server replacement for "_printf"
_printf	Simple version of "printf"

All the "C" library I/O functions are implemented via the "C" remote I/O protocol (you must use the "cio.exe" I/O server when loading the program). Starting with **Transputer Toolset** Version 91.1 the I/O functions may be also be used "simultaneously" by multiple processes on the root node. To make this possible the remote I/O protocol code is now protected from concurrent access using a semaphore. See the CIO documentation for more information about this package.

Thanks to the addition of **Software Virtual Channels** to the **Transputer Toolset**, you may also optionally perform "C" I/O from nodes other than the root. Simply select the appropriate virtual channel entry point when linking the program, tell LD-NET you will be using virtual channels in the appropriate "Network Information File" (file extension ".nif"), and you are ready to go. See the next section(s), and the **Transputer Virtual Channel Communcations** section for more detailed information.

If you plan on sharing files among nodes in your network the CIO documentation should be consulted for details about concurrent file I/O.

User additions to the provided set of I/O functions are supported using the CIOEXT server extension facility. See the "_cioext" library function description and the CIO documentation for more information.

Alternatives to "_main"

As many of you know, "_main" is a traditional name for the initial entry address for a "C" program. The "_main" entry point function is charged with performing whatever initialization/interpretation a particular host environment requires in order to provide a proper environment for the user "C" program. When finished, "_main" passes control to the the user supplied "main" function for execution of the application code.

The **Transputer Toolset** also uses a "_main" approach to program startup. By default, TLNK will link in a "_main" function which performs the following actions:

A. Sets up the "_node_number", "_boot_chan_in", "_boot_chan_out", "_host_chan_in", "_host_chan_out", "_HostIn" and "_HostOut" external variable values. See the **Transputer Miscellaneous** section for information about what these variables hold.

B. Sets up the "argc"/"argv" command line arguments.

C. Calls the user supplied "main" function.

D. If returned to, calls "exit".

As the **Transputer Toolset** can be used in a variety of ways, using the default "_main" entry point may not always be appropriate. For example, "_main" assumes you will only be performing "C" library I/O calls on the root node using the CIO I/O server. It also assumes you will not be using **Software Virtual Channels**. If one of these assumptions is incorrect you will probably want to use one of the provided alternatives to "_main":

"_ns_main"

This entry point function performs the same general operations as "_main", EXCEPT, no host I/O is performed. Since no host I/O is assumed, this function must dummy up the "argc"/"argv" command line arguments as it can't actually find out what they might have been. Why should you use this entry point when "_main" can do more?

A. If you will not be using the CIO host I/O server, or the node that this program will be loaded onto isn't the root node, you can't perform any standard "C" library I/O anyway (at least without **Software Virtual Channels** which we will discuss in the next two sections). If you link a program with the default "_main" entry point, and run it on the root node, it will immediately attempt to contact CIO to get the command line arguments. If CIO isn't present the Transputer program will simply hang!

B. Since "_main" uses CIO to get the host command line arguments, it requires that the Transputer program include the code for communicating with CIO. While this is automatically performed for you, it also requires additional program memory space. If you are running a small program on a Transputer configured without external memory this "overhead" code may be very undesirable!

For both these reasons, "_ns_main", has been provided. Assuming no other "C library I/O functions are called within the application program, this will obviate the need for CIO. In conjunction with "_ns_main", two other functions are provided ("_ns_exit" and "_ns_printf"), which provide replacements for "exit" and "_printf", respectively, when running without CIO. The output from "_ns_printf()" is sent over the "bootstrap" link as straight ASCII text and may be displayed using a simple I/O server such as TIO.

Of course, you may wish to use both "_main" and "_ns_main" on different programs in a network of Transputers. Using the default "_main" for the root node and "_ns_main" for the other nodes allows the root node to perform host I/O while the remote nodes gain the benefits of smaller program size. Prior to the introduction of **Software Virtual Channels** this was almost a mandatory configuration! Note that you could always use link communication from remote nodes to the root node and then allow the root to do the actual host I/O. This sort of configuration is also the approach used by most other standalone "C" development packages that compete with the **Transputer Toolset**.

If you wish to use the "_ns_main" entry point, and are using the TCC command driver to compile/link your program, add a "-e _ns_main" directive to the TCC command line. See the TCC documentation for more detailed information.

"_vcmain"

This entry point function corresponds to "_main", except it also supports **Software Virtual Channels**. Use of either this entry point, or "_ns_vcmain" described in the next section, is ESSENTIAL if virtual channels are to be used on a particular node! This entry point may be used with programs linked for all nodes in a Transputer network. Using this entry point has the advantage that it allows all nodes in the network to get a copy of the command line arguments, and to perform host I/O. It also permits host I/O requests to co-exist with user application virtual channel communication, all of which use the same basic physical links. Finally, it permits various sorts of network utilities such as debuggers and profilers to transparently access any node in the network. Of course, with all these advantages, there are some drawbacks:

A. This entry point causes the creation of a link "sentinel" input process for every physical link which is used by any user or host I/O virtual channel. It also causes the allocation of some heap memory to store routing information and virtual channel control information. This overhead makes fitting an application program on a Transputer without external memory somewhat more difficult. In addition, since a more complex multi-threaded protocol must be used on all physical channels (particularly including that between the root node and the host), there is more runtime overhead involved in host I/O for the root node compared to a root node program linked with the default "_main" entry point.

Note that a MAJOR amount of effort has been expended in the design and implementation of the **Software Virtual Channel** facility to minimize both the memory overhead and the performance loss, still, until the Transputers which handle virtual channels in hardware are available, you pay a penalty! See the **Transputer Virtual Channel Communications** section for related information.

B. In general, if you wish to use virtual channels (or non-root host I/O), on any node on the network, you must link all the programs to be run on the network with either the "_vcmain" or the "_ns_vcmain" entry points. See the next section for information about "_ns_vcmain". This all-or-nothing behavior is required since if a node wishes to communicate with the host, or another node to which it isn't directly connected, the messages must be relayed through a set of intermediate nodes. These intermediate nodes thus must also be capable of supporting virtual channels.

There are ways to work around this: You can tell LD-NET that a particular node will not be doing any host I/O, and that ALL the physical links associated with the node should be reserved for "raw" (non-virtual channel protocol), use. If you decide to do this you can link the program for that node with the "_ns_main" entry point as it will not require any of the virtual channel stuff. You can also link a program with either "_vcmain" or "_ns_vcmain" so that you still have host I/O access, and yet at the same time have LD-NET reserve some of the physical links connected to the node for "raw" use. These options allow an advanced network application which uses virtual channels where convenient, and direct physical link usage were maximum throughput is needed! Note, of course, that since telling LD-NET that a link should be reserved for "raw" usage constrains it from using it during the mapping of host I/O and user virtual channels to the physical links in the network, doing so may reduce the performance of some virtual channel communication paths.

If you wish to use the "_vcmain" entry point, and are using the TCC command driver to compile/link your program, add a "-e _vcmain" directive to the TCC command line. See the TCC documentation for more detailed information.

"_ns_vcmain"

The "_ns_vcmain" entry point is like "_vcmain", except no host I/O is provided for. This entry point is analogous to "_ns_main" in the sense that it assumes the program will need no host I/O and can thus save the overhead of having to link in the CIO protocol code. As with "_ns_main", the command line arguments are dummied up. As mentioned in the previous section, you can freely mix programs using the "_ns_vcmain" and the "_vcmain" entry points in a network using **Software Virtual Channels**. If you wish to use virtual channels, but do not require host I/O (ie. CIO), you can link ALL the programs in the network with "_ns_vcmain"! One drawback to using "_ns_vcmain" is that since it has no provisions for host I/O, nodes running programs linked with "_ns_vcmain" may not be interrogated by host-based network analysis tools such as debuggers or profilers.

If you wish to use the "_ns_vcmain" entry point, and are using the TCC command driver to compile/link your program, add a "-e _ns_vcmain" directive to the TCC command line. See the TCC documentation for more detailed information.

Miscellaneous

The following miscellaneous functions are provided (host time related functions are only available for the 32 bit Transputers):

asctime	Format "broken-down" time
ctime	Format local time
errno	System error variable
getenv	Get environment variable value
gmtime	Convert time to "broken-down" UTC time
isort	Insertion sort
localtime	Convert time to "broken-down" local time
longjmp	Non-local jump
perror	Write "errno" message to "stderr" stream
qsort	Quick sort
setjmp	Register environment for non-local jump
ssort	Shell sort
system	Invoke host command processor
time	Read system calendar time
va_arg	Access processed variable argument
va_end	Terminate variable argument processing
va_start	Initialize variable argument processing
_cioext	Use server extension facility

Transputer Specific Library Functions

Introduction

A collection of functions is included in the library which implements a Transputer-oriented concurrency and interface library. The specification of many of the routines in this library is from a paper titled "Processes, Channels and Semaphores (Version 2)" written by Jeffrey Mock of Pixar. A copy of this paper is included in the documentation package by permission of the author. Several notes about the paper and associated routines are in order:

1. The implementation of the "SemP" and "SemV" semaphore routines in the library is modeled after the version Jeffrey did and doesn't support mixing high and low priority processes. Another set of routines (named "HSemP" and "HSemV"), is available as an extension to work with mixed priority tasks (although they are slower than the mono-priority versions).

2. The include file specified in the paper has been changed from "con.h" to "conc.h" for the **Transputer Toolset** implementation.

3. When Jeff wrote his paper ANSI "C" wasn't a major issue. The **Transputer Toolset** implementation of his library has been updated to use "void *" pointers for the communication I/O buffer pointers and to use "void" as the return type for functions which are to be run in parallel. These changes are consistent with what other vendors using Jeff's approach have done.

The concurrency model which Jeffrey implemented is often called "OCCAM-like"; since he defined routines which allow many of the control structures and process models, supported by OCCAM, to be used with "C" programs. In addition to this paradigm, the **Transputer Toolset** library contains routines which support a "Fork/Join" concurrency model. The inclusion of both approaches, together with the excellent hardware support the Transputer provides for concurrent programming in general, makes for simple and efficient creation of programs containing many independent processes.

Transputer Channel Communication

The following functions are provided:

ChanAlloc	Dynamically allocate a channel
ChanFree	Free a dynamically allocated channel
ChanIn	Read message from a channel
ChanInChanFail	Read message from channel with aux. reset
ChanInChar	Read byte from a channel
ChanInInt	Read word from a channel
ChanInTimeFail	Read message from channel with timeout
ChanOut	Write message to channel
ChanOutChanFail	Write message to channel with aux. reset
ChanOutChar	Write byte to channel
ChanOutInt	Write word to channel
ChanOutTimeFail	Write message to channel with timeout
ChanReset	Reset channel

Transputer Virtual Channel Communication

The following functions are provided:

VChan	Get a virtual channel pointer
VChanIn	Read message from a virtual channel
VChanInChar	Read byte from a virtual channel
VChanInInt	Read word from a virtual channel
VChanOut	Write message to virtual channel
VChanOutChar	Write byte to virtual channel
VChanOutInt	Write word to virtual channel
VChanVIn	Read variable length from virtual channel
VChanVOut	Write variable length to virtual channel

These functions provide a user program interface to either the **Software Virtual Channel** facility, for networks of existing T2/T4/T8 processors, or the underlying hardware virtual channel mechanism provided with upcoming Transputers. The concept of a "virtual channel" comes from the next generation of Transputers designed by INMOS. The appropriate INMOS documentation should be consulted for information about the motivation and philosophy behind the design of virtual channels.

The basic concept is that of a full duplex communication path between any two processes, on any two processors, in a Transputer network. Details of message protocol and routing are completely hidden by the hardware and software implementations, the user merely reads and writes messages as if to a regular "soft" channel. A further advantage to this approach is that the actual connection topology for virtual channels is determined at network load time, and may be changed between program executions. With correct application program design no recompilation is necessary to adapt to widely differing network topologies.

Although detailed explanations are deferred to the individual function descriptions (and the LD-NET documentation), the following is an outline of how virtual channels work within the **Transputer Toolset** framework:

To configure a virtual channel, you add a channel definition to the "network information file" used by LD-NET to load a particular network. The channel definition specifies which node each end of the channel is connected to, and more particularly, which "logical" channel on each node is to be used. From the perspective of the application program, the "logical" channel number is converted to a virtual channel pointer by a call to "VChan". Given the virtual channel pointer, the program can perform virtual channel I/O using any of the virtual channel communications routines, such as "VChanIn", "VChanOut", etc. Since virtual channels are full duplex, you may have both an input and output process using the same virtual channel, on the same node, at the same time.

"logical" channels are arbitrary, user specified, numbers which can range from 6 to 32767. Note that using smaller numbers helps to minimize memory overhead. Although the "logical" channel number information would normally be compiled into a program image, the information about what is connected to the other end is only specified by the "network information file" (".nif" file), at load time. This allows the easy construction of program modules which don't care about specific network topology. Of course, you could also have the program choose from a selection of configured "logical" channels by, for example, reading the desired "logical" channel number from a configuration file on the host.

In order to actually use virtual channels with the program running on a particular node you must also link that program with a special entry point (either "_vcmain" or "_ns_vcmain"). See the **Alternatives to "_main"** section for more detail on this.

Of course, the **Software Virtual Channel** stuff has been carefully designed to be STRICTLY compatible, at the user program interface level, with the published specifications for the new Transputers which feature hardware virtual channel support. When the appropriate Transputers are available you need merely recompile your existing application code and go! Thus the descriptions of these functions are independent of what type of Transputer your program will be (eventually), running with.

To aid backward compatibility to existing T2/T4/T8 code, the virtual channel functions accept regular "soft" channel pointers in addition to those obtained from the "VChan" function. Since T2/T4/T8 "soft" channels are simplex only, and require non-zero length messages, using them isn't strictly portable to future Transputers. You can obtain a strictly conforming virtual channel, which IS portable, by simply telling LD-NET that both ends of a virtual channel communication are on the same node. Using the "native" soft channels has a performance advantage when the communication is local to a node, so of course, you must decide where to make the tradeoff.

Transputer Channel Status Testing

The following functions are provided:

ProcAlt	Block until input channel is ready
ProcAltList	Block until input channel is ready
ProcSkipAlt	See if input channel is ready
ProcSkipAltList	See if input channel is ready
ProcTimerAlt	Block until channel or timer ready
ProcTimerAltList	Block until channel or timer ready

Transputer Virtual Channel Status Testing

The following functions are provided:

VProcAlt	Block until virtual channel is ready
VProcAltList	Block until virtual channel is ready
VProcSkipAlt	See if virtual channel is ready
VProcSkipAltList	See if virtual channel is ready
VProcTimerAlt	Block until virtual channel/timer ready
VProcTimerAltList	Block until virtual channel/timer ready

These functions provide input alternation facilities for virtual channels. Note that virtual channel communication may occur purely for synchronization purposes, by sending and receiving zero length messages. The various virtual channel "Alt" routines consider a zero length message to be a valid termination condition.

As with the other virtual channel functions, you may mix "soft" and virtual channels together, with the limitation that older Transputers are not specified to operate correctly with zero length messages.

Transputer Concurrency (Jeffrey Mock Model)

The following functions are provided:

ProcAlloc	Dynamically allocate process
ProcFree	Free a dynamically allocated process
ProcInit	Initialize process
ProcPar	Start process(es)
ProcParam	Modify process parameters
ProcParList	Start list of processes
ProcPriPar	Start processes at mixed priorities
ProcRun	Start process at current priority
ProcRunHigh	Start process at high priority
ProcRunLow	Start process at low priority
ProcStop	Kill process
ProcToHigh	Make current process be high priority
ProcToLow	Make current process be low priority
ProcWait	Block for specified time

Transputer Concurrency (Fork/Join Model)

The following functions are provided:

PFork	Fork process at current priority
PForkHigh	Fork process at high priority
PForkInit	Initialize process forking structure
PForkLow	Fork process at low priority
PHalt	Kill process and save state
PJoin	Block until forked processes terminate
PRun	Start process
PSetup	Initialize process structure
PStop	Kill process

These routines are provided with the **Transputer Toolset** for cases where code being ported to the Transputer might already use this flavor of multiprogramming, as well as for new applications for which this model is appropriate. In general, these routines are somewhat lower level than similar functions described in the Jeffrey Mock paper, and have the corresponding benefit of lower overhead.

Note that these routines use the OCCAM notion of what a process descriptor is (ie. a workspace pointer and priority OR'ed together). The "PDes" data structure (defined in file "conc.h"), serves exactly this purpose.

Transputer Semaphore Support

The following functions are supported:

HSemP	Mixed-priority semaphore "P" operation
HSemV	Mixed-priority semaphore "V" operation
SemAlloc	Dynamically allocate semaphore
SemFree	Free a dynamically allocated semaphore
SemP	Mono-priority semaphore "P" operation
SemV	Mono-priority semaphore "V" operation
_HSemP	Primitive for "HSemP"
_HSemV	Primitive for "HSemV"
_SemP	Primitive for "SemP"
_SemV	Primitive for "SemV"

Transputer Timing and Scheduling

The following functions are supported:

GetHiPriQ	Get high priority queue pointers
GetLoPriQ	Get low priority queue pointers
ProcAfter	Block until specified time
ProcCall	Call function with new stack
ProcGetPriority	Determine current process priority
ProcReschedule	Move process to the back of the queue
SetHiPriQ	Initialize high priority queue pointers
SetLoPriQ	Initialize low priority queue pointers
SetTime	Set current priority timer
Time	Read current priority timer

Transputer Miscellaneous

The following functions are supported on Transputers which support the corresponding instructions:

BitCnt	Count bits set in a word
BitRevNBits	Variable bit reversal
BitRevWord	Reverse bits in word
Move2D	2-D block move
Move2DNonZero	2-D (non-zero byte) block move
Move2DZero	2-D (zero byte) block move
restorefp	Restore floating point pseudo-registers
savefp	Save floating point pseudo-registers
_boot_chan_in	Pointer to bootstrap input channel
_boot_chan_out	Pointer to bootstrap output channel
_host_chan_in	Pointer to host input channel
_host_chan_out	Pointer to host output channel
_HostIn	Pointer to host input function
_HostOut	Pointer to host output function
_node_number	Network node address

The "_boot_chan_in" and "_boot_chan_out" variables are declared in "conc.h". They are pointers to the "hard" input and output channels which were used to bootstrap the Transputer node.

The "_host_chan_in" and "_host_chan_out" variables are also declared in "conc.h"; they hold pointers to the channels to use when communicating with the host. If the program has been linked with either "_main", "_ns_main" or "_ns_vcmain", these pointers will simply be copies of "_boot_chan_in" and "_boot_chan_out" (respectively). If the program has been linked with "_vcmain" and loaded with LD-NET configured to allow virtual channels, these variables will hold the virtual channel pointers for reading from, and writing to, the host I/O server (CIO). See the **Alternatives to "_main"** section for more information about the various entry points.

The "_HostIn" and "_HostOut" variables are function pointers which are declared in "conc.h". These variables are set by the various entry point functions to provide the host I/O code in the library with a pointer to the correct functions for either "hard" or virtual channel I/O to the host. The "_HostIn" variable will point to the "ChanIn" function if non-virtual channel host I/O is needed. If virtual channel host I/O is required, "_HostIn" will point to "VChanIn". The "_HostOut" function is used in a similar fashion to point to the appropriate output function, either "ChanOut" or "VChanOut". Note that these variables are not intended for general application program usage as doing so will likely foul up the communication protocol between the host I/O code residing on the Transputer, and CIO running on the host.

The "_node_number" variable is again declared in "conc.h"; it holds the user supplied network address for the node. This variable is often quite useful in application programming as it allows modified runtime program behavior depending on the address of the node. The example programs in the following section take advantage of this feature.

See the LD-ONE and, particularly, the LD-NET documentation for more information about many of these variables.

Examples

Introduction

The examples provided in this section focus on the Transputer-specific aspects of the library. For information about using the standard "C" library routines, the following two books are recommended:

"C" A Reference Manual
Samuel P. Harbison/Guy L. Steele Jr.
Prentice-Hall, Inc.
Englewood Cliffs, NJ 07632

The "C" Programming Language (Second Edition)
Brian W. Kernighan/Dennis M. Ritchie
Prentice-Hall, Inc.
Englewood Cliffs, NJ 07632

Sample Programs

Being traditional, we start with a Transputer version of "Hello World":

```
#include  <stdio.h>
#include  <conc.h>

main()
  {
  printf("Hello World (Node %d)\n",_node_number);
  }
```

This code is provided as "exam1.c" in the example code directory.

The "_node_number" variable holds the network address for the processor which is running the program. See the documentation for the LD-ONE and LD-NET programs, and the **Transputer Miscellaneous** discussion in the previous section, for more information about "_node_number". An external declaration for "_node_number" is contained in "conc.h".

This program should be linked with either "_main" or "_vcmain" as an entry point. "_main" is the default and is appropriate for use with a single processor. "_vcmain" uses the **Software Virtual Channel** facility, in conjunction with LD-NET, to allow the program to be run on an arbitrary number of processors in a network. In either case you should run with CIO as the host I/O server.

We can also get the same result without requiring that the CIO host server be run. Assuming the entry point is changed to "_ns_main", and the TIO program is used to display text written to the Transputer link:

```
#include  <stdio.h>
#include  <conc.h>

main()
  {
  _ns_printf("Hello World (Node %d)\n",_node_number);
  }
```

This code is provided as "exam2.c" in the example code directory.

You will have to manually terminate TIO after the message is displayed (usually by typing control-c). Note that "_ns_printf" stands for NoServer-printf (no server protocol is used). The TIO program writes whatever it reads from the link to "stdout" as straight ASCII. The "exam2.c" program should only be run on the root node.

Suppose we wish to continue to avoid using the CIO server, but want a more powerful print formatting routine than "_ns_printf" (which was designed to use minimal memory, not be full-featured). For example:

```
#include  <stdio.h>
#include  <conc.h>

main()
  {
  char              buf[80];

  sprintf(buf,"Hello World (Node %.4d)\n",_node_number);
  ChanOut(_boot_chan_out,buf,strlen(buf));
  }
```

This code is provided as "exam3.c" in the example code directory.

Again, you will have to manually terminate TIO after the message is displayed. This program should only be run on the root node. Note the use of the "ChanOut" channel communication function, and the predefined channel pointer "_boot_chan_out" (for information about "_boot_chan_out" see the **Transputer Miscellaneous** discussion in the previous section). This example still doesn't require any protocol on the link, but shows how messages of arbitrary complexity may be generated and written to a link. Since CIO will normally be used as the host server, and all the usual "C" primitives may then be used for I/O from the root Transputer, this isn't usually done. However, there is one case where this type of thing comes in very handy:

As most programmers are humans, and humans are generally fallible, they tend to make programming mistakes. Finding these mistakes can be particularly challenging when they occur in a program distributed across a network of processors. If you have a network capable debugger, and the problem is nice enough to not blow away the network communication kernel (no memory protection after all), great! If not, a viable debugging approach is to have available a spare computer, with a link interface, to use as a "probe". This computer runs a copy of something like TIO, and the "probe" link is connected to a spare link on the Transputer node which is having trouble. This allows the "printf" approach to debugging without disturbing the normal I/O or link usage. It does require a spare link, and patience, however! Of course, if you have a network source level debugger, this may seem antiquated, but it does allow you to (eventually), find even "impossible" bugs!

CIO is assumed to be the host I/O server for the rest of the sample programs. Similarly, "_main" or "_vcmain" (as appropriate), is assumed to be the entry point symbol specified to TLNK ("_main" is the default). If you are using the TCC command driver program you specify the "_vcmain" entry point by adding a "-e _vcmain" directive to the TCC command line.

The previous examples have been programs which either run on a single processor, or use the **Software Virtual Channel** facility to "transparently" run on more than one processor. While some Transputer systems fall into this category, many more involve multiple nodes in some type of network where nodes are assigned different "jobs". The following program generalizes the "Hello World" program to run on more than one Transputer. It does so WITHOUT using virtual channels:

```
#include  <stdio.h>
#include  <conc.h>

main()
  {
  Channel           *chan;
  int               id = 0;

  if(_node_number == 1)
    {
/*
 *  We are the root Transputer, inform the user as the
 *  various node addresses arrive.  Note that we listen for
 *  input from the bootstrap link also but this doesn't
 *  matter since it shouldn't generate any output!
 */
    printf("Root node ID # was %d\n",_node_number);
    printf("Booted from input channel: %p\n",_boot_chan_in);
    while(1)
      {
      chan = LINK0IN +
        ProcAlt(LINK0IN,LINK1IN,LINK2IN,LINK3IN,0);
      printf("\nMessage on input channel: %p\n",chan);
      ChanIn(chan,&id,2);
      printf("Message was node ID # %d\n",id);
      }
```

```
    }
  else
    {
/*
 *  We are not the root, look at all input links and send
 *  any node addresses we get out the link we were
 *  bootstrapped on.
 */
    ChanOut(_boot_chan_out,&_node_number,2);
    while(1)
      {
      chan = LINK0IN +
        ProcAlt(LINK0IN,LINK1IN,LINK2IN,LINK3IN,0);
      ChanIn(chan,&id,2);
      ChanOut(_boot_chan_out,&id,2);
      }
    }
  }
```

This code is provided as "exam4.c" in the example code directory. It should be linked with the "_main" entry point.

