THESIS

DYNAMIC RECONFIGURATION AND
LINK FAULT TOLERANCE
IN A TRANSPUTER NETWORK

by

Winfred Prescott Pikelis

June 1989

Thesis Advisor: Uno R. Kodres

Approved for public release; distribution unlimited
Dynamic Reconfiguration and Link Fault Tolerance in a Transputer Network

Winfred Prescott Pikelis

Master's Thesis

From June 1989 To

Supplementary Notation: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

19 Abstract (continue on reverse if necessary and identify by block number)

This thesis explores dynamic reconfiguration and link fault tolerance in a Transputer network using software controlled crossbars. A message exchange system was designed, implemented and evaluated to facilitate various aspects of dynamic interconnectivity between processing nodes, as well as detection and recovery from failed network links without loss of data. As implemented, the message exchange can be embedded with application code which can direct network topology to facilitate code execution.
Dynamic Reconfiguration and Link Fault Tolerance
In a Transputer Network
by
Winfred Prescott Pikelis
Lieutenant, United States Navy
B.S., University of Nebraska, 1980
Submitted in partial fulfillment of the requirements
for the degree of
MASTER OF SCIENCE IN COMPUTER SCIENCE
from the
NAVAL POSTGRADUATE SCHOOL
June 1989

Author:  
Winfred P. Pikelis

Approved by:  
Uno R. Kodres, Thesis Advisor
Gregory D. Buzzard, Second Reader
Robert B. McGhee,
Chairman, Department of Computer Science
Kneale T. Marshall,
Dean of Information and Policy Sciences
ABSTRACT

This thesis explores dynamic reconfiguration and link fault tolerance in a Transputer network using software controlled crossbars. A message exchange system was designed, implemented and evaluated to facilitate testing various aspects of dynamic interconnectivity between processing nodes as well as detection and recovery from failed network links without loss of data. As implemented, the message exchange can be embedded with application code which can direct network topology to facilitate code execution.
DISCLAIMER AND ACKNOWLEDGMENT

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made within the time available to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

Many terms used in this thesis are registered trademarks of commercial products. Rather than attempting to cite each individual occurrence of a trademark, all registered trademarks appearing in this are listed below the firm holding the trademark.

INMOS Limited, Bristol, United Kingdom:

- Transputer
- IMS T800
- IMS T414
- IMS T212
- Transputer Development System

- Occam
- IMS B012
- IMS B004
- IMS C004
- IMS B401 TRAM
# TABLE OF CONTENTS

## I. INTRODUCTION

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. BACKGROUND</td>
<td>1</td>
</tr>
<tr>
<td>1. Single High Speed Processor</td>
<td>1</td>
</tr>
<tr>
<td>2. Multiple Processors Using A Shared Bus</td>
<td>1</td>
</tr>
<tr>
<td>3. Multiple Processors Using Point-To-Point Communications</td>
<td>2</td>
</tr>
<tr>
<td>B. TRANSPUTER TECHNOLOGY</td>
<td>3</td>
</tr>
<tr>
<td>C. MOTIVATION</td>
<td>4</td>
</tr>
<tr>
<td>D. OBJECTIVES</td>
<td>5</td>
</tr>
<tr>
<td>E. THESIS ORGANIZATION</td>
<td>6</td>
</tr>
</tbody>
</table>

## II. THE TRANSPUTER

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. COMMUNICATING SEQUENTIAL PROCESSES</td>
<td>7</td>
</tr>
<tr>
<td>B. PROCESSOR ARCHITECTURE</td>
<td>8</td>
</tr>
<tr>
<td>1. Central Processing Unit (CPU) and Sequential Operation</td>
<td>8</td>
</tr>
<tr>
<td>2. Link Engines and Communications</td>
<td>12</td>
</tr>
<tr>
<td>3. Memory</td>
<td>15</td>
</tr>
<tr>
<td>4. Timers</td>
<td>16</td>
</tr>
<tr>
<td>5. External Event Input</td>
<td>17</td>
</tr>
<tr>
<td>6. Floating Point Unit (FPU)</td>
<td>17</td>
</tr>
<tr>
<td>7. Benchmarks Performance Comparisons</td>
<td>18</td>
</tr>
</tbody>
</table>
IV. MESSAGE EXCHANGE FUNCTIONAL DESCRIPTION .......... 50

