PARALLEL PROCESSING USING TRANSPUTERS
Spring 1997

Prepared By : Tarak Modi
Advisor : Dr. Wells

TRANSPUTERS : HARDWARE DESCRIPTION

A Brief Background

The transputer was introduced to the world in 1985 by INMOS limited. The transputer was a VLSI chip, and at that time was heralded as a revolution in the world of computing. The transputer is essentially a high performance microprocessor. One of its most significant features is its ability to perform multi-tasking in hardware, and its sub micro-second context switching. Context switch time denotes the time it takes for a higher priority task to preempt a lower priority task, and start executing. Communication between processes is also provided by hardware, both for internal data transfers, and transfers between different processors.

The first processor in the transputer series was T414 and boasted an unprecedented speed of 10 MIPS (millions of instructions per second), with the capability of multitasking in hardware. Dedicated on chip link controllers allowed for communications between processes running on different processors (transputers), with a minimal processor overhead. The T414 is a 32-bit processor with 2K on-chip RAM and four interprocessor links, capable of addressing upto 4Gbytes of external memory using multiplexed address and data lines.
Since then, other processors have been developed by INMOS

- **T212 :** A 16-bit version of the T414 with 2K of onboard RAM and a 64K address range. using separate address and data buses.
- **M212 :** A T212 with two of the four links replaced by a built-in disc controller circuit.
- **T800 :** Essentially, a revamped T414 with a floating point coprocessor integrated on the chip, more instructions, and on-chip RAM increased to 4K. This is one of the more popular transputers sice it combines high performance floating point processing (1.5 MFLOPS) with the possibilities of parallel processing.
- **T222 :** A T212 with 4K of on chip RAM.
- **T425 :** An updated T414 with debugging instructions added such as for single stepping.
- **T801 :** A T800 with separate address and data lines for faster memory access.
- **T805 :** An updated T800 with debugging instructions added such as for single stepping.

When the transputer was first introduced to the market, INMOS essentially forgot to mention the fact that the transputer was actually a microprocessor, different from other microprocessors at the time, but still a microprocessor. For a long time there was also a lack of information on the instruction set and assembly level features of the transputers. In fact INMOS claimed that the transputer was specifically designed to efficiently execute the high level language Occam. They suggested that Occam was the “assembly language” for the transputers. Occam is a high level language with a strong support for parallel processing. The transputer and Occam were jointly developed with each other in mind. As a result Occam programs on the transputer compile to very efficient and tight object code. However, this meant that engineers at that time could not use there beloved FORTRAN programming language. At the same time assembly level programmers could no longer work at the low level. Additionally, there were some things that could absolutely not be done in Occam. A state of anarchy followed, which finally led INMOS to publish “The Compiler Writer’s Guide”, which gave a detailed explanation of the instruction set for the transputer family.

Transputer multicomputers are appealing for two main reasons. The first one is flexibility, both in the implementation of user-configurable networks, and in a growing software base, which includes, among other languages, C, C++, Occam, Pascal, Modula-2 and Prolog. The second reason is affordability. One- and four-transputer boards are now available for most popular personal computers and workstations, making transputer-based multiprocessors available for as low as $450 per node, bringing the exploration of parallel programming within the reach of most universities and colleges, and even individuals.

Processes And Concurreny in the Transputer

The transputer uses processes as the basic software building block and provides the direct implementation of a process in hardware. A process is an independent computation (or a sequence of computations) that communicates with other processes. This communication on the transputer is achieved by explicitly defined channels. A process itself may consist of subprocesses by time-sharing.

The transputer provides a number of links which support point-to-point communications between transputers, hence allowing processes to be distributed over a network of transputers. Therefore, it is possible to build a system with a network of transputers, with each transputer in the system executing a set of processes and each transputer capable of communicating with others on the network. It should be noted however that each transputer can only communicate directly with another physically connected transputer. It is the ability to specify a hardwired fuction as a process which provides the architectural framework of transputers the capability to be used in highly specialized and diverse applications.

Transputer Harware Description

The transputer is a component designed to exploit the potential of VLSI technology. This technology allows for a large number of identical components to be manufactured cheaply. For this matter it is desirable to be able to implement a concurrent system using identical components. As mentioned earlier the transputer is a high performance microprocessor which has been designed to facilate interprocess and interprocessor communication. The transputer architecture defines a family of programmable VLSI components. The principle features of the transputer architecture are

A Programmable Device
Transputer systems can be designed and programmed using a variety of different languages including assembly level mnemonics, occam, concurrent Ada, and concurrent versions of C and C++.

A Multitasking processor with Memory on a single chip
The transputer processor supports multitasking, and enforces two levels of priority for concurrent tasks. Besides allowing the creation of parallel applications where communication and computation functions can be implemented cleanly in separate modules, the multitasking feature provides us with the opportunity to start experimenting with parallelism in a shared memory environment. Almost every operation that a transputer performs involves intefacing with memory. By providing on chip memory this interface time has been greatly reduced.

Links and channel communication
Transputers communicate with each other via high-speed serial links. The small number of communication wires required and the simplicity of the protocol makes the networking of transputers simple and in most-cases user configurable. This feature allows for the easy evaluation of various network configurations such as rings, meshes, hypercubes, or trees. At the software level, the hardware links are

defined as channels. Hence, while transputers communicate with each other via links, the tasks residing in them communicate through channels. Because the transputer implements communication primitives at its

lowest level (in microcode), and because the access to the serial links is memory-mapped, the transputer offers a powerful feature to the programmer; primitive supporting the communication between tasks residing on different neighboring transputers can also be used to allow tasks residing on the same transputer to communicate. In other words, tasks residing on the same transputer can communicate by sending a message to each other with the same primitive they would use to exchange messages with a task residing on a neighboring transputer. This powerful feature allows the programmer to study first hand some of the complex issues dealing with mapping and folding parallel applications to small networks

Virtual Channels

Virtual channels are a technique for circumventing the limited-fanout of the multicomputer nodes. While a node may be physically limited to four Input/Output links, as is the case for the transputer, a software or hardware routing mechanism can be introduced to allow the transfer of message between different transputers as if they were connected by a channel, even though they may not be directly connected to each other, and their four links may already be connected to other transputers. Hence a virtual path can be created between arbitrary pairs of nodes. This feature, invisible to the programmer, offer a powerful medium for implementing arbitrary networks that can be dynamically configured and modified.

Hence a transputer circuit contains a processor, a small amount of memory, a co-processor, and four high-speed bi-directional links. Although such simple a list may not be impressive, the elegant design of the transputer provides a rich environment for parallel processing.

Worth mentioning is the fact that combining these entities on a single silicon chip was not a technological breakthrough in itself. Other companies, such as Intel or Motorola had introduced versions of their popular 8-bit processors with local memory and interfacing hardware. The novelty was the combination of several factors. One of the most important factor was the introduction of a high-level language, occam whose features were directly supported by the transputer hardware. and that made the transputer a building block for parallel computers. The second prominent factor was the ease with which transputers could be connected to each other with as little as a few electrical wires.

The four bi-directional input/output (I/O) ports of the transputer are designed to interface directly with the ports of other transputers, very much like the pegs on top of Lego blocks fit directly in their bottom cavities. This feature allows for several transputers to fit on a small footprint, with very little extra logic circuits, making it possible to easily fit four transputers with some memory on a PC daughter board (ISA bus), or microchannel board.

IMS T800 Architecture

The IMS T800, with its on-chip floating point unit is only 20% larger in area than the T414. This small size and high performance design emerges from a careful note od silicon economics. The IMS T800 architecture is similar to that of the T414. However, in addition to the memory, links, central processing unit (CPU), and the external memory interface, it includes a floating point unit (FPU), which operates concurrently and under the control of the CPU. Figure 1 shows a block diagram for the T800.

The IMS T800, like the T414 operates at two priority levels. The FPU stack is duplicated so that when the T800 switches from low to high, none of the states in the FPU are written to memory. This reduces the worst-case response significantly. Furthermore, duplication of the resgister stack enables the floating-point arithmetic to be used in an interrupt mode

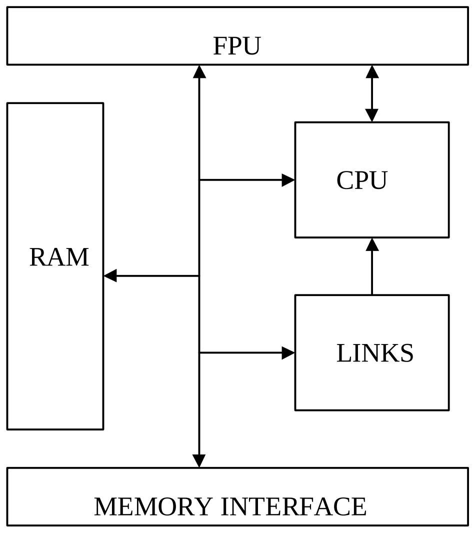


Figure 1

Registers

The T800 CPU just like the T414, contains three registers (A, B, C) used for integer and address arithmetic, which form a hardware stack. Loading a value into the stack pushes B into C, and A into B before loading A. Storing a value from A pops B into A, and C into B. In addition, there is an operand register that helps in the formation of instruction operands, an Instruction Pointer that points to the next instruction to be executed, and the workspace pointer that points to an area of memory where local variable are held. In addition to these six registers there are four registers theat handle the two active process queues, namely Fptr0, Fptr1, Bptr0, and Bptr1, and two timer registers Time0 and Time1. There are two single bit flags for dealing with erreors, Error and HaltOnError. Also, the first dew locations of memory are used for specific purposes.

Similarly the FPU contains a three register floating-point evaluation stack, consisting of AF, BF, and CF registers. Loading and storing values from the evaluation stack in the FPU functions similar to that of the CPU. The addresses of the floating-point values are formed on the CPU stack, and values are transferred between the addressed memory locations and the FPU stack under the control of the CPU. As the CPU stack is only used to hold addresses of the floating-point values, the word length of the CPU is independent of that of the FPU. Consequently, it would be possible to use the same FPU with a 16-bit transputer such as the T212, for example.

The workspace pointer (W) deserves more attention. It holds an address consisting of a word selector and a byte offset within that word. In the 32-bit transputers two byte offset bits (the least significant two) are used allowing byte offsets of 0, 1, 2 and 3. The byte offset in W is always 0 so that it always points to a word boundary in memory. The least significant bit instead is used to store the process priority, which is 0 for a high priority and 1 for a low priority. This combination of the workspace address and the priority bit is referred to as the process descriptor. A few words of memory just below the workspace pointer are used by various parts of the scheduling hardware as follows (relative to address pointed to by W) :

-1	holds the IP of a descheduled process
-2	maintain a list of active but descheduled processes.
-3	Used during channel communication to hold the address of the data to be transferred.
-4	flag used during timer ALTs to indicate a valid time to wait for.
-5	used during time ALTS to hold a time to wait for.

Communication Links And Protocol
One of the aims of the transputer architecture is to provide a family of compatible components that can comunicate with each other using minimal amout of external logic, irrespective of the individual internal clock rates. To achieve this, each transputer is equipped with INMOS links, and can communicate using an asynchronous bit-serial protocol. Each transputer has a fixed number of such links, typically four. Each link is bi-directional, and can be used to develop a variety of topologies. Since the communication is asynchronous the relative skew must be kept within tolerance, and is an important consideration while implementing different network topologies.