A copy of this program is loaded and run on every node in the network. The root node then collects the network addresses as they trickle in and sends them to CIO for display on the host system. You will have to manually terminate the execution of CIO when the last message has been displayed since no provision is made in the program to break out of the "infinite" read loop. For information about configuring and loading programs on a network see the LD-NET documentation.

While this program is fairly simple-minded, it does provide an idea of how a more sophisticated program might be constructed for use in a "farm" application. For our purposes, a "farm" application is defined to be one in which the exact topology of the network is ignored, each node (except the root), is considered to be equivalent, and all nodes work on independent pieces of the project. Many simulation and numerical analysis applications fit into the "farm" model fairly well. The following "prime" solver is an example of this type of program (warning: the algorithm used is quite stupid):

```
#include  <stdio.h>
#include  <conc.h>
#define   MAX_PRIME 10000     /* Largest # to test */
#define   NUM_NODES 4         /* # of Transputer nodes */
#define   INCR      (NUM_NODES * 2)     /* Prime interval */

main()
  {
  Channel           *chan;
  unsigned     int  child_data = 0;
  unsigned     int  i,j;
  unsigned     int  num_primes = 0;
/*
 *  Check 1/N of the numbers for primality, where N
 *  is the number of Transputer nodes.
 */
  for(i = (_node_number * 2) + 1; i < MAX_PRIME; i+= INCR)
    {
    for(j = 3; j < i; j++)
      {
      if((i % j) == 0)
        break;
      }
    if(j >= i)
      num_primes++;
    }
/*
 *  Report results back either to user or parent node.
 */
  if(_node_number == 1)       /* Root node */
    {
    num_primes++;             /* Acct for 2 being prime */
    for(i = 1; i < NUM_NODES; i++)
      {
      chan = LINK0IN +
        ProcAlt(LINK0IN,LINK1IN,LINK2IN,LINK3IN,0);
      ChanIn(chan,&child_data,2);
      num_primes += child_data;
      }
    printf("%u primes less than %u found\n",
      num_primes,MAX_PRIME);
    }
  else                        /* Non-root node */
    {
    ChanOut(_boot_chan_out,&num_primes,2);
    while(1)
      {
      chan = LINK0IN +
        ProcAlt(LINK0IN,LINK1IN,LINK2IN,LINK3IN,0);
      ChanIn(chan,&child_data,2);
      ChanOut(_boot_chan_out,&child_data,2);
      }
    }
  }
```

This code is provided as "exam5.c" in the example code directory. It should be linked with the "_main" entry point.

Note that the "NUM_NODES" macro must be set to the number of Transputer nodes which are used. Also, the node numbers (as listed in the ".nif" file), must be consecutive starting with one.

Of course, with a few modifications, the number of processor nodes could be determined at run time instead of compile time. Also, most real "farm" applications require that data be received from the root node (or host), as well as sent to it. Again, the changes are straightforward, but beyond the scope of this (trivial), example.

Although the "farm" model is appropriate for many applications, it is less than ideal for some. A good example is any application which is naturally "pipelined". To get maximum performance with this sort of application it is necessary to make the network topology "fit" the application. One application class which often benefits from this is "digital signal processing" (DSP). The following simple filter application is assumed to be executed on four Transputer nodes connected in a ring. The following ring topology is assumed for this example:

Node 1 is connected to node 2 through link 1.
Node 1 is connected to node 4 through link 2.
Node 2 is connected to node 3 through link 1.
Node 2 is connected to node 1 through link 2.
Node 3 is connected to node 4 through link 1.
Node 3 is connected to node 2 through link 2.
Node 4 is connected to node 1 through link 1.
Node 4 is connected to node 3 through link 2.

This topology is provided in a LD-NET compatible format as "pipe4.nif" in the example code directory.

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <math.h>
#include  <conc.h>

#define   DATA_SETS      100  /* # of data sets to do */
#define   NUM_POINTS     32   /* # of complex data points */
#define   LOG_POINTS     5    /* Log base 2 of NUM_POINTS */
#define   WS_SIZE        8192 /* Workspace size/process */

COMPLEX   in_data[NUM_POINTS];
COMPLEX   out_data[NUM_POINTS];

void
inbuf(Process *p)             /* Write data to pipeline */
  {
  int               i,j;

  for(i = 0; i < DATA_SETS; i++)
    {
    for(j = 0; j < NUM_POINTS; j++)     /* Random */
      {
      out_data[j].real = frand();
      out_data[j].imag = frand();
      }
    ChanOut(LINK1OUT,out_data,sizeof(out_data));
    }
  }

void
outbuf(Process *p)            /* Read data from pipeline */
  {
  int               i;

  for(i = 0; i < DATA_SETS; i++)   /* Read and toss */
    ChanIn(LINK2IN,in_data,sizeof(in_data));
  }

main()
  {
  if(_node_number == 1)       /* Root node */
    {
    Process         *in,*out;

    if((in = ProcAlloc(inbuf,WS_SIZE,0)) == NULL)
      {
      printf("No memory for process 'inbuf'\n");
      exit(1);
      }
    if((out = ProcAlloc(outbuf,WS_SIZE,0)) == NULL)
      {
      printf("No memory for process 'outbuf'\n");
      exit(1);
      }
```

```
    ProcPar(in,out,NULL);     /* Start I/O processes */
    ProcFree(in);             /* Return heap storage */
    ProcFree(out);
    printf("All done\n");
    }
  else if(_node_number == 2)  /* 1st pipeline stage */
    {
    while(1)
      {
      ChanIn(LINK2IN,in_data,sizeof(in_data));
      fft(in_data,LOG_POINTS);
      ChanOut(LINK1OUT,in_data,sizeof(in_data));
      }
    }
  else if(_node_number == 3)  /* 2nd pipeline stage */
    {
    int             i;

    while(1)
      {
      ChanIn(LINK2IN,in_data,sizeof(in_data));
      in_data[0].real = in_data[0].imag = 0.0;    /* DC */
      for(i = 0; i < NUM_POINTS; i++)
        {
        in_data[i].real /= NUM_POINTS;  /* Normalize */
        in_data[i].imag /= NUM_POINTS;
        }
      ChanOut(LINK1OUT,in_data,sizeof(in_data));
      }
    }
  else                        /* 3rd pipeline stage */
    {
    while(1)
      {
      ChanIn(LINK2IN,in_data,sizeof(in_data));
      ifft(in_data,LOG_POINTS);
      ChanOut(LINK1OUT,in_data,sizeof(in_data));
      }
    }
  }
```

This code is provided as "exam6.c" in the example code directory. It should be linked with the "_main" entry point.

This application has two processes running on the root node; one filling the pipeline (with random data), and one removing the resulting data. Each of the three nodes in the pipeline has one process which reads a input data array, performs a computation on it, and writes the result to the next stage.

The first stage in the pipeline performs a forward FFT to convert the (presumably), time series input data into the frequency domain. The second stage normalizes the data and removes the "DC" component. The third stage performs a inverse FFT to transform the data back into the time domain. If you are not familiar with this type of DSP operation, don't worry: the important point is the general organization of the pipeline of processors, and the read/compute/write action at each node.

Note that in this case each node is running the same program and we determine at run time what a particular node should do. This has the disadvantage that each node contains code which it will never use. In larger applications it is common to actually run a physically different program on each node. See the LD-NET documentation for more information on loading programs on networks of Transputers.

Of course, in any real application, some investigation into the execution profile of each program in the "pipeline" should be performed to determine how to distribute the execution tasks across the available processors ("load balancing").

Another point to note is that there is no overlapping between link I/O and computation on the nodes in the pipeline. In this particular case this isn't much of a problem since the FFT operations swamp out the time taken by the I/O (particularly on Transputers without hardware floating point support). For the usual case were I/O is more of a bottle-neck, the following example program shows a skeleton of how you might set up a pipeline application with, and without, buffer processes attached to the input and output links at each stage of the pipe:

First we modify the previous example to stub out the pipeline computation actions:

```
#include  <stdio.h>
#include  <conc.h>

#define   DATA_SETS      250  /* # of data sets to do */
#define   NUM_POINTS     2048 /* # of data points */
#define   WS_SIZE        8192 /* Workspace size/process */

int       in_data[NUM_POINTS];
int       out_data[NUM_POINTS];

void
inbuf(Process *p)             /* Root node: Write to pipe */
  {
  int               i;

  for(i = 0; i < DATA_SETS; i++)   /* Random data */
    ChanOut(LINK1OUT,out_data,sizeof(out_data));
  }
```

```
void
outbuf(Process *p)            /* Root node: Read frm pipe */
  {
  int               i;

  for(i = 0; i < DATA_SETS; i++)   /* Read and toss */
    ChanIn(LINK2IN,in_data,sizeof(in_data));
  }

main()
  {
  if(_node_number == 1)       /* Root node */
    {
    Process         *in,*out;

    if((in = ProcAlloc(inbuf,WS_SIZE,0)) == NULL)
      {
      printf("No memory for process 'inbuf'\n");
      exit(1);
      }
    if((out = ProcAlloc(outbuf,WS_SIZE,0)) == NULL)
      {
      printf("No memory for process 'outbuf'\n");
      exit(1);
      }
    ProcPar(in,out,NULL);     /* Start I/O processes */
    ProcFree(in);        /* Return heap storage */
    ProcFree(out);
    printf("All done\n");
    }
  else if(_node_number == 2)  /* 1st pipeline stage */
    {
    while(1)
      {
      ChanIn(LINK2IN,in_data,sizeof(in_data));
/*
 *  Add the computation for the 1st stage of the pipeline.
 */
      ChanOut(LINK1OUT,in_data,sizeof(in_data));
      }
    }
  else if(_node_number == 3)  /* 2nd pipeline stage */
    {
    while(1)
      {
      ChanIn(LINK2IN,in_data,sizeof(in_data));
/*
 *  Add the computation for the 2nd stage of the pipeline.
 */
      ChanOut(LINK1OUT,in_data,sizeof(in_data));
      }
    }
```

```
  else              /* 3rd pipeline stage */
    {
    while(1)
      {
      ChanIn(LINK2IN,in_data,sizeof(in_data));
/*
 *  Add the computation for the 3rd stage of the pipeline.
 */
      ChanOut(LINK1OUT,in_data,sizeof(in_data));
      }
    }
  }
```

This code is provided as "exam7.c" in the example code directory. It should be linked with "_main" for an entry point.

Now we present another version with buffer processes attached to the input and output links of each processor in the pipeline. Note, in this example two processes are running on the root Transputer, and three processes on the rest of the nodes:

```
#include  <stdio.h>
#include  <conc.h>

#define   DATA_SETS      250  /* # of data sets to do */
#define   NUM_POINTS     2048 /* # of data points */
#define   WS_SIZE        8192 /* Workspace size/process */

int       comp_data[NUM_POINTS];   /* Buffer for compute */
int       in_data[NUM_POINTS];     /* Buffer for input */
int       out_data[NUM_POINTS];    /* Buffer for output */

void
inbuf(Process *p)             /* Root node: Write to pipe */
  {
  int               i;

  for(i = 0; i < DATA_SETS; i++)   /* Random data */
    ChanOut(LINK1OUT,out_data,sizeof(out_data));
  }

void
outbuf(Process *p)            /* Root node: Read frm pipe */
  {
  int               i;

  for(i = 0; i < DATA_SETS; i++)   /* Read and toss */
    ChanIn(LINK2IN,in_data,sizeof(in_data));
  }
```

```
void
pinbuf(Process *p,Channel *cin)
  {            /* Pipeline node: Read from buffer process */
  while(1)
    {
    ChanIn(LINK2IN,in_data,sizeof(in_data));
    ChanOut(cin,in_data,sizeof(in_data));
    }
  }

void
poutbuf(Process *p,Channel *cout)
  {            /* Pipeline node: Write to buffer process */
  while(1)
    {
    ChanIn(cout,out_data,sizeof(out_data));
    ChanOut(LINK1OUT,out_data,sizeof(out_data));
    }
  }

main()
  {
  if(_node_number == 1)       /* Root node */
    {
    Process         *in,*out;

    if((in = ProcAlloc(inbuf,WS_SIZE,0)) == NULL)
      {
      printf("No memory for process 'inbuf'\n");
      exit(1);
      }
    if((out = ProcAlloc(outbuf,WS_SIZE,0)) == NULL)
      {
      printf("No memory for process 'outbuf'\n");
      exit(1);
      }
    ProcPar(in,out,NULL);     /* Start I/O processes */
    ProcFree(in);             /* Return heap storage */
    ProcFree(out);
    printf("All done\n");
    }
  else
    {
    Channel         *cin,*cout;
    Process         *in,*out;

    if((cin = ChanAlloc()) == NULL)
      exit(1);                /* No memory, quit */
    if((in = ProcAlloc(pinbuf,WS_SIZE,1,cin)) == NULL)
      exit(1);                /* No memory, quit */
    ProcRun(in);              /* Run async. in process */
    if((cout = ChanAlloc()) == NULL)
      exit(1);                /* No memory, quit */
```

```
    if((out = ProcAlloc(poutbuf,WS_SIZE,1,cout)) == NULL)
      exit(1);                /* No memory, quit */
    ProcRun(out);             /* Run async. out process */
    if(_node_number == 2)     /* 1st stage */
      {
      while(1)
        {
        ChanIn(cin,comp_data,sizeof(comp_data));
/*
 *  Add the computation for the 1st stage of the pipeline.
 */
        ChanOut(cout,comp_data,sizeof(comp_data));
        }
      }
    else if(_node_number == 3)     /* 2nd stage */
      {
      while(1)
        {
        ChanIn(cin,comp_data,sizeof(comp_data));
/*
 *  Add the computation for the 2nd stage of the pipeline.
 */
        ChanOut(cout,comp_data,sizeof(comp_data));
        }
      }
    else                      /* 3rd stage */
      {
      while(1)
        {
        ChanIn(cin,comp_data,sizeof(comp_data));
/*
 *  Add the computation for the 3rd stage of the pipeline.
 */
        ChanOut(cout,comp_data,sizeof(comp_data));
        }
      }
    }
  }
```

This code is provided as "exam8.c" in the example code directory. It should be linked with "_main" as an entry point.

Note that we used internal ("soft"), channels to communicate between the buffer processes and the computation process.

The previous examples have exclusively used functions from the "Jeffrey Mock"-specified process primitives in the library. To show how the "Fork/Join" process primitives might be used, the next example is another way to accomplish the same results as in the preceding example:

```
#include  <stdio.h>
#include  <conc.h>

#define   DATA_SETS      250  /* # of data sets to do */
#define   NUM_POINTS     2048 /* # of data points */
#define   WS_SIZE        256  /* Workspace size/process */

int       comp_data[NUM_POINTS];   /* Buffer for compute */
int       in_data[NUM_POINTS];     /* Buffer for input */
int       out_data[NUM_POINTS];    /* Buffer for output */
char      inbufws[WS_SIZE];   /* Workspace for "inbuf" */

void
inbuf()                       /* Root node: Write to pipe */
  {
  int               i;

  for(i = 0; i < DATA_SETS; i++)   /* Random data */
    ChanOut(LINK1OUT,out_data,sizeof(out_data));
  }

void
pinbuf(Channel *cin)
  {            /* Pipeline node: Read from buffer process */
  while(1)
    {
    ChanIn(LINK2IN,in_data,sizeof(in_data));
    ChanOut(cin,in_data,sizeof(in_data));
    }
  }

void
poutbuf(Channel *cout)
  {            /* Pipeline node: Write to buffer process */
  while(1)
    {
    ChanIn(cout,out_data,sizeof(out_data));
    ChanOut(LINK1OUT,out_data,sizeof(out_data));
    }
  }
```

```
main()
  {
  if(_node_number == 1)       /* Root node */
    {
    int             i;
    Forkblk         f;
    PDes            p;

    PForkInit(f,2);           /* 2 processes on root node */
    p = PSetup(inbufws,inbuf,WS_SIZE,0);
    PFork(f,p);               /* Start "inbuf" process */
    for(i = 0; i < DATA_SETS; i++) /* Read/toss */
      ChanIn(LINK2IN,in_data,sizeof(in_data));
    PJoin(&f);                /* Synchronize processes */
    printf("All done\n");
    }
  else
    {
    Channel         cin = NOPROCESS;
    Channel         cout = NOPROCESS;
    char            inws[WS_SIZE],outws[WS_SIZE];

    PRun(PSetup(inws,pinbuf,WS_SIZE,1,&cin) | 1);
    PRun(PSetup(outws,poutbuf,WS_SIZE,1,&cout) | 1);
    if(_node_number == 2)     /* 1st stage */
      {
      while(1)
        {
        ChanIn(&cin,comp_data,sizeof(comp_data));
/*
 *  Add the computation for the 1st stage of the pipeline.
 */
        ChanOut(&cout,comp_data,sizeof(comp_data));
        }
      }
    else if(_node_number == 3)     /* 2nd stage */
      {
      while(1)
        {
        ChanIn(&cin,comp_data,sizeof(comp_data));
/*
 *  Add the computation for the 2nd stage of the pipeline.
 */
        ChanOut(&cout,comp_data,sizeof(comp_data));
        }
      }
```

```
    else                        /* 3rd stage */
      {
      while(1)
        {
        ChanIn(&cin,comp_data,sizeof(comp_data));
/*
 *   Add the computation for the 3rd stage of the pipeline.
 */
        ChanOut(&cout,comp_data,sizeof(comp_data));
        }
      }
    }
  }
```

This code is provided as "exam9.c" in the example code directory. It should be linked with "_main" as the entry point.

Several points to note in this example:

1. The processes created with the "Fork/Join" primitives do not provide an automatic ("Process *"), first parameter to spawned processes. In the "Fork/Join" model there is no requirement for the "Process" data structure.

2. We have chosen to allocate the stack/workspace for the processes explicitly (either globally or as an "auto" variable), rather than use "ProcAlloc" to get the required memory from the heap. The memory could have been allocated from the heap using a call to "malloc", but it isn't required.

3. We explicitly declared and initialized the "soft" channels being used. Again, we could have used "ChanAlloc" to get the memory from the heap and initialize it, but it isn't required (and takes more time).

We have now seen examples of programs which were both independent, and completely dependent, on a particular network topology. Between these two approaches there is a vast middle ground where optimum topology is somewhat important (or changes during different phases of program execution), but other factors take precedence. Although Transputer networks exist which are completely electrically programmable, nearly all have at least some contraints on allowed topologies.

Assuming your application needs to communicate with a node with which it isn't directly connected, and the Transputers you are using don't support hardware virtual channels, you have two options:

1. Build the actual communication topology into the application. This type of application is typified by communication instructions such as: "send this message out link 2", or, "if it comes in on link 1, send it out link 3", and so forth. This may be the best approach for fairly simple applications which execute on a fixed topology. This will often provide the highest possible communication performance if the majority of communication is between processes on neighboring nodes which have exclusive use of the connecting link.

2. Use the **Software Virtual Channel** facility to allow sending and receiving processes to function without reference to physical connections, or network topology. This approach offers slightly reduced communication performance, but often simplifies the job of programming a network application. It also generally improves the topology independence of the resulting application. See the LD-NET documentation and the **Alternatives to "_main"** section for general comments concerning configuring and linking programs for use with virtual channels. See the **Transputer Virtual Channel Communications** and **Transputer Virtual Channel Status Testing** sections for information about how to make use of virtual channels in user applications.

To illustrate some of the advantages of virtual channels we present a simple gate-level hardware simulator:

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <conc.h>

#define   WS_SIZE   1024 /* Workspace size/process */

void  *
get_ws(void)
  {            /* Get a process workspace */
  void              *rval;

  if((rval = malloc(WS_SIZE)) == NULL)
    {
    printf("Unable to 'malloc' storage for process
      workspace\n");
    exit(1);
    }
  return (rval);
  }
```

```
void
nand(int input1,int input2,int output)
  {            /* Perform logical NAND operation */
  Channel           *in1;
  Channel           *in2;
  Channel           *out;

  in1 = VChan(input1);        /* Get channel addresses */
  in2 = VChan(input2);
  out = VChan(output);

  if((in1 == NULL) || (in2 == NULL) || (out == NULL))
    PStop();        /* Die if not needed */

  while(1)
    VChanOutChar(out,! (VChanInChar(in1) &
      VChanInChar(in2)));
  }

void
wired_or(int input,int output1,int output2)
  {            /* Duplicate input on two outputs */
  int               data;
  Forkblk           f;
  Channel           *in;
  Channel           *out1;
  Channel           *out2;
  PDes              p;
  void              *ws;

  in = VChan(input);     /* Get channel addresses */
  out1 = VChan(output1);
  out2 = VChan(output2);

  if((in == NULL) || (out1 == NULL) || (out2 == NULL))
    PStop();        /* Die if not needed */

  ws = get_ws();    /* Get workspace for parallel output */

  while(1)
    {
    data = VChanInChar(in);
    PForkInit(f,2); /* Outputs to be done in parallel */
    p = PSetup(ws,VChanOutChar,WS_SIZE,2,out1,data);
    PFork(f,p);
    VChanOutChar(out2,data);
    PJoin(&f);      /* Synchronize processes */
    }
  }
```

```
void
main()
  {
  Forkblk           f;
  int               i;
  Channel           *in;
  Channel           *out1;
  Channel           *out2;
  PDes              p;
  void              *ws;
/*
 *  Convert logical VChan numbers to channel pointers.
 */
  in = VChan(6);
  out1 = VChan(7);
  out2 = VChan(8);

  if((in == NULL) || (out1 == NULL) || (out2 == NULL))
    {
    printf("Virtual channels not correctly configured\n");
    exit(1);
    }
/*
 *  Run three NAND processes.
 */
  PRun(PSetup(get_ws(),nand,WS_SIZE,3,9,10,11) | 1);
  PRun(PSetup(get_ws(),nand,WS_SIZE,3,12,13,14) | 1);
  PRun(PSetup(get_ws(),nand,WS_SIZE,3,15,16,17) | 1);
/*
 *  Run two WIRED-OR processes.
 */
  PRun(PSetup(get_ws(),wired_or,WS_SIZE,3,18,19,20) | 1);
  PRun(PSetup(get_ws(),wired_or,WS_SIZE,3,21,22,23) | 1);
/*
 *  Stimulate "external" logic function and display results.
 */
  ws = get_ws();    /* Get workspace for parallel output */

  printf("Truth Table For Externally Configured
    Function\n\n");
  printf("      Input1  Input2    |     Output\n");
  printf("      ------------------------------\n");
  for(i = 0; i < 4; i++)
    {
    printf("\t%d\t%d\t|\t",(i & 1) != 0,(i & 2) != 0);
    PForkInit(f,2); /* Outputs to be done in parallel */
    p = PSetup(ws,VChanOutChar,WS_SIZE,2,out1,
      ((i & 1) != 0));
    PFork(f,p);
    VChanOutChar(out2,((i & 2) != 0));
    PJoin(&f);      /* Synchronize processes */
    printf("%d\n",VChanInChar(in));
    }
  }
```

This code is provided as "exam10.c" in the example code directory. It should be linked with "_vcmain" as the entry point. To run this example on a single node, use LD-NET with either "and.nif", "or.nif" or "nand.nif" as the "Network Information File". These files contain virtual channel configurations which use the three NAND gates and two WIRED-OR connections available in the program to compute various boolean functions of two inputs.

Examining the "exam10.c" code from the beginning:

The "get_ws" function is used to obtain a process workspace from the heap using "malloc". Note that this differs from the "exam9.c" program where the concurrent process workspaces were taken from the initial process workspace.

The "nand" function reads two one byte boolean values from two virtual channels. It computes the boolean NAND function of the inputs and writes the result out a third virtual channel. In the "exam10.c" program up to three copies of this function may be running concurrently.

The "wired_or" function reads a one byte value from a virtual channel and writes the same value in parallel to two other virtual channels. The writes to the two output virtual channels must be done in parallel to avoid any output order dependent behavior. In the "exam10.c" program up to two copies of this function may be running concurrently.

The "main" function starts three copies of the "nand" function and two copies of the "wired_or" function as concurrent processes. It then writes all four combinations of two boolean values to the two output virtual channels. Finally, it reads from a third virtual input channel the results of the computation, and displays a truth table to the user.