A. OVERVIEW .......................................... 50

B. SYSTEM OPERATIONS ................................. 52
   1. Link Control Command Description ............... 52
   2. Link Failure And Recovery Procedures .......... 52
   3. Link Control Command Action Summary .......... 55

C. CONTROLLER MODULE ................................ 57
   1. Subprocedure: reset.or.nil .................... 57
   2. Subprocedure: c4.cmd .......................... 58
   3. Subprocedure: get.current.tie ................ 59
   4. Subprocedure: make.conn ....................... 60
   5. Subprocedure: break.conn ....................... 63
   6. Main Controller Code ............................ 64

D. TRAM MODULE ....................................... 69
   1. Manager Process ............................... 70
   2. Agent Process ................................ 74
   3. User Process .................................. 75
   4. TRAM Code Configuration ....................... 76
   5. Global Configuration ............................ 77

V. MESSAGE EXCHANGE PERFORMANCE EVALUATION .......... 80

A. DATA TRANSFER VARIATIONS AND SYSTEM EFFICIENCY .. 81
   1. Data Block Size .............................. 81
   2. Data Routing Via Multiple Crossbars ............ 88

B. TRAM CODE STRUCTURE .............................. 91

C. LINK FAULT RECOVERY PERFORMANCE .................. 93
LIST OF FIGURES

2.1 Transputer Architecture Block Diagram ................. 9
2.2 Transputer Processor Block Diagram .................... 10
2.3 Instruction Format ..................................... 11
2.4 Hardware Communication Link Block Diagram ........... 13
2.5 Serial Communication Link Protocol .................... 14
2.6 T800 Transputer Memory Map ......................... 15
2.7 Process Workspace ..................................... 20
2.8 Linked Process List .................................... 21
2.9 Single Element Buffer .................................. 25
2.10 Two Element Buffer ................................... 26
2.11 Single And Multiprocessor Equivalence ............... 30

3.1 IMS C004 Block Diagram .............................. 33
3.2 IMS C004 Hardware Implementation ................... 35
3.3 IMS E012 Slot And Edge Connector Arrangement ...... 38
3.4 B012 Configuration Variations
    Via P2 and K1 Jumpers ................................ 39
3.5 Link Organization Between
    B012 Slots and C004 Crossbars ....................... 41
3.6 B012 Edge Connectors P1 And P2 ...................... 42
3.7 Transputer Control Hierachy ........................... 43
3.8 T212 C004 Controller Link Organization ............. 44
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>TRANSPUTER PERFORMANCE BENCHMARK COMPARISONS</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>C004 CROSSBAR CONTROL COMMANDS</td>
<td>37</td>
</tr>
<tr>
<td>4.1</td>
<td>LINK CONTROL COMMANDS AND PROGRAM MODULE ACTIONS</td>
<td>56</td>
</tr>
<tr>
<td>4.2</td>
<td>JUMPER PLACEMENT FOR C004 TO C004 CONNECTIONS</td>
<td>62</td>
</tr>
<tr>
<td>5.1</td>
<td>SYSTEM PERFORMANCE DATA: UNLOADED, NO FAULTS</td>
<td>83</td>
</tr>
<tr>
<td>5.2</td>
<td>SYSTEM PERFORMANCE DATA: UNLOADED, WITH FAULTS</td>
<td>83</td>
</tr>
<tr>
<td>5.3</td>
<td>SYSTEM PERFORMANCE DATA: LOADED, NO FAULTS</td>
<td>84</td>
</tr>
<tr>
<td>5.4</td>
<td>SYSTEM PERFORMANCE DATA: LOADED, WITH FAULTS</td>
<td>84</td>
</tr>
<tr>
<td>5.5</td>
<td>LINK ROUTING ANALYSIS RESULTS</td>
<td>90</td>
</tr>
<tr>
<td>5.6</td>
<td>PERFORMANCE ASPECTS OF FAULT DETECTION AND RECOVERY</td>
<td>95</td>
</tr>
</tbody>
</table>
I. INTRODUCTION

A. BACKGROUND

Computer technologies continue to evolve along a variety of paths, each with an accompanying set of advantages and disadvantages. Typically, the primary design goal in the development of new, superior computer is to achieve processing performance beyond what is currently available in a given cost range. The speed with which the assigned task is accomplished is a key measure of a computer’s ability.