The messages themselves are transmitted as a sequence of data packets, each of which is acknowledged by an acknowledge packet. Each link consists of a pair of channels, one in each direction. Data packets are multiplexed with acknowledge packets. The acknowledge packets are useful to signal the reception of data, and to maintain smoothe flow control of data. Each data packet consists of two '1' bits followed by eight data bits, followed by one '0' bit. After transmitting a data packet, the sender transputer waits for an acknowledge packet from the receiver. The acknowledge packet consists of one '1' bit followed by one '0' bit.

Communication over both internal and external channels is essentially byte oriented. The only difference as far as software is concerned is that the control word of an external link is at one of the addresses from 0x80000000 to 0x8000001C, rather than an arbitrary address in memory. The same instructions are used for both internal and external channel communication. Hardware determines whether the communication is internal or external, based on the address of the word.

Communications over internal channels observes the following protocol. Before any communication is attempted, the channel word must be initialized to MinInt. This is normally done when the channel is declared and space is reserved for the channel word. When an input or output is subsequently desired, the channel word is inspected. If it is MinInt, the process descriptor of the current process is placed in the channel word, and the process is descheduled. The IP and the address of the data to be transferred are at locations -1 and -3 relative to the workspace pointer, respectively. If however, the channel word was not MinInt, it has to be the process descriptor of some other process, already waiting to communicate, and the communication is performed. Once the data transfer is complete, the process that was descheduled while waiting for communications to commence, is rescheduled, and the channel word is reset to MinInt.

Instruction Encoding
All transputers share the same basic istruction set. It contains a small number of instructions, each with the same format, chosen to give a compact representation of the operations most frequently occuring in programs. Each instruction consists of a single byte divided into two four-bit parts (nibbles); namely function and data. The most significant nibble is the function code. The 16 functions include loads, stores, jumps, and calls. The least significant nibble is the data code.

All instructions place the data code into the least significatant four bits of the operand register (O), left shifting the existing contents four bits. The contents of O are then used as the operand for the function specified by the function code. Instructions normally clear O after they have executed. However a Prefix instruction does nothing after loading the four bits in O. Thus, by using a series of Prefix instructions the range of the data may be extended. A Negative Prfix instruction complements the contents of O after shifting. Finally, an Operate instruction uses its operand as an opcode, giving the transputer access to instructions beyound the basic 16. Since the operand can be extended by prefixing, atransputer may have an arbitrary number of instructions. The only limitaion to this extenibility is that only 13 instructions (the basic ones excluding Prefix, Negative Prefix, and Operate) can have immediate operands. The rest must have implicit operands. For example, the Add instruction adds the contents of A and B together storing the results in A.

The IMS T800 also has additional instructions which load into, operate on, and store from the floating-point register stack. It also has new instructions supporting color graphics, pattern recognition, and implementing error correction codes. It retains the instruction set of the T414. The extension is possible because of the extensible instruction encoding used in transputers.

Instruction Set Overview

This section presents a discussion on the instruction set for the T414. The T414 has 100 instructions which can be grouped as follows :

- 16 addressing and memory acces instructions
- 6 branching and program control
- 41 arithmetic and logical
- 12 process scheduling and control
- 16 inter-process communication
- 9 miscellaneous

Addressing and Memory Access Instructions

There are two ways of addressing memory; namely to specify the address as a fixed offset from the address in the workspace pointer or the A register. Specifying the address as a fixed offset in the workspace pointer is called "local access", since the workspace pointer conventionally points to an area of memory used to hold the local variables for a procedure. Specifying the address as a fixed offset in the A register is called "non-local access".

Instructions are provided for reading and writing bytes or blocks of bytes to and from the memory using the above mentioned addressing schemes

Instruction	Brief Description
LDL n	Load Local. Loads a word into Reg A which lies at an offset of n words from the address pointed to by the workspace pointer.
STL n	Store Local. Stores a word from Reg A to the memory location which lies at an offset of n words from the address pointed to by the workspace pointer.
LDNL n	Load Non-Local. Loads a word into Reg A which lies at an offset of n words from the address pointed to by the A Register.
STNL n	Store Non-Local. Stores a word from Reg A to the memory location which lies at an offset of n words from the address pointed to by the A Register.
LDLP n	Load Local Address. Loads into the reg A the effective address that is calculated instead of the contents of the address
LDNLP n	Load Non-Local Address. Loads into the reg A the effective address that is

	calculated instead of the contents of the address
LB	Load Byte. Loads the byte at the address contained in Reg A into Reg A thus overwriting the previous contents of Reg A.
SB	Store Byte. Stores the byte contained in the bottom eight bits of Reg B in the memory location specified by the address contained in Reg A.
MOVE	Move Message. Copies a block of bytes from the address in Reg C to the address in Reg B. The number of bytes to be moved is specified in Reg A.
WSUB	Word Subscript. Increments the address in Reg A by the number of words specified in Reg B.
BSUB	Byte Subscript. Increments the address in Reg A by the number of bytes specified in Reg B.
WCNT	Word Count. Breaks the address in Reg A into word address and the byte offset components, storing them in Regs A and B respectively.
BCNT	Byte Count. Multiplies the value in Reg A by the number of bytes in a word
AJW n	Adjust Workspace. Increments the workspace pointer by n
GAJW n	General Adjust Workspace. Exchanges the contents of the workspace pointer and Reg A.
LDPI	Load Pointer to Instruction. Calculates an address, which consists of the current value of the IP incremented by the contents of Reg A and then stores this value in Reg A.

Branching and Program Control Instructions
The transputer provides only has six instructions for altering the flow of control of the program, which brings it close to being a RISC processor.

Instruction	Brief Description
CJ n	Conditional Jump. Examines the contents of Reg A. If the contents are 0 (zero), the IP is incremented by n. The value in A is popped in either case
J n	Jump. Increments the IP by n.
LEND	Loop End. Used to implement deterministic loops. Reg A has the displacement which is subtracted from the IP if the instruction succeeds. Reg B has the address of a two word control block. Each time the instruction is executed the contents of the first word are incremented, and the sceond decremented. If the value in the second word is greater than the first the branch is taken by replacing the contents of IP with (IP - contents of Reg A).
CALL n	Call. Decrements the workspace pointer by four words, stores the contents of C, B, A and the IP at these memory locations, and then increments IP by n.
RET	Return. Loads the IP with the contents of the workspace pointer and increments the workspace pointer by 4 words
GCALL	General Call. Exchanges the contents of Reg A and the IP

Arithmetic and Logical Instructions

Instruction	Brief Description
REV	Reverse. Swaps the contents of Regs A and B.
LDC n	Load Constant. Loads n in A and pushes the previous contents of Reg A to B,

	previous contents of B to C, and losing the previous contents of C.
MINT	Minimum Integer. Pushes the constant MinInt onto the evaluation stack.
ADC n	Add Constant. Adds n to the contents of A.
ADD	Signed Addition. A <= A+B. Error flag is set on overflow
SUB	Signed Subtraction.
MUL	Multiply.
DIV	Divide.
REM	Remainder.
SUM	Unsigned Addition.
DIFF	Unsigned Subtraction.
PROD	Unsigned Multiplication
FMUL	Fractional Multiply
EQC n	Equals Constant. Compares n with the contents of A. If they are equal true (1) is stored in A otherwise a false (0) is stored in A.
GT	Greater Than. If B > A, A<= true (1) else A<= false (0).
CSUB0	Check Subscript from Zero
CCNT1	Check Count from One
AND	Bitwise AND
OR	Bitwise OR
XOR	Bitwise XOR
NOT	Complement
SHL	Shift Left
SHR	Shift Right
LADD	Long Add
LSUM	Long Sum
LSUB	Long Subtract
LDIFF	Long Difference
LMUL	Long Multiply
LDIV	Long Divide
LSHL	Long Shift Left
LSHR	Long Shift Right
XDBLE	Extend to Double. Converts a single word value into a double word value
CSNGL	Check Single. Sets the error flag if the value cannot be squeezed into a single word value.
XWORD	Extend to Word
CWORD	Check Word
NORM	Normalize. Normalizes the double word in BA by shifting it left until the top bit is set.
CFLERR	Check Floating Point Infinity or Not A Number (NAN).
LDINF	Load Single Length Infinity.
POSTNORMSN	Post Normalize Correction.
ROUNDSN	Round Single Length Floating Point Number.
UNPACKSN	Unpack Single Length Floating Point Number.

The last five instructions are not included in the T800 transputer, since it has a floating point coprocessor.

Process scheduling and Control Instructions
The transputer has built in mechanisms to support the concurrent execution of processes. Processes may be operated at two levels of priority. Two queues of active processes are maintained, a high priority process queue and a low priority process queue. A process can be in one of four states : executing, waiting

to execute (it is in one of the active process queues), waiting for a timer event (it is in a timer queue) or waitng for a communication event (it is not in any queue). A high priority process will continue to execute untill it terminates or waits for a timer or communication event to take place. In this case if there are any other high priority processes waiting to be executed, the process from the head of the high priority process queue is scheduled. If there are no high priority processes waiting to execute the next low priority process will be scheduled. A low priority process can be preempted by a high priority process at any time. Low priority processes are time-sliced. If a low priority process executes a loop end (LEND) or a Jump(J) instruction, and it has been executing for more than its time-slice period then it is descheduled and placed at the end of the low priority process queue. The low priority process at the head of the queue will commence execution until the nect time-slice or it is preempted by a high priority process.

Instruction	Brief Description
STARTP	Start Process. Adds a new process to the back of the active process queue. Reg A holds the address of the workspace that the new process will use, and Reg B holds the offset from IP that the new process will execute. The new process is set at the same priority as the current process, and the current process will continue execution with the nect instruction.
ENDP	End Process. Conditionally terminates the process. It decrements a count in memory. If this count is non-zero it terminates (the next active process is taken and the current process is not added to the end of the queue) otherwise the process continues but at a different execution address and workspace counter. It takes one parameter, an address in Reg A. This address points to the workspace of the parent process At location 0 in this workspace is the restart address for the parent process, location 1 holds the count of the child processes. If this count decrements to zero the workspace pointer is set to the value in the A register, and its IP to the value pointed to by Reg A.
STOPP	Stop Process. Stops the current process. This process is not added back to the end of the queue. The nect active process is scheduled.
STOPERR	Stop On Error. Same as STOPP, but only if the error flag is set.
RUNP	Run Process. Starts a new process. Reg A has the process descriptor. A process descriptor is the address of the workspace for that process with the bottom bit set to the priority of that process. (A workspace lies on a word boundary making the last few bits unused for address specification). Based on the priority the process is added to the end of the appropriate queue. The IP for that process is set to one word below the workspace address.
STHF	Store High Priority Front Pointer
STHB	Store High Priority Back Pointer
STLF	Store Low Priority Front Pointer
STLB	Store Low Priority Back Pointer
SAVEH	Save Low Priority Queue Registers.
SAVEL	Save Low Priority Queue Registers.
LDPRI	Load Priority. Loads 0 or 1 into A based on the priority of the current process

Inter-Process Communication Instructions

Inter-Process communication is carried out using channels. A channel is an abstract connection between exactly two processes. One process sends a stream of bytes down the channel to another process, which reads and stores them. If a process tries to send data to another process which is not ready, then the sending process will be descheduled until the receiving process becomes ready, and thus synchronization is achieved.