Some comments on the program:

1. All the virtual channels were used for one-way communication. This was appropriate for this application, but some other application might take advantage of the two-way nature of virtual channels. After all, this is the most significant difference between virtual channels and regular T2/T4/T8 channels!

2. Only one node was used. This example ignores the benefits that virtual channel use provides for network topology independence. The use of the different ".nif" files for the various boolean functions does show some of the benefits of load-time reconfiguration.

If you are planning an application with heavy virtual channel message volume, and you are using the **Software Virtual Channel** facility, you should still plan on paying some attention to the mapping of your problem to the network topology. Your application will run correctly even if you don't, but the run-time performance may not be optimal. LD-NET incorporates a sophisticated routing generation algorithm, still if all the communication must transit completely across the network, it won't happen as fast as if the source and destination are reasonably close together (in a topological sense). Even topological independence involves tradeoffs!

This completes the **Sample Program** section. At this point you should be ready to start trying things out (and making your own mistakes). During the course of developing your application, please consider sending a copy of any non-proprietary, generally-useful, programs you create to Logical Systems (for inclusion on a future "sampler" disk).

abs

compute integer absolute value

SYNOPSIS

```
#include <stdlib.h>

int abs(int i)
```

DESCRIPTION

The **abs** function returns the integer absolute value of an integer argument **i**.

RETURN VALUE

See above

RELATED FUNCTIONS

labs, fabs, fabsf

acos/acosf

compute arc-cosine

SYNOPSIS

#include <math.h>

double acos(double x)

float acosf(float x)

DESCRIPTION

The **acos** function returns the principal value of the arc-cosine of **x** (in radians). The **acosf** function does the same thing for a single precision argument.

RETURN VALUE

A domain error ("errno" is set to EDOM), occurs if the absolute value of **x** is greater than one. Otherwise, the return value falls in the range [0,PI] radians.

RELATED FUNCTIONS

asin, atan, cos, sin, tan

addfree

add memory for heap allocation

SYNOPSIS

```
#include <stdlib.h>

void addfree(void *ptr, size_t size)
```

DESCRIPTION

Add a region of memory to be used by the "C" heap allocation routines. **ptr** should point to the start of the region to add, **size** is the length in bytes. Regions added to the heap in this fashion are disjoint from other parts of the heap. This means blocks allocated from this heap region aren't mergable with those from other regions when freed (for the purpose of creating a block larger than any one region).

You should ensure that the region of memory you are adding isn't used in any other way by your program, including as part of the default heap!

RETURN VALUE

None

RELATED FUNCTIONS

calloc, cfree, free, malloc, realloc

asctime

format broken-down time

SYNOPSIS

```
#include <time.h>

char *asctime(const struct tm *timeptr)
```

DESCRIPTION

The **asctime** function converts the broken-down time in the structure pointed to by **timeptr**, into an ASCII string representation. A sample of the string representation:

```
"Sat Aug 15 01:03:51 1972\n\0"
```

RETURN VALUE

The **asctime** function returns a pointer to the converted string. Note that the string is stored in a static data area which is shared with the "ctime" function. The string will be overwritten upon subsequent calls to either function.

RELATED FUNCTIONS

ctime, gmtime, localtime, time

asin/asinf

compute arc-sine

SYNOPSIS

#include <math.h>

double asin(double x)

float asinf(float x)

DESCRIPTION

The **asin** function returns the principal value of the arc-sine of x (in radians). The **asinf** function does the same thing for a single precision argument.

RETURN VALUE

A domain error ("errno" is set to EDOM), occurs if the absolute value of **x** is greater than (1). Otherwise, the return value falls in the range [-PI/2,+PI/2] radians.

RELATED FUNCTIONS

acos, atan, cos, sin, tan

atan/atanf

compute arc-tangent

SYNOPSIS

#include <math.h>

double atan(double x)

float atanf(float x)

DESCRIPTION

The **atan** function returns the principal value of the arc-tangent of **x** in radians. The **atanf** function does the same thing for a single precision argument.

RETURN VALUE

The return value falls in the range [-PI/2,+PI/2] radians.

RELATED FUNCTIONS

acos, asin, atan2, cos, sin, tan

atan2/atan2f

compute arc-tangent of x/y

SYNOPSIS

```
#include <math.h>

double atan2(double x, double y)

float atan2f(float x, float y)
```

DESCRIPTION

The **atan2** function returns the principal value of the arc-tangent of **x**/**y** using the signs of both arguments to determine the quadrant of the return value (in radians). The **atan2f** function does the same thing for single precision arguments.

RETURN VALUE

A domain error ("errno" is set to EDOM), occurs if both **x** and **y** are zero. Otherwise, the return value falls in the range [-PI,+PI] radians.

RELATED FUNCTIONS

atan

atof/atoi/atol

convert strings to numbers

SYNOPSIS

```
#include <stdlib.h>

double atof(const char *cptr)

int atoi(const char *cptr)

long atol(const char *cptr)
```

DESCRIPTION

The **atof** function converts an ASCII numeric string pointed to by **cptr** into the equivalent floating point number. The **atoi** and **atol** functions do the same thing for integers, and long integers, respectively. In all cases, the strings are assumed to be base 10, whitespace is allowed before the numeric string, and the first unrecognized character stops the conversion.

atof recognizes a numeric string consisting of an optional sign, a digit sequence with optional decimal point, and an optional exponent ('e' or 'E', optional sign and exponent digit sequence).

atoi and **atol** recognize integral numbers consisting of an optional sign and digit sequence.

RETURN VALUE

The converted value.

RELATED FUNCTIONS

strtod, strtol, strtoul

bcmp

compare bytes

SYNOPSIS

```
#include <string.h>

int bcmp(const void *s1, const void *s2, size_t n)
```

DESCRIPTION

The **bcmp** function compares byte string **s1** against byte string **s2** for **n** bytes, returning zero if they are identical, nonzero otherwise.

RETURN VALUE

See above

RELATED FUNCTIONS

memcmp

bcopy

string copy with length

SYNOPSIS

#include <string.h>

void *bcopy(const void *src, void *dst, size_t n)

DESCRIPTION

The **bcopy** function copies **n** bytes from the location pointed to by **src** to the location pointed to by **dst**.

The **bcopy** function is equivalent to "memcpy", but with a reversed **src** and **dst** argument order.

This function is implemented "inline". To get a functional version, precede your call with:

#undef bcopy

RETURN VALUE

Returns the value of **dst**.

RELATED FUNCTIONS

memcpy, memmove, strcpy

BitCnt

count the number of 1 bits set

SYNOPSIS

```
#include <conc.h>

int BitCnt(int i)
```

DESCRIPTION

Supplied only for Transputers which support the corresponding instruction. The **BitCnt** function takes a single integer argument and returns the number of 1 bits set.

This function is implemented "inline". To get a functional version, precede your call with:

```
#undef BitCnt
```

RETURN VALUE

See above

RELATED FUNCTIONS

BitRevNBits, BitRevWord

BitRevNBits

variable bit reversal

SYNOPSIS

```
#include <conc.h>

int BitRevNBits(int numbits, int data)
```

DESCRIPTION

Supplied only for Transputers which support the corresponding instruction. The **BitRevNBits** function takes two integer arguments; **numbits** specifies how many of the bits in **data** should be reversed (counting from least significant bit). The result of the reversal is returned. All high order bits not involved in the reversal are zeroed.

This function is implemented "inline". To get a functional version, precede your call with:

```
#undef BitRevNBits
```

RETURN VALUE

See above

RELATED FUNCTIONS

BitCnt, BitRevWord

BitRevWord

reverse bits in word

SYNOPSIS

```
#include <conc.h>

int BitRevWord(int data)
```

DESCRIPTION

Supplied only for Transputers which support the corresponding instruction. The **BitRevWord** function reverses the bit pattern in **data** end-for-end, and returns the result.

This function is implemented "inline". To get a functional version, precede your call with:

```
#undef BitRevWord
```

RETURN VALUE

See above

RELATED FUNCTIONS

BitCnt, BitRevNBits

bzero

set a range of memory to a value

SYNOPSIS

#include <string.h>

void bzero(void *dst, size_t n)

DESCRIPTION

The **bzero** function writes **n** zero bytes into memory starting at **dst**.

RETURN VALUE

None

RELATED FUNCTIONS

memset

cabs/cabsf

compute complex absolute value

SYNOPSIS

#include <math.h>

double cabs(COMPLEX x)

float cabsf(COMPLEXF x)

DESCRIPTION

The **cabs** function returns the absolute value of a complex number **x**. A call to **cabs** is equivalent to the following expression:

hypot(x.real,x.imag)

The **cabsf** function does the same thing for a single precision argument.

RETURN VALUE

See above

RELATED FUNCTIONS

fft, ifft

calloc

allocate and initialize heap memory

SYNOPSIS

#include <stdlib.h>

void *calloc(size_t nmemb, size_t size)

DESCRIPTION

The **calloc** function allocates, and bitwise zeroes, a region of heap memory large enough to hold **nmemb** items of **size** bytes each.

RETURN VALUE

If it is impossible to satisfy the request, or **size** is zero, a NULL pointer is returned. Otherwise, a pointer is returned to the start of the allocated region.

RELATED FUNCTIONS

addfree, cfree, free, malloc, realloc

ceil/ceilf

compute "ceiling"

SYNOPSIS

#include <math.h>

double ceil(double x)

float ceilf(float x)

DESCRIPTION

The **ceil** and **ceilf** functions return the smallest floating point integral value not less than **x**.

RETURN VALUE

See above

RELATED FUNCTIONS

floor, floorf

cfree

free heap memory

SYNOPSIS

#include <stdlib.h>

void cfree(void *ptr)

DESCRIPTION

The **cfree** function returns the previously allocated region of heap memory pointed to by **ptr** to the heap free storage pool.

The value of **ptr** must be the result of an earlier call to "calloc", "malloc", or "realloc"; or the results are undefined.

The **cfree** function is equivalent to the "free" function.

RETURN VALUE

None

RELATED FUNCTIONS

addfree, calloc, free, malloc, realloc

ChanAlloc/ChanFree

communication channel allocation

SYNOPSIS

```
#include <conc.h>

Channel *ChanAlloc(void)

void ChanFree(Channel *ptr)
```

DESCRIPTION

The **ChanAlloc** function returns a pointer to an initialized software "channel". The memory for the "channel" is obtained from the heap.

The **ChanFree** function takes a "channel" previously allocated using **ChanAlloc** and returns it to the heap free storage pool.

Note that "channels" may also be statically allocated (either globally or as "auto" variables), if the dynamic creation and destruction allowed by **ChanAlloc** and **ChanFree** isn't required.

For additional information see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

ChanAlloc either returns a pointer to the allocated "channel", or NULL if no heap memory was available.

There is no return value from **ChanFree**.

RELATED FUNCTIONS

ChanIn, ChanInChanFail, ChanInChar, ChanInInt, ChanInTimeFail, ChanOut, ChanOutChanFail, ChanOutChar, ChanOutInt, ChanOutTimeFail, ChanReset

ChanIn/ChanInChar/ChanInInt
reading from channels

SYNOPSIS

```
#include <conc.h>

void ChanIn(Channel *c, void *ptr, int n)

char ChanInChar(Channel *c)

int ChanInInt(Channel *c)
```

DESCRIPTION

The **ChanIn** function reads **n** bytes of data, from the "channel" pointed to by **c**, to the buffer pointed to by **ptr**. The **ChanInChar** and **ChanInInt** functions may be used to read, and return, the value of a byte or word, respectively, read from the "channel" pointed to by **c**.

The Transputer supports a message passing paradigm, in hardware, using the concept of "channels". Briefly, "channels" are unidirectional message passing funnels between two processes, or processors (consult the appropriate INMOS documentation for a more detailed explanation). The **ChanIn**, **ChanInChar**, and **ChanInInt** functions allow for the reading of data from a "channel"; the complementary "ChanOut", "ChanOutChar" and "ChanOutInt" routines may be used to write data to a "channel".

In the Transputer model, "channel" communication may also be used as a scheduling mechanism. This works because no buffering is associated with a "channel"; the communication occurs only when both the reading and writing processes are ready.

In Transputer terminology, "channels" are classified as either "hard" or "soft". "hard channels" are those associated with the physical links between Transputers, "soft channels" are those used for intra-processor communication. One implementation difference between these two classes of channels may sometimes be usefully exploited:

The message length for the reading and writing process doing "channel" communication must be identical for "soft channels", but may be different if "hard channels" are used. For example, this means that a process using a "hard channel" can send a 6 byte message as one "ChanOut" call, while the corresponding **ChanIn** call on the connected processor could be broken into two **ChanIn** calls for 3 bytes each. This feature is particularly useful when implementing packet based communication protocols (the overhead portion of the packet can be received separately, and used to configure the length of the **ChanIn** call used to read the rest of the packet).

In order to be used, "channels" must be initialized. This is accomplished automatically by the "ChanAlloc" function, and should also be done by the programmer for "channels" created in other ways. For example, a "channel" might be declared and initialized in "C" using the following statement:

```
Channel xyz = NOPROCESS;
```

Where "xyz" is the name of the "channel", and the macro NOPROCESS is used to supply the correct initial value. The NOPROCESS macro definition, and symbolic definitions for the addresses of the "hard channels", may be found in the "conc.h" include file.

These functions are implemented "inline". To get a functional version, precede your call with:

```
#undef ChanIn
```

Or

```
#undef ChanInChar
```

Or

```
#undef ChanInInt
```

For additional information about the "channel" primitives, see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

See above

RELATED FUNCTIONS

ChanAlloc, ChanFree, ChanInChanFail, ChanInTimeFail, ChanOut, ChanOutChanFail, ChanOutChar, ChanOutInt, ChanOutTimeFail, ChanReset

ChanInChanFail/ChanInTimeFail
error read from channel

SYNOPSIS

```
#include <conc.h>

int ChanInChanFail(Channel *c1, void *ptr, int n, Channel *c2)

int ChanInTimeFail(Channel *c, void *ptr, int n, int t)
```

DESCRIPTION

The **ChanInChanFail** function is equivalent to the "ChanIn" function up through the first three parameters. In addition, a fourth parameter has been added (**c2**), which allows the "channel" read operation on **c1** to be aborted when an "int" is written to the auxiliary channel **c2** (and thus **c2** becomes ready for reading by **ChanInChanFail**). This allows another process to "timeout" a communication operation. The **ChanInChanFail** function automatically resets channel **c1** if the communication is aborted. Note that if channel **c1** is a "hard" channel, and communication was in progress, the reset will cause the communication operation to never complete (from the point of view of the other processor).

The **ChanInTimeFail** function is like **ChanInChanFail**, except the fourth parameter is a "time" value which is compared to the hardware clock associated with the current priority level. When the hardware clock is "after" the specified value **t**, the communication is aborted and the "channel" is reset.

For additional information about the "channel" primitives, see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

Either function returns zero if the communication completed, or one if the communication was aborted.

RELATED FUNCTIONS

ChanAlloc, ChanFree, ChanIn, ChanInChar, ChanInInt, ChanOut, ChanOutChanFail, ChanOutChar, ChanOutInt, ChanOutTimeFail, ChanReset

ChanOut/ChanOutChar/ChanOutInt
writing to channels

SYNOPSIS

```
#include <conc.h>

void ChanOut(Channel *c, void *ptr, int n)

void ChanOutChar(Channel *c, int byte)

void ChanOutInt(Channel *c, int word)
```

DESCRIPTION

The **ChanOut** function writes **n** bytes of data, to the "channel" pointed to by **c**, from the buffer pointed to by **ptr**. The **ChanOutChar** and **ChanOutInt** functions may be used to write the value of a **byte** or **word**, respectively, to the "channel" pointed to by **c**.

See the discussion for the corresponding read operations ("ChanIn" ...), for a more lengthy introduction.

These functions are implemented "inline". To get a functional version, precede your call with:

```
#undef ChanOut
```

Or

```
#undef ChanOutChar
```

Or

```
#undef ChanOutInt
```

For additional information about the "channel" primitives, see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

None

RELATED FUNCTIONS

ChanAlloc, ChanFree, ChanIn, ChanInChanFail, ChanInChar, ChanInInt, ChanInTimeFail, ChanOutChanFail, ChanOutTimeFail, ChanReset

ChanOutChanFail/ChanOutTimeFail
error write to channel

SYNOPSIS

```
#include <conc.h>

int ChanOutChanFail(Channel *c1, void *ptr, int n, Channel *c2)

int ChanOutTimeFail(Channel *c, void *ptr, int n, int t)
```

DESCRIPTION

The **ChanOutChanFail** function is equivalent to the "ChanOut" function up through the first three parameters. In addition, a fourth parameter has been added (**c2**), which allows the "channel" write operation on **c1** to be aborted when an "int" is written to the auxiliary channel **c2** (and thus **c2** becomes ready for reading by **ChanOutChanFail**). This allows another process to "timeout" a communication operation. The **ChanOutChanFail** function automatically resets channel **c1** if the communication is aborted. Note that if channel **c1** is a "hard" channel, and communication was in progress, the reset will cause the communication operation to never complete (from the point of view of the other processor).

The **ChanOutTimeFail** function is like **ChanOutChanFail**, except the fourth parameter is a "time" value which is compared to the hardware clock associated with the current priority level. When the hardware clock is "after" the specified value **t**, the communication is aborted and the "channel" is reset.

For additional information about the "channel" primitives, see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

Either function returns zero if the communication completed, or one if the communication was aborted.

RELATED FUNCTIONS

ChanAlloc, ChanFree, ChanIn, ChanInChanFail, ChanInChar, ChanInInt, ChanInTimeFail, ChanOut, ChanOutChar, ChanOutInt, ChanReset

ChanReset

reset a channel

SYNOPSIS

```
#include <conc.h>

int ChanReset(Channel *c)
```

DESCRIPTION

The **ChanReset** function is an extremely low level function which is used to reset and initialize a Transputer channel. The value returned reflects the current status of the channel.

Under normal circumstances, use of this function should NEVER be required, and it is included only to provide a "C" binding for the Transputer "resetch" instruction. Consult the appropriate INMOS instruction set documentation for further information.

This function is implemented "inline". To get a functional version, precede your call with:

```
#undef ChanReset
```

RETURN VALUE

The current status of the channel being reset. This is either the process descriptor of the process blocked on the channel, or the value of the NOPROCESS macro (if the channel was not in use).

RELATED FUNCTIONS

ChanAlloc, ChanFree, ChanIn, ChanInChanFail, ChanInChar, ChanInInt, ChanInTimeFail, ChanOut, ChanOutChanFail, ChanOutChar, ChanOutInt, ChanOutTimeFail

_cioext

host specific server extensions

SYNOPSIS

```
#include <cioext.h>

int _cioext(int fn, void *ibuf, int ilen, void *obuf, int olen)
```

DESCRIPTION

The **_cioext** function provides a user extensible, remote procedure call mechanism for use with the CIO host I/O server. The arguments to **_cioext** are:

1. **fn -** The desired user extension function code.

2. **ibuf** - A pointer to the input buffer for the user extension function.

3. **ilen** - The number of bytes in **ibuf**. The input buffer must be shorter than MAX_MSG_DATA as defined in cio.h (25000 bytes in the standard **Transputer Toolset** release).

4. **obuf** - A pointer to the output buffer for the user extension function.

4. **olen** - The number of bytes to return to **obuf**. The output buffer must be shorter than MAX_MSG_DATA as defined in cio.h (25000 bytes in the standard **Transputer Toolset** release).

The actual functions which are available are dependent on the capabilities implemented in the "cioext.c" file (which is linked with CIO). See the **CIO "C" I/O Driver User Guide** manual for further information.

RETURN VALUE

If the input or output buffers are too large -1 is returned. Otherwise the return value is completely dependent on the implementation of the specific function being invoked.

RELATED FUNCTIONS

None

clearerr

clear stream status

SYNOPSIS

#include <stdio.h>

void clearerr(FILE *stream)

DESCRIPTION

The **clearerr** function resets the "eof" and "error" status flags associated with the specified **stream**.

RETURN VALUE

None

RELATED FUNCTIONS

feof, ferror, rewind

close

low level file close

SYNOPSIS

#include <stdio.h>

int close(int handle)

DESCRIPTION

Closes the file with file descriptor **handle**.

The file must have been previously opened by a call to "open" or "creat".

Note that termination of a program via "exit", or a return from "main", also closes all open files.

As this function is not supported in the ANSI standard library, its use in new work is discouraged (see the "fopen", "fclose", "fread", and "fwrite" functions for the recommended replacements).

RETURN VALUE

The **close** function returns zero if the file was successfully closed. A return value of (-1), indicates an error, and "errno" is set appropriately.

RELATED FUNCTIONS

creat, dup, dup2, open, read, write

cos/cosf

compute cosine

SYNOPSIS

#include <math.h>

double cos(double x)

float cosf(float x)

DESCRIPTION

The **cos** function returns the cosine of **x** (measured in radians). The **cosf** function does the same thing for a single precision argument.

RETURN VALUE

A large magnitude argument may yield a result with little significance.

RELATED FUNCTIONS

acos, asin, atan, sin, tan

cosh/coshf

compute hyperbolic cosine

SYNOPSIS

#include <math.h>

double cosh(double x)

float coshf(float x)

DESCRIPTION

The **cosh** function returns the hyperbolic cosine of **x**. The **coshf** function does the same thing for a single precision argument.

RETURN VALUE

A range error ("errno" is set to ERANGE), occurs if the magnitude of **x** is too large.

RELATED FUNCTIONS

sinh, tanh

creat

low level file creation

SYNOPSIS

```
#include <stdio.h>
#include <fcntl.h>

int creat(char *path, int mode)
```

DESCRIPTION

The **creat** function either creates a new file, or opens and truncates an existing file. If the file specified by **path** doesn't already exist, it is created with read/write permission as set by **mode**. If the file already exists, and its permissions allow writing, the file is truncated to zero length and opened for writing.

The **mode** parameter applies only to newly created files, and determines what permission setting the file should be given when closed.

The following symbolic bit mask macro definitions are used with **creat**:

S_IREAD - Allow file reading
S_IWRITE - Allow file writing

If the values of the above two macros are bitwise OR'ed together, and passed as the **mode** parameter, the file is assumed to allow both reading and writing.

As this function is not supported in the ANSI standard library, its use in new work is discouraged (see the "fopen", "fclose", "fread", and "fwrite" functions for the recommended replacements).

RETURN VALUE

The **creat** function returns a positive value (a "handle"), for the file if the open/create is successful, otherwise (-1) is returned and "errno" is set appropriately.

RELATED FUNCTIONS

close, dup, dup2, open, read, write

ctime

format local time

SYNOPSIS

#include <time.h>

char *ctime(const time_t *timer)

DESCRIPTION

The **ctime** function converts the calendar time specified by **timer** into local time in the form of an ASCII string. A sample of the string representation:

"Sat Aug 15 01:03:51 1972\n\0"

The **ctime** function may be thought of as a functional equivalent for the following expression:

asctime(localtime(timer))

RETURN VALUE

The **ctime** function returns a pointer to the converted string. Note that the string is stored in a static data area which is shared with the "asctime" function. The string will be overwritten upon subsequent calls to either function.

RELATED FUNCTIONS

asctime, gmtime, localtime, time

div

compute integer quotient and remainder

SYNOPSIS

```
#include <stdlib.h>

div_t div(int num, int denom)
```

DESCRIPTION

The **div** function returns the integer quotient and remainder of dividing **num** by **denom**. If the division is inexact, the sign of the quotient will be the same as the algebraic quotient, and the magnitude of the quotient will be the largest integer less than the magnitude of the algebraic quotient. If the result can be represented, then:

num = quotient * **denom** + remainder

RETURN VALUE

The quotient and remainder are returned in a structure of type "div_t", which has two integer members, "quot" and "rem".

RELATED FUNCTIONS

ldiv

dup/dup2

low level file handle duplication

SYNOPSIS

#include <stdio.h>

int dup(int handle)

int dup2(int handle, int handle2)

DESCRIPTION

The **dup** and **dup2** functions assign a second handle to a function which already has one.

The **dup** function returns the next available handle for the file specified by **handle**.

The dup2 function forces **handle2** to refer to the same file as **handle**. This will require closing any existing file which is opened with **handle2**.

These functions are not supported in the ANSI standard library, thus use in new work is discouraged (see the "fopen", "fclose", "fread", and "fwrite" functions for the recommended replacements).

RETURN VALUE

Both **dup** and **dup2** return (-1) on failure and set the value of "errno" as appropriate. On success, **dup** returns the second handle, **dup2** returns zero.

RELATED FUNCTIONS

close, creat, open, read, write

exit

terminate program execution

SYNOPSIS

```
#include <stdlib.h>

void exit(int status)
```

DESCRIPTION

The **exit** function terminates program execution and returns control to the host environment. The **status** parameter is used to return a exit code for the program.

