Transputer 'C' Library Description

Version 91.1
11/15/91

Copyright 1987-1991 by Logical Systems

Contents

1 Introduction
- Overview

2 Normal "C" Library Functions
- Character Classification and Mapping
- Numeric Conversion
- Math
  - Emulated Math on Transputers
- Heap Management
  - Heap Placement and Size
- String Manipulation
- Input/Output
  - Alternatives to "_main"
    - "_ns_main"
    - "_vcmain"
    - "_ns_vcmain"
- Miscellaneous

3 Transputer Specific Library Functions
- Introduction
- Transputer Channel Communications
- Transputer Virtual Channel Communications
- Transputer Channel Status Testing
- Transputer Virtual Channel Status Testing
- Transputer Concurrency (Jeffrey Mock Model)
- Transputer Concurrency (Fork/Join Model)
- Transputer Semaphore Support
- Transputer Timing and Scheduling
- Transputer Miscellaneous

4 Examples
- Introduction
- Sample Programs

5 Detailed Library Function Descriptions

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).