For two processes running on the same transputer, a channel is implemented by using a word somewhere in memory that is shared by the two processes. This word is referred to as the channel word. Before any communication can take place this word must be inialized to MinInt (Minimum Integer). This with the transputer's signed address space is interpreted as the lowest address in memory. This address can never be the address of a valid workspace. The channel word would normally be initialized by the parent process of two communicating children. When a process tries to communicate, it examines the contents of the channel word. If the contents are MinInt, the other process is not ready, so it stores the process descriptor in the channel word and deschedules. The IP is stored in location -1 of its workspace and the address of the data to be transferred from or to at location -3. When the other process is ready, it examines the contents of the channel word, finds a valid process descriptor and concludes that the other process is ready. It transfers the block of bytes and restarts the other process. All this is implemented within the microcode of the transputer and hence is extremely fast and efficient.

For two processes running on two separate transputers, the two transputers must be connected by hardware INMOS links. Here the communication via channels is taken care of via the hardware within the transputers themselves. When a process on one transputer is ready to send data out to another process on another transputer, the sending process is descheduled. Hardware determines when the other transputer is ready, and transfers the data using Direct Memory Access (DMA), after which the sending process is rescheduled. While this is happening other processes may be executed. At machine code level it appears the same as the former case; the same instructions are used. The distinguishing factor is that in external communication the address of the channel word passes as a parameter in one of the first few locations of memory. The microcode instruction detects the address in one of the reserved locations and pass their parameters on the link hardware for the lik associated with that address.

The proble with the above communication procedures is that the sending process hangs indefinitely until the other process is ready. The transputer also has a suite of instructions, collectively referred to as ALT instructions which allow a series of events to be chosen. The way that ALTs are implemented is described as follows. The first thing that is done is to start the ALT by setting some flags to indicate that an ALT sequence is taking place. Then various guards must be enabled; this is the process of checking channel words used by the guards to see if any of the processes are ready to communicate, finding the earliest time that a timer guard is waiting for and so on. Following this is the wait, which is where the process goes to sleep, until such time that a guard fires. Next, each guard is disabled, and finally, the ALT is ended.

Instruction	**Brief Description**
OUT	Output Message. Reg A contains the number of bytes to transfer, B points to the channel word, and C points to the start of the data to be sent.
OUTWORD	Output Word.
OUTBYTE	Output Byte.
IN	Input Message.
RESETCH	Reset Channel. Sets the internal channel word to MinInt or resets the link hardware for an external channel.
ALT	ALT Start. Sets the word at loaction -3 in the workspace to MinInt+1. This indicates that no guards have been fired yet. This location also corresponds to the location where the data address is stored in an IN or OUT instruction. This means that another process will be able to tell that the process is an ALT process because MinInt can never be a valid date address.
TALT	Timer ALT Start. Does everything that ALT does and then sets location -4 to MinInt+2, indicating that no valid time has yet been stored.
ENBC	Enable Channel.
ENBS	Enable Skip.
ENBT	Enable Timer.
ALTWT	ALT Wait.

TALTWT	Timer ALT Wait.
DISC	Disable Channel.
DISS	Disable Skip.
DIST	Disabel Timer.
ALTEND	ALT End. Performs a relative jump based on the contents of location 0.

Miscellaneous Instructions

Instruction	Brief Description
STTIMER	Store Timer. Sets the values of the low and high priority timer registers to the value in Reg A.
LDTIMER	Load Timer. Pushes the value of the current priority timer register into Reg A.
TIN	Timer Input. Suspends the current process until the time in the current timer register is after the value specified in Reg A. Other processes may execute in the mean time. The current process in inserted into a timer list of sorted processes. Each time the timer register is incremented it is checked against the process at the head of the list, and if necsssary the process is reactivated.
SETERR	Set Error Flag.
TESTERR	Test Error Flag. A<= false(0) if the error flag is set.
CLRHALTERR	Clear Halt on Error Flag.
SETHALTERR	Set Halt on Error Flag.
TESTHALTERR	Test Halt on Error Flag.
TESTPRANAL	Test Processor Analyzing.

Example Assembly Level Program

Example : Implement the following C code

```
for( i=3; I<=8; I++)
   j=j+i;
```

Solution :

```
        LDC     3;                  --line 1
        STL     block;              --line 2
        LDC     8+3-1;              --line 3
        STL     block+1;            --line 4
L1:     LDL     j;                  --line 5
        LDL     block;              --line 6
        ADD;                        --line 7
        STL     j;                  --line 8
        LDC     L2-L1;              --line 9
        LEND;                       --line 10
L2:     ....
```

In line 1, the value 3 is loaded into Reg A, line 2 stores this value from Reg A to the memory location at an offset of “block” from the address pointed to by the workspace pointer.
In line 3, the value 8+3-1 = 10, is stored in Reg A, pushing the previous value from Reg A (3) into Reg B.

Once again in line 4, the value from Reg A is stored in the memory location at an offset of "block+1" from the address pointed to by the workspace pointer. Reg A once agian holds 3 now.
Lines 1 through 4 makes up the control block
In line 5, the value in the memory location at an offset of "j" from the address pointed to by the workspace pointer is loaded in Reg A, pushing the previous contents of B to C, and A to B.
In line 6, the value in the memory location at an offset of "block" from the address pointed to by the workspace pointer is loaded in Reg A, pushing the previous contents of B to C, and A to B.
At this point Reg C has the offset of the control block relative to the workspace pointer.
Line 7 adds the contents of Reg A and B (i+j) and stores the result in Reg A.
Line 8 stores this value in Reg A in is stored in the memory location at an offset of "j" from the address pointed to by the workspace pointer. Also, the contents of B are popped into A.
Line 9 loads A with the displacement.
Line 10 is very critical. At this point Reg A contains the displacement, and Reg B contains the control block (block, and block+1). The contents of "block" are incremented by one, and the contents of "block+1" are decremented by one. As long as the contents of "block" are less than that in "block+1" the contents of Reg A are subtracted from the IP, and the loop is taken.

OCCAM : THE NATIVE LANGUAGE

Introduction

Occam is an abstract programming language whose development has been closely associated with that of the transputer. (Both have been designed by INMOS). Although linked with the transputer, the importance of occam goes beyond its implementation on any hardware system. It is a result of many years of research, and is a concurrent programming language that is simple, elegant and powerful.

The name occam was chosen in recognition of its simplicity. Wiiliam of Occam, a 14th century philosopher said "Entities are not to be multiplied beyond necessity", which is a plea to keep things simple.

InterProcess Communication in Occam

Occam uses a single straightforward structure that encompasses both ease of programming, and ease of implementation. A synchronous communication method was chosen that combines in a single primitive, the needs of data communication and synchronization. The communication is built on the use of what is called a channel. Two factors have in particular influenced this.

1. Occam incorporates an exhaustive view of concurrency; programs consist of a large number of processes and it would be inconvenient to name them all. Indirect naming allows processes to be anonymous.
2. Modifications to occam programs are easier to accomodate if the communication between processes takes place via an explicitly defined intermediary.

The channel is unidirectional and can only be used by one calling process and one called process. Other characteristics of the channel have also been influenced by the need to associate a channel with a link between adjacent transputers. The commands for writing to and reading from have a simple and terse format. Let 'ch' be the channel. Then the command to write to this channel, the value contained in the variable X, would be :

```
ch ! X
```

The symbol ! indicates output.
Similarly, to read the channel and store the results in the variable X, the command (from another process) would be :

```
ch ? X
```

The symbol ? indicates input

The first process to execute one of these commands is suspended until the other process catches up. Both processes will then proceed, independently and concurrently.

Deadlocks And Indefinite Postponements

Deadlock is a situation that occurs when two or more processes are waiting on each other and it is impossible for any one of them to proceed without external arbitration.
Consider two occam processes P1 and P2 wishing to exchange some data. Let chan1 and chan2 be the two channels.

```
P1 :    chan1 !  A
        chan2 ?  B

P2 :    chan1 ?  X
        chan2 !  Y
```

The above code will work correctly under all circumstances.
However, if the two statements in process P2 are reversed

```
P2 :    chan2 !  Y
        chan1 ?  X
```

both processes P1 and P2 will be suspended indefinitely. P1 can not proceed until P2 has read from chan1, and P2 can not proceed until P1 has read from chan2. Predicting deadlocks is not easy, and they sometimes occur rarely, but with devastating results.

A less severe, though still significant, condition is that of indefinite postponement (also referred to as lockout or starvation). This may occur, for example, when a low priority process tries to access a busy resource. Even if the postponement is not indefinite, but merely indeterminate, the program behaviour may be unpredicatable. It is very important that the run-time system implementing the concurrent execution of the processes implements a fair scheduling algorithm, such as the first-in first-out (FIFO) queue or a round-robin algorithm.

Occam Processes

The basic model for an occam program is a network of communicating processes. Communication is defined via channels with each process within the program performing a sequence of actions which may proceed indefinitely. A process may also contain other processes so that a heirarchical structure is supported. At the top level the entire occam program is a single process.

An occam process can be of one of the following types.

- Primitive Process
- Block
- Constructor
- Procedure Instance

Primitive Processes

Occam supports five primitive processes : STOP, SKIP, assignment, input, and output.

- STOP is a process that has no action but which never terminates. Its effect is to therefore inhibit any parent process from continuing. It is quite drastic, and mostly only executed in response to some error condition.
- SKIP is the inverse of the STOP process although it has no effect. SKIP is always ready to execute and terminates immediately. It is a null process.

- An assignment has the form
  v := e
  where v is the variable, and e is the expression of the same type. An assignment process will normally terminate immediately, unless an error occurs and a STOP process is executed.
- A novel feature of occam is that communication is viewed at the same level as assignment. As mentioned in the previous section, ch ! e, is an output and, ch ? x, is an input.

Blocks and Channels

A block process is a specification followed by a process. Blocks may be contained within other blocks, and the usual scope rules apply. There are three types of channels; namely input, output and timer channels.

- Input and output channels are declared using the CHAN keyword as follows :

  [Number] CHAN OF <Protocol type> <channel name list> :
  where, Number is only required to declare an array of channels
  Protocol type specifies the channel protocol
  channel name list is the list of channel names.

- Timer channels are declared as follows :
  Timer <list of names> :
  A timer channel has the characteristic that its always ready to output. Obviously, only read actions on timers are allowed.

Constructors

Constructors provide the glue for putting together the primitive processes. Five distinct constructors are provided : SEQ, WHILE, IF, PAR, and ALT. With each of these a replicator may be attached (except in WHILE) to give an extra dimension.

- SEQ provides for sequential execution of a collection of processes. The collection may be of any size.
- To execute a sequence a number of times, the WHILE construct is used. The sequence is executed as long as the expression in the WHILE construct evaluates to TRUE.

```
eg.     INT A:
        SEQ
          A := 0;
          WHILE A<1024
            SEQ
              in ? A
              A := A * A
              out ! A
```

This code executes as long as A < 32 or A > -32.
A replicator is used to duplicate a component process a number of times, just like a for loop in many other high level languages.

```
eg.     SEQ I= 0 FOR 10
          CNT ! I
```

transmits integers 0 through 9
A replicator can only be applied to a single subprocess. To execute a collection of subprocesses, they must be wrapped into a SEQ process.