Other than the **status** code, calling **exit** is equivalent to returning from the "main" function:

1. All open output streams are flushed.

2. All open streams and files are closed.

RETURN VALUE

There is (and can be), no return value, since control is passed to the host environment after **exit** is called.

RELATED FUNCTIONS

None

exp/expf

compute exponential

SYNOPSIS

#include <math.h>

double exp(double x)

float expf(float x)

DESCRIPTION

The **exp** function returns the floating point exponential function of **x**. The **expf** function does the same thing for a single precision argument.

RETURN VALUE

A range error ("errno" is set to ERANGE), occurs if **x** is too large.

RELATED FUNCTIONS

log, log10, pow

fabs/fabsf

compute floating point absolute value

SYNOPSIS

```
#include <math.h>

double fabs(double x)

float fabsf(float x)
```

DESCRIPTION

The **fabs** function returns the floating point absolute value of a floating point argument **x**. The **fabsf** function does the same thing for a single precision argument.

On floating point Transputers this function is implemented "inline". To get a functional version, precede your call with:

```
#undef fabs
```

Or,

```
#undef fabsf
```

RETURN VALUE

See above

RELATED FUNCTIONS

abs, labs

fclose/fcloseall

close streams

SYNOPSIS

#include <stdio.h>

int fclose(FILE *stream)

int fcloseall(void)

DESCRIPTION

The **fclose** function causes any buffered data for the specified **stream** to be written out, and the **stream** to be closed. The **fcloseall** function does the same thing for all currently open streams.

Note that termination of a program via "exit", or a return from "main", also closes all open files.

RETURN VALUE

These functions return EOF to indicate an error. The **fclose** function returns zero for success. The **fcloseall** function returns the number of open streams which were closed.

RELATED FUNCTIONS

close, fflush, fopen

fdopen

convert handle to stream

SYNOPSIS

```
#include <stdio.h>

FILE *fdopen(int handle, const char *mode)
```

DESCRIPTION

The **fdopen** function associates a stream with a file previously opened with **handle**. The allowed format for the **mode** string is the same as used with the "fopen" function, but must not conflict with the original access permission when the file was opened using "open" or "creat".

As this function is not supported in the ANSI standard library, its use in new work is discouraged (see the "fopen", "fclose", "fread", and "fwrite" functions for the recommended replacements).

RETURN VALUE

The **fdopen** function returns NULL on error, otherwise a pointer to the opened stream.

RELATED FUNCTIONS

fileno, fopen

feof/ferror

check stream status

SYNOPSIS

```
#include <stdio.h>

int feof(FILE *stream)

int ferror(FILE *stream)
```

DESCRIPTION

The **feof** function returns a non-zero value if EOF has been reached for the specified **stream** (assumed to be open for reading). The **ferror** function returns non-zero when an error has been detected during reading or writing of the specified **stream**.

RETURN VALUE

See above. The functions return non-zero if the predicate being tested is TRUE, zero if FALSE.

RELATED FUNCTIONS

clearerr

fflush

flush stream

SYNOPSIS

#include <stdio.h>

int fflush(FILE *stream)

DESCRIPTION

The **fflush** function causes any buffered data for the named stream to be written, if the stream is open for writing; or purged, if the stream is open for reading. The **fflush** call also purges anything buffered by a prior "ungetc" call.

RETURN VALUE

The fflush function returns EOF on error and zero for success.

RELATED FUNCTIONS

fclose

fft/fftf

compute forward FFT transform

SYNOPSIS

#include <math.h>

void fft(COMPLEX x[], int logsize)

void fftf(COMPLEXF x[], int logsize)

DESCRIPTION

The **fft** function performs a radix-2, forward, Fast Fourier Transform on an input array of complex floating point numbers (**x**). The **fftf** function does the same thing for an array of single precision floating point numbers. The transform is done in-place, so the output data is also stored into **x**. The **logsize** parameter is the log (base 2), of the number of complex numbers in the array. In the following definition, N is equal to the number of elements in the array (2**__logsize__):

$$x'(k) = \sum_{n=0}^{N-1} x(n) e^{-j(2(PI)/N)kn}$$

Note that there is considerable disagreement in the literature about what constitutes the "forward" transform, and what should be called the "inverse" (see the "ifft" routine description). Also, for a complete forward-inverse transform to produce the original data values, a scaling factor of 1/N must be applied to the data. Some authorities apply it all on the forward or inverse transforms, others apply 1/sqrt(N) to each. With this in mind, the **fft** and "ifft" functions do not perform any scaling. If your application requires it, you must apply it directly to the individual data points produced by one of the transforms.

RETURN VALUE

None. Note that for floating point Transputers, the **fftf** function has been hand optimized, for speed, in assembly language.

RELATED FUNCTIONS

ifft

fgetc

read character

SYNOPSIS

```
#include <stdio.h>

int fgetc(FILE *stream)
```

DESCRIPTION

The **fgetc** function obtains the next character (if available), from the input **stream**. The "getc" macro is functionally equivalent (ignoring side effects).

RETURN VALUE

The **fgetc** function returns EOF on error, or the character read. The character is treated as unsigned for purposes of distinguishing it from the error return.

RELATED FUNCTIONS

fputc, getc, getchar, putc, putchar, ungetc

fgets

read a line

SYNOPSIS

#include <stdio.h>

char *fgets(char *ptr, int n, FILE *stream)

DESCRIPTION

The **fgets** function reads at most **n**-1 characters from the **stream**, and places them in memory starting at **ptr**. A newline, or EOF, also terminates the read operation (the newline is retained). A '\0' character is added after the last character read.

RETURN VALUE

The **fgets** function returns NULL if an error was detected (or EOF found before any characters were read). On success **ptr** is returned.

RELATED FUNCTIONS

fputs, gets, puts

fileno

convert stream to handle

SYNOPSIS

```
#include <stdio.h>

int fileno(FILE *stream)
```

DESCRIPTION

The **fileno** function converts a **stream** into the corresponding file handle.

As this function is not supported in the ANSI standard library, its use in new work is discouraged (see the "fopen", "fclose", "fread", and "fwrite" functions for the recommended replacements).

RETURN VALUE

See above

RELATED FUNCTIONS

fdopen

floor/floorf

compute "floor"

SYNOPSIS

#include <math.h>

double floor(double x)

float floorf(float x)

DESCRIPTION

The **floor** and **floorf** functions return the largest floating point integral value not greater than **x**.

RETURN VALUE

See above

RELATED FUNCTIONS

ceil, ceilf

fmod/fmodf

compute floating point remainder

SYNOPSIS

#include <math.h>

double fmod(double x, double y)

float fmodf(float x, float y)

DESCRIPTION

The **fmod** and **fmodf** functions compute the floating point remainder of **x**/**y**.

RETURN VALUE

The functions return the value **x** - (i * **y**), for some integer i such that if **y** is non-zero, the result has the same sign as **x** and magnitude less than the magnitude of **y**. If **y** is zero the results are undefined.

RELATED FUNCTIONS

None

fopen

open stream

SYNOPSIS

#include <stdio.h>

FILE *fopen(const char *path, const char *mode)

DESCRIPTION

The **fopen** function opens the file specified by **path**, for reading and/or writing, as indicated by **mode**. The mode string specifies the allowed operations (all settings assume the file will be text and CR/LF mapping may be used by the host OS if necessary):

"a" - Open the file for appending (at the end). The file is created if it doesn't exist.
"a+" - Open the file for reading and appending (at the end). The file is created if it doesn't exist.
"r" - Open the file for reading (the file must already exist).
"r+" - Open the file for reading and writing (the file must already exist).
"w" - Open/create the file for writing. If the file exists the contents are purged.
"w+" - Open/create the file for reading and writing. If the file exists the contents are purged.

To open a "binary" file without any CR/LF mapping nonsense, append a 'b' character at the end of the mode string (or optionally before the '+', if one is used). For example, to open a binary file in "r+" mode, either of the following are legal:

"r+b" or "rb+"

RETURN VALUE

The **fopen** function returns NULL on error, otherwise the new stream pointer.

RELATED FUNCTIONS

fclose

fprintf

formatted write to stream

SYNOPSIS

```
#include <stdio.h>

int fprintf(FILE *stream, const char *format, ...)
```

DESCRIPTION

The **fprintf** function writes to the specified **stream**. The write is formatted under the control of the **format** argument, which specifies how subsequent arguments (if any), are to be converted for output. See the "printf" function description for more information about formatting options.

RETURN VALUE

The number of characters written (negative values indicate an error).

RELATED FUNCTIONS

printf, sprintf, vfprintf, vprintf, vsprintf

fputc

write a character

SYNOPSIS

#include <stdio.h>

int fputc(int c, FILE *stream)

DESCRIPTION

The **fputc** function writes the character specified by c to the output **stream**.

RETURN VALUE

The **fputc** function returns the character written (or EOF on error).

RELATED FUNCTIONS

fgetc, getc, getchar, putc, putchar, ungetc

fputs

write a line

SYNOPSIS

#include <stdio.h>

int fputs(const char *ptr, FILE *stream)

DESCRIPTION

The **fputs** function writes a '\0' terminated string (pointed to by **ptr**), to the specified **stream**. The '\0' is not written.

RETURN VALUE

The **fputs** function returns EOF if an error occurs; otherwise a non-negative value.

RELATED FUNCTIONS

fgets, gets, puts

frand/frandf

floating point random number

SYNOPSIS

#include <stdlib.h>

double frand(void)

float frandf(void)

DESCRIPTION

The **frand** function returns a sequence of pseudo-random floating point values in the range of zero to one (inclusive). The **frand** function uses the same seed as the integer "rand" function, and thus, may also be initialized using the "srand" function. The **frandf** function does the same thing as **frand**, but produces a single precision result.

RETURN VALUE

See above

RELATED FUNCTIONS

rand, srand

fread

read from stream

SYNOPSIS

#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmembs, FILE *stream)

DESCRIPTION

The **fread** function reads **nmembs** worth of data items, each of **size** bytes, from **stream**, into memory starting at the address specified by **ptr**.

RETURN VALUE

The number of data items successfully read. This number may be less than **nmembs** if an error or EOF is encountered.

RELATED FUNCTIONS

fwrite

free

free heap memory

SYNOPSIS

#include <stdlib.h>

void free(void *ptr)

DESCRIPTION

The **free** function returns the previously allocated region of heap memory pointed to by **ptr** to the heap free storage pool.

The value of **ptr** must be the result of an earlier call to "calloc", "malloc", or "realloc"; or the results are undefined.

The **free** function is equivalent to the "cfree" function.

RETURN VALUE

None

RELATED FUNCTIONS

addfree, calloc, cfree, malloc, realloc

freopen

redirect stream

SYNOPSIS

#include <stdio.h>

FILE *freopen(const char *path, const char *mode, FILE *stream)

DESCRIPTION

The **freopen** function closes whatever is associated with **stream** and opens the file specified by **path** and assigns **stream** to it. The allowed format for the **mode** string is the same as used with the "fopen" function.

This function is conventionally used to redirect the streams which are automatically opened for the user program ("stdio", "stdout", and "stderr").

RETURN VALUE

The **freopen** function returns NULL on error and closes the original stream. On success, **freopen** returns a pointer to the "new" stream.

RELATED FUNCTIONS

fopen

frexp/frexpf

decompose floating point number

SYNOPSIS

```
#include <math.h>

double frexp(double value, int *exp)

float frexpf(float value, int *exp)
```

DESCRIPTION

The **frexp** function breaks a floating point **value** into a normalized fraction and a integral power of two. The integral value is stored in the "int" pointed to by **exp**. The **frexpf** function does the same thing for a single precision argument.

RETURN VALUE

These functions return a floating point number, such that the number has magnitude in the range [1/2,1] or zero, and **value** equals the number raised to the power ***exp**. If **value** is zero, both parts of the result are also.

RELATED FUNCTIONS

ldexp

fscanf

formatted read from stream

SYNOPSIS

```
#include <stdio.h>

int fscanf(FILE *stream, const char *format, ...)
```

DESCRIPTION

The **fscanf** function reads from the specified **stream**. The read is formatted under the control of the **format** argument, which specifies the legal input text sequences and conversion instructions. Subsequent arguments (if any), are used as pointers to the objects which receive converted data from the input. See the "scanf" function description for more information about formatting options.

RETURN VALUE

The **fscanf** function returns EOF if an input failure occurs before any conversions. Otherwise, **fscanf** returns the number of input sequences which were matched, converted, and assigned. The number of sequences returned, may be less than the number provided for in the **format** string, if a match fails, or EOF is encountered.

If EOF occurs in the middle of matching an input sequence, such that the match may already be considered successful (not counting optionally matched whitespace), the EOF is considered to merely terminate the current match. Any remaining conversions, however, will be immediately aborted if they require input.

RELATED FUNCTIONS

scanf, sscanf, strtod, strtol, strtoul

fseek

change position within stream

SYNOPSIS

#include <stdio.h>

int fseek(FILE *stream, long int offset, int origin)

DESCRIPTION

The **fseek** function changes the current read/write position within the specified **stream**. The new position is computed as an **offset** (in bytes), from the measuring position specified by **origin**. Three macro definitions in "stdio.h" indicate the legal values for **origin**:

1. SEEK_SET - The value of **offset** is measured from the start of the stream.

2. SEEK_CUR - The value of **offset** is measured from the current position in the stream.

3. SEEK_END - The value of **offset** is measured from the end of the stream.

Note that for host environments which perform CR/LF mapping, seeks on "text" streams may produces unexpected results unless the value used for **offset** is either zero, or was returned from a previous call to the "ftell" function. If a value returned from a "ftell" call is used, the **origin** value to the **fseek** call should be "SEEK_CUR".

RETURN VALUE

Zero on success, non-zero if the request is illegal. An example of an illegal request: calling **fseek** with an **origin** value of "SEEK_SET", and a negative **offset**, (seeking before the beginning of a file).

RELATED FUNCTIONS

ftell, lseek, rewind

ftell

report current position within stream

SYNOPSIS

```
#include <stdio.h>

long int ftell(FILE *stream)
```

DESCRIPTION

The **ftell** function reports the current read/write position within the specified **stream**. For a binary stream, this value is the number of bytes from the beginning of the file. If the host environment performs CR/LF mapping within "text" streams, the reported value has no simple definition, and is mainly used to keep track of a particular stream position which will later be returned to using a "fseek" function call.

RETURN VALUE

The current stream position value is returned on success, (-1L) on failure.

RELATED FUNCTIONS

fseek, lseek, rewind

fwrite

write to stream

SYNOPSIS

#include <stdio.h>

size_t fwrite(const void *ptr, size_t size, size_t nmembs, FILE *stream)

DESCRIPTION

The **fwrite** function writes **nmembs** worth of data items, each of **size** bytes, to **stream**, from memory starting at the address specified by **ptr**.

RETURN VALUE

The number of data items successfully written. This number may be less than **nmembs** if an error is encountered.

RELATED FUNCTIONS

fread

getc/getch/getchar/getche

read a character

SYNOPSIS

```
#include <stdio.h>

int getc(FILE *stream)

int getch(void)

int getchar(void)

int getche(void)
```

DESCRIPTION

The **getc** macro obtains the next character (if available), from the input **stream**. Since **getc** is defined to be a macro, the stream argument may be evaluated more than once, and must not have side-effects. To get a functional version, use either the "fgetc" equivalent, or precede the function call with:

```
#undef getc
```

The **getch** function is probably specific to MS-DOS. It reads, without echoing, a single character from the console. When reading a cursor key, or function key, **getch** must be called twice (the second call gets the actual scan code).

The **getchar** macro obtains the next character (if available), from the input stream "stdin". To get a functional version, use either the "fgetc(stdin)" equivalent, or precede the function call with:

```
#undef getchar
```

The **getche** function is probably specific to MS-DOS. It reads, and echoes, a single character from the console. When reading a cursor key, or function key, **getche** must be called twice (the second call gets the actual scan code).

RETURN VALUE

The **getch** and **getche** functions returns the character, or portion thereof, read. There is no error return.

The **getc** and **getchar** macros/functions return EOF on error, or the character read. The characters are treated as unsigned for purposes of distinguishing them from the error return.

RELATED FUNCTIONS

fgetc, fputc, putc, putchar, ungetc

getenv

read environment value

SYNOPSIS

```
#include <stdlib.h>

char *getenv(const char *name)
```

DESCRIPTION

The **getenv** function searches an "environment list", provided by the host environment, for a string that matches **name**. If a matching string is found, a pointer is returned to the string "value" associated with it. The memory associated with the returned value should not be modified by the programmer. Since it is possible for the host environment to change, subsequent calls to **getenv** (for the same **name**), may return different pointers (and assigned values).

In the **Transputer Toolset** implementation of **getenv**, the "value" is stored in memory obtained from the heap using the "malloc" function. If necessary, the memory may be returned to the heap using a call to "free" (with the pointer returned from the original call to **getenv**).

RETURN VALUE

See above. If no match for **name** is found, **getenv** returns NULL.

RELATED FUNCTIONS

None

GetHiPriQ/GetLoPriQ

save queue pointers

SYNOPSIS

```
#include <conc.h>

void GetHiPriQ(int ptrs[])

void GetLoPriQ(int ptrs[])
```

DESCRIPTION

These functions provide "C" bindings for the Transputer "saveh" (**GetHiPriQ**), and "savel" (**GetLoPriQ**), instructions. The **ptrs** array will have the queue "front" pointer stored in **ptrs**[0], and the queue "back" pointer in **ptrs**[1] upon return. See the appropriate INMOS documentation for more information about the "saveX" instructions.

These functions are implemented "inline". To get a functional version, precede your call with:

```
#undef GetHiPriQ
```

Or,

```
#undef GetLoPriQ
```

RETURN VALUE

None

RELATED FUNCTIONS

SetHiPriQ, SetLoPriQ

gets

read a line

SYNOPSIS

#include <stdio.h>

char *gets(char *ptr)

DESCRIPTION

The **gets** function reads characters from the input stream "stdin", into memory starting at **ptr**. Characters are read until EOF, or a newline character is read (it is discarded). A '\0' character is added after the last character read.

You must ensure that the character array (pointed to by **ptr**), is large enough to hold the worst case input line, otherwise corruption of other data or code is likely. For new work we suggest you use the "fgets" function instead.

RETURN VALUE

The **gets** function returns NULL if an error was detected (or EOF was found before any characters were read). On success **ptr** is returned.

RELATED FUNCTIONS

fgets, fputs, puts

gmtime

determine Coordinated Universal Time

SYNOPSIS

```
#include <time.h>

struct tm *gmtime(const time_t *timer)
```

DESCRIPTION

The **gmtime** function converts the calendar time specified by **timer** into UTC time, and stores the resulting broken-down time into an internal static data area.

RETURN VALUE

The **gmtime** function returns a pointer to the converted broken-down time. The broken-down time data area is static, and is shared with the "asctime", "ctime" and "localtime" functions. The data will be overwritten upon subsequent calls to any of these four functions.

RELATED FUNCTIONS

asctime, ctime, localtime, time

hypot/hypotf

compute Euclidean distance

SYNOPSIS

```
#include <math.h>

double hypot(double x, double y)

float hypotf(float x, float y)
```

DESCRIPTION

The **hypot** function returns the length, of the hypotenuse, of a right triangle whose sides have lengths **x** and **y**. The **hypotf** function does the same thing for single precision arguments. A call to these functions is roughly equivalent to a "sqrt" function call with the following arguments:

```
sqrt(x*x + y*y)
```

Note that the equivalent "sqrt" function call will often be less accurate than a call to **hypot**; particularly when the magnitudes of **x** and **y** are greatly different.

RETURN VALUE

See above. A range error ("errno" is set to ERANGE), or domain error ("errno" is set to EDOM), may occur depending on the parameter values.

RELATED FUNCTIONS

sqrt

HSemP/_HSemP/HSemV/_HSemV
mixed-priority semaphores

SYNOPSIS

```
#include <conc.h>

int HSemP(Semaphore s)

int _HSemP(Semaphore *s)

int HSemV(Semaphore s)

int _HSemV(Semaphore *s)
```

DESCRIPTION

The **HSemP** and **HSemV** macros implement semaphore operations on the Transputer. They are equivalent to the "SemP" and "SemV" functions, but may be used where processes at mixed priority levels will be sharing the same semaphore (**s**). These functions ensure correct mixed priority operation by internally switching to high priority during the semaphore access (with a somewhat higher overhead than the mono-priority versions). The **_HSemV** and **_HSemP** functions are primitives used by the macros, and are generally not called directly.

See the description of the "SemP" and "SemV" functions for more information about using these macros.

RETURN VALUE

These functions return the "use" field from the **s** "Semaphore" structure.

RELATED FUNCTIONS

SemAlloc, SemFree, SemP, SemV

ifft/ifftf

compute inverse FFT transform

SYNOPSIS

#include <math.h>

void ifft(COMPLEX x[], int logsize)

void ifftf(COMPLEXF x[], int logsize)

DESCRIPTION

The **ifft** function performs a radix-2, inverse, Fast Fourier Transform on an input array of complex floating point numbers (**x**). The **ifftf** function does the same thing for an array of single precision floating point numbers. The transform is done in-place, so the output data is also stored into **x**. The **logsize** parameter is the log (base 2), of the number of complex numbers in the array. In the following definition, N is equal to the number of elements in the array (2**__logsize__):

$$x'(k) = \sum_{n=0}^{N-1} x(n)e^{j(2(PI)/N)kn}$$

Note that there is considerable disagreement in the literature about what constitutes the "forward" transform, and what should be called the "inverse" (see the "fft" routine description). Also, for a complete forward-inverse transform pair to produce the original data values, a scaling factor of 1/N must be applied to the data. Some authorities apply it all on the forward or inverse transforms, others apply 1/sqrt(N) to each. With this in mind, the **ifft** and "fft" functions do not perform any scaling. If your application requires it, you must apply it directly to the individual data points produced by one of the transforms.

RETURN VALUE

None. Note that for floating point Transputers, the **ifftf** function has been hand optimized, for speed, in assembly language.

RELATED FUNCTIONS

fft

index

search string for character

SYNOPSIS

#include <string.h>

char *index(const char *ptr, int c)

DESCRIPTION

The **index** function searches a string, pointed to by **ptr**, for the first occurrence of the character **c**. A pointer to where the character was found is returned, or NULL, if a '\0' string terminator was found instead. The '\0' character is considered part of the string and may be searched for. This function is called "strchr" in ANSI libraries, and use of that name is recommended for all new work.

RETURN VALUE

See above

RELATED FUNCTIONS

rindex, strchr, strrchr

is*

classify characters

isalnum
isalpha
isascii
iscntrl
isdigit
isgraph
islower
isprint
ispunct
isspace
isupper
isxdigit

SYNOPSIS

```
#include <ctype.h>

int isalnum(int c)

int isalpha(int c)

int isascii(int c)

int iscntrl(int c)

int isdigit(int c)

int isgraph(int c)

int islower(int c)

int isprint(int c)

int ispunct(int c)

int isspace(int c)

int isupper(int c)

int isxdigit(int c)
```

DESCRIPTION

These macros classify an argument, **c**, as to whether it belongs in a particular set. The macros are designed to evaluate the argument only once, and are safe for use with arguments with side-effects.

The following descriptions cover the individual macros (and indicate what combination of the other macros is equivalent):

isalnum - True for alphanumeric characters (**isalpha** || **isdigit**).

isalpha - True for lower or upper case letters (**islower** || **isupper**).

isascii - True for ASCII characters (0x00-0x7F).

iscntrl - True for control characters (0x00-0x1F, 0x7F).

isdigit - True for a decimal digit ('0'-'9').

isgraph - True for printing characters except space (0x21-0x7E).

islower - True for lower case letters ('a'-'z').

isprint - True for printing characters (0x20-0x7E).

ispunct - True for printing characters other than space (and except anything **isalnum** is true for).

isspace - True for whitespace characters (' ', '\f', '\n', '\r', '\t', '\v').

isupper - True for upper case letters ('A'-'Z').

isxdigit - True for hexadecimal digits ('0'-'9', 'a'-'f', 'A'-'F').

RETURN VALUE

All the macros evaluate to a non-zero truth value if the predicate is satisfied; otherwise zero to indicate failure.

RELATED FUNCTIONS

toascii, tolower, toupper

isort

insertion sort

SYNOPSIS

#include <stdlib.h>

void isort(void *base, size_t nmemb, size_t size, int (*compare)(const void *, const void *))

DESCRIPTION

The **isort** function is an implementation of the "insertion sort" algorithm which has the same calling sequence as the standard "qsort" function. This function may be used to advantage (compared to "qsort"), when the number of items to be sorted is small (less than 10 or 20).

See the "qsort" function description for detailed parameter explanations.