1. Single High Speed Processor

One path toward this goal concentrates efforts on improving the abilities of an individual processor: the device at the heart of most computers. Vast improvements have been realized in improving the processor architecture and shrinking the dimensions of the components to reduce the time needed to physically move control and data signals from one point to another within the processor. Even electrons moving at near the speed of light require a finite time to travel finite distances, thus imposing an upper bound upon maximum obtainable performance. More improvements in this approach will no doubt occur, however, they will be realized with ever greater expense for each small increase of processing speed.

2. Multiple Processors Using A Shared Bus

Another approach involves dividing the task among multiple processors, each assigned a portion of the task. This allows processing of different pieces of the task to occur simultaneously or in parallel. Ideally, the time required to complete a given task that is divided in such a manner should decrease proportionally with the number of processors assigned. That is, doubling the number of processors should result in reducing
the time required to complete a given task by half. In practice, however, the addition of processors does not result in linear performance improvements since each task will always contain some portion which must be executed sequentially, thus voiding the parallel processing capability available. Amdahl [Ref. 1] explained that this occurrence places an upper limit on potential gains made in parallel processing, regardless of the number of processors used. This paper assumes, however, that a given task can be divided sufficiently to warrant a large degree of parallel processing.

Another obstacle exists preventing parallel machines from achieving maximum potential performance gains. Processors working in parallel will need to share data as the processing proceeds. This sharing of resources among many processors leads again to the problem of stretching the capabilities of individual components. In this situation, the common data path and common memory storage devices will become a performance restriction as the number of processors involved is increased [Ref. 2:p. 5]. Greater advances in processor technologies over bus and backplane technologies further exacerbates this problem, since by adding faster processors to a given balanced multiprocessor bus, the bus will become overloaded and hence slow the entire system.

3. **Multiple Processors Using Point-To-Point Communications**

To avoid the potential bottlenecks associated with having all processors sharing a common communication resource, processors can communicate more directly using distributed connections, that is, each processor has its own finite set of independent connections. As the number of nodes in a network grows, however, a point is reached in which all pairs of processors will no longer be directly connected. The transport mechanisms used for data communications and which must deal with these problems can be classified into two broad categories: packet switching and circuit switching.
a. Packet Switching

In a large network where all points cannot directly communicate, the connection must be completed through intermediaries. This is known as packet switching or packet routing. Data is carved into blocks of convenient size and passed with routing information from node to node until the desired destination is reached. Packet routing is used extensively in hypercube architecture computers which are a network of processors symmetrically interconnected so as to minimize the distance between any two nodes. Although this method is much less restrictive than a shared bus, some overhead is still incurred as processing time is dedicated to the task of passing data.

b. Circuit Switching

In order to avoid the need for passing data along intermediate nodes, additional hardware can be used to assign specific temporary paths between processors wishing to communicate. This is known as circuit switching and exists quite commonly in the form of a telephone exchange. When one party dials another, a connection is made and dedicated for that specific communication for the duration of the call. When the call is complete, the connection is broken and may be used for a new connection. This clearly avoids the overhead of intermediate nodes passing data, however, new overhead is created in the task of managing the available connections. In circuit switching, the connection must be made and broken for each communication, and the event of "busy" connections must be handled.

B. TRANSPUTER TECHNOLOGY

Another technology has evolved which attempts to combine the advantages of the examples mentioned above while hoping to avoid the difficulties. A device called the Transputer combines a processor with its own local memory storage and four specialized data paths or links designed to communicate directly with other Transputers. Thus, as the
number of processors and processing power in the network grows, the amount of memory and links available also expand. By providing its own direct communications path, the problems of bus contention are largely overcome. However, every Transputer in a network larger than five nodes cannot communicate directly with each other since each Transputer contains only four hardware links.