- Occam provides conditional branching based on the IF construct.

eg.
```
IF
  X < 0
    Y := -1
  X > 0
    Y := 1
```

- In order to indicate that two or more processes ar concurrent the PAR construct is used. All subprocesses of PAR must be completely independent or interact using channels. From a program execution standpoint PAR has two main uses
  1. To express concurrency, or to indicate candidates for true parallel execution.
  2. To introduce non determinacy into the program

  PAR itself is a process, and terminates when and only when all of its constituent subprocesses terminate. PAR can also be used in conjunction with a replicator when a number of similar processes are to be generated.

eg.
```
VAL INT N IS 10
[N+1]CHAN OF INT C:
PAR
  PAR P= 0 FOR N
    INT BufferElement :
    WHILE TRUE
      SEQ
        C[P] ? BufferElement
        C[P+1] ! BufferElement
```

eg. If P(I) and Q(I) are two occam processes whose actions depend on the value of I then the following code :

```
PAR
  P(1)
  P(2)
  P(3)
  Q(1)
  Q(2)
  Q(3)
```

could be written more concisely as

```
PAR I = 1 FOR 3
  PAR
    P(I)
    Q(I)
```

When a PAR process is executed, a number of subprocesses are created. With a replicated PAR process the number of subprocesses created is determined by the number of replications. Each of these subprocess in turn may create a number of other subprocesses. Occam, however does not allow dynamic process creation, and hence the number of processes and subprocesses under each one of these processes must be known at compile time. This restriction is made to allow occam to determine memory allocations and calculate queue lengths. Therefore the second field of a replicator used with a PAR is restricted to an expression that can be calculated at compile time. Consider the code fragment

```
SEQ
  NumProcesses ? N
  PAR I = 1 FOR N
     SubProcess(I)
```

This is not allowed.
However, the above code fragment can be re-written as follows

```
VAL INT MAX IS 100
NumProcesses ? N
SEQ
  IF
    N > 100
    SKIP
 PAR I = 1 TO MAX
      IF
        I <= N
          SubProcess(I)
        TRUE
           SKIP
```

Here, the maximum number of processes (MAX) is always generated. However, all processes greater than the number of processes required are merely SKIP processes, which terminate immediately.

The PAR constructor also allows the concurrent processes generated to have different priorities. The priority of a process determines how much relative processing time the process gets. The higher the priority of a process, the more processing time it gets.

eg.
```
PRI PAR
  P1
  PAR
     P2
     P3
     P4
   P5
```

In order to assign priority to the concurrent processes, a variant of PAR, PRI PAR must be used. As in the example P1 has highest priority, P5 has lowest priority, and processes P2, P3 and P4 have a priority in between those of P1 and P5. The priorities are assigned from highest to lowest to the processes in the order in which they appear.

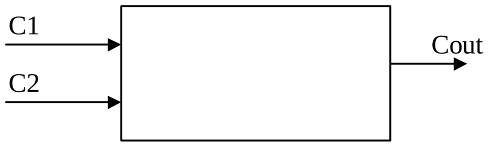
Once a program has been designed, developed, and verified; its final execution may be assigned to a neumber of processors. Another variant of the PAR constructor is used for this purpose. The PLACED PAR indicates that the associated subprocesses are not only concurrent, but they are to alloted to different processors, and hence will be truly parallel.

eg.

```
PLACED PAR
   PROCESSOR 1
      P1
   PROCESSOR 2
      P2
```

- The ALT constructor allows a flexible construct of selecting one process of various alternative processes.

eg.



```
INT X:
WHILE  TRUE
    ALT
       C1 ? X
          Cout ! X
       C2 ? X
          Cout ! X
```

For each execution of ALT one, and only one, of the alternatives is selected. The selection of the alternative is based on a guard condition. When a guard condition is ready, the process associated with it is executed. If no guard is ready, the ALT process is suspended until there is one.

eg.

```
TIMER Clock:
CHAN OF  INT IS Input
INT time, X :
SEQ
   Clock ? time
   ALT
      Input ? X
        -- Normal Action
      Clock ? AFTER time PLUS 1000
        -- Timeout Action
```

Depending on which guard condition gets ready first, normal action or timeout action is taken.

Like the PAR construct, a replicator may be used with the ALT construct.
If more than one guard condition becomes ready at the same time, the one chosen for execution is not defined in the language. Hence a variant of the ALT, PRI ALT is provided, which prioritizes the guard conditions. Once again, priorities to the guard conditions are assigned the same way as in the PRI PAR construct; the one textually first gets the highest priority and so on.

Channel Protocols

In order to avoid run-time errors, and also, to make channel communications more efficient, occam has added protocols to the definition of the language. Once a channel is defined, both input and output operations must be compatible with the defined protocol. A protocol is a statement about the type of object using the channel.

Simple Protocols

A simple protocol is used to pass a single object, or a variable array down a channel. A simple protocol need not be explicitly defined; it can be used directly in the definition of the channel.

Simple protocol for Primitive types
A primitive protocol is merely a simple scaler type, for example, an integer.
eg. CHAN OF INT C1 :
The operation C1 ! X or C1 ? X is compatible with the protocol as long as X is of type INT.

Simple protocol for Fixed Length Array types
An array is a group of elements all of the same type. For example, in order to communicate an array of five integers the following code fragment may be used

```
CHAN OF [5]INT C5:
PAR
   [5]INT OUT :
   SEQ
      C5 ! OUT
   [5]INT IN :
   SEQ
      C5 ? IN
```

Another variant of the code would be, for example,

```
CHAN OF [5]INT C5:
PAR
   INT OUT1, OUT2, OUT3, OUT4, OUT5:
   SEQ
      C5 ! [OUT1, OUT2, OUT3, OUT4, OUT5]
   INT IN1, IN2, IN3, IN4, IN5:
   SEQ
      C5 ? [IN1, IN2, IN3, IN4, IN5]
```

In both the above code fragments, the usage of the channel is compatible with the protocol since each time five integer variables are used.

Simple protocol for Variable Length Array types
This is a more general type protocol than the previous one. The usage is as follows

```
CHAN OF BYTE::[]INT C256:
```

The above declaration would allow upto 256 integer values (BYTE is a one byte location) to be sent or received over C256.

```
eg.   C256 ! 7::[23,24,56,34,1,2,567]
      C256 ! 17::[A FROM 6 FOR 23]    where A is an array of atleast 23 integers.
      C256 ? num::[B FROM 0 FOR num]
```

In each of the above three examples, the first value (in this case of type BYTE) is the number of data elements (in this case type INTs) to be transferred, and then the actual data follows.

Simple protocol for Record types
Records are a collection of elements of different types, which are grouped together as a single entity. For example, a student record may contain is name, major, and grade.

eg.
```
RECORD IntAndByte IS (INT,BYTE):
CHAN OF IntAndByte Crec:
PAR
  IntAndByte IBRec:
  INT I:
  BYTE B:
  SEQ
    Crec ! IBRec
    Crec ? (I,B)
    Crec ! (45,'A')
```

Sequential Protocols

Arrays and Records can be used to communicate groups of objects down channels as single logical transactions. A sequential protocol allows such groups to be communicated without the use of such structured types. A sequential protocols gives the type of each object being transferred.

eg. C ! x1;x2;x3;x4 is compatible with the protocol P1;P2;P3;P4 as long as x1 is xompatible with P1,and so on.

A sequential protocol must be defined before it can be used

eg.
```
PROTOCOL  FiveInt IS INT;INT;INT;INT;INT:
CHAN OF FiveInt C5:
PAR
  ch ! 2,45,3345,23,5678
  ch ? in1,in2,in3,in4,in5
```

Variant Protocols

What if two processes wished to pass data of different types and in no particular order ? While solutions exist using simple and sequential protocols, they are too expensive in terms of the number of channels used and in the possiblity of a deadlock situation arising. Variant protocols come to the rescue.

eg.
```
PROTOCOL INT.OR.REAL
  CASE
    Fixed ; INT
    Float ; REAL32
 :
CHAN OF INT.OR.REAL C.all
PAR
  SEQ
    C.all ! Fixed ; I
    C.all ! Float ; R
  SEQ
    C.all ? CASE
```

```
Fixed ; J
Float ; X
```

ANY Protocols
If the channels is defined to have a protocol ANY, then all possible uses of the channel are legal, and the compiler does not perform any type checking. The use of ANY protocol must be very restricted and with great care.

Occam Input And Output

High Level Input And Outut
The basic model for I/O in occam for channels to pass data between the program and the program's environment. A typical program would therefore have many internal channles and a few external ones. The external channels would be half inside the program and half outside the program. Such channels are specified by explicitely allocating the channel

eg.
```
CHAN OF INT C:
PLACE C AT X:
```
where X is an implementation dependent constant.

On some implementations of occam, 1 indicated the VDU screen, and 2 indicated the keyboard

eg.
```
CHAN OF ANY Screen :
PLACE Screen AT 1:
CHAN OF INT Keyboard :
PLACE Keyboard AT 2:
INT char:
WHILE TRUE
  SEQ
    Keyboard ? char
    Screen ! char
```

Other channel addresses can be used to interact with files peripheral devices. If an implementation supports such facilities then there will be addresses defined. A more useful interface to a program's environment, however, would be achieved if standard library PROCs were available that had the appropriate channel definitions hidden.

Low Level Input And Output
As the transputer does not provide special instructions for controlling external devices, occam only caters memory-mapped I/O. With memory-mapped I/O, the registers that form the interface between the main processor and the device are located at specific addresses of the processor's address space. To enable an occam program to use these memory addresses, they must also be within the program's address space. These memory locations are used to control the external device, and to send and receive data back and forth, from the transputer and the external device.

An Example Of A Systolic Algorithm

One of the reasons of having concurrency within the programming language is to be able to present, and implement algorithms that are designed for parallel execution. A systolic algorithm is one that has a collection of identical processes through which data flows. In a "pure" systolic algorithm, each process executes identical sequence of instructions. A system of transputers is ideally suited to implement such an algorithm. Each transputer can hold a process, and data can flow through the links. Although each transputer is running the same process, the code will contain, fo example, "if" processes, which will cause different instructions on different transputers.

Let us consider a well known problem of matrix multiplication. Let A and B be two M X M square matrices, and R is a M X M matrix such that

R = A * B

where the element of R in the ith row and jth column is given by

$r(i,j) = \sum a(i,k) * b(i,k)$ where k ranges from 1 to M.

A parallel multiplication algorithm would map each element of R onto a process. Each process receives a flow of values (matrices A and B). A typical process is shown

```
PROC Multiply( CHAN OF INT N,S,E,W)
    INT Result :
    INT A,B :
    SEQ
        Result := 0
        SEQ I = 1 FOR M
            SEQ
                PAR
                    N ? A
                    W ? B
                Result := Result + (A*B)
                PAR
                    S ! A
                    E ! B
```

M pairs of values are read, Result is updated accordingly, and the values are passed on. For a M X M matrix, M * (M+1) channels are required in both the vertical and horizontal planes.