RETURN VALUE

None

RELATED FUNCTIONS

qsort, ssort

kbhit

check console for keystroke

SYNOPSIS

```
#include <stdio.h>

int kbhit(void)
```

DESCRIPTION

The **kbhit** function returns a non-zero value if a key has been pressed on the system console, zero if not. This function is not part of the ANSI library, and isn't supported under many host environments.

RETURN VALUE

See above

RELATED FUNCTIONS

None

labs

compute long absolute value

SYNOPSIS

#include <stdlib.h>

long labs(long l)

DESCRIPTION

The **labs** function returns the long integer absolute value of a long integer argument **l**.

RETURN VALUE

See above

RELATED FUNCTIONS

abs, fabs, fabsf

ldexp/ldexpf

compose floating point number

SYNOPSIS

#include <math.h>

double ldexp(double value, int exp)

float ldexpf(float value, int exp)

DESCRIPTION

The **ldexp** function builds a floating point number by multiplying the **value** by the quantity 2****exp**. The **ldexpf** function does the same thing for a single precision argument.

RETURN VALUE

See above. A range error ("errno" is set to ERANGE), occurs if the number is not representable in the floating point format.

RELATED FUNCTIONS

frexp

ldiv

compute long integer quotient and remainder

SYNOPSIS

```
#include <stdlib.h>

ldiv_t ldiv(long int num, long int denom)
```

DESCRIPTION

The **ldiv** function returns the long integer quotient and remainder of dividing **num** by **denom**. If the division is inexact, the sign of the quotient will be the same as the algebraic quotient, and the magnitude of the quotient will be the largest integer less than the magnitude of the algebraic quotient. If the result can be represented, then:

num = quotient * **denom** + remainder

RETURN VALUE

The quotient and remainder are returned in a structure of type "ldiv_t", which has two long integer members, "quot" and "rem".

RELATED FUNCTIONS

div

localtime

determine local time

SYNOPSIS

#include <time.h>

struct tm *localtime(const time_t *timer)

DESCRIPTION

The **localtime** function converts the calendar time specified by **timer**, into local time, and stores the resulting broken-down time into an internal static data area.

RETURN VALUE

The **localtime** function returns a pointer to the converted broken-down time. The broken-down time data area is static, and is shared with the "asctime", "ctime" and "gmtime" functions. The data will be overwritten upon subsequent calls to any of these four functions.

RELATED FUNCTIONS

asctime, ctime, gmtime, time

log/logf

compute natural logarithm

SYNOPSIS

```
#include <math.h>

double log(double x)

float logf(float x)
```

DESCRIPTION

The **log** function returns the floating point natural logarithm of **x**. The **logf** function does the same thing for a single precision argument.

RETURN VALUE

A domain error ("errno" set to EDOM), occurs if **x** is negative. A range error ("errno" is set to ERANGE), occurs if **x** is zero.

RELATED FUNCTIONS

exp, log10, pow

log10/log10f

compute base-10 logarithm

SYNOPSIS

```
#include <math.h>

double log10(double x)

float log10f(float x)
```

DESCRIPTION

The **log10** function returns the floating point base-10 logarithm of **x**. The **log10f** function does the same thing for a single precision argument.

RETURN VALUE

A domain error ("errno" set to EDOM), occurs if **x** is negative. A range error ("errno" is set to ERANGE), occurs if **x** is zero.

RELATED FUNCTIONS

exp, log, pow

longjmp

non-local jump

SYNOPSIS

#include <setjmp.h>

void longjmp(jmp_buf env, int val)

DESCRIPTION

A call to the **longjmp** function causes the current function and stack environment to be replaced with the conditions which existed when the most recent call to the "setjmp" function occurred which used the same "jmp_buf" structure (**env**). If the "setjmp" function hasn't been called with **env**, or was not called from a function earlier in the set of nested calls, the results are undefined.

After the **longjmp** call is completed, all local variable information present at the previous call to "setjmp" is available, although changes which occurred chronologically after the "setjmp" call will generally be in effect.

RETURN VALUE

After the **longjmp** call, the program continues execution as if it had just returned from "setjmp" with a return value of **val**. Since the return value of "setjmp" is used to distinguish between the initial registration call to "setjmp", and the subsequent activation through the **longjmp** call, the value of **val** must not be zero (it will be mapped into a value of one automatically if zero is used).

RELATED FUNCTIONS

setjmp

lseek

change position within file

SYNOPSIS

```
#include <stdio.h>

long int lseek(int handle, long int offset, int origin)
```

DESCRIPTION

The **lseek** function changes the current read/write position within the file specified by **handle**. The new position is computed as an **offset** (in bytes), from the measuring position specified by **origin**. Three macro definitions in "stdio.h" indicate the legal values for **origin**:

1. SEEK_SET - The value of **offset** is measured from the start of the file.

2. SEEK_CUR - The value of **offset** is measured from the current position in the file.

3. SEEK_END - The value of **offset** is measured from the end of the file.

The **lseek** function is not supported in the ANSI standard library, thus use in new work is discouraged (the "fseek" function is the recommended replacement).

RETURN VALUE

The new read/write position (in bytes), as measured from the start of the file. A negative return value indicates an error.

RELATED FUNCTIONS

close, creat, fseek, ftell, open, read, rewind, write

malloc

allocate heap memory

SYNOPSIS

```
#include <stdlib.h>

void *malloc(size_t size)
```

DESCRIPTION

The **malloc** function allocates a region of heap memory large enough to hold an object of **size** bytes.

RETURN VALUE

If it is impossible to satisfy the request, or **size** is zero, a NULL pointer is returned. Otherwise, a pointer is returned to the start of the allocated region.

RELATED FUNCTIONS

addfree, calloc, cfree, free, realloc

memccpy

string copy with length and terminator

SYNOPSIS

```
#include <string.h>

void *memccpy(const void *dst, const void *src, int c, size_t n)
```

DESCRIPTION

The **memccpy** function copies bytes from **src** to **dst**, until either **n** bytes have been copied, or a character equal to **c** is copied, whichever happens first.

RETURN VALUE

If the character **c** is copied, the return value will be the next address after the copied character in the **dst** string. If **c** is not found, the copy is terminated because of the maximum length **n**, and a NULL pointer is returned.

RELATED FUNCTIONS

memccpy, memcpy, strcpy

memchr

string search with length

SYNOPSIS

#include <string.h>

void *memchr(const void *ptr, int c, size_t n)

DESCRIPTION

The **memchr** function searches at most **n** bytes, starting at **ptr**, for the byte value **c**.

RETURN VALUE

If the character **c** is found, the return value will be a pointer to it. If **c** is not found, NULL is returned.

RELATED FUNCTIONS

memccpy

memcmp

compare bytes

SYNOPSIS

```
#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n)
```

DESCRIPTION

The **memcmp** function compares byte string **s1** against byte string **s2** for **n** bytes, returning zero if they are identical, a negative integer if string **s1** is lexicographically less than string **s2**, and a positive integer otherwise.

RETURN VALUE

See above

RELATED FUNCTIONS

bcmp

memcpy

string copy with length

SYNOPSIS

#include <string.h>

void *memcpy(void *dst, const void *src, size_t n)

DESCRIPTION

The **memcpy** function copies **n** bytes from the location pointed to by **src** to the location pointed to by **dst**.

The **memcpy** function is equivalent to "bcopy", but with a reversed **src** and **dst** argument order.

This function is implemented "inline". To get a functional version, precede your call with:

#undef memcpy

RETURN VALUE

Returns the value of **dst**.

RELATED FUNCTIONS

bcopy, memmove, strcpy

memmove

overlapping string copy with length

SYNOPSIS

```
#include <string.h>

void *memmove(void *dst, const void *src, size_t n)
```

DESCRIPTION

The **memmove** function copies **n** bytes from the location pointed to by **src** to the location pointed to by **dst**. The **memmove** function is like "memcpy", except it correctly handles copying where the **src** and **dst** strings overlap.

RETURN VALUE

Returns the value of **dst**.

RELATED FUNCTIONS

bcopy, memcpy, strcpy

memset

string initialize

SYNOPSIS

```
#include <string.h>

void *memset(void *ptr, int c, size_t n)
```

DESCRIPTION

The **memset** function writes **n** copies of the character **c** into memory starting at **ptr**.

RETURN VALUE

Returns the value of **ptr**.

RELATED FUNCTIONS

bcopy, memcpy, memmove, strcpy

modf/modff

truncate floating point number

SYNOPSIS

```
#include <math.h>

double modf(double x, double *y)

float modff(float x, float *y)
```

DESCRIPTION

The **modf** function breaks a floating point number, **x**, into integral and fractional parts. Both components retain the same sign as **x**. The integral part is stored as a floating point number at the address pointed to by **y**; the fractional part is returned as the value of the function. The **modff** function does the same thing for single precision arguments.

RETURN VALUE

See above.

RELATED FUNCTIONS

None

Move2D/Move2DNonZero/Move2DZero
2D block move

SYNOPSIS

```
#include <conc.h>

void Move2D(void *src, void *dst, int w, int n, int sw, int dw)

void Move2DNonZero(void *src, void *dst, int w, int n, int sw, int dw)

void Move2DZero(void *src, void *dst, int w, int n, int sw, int dw)
```

DESCRIPTION

These functions provide "C" bindings for the two dimensional block move instructions supported by some Transputers. The functions are only available on Transputers which support the instructions.

The arguments to all three functions are identical:

src - The source base address for the move.

dst - The destination base address for the move.

w - The width in bytes of the rows to be copied.

n - The number of rows to be copied.

sw - The source array "stride" width in bytes.

dw - The destination array "stride" width in bytes.

The **Move2D** function uses the "move2dinit" and "move2dall" instructions to perform a two dimensional block move, without worrying about the values of the bytes moved.

The **Move2DNonZero** function is the same as **Move2D**, except only bytes which are non-zero are copied from the source to the destination (uses the "move2dnonzero" instruction instead of "move2dall").

The **Move2DZero** function is the same as **Move2D**, except only bytes which are zero are copied from the source to the destination (uses the "move2dzero" instruction instead of "move2dall").

These functions are implemented "inline". To get a functional version, precede your call with:

#undef Move2D

Or,

#undef Move2DNonZero

Or,

#undef Move2DZero

See the applicable INMOS documentation for more information about the underlying instructions used by these functions.

RETURN VALUE

None

RELATED FUNCTIONS

None

_ns_exit

non-server exit

SYNOPSIS

```
#include <stdlib.h>

void _ns_exit(void)
```

DESCRIPTION

The **_ns_exit** function is an analog to "exit" which is used when a program running on the Transputer isn't connected to a host server. Under these circumstances, (there is no environment to return a status value to), this function simply stops the current process.

This function is one of three functions which are provided for use when no host server is desired. The others:

"_ns_main" - A replacement for "_main" (specified to TLNK as the desired entry point). Does everything the normal "_main" does except request command line arguments from the host.

"_ns_printf" - Equivalent in capabilities to "_printf", except no I/O server is assumed. All output is written to the "_host_chan_out" "hard" channel as plain ASCII text (no protocol).

There are a number of reasons to use these functions in lieu of the full fledged equivalents:

1. They are very simple, and are useful during the early stages of a port of the **Transputer Toolset** (when not everything might be working correctly).

2. They use a very small amount of ram. This is useful when the program being run doesn't require anything beyond "_printf" for I/O, and you wish to get the last ounce of performance out of it by loading everything possible in on-chip ram. A related case is when hardware development is underway, and only on-chip ram is currently functioning (or that is all the design allows the Transputer).

To use these three functions, they must replace the normal protocol-driven host I/O services. This is done by specifying "_ns_main" as the entry point to TLNK, and ensuring that no routine which requires the host server is called during program execution.

RETURN VALUE

None

RELATED FUNCTIONS

_ns_printf

_ns_printf

non-server _printf

SYNOPSIS

```
#include <stdio.h>

void _ns_printf(const char *format, ...)
```

DESCRIPTION

The **_ns_printf** function is an analog to "_printf" which is used when a program running on the Transputer isn't connected to a host server. All output is written to the "_host_chan_out" "hard" channel as plain ASCII text (no protocol).

This function is one of three functions which are provided for use when no host server is desired. The others:

"_ns_main" - A replacement for "_main" (specified to TLNK as the desired entry point). Does everything the normal "_main" does except request command line arguments from the host.

"_ns_exit" - Like, "exit", but doesn't require host server.

There are a number of reasons to use these functions in lieu of the full fledged equivalents:

1. They are very simple, and are useful during the early stages of a port of the **Transputer Toolset** (when not everything might be working correctly).

2. They use a very small amount of ram. This is useful when the program being run doesn't require anything beyond "_printf" for I/O, and you wish to get the last ounce of performance out of it by loading everything possible in on-chip ram. A related case is when hardware development is underway, and only on-chip ram is currently functioning (or that is all the design allows the Transputer).

To use these three functions, they must replace the normal protocol-driven host I/O services. This is done by specifying "_ns_main" as the entry point to TLNK, and ensuring that no routine which requires the host server is called during program execution.

WARNING: UNLIKE THE REGULAR "_printf", THIS FUNCTION IS NOT SAFE TO USE CONCURRENTLY. RESTRICT YOUR USE OF "_ns_printf" TO A SINGLE PROCESS.

RETURN VALUE

None

RELATED FUNCTIONS

_ns_exit

open

low level file open

SYNOPSIS

```
#include <stdio.h>
#include <fcntl.h>

int open(char *path, int flags, int mode)
```

DESCRIPTION

The **open** function opens the file specified by **path**. The allowed operations on the file are controlled by the setting of **flags**, and the **mode** parameter is used to specify the desired access permissions when **open** is told to create the file.

The following symbolic bit mask definitions are used with the **flags** parameter:

O_APPEND - Position the file pointer to the end of the file prior to performing a write.
O_BINARY - Open the file in untranslated mode (no CR/LF mapping nonsense).
O_CREAT - If the file doesn't exist create it and open it for writing. See the following description for the mode **flag**.
O_EXCL - Return an error if the file exists (used with O_CREAT).
O_NONBLOCK - Return if the I/O would block.
O_RDONLY - Open file only for reading.
O_RDWR - Open the file for reading and writing.
O_TEXT - Open the file in translated mode (perform CR/LF mapping if host OS requires it).
O_TRUNC - Open the file and toss contents.
O_WRONLY - Open file only for writing.

If the values of the above macros are bitwise OR'ed together, and passed as the **flags** parameter, the actions are presumed to be combined (not all combinations are legal however).

Note that not all of the macro definitions are usable with all the host operating systems which exist. In particular, the O_BINARY and O_TEXT flags are commonly restricted to PC's and VMS, while the O_NONBLOCK flag is used mostly with UNIX and derivatives. If a flag has no meaning in a particular environment it may still be used in programs without any adverse effects.

The **mode** parameter applies only when the O_CREAT bit is part of the **flags** argument, and determines what permission setting the file should be given when closed (see the "creat" function description).

The following symbolic bit mask macro definitions are allowed with **mode**:

S_IREAD - Allow file reading
S_IWRITE - Allow file writing

If the values of the above two macros are bitwise OR'ed together, and passed as the **mode** parameter, the file is assumed to allow both reading and writing.

As this function is not supported in the ANSI standard library, its use in new work is discouraged (see the "fopen", "fclose", "fread", and "fwrite" functions for the recommended replacements).

RETURN VALUE

The **open** function returns a positive value (a "handle"), for the file if the open/create is successful, otherwise (-1) is returned and "errno" is set appropriately.

RELATED FUNCTIONS

close, creat, dup, dup2, read, write

perror

write "errno" message

SYNOPSIS

```
#include <stdio.h>

void perror(const char *ptr)
```

DESCRIPTION

The **perror** function writes the error message corresponding to the current value of "errno" to the "stderr" output stream. The error message printed is composed of the string pointed to by **ptr** (if non-NULL), followed by a colon and space, and the error message which corresponds to the value of "errno".

There is a basic problem when using this function (or "strerror"), to print the meaning of a value in "errno":

1. The value of "errno" may be the result of a host server I/O error; whereupon the host environment definitions for "errno" should be used.

2. The value of "errno" may be a result of a error detected with the runtime library on the Transputer; thus the **Transputer Toolset** "errno" definitions should be used.

The solution actually adopted was to only report the "errno" values (and naming scheme), which are used by the library functions running on the Transputer. For errors which happen on the host system, the results of the **perror** function aren't defined.

A further problem with "errno" arises because it is a shared global variable. In a program consisting of many processes, each of which uses the same "errno", how do you determine which process detected the error?

RETURN VALUE

None

RELATED FUNCTIONS

strerror

PFork/PForkHigh/PForkLow

process forking

SYNOPSIS

void PFork(Forkblk f, PDes p)

void PForkHigh(Forkblk f, PDes p)

void PForkLow(Forkblk f, PDes p)

DESCRIPTION

These three routines start a new process at either the current priority (**PFork**), high priority (**PForkHigh**), or low priority (**PForkLow**). These routines are part of the "Fork/Join concurrency model" library package provided with the **Transputer Toolset**. The routines take two parameters; a "Forkblk" structure **f**, and a process descriptor structure **p**.

The "Forkblk" structure, **f**, must have been previously initialized with a call to "PForkInit". The "PDes" structure, **p**, must have been previously initialized with a call to "PSetup". The forked process terminates by returning from the initial entry function specified to "PSetup" when **p** was initialized.

WARNING: THESE FUNCTIONS ARE IMPLEMENTED AS MACROS! THEY SHOULD NOT BE CALLED WITH A "p" PARAMETER WHOSE EVALUATION INVOLVES "SIDE EFFECTS", AS THE PARAMETER WILL BE EVALUATED MORE THAN ONCE! DOING SOMETHING LIKE:

```
PFork(f,PSetup(...stuff...));
```

WILL NOT WORK! DO THE FOLLOWING INSTEAD:

```
PDes p;
...
p = PSetup(...stuff...);
PFork(f,p);
```

See the **Transputer Concurrency (Fork/Join Model)** section at the beginning of this manual for more information.

RETURN VALUE

These routines have no return value and are implemented as macros.

RELATED FUNCTIONS

PForkInit, PHalt, PJoin, PRun, PSetup, PStop

PForkInit
initialize process fork structure

SYNOPSIS

void PForkInit(Forkblk f, int n)

DESCRIPTION

The **PForkInit** macro initializes a "Forkblk" structure. This function is part of the "Fork/Join concurrency model" library package provided with the **Transputer Toolset**. The function takes two parameters; a "Forkblk" structure to initialize, **f**, and the number of processes which will be forked, **n**.

The fork/join operation is similar to using one of the "ProcPar*" functions, except the originating parent process continues execution in parallel with the forked child. This avoids the allocation of an extra stack (workspace), and also the need for an initial entry function for the extra process. The parameter, **n**, thus specifies the total number of outstanding processes which will be synchronized with the later "PJoin" call (including the original parent process). After the "PJoin" call (which will block the parent until all child processes terminate), the parent process will awake and continue execution. Note that all the child processes need not be forked at the same time. If the number of children to be forked is not known in advance, the number can be adjusted at run time with the proviso that it always remain positive. For example, the following program skeleton shows one approach to adjusting the number dynamically:

```
PForkInit(f,1);
for( ... )
     {
     ...
     p = PSetup(...)
     pri = ProcToHigh();
     f.count++;
     if(pri)
          (void) ProcToLow();
     PFork(f,p);
     ...
     }
PJoin(f);
```

WARNING: THIS FUNCTION IS IMPLEMENTED AS A MACRO! IT SHOULD NOT BE CALLED WITH A "f" PARAMETER WHOSE EVALUATION INVOLVES "SIDE EFFECTS", AS THE PARAMETER WILL BE EVALUATED MORE THAN ONCE!

See the **Transputer Concurrency (Fork/Join Model)** section at the start of this manual for more information.

RETURN VALUE

This routine has no return value and is implemented as a macro.

RELATED FUNCTIONS

PFork, PForkHigh, PForkLow, PHalt, PJoin, PRun, PSetup, PStop

PHalt

kill current process and save descriptor

SYNOPSIS

```
#include <conc.h>

void PHalt(PDes *pid)
```

DESCRIPTION

The **PHalt** function kills the current process and stores the terminated process descriptor in **pid**. Since the process descriptor is saved, the process may be later restarted using "PRun". This function is part of the "Fork/Join concurrency model" library package provided with the **Transputer Toolset**.

This function is implemented "inline". To get a functional version, precede your call with:

```
#undef PHalt
```

RETURN VALUE

None

RELATED FUNCTIONS

PFork, PForkHigh, PForkInit, PForkLow, PJoin, PRun, PSetup, PStop

PJoin

merge previously forked processes

SYNOPSIS

```
#include <conc.h>

void PJoin(Forkblk *f)
```

DESCRIPTION

The **PJoin** function blocks until all processes which are associated with the "Forkblk" structure (**f**), terminate. The "Forkblk" structure is assumed to have been initialized by a call to "PForkInit", and subsequent process fork actions (calls to "PFork", "PForkHigh" or "PForkLow"). This function is part of the "Fork/Join concurrency model" library package provided with the **Transputer Toolset**.

See the **Transputer Concurrency (Fork/Join Model)** section at the start of this manual for more information.

RETURN VALUE

None

RELATED FUNCTIONS

PFork, PForkHigh, PForkInit, PForkLow, PHalt, PRun, PSetup, PStop

pow/powf

compute x to the y power

SYNOPSIS

#include <math.h>

double pow(double x, double y)

float powf(float x, float y)

DESCRIPTION

The **pow** function returns the the value of **x** raised to the power **y**. The **powf** function does the same thing for single precision arguments.

RETURN VALUE

See above. A domain error occurs if **x** is negative and **y** is non-integral ("errno" is set to EDOM). A range error ("errno" is set to ERANGE), may also occur.

RELATED FUNCTIONS

exp, log, log10

printf

formatted write

SYNOPSIS

```
#include <stdio.h>

int printf(const char *format, ...)
```

DESCRIPTION

The **printf** function writes to the "stdout" stream. The write is formatted under the control of the **format** argument, which specifies how subsequent arguments (if any), are to be converted for output. If there are too few arguments to match the requirements of the **format** string, the results are undefined. If arguments remain after meeting the requirements of the **format** string, they are evaluated and ignored. The **printf** function returns when it finishes evaluating the **format** string.

The **format** string consists of a '\0' terminated character sequence composed of ordinary characters (other than '%'), which are simply copied to "stdout". The '%' character is used to mark the beginning of a format specification.

Format Specification

A format specification is generally used to describe how an argument to "printf" should be converted, formatted, and written to "stdout". After the '%' character, the following fields make up a format specification:

1. Zero or more "flags". These are general qualifiers to the format specification.

2. An optional, non-zero, decimal integer specifying a minimum field width for the result. If the converted and formatted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the "left adjustment" flag, described later, has been given), to the minimum field width.

3. An optional "precision" qualifier which gives the minimum number of digits to appear for the 'd', 'i', 'o', 'u', 'x', and 'X' conversions, the number of digits to appear after the decimal point for 'e', 'E', and 'f' conversions, the maximum number of significant digits for the 'g' and 'G' conversions, or the maximum number of characters to be written from a string with the 's' conversion. The "precision" qualifier takes the form of a period, followed by an optional decimal integer (zero is assumed if the integer is omitted).

4. An optional 'h' qualifier indicating that the argument to the following 'd', 'i', 'o', 'u', 'x', or 'X' conversion will be a "short int" or "unsigned short int" (or that the argument to a 'n' conversion will be a "short int" pointer). In place of the 'h', a 'l' may appear, which has the same general meaning as 'h', except all arguments are assumed to be "long", rather than "short". The final optional alternative to the 'h' and 'l' qualifiers is 'L', which indicates the argument to the following 'e', 'E', 'f', 'g', or 'G' conversion will be of type "long double", rather than "double". If the 'h', 'l', or 'L' qualifiers is used with a conversion other than one of the ones listed, the results are undefined.

5. Finally, the character which specifies which conversion is to be performed.

Note that either the minimum field width, or the "precision" (or both), may use '*' in place of the decimal integer, to allow the values to be dynamically configured. The '*' is replaced with the value of an additional argument to **printf** (at run time). If either '*' option is used, corresponding integer argument(s) must be supplied to **printf** (in order), before the argument which is the subject of the conversion. A negative field width argument is taken as a '-' flag, followed by a positive field width. A negative precision is ignored.

Flags

The legal "flags" are:

'-' - Left justify the conversion within the field.

'+' - Force a sign to be written for signed conversions.

' ' - Prepend a space before the result for signed conversions if the result was non-negative (and a '+' sign wouldn't normally be printed). The '+' flag takes precedence over this flag if both are used.