Although the number of communication paths grows as Transputers are added to the network, establishing a dedicated path directly between any two processors will not always be possible. To allow communications between any two nodes, the message must either be passed via intermediate nodes (packet routing) or additional hardware must be employed to connect a dedicated path (circuit switching). Because of the synchronous nature of communication in Transputers, packet routing may introduce deadlock if it is not implemented carefully [Ref. 3:p. 110].

As a further complication, growth in the number of links between Transputers increases the probability of communications failure. When communicating via links a failure of the medium causes the link engines involved to wait forever which will likely result in failure of the processing in progress. Even without link failures or synchronization limitations, packet routing systems incur a potentially significant overhead in passing data between nodes.

C. MOTIVATION

Modern weapons systems depend heavily upon the availability of powerful processors to monitor equipment, evaluate sensor inputs, display command and control information and calculate trajectories and weapons intercepts. Weapons systems control computers can be described by following characteristics [Ref. 4:p. 11]:

- Physically distributed: specialized computers are located throughout the platform, each dedicated to specific tasks but also in communication with the others.
• Fault tolerant: loss of some computers or some communications paths should not bring the entire integrated system to a halt,

• High performance: to handle demands of sophisticated sensors, and

• Flexible and extensible: to respond quickly to a changing threat environment.

The AEGIS Modelling Laboratory at the Naval Postgraduate School considers the Transputer’s architecture and capabilities as potentially useful to fulfill the greater demands of future weapons systems [Ref 4:p. 11]. Although parallel processors in general, and Transputers in particular can satisfy the above requirements, communications between processors may still present unacceptable restrictions either in ability to link any two Transputers in the network or in delays in obtaining the connection.

D. OBJECTIVES

According to Lauwereins [Ref. 5:p. 223], "...The two major problems encountered when interconnecting cooperating microcomputers are the detection of parallelism in the application program and the overload of communication lines." No attempt is made here to develop methods for detecting parallelism. Instead, this thesis explores dynamic reconfiguration and link fault tolerance in a network of Transputers as a method for conducting concurrent processing without prohibitive control overhead and thereby overcoming the difficulties noted above.

The message exchange is a combination of procedures and "off-the-shelf" hardware allowing direct communication between any two processors in the network. In the message exchange, specific connections are dynamically requested, created and terminated as required by the application code with the use of program controlled switches. Thus, its development served as a test vehicle in developing routines for software controlled link switches and link fault tolerance. The message exchange concept is in contrast to packet routing architectures in which all data as well as control signals are passed via intermediate nodes to reach non-adjacent destination nodes.
Communication fault detection and recovery procedures are implemented to increase the reliability of the links and to ensure that requested connections are eventually completed when link resources become available. With a functional message exchange system, a programmer can insert specific task instructions within this structure and have program controlled access to all processors in the network without additional code to manage the communications.

E. THESIS ORGANIZATION

Chapter II describes in detail the various tools used, including Transputer architecture and operation, the principles upon which it is based, and fundamentals of the programming language employed.

Chapter III describes the specific equipment upon which the message exchange was implemented.

Chapter IV explains the code and structures of the message exchange program. This includes explanations of the various modules of the program in both normal operation and in the event of communications failure.

Chapter V discusses the testing and evaluation of the message exchange and the aspects of program structure and communications load which affect performance.

Chapter VI presents the conclusions reached as a result of development, using and evaluating the message exchange. Recommendations for further research concerning Transputers, integrated weapons systems and this project is also included.
II. THE TRANSPUTER

A. COMMUNICATING SEQUENTIAL PROCESSES

Transputer is a combination of the words transistor and computer to emphasize its intended purpose: a single-chip processor to be used as a fundamental building block in large collections of parallel processors in much the same way that transistors are basic components of more complex and capable devices [Ref. 6:p. 25]. By using a collection of Transputers, large parallel systems can be constructed which operates concurrently and communicates through links. The Transputer was developed by INMOS Limited of Bristol, United Kingdom, and has since expanded into a family of very large scale integrated (VLSI) components with different capabilities. A typical member of the Transputer family is a single chip containing processor, memory, and communication links which provide point to point connection between Transputers.