The following harness program is required to multiply the two matrices

```
[M*(M+1)]CHAN OF INT Vert,Horz:
PAR
    PAR I = 1 FOR M
        PAR J = 1 FOR M
            Multiply(Vert[M*i)+j], Vert[(M*i)+(j+1)], Horz[M*i)+j], Horz[(M*(i+1))+j])
```

M is a constant. Also, processes for generating matrices A and B are required, but are not shown.

Tranforming the matrix PROC

Step 1.

```
SEQ I = 1 FOR M
  SEQ
    PAR
      N ? A
      W ? B
    Result := Result + (A*B)
    PAR
      S ! A
      E ! B
```

Step 2.

```
SEQ I = 1 FOR M
  SEQ
    PAR
      N ? A
      W ? B
    PAR
      Result := Result + (A*B)
      PAR
        S ! A
        E ! B
```

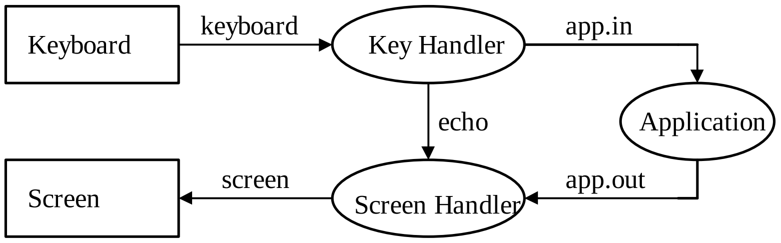
Step 3.

```
SEQ I = 1 FOR M
  SEQ
    PAR
      N ? A
      W ? B
    PAR
      Result := Result + (A*B)
      S ! A
      E ! B
```

Transformations are logical techniques applied to a parallel program to obtain a functionally equivalent (and hopefully more efficient) version of the program. Although, research in still ongoing on these techniques, they seem to hold great promise for the occam programmer.

An Example System Design

System Topology



The above diagram show a fuctional division of a generic application into a keyboard handler, screen handler, and the application itself. Such a division is for ease of programming, rather than performance. The first step in the system design would be to design separate procedures for the keyboard handler, the screen handler, and the application. An example of a keyboard handler is shown

```
PROC keyboard.handler( CHAN OF INT in, out, to.screen)
  INT ch:
  VAL stopch IS INT 27:
  BOOL running:

  SEQ
    running := TRUE
    WHILE running
      SEQ
        in ? ch
        PAR
          out ! ch
          to.screen ! ch
        IF
          ch = stopch
            running = FALSE
          TRUE
            SKIP
:
```

The screen handler procedure would require the use of an ALT construct to enable it to accept input from the application and the keyboard handler.

Once all three procedures have been tested individually, and as a system on a single transputer system, the next step would be to adapt it for the target system. Consider a three transputer system in our case. A typical assignment would be as shown

```
...SC keyboard.handler
...SC screen.handler
...SC application
```

```
CHAN OF INT keyboard, screen, echo, app.in, app.out:

PLACED PAR
  PROCESSOR 0
    PLACE keyboard AT link0in:
    PLACE echo AT link1out:
    PLACE app.in AT link2out:

    keyboard.handler( keyboard, app.in, echo )

PLACED PAR
  PROCESSOR 0
    PLACE screen AT link0out:
    PLACE echo AT link1in:
    PLACE app.out AT link2in:

    screen.handler( screen, app.out, echo )

PLACED PAR
  PROCESSOR 0
    PLACE app.in AT link0in:
    PLACE app.out AT link1out:

    application( app.in, app.out )
```

Programming Transputers Using C

The software used to develop the examples in this paper is, an ANSI C compiler augmented with libraries written by Logical Systems to support parallel constructs. The package contains the standard preprocessor, compiler, assembler and linker programs that most C programmers will be familiar with, but also new utilities, such as a loader and a host driver, required by the parallel environment. Each program is covered in great technical detail in Logical Systems' C93.1 User Manual.

Communication Primitives

Channels, hard channels, soft channels
Tasks executing on different transputers communicate with each other through channels. A channel is a means of communication for parallel tasks. It is very tempting to associate a channel with a transputer link. In many cases the two will be the same. In such instances the channel is referred to as a hard channel. Transputers are also designed to run internal "parallel" tasks as well, through a technique called multitasking. When tasks reside on the same transputer, they can use the same mechanism to communicate with each other. The concept of channel encompasses that of the link, and extends beyond it. It is one level of abstraction away from the hardware link.

We will start with transputer-to-transputer communication, examining how programs running on neighboring transputers exchange information. This will allow us to divide our application among several transputers. The next step will be to look at how parallelism can be implemented within a transputer, and to create parallel tasks running on a single transputer. Once we know how to control inter-processor parallelism, as well as internal multitasking, we will combine the two and build a general framework for parallel programs.

The Network Topology

For our multi-transputer system, let us consider the simple network topology offered by a linear chain. We will do this for two reasons. First, a chain is compatible with all transputers, and second, the chain is sufficiently simple that it will allow us to concentrate on the task-level parallelism rather than on the network parallelism. This configuration is just as easily implemented with single transputer boards as it is with more sophisticated multiprocessor systems. The chain is also an important network configuration to study for systems based on T400 transputers which have only two operational links, and because it will introduce us to the discipline of carefully planning and laying out our application on a physical network.

To program an application on any network, it is essential to know the exact details of the interconnect linking the transputers to each other. Since each transputer has two potential links it can use to exchange information with its neighbors, we need to know how each transputer is connected to its neighbor, and which links they are both using to communicate with each other.

Parent node-Child node

Our goal is to, as much as possible, wire the transputers in a uniform way. This uniformity calls for some definitions. The first definitions that we will introduce are those of **parent nodes** and **children nodes** in the chain. The parent/child relationship is similar to that in a tree, except that in our case the tree is unary (a chain). The **Root** transputer coincides with the root of the tree. Each transputer in the chain talks to its parent node over the same link, and to its child node over the same link. Since the root transputer communicates with the PC host through its Link0 (the PC/link) by default, this automatically sets the rules of our convention.

Rule 1
The PC host will be referred to as the parent of the root node. In turn, the root node is the parent of the next node down the chain.
Rule 2
We will assume that the parent node is always on the left of its child node (if any). The leftmost entity is therefore the PC host. The next to leftmost is the root node. A child node will therefore be on the right of its parent.
Rule 3
Each transputer will communicate with its parent via its Link0, and with its child (if any) through its Link1. The Root Transputer will be given the numeral 1, its child node will be labeled 2, and so on.

Transputer to Transputer Communication

Let's start with a simple program that computes prime numbers. This algorithm is an algorithm of choice when dealing with parallelism, as it can be easily be tailored to many approaches. We have chosen an implementation of the prime-finding algorithm that is not the most efficient, but extremely simple to implement.

The algorithm works as follows. To find if integer x is prime, the program attempts to divide x by all integers lower than sqrt(x) Testing up to sqrt(x) is sufficient, but requires a square root or a multiplication operation.

```
#include <stdio.h>
#include <stdlib.h>
#include <conc.h>                          /* transputer library */

/* ============================ GLOBALS ============================== */
#define INTSIZE            (int) sizeof(int)
#define INTERVAL            100
#define SENTINEL            -1

main()
{
    int NoPrimes = 0, x, j;
    if (_node_number==1)
    {
        do
        {
            /*--- get input-channel address ---*/
            ChanIn(LINK1IN, &x, INTSIZE);
            if (x!=SENTINEL)
            {
                printf("%8d", x);
                NoPrimes++;
            }

        } while (x!=SENTINEL);
        printf("\nReceived %d primes \n", NoPrimes);
        exit(0);
    }
    else
    {
        for (x = 1; x<INTERVAL; x++)
        {
            /*--- if prime then send number to Node 1 ---*/
            if (IsPrime(x))
                ChanOut(LINK0OUT, &x, INTSIZE);
        }

        /*--- signal Node 1 that we are done ---*/
        x = SENTINEL;
        ChanOut(LINK0OUT, &x, INTSIZE);
    }
}

int IsPrime(int x)
{
    int i;                  /*  0 1 2 3 4 5 6 7 8 9 */
    static int SmallPrimes[10] = {0,0,1,1,0,1,0,1,0,0};
    if (x<10) return SmallPrimes[x];
    if (x%2==0) return 0;
    if (x%3==0) return 0;
    if (x%5==0) return 0;
    for (i = 2; i*i<=x; i++)
        if (x%i==0) return 0;
    return 1;
}
```

The output of the program is shown below.

```
 2      3      5        7      11      13      17      19
23     29     31      37      41      43      47      53
59     61     67      71      73      79      83      89
97
Received 25 primes
```

Let's take a closer look at the code.
First, it contains only the standard main() function, which is divided into two parts. The first section contains the code that runs on the root transputer (Transputer 1), while the second one contains the code running on Transputer 2. The same program is loaded on both transputers, but a different section of the code is actually used by each. This mode of programming is called farm programming.

Farm programming
Farm programming consists in putting in one program the many different tasks that must be executed by the different transputers. This program is then loaded on all the transputers, and each transputer selects the section of the program that it has been assigned to. The selection is done by testing the predefined

variable **_node_number**, which contains a different value depending on which transputer tests it. The best place to put the test is in main().

```
main()
{
  if (_node_number==1)
    {
         /* code to be executed by Transputer #1 */
   }
    else
    {
       /* code to be executed by Transputer #2 */
    }
}
```

Note that we could have written the above test as two if-statements, one testing _node_number against 1, the other against 2, with exactly the same result. This method of programming may become quite inefficient when the network contains many transputers. In such a case the program is partitioned into many blocks, only one of which is executed by each transputer which must still allocate enough memory to store the whole program. This method has the advantage of being easy to implement, however, since only one code file needs to be created and compiled. This is the method we shall adopt in this chapter.

conc.h library
Let's go back to the program and analyze it line by line. The first unfamiliar statement is the #include <conc.h> statement. The conc.h library, first implemented by Jeffrey Mock of Pixar, contains functions and variables implementing an Occam-like concurrency model. It must be included every time we write a program for the transputer.

Let's look now at the way Node 1, the Root, receives information from Node 2.

```
if (_node_number == 1)
        do
        {
                /*--- get input-channel address ---*/
                ChanIn(LINK1IN, (char *) &x,INTSIZE);
                 if (x != SENTINEL)
                 {
                        printf("%8d",x);
                        NoPrimes++;
                }
        } while (x != SENTINEL);
        printf("\nReceived %d primes \n",NoPrimes);
        exit();
```

ChanIn()
The ChanIn function stands for Channel-Input. It receives information from a channel, here LINK1IN (a constant predefined in conc.h), and stores it at the address specified by the type-cast pointer &x. LINK1IN is the memory address of the I/O Port corresponding to the input side of the link. The I/O ports are memory-mapped. LINK1IN is thus a macro representing the memory address of the port. INTSIZE is a macro that is defined at the beginning of the program as representing the size of an integer. Hence the function of ChanIn is to receive some amount of information from a channel and to store that information

at a location defined by a pointer. The size of the information received is defined by an integer representing the number of bytes contained in the message. The prototype for ChanIn is:

```
void ChanIn(Channel *, void *, int)
```

The third argument needed by ChanIn is the number of bytes contained in the message. The sizeof function will often be used here. #define will come in handy for that purpose. The example below shows how we can transfer an integer, a string of 20 characters, and an array of 10 integers.

```
int i, table[10];
char string[20];

ChanIn(LINK1IN, &i, (int) sizeof(int));

ChanIn(LINK0IN, string, (int) sizeof(string));

ChanIn(LINK1IN, table, (int) sizeof(table));
```

So, the main loop executed by Node 1 is fairly simple. It gets an integer from the channel LINK1IN, and stores it in x. If the value of x is not equal to the sentinel, it prints it and increments a counter. When a value equal to the sentinel is received, it exits the loops and stops.