'#' - Use an "alternate" form for the conversion. For 'o', this increases the precision to force the first digit of the result to be zero. For 'x' or 'X', a non-zero result will have '0x' or '0X' (respectively), prepended. For the 'e', 'E', 'f', 'g', and 'G' conversions, the result will always have a decimal point (the decimal point is not normally written unless at least one digit follows it). For the 'g' and 'G' conversions, trailing zeros (otherwise pruned), will be written. The '#' flag produces undefined results with conversions other than these.

'0' - For the 'd', 'i', 'o', 'u', 'x', 'X', 'e', 'E', 'f', 'g', and 'G' conversions, leading zeros (following any base or sign indication), are used to pad the result to the field width. This is in lieu of the padding with spaces which would normally occur. The '-' flag takes precedence over the '0' flag. If an explicit precision is specified for the 'd, 'i', 'o', 'u', 'x', and 'X' conversions, the '0' flag is also ignored. The '0' flag produces undefined results with conversions other than these.

Conversions

The following are the legal conversion characters:

'c' - A "int" argument is converted to "unsigned char", and the resulting character written out.

'd'
'i'
'o'
'u'
'x'
'X' - A "int" argument is converted to signed decimal ('d' or 'i'), unsigned octal ('o'), unsigned decimal ('u'), or unsigned hexadecimal ('x' or 'X'). The letters "abcdef" are used for the six additional digits needed with 'x' ("ABCDEF" with 'X'). The "precision" qualifier specifies the minimum number of digits to appear (expanded, if necessary, with leading zeros). If no "precision" is specified, the default is one digit. If zero is converted with a precision of zero, nothing will be printed.

'e'
'E' - A "double" argument is converted in the style [-]d.dddeSdd ('S' represents a '+' or '-' sign). The 'E' form will substitute 'E' for the 'e' exponent field character. The number of digits after the decimal point is equal to the "precision". If no "precision" is specified, the default is six digits. If the "precision" is zero, no decimal point is written. The exponent always contains at elast two digits. The result is rounded to the number of digits written.

'f' - A "double" argument is converted to decimal notation in the style [-]ddd.ddd, where the number of digits after the decimal point is equal to the "precision". If no "precision" is specified, the default is six digits. If the "precision" is zero, no decimal point is written. If a decimal point appears, it will be followed by at least one digit. The result is rounded to the number of digits written.

'g'
'G' - A "double" argument, to be converted with 'g', is equivalent to using either 'f' or 'e', depending on the characteristics of the value being converted. If the 'G' conversion is used, the conversion is equivalent to using 'E'. The "precision" specifies the number of significant digits (minimum one). The 'e' conversion will only be used if the exponent is less than (-4), or greater than (or equal to), the "precision". Trailing zeros are removed from the fractional part of the result. A decimal point only appears if followed by a digit.

'n' - A "integer" pointer argument is assumed. The current number of characters written (so far), by this **printf** call, is stored at the address specified in the argument.

'p' - A "void" pointer is displayed in an implementation dependent manner (hexadecimal notation for the **Transputer Toolset**).

's' - The argument is assumed to be a pointer to an array of characters (a '\0' terminated "string"). No more characters are written than the "precision" (if specified).

'%' - A '%' is written to "stdout" (ie. a "%%" format specification writes a single '%').

Conversion specifications other than these produce undefined results.

Don't Worry

In no case does a small field width cause truncation of a result! If the result of a conversion is wider than the field, the field width is expanded. There is, however, a maximum field width (per-conversion), of 509 characters in this implementation.

RETURN VALUE

The number of characters written (negative values indicate an error).

RELATED FUNCTIONS

fprintf, sprintf, vfprintf, vprintf, vsprintf

EXAMPLES

To print a date and time in the form "Saturday, June 2, 10:01", where "weekday" and "month" are pointers to '\0' terminated strings:

```
#include <stdio.h>

print_date(char *weekday, char *month, int day, int hour, int min)
      {
      printf("%s, %s %d, %.2d:%.2d\n", weekday, month, day, hour,
min);
      }
```

To print PI to 4 decimal places:

```
#include <stdio.h>
#include <math.h>

main()
      {
      printf("pi=%.4f", 4.0 * atan(1.0));
      }
```

_printf

simple formatted write

SYNOPSIS

#include <stdio.h>

int _printf(const char *format, ...)

DESCRIPTION

The **_printf** function is a stripped down version of the normal "printf" function. It is mainly used where the full facilities of "printf" are not required, and the vastly smaller amount of memory consumed is helpful. It doesn't support floating point, option "flags", field widths or precision values. It also doesn't support any return value (despite the prototype). See the "printf" function description for more information.

RETURN VALUE

None

RELATED FUNCTIONS

printf

ProcAfter

suspend process until specified time

SYNOPSIS

#include <conc.h>

void ProcAfter(int time)

DESCRIPTION

The **ProcAfter** function blocks the current process until the value of the timer for the current priority level is after **time**. Note that the timer runs at different rates for high (1 uS/tick), and low priority (64 uS/tick), processes.

This function is implemented "inline". To get a functional version, precede your call with:

#undef ProcAfter

For additional information see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

None

RELATED FUNCTIONS

ProcWait, SetTime, Time

ProcAlloc

allocate asynchronous process

SYNOPSIS

```
#include <conc.h>

Process *ProcAlloc(void (*f)(), int s, int np, int p1, int p2, ...)
```

DESCRIPTION

The **ProcAlloc** function builds a stack frame for a new process using memory obtained from the heap. It also allocates memory for a "Process" structure which will define the stack frame characteristics. The stack frame is set to a total size of **s** bytes, and is initialized with the values of **np** words worth of integral parameters (**p1**, **p2**, ...). The initial function to execute in the new process is set using function pointer **f**; it will be passed the parameters (**process_ptr**, **p1**, **p2**, ...), upon process creation. The **process_ptr** parameter will be of type "Process *" and will be the address of the "Process" structure returned by the **ProcAlloc** call.

The **s** (stack size), parameter may be set to zero, in which case 64K bytes will be allocated on 32 bit Transputers (2K bytes on 16 bit Transputers). Remember that the default heap size is only 128K bytes (32K bytes on the 16 bit Transputers), and counting overhead, only one 64K region may be successfully allocated on a 32 bit Transputer. Modification of the default heap size will allow allocation up to the limits of physical memory availability (see the **Heap Management** section at the beginning of this manual for more information). The actual amount of stack space required will vary based on the demands the process places on "auto" variables, and the nesting depth of function calls, etc. Keep in mind, for example, that the "printf" family of library functions requires 600 or 700 bytes of stack space! There is also a "minimum" stack size specification which is allowed based on having space to set up the function linkage, and store the parameters, and the requested allocation will be increased to this "minimum" if necessary.

On successful allocation, **ProcAlloc** returns a pointer to an initialized "Process" structure (memory for which it has also allocated from the heap). This structure may be later used with one of the process creation functions such as "ProcRun", "ProcRunHigh", "ProcRunLow", "ProcPar", "ProcParList", or "ProcPriPar".

When (or if), a process allocated in this fashion is no longer in use, the memory used may be returned to the heap using the "ProcFree" function. If it is just desired to change the parameters in the process, a call to "ProcParam" may be used (without requiring returning the old stack frame and getting a new one).

Note that if a more "static" approach to stack frame creation is desired, the "ProcInit" function should be considered in lieu of **ProcAlloc**. If "ProcInit" is used, the "Process" structure may also be statically allocated and the use of the heap entirely avoided. The "Fork/Join" functions ("PFork", "PJoin", PSetup", "PRun", ...), could also be used to completely eliminate the need for a "Process" structure.

For additional information about process primitives, see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

The **ProcAlloc** function returns NULL if it was unable to allocate the requested stack frame. Otherwise, a pointer to the initialized "Process" structure is returned.

RELATED FUNCTIONS

PFork, PForkHigh, PForkInit, PForkLow, PJoin, ProcFree, ProcInit, ProcPar, ProcParam, ProcParList, ProcPriPar, ProcRun, ProcRunHigh, ProcRunLow, PRun, PSetup

ProcAlt*

determine the status of channels

ProcAlt
ProcAltList
ProcSkipAlt
ProcSkipAltList
ProcTimerAlt
ProcTimerAltList

SYNOPSIS

```
#include <conc.h>

int ProcAlt(Channel *c1, ...)

int ProcAltList(Channel **clist)

int ProcSkipAlt(Channel *c1, ...)

int ProcSkipAltList(Channel **clist)

int ProcTimerAlt(int time, Channel *c1, ...)

int ProcTimerAltList(int time, Channel **clist)
```

DESCRIPTION

These functions allow a process to determine the status of one or more Transputer input channels.

ProcAlt, **ProcSkipAlt**, and **ProcTimerAlt** take an explicit NULL terminated list of pointers to channels as parmeters. **ProcAltList**, **ProcSkipAltList**, and **ProcTimerAltList** take a NULL terminated array of pointers to channels as a parameter. Note that the channel pointers are prioritized based on the order they are listed in.

ProcAlt and **ProcAltList** cause the current process to block until one of the channels in its argument list is ready for input. On completion, the routine returns a zero based index into the parameter list for the ready channel.

ProcSkipAlt and **ProcSkipAltList** check specified channels. If one of the channels is ready for input, a zero based index into the parameter list is returned, otherwise (-1) is returned. These routines do not block waiting for one of the channels, they return immediately.

ProcTimerAlt and **ProcTimerAltList** block the current process until one of the channels is ready for input or the value of the clock (as read by the "Time" function), is after the **time** parameter. If the routine times out, a -1 is returned, otherwise a zero based index into the parameter list is returned.

For additional information see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

See above

RELATED FUNCTIONS

ChanIn, ChanInChar, ChanInInt, ChanOut, ChanOutChar, ChanOutInt, ProcAfter, ProcWait, Time

ProcCall

call a function with a new workspace

SYNOPSIS

```
#include <conc.h>

int ProcCall(int (*func), void *ws, int wssize, int p1, ...)
```

DESCRIPTION

The **ProcCall** function allows you to call a function and specify the region of memory it should use for its workspace (stack). A pointer to the function to call is passed as **func**, the start of the new workspace to use as **ws**, and the workspace size as **wssize** (in bytes). Up to five integer parameters may be passed to the function (**p1**, ...).

Because the Transputer has internal ram which is MUCH faster to access than normal ram, this function has been implemented to allow the programmer to dynamically utilize this scarce resource. Moving the workspace of computationally intensive functions into the fast ram is usually a good first cut at improving execution speed.

It is not uncommon for "C" programs to execute 40% faster when the workspace is moved into internal memory. The overhead of **ProcCall** is about 5 times the overhead of a normal function call.

The fast ram may be used in a static fashion for the stack (as well as program and data storage), by directives to the TLNK linker. See the **TLNK Transputer Linker User Guide** for more information about these possibilities.

For additional information see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**. Note that the "internal" keyword used in the example in his paper is not needed with the **Transputer Toolset**, and is replaced by directives to TLNK.

RETURN VALUE

The value returned is that produced by the call to **func**.

RELATED FUNCTIONS

None

EXAMPLE

Call a function named "compute" with one parameter (the integer value 13):

```
#include <conc.h>

#define SIZE 512
char stack[SIZE];
extern int compute();

main()
      {
      int result;

      result = ProcCall(compute, stack, SIZE, 13);
      }
```

In this case, "compute" is executed with its stack stored in 512 bytes of memory (the "stack" array). Often, the array reference is replaced by the physical address of the desired region in internal ram. You must ensure that the region you specify is both, unused during the execution time of this function, and large enough for the stack space the function (and any nested calls), require.

ProcGetPriority

get current process priority level

SYNOPSIS

```
#include <conc.h>

int ProcGetPriority(void)
```

DESCRIPTION

The **ProcGetPriority** function atomically returns the current priority level.

This function is implemented "inline". To get a functional version, precede your call with:

```
#undef ProcGetPriority
```

For additional information see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

0 if the current process is high priority, 1 if low.

RELATED FUNCTIONS

ProcToLow, ProcToHigh

ProcFree

deallocate asynchronous process

SYNOPSIS

```
#include <conc.h>

void ProcFree(Process *p)
```

DESCRIPTION

The **ProcFree** function returns a "Process" structure pointed to by **p** (and the associated stack frame it defines), to the heap free memory pool. The structure pointed to by **p** MUST have been originally obtained by a call to "ProcAlloc", or the results are undefined. You should also ensure that no currently running (or ever executable), process is using the stack frame being deallocated.

For additional information about process primitives, see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

None

RELATED FUNCTIONS

ProcAlloc, ProcInit, ProcPar, ProcParam, ProcParList, ProcPriPar, ProcRun, ProcRunHigh, ProcRunLow

ProcInit

initialize asynchronous process

SYNOPSIS

```
#include <conc.h>

void ProcInit(Process *p, void (*f)(), void *sp, int s, int np, int p1, int p2, ...)
```

DESCRIPTION

The **ProcInit** function takes a pointer **p** to a "Process" structure, and related information about the process, and initializes the "Process" structure accordingly. The required process information is:

1. A pointer **f** to the function to call upon process creation.

2. A stack frame pointed to by **sp**, which is **s** bytes long.

3. A list of **np** words worth of integral parameters to be passed to the initial function (**p1**, **p2**, ...).

The **ProcInit** function is called by "ProcAlloc" to initialize the stack frames it allocates from the heap. **ProcInit** may be called directly when you wish to use memory obtained elsewhere for the stack frame and "Process" structure.

For additional information about process primitives, see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

None

RELATED FUNCTIONS

ProcAlloc, ProcFree, ProcPar, ProcParam, ProcParList, ProcPriPar, ProcRun, ProcRunHigh, ProcRunLow

ProcPar/ProcParList

run asynchronous processes

SYNOPSIS

```
#include <conc.h>

void ProcPar(Process *p1, Process *p2, ...)

void ProcParList(Process **plist)
```

DESCRIPTION

The **ProcPar** function takes a NULL terminated parameter list of "Process" pointers, and runs all the associated processes (**p1**, **p2**, ...), in parallel. Control is returned to the calling function when all the initiated processes terminate. The **ProcParList** function does the same thing but uses a pointer to a NULL terminated vector of "Process" pointers (**plist**), as a parameter instead.

In either case, the processes are run at the same priority as the calling function.

For additional information about process primitives, see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

None

RELATED FUNCTIONS

ProcAlloc, ProcFree, ProcInit, ProcParam, ProcPriPar, ProcRun, ProcRunHigh, ProcRunLow

ProcParam

modify asynchronous process parameters

SYNOPSIS

```
#include <conc.h>

void ProcParam(Process *p, int p1, int p2, ...)
```

DESCRIPTION

The **ProcParam** function takes a pointer **p** to a "Process" structure, and a set of integral parameters (**p1**, **p2**, ...), for the function to be called by that process, and initializes the process stack frame so that the function will get the parameters when the process is created. The "Process" structure must have been previously initialized by either "ProcAlloc" or "ProcInit" for correct operation. The **ProcParam** function is used when it is desired to re-use an existing "Process" structure and stack frame and just change the parameters to the function call. The stack frame must not be in use by a running process or the results are undefined.

For additional information about process primitives, see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

None

RELATED FUNCTIONS

ProcAlloc, ProcFree, ProcInit, ProcPar, ProcParList, ProcPriPar, ProcRun, ProcRunHigh, ProcRunLow

ProcPriPar

run mixed priority processes

SYNOPSIS

```
#include <conc.h>

void ProcPriPar(Process *phigh, Process *plow)
```

DESCRIPTION

The **ProcPriPar** function takes two "Process" pointer arguments, and runs the associated processes in parallel. The first process argument (**phigh**), is run as a high priority process, the second (**plow**), is run at low priority. Control is returned to the calling function when both processes terminate.

For additional information about process primitives, see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

None

RELATED FUNCTIONS

ProcAlloc, ProcFree, ProcInit, ProcPar, ProcParam, ProcParList, ProcRun, ProcRunHigh, ProcRunLow

ProcReschedule

suspend and delay current process

SYNOPSIS

```
#include <conc.h>

void ProcReschedule(void)
```

DESCRIPTION

The **ProcReschedule** function blocks the current process and places it at the end of the process queue at the current priority. This function may be used to implement a "busy wait" mechanism while giving other processes a chance to execute. Where possible, you should avoid this function in favor of one of the more efficient, and elegant, approaches supported by the Transputer hardware. For example, if you need to wait on input data on a channel, use one of the "ProcAlt" family of functions instead.

This function is implemented "inline". To get a functional version, precede your call with:

```
#undef ProcReschedule
```

For additional information see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

None

RELATED FUNCTIONS

None

ProcRun/ProcRunHigh/ProcRunLow
run asynchronous process

SYNOPSIS

```
#include <conc.h>

void ProcRun(Process *p)

void ProcRunHigh(Process *p)

void ProcRunLow(Process *p)
```

DESCRIPTION

These functions create a new process using the information in the "Process" structure pointed to by **p**.

The **ProcRun** function causes the new process to have the same priority as the process which calls **ProcRun**. The **ProcRunHigh** function forces the process to execute at high priority, while **ProcRunLow** forces low priority execution.

The "Process" structure must have been previously initialized by a call to "ProcAlloc" or "ProcInit" (possibly modified by a call to "ProcParam").

For additional information about process primitives, see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

None

RELATED FUNCTIONS

ProcAlloc, ProcInit, ProcPar, ProcParam, ProcParList, ProcPriPar

ProcStop

kill current process

SYNOPSIS

#include <conc.h>

void ProcStop(void)

DESCRIPTION

The **ProcStop** function kills the current process. As the process descriptor information is not saved (see the "Phalt" function), there is no way to later restart a process killed in this fashion.

This function is implemented "inline". To get a functional version, precede your call with:

#undef ProcStop

For additional information see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

This function is equivalent to the "PStop" function.

RETURN VALUE

None

RELATED FUNCTIONS

PHalt, PStop

ProcToHigh

change to high priority

SYNOPSIS

```
#include <conc.h>

int ProcToHigh(void)
```

DESCRIPTION

The **ProcToHigh** function forces the current process to execute at high priority. It is designed to be used with the "ProcToLow" function to protect critical code sections.

RETURN VALUE

The previous priority level is returned: zero if it was already at high priority, one if it was at low priority.

RELATED FUNCTIONS

ProcToLow

ProcToLow

change to low priority

SYNOPSIS

```
#include <conc.h>

int ProcToLow(void)
```

DESCRIPTION

The **ProcToLow** function forces the current process to execute at low priority. It is designed to be used with the "ProcToHigh" function to protect critical code sections.

RETURN VALUE

The previous priority level is returned: one if it was already at low priority, zero if it was at high priority.

RELATED FUNCTIONS

ProcToHigh

ProcWait

suspend process for specified time

SYNOPSIS

```
#include <conc.h>

void ProcWait(int time)
```

DESCRIPTION

The **ProcWait** function blocks the current process for at least **time** timer ticks. Note that the timer runs at different rates for high (1 uS/tick), and low priority (64 uS/tick), processes.

This function is implemented "inline". To get a functional version, precede your call with:

```
#undef ProcWait
```

For additional information see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

None

RELATED FUNCTIONS

ProcAfter, SetTime, Time

PRun

process fork primitive

SYNOPSIS

#include <conc.h>

void PRun(PDes p)

DESCRIPTION

The **PRun** function takes an initialized "PDes" structure, **p**, and starts the associated process. This function is part of the "Fork/Join concurrency model" library package provided with the **Transputer Toolset**, and is a primitive used by the "PFork", "PForkHigh", and "PForkLow" macros. Boiled down, this function is a "C" binding for the Transputer "runp" instruction.

The "PDes" structure, **p**, must have been initialized with a previous call to the "PSetup" function. Note that "PSetup" doesn't provide the low order priority bit (if needed). To run a process at low priority you execute something like:

```
PRun(PSetup(...) | 1);
```

Other process descriptors which may be used with **PRun** come from functions such as "PHalt", "ChanReset", etc. See the appropriate INMOS documentation concerning the "runp" instruction for more information.

This function is implemented "inline". To get a functional version, precede your call with:

#undef PRun

See the **Transputer Concurrency (Fork/Join Model)** section at the start of this manual for more information.

RETURN VALUE

None

RELATED FUNCTIONS

PFork, PForkHigh, PForkInit, PForkLow, PHalt, PJoin, PSetup, PStop

PSetup

initialize process fork process

SYNOPSIS

```
#include <conc.h>

PDes PSetup(void *ws, void (*func)(), int wsize, int psize, ...)
```

DESCRIPTION

The **PSetup** function initializes a workspace and process descriptor for used with a later "process fork" operation ("PFork", "PForkHigh" or "PForkLow"). The "PDes" structure returned is an initialized process descriptor which points to the new "forkable" process created by **PSetup**. The **PSetup** function is analogous to "ProcInit", except the "Process" structure is not required. The parameters include a pointer to the memory area to use for the new process workspace (stack), **ws**, the size of the workspace in bytes, **wsize**, a pointer to the function to serve as entry point for the new process, **func**, and a specification of how many words worth of memory the parameters to be passed to the function will occupy, **psize**. If **psize** is non-zero, the desired parameters should follow **psize** in the function argument list.

The **PSetup** function is part of the "Fork/Join concurrency model" library package provided with the **Transputer Toolset**. See the **Transputer Concurrency (Fork/Join Model)** section at the start of this manual for more information.

RETURN VALUE

See above

RELATED FUNCTIONS

PFork, PForkHigh, PForkInit, PForkLow, PHalt, PJoin, PRun, PStop

PStop

kill current process

SYNOPSIS

```
#include <conc.h>

void PStop(void)
```

DESCRIPTION

The **PStop** function kills the current process. As the process descriptor information is not saved (see the "PHalt" function), there is no way to later restart a process killed in this fashion. This function is part of the "Fork/Join concurrency model" library package provided with the **Transputer Toolset**.

This function is implemented "inline". To get a functional version, precede your call with:

```
#undef PStop
```

This function is equivalent to the "ProcStop" function.

RETURN VALUE

None

RELATED FUNCTIONS

PFork, PForkHigh, PForkInit, PForkLow, PHalt, PJoin, PRun, PSetup

putc/putchar

write a character

SYNOPSIS

#include <stdio.h>

int putc(int c, FILE *stream)

int putchar(int c)

DESCRIPTION

The **putc** macro writes the character specified by c to the output **stream**. Since **putc** is defined to be a macro, the arguments may be evaluated more than once, and must not have side-effects. To get a functional version, use either the "fputc" equivalent, or precede the function call with:

#undef putc

The putchar macro is identical to putc, but without the stream argument ("stdout" is assumed). The same precautions relative to argument side effects (and solutions), apply.

RETURN VALUE

These functions return the character written (or EOF on error).

RELATED FUNCTIONS

fgetc, fputc, getc, getchar, ungetc

puts

write a line

SYNOPSIS

#include <stdio.h>

int puts(const char *ptr)

DESCRIPTION

The **puts** function writes the string pointed to by **ptr** to the "stdout" stream. A newline is written to replace the terminating '\0' character.

RETURN VALUE

The **puts** function returns EOF if an error occurs; otherwise a non-negative value.

RELATED FUNCTIONS

fgets, fputs, gets

qsort

quick sort

SYNOPSIS

#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size, int (*compare)(const void *, const void *))

DESCRIPTION

The **qsort** function is an implementation of the "quick sort" algorithm. It sorts an array of **nmemb** objects, the first object of which **base** points to. Each object is **size** bytes in size. The array is sorted in ascending order using a programmer supplied comparison function (pointed to by **compare**).

The **compare** function takes two arguments which point to two objects to be compared. The **compare** function should return a positive value if the first object is considered to be "greater-than" the second object; a value of zero if the objects are considered to be "equal"; otherwise a negative value.

The order "equal" elements have in the resulting sorted array is undefined (in sort terminology, **qsort** is "unstable").

Two other sort routines are provided with the same calling sequence as **qsort**, for use where the data set matches special circumstances. See the "isort" and "ssort" descriptions for more information.

RETURN VALUE

None

RELATED FUNCTIONS

isort, ssort

rand

integral random number

SYNOPSIS

#include <stdlib.h>

int rand(void)

DESCRIPTION

The **rand** function returns a sequence of pseudo-random integers in the range of zero to 2**31. No other library function calls **rand**; but the "frand" function shares the same "seed". The "seed" may be set for **rand** using the "srand" function.

RETURN VALUE

See above

RELATED FUNCTIONS

frand, srand

read

low level file read

SYNOPSIS

```
#include <stdio.h>

int read(int handle, char *buf, size_t n)
```

DESCRIPTION

The **read** function reads **n** bytes, from the file opened with **handle**, to the buffer pointed to by **buf**.

The file must have been previously opened by a call to "open" or "creat", and be readable.

As this function is not supported in the ANSI standard library, its use in new work is discouraged (see the "fopen", "fclose", "fread", and "fwrite" functions for the recommended replacements).