Fundamental to the design of the Transputer is the concept of a process and how it relates to concurrency. A process consists of a list of instructions intended to be executed in sequence, and can be thought of as the program in execution in a single processor or as the entity to which the processor is currently assigned [Ref. 7:p. 55]. This definition applies not only to parallel processors but to a single processor whose processing power is sequentially shared, or time sliced, among multiple processes.

Hoare’s [Ref. 8] Communicating Sequential Processes (CSP) is one model for concurrent or parallel programming, and is central to the design of the Transputer. In CSP, a program is a collection of processes which can be combined to execute sequentially on a single processor or in parallel on multiple processors. The data space for any process executing in parallel is disjoint, thus alleviating the need for sharing
memory between processors. Although shared memory is not available, processes must still communicate with each other. Therefore, CSP utilizes message passing between any pair of parallel processes via declared communications channels between the two processes.

B. PROCESSOR ARCHITECTURE

A block diagram of a T800 Transputer is shown in Figure 2.1 and the major components are discussed in the following sections. Internally, a Transputer consists of a memory, processor and communications system connected via a 32 bit bus. The bus also connects to the external memory interface enabling additional local memory to be used.

The processor, memory and communications system each occupy about 25% of the total silicon area, the remainder being used for power distribution, clock generators and external connections [Ref. 9:p. 27].

1. Central Processing Unit (CPU) and Sequential Operation

The processor contains 32 bit processing logic, 32 bit integer arithmetic unit, instruction and work pointers and an operand register. The on-chip four kilobyte high speed memory, which can store both data and code, is directly accessed by the processor. Where larger amounts of memory are required, the processor can access a maximum of four gigabytes of memory via the external memory interface [Ref 9:p. 47]. Figure 2.2 shows a block diagram of the processor.

a. Register Set

The design of the Transputer processor exploits the availability of fast on-chip memory by employing only six registers used in the execution of a sequential process. The combination of very few registers and a simple instruction set enables the processor to have relatively simple and fast data paths and control logic. The six registers are [Ref 9:p. 28]:
The workspace pointer (W) which points to an area of memory storage where local variables are kept.

The instruction pointer (I) which points to the next instruction to be executed.

The operand register (O) which is used in the formation of instruction operands.

The A, B and C registers which form an evaluation stack.

A, B and C are sources and destinations for most arithmetic and logical operations. 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.
Figure 2.2. Transputer Processor Block Diagram

Expressions are evaluated on the evaluation stack and instructions refer to the stack implicitly. For example, the *add* instruction pops the first two values (A and B) from the stack, performs the addition, then pushes the result back onto the stack. The use of a stack removes the need for additional instructions to explicitly maneuver data during the performance of an operation. Statistics gathered from a large number of programs show that three registers provide an effective balance between code compactness and implementation complexity. [Ref 9:p. 47]

**b. Machine Instruction Format**

The Transputer instruction set has been designed for simple and efficient compilation of high-level languages. All instructions have the same format and are designed to give a compact representation of frequent program operations. Instructions are eight bits long and divided into two parts. The low-order four bits are the data and the high-order four bits are the opcode or function as shown in Figure 2.3. The data is loaded into the lower four bits of the 32 bit operand register which is operated upon by the opcode. This allows 32 bits of data to be used if required.
Instructions operate on the entire Operand Register as the operand
All instructions load their data into the lower 4 bits

Figure 2.3. Instruction Format

Four bits can provide identification for 16 instructions which are known as
direct functions. Thirteen direct functions actually manipulate the processor. These
single byte instructions are the most frequently used such as store, load, calls and jumps.
The three remaining instructions: Pfix, Nfix, and Opr manipulate the operand register by
constructing and executing larger instructions. [Ref 9:p. 29]

Pfix loads its four data bits into the operand register then shifts the operand
register left four bits. Nfix is similar except the contents of the operand register are
complemented prior to shifting left. Consequently, operands can be extended to any
length up to the length of the operand register by a sequence of prefix instructions.
Operands in the range of -256 to +255 (eight bits and two operation types) can be
represented using one prefix instruction. Finally, Opr causes its operand to be interpreted
as an operation, known as an indirect function, to be executed on the values held in the
evaluation stack. This allows up to 16 such instructions to be encoded in a single byte
instruction. However, the prefix instructions can be used to extend the operand of an Opr
instruction just like any other. The instruction representation therefore provides for an
indefinite number of operations. [Ref 9:p. 30]
Encoding of the indirect functions is chosen so that the most frequently occurring operations are represented without the use of a prefix instruction. These include arithmetic, logical and comparison operations such as add, exclusive or and greater than. Less frequently occurring operations have encoding which require a single prefix operation. Measurements show that about 70% of executed instructions are encoded in a single byte; that is, without the use of prefix instructions. Many of these instructions, such as load constant and add require just one processor cycle [Ref 9:p. 49].