Before moving on to Node 2, let's analyze the last statements executed by Node 1:

```
if (_node_number == 1)
{
 /*--- get numbers from Node 2 ---*/
do
{
   ChanIn(LINK1IN, &x,INTSIZE);
   if (x != SENTINEL)
  {
          printf("%8d",x);
          NoPrimes++;
  }
} while (x != SENTINEL);
printf("\nReceived %d primes \n",NoPrimes);
exit(0);
}
```

The call to exit() terminates the portion of the code executed by the root transputer. When the Root transputer reaches the closing bracket of the main function, its execution of the program effectively stops, and it returns control to some internal kernel. But the host computer, the PC, is still executing cio, and will continue executing it until the user stops it with a control-break sequence on the keyboard, or by resetting the computer. How did cio enter the picture ? In the network information file! Can we tell cio that the transputer computation is done and that control of the host computer can be returned to DOS? The answer is by having the root transputer call the exit() function. Here exit() not only terminates the program, it also terminates the cio server.

Exit Rule
Always terminate the code executed by the Root processor with an exit statement. Exit passes a signal to the cio driver running on the host computer instructing it to stop and to return to DOS.

Let's continue now with the code executed by Node 2.

```
else
{
        /*--- scan interval--*/
        for (x = 1; x < INTERVAL; x++)
             if (IsPrime(x))
                  ChanOut(LINK0OUT, &x, INTSIZE);

        /*--- signal Node 1 that we are done ---*/
         x = SENTINEL;
         ChanOut(LINK0OUT, &x, INTSIZE);
}
```

Node 2 executes a loop in which x takes all the values between 1 and INTERVAL, defined as 100. For each new value, x is tested by IsPrime(). If IsPrime() returns a non zero value, x is sent to Node 1 with a call to ChanOut.

ChanOut()
ChanOut stands for Channel Output. It is the companion of the ChanIn function executed by Node 1. Its syntax is exactly the same as that of ChanIn:

```
void ChanOut(Channel *, void *, int)
```

It needs a channel pointer indicating which channel the information is sent over. In our case, the predefined constant LINK0OUT is used. It corresponds to LINK0, and specifies the outgoing direction. This information is then followed by a pointer to the area of memory containing the information to send. Here again a void pointer identifies the memory address of the information to be sent, and the third argument specifies how many bytes constitute the message.

Channel pointers

Why are the channel pointers used by Node 1 and Node 2 different? Node 1 receives its information from a channel associated with a predefined constant LINK1IN, while Node 2 sends its information out through a channel identified by LINK0OUT. Is this correct? The answer is yes. Let's go back quickly to Figure 4-1. Our adopted convention is that a parent talks to its child through its Link 1, while a child responds through its Link 0. So LINK1IN is a constant representing the Link 1 input of a transputer, while LINK0OUT represent the output of Link 0. Because hardware links are bi-directional, two constants are needed per link. These constants are defined in conc.h and allow us to Chan(nel) information In and Out: LINK0IN, LINK0OUT, LINK1IN, LINK1OUT, LINK2IN, LINK2OUT, LINK3IN, and LINK3OUT. Following the convention established at the beginning of this chapter, we now have a new rule to adopt:

Channel Input/Output Rule
On a chain of transputers, communication between a node (parent) and its child node must take one of the following forms only:

Parent ChanOut(LINK1OUT,...)	$\longrightarrow$	Child ChanIn(LINK0IN,...)
Parent ChanIn(LINK1IN,...)	$\longleftarrow$	Child ChanOut(LINK0OUT,...)

When communicating over channels, two important aspects to remember are: Firtst, the communication is blocking and second, there is no buffering of the information.

Blocking
Blocking means that no transfer can take place unless both tasks exchanging data are ready to do so. In the context of our example, this means that if Node 2 is the first one to reach the ChanOut(LINK0OUT,...) statement, then it will not send any information until Node 1 executes the matching ChanIn(LINK1IN,...) statement. Moreover, the task running on Node 2 goes to sleep as soon as it executes the ChanOut statement, and does not awake until the transfer has completed. This was implemented by INMOS to allow other tasks to run on a transputer while one is awaiting a transfer through a channel.

No buffering of information
The absence of buffering means that no memory is used to hold the message being transferred (or received) during the communication. The message is sent as is, from first byte to last byte. The side-effect of this is that when a task starts sending a message to another task, the receiving task does not receive a header telling it how many bytes are coming. The receiving task must know in advance how many bytes to expect. Unpredictable results will occur if too many or too few bytes are sent by a transmitting task. In such cases, the computation will not be successful.

In our example, this means that both tasks must know that integers are exchanged on each transfer. This is easily accomplish by passing the macro INTSIZE to all ChanIn and ChanOut call. But because each prime is sent as it is computed, and because Node 1 does not know a priori how many primes will be found by Node 2, some mechanism must be used to insure that both Node 1 and Node 2 reach the end of the computation. The solution is for Node 2 to pass a sentinel to Node 1 when it completes its computation.

Matching ChanIn and ChanOut
When transferring information, the number of bytes sent by ChanOut must match exactly the number of bytes expected by ChanIn. Let us consider the situation when Node 2 sends a prime to Node 1, but Node 1 is still in the process of printing the previous prime. The process on Node 2 enters an inactive state. When the process on Node 1 executes a ChanIn on the corresponding channel, it too goes into inactive mode. This is very important. Both tasks go into inactive state. The designers of the transputer (INMOS) implemented this feature so that other tasks could share the transputer while the communicating tasks were involved with data transfer. The data transfer is not under software control, but under the authority of the link controller in the transputer, and the tasks simply rest until the hardware has transferred the data. Finally, when the transfer is over, both tasks are awaken and resume execution. This is true here since each transputer executes only one task. Were either one of the transputers running multiple tasks, then it is possible that the sending or receiving task could have remained inactive for some time. But enough of this for now; we'll come back to this point later on when we deal with multitasking.

A slight relaxation of the Matching Rule when transfer takes place on hard channels is that it is not absolutely necessary to have one ChanIn statement for each ChanOut statement. It is possible to have a sending node execute two ChanOut statements transferring 50 bytes each to a receiving node which executes only one ChanIn statement for 100 bytes. As long as the total number of bytes sent matches the total number of bytes received, everything will work fine. The code segment below shows how Node 2 can send four characters separately to Node 1 which gets them as one integer (the transputer is a 32-bit processor, representing integers as 32 bits, or 4-bytes). Although this does not make much sense with characters and integers, this property will come in handy when we work with arrays, where arrays can be expected in full (only one ChanIn) but sent in small sections (several ChanOuts).

```
int x;
char a, b, c, d;

if (_node_number == 1)
{
        ...
        ChanIn(LINK1IN, &x, (int) sizeof(int));
```

```
                ...
        }
        if (_node_number == 2)
        {
                ...
                ChanOut(LINK0OUT,&a,1);
                ChanOut(LINK0OUT,&b,1);
                ChanOut(LINK0OUT,&c,1);
                ChanOut(LINK0OUT,&d,1);
                ...
        }
```

Alternation with ProcSkipAlt

Our first program works fine, but it really does not fit the description of a parallel program. Node 1 could have just as well computed the primes and printed them. Node 2 was not necessary to do that. However, it still provides us with a simple example to illustrate channel communication between nodes of a transputer network. Let's improve our program (and its performance) by having Node 1 carry out some of the computation. Our new problem is described by the following requirements:

•Node 1 will compute all the primes between 1 and 99, and will print them. It will also receive primes from Node 2, and will print them as well. We do not require the output to be a sorted.

•Node 2 will compute all the primes between 100 and 200 and will send them to Node 1.

Let's think about the first requirement. Node 1 now has two jobs:

1. It must compute and print primes.
2. It must receive primes from Node 2 and print them as well.

It would be unwise to code Node 1's task as two consecutive loops for the two jobs. If we do this, then we loose any performance benefit from the parallelization of our initial problem, since the two loops would serialize the computation. Node 1 would first compute and print all the primes between 1 and 99, and then it would start accepting the primes from Node 2 and print them. But because the ChanIn and ChanOut functions are unbuffered and blocking, Node 2 would get blocked the first time it tries to "chanout" its first prime, and wouldn't get released from this state until Node 1 would start receiving the primes. At this point Node 2 would have computed only one prime, and would still have to go through the rest of its numbers to find the remaining primes. This is the main restriction associated with channel communication on the transputer. We must remember that communication is blocking. If we are not careful we can miss on harvesting all the power provided by the multiprocessor.

What we need to do instead, is to mix the two loops that Node 1 must run, the one computing the primes, and the one relaying the primes from Node 2. Thus, we can code the task running on Node 1 using the following model:

```
/*--- Pseudo code running on Node 1 ---*/
for (x= 1; x< 100; x++)
{
    if (x is prime)
    {
        print it;
        increment counter;
    }
    if (Node 2 has a prime to send)
    {
        Get the prime from Node 2;
        if not the sentinel print it;
        increment counter;
    }
}
while (we haven't received the sentinel from Node 2)
{
    wait for a prime from Node 2;
    if not the sentinel print it;
    increment counter;
}
```

Testing when Node 2 has primes to send

The first for loop embeds two inner blocks, one computing the primes between 1 and 99, the other getting primes from Node 2. How should we implement the portion...

```
    if (Node 2 has a prime to send)
    {
        Get the prime from Node 2;
        if not the sentinel print it;
        increment counter;
    }
```

Should we use ChanIn(LINK1IN,...) to test if Node 2 has a prime to send. No. The reason is that ChanIn() is blocking, and if we use it in the test, the task running on Node 1 will be blocked until Node 2 finds a prime to send. This means that the for (x=1; x< 100; x++) loop would alternate between printing a prime found by Node 1 (less than 100) and a prime found by Node 2 (larger than 100). This is still too restrictive, if Node 2 takes a long time finding primes, then Node 1 is slowed down unnecessarily. We need a non-blocking mechanism to check if there is some activity on the channel connecting Node 2 to Node 1. **ProcSkipAlt** is the solution.

```
            int ProcSkipAlt(Channel *, ...,Channel *, 0)
```

ProcSkipAlt is an alternation function that receives a list of channel pointers terminated by a 0 (NULL), and that checks whether these channels are ready for input. Alternation refers to the fact that the action taken by the function, or its result is not deterministic, but rather depends on the list of elements past to the function. Alternatively, the result may depend on the first parameter, or the second, or the fourth, or none of them, and this cannot be predicted in general. With ProcSkipAlt, if a channel in the list is ready for input, an index corresponding to the position of the channel in that list is returned. If none of the channels are ready, then -1 is returned.