RETURN VALUE

The **read** function returns the number of bytes actually read. A return value of zero indicates EOF, (-1) indicates an error ("errno" will be set appropriately).

If the file being read has been opened in a "text" mode, and the host OS performs CR/LF mappings, the return value may be less than what was requested WITHOUT indicating that no more bytes are available in the file (beyond those fetched by the current **read** call).

Note the difficulty that reading a 65535 byte file on a computer with a 16 bit "int" size causes (the non-error and error return values are the same).

RELATED FUNCTIONS

close, creat, dup, dup2, open, write

realloc

change allocation of heap memory

SYNOPSIS

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size)
```

DESCRIPTION

The **realloc** function changes the size of an existing allocated region of heap memory. The contents of the region will be unchanged up to the lesser of the old and new **size** values. If the new **size** is larger, the value of the newly allocated portion of the region is undefined. If the value of **ptr** is NULL, the function behaves like the "malloc" function called with **size** as the parameter. If the requested space is unavailable, the region pointed to by **ptr** is not changed. If **size** is zero, and the value of **ptr** wasn't NULL, the **ptr** region is returned to the heap free storage pool.

If the value of **ptr** wasn't NULL, and the value wasn't the result of an earlier call to "calloc", "malloc" or "realloc", the results are undefined.

RETURN VALUE

If it is impossible to satisfy the request, or **size** is zero, a NULL pointer is returned. Otherwise, a pointer is returned to the start of the (possibly moved), allocated region.

RELATED FUNCTIONS

addfree, calloc, cfree, free, malloc

remove

delete file

SYNOPSIS

#include <stdio.h>

int remove(const char *pathname)

DESCRIPTION

The **remove** function deletes the file specified by **pathname**. If the file is currently open the results are undefined.

RETURN VALUE

If the deletion is successful, zero is returned; otherwise non-zero.

RELATED FUNCTIONS

rename, unlink

rename

change file name

SYNOPSIS

```
#include <stdio.h>

int rename(const char *oldname, const char *newname)
```

DESCRIPTION

The **rename** function changes the name of the file pointed to by **oldname**, into the name pointed to by **newname**. If a file already exists using the **newname** string as a name, the results are undefined.

RETURN VALUE

If the renaming is successful, zero is returned; otherwise non-zero. If not successful, the original name is still in force.

RELATED FUNCTIONS

None

restorefp

restore floating point pseudo registers

SYNOPSIS

```
#include <conc.h>

void restorefp(FPstate *fp)
```

DESCRIPTION

The **restorefp** function is only required for programs which use floating point, and are running on Transputers which lack hardware floating point support. It allows restoring the floating point emulation registers which were saved as a result of a previous call to "savefp". As mentioned in the "savefp" description, use of these functions is only required when floating point operations are being done by processes at both low and high priority.

See the "savefp" function description for more information about how these functions should be used.

RETURN VALUE

None

RELATED FUNCTIONS

savefp

rewind

rewind stream

SYNOPSIS

#include <stdio.h>

void rewind(FILE *stream)

DESCRIPTION

The **rewind** function rewinds the current read/write position for **stream** to the very beginning. It also clears the "error" and "eof" flags for the **stream**.

RETURN VALUE

None

RELATED FUNCTIONS

fseek, ftell, lseek

rindex

reverse search string for character

SYNOPSIS

#include <string.h>

char *rindex(const char *ptr, int c)

DESCRIPTION

The **rindex** function searches a string, pointed to by **ptr**, for the last occurrence of the character **c**. A pointer to where the character was found is returned, or NULL, if a '\0' string terminator was found instead. The '\0' character is considered part of the string and may be searched for. This function is called "strrchr" in ANSI libraries, and use of that name is recommended for all new work.

RETURN VALUE

See above

RELATED FUNCTIONS

index, strchr, strrchr

savefp

save floating point pseudo registers

SYNOPSIS

```
#include <conc.h>

void savefp(FPstate *fp)
```

DESCRIPTION

The **savefp** function is only required for programs which use floating point, and are running on Transputers which lack hardware floating point support. It allows saving the floating point emulation registers for later restoration with a call to "restorefp". Use of these functions is only required when floating point operations are being done by processes at both low and high priority.

On floating point Transputers, a context switch from low to high priority results in the automatic saving of the low priority floating point registers by the processor. For Transputers which lack floating point support, this action is simulated by having the high priority process call "savefp" to save the previous "pseudo floating point register" values prior to performing any floating point operations. When the process is finished with all floating point use, a matching call to "restorefp" may be used to restore the previous values.

Using this technique, the floating point emulation routines may be shared by both low and high priority processes without conflict. Note that only high priority processes which use floating point will require this protection.

The parameter to **savefp** should be a structure of type FPstate (**fp**), whose storage duration lasts between the time **savefp** is called, and the (later), matching call to "restorefp".

RETURN VALUE

None

RELATED FUNCTIONS

restorefp

scanf

formatted read

SYNOPSIS

```
#include <stdio.h>

int scanf(const char *format, ...)
```

DESCRIPTION

The **scanf** function reads from the "stdin" stream. The read is formatted under the control of the **format** argument, which specifies the legal input text sequences and conversion instructions. Subsequent arguments (if any), are used as pointers to the objects which receive converted data from the input. If there are too few arguments to match the requirements of the **format** string, the results are undefined. If arguments remain after meeting the requirements of the **format** string, they are evaluated and ignored. **scanf** returns when the input text fails to match the requirements of the **format** string, or when the **format** string is exhausted.

The **format** string consists of a '\0' terminated character sequence composed of:

1. Whitespace characters. These match zero or more whitespace characters read from the input (the whitespace read is ignored).

2. Ordinary characters (non-'%'). These must match exactly with characters read from the input (which are ignored after being read).

3. Format specifications (starting with '%'). These are generally used to describe what input text is a legal match for this part of the **format** string, how to read and convert the text, and how to store the result.

Format Specification

After the '%' character, the following fields make up a format specification:

1. An optional assignment suppression character ('*'). This causes any text read while matching this format specification to be discarded. No argument is need in the call to **scanf** to store the results of this conversion.

2. An optional decimal integer which specifies the maximum field width.

3. An optional 'h', 'l', or 'L' character, which defines the size of the receiving object. The 'd', 'i', and 'n' conversion specifiers are preceded with 'h' to indicate the corresponding argument is a pointer to "short int", or by 'l', to indicate a pointer to "long int". The 'o', 'u', and 'x' conversion specifiers are treated in the same fashion, except the parameters are taken to be "unsigned short int", and "unsigned long int", respectively. The 'l' character may be used with the 'e', 'f', and 'g' conversion specifiers to indicate a pointer to "double" ("float" is the default), or, alternatively, 'L' may be used to indicate a pointer to "long double". If the 'h', 'l', or 'L' characters are used with any other conversion specifier, the results are undefined.

4. Finally, the character which specifies the desired input sequnce to match and convert.

Note that input whitespace is ignored for the purposes of reading and matching conversions (except with the '[', 'c' and 'n' conversion specifiers).

Conversions

The following are the legal conversion characters:

'c' - Matches a sequence of input characters up to the specified field width (default of one). The corresponding argument will be a pointer to a character array large enough to store the matched sequence.

'd' - Matches an optionally signed decimal integer. The format matched is equivalent to that accepted by the "strtol" function with a "base" argument of ten. The corresponding argument will be a pointer to "int".

'e'
'E'
'f'
'g'
'G' - Matches an optionally signed floating point number. The format matched is equivalent to that accepted by the "strtod" function. The corresponding argument will be a pointer to "float".

'i' - Matches an optionally signed integer, whose format is either decimal, hexadecimal, or octal. The format matched is equivalent to that accepted by the "strtol" function with a "base" argument of zero. The corresponding argument will be a pointer to "int".

'n' - No input is matched or read. The corresponding argument will be a pointer to "int", into which a current count of the input characters read by this call to **scanf** will be written. Evaluation of this conversion specifier doesn't affect the return value from **scanf** (the # of successful assignments).

'o' - Matches an optionally signed octal integer. The format matched is equivalent to that accepted by the "strtoul" function with a "base" argument of eight. The corresponding argument will be a pointer to "unsigned int".

'p' - Matches an implementation defined sequence which is produced by the 'p' conversion specifier in the "printf" function. This "address" value takes the form of a normal hexadecimal integer in the **Transputer Toolset**.

's' - Matches a sequence of non-whitespace characters. The coresponding argument will be a pointer to a character array large enough to store the string (and the terminating '\0' which is automatically added).

'u' - Matches an optionally signed decimal integer. The format matched is equivalent to that accepted by the "strtoul" function with a "base" argument of ten. The corresponding argument will be a pointer to "unsigned int".

'x'
'X' - Matches an optionally signed hexadecimal integer. The format matched is equivalent to that accepted by the "strtoul" function with a "base" argument of sixteen. The corresponding argument will be a pointer to "unsigned int".

'[' - Matches a non-empty set of characters from a set of "expected" characters. The corresponding argument will be a pointer to a character array large enough to store the string (and the terminating '\0' which is automatically added). The conversion specifier includes all subsequent characters in the **format** string until a matching ']' is encountered. The characters between the brackets constitute the "expected" set, unless a '^' character immediately follows the '[' (in which case all characters EXCEPT THOSE listed, constitute the "expected" set). As a special case, the ']' is allowed to be in the "expected" set if it immediately follows the '[' (or to not be in the "expected" set, if it immediately follows a "[^"). When the special case is in effect, the concluding ']' will be the second one encountered in the **format** string.

'%' - Matches a literal '%' in the input. This allows "%%" to behave like an ordinary character in the **format** string.

Conversion specifications other than these produce undefined results.

Caveat

There is a maximum field width (per-conversion), of 509 characters, for this implementation.

RETURN VALUE

The **scanf** function returns EOF if an input failure occurs before any conversions. Otherwise, **scanf** returns the number of input sequences which were matched, converted, and assigned. The number of sequences returned, may be less than the number provided for in the **format** string, if a match fails, or EOF is encountered.

If EOF occurs in the middle of matching an input sequence, such that the match may already be considered successful (not counting optionally matched whitespace), the EOF is considered to merely terminate the current match. Any remaining conversions, however, will be immediately aborted if they require input.

RELATED FUNCTIONS

fscanf, sscanf, strtod, strtol, strtoul

EXAMPLES

The program fragment:

```
#include <stdio.h>
int n,i; float x; char name[49];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
24 54.31E-1 johnson
```

will assign the value 3 to 'n', 24 to 'i', 5.431 to 'x', and store the string "johnson\0" into array "name". Or:

```
#include <stdio.h>
int i; float x; char name[51];
scanf("%2d%f%*d %[0123456789]", &i, &x, name);
```

with the input line:

```
45678 0123 55:72
```

will assign the value 45 to 'i', 678.0 to 'x', will skip 0123, and store the string "55\0" into array "name". The next character to be read will be ':'.

To repeatably read a quantity, unit of measure, and the name of an item, the following code fragment:

```
#include <stdio.h>
int cnt; float quant; char units[25]; item [25];
while(!eof(stdin) && !ferror(stdin))
      {
      cnt = scanf("%f%24s of %24s",&quant, &units, &item);
      scanf("%*[^\n]"); /* Slurp rest of line */
      }
```

with input lines:

```
1 quart of milk
100 degrees Celsius
loads of fun
```

will produce a series of actions equivalent to:

```
1.    quant = 1.0; strcpy(units,"quart"); strcpy(item,"milk"); cnt = 3;
2.    quant = 100.0; strcpy(units,"degrees"); cnt = 2; /* No "of" */
3.    cnt = 0; /* No quantity */
4.    cnt = EOF; /* No input data */
```

SemAlloc/SemFree

semaphore allocation

SYNOPSIS

```
#include <conc.h>

Semaphore *SemAlloc(void)

void SemFree(Semaphore *s)
```

DESCRIPTION

The **SemAlloc** function returns a pointer to an initialized semaphore. The memory for the semaphore is obtained from the heap.

The **SemFree** function takes a semaphore, **s**, previously allocated using **SemAlloc**, and returns it to the heap free storage pool.

Note that semaphores may also be statically allocated (either globally, or as "auto" variables), if the dynamic creation and destruction allowed by **SemAlloc** and **SemFree** isn't required. To "statically" allocate/initialize a semaphore, the following might be used:

```
Semaphore xyz = SEMAPHOREINIT;
```

For additional information, see the included paper by Jeff Mock: **Processes, Channels and Semaphores**.

RETURN VALUE

See above

RELATED FUNCTIONS

HSemP, HSemV, SemP, SemV

SemP/_SemP/SemV/_SemV

mono-priority semaphores

SYNOPSIS

```
#include <conc.h>

int SemP(Semaphore s)

int _SemP(Semaphore *s)

int SemV(Semaphore s)

int _SemV(Semaphore *s)
```

DESCRIPTION

The **SemP** and **SemV** macros implement semaphore operations on the Transputer. These macros must not be used in situations where the same semaphore is shared among processes at different priority levels (use "HSemP" and "HSemV" in those cases). The **SemP** macro implements the "P" operation on a semaphore; it blocks the current process until the semaphore, **s**, is free. The **SemV** macro implements the complementary "V" operation, freeing the semaphore and starting the first process (if any), waiting in the semaphore queue. Note that the parameter to these functions is the semaphore proper, not a pointer to it. The **_SemP** and **_SemV** functions are primitives used by the macros (they should not be called directly).

The semaphore implementation is such that the routines may be used either as "binary" or "counting" semaphores.

Semaphores to be used by these routines must be initialized before use. This is automatically handled by the "SemAlloc" function, and should be performed manually if the semaphore is "statically" declared. See the "SemAlloc" function description for more information.

WARNING: THE SemP and SemV FUNCTIONS ARE IMPLEMENTED AS MACROS! THEY SHOULD NOT BE CALLED WITH A "s" PARAMETER WHOSE EVALUATION INVOLVES "SIDE EFFECTS", AS THE PARAMETER WILL BE EVALUATED MORE THAN ONCE!

For additional information, see the included paper by Jeff Mock: **Processes, Channels and Semaphores**.

RETURN VALUE

These functions return the "use" field from the **s** "Semaphore" structure.

RELATED FUNCTIONS

HSemP, HSemV, SemAlloc, SemFree

setjmp

initialize non-local jump

SYNOPSIS

```
#include <setjmp.h>

int setjmp(jmp_buf env)
```

DESCRIPTION

A call to the **setjmp** function saves a copy of the calling environment in the "jmp_buf" argument, **env**, for later use by a "longjmp" call. See the description of the "longjmp" function for more information.

RETURN VALUE

If the return is from the initial, direct, invocation of **setjmp**, zero is returned. If the return is a result of a subsequent call to the "longjmp" function, the return value is non-zero.

RELATED FUNCTIONS

longjmp

SetHiPriQ/SetLoPriQ

set queue pointers

SYNOPSIS

```
#include <conc.h>

void SetHiPriQ(void *front, void *back)

void SetLoPriQ(void *front, void *back)
```

DESCRIPTION

These functions provide "C" bindings for the Transputer "sthf/sthb" (**SetHiPriQ**), and "stlf/stlb" (**SetLoPriQ**), instructions. The **front** parameter is used to set the queue front pointer, the **back** parameter sets the queue back pointer. The values being set should be word aligned. See the appropriate INMOS documentation for more information about the "stXX" instructions.

These functions are implemented "inline". To get a functional version, precede your call with:

```
#undef SetHiPriQ
```

Or,

```
#undef SetLoPriQ
```

RETURN VALUE

None

RELATED FUNCTIONS

GetHiPriQ, GetLoPriQ

SetTime

set timer value for current priority

SYNOPSIS

#include <conc.h>

void SetTime(int time)

DESCRIPTION

The **SetTime** function sets the timer for the current priority level to **time**. Note that the timer runs at different rates for high (1 uS/tick), and low priority (64 uS/tick), processes.

This function is implemented "inline". To get a functional version, precede your call with:

#undef SetTime

For additional information see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

None

RELATED FUNCTIONS

ProcAfter, ProcWait, Time

sin/sinf

compute sine

SYNOPSIS

#include <math.h>

double sin(double x)

float sinf(float x)

DESCRIPTION

The **sin** function returns the sine of **x** (measured in radians). The **sinf** function does the same thing for a single precision argument.

RETURN VALUE

A large magnitude argument may yield a result with little significance.

RELATED FUNCTIONS

acos, asin, atan, cos, tan

sinh/sinhf

compute hyperbolic sine

SYNOPSIS

#include <math.h>

double sinh(double x)

float sinhf(float x)

DESCRIPTION

The **sinh** function returns the hyperbolic sine of **x**. The **sinhf** function does the same thing for a single precision argument.

RETURN VALUE

A range error ("errno" is set to ERANGE), occurs if the magnitude of **x** is too large.

RELATED FUNCTIONS

cosh, tanh

sprintf

formatted write to memory

SYNOPSIS

#include <stdio.h>

int sprintf(char *ptr, const char *format, ...)

DESCRIPTION

The **sprintf** function writes characters into the array pointed to by **ptr**. The write is formatted under the control of the **format** argument, which specifies how subsequent arguments (if any), are to be converted. See the "printf" function description for more information about formatting options.

RETURN VALUE

The number of characters written into the array (not counting the terminating '\0' character).

RELATED FUNCTIONS

fprintf, printf, vfprintf, vprintf, vsprintf

sqrt/sqrtf

compute square root

SYNOPSIS

```
#include <math.h>

double sqrt(double x)

float sqrtf(float x)
```

DESCRIPTION

The **sqrt** function returns the floating point square root of a non-negative floating point argument **x**. The **sqrtf** function does the same thing for a single precision argument.

On floating point Transputers this function is implemented "inline". To get a functional version, precede your call with:

```
#undef sqrt
```

Or,

```
#undef sqrtf
```

RETURN VALUE

A domain error ("errno" is set to EDOM), occurs if **x** is negative.

RELATED FUNCTIONS

None

srand

set random number seed

SYNOPSIS

#include <stdlib.h>

void srand(unsigned int)

DESCRIPTION

The **srand** function sets the pseudo-random number "seed" value, for subsequent calls to "frand" and "rand". Explicitly calling **srand**, with the same value, allows restarting the same sequence. The initial setting of the "seed" is equivalent to calling **srand** with a seed value of one.

RETURN VALUE

None

RELATED FUNCTIONS

frand, rand

sscanf

formatted read from memory

SYNOPSIS

```
#include <stdio.h>

int sscanf(const char *ptr, const char *format, ...)
```

DESCRIPTION

The **sscanf** function reads characters from the array pointed to by **ptr**. The read is formatted under the control of the **format** argument, which specifies the legal input text sequences and conversion instructions. Subsequent arguments (if any), are used as pointers to the objects which receive converted data from the input. See the "scanf" function description for more information about formatting options.

RETURN VALUE

The **sscanf** function returns EOF if an input failure (running into the '\0' string terminator), occurs before any conversions. Otherwise, **sscanf** returns the number of input sequences which were matched, converted, and assigned. The number of sequences returned, may be less than the number provided for in the **format** string, if a match fails, or '\0' is encountered.

If running into '\0' occurs in the middle of matching an input sequence, such that the match may already be considered successful (not counting optionally matched whitespace), the '\0' is considered to merely terminate the current match. Any remaining conversions, however, will be immediately aborted if they require input.

RELATED FUNCTIONS

fscanf, scanf, strtod, strtol, strtoul

ssort

shell sort

SYNOPSIS

#include <stdlib.h>

void ssort(void *base, size_t nmemb, size_t size, int (*compare)(const void *, const void *))

DESCRIPTION

The **ssort** function is an implementation of the "shell sort" algorithm which has the same calling sequence as the standard "qsort" function. This function may be used to advantage for medium sized and large data sets which provoke worst case "qsort" performance (either almost already sorted or anti-sorted input data, depending on implementation). Note that for more arbitrary input data, "qsort" will almost always be faster.

See the "qsort" function description for detailed parameter explanations.

RETURN VALUE

None

RELATED FUNCTIONS

isort, qsort

strcat

string concatenation

SYNOPSIS

```
#include <string.h>

char *strcat(char *dst, const char *src)
```

DESCRIPTION

The **strcat** function appends the string pointed to by **src**, to the end of the string pointed to by **dst**. A copy of the first character of the **src** string overwrites the '\0' string terminator at the original end of the **dst** string; a copy of the second character in the **src** string is written in the next location beyond the end of **dst**, etc. The '\0' terminator in the **src** string is included in the copy to form a new **dst** string termination. If the resulting **dst** string overlaps the **src** string, the results are undefined.

RETURN VALUE

The **dst** address is returned.

RELATED FUNCTIONS

strncat

strchr

search string for character

SYNOPSIS

#include <string.h>

char *strchr(const char *ptr, int c)

DESCRIPTION

The **strchr** function searches a string, pointed to by **ptr**, for the first occurrence of the character **c**. A pointer to where the character was found is returned, or NULL, if a '\0' string terminator was found instead. The '\0' character is considered part of the string and may be searched for.

RETURN VALUE

See above

RELATED FUNCTIONS

strrchr

strcmp

string compare

SYNOPSIS

```
#include <string.h>

int strcmp(const char *s1, const char *s2)
```

DESCRIPTION

The **strcmp** function compares the string pointed to by **s1** and the string pointed to by **s2**.

RETURN VALUE

The **strcmp** function returns a value greater than zero if string **s1** is lexicographically greater than string **s2**. If string **s1** is equal to **s2**, a value of zero is returned. If string **s1** is lexicographically less than string **s2**, a negative value is returned.

RELATED FUNCTIONS

memcmp, strncmp

strcpy

string copy

SYNOPSIS

#include <string.h>

char *strcpy(char *dst, const char *src)

DESCRIPTION

The **strcpy** function copies the string pointed to by **src**, to the address pointed to by **dst**. The '\0' string termination is part of the string copied. If the resulting **dst** string overlaps the **src** string, the results are undefined.

This function is implemented as a macro and may be "inlined" depending on the value of **src**. To get a functional version, precede your call with:

#undef strcpy

RETURN VALUE

The **dst** address is returned.

RELATED FUNCTIONS

memcpy, memmove, strncpy

strcspn

character class complement substring search

SYNOPSIS

#include <string.h>

size_t strcspn(const char *s1, const char *s2)

DESCRIPTION

The **strcspn** function returns the length of the maximum initial segment of the string pointed to by **s1** which contains no characters from the string pointed to by **s2**.

RETURN VALUE

See above

RELATED FUNCTIONS

strpbrk, strspn

strerror

convert error into string

SYNOPSIS

```
#include <string.h>

char *strerror(int errnum)
```

DESCRIPTION

The **strerror** function takes an argument, **errnum**, which represents a value to which the "errno" variable could be set, and returns a string describing the error. The string buffer used is static, and may be overwritten by a subsequent call to **strerror**.

There is a basic problem when using this function (or "perror"), to translate the meaning of a value taken from "errno":

1. The value of "errno" may be the result of a host server I/O error; whereupon the host environment definitions for "errno" should be used.

2. The value of "errno" may be the result of an error detected by the runtime library on the Transputer; thus the **Transputer Toolset** "errno" definitions should be used.

The scheme actually adopted was to only report the "errno" values (and naming scheme), which are used by the library functions running on the Transputer. For errors which happen on the host system, the results of the **strerror** function aren't defined.

A further problem with "errno" arises because it is a shared global variable. In a program consisting of many processes, each of which uses the same "errno", how do you determine which process detected the error?

RETURN VALUE

A pointer to the error message.

RELATED FUNCTIONS

perror

strlen

string length

SYNOPSIS

#include <string.h>

size_t strlen(const char *ptr)

DESCRIPTION

The **strlen** function returns the length in bytes (not counting the '\0' terminator), of the string pointed to by **ptr**.

This function is implemented as a macro and may be "inlined" depending on the value of **ptr**. To get a functional version, precede your call with:

#undef strlen

RETURN VALUE

See above

RELATED FUNCTIONS

None

strncat

string concatenation with length

SYNOPSIS

#include <string.h>

char *strncat(char *dst, const char *src, size_t n)

DESCRIPTION

The **strncat** function appends up to **n** characters of the string pointed to by **src**, to the end of the string pointed to by **dst**. A copy of the first character of the **src** string overwrites the '\0' string terminator at the original end of the **dst** string; a copy of the second character in the **src** string is written in the next location beyond the end of **dst**, etc. A '\0' terminator is always written to the end of the new **dst** string. If the resulting **dst** string overlaps the **src** string, the results are undefined.

RETURN VALUE

The **dst** address is returned.

RELATED FUNCTIONS

strcat

strncmp

string compare with length

SYNOPSIS