2. Link Engines and Communications

Four identical INMOS bidirectional links provide communication between processors, and between processors and other compatible signal sources. Each link consists of an input and output channel, and can perform simultaneous, independent input and output communications. Instructions are included in the Transputer’s instruction set for performing input and output operations using the links. A block diagram of a communications link is shown in Figure 2.4. Each link engine also includes an independent direct memory access (DMA) controller and serial communication logic. Initiating a transfer down a link takes about a microsecond (20 processor cycles), but once the transfer is started it will proceed with minimal direction from the processor, consuming typically only four processor cycles (per active link engine) every four microseconds [Ref. 10:p. 15]. Although communications are synchronized between sender and receiver, they are not synchronous with clock input (ClockIn) to the Transputer and are insensitive to phase. Thus, links from independently clocked systems may communicate providing only that frequencies are within specifications [Ref 9:p. 370].
To execute an external communication via a hardware link, the communication link address, size and location of a message are specified. This initializes the DMA controller with the size and location of the message to be communicated. The process executing the communication instruction is suspended and a pointer for later resuming the process is saved. If available, another ready process from an active process list may then be executed. When both the sending and receiving links have been initialized in this manner, the message communication proceeds. When the communication is complete, the process which was suspended for communication is allowed to resume. A more detailed description of parallel process management is provided in [Ref 9].

Messages are transmitted by the links one byte at a time in a bit-serial format. After a receiver has recognized the reception of a byte and is capable of receiving another, the receiver transmits an acknowledge message. The transmitter will await reception of the acknowledge message before transmitting the next message byte. Since
the link hardware performs no error checking on messages, the purpose of the acknowledge message is solely to control the flow of message bytes between the links. Figure 2.5 shows a formatted message byte and an acknowledge message. INMOS recommends that error checking should be provided in software if external link length exceeds 0.9 meters. Additional hardware is available which can translate the link protocol into forms suitable for long haul communications and formats compatible with different system. Information regarding INMOS link connections and conversion to and from other protocols or signal technologies, including fiber optics, can be found in [Ref. 11], and [Ref. 12].

![Serial Communication Link Protocol](image)

Internal communications, that is, communications between parallel processes executing on a single Transputer, can also be performed. In this case, memory transfers and software register equivalents are used instead of the DMA link engines since all activity takes place within the same processor. A memory address and the size and location of a message are specified, then the communications instruction is executed just as in the external communication case. When both sending and receiving processes are ready to communicate, the task is accomplished by performing a memory-to-memory transfer of the message data. From the high level language programmer's view, internal and external link communications are handled identically.
3. Memory

Memory addressing starts at the most negative 32 bit address (hexadecimal 80000000) with four kilobytes of fast, static memory on-chip for high rates of data throughput. Each memory access takes one processor cycle [Ref 9:p 70]. It is important to note that the on-chip memory, which is rated at 50 nanosecond cycle time, is not used as a memory cache for off-chip memory, but as the first portion of one linear address range. A total of four gigabytes of memory can be addressed by the Transputer and accessed at a sustained rate of 26.6 megabytes per second (at 20 MHz clock speed) [Ref 9:p. 43], some of which is dedicated for specific purposes in Transputer operation. A memory map is shown in Figure 2.6.

| Memory Configuration, Bootstrapping Info | #7FFFFFFFE |
| Off-Chip (External) Memory | #7FFFFF6C |
| On-Chip (Internal) Memory | #80001000 |
| Register Save Locations, Queue Pointers | #80000070 |
| Communications Links | #80000020 |
| | #80000000 |

Figure 2.6. T800 Transputer Memory Map

Off-chip memory is accessed through an external memory interface controlling a multiplexed 32 bit address and data bus. Since this one bus must be shared between address and data information access to external memory is considerably slower than on-chip. A range of three to five processor cycles is typically required to access off-chip.
memory. For this reason, the programmer pursuing performance optimization should be aware of code and data placement in memory and ensure that frequently used variables and code segments are placed in on-chip memory. If faced with a choice, on-chip memory is better suited for data, since four instructions, eight bits each, can be obtained in one 32 bit fetch, hence code accesses are less frequent [Ref 10:p. 2].