Examples
Assume that we have a transputer with four active links, and assume that the neighbor it is attached to via LINK1 has initiated a transfer to this transputer. The following calls show the value returned by ProcSkipAlt in that case.

```
ProcSkipAlt(LINK0IN, LINK1IN, LINK2IN, 0);      /* returns 1  */
ProcSkipAlt(LINK0IN, 0);                        /* returns -1 */
ProcSkipAlt(LINK1IN, LINK2IN, LINK0IN, 0);      /* returns 0   */
```

We can now translate the pseudo code for Node 2 as follows:

```
if (ProcSkipAlt(LINK1IN,0) != -1)
{
        ChanIn(LINK1IN, &x, INTSIZE);
        printf("%8d",x);
        NoPrimes++;
}
```

If Node 2 hasn't initiated a transfer yet, then ProcSkipAlt returns -1 and the loop is skipped. Node 1 can then test another integer for its "prime" characteristic. The code for the whole program is now shown in its entirety.

```
#include <stdio.h>
#include <stdlib.h>
#include <conc.h>                          /* transputer library */

/* =========================== DEFINITIONS =========================== */
#define INTSIZE   ((int) sizeof(int))
#define INTERVAL1 100
#define INTERVAL2 200
#define SENTINEL  -1

main()
{
   int NoPrimes = 0, x, y = 0, j;

   if (_node_number==1)
   {
      for (x = 1; x<INTERVAL1; x++)
      {
         if (IsPrime(x))
         {
            printf("%8d", x);
            NoPrimes++;
         }

         /*--- check if Node 2 has a prime ready ---*/
         if (ProcSkipAlt(LINK1IN, 0)!=-1)
         {
            ChanIn(LINK1IN, &y, INTSIZE);
            if (y!=SENTINEL)
            {
               printf("%8d", y);
               NoPrimes++;
            }
         }

      }

      /*--- Keep relaying primes if Node 2 isn't done yet ---*/
      while (y!=SENTINEL)
      {
         ChanIn(LINK1IN,  &y, INTSIZE);
         if (y!=SENTINEL)
         {
                printf("%8d", y);
                NoPrimes++;
            }
        }
        printf("\nReceived %d primes \n", NoPrimes);
        exit(0);
    }
    else
    {
        /*--- scan interval ---*/
        for (x = INTERVAL1; x<INTERVAL2; x++)
        {
            if (IsPrime(x))
                ChanOut(LINK0OUT,  &x, INTSIZE);
        }

        /*--- signal Node 1 that we are done ---*/
        x = SENTINEL;
        ChanOut(LINK0OUT,  &x, INTSIZE);
    }
}

int IsPrime(int x)
{
    int i;              /*  0 1 2 3 4 5 6 7 8 9 */
    static int SmallPrimes[10] = {0,0,1,1,0,1,0,1,0,0};
    if (x<10) return SmallPrimes[x];
    if (x%2==0) return 0;
    if (x%3==0) return 0;
    if (x%5==0) return 0;
    for (i = 2; i*i<=x; i++)
        if (x%i==0) return 0;
    return 1;
}
```

The code executed by Node 2 is the same as in our first program, except that Node 2 computes primes between 100 and 200, coded as the macros INTERVAL1 and INTERVAL2, respectively.

Let's look at the task executed by Node 1. The main for-loop computes the primes and relays those sent by Node 2. Note that we are using the temporary integer variable x in the prime-generating loop and y in the relay loop. By having two variables instead of one, we can easily "remember" whether we received the sentinel from Node 2. Note that this is not the most efficient way of writing the code for Node 1. Because Node 1 does not check if Node 2 has a prime ready for output inside its j-loop, it is slightly slowing Node 2 down.

Program output

```
101      2     103      3     107     109      5     113     127
7     131     137     139     149      11     151     157      13
163     167     173     179      17     181     191      19     193
197     199      23      29      31      37      41      43      47
53      59      61      67      71      73      79      83      89
97
Received 46 primes
```

Observe that Node 2 is turning out primes very fast, compared to Node 1. Remember, Node 1 is the one outputting the primes less than 100, while Node 2 computes the primes greater than 100. In fact, when Node 2 has finished testing all its 100 numbers (when 199 is output), Node 1 still hasn't advanced past a third of its 100 numbers (it hasn't outputted 23 yet). The reason is that Node 2 executes a highly specialized task and spends all its time testing numbers and channeling them out, while Node 1 is involved with testing numbers, channeling numbers in from Node 2, and channeling numbers out to the host through printf statements, not mentioning updating a counter! The net result is that Node 1 is about three times slower than Node 2.

Other alternation functions

The conc.h library contains two other important alternation functions: ProcAlt, and ProcTimerAlt.
int ProcAlt(Channel *, ..., Channel *, 0)
int ProcTimerAlt(int, Channel *, ..., Channel *, 0)

ProcAlt is similar to ProcSkipAlt, except that it is blocking, i.e. it waits until one of the channels specified in the list becomes active, and then returns to its caller. We will look at an example using ProcAlt next. ProcTimerAlt, like ProcAlt, is blocking. However, it has a time-out feature which enables it to unblock when a specified time is reached. If you remember our discussion in Chapter 2, each transputer maintains two timers, and ProcTimerAlt allows us to enter a blocking ProcAlt call, with the option to return if too much time (defined by the user) has elapsed. This is very useful for debugging purposes, and in environments requiring high-reliability.

Adding a relay node

Finally, let us consider a function that is omnipresent in any message-passing multiprocessor system: That of relaying information, or messages, from neighbors on one side of the network to neighbors on another side. Relaying information is important in a message-passing environment. In all networks only one transputer in the network is attached to the host, and the initial distribution of data or the collection of results will require nodes deep in the network to exchange information with the root transputer attached to the host. There is no choice but for this information to be relayed by other nodes. To illustrate this concept, we use our same prime-finding algorithm, but this time Node 1 and 3 are computing the primes, while Node 2 acts as a relay between them.

main function

Using our previous top-down approach, we can organize main as a switch starting one of three possible tasks.

```
main()
{
    switch (_node_number)
    {
        case 1: Task1();
              exit(0);
        case 2: Task2();
              break;
        case 3: Task3();
    }
}
```

The top-down approach here will allow us to modularize our program. We are still using the farm programming approach, and all three transputers receive the same copy of the program. Different sections of the program are dedicated to different transputers.

We start our analysis of the program with the farthest node, Node 3, which executes Task 3.

Task3()
Node 3 does exactly what Node 2 was doing in our previous examples. Task3 is hence a simple encapsulation of the code we saw before:

```
void Task3(void)
{
    int x, j;

    /*--- scan interval ---*/
    for (x = INTERVAL1; x<INTERVAL2; x++)
    {
        /*--- if prime then send number to Node 1 ---*/
        if (IsPrime(x))
            ChanOut(LINK0OUT, &x, INTSIZE);
    }

    /*--- signal Node 2 and Node 1 that we are done ---*/
    x = SENTINEL;
    ChanOut(LINK0OUT, &x, INTSIZE);
}
```

Task3 does not store the primes internally but sends them as soon as they are found. It thus blocks every times it has found a new prime. Task3 sends its prime to Task2 running on Node 2 over LINK0OUT. Task2 receives the numbers on its LINK1IN channel.

Task2()
The job of Task2 is to get integer primes sent by Node 3 and to relay them to Node 1. Hence, each ChanIn operation is associated to a ChanOut operation. A first (hasty) solution would be to understand this as "each ChanIn operation is followed by a ChanOut operation", and code the task as follows:

```
/*--- hasty implementation of Task2's inner loop ---*/
do
{
    ChanIn(LINK1IN, (char *) &x, INTSIZE);
    ChanOut(LINK0OUT, (char *) &x, INTSIZE);
} while (x != SENTINEL);
```

This implementation of Task2 would work, but it would be inefficient. The reason is that Node 3 could (and does) generate primes faster than Node 1 can accept them, and because Node 2 uses a loop containing two blocking functions for receiving and sending information between the two remote tasks, it runs only as fast as the slowest of its two counterparts, which in our case is Task1. So this type of relay is inefficient in that it does not decouple the input from the output. The slow task (Task1) slows down the faster one (Task3) through the relaying implemented by Task2.

Using a buffer
The solution is to introduce a buffer in Node 2 to decouple the two channels. A queue is an ideal candidate to implement the buffer. The algorithm is simple

```
while (not done)
{
    wait for either Node 1 or Node 3 to request a transfer;
    if (Node 1 is ready)
    {
        Dequeue a prime;
        Send it to Node 1;
        If the integer sent was the sentinel we're done;
    }
    if (Node 3 is ready)
    {
        Get a prime;
        Enqueue it;
    }

}
```

The outer loop repeats until the SENTINEL is received from Node 3 and passed on to Node 1. Task2 can either send a prime to Task1, or receive a prime from Task3. It must thus monitor both, so that it can respond to the first one requesting a transfer. This is the function of ProcAlt which is given a list of channels to monitor. It would be tempting to have Task2 do this:

```
ProcAlt(LINK0OUT, LINK1IN, NULL);
```

But this is wrong. The reason is that ProcAlt can only monitor input channels. Therefore, we must use a trick here, and make Task1 send Task2 a short message whenever it is ready to receive a prime from Task2. Because we do not want Task1 to have to wait on Task2 if its queue is empty, Task2 will send "pad" integers (fake primes) back if it cannot send a real prime. This way Task1 will always be guarantied to receive something within a short period of time, so that it is not slowed down too much.

```
void Task2(void)
{
    QueueType Q
    int x;
    int Done = 0;
    char dummy;            /* used to get "ready" signal from Node 1 */

    while (!Done)
    {
        /*--- find out which node is ready first (give priority to
        Node 3, as Node 1 will send many signals ---*/




        index = ProcAlt(LINK1IN, LINK0IN, 0);
        switch (index)
        {
            case  0 : /* LINK1IN from Node 3 */
                    ChanIn(LINK1IN, &x, INTSIZE);
                    Enqueue(&Q,x);
                    break;
            case  1 : /* LINK0IN from Node 1 */
                    ChanIn(LINK0IN, &dummy, 1);
                    if (Empty(&Q))
                        x = PAD;  /* tell Node 1 we have nothing */
                    else
```

```
                        x = Dequeue(&Q);
                    /*--- send integer or pad ---*/
                    ChanOut(LINK0OUT, &x, INTSIZE);
                    Done = (x==SENTINEL);
                    break;
        }
    }
}
```

QueueType, Enqueue(), Dequeue(), and Empty() define a queue data-type and some of its standard operators.