```
#include <string.h>

int strncmp(const char *s1, const char *s2, size_t n)
```

DESCRIPTION

The **strncmp** function compares up to **n** characters from the string pointed to by **s1** to the string pointed to by **s2**. The '\0' string terminator also stops the comparsion if the strings are shorter than **n** bytes in length.

RETURN VALUE

The return value of **strncmp** is based on at most the first **n** characters in the strings being compared. The **strncmp** function returns a value greater than zero if string **s1** is lexicographically greater than string **s2**. If string **s1** is equal to **s2**, a value of zero is returned. If string **s1** is lexicographically less than string **s2**, a negative value is returned.

RELATED FUNCTIONS

memcmp, strcmp

strncpy

string copy with length

SYNOPSIS

```
#include <string.h>

char *strncpy(char *dst, const char *src, size_t n)
```

DESCRIPTION

The **strncpy** function copies up to **n** characters from the string pointed to by **src**, to the address pointed to by **dst**. The '\0' string termination is part of the string copied and is counted against the maximum length **n**. If the resulting **dst** string overlaps the **src** string, the results are undefined. If the **src** string was shorter than **n** bytes in length (counting terminator), '\0' padding bytes are appended to the **dst** string until **n** bytes total have been written.

RETURN VALUE

The **dst** address is returned.

RELATED FUNCTIONS

memcpy, memmove, strcpy

strpbrk

character class string search

SYNOPSIS

#include <string.h>

char *strpbrk(const char *s1, const char *s2)

DESCRIPTION

The **strpbrk** function finds the first occurrence, in the string pointed to by **s1**, of any character in the string pointed to by **s2**.

RETURN VALUE

A pointer to the character found, or NULL if no character from **s2** is found in **s1**.

RELATED FUNCTIONS

memchr, strchr

strrchr

reverse search string for character

SYNOPSIS

#include <string.h>

char *strrchr(const char *ptr, int c)

DESCRIPTION

The **strrchr** function searches a string, pointed to by **ptr**, for the last occurrence of the character **c**. A pointer to where the character was found is returned, or NULL, if a '\0' string terminator was found instead. The '\0' character is considered part of the string and may be searched for.

RETURN VALUE

See above

RELATED FUNCTIONS

strchr

strspn

character class substring search

SYNOPSIS

#include <string.h>

size_t strspn(const char *s1, const char *s2)

DESCRIPTION

The **strspn** function returns the length of the maximum initial segment of the string pointed to by **s1** which contains only characters from the string pointed to by **s2**.

RETURN VALUE

See above

RELATED FUNCTIONS

strcspn, strpbrk

strstr

substring search

SYNOPSIS

#include <string.h>

char *strstr(const char *s1, const char *s2)

DESCRIPTION

The **strstr** function returns a pointer to the first occurrence of the string pointed to by **s2**, in the string pointed to by **s1**. The '\0' string terminator is not counted as part of the search.

RETURN VALUE

On success, a pointer to the matching string found in the string pointed to by **s1**. On failure, a NULL pointer is returned. If **s2** is of zero length, s1 is returned.

RELATED FUNCTIONS

strcspn, strpbrk, strspn

strtod/strtol/strtoul

convert strings to numbers

SYNOPSIS

```
#include <stdlib.h>

double strtod(const char *nptr, char **endptr)

long strtol(const char *nptr, char **endptr, int base)

unsigned long strtoul(const char *nptr, char **endptr, int base)
```

DESCRIPTION

The **strtod** function converts an ASCII numeric string pointed to by **nptr** into the equivalent floating point number. The **strtol** and **strtoul** functions do the same thing for long integers, and unsigned long integers, respectively. In all cases, whitespace is allowed before the numeric string and the first unrecognized character (in the selected **base**), stops the conversion. Also, if **endptr** is non-NULL, a pointer to the unrecognized character is stored into the object pointed to by **endptr**.

strtod recognizes a numeric string consisting of an optional sign, a digit sequence with optional decimal point, and an optional exponent ('e' or 'E', optional sign and exponent digit sequence).

strtol and **strtoul** recognize integral numbers consisting of an optional sign and digit sequence in a specified **base**. If the value of **base** is zero, the expected digit sequence is that of a "C" integer constant (ie. decimal, hexadecimal, or octal notation, without any integer suffix). If the value of **base** is between 2 and 36, the expected digit sequence is a sequence of numbers and letters representing an integer with radix specified by **base**. The letters from 'a' ('A'), through 'z' ('Z'), are used for the values 10 to 35 when **base** is greater that 10; only digits and letters whose value is less than the **base** are allowed. If the value of **base** is 16, the "0x" and 0X" hexadecimal prefixes are permitted.

RETURN VALUE

The converted value. If no conversion could be performed (or the correct value would cause underflow), zero is returned. If the correct value would cause overflow **strtod** returns +/- HUGE_VAL, **strtol** returns LONG_MAX/LONG_MIN, and **strtoul** returns ULONG_MAX. If underflow or overflow is detected, "errno" is set to ERANGE.

RELATED FUNCTIONS

atof, atoi, atol

strtok

break string into tokens

SYNOPSIS

#include <string.h>

char *strtok(char *s1, const char *s2)

DESCRIPTION

The **strtok** function returns pointers into the string pointed to by **s1**, which are delimited by characters in the string pointed to by **s2**. The **strtok** function is designed to be used repetitively to decompose a string into individual '\0' terminated substrings. The **strtok** function remembers, between calls, where it was in processing the string pointed to by **s1**. This, the **s1** argument should be replaced with a NULL pointer for the second and subsequent calls to **strtok** for the same (**s1**), string. The delimiter string, **s2**, may vary from one call to the next.

The first call to **strtok** returns a NULL pointer if the first character in the string pointed to by **s1**, is also in the delimiter string, **s2**. If not, **strtok** searches for the first character in **s1** which is in **s2**, replaces it with a '\0' terminator, and returns the pointer **s1**.

Each subsequent call behaves in a similar fashion, except, the search begins at the first character after the '\0' terminator written by the previous call to **strtok**.

RETURN VALUE

A pointer to the start of the next token, or NULL if no token matching the delimiting characters could be found.

RELATED FUNCTIONS

None

system

call host command processor

SYNOPSIS

```
#include <stdlib.h>

int system(const char *ptr)
```

DESCRIPTION

The **system** function passes the string pointed to by **ptr**, to the host environment command processor for execution. If **ptr** is NULL, the **system** function may be used to determine if a command processor exists.

RETURN VALUE

If **ptr** was NULL, a non-zero return value indicates a command processor exists. If **ptr** was non-NULL (a string), the return value is assumed to be a value returned from the command processor after execution of the supplied string.

RELATED FUNCTIONS

exit

tan/tanf

compute tangent

SYNOPSIS

#include <math.h>

double tan(double x)

float tanf(float x)

DESCRIPTION

The **tan** function returns the tangent of **x** (measured in radians). The **tanf** function does the same thing for a single precision argument.

RETURN VALUE

A large magnitude argument may yield a result with little significance.

RELATED FUNCTIONS

acos, asin, atan, cos, sin

tanh/tanhf

compute hyperbolic tangent

SYNOPSIS

#include <math.h>

double tanh(double x)

float tanhf(float x)

DESCRIPTION

The **tanh** function returns the hyperbolic tangent of **x**. The **tanhf** function does the same thing for a single precision argument.

RETURN VALUE

See above

RELATED FUNCTIONS

cosh, sinh

time

determine calendar time

SYNOPSIS

#include <time.h>

time_t time(time_t *timer)

DESCRIPTION

The **time** function determines the current calendar time.

RETURN VALUE

The **time** function returns the best approximation to the current time the host system can compute. If **timer** is not a NULL pointer, the return value is also stored in the object it points to.

RELATED FUNCTIONS

asctime, ctime, gmtime, localtime

Time

get timer value for current priority

SYNOPSIS

```
#include <conc.h>

int Time(void)
```

DESCRIPTION

The **Time** function atomically returns the value of the timer for the current priority level. Note that the timer runs at different rates for high (1 uS/tick), and low priority (64 uS/tick), processes.

This function is implemented "inline". To get a functional version, precede your call with:

```
#undef Time
```

For additional information see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

See above

RELATED FUNCTIONS

ProcAfter, ProcWait, SetTime

tmpfile

create temporary file

SYNOPSIS

```
#include <stdio.h>

FILE *tmpfile(void)
```

DESCRIPTION

The **tmpfile** function creates a temporary binary file which is automatically deleted when it is closed (or the program terminates). The file is opened in the "wb+" mode (see the "fopen" function description for more information).

RETURN VALUE

A pointer to the opened stream, or NULL if the file couldn't be created.

RELATED FUNCTIONS

fopen, tmpnam

tmpnam

create unique filename

SYNOPSIS

#include <stdio.h>

char *tmpnam(char *ptr)

DESCRIPTION

The **tmpnam** function creates a filename which is designed to not conflict with existing files. The **tmpnam** function generates a different name each time it is called, up to at least TMP_MAX times.

If the argument, **ptr**, is non-NULL, the generated name is stored starting at the location it points to. The return value will be set to **ptr**.

If **ptr** is NULL, the return value will point to an internal buffer area, which may be overwritten on subsequent **tmpnam** calls.

There is an inherent race condition in using this function involving the time between the **tmpnam** call and the subsequent file creation call. We suggest you use the "tmpfile" function for all new work.

RETURN VALUE

See above

RELATED FUNCTIONS

tmpfile

toascii/tolower/toupper

map characters

SYNOPSIS

#include <ctype.h>

int toascii(int c)

int tolower(int c)

int toupper(int c)

DESCRIPTION

These macros/functions map various classes of characters into other classes. The argument, **c**, is the character to be mapped. These macros/functions are designed to evaluate the argument only once, and are safe for use with arguments with side-effects. The individual descriptions:

toascii - Map **c** into the ASCII character set (**c** & 0x7F).

tolower - If **c** is a upper case character, convert it to the lower case equivalent. Leaves other characters unchanged.

toupper - If **c** is a lower case character, convert to to the upper case equivalent. Leaves other characters unchanged.

RETURN VALUE

The (possibly), converted character.

RELATED FUNCTIONS

isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit

ungetc

push back a character

SYNOPSIS

#include <stdio.h>

int ungetc(int c, FILE *stream)

DESCRIPTION

The **ungetc** function pushes the character specified by c, back onto the input stream. Only one ungetc operation is guaranteed to work without an intervening read operation (although some host systems will support more). Characters pushed back onto a stream are read in the reverse order they were pushed.

Use of one of the file positioning functions ("fseek", "rewind", ...), will cause any pushed back characters to be discarded. A successful call to "ungetc" will clear the EOF flag for the stream. EOF may not be pushed back.

RETURN VALUE

The **ungetc** function returns the character pushed back (or EOF on error).

RELATED FUNCTIONS

fgetc, fputc, getc, getchar, putc, putchar

unlink

delete file

SYNOPSIS

```
#include <stdio.h>

int unlink(const char *pathname)
```

DESCRIPTION

The **unlink** function deletes the file specified by **pathname**. If the file is currently open the results are undefined.

The **unlink** function is not part of the ANSI standard library. We recommend use of the equivalent "remove" function for new work.

RETURN VALUE

If the deletion is successful, zero is returned; otherwise non-zero.

RELATED FUNCTIONS

remove, rename

va_arg

variable argument access macro

SYNOPSIS

#include <stdarg.h>

type va_arg(va_list ap, type_expression)

DESCRIPTION

The **va_arg** macro expands to an expression which has the **type** and value of the next argument in the argument list managed by the "va_list" structure, **ap**. Each invocation of **va_arg** modifies **ap** so that successive arguments are returned. The "va_list" structure, **ap**, must have been initialized by a call to "va_start" prior to any use of **va_arg**. The parameter, **type_expression**, is a type name specified such that appending a '*' to the parameter will yield a legal "C" type expression which is a pointer to the specified **type**.

See the description of the "va_end" macro for further information (and an example of use).

RETURN VALUE

The value of the next argument in the variable argument list (assumed to be of **type** type).

RELATED FUNCTIONS

va_end, va_start

va_end

variable argument termination macro

SYNOPSIS

```
#include <stdarg.h>

void va_end(va_list ap)
```

DESCRIPTION

The **va_end** macro terminates processing of the arguments in a variable argument list, **ap**. This macro should be called after the argument list is initialized by a call to "va_start", and has been accessed through a series of "va_arg" calls. The example shows the usage of the three macros.

RETURN VALUE

None

RELATED FUNCTIONS

va_arg, va_start

EXAMPLE

The function, "argtbl", builds an array of pointers to strings which represents the parameters passed to "argtbl". The argument array is then passed to function "argtbl2" for further processing:

```
     #include <stdarg.h>

     #define MAX_ARGS 30 /* Whatever the application
requires */

     void argtbl(int num_ptrs, ...)
          {
          va_list ap;
          char *arg_ptrs[MAX_ARGS];
          int ptr_num = 0;

          if(num_ptrs > MAX_ARGS)
               num_ptrs = MAX_ARGS;
          va_start(ap, num_ptrs);
          while(ptr_num < num_ptrs)
               arg_ptrs[ptr_num++] = va_arg(ap, char *);
          va_end(ap);
          argtbl2(num_ptrs, arg_ptrs);
          }
```

Note that in this example, each call to "argtbl" must have a first parameter which defines how many following parameters are in the argument list.

va_start

variable argument initialization macro

SYNOPSIS

```
#include <stdarg.h>

void va_start(va_list ap, right_param)
```

DESCRIPTION

The **va_start** macro initializes the "va_list" structure, **ap**, such that subsequent calls to the "va_arg" macro will evaluate to the arguments in the enclosing function parameter list. These parameters will be located immediately after the last listed formal parameter, **right_param** (right before the ", ...").

See the description of the "va_end" macro for further information (and an example of use).

RETURN VALUE

None

RELATED FUNCTIONS

va_arg, va_end

VChan

get virtual channel pointer

SYNOPSIS

```
#include <conc.h>

Channel *VChan(int lchan)
```

DESCRIPTION

The **VChan** function translates a logical channel number (**lchan**), into the corresponding virtual channel pointer. The virtual channel pointer which is returned may then be used with the various virtual channel I/O routines.

The user selected **lchan** value can range from 6 to 32767. It must be the same value used to refer to the same channel (on a given node), in the "Network Information File" to be used with LD-NET. LD-NET uses the information during loading to configure and route virtual channels. **VChan** is basically a mechanism to allow the application program to get a pointer to a virtual channel that LD-NET sets up during the loading process.

Whenever possible, smaller logical channel numbers should be used since it minimizes the memory overhead of the virtual channel control information. See the LD-NET documentation for additional information about logical channels and associated memory usage.

Channel pointers returned by **VChan** may only be used with the various virtual channel I/O routines. Using a virtual channel pointer with a non-virtual channel I/O routine will not work, and may "crash" the Transputer node.

RETURN VALUE

A pointer to the corresponding virtual channel. If **lnum** is outside the range of 6 to 32767 (or wasn't mentioned in the "Network Information File" used by LD-NET to load this node), NULL will be returned.

RELATED FUNCTIONS

VChanIn, VChanInChar, VChanInInt, VChanOut, VChanOutChar, VChanOutInt, VChanVin, VChanVOut, VProc*

VChanIn/VChanInChar/VChanInInt
reading from virtual channels

SYNOPSIS

```
#include <conc.h>

void VChanIn(Channel *c, void *ptr, int n)

char VChanInChar(Channel *c)

int VChanInInt(Channel *c)
```

DESCRIPTION

The **VChanIn** function reads **n** bytes of data, from the virtual channel pointed to by **c**, to the buffer pointed to by **ptr**. The **VChanInChar** and **VChanInInt** functions may be used to read, and return, the value of a byte or word, respectively, read from the virtual channel pointed to by **c**.

These functions are analogous to the "ChanIn", "ChanInChar" and "ChanInInt" functions (see descriptions elsewhere in this manual), but these functions support both virtual and regular ("soft"), channels. When used with regular channels the two sets of functions are equivalent.

When used with a virtual channel, the **VChanIn** function allows the receipt of a zero length message. This "synchronization-only" behavior is not allowed with regular channels on T2/T4/T8 processors. When receiving zero length messages, the specific destination buffer pointer value (**ptr**), is unimportant. A value of NULL is a good choice since it helps document what is going on.

All messages read using one of these functions must be matched by a corresponding virtual channel output function call whose message length EXACTLY matches. If variable length data transfer is needed, you should first send a message header specifying the size of the variable length data to come, followed by sending the actual data. This allows the input and output operations to always have matching message lengths. See the "VChanVIn" and "VChanVOut" functions for another approach to variable length virtual channel communication.

Since virtual channels are bi-directional, it is perfectly permissable to have processes concurrently performing input and output on the same "end" of a virtual channel. Since each "end" of a virtual channel is independent, you can potentially have four processes working on the same virtual channel (although perhaps not all on the same node since the "ends" may be on different nodes). As regular T2/T4/T8 channels are uni-directional you may only have one input and one output process working on the same channel, even if you use them with these virtual channel input functions.

For additional information about "regular channel" primitives, see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

See above

RELATED FUNCTIONS

VChan, VChanOut, VChanOutChar, VChanOutInt, VChanVIn, VChanVOut, VProc*

VChanOut/VChanOutChar/VChanOutInt

writing to virtual channels

SYNOPSIS

```
#include <conc.h>

void VChanOut(Channel *c, void *ptr, int n)

void VChanOutChar(Channel *c, int byte)

void VChanOutInt(Channel *c, int word)
```

DESCRIPTION

The **VChanOut** function writes **n** bytes of data, to the virtual channel pointed to by **c**, from the buffer pointed to by **ptr**. The **VChanOutChar** and **VChanOutInt** functions may be used to write the value of a **byte** or **word**, respectively, to the virtual channel pointed to by **c**.

When used with a virtual channel, the **VChanOut** function allows the transmission of a zero length message. This "synchronization-only" behavior is not allowed with regular channels on T2/T4/T8 processors. When sending zero length messages, the specific source buffer pointer value (**ptr**), is unimportant. A value of NULL is a good choice since it helps document what is going on.

See the discussion for the corresponding input operations ("VChanIn" ...), for a more lengthy introduction.

For additional information about "regular channel" primitives, see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

None

RELATED FUNCTIONS

VChan, VChanIn, VChanInChar, VChanInInt, VChanVIn, VChanVOut, VProc*

VChanVIn/VChanVOut

variable length virtual channel I/O

SYNOPSIS

```
#include <conc.h>

int VChanVIn(Channel *c, void *ptr, int n)

void VChanVOut(Channel *c, void *ptr, int n)
```

DESCRIPTION

The **VChanVIn** function reads at most **n** bytes of data, from the virtual channel pointed to by **c**, to the buffer pointed to by **ptr**. The return value is the number of bytes actually read, or -1 if the message is longer than **n** bytes. The **VChanVOut** function writes **n** bytes of data, to the virtual channel pointed to by **c**, from the buffer pointed to by **ptr**.

These functions are similar to "VChanIn"/"VChanOut", except for the variable length capability. These functions should only be used with virtual channels since regular T2/T4/T8 "soft" channels do not support this capability. For proper operation, messages to be received by the **VChanVin** function should always be sent by the **VChanVOut** function (and vice-versa).

Note that it is perfectly permissable for the output message length specified to **VChanVOut**, or the maximum input message length specified to **VChanVIn**, to be zero. This provides the capability of a "synchronization-only" message. When receiving or sending zero length messages, the specific buffer pointer value (**ptr**), is unimportant. A value of NULL is a good choice since it helps document what is going on.

See the "VChanIn" function description for more information about virtual channel usage.

RETURN VALUE

See above

RELATED FUNCTIONS

VChan, VChanIn, VChanInChar, VChanInInt, VChanOut, VChanOutChar, VChanOutInt, VProc*

vfprintf

vararg formatted write to stream

SYNOPSIS

```
#include <stdio.h>
#include <stdarg.h>

int vfprintf(FILE *stream, const char *format, va_list arg)
```

DESCRIPTION

The **vfprintf** function is identical to the "fprintf" function, except it uses a "packaged" argument structure, instead of a variable length argument list. The "packaged" argument structure is initialized by a "va_start" macro call, followed by an optional "va_arg" macro call for each argument to be added. The **vfprintf** function does not invoke the "va_end" macro. See the descriptions of the "va_start", "va_arg", and "va_end" macros, for more information.

RETURN VALUE

The number of characters written (negative values indicate an error).

RELATED FUNCTIONS

fprintf, printf, sprintf, va_arg, va_end, va_start, vprintf, vsprintf

vprintf

vararg formatted write

SYNOPSIS

```
#include <stdio.h>
#include <stdarg.h>

int vprintf(const char *format, va_list arg)
```

DESCRIPTION

The **vprintf** function is identical to the "printf" function, except it uses a "packaged" argument structure, instead of a variable length argument list. The "packaged" argument structure is initialized by a "va_start" macro call, followed by an optional "va_arg" macro call for each argument to be added. The **vprintf** function does not invoke the "va_end" macro. See the descriptions of the "va_start", "va_arg", and "va_end" macros, for more information.

RETURN VALUE

The number of characters written (negative values indicate an error).

RELATED FUNCTIONS

fprintf, printf, sprintf, va_arg, va_end, va_start, vfprintf, vsprintf

VProcAlt*

determine the status of virtual channels

VProcAlt
VProcAltList
VProcSkipAlt
VProcSkipAltList
VProcTimerAlt
VProcTimerAltList

SYNOPSIS

```
#include <conc.h>

int VProcAlt(Channel *c1, ...)

int VProcAltList(Channel **clist)

int VProcSkipAlt(Channel *c1, ...)

int VProcSkipAltList(Channel **clist)

int VProcTimerAlt(int time, Channel *c1, ...)

int VProcTimerAltList(int time, Channel **clist)
```

DESCRIPTION

These functions allow a process to determine the status of one or more input virtual channels. On T2/T4/T8 processors these functions may also be used with regular "soft" channels with the restriction that the associated input must be of non-zero length. These functions operate correctly with either zero or non-zero length input messages when used with virtual channels.

VProcAlt, **VProcSkipAlt**, and **VProcTimerAlt** take an explicit NULL terminated list of pointers to channels as parmeters. **VProcAltList**, **VProcSkipAltList**, and **VProcTimerAltList** take a NULL terminated array of pointers to channels as a parameter. Note that the channel pointers are prioritized based on the order they are listed in.

VProcAlt and **VProcAltList** cause the current process to block until one of the channels in its argument list is ready for input. On completion, the routine returns a zero based index into the parameter list for the ready channel.

VProcSkipAlt and **VProcSkipAltList** check specified channels. If one of the channels is ready for input, a zero based index into the parameter list is returned, otherwise (-1) is returned. These routines do not block waiting for one of the channels, they return immediately.

VProcTimerAlt and **VProcTimerAltList** block the current process until one of the channels is ready for input or the value of the clock (as read by the "Time" function), is after the **time** parameter. If the routine times out, a -1 is returned, otherwise a zero based index into the parameter list is returned.

These functions are equivalent to the "ProcAlt*" functions described elsewhere in this manual, except they can handle both regular and virtual channels. For additional information on the non-virtual channel versions (which is also applicable to these functions), see the included paper by Jeff Mock: **Processes, Channels, and Semaphores**.

RETURN VALUE

See above

RELATED FUNCTIONS

ProcAfter, ProcWait, Time, VChan, VChanIn, VChanInChar, VChanInInt, VChanOut, VChanOutChar, VChanOutInt, VChanVin, VChanVOut

vsprintf

vararg formatted write to memory

SYNOPSIS

```
#include <stdio.h>
#include <stdarg.h>

int vsprintf(char *ptr, const char *format, va_list arg)
```

DESCRIPTION

The **vsprintf** is identical to the "sprintf" function, except it uses a "packaged" argument structure, instead of a variable length argument list. The "packaged" argument structure is initialized by a "va_start" macro call, followed by an optional "va_arg" macro call for each argument to be added. The **vsprintf** function does not invoke the "va_end" macro. See the descriptions of the "va_start", "va_arg", and "va_end" macros, for more information.

RETURN VALUE

The number of characters written (not counting the '\0' terminating character).

RELATED FUNCTIONS

fprintf, printf, sprintf, va_arg, va_end, va_start, vfprintf, vprintf

write

low level file write

SYNOPSIS

#include <stdio.h>

int write(int handle, char *buf, size_t n)

DESCRIPTION

The **write** function writes **n** bytes of data, to the file opened with **handle**, from the buffer pointed to by **buf**.

The file must have been previously opened by a call to "open" or "creat", and be writable.

As this function is not supported in the ANSI standard library, its use in new work is discouraged (see the "fopen", "fclose", "fread", and "fwrite" functions for the recommended replacements).

RETURN VALUE

The **write** function returns the number of bytes actually written. A return value of (-1) indicates an error and "errno" will be set appropriately.

Note the difficulty that writing a 65535 byte file on a computer with a 16 bit "int" size causes (the non-error and error return values are the same).

RELATED FUNCTIONS

close, creat, dup, dup2, open, read