Additional hardware is provided on the external memory interface to provide refresh cycles for dynamic random access memory. Control lines and signals are also provided to facilitate peripheral device direct memory access to the external segment of memory.

4. Timers

The Transputer provides two 32 bit timer clocks which 'tick' periodically. These timers provide accurate process timing, allowing processes to deschedule themselves in a specific time and allowing the programmer to execute instructions at either an absolute time or after a relative interval from a current time. One timer is accessible only to high priority processes and is incremented every microsecond, cycling completely in approximately 4295 seconds. The other is accessible only to low priority processes and is incremented every 64 microseconds giving exactly 15625 ticks in one second. It has a full period of approximately 76 hours. [Ref 9:p. 52]

Instructions are provided for initializing and reading the current timer values. Additionally, instructions are provided for suspending execution of a process until a specified timer value is reached. To implement this instruction, the Transputer maintains a linked list of suspended processes waiting on timer values. Separate lists are maintained for the high and low priority timers.

A process can arrange to perform a timer input in which case it will become ready to execute after a specified time has been reached. The timer input instruction
requires a time to be specified. If this time is in the past then the instruction has no effect. If the time is in the future the process is deschedule and rescheduled when the time is reached.

5. External Event Input

EventReq and EventAck provide an asynchronous handshaking interface between an external event and an internal process and behaves similarly to an external interrupt input. To the Transputer, this external event input appears as a communications channel which is capable of transmitting a signal to a user’s program.

A user’s program requesting input from the external event channel will be suspended if the external event input is not being asserted. Then, when the external event input is asserted the process will be added to the end of an active process list to wait its turn for execution. Either high or low priority processes may request input from the external event channel, but only one process may use the event channel at any given time.

6. Floating Point Unit (FPU)

The 64 bit FPU of the T800 Transputer provides single and double length arithmetic floating point standard ANSI-IEEE 745-1985. It is able to perform floating point arithmetic concurrently with the CPU sustaining in excess of 2.25 Mflops on a 30 MHz device. All data communication between memory and the FPU occurs under control of the CPU. [Ref 9:p. 62]

The FPU consists of a microcoded computing engine with a three element floating point evaluation stack for manipulation of floating point numbers. Each stack register can hold either 32 bit or 64 bit data, indicated by an associated flag which is set when a floating point value is loaded. The FPU stack behaves similar to the CPU stack described in Chapter II.
7. Benchmarks Performance Comparisons

Shepherd [Ref. 13:pp. 10-13] provides the benchmark data shown in Table 2.1 comparing members of the Transputer family with various other machines. Dhrystones and Whetstones are standard algorithms used in measurement of processor capability. Reference 13 includes the test code used, as well as complete reference listings of all third-party tests and vendor publications.