Task1()
The code executed by Node 1 is almost the same, with one exception. In the previous 2-node version of this program, Node 2 was blocking on a ChanOut every time it found a prime. It was easy for Node 1 to detect such a condition with a ProcSkipAlt. In our new 3-node version, however, Node 2 is not blocking anymore when it has a prime in its queue. Instead, it waits for either Task1 or Task2 to show some activity on the channels. Therefore, we cannot keep the ProcSkipAlt statement in Task1, otherwise Task1 and Task2 would never talk to each other! Take a second to see why. If none of them commit to block on the channel connecting them, then no communication can occur.

```
void Task1(void)
{
    int x, j, y = 0, NoPrimes = 0;
    char dummy;           /* used to tell Node 2 "ready to receive" */

    /*--- Compute, print, and relay primes ---*/
    for (x = 1; x<INTERVAL1; x++)
    {
        /*--- if x is prime then print it ---*/
        if (IsPrime(x))
        {
            printf("%8d", x);
            NoPrimes++;
        }
        /*--- if Node 2 already done, continue ---*/
        if (y==SENTINEL) continue;

        /*--- otherwise tell it we're ready to receive ---*/

        ChanOut(LINK1OUT, &dummy, 1);/* send a dummy byte */

        ChanIn(LINK1IN, &y, INTSIZE);
        if ((y!=SENTINEL) && (y!=PAD))
        {
            printf("%8d", y);
            NoPrimes++;
        }
    }

    /*--- Keep relaying primes if Node 2 isn't done yet ---*/
    while ((y!=SENTINEL))
    {
        ChanOut(LINK1OUT, &dummy, 1);
        ChanIn(LINK1IN, &y, INTSIZE);
        if ((y!=SENTINEL) && (y!=PAD))
        {
            printf("%8d", y);
            NoPrimes++;
        }
    }
    printf("\nReceived %d primes\n", NoPrimes);
}
```

Notice that every time Node 1 is ready to receive a prime from Node 2, it sends the char variable dummy to indicate its readiness. Because Node 2 may not have a prime in its queue, it may return a pad integer, and Task1 must test for this condition, and discard them if necessary. The second point of interest is that in the final while loop, Task1 must still use the sending of dummy since Node 2 can only send a prime when it receives this signal from Node 1. Because Task1 in that case would start sending a lot of dummy bytes, Task2 uses LINK1IN first in its list of arguments in ProcAlt, so that it will choose to receive a prime from Node 3 before receiving a dummy signal from Node 1.

Multitasking on the same Transputer

A Simple Example

Consider a program that runs on the root transputer and creates two tasks. Both tasks share the same transputer in round-robin fashion. The two tasks are completely independent, and do not communicate with each other. Task2 computes the 64th term of the fibonacci series, and Task2 converts the characters of an input string to uppercase.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "conc.h"

#define WS_STACK_SIZE 1024

void Task1(Process *, char *);
void Task2(Process *, int *x);

main(void)
{
     Process *Task1P, *Task2P;
     char string[] = "This is the test string Task1 is working on...";
     int  FibNumber = 64;

     /*--- allocate storage for the processes ---*/
     Task1P = ProcAlloc(Task1, WS_STACK_SIZE, (int) sizeof(char *),
                  string);
     Task2P = ProcAlloc(Task2, WS_STACK_SIZE,
                  (int)sizeof(int*),&FibNumber);

     /*--- verify that storage could be allocated ---*/
     if ((Task1P==NULL) || (Task2P==NULL))
     {
          printf("Error allocating processes\n");
          exit(1);
     }

     /*--- start the two tasks ---*/
     ProcPar(Task1P, Task2P, 0);
     printf("Tasks done\nTask1's result: %s\nTask2's result: %d\n\n",
          string, FibNumber);
     ProcFree(Task1P);
     ProcFree(Task2P);
     exit(0);
}

void Task1(Process *p, char *string)
{
     char *q = string;

     while (*q = toupper(*q)) q++;
}

void Task2(Process *p, int *x)
{
     int i, temp = 1, fibn = 1, fibn_1 = 1;

     for (i = 3; i<=*x; i++)
     {
          temp = fibn_1+fibn;
          fibn_1 = fibn;
          fibn = temp;
     }

     *x = fibn;
}
```

Task1P and Task2P are pointers to processes, which is the term used by Logical Systems to identify a task. The type Process is defined in "conc.h". The pointers point to an area of memory that is private to each class. This area contains vital process information such as the execution stack. Each task Task1 and Task2 are separate functions, defined exactly lke regular functions, except that the first arguement is a process pointer. This pointer is simply declared and is never accessed or intialized by the programmer.

ProcAlloc
ProcAlloc is used to dynamically allocate memory for the process or, the task. The parameters are in the following order :

- The name of the function containing the code to be executed by the task,
- The stack size in bytes
- The number of words to hold parameters in the stack
- A list of the parameters that the function receives (excluding the process pointer parameter of the function)

ProcPar
ProcPar takes a null terminated list of process pointers and launches the execution of their associated functions. ProcPar blocks until all the processes it launched terminate. Hence in our case main would also be a parallel task.

ProcParList
The following code shows a main function with the individual task pointers replaced by an array of process pointers. This is useful when maintaining a large number of processes.

```
main(void)
{

     Process *TaskArray[3];
     char string[] = "This is the test string Task1 is working on...";
     int  FibNumber = 64;

     /*--- allocate storage for the processes ---*/
     TaskArray[0] = ProcAlloc(Task1, WS_SIZE, (int) sizeof(char *),
                         string);
     TaskArray[1] = ProcAlloc(Task2, WS_SIZE, (int) sizeof(int *),
                         &FibNumber);
     TaskArray[2] = NULL;

     /*--- verify that storage could be allocated ---*/
     if ((TaskArray[0]==NULL) || (TaskArray[1]==NULL))
     {
          printf("Error allocating processes\n");
          exit(1);
     }

     /*--- start the two tasks ---*/
     ProcParList(TaskArray);
     printf("Tasks done\nTask1's result: %s\nTask2's result: %d\n\n",
          string, FibNumber);
     ProcFree(TaskArray[0]);
     ProcFree(TaskArray[1]);

     exit(0);
}
```

ProcFree
The memory allocated by the ProcAlloc function is returned to the heap.

Unsynchronized Exection of Multitasks
Consider a situation where processes are to be launched, but the parent process (the process launching the processes) does not want to be blocked. An example of such a situation would be to create tasks that receive messages from communication channels and routes them to other nodes. In such situations, ProcRun is used instead of ProcPar.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "conc.h"

#define WS_SIZE 1024
```

```
void Task1(Process *, char *, int *);
void Task2(Process *, int *x, int *);

main(void)
{
     Process *Task1P, *Task2P;
     char string[] = "This is the test string Task1 is working on...";
     int  FibNumber = 64;
     int  Task1Flag = 0, Task2Flag = 0;

     /*--- allocate storage for the processes ---*/
     Task1P = ProcAlloc(Task1, WS_SIZE, 2, string, &Task1Flag);
     Task2P = ProcAlloc(Task2, WS_SIZE, 2, &FibNumber, &Task2Flag);

     /*--- verify that storage could be allocated ---*/
     if ((Task1P==NULL) || (Task2P==NULL))
     {
          printf("Error allocating processes\n");
          exit(1);
     }

     /*--- start the two tasks ---*/
     ProcRun(Task2P);
     ProcRun(Task1P);
     while ((Task1Flag==0) || (Task2Flag==0))
          /* wait */;

     if (Task1Flag)
          printf("Task1 finished first\n");
     else
          printf("Task2 finished first\n");

     while (Task1Flag*Task2Flag==0)
          /* wait */;

     printf("Both tasks done\nTask1's result: %s\nTask2's result:
            %d\n\n", string, FibNumber);
     ProcFree(Task1P);
     ProcFree(Task2P);
     exit(0);
}

void Task1(Process *p, char *string, int *Task1Flag)
{
     char *q = string;

     while (*q = toupper(*q)) q++;
     *Task1Flag = 1;
}

void Task2(Process *p, int *x, int *Task2Flag)
{
     int i, temp = 1, fibn = 1, fibn_1 = 1;

     for (i = 3; i<=*x; i++)
     {
          temp = fibn_1+fibn;
          fibn_1 = fibn;
          fibn = temp;
```

```
    }

    *x = fibn;
    *Task2Flag = 1;
}
```

Communications Between Multitasks

Several options for communicating between multitasks on the same transputers exist.

- Shared Variables
- Data locked within a semaphore
- Message passing through soft channels

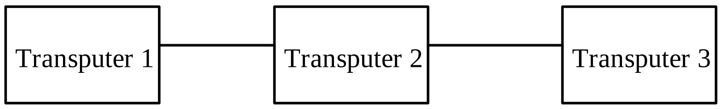
Message passing using soft channels is natural of the transputer. Soft channels emulate channels in memory. The operation is basically a byte move operation from one part of the memory to another. The functions used for soft channel communication remain the same as the hard channel communication.

```
/* Global Area.... */
.
.
Channel  *softChannel;
.
.
softChannel = ChanAlloc();
.
.

/* Task1 */
.
.
.
char buffer[50];
while(!Done)
 {
        .
        .
        ChanOut(softChannel, (char*) buffer, 50*sizeof(char));
        .
        .
 }
.
.

/* Task 2 */
.
.
.
char buffer[50];
while(!Done)
 {
        .
        .
        ChanIn(softChannel, (char*) buffer, 50*sizeof(char));
```

```
            .
            .
 }
.
.
```

Virtual Channels

Virtual channels as the name indicates, are channels that can be used in the same way as hard and soft channels. The main difference from hard channels, and their power is their ability to connect transputers which are not neighbors. A transputer may have any number (limited by the memory) of virtual channels. This is possible because Logical Systems introduces an extra layer of software in bewteen the hardware and the application code. At this level, each node runs a group of high priority tasks whose job is to route messages exchanged over virtual channels. This is however invisible to the programmer.



Consider the above network of transputers. If a task on transputer 1 needed to communicate with a node on transputer 3, it would have to pass through transputer 2. Using virtual channels however the pass through transputer 2 is made transparent.

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

main()
{
        if(node_number == 1)
        {
                Channel *vc13;
                int number = 1234;
                vc13 = VChan(2);
                VChanOutInt(vc13, number)
        }
        else if(node_number == 2)
        {
                .
                .
                .
        }
        else if(node_number == 3)
        {
                Channel *vc31;
                int temp;
                vc31 = VChan(3);
                temp = VChanInInt(vc31)
        }
```

In order for this program to function properly, the information regarding the routing of virtual channels must be contained in a ".nif" file. The file for our example would be as follows:

```
.
.
vchan;
.
.
.
1[2],3[3];
.
.
.
```

The above declararions indicate that the program uses virtual channels, and that the virtual channel on transputer 1 defined by the handle 2 is connected to the virtual channel on transputer 3 defined by handle 3.

Summary

Communications between transputers is blocking. When a task is blocked, it becomes inactive. The task becomes active again inly when the transmission is over and no other tasks are ahead in the active queue. ChanIn and ChanOut are blocking functions that can be used when communication with only one remote task is necessary. When communications over several channels is required, the blocking function ProcAlt may be used. However this function can only monitor incoming channels. A transfer between two transputers can occur only when the tasks on both transputers are committed to the transfer by calling ChanIn or ChanOut.

Creating concurrent tasks on a transputer requires several elements :

- The code of each task be encapsulated in a function
- A pointer to the process representing the task be initialized
- A list of parameters for each process
- A private execution stack for each process

ProcPar introduces automatic synchronization of the tasks it launches by blocking until all the tasks it launched terminate. ProcRun launches the tasks which run unsynchronized. Concurrent tasks on the same transputer can exchange information through the use of shared variables when there is one producer and several consumers, or by using semaphores when more than one task can modify the data, or by message passing through soft channels. The same functions may be used for hard and soft channels. Virtual channels provide the benefits of creating virtual networks independent of the actual physical network layout of the transputers, and also allow each processor to carry out concurrent I/O with the host. However, virtual channels add overhead and eat up some of the processor cycles, and hence may cause some performance degradation in some cases.