**TABLE 2.1**

**TRANSPUTER PERFORMANCE BENCHMARK COMPARISONS**

<table>
<thead>
<tr>
<th>Computer Evaluated</th>
<th>Dhrystones per Second</th>
<th>Single Precision Whetstones</th>
<th>Double Precision Whetstones</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 3090/200</td>
<td>31,250</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>T800-30 (projected)</td>
<td>n/a</td>
<td>6,000</td>
<td>3,800</td>
</tr>
<tr>
<td>T800-20</td>
<td>8,547</td>
<td>4,000</td>
<td>2,500</td>
</tr>
<tr>
<td>VAX 8600</td>
<td>6,423</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Intel 80386-16</td>
<td>4,300</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Motorola 68020-17</td>
<td>3,977</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Intel 80286/80287</td>
<td>1,976</td>
<td>300</td>
<td>n/a</td>
</tr>
<tr>
<td>VAX 11/780 (+FPA)</td>
<td>1,650</td>
<td>1,083</td>
<td>715</td>
</tr>
<tr>
<td>SUN 3</td>
<td>n/a</td>
<td>860</td>
<td>790</td>
</tr>
<tr>
<td>T414-20</td>
<td>n/a</td>
<td>663</td>
<td>163</td>
</tr>
<tr>
<td>Motorola MC 68000-8</td>
<td>1,136</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>IBM PC-RT (+FPA)</td>
<td>n/a</td>
<td>200</td>
<td>n/a</td>
</tr>
<tr>
<td>Intel 8086/8087</td>
<td>n/a</td>
<td>178</td>
<td>152</td>
</tr>
<tr>
<td>IBM PC-RT</td>
<td>n/a</td>
<td>12</td>
<td>n/a</td>
</tr>
</tbody>
</table>

C. PROCESSOR OPERATION - PARALLEL MODE

1. Process Representation

In both sequential and parallel operation, the fundamental unit of execution is the process. A process may consist of many sub-processes executing concurrently by sharing the processor's resources. A process may be assigned as either a high or low priority in execution. High priority processes execute without external interruption, while low priority processes run until blocked by communications or timer inputs.
Parallel operation is based on the following assumptions [Ref 4:p. 24]:

- The quickest context switch is made by saving the least amount of data for any given process,
- A process must perform I/O, or
- A process not performing I/O is in a loop and must eventually execute a loop end instruction or jump instruction, and
- A high priority process needs to execute as soon as it is ready, and executes until completion or blocked for communication.

A concurrent process may be active at any time, in which case it is either executing or on a list waiting to be executed. Inactive processes are ready to input, ready to output, or waiting until a specified time.

Concurrency administration in the Transputer is implemented in hardware, unlike most multi-tasking systems in which control takes place in software. The following hardware support is provided for implementation of parallel operations [Ref 9:p. 31]:

- Two Timing Registers,
- Four Process Queue Registers, and
- Special registers for saving some process context switch data.

2. Process Priority and Interrupts

Each concurrent process is represented by a vector of words in memory called the process workspace as shown in Figure 2.7. A process is in one of three states: executing, ready, or blocked and the Workspace Pointer Register points to the executing process. This space is used to hold the local variables and temporary values manipulated by a process. The workspace is organized as a falling stack with end-of-stack addressing. All local variables are addressed as positive offsets from the Workspace Pointer. [Ref 9:pp. 28-31]
3. Process Scheduling

The scheduler operates to avoid having inactive processes consume any processor time. Active processes waiting to be executed are held on a linked list of workspaces implemented using two registers: one pointing to the first process on the list and the other to the last. Figure 2.8 shows process S in execution, processes R, Q and P awaiting execution.

A high priority process is executed (chosen in order of receipt) until it is unable to proceed because it is awaiting an input or output, or waiting for the timer. Whenever a process is unable to proceed, its instruction pointer is saved in its workspace and the next
process is taken from the list. Actual process switching times are very small as little process state information needs to be saved; even the evaluation stack is not saved in process rescheduling [Ref 9:p. 31].

![Linked Process List](image)

**Figure 2.8. Linked Process List**

When a low priority task is executing and a high priority task is ready, it preempts the low priority task. Generally, this takes place at the end of the current instruction. Some instructions, like block moves and I/O, are interruptible. In the event of such an interrupt the state of the low priority process is saved in special system memory locations at the low end of on-chip memory and the workspace pointer is placed at the head of the low priority queue. Note: high priority tasks are not pre-empted. [Ref 4:pp. 25-28]

The process context switch time is low since only six registers need be saved and stored in fast, on-chip memory. There is full machine instruction level support for context switching which yields very low overheads. Occam context switching overhead, about one microsecond, is comparable to conventional control structures (e.g., loops, procedure calls). [Ref. 14:p. 139]
4. Time Slice Periods

Low priority processes share the processor in time sliced intervals. A time slice period is defined as 5120 cycles of the external five megahertz clock, or about one millisecond. When a running low priority process has consumed its allotted time slice, the scheduler attempts to deschedule the executing process to make the processor available for the next processes on the queue. [Ref 9:pp. 50-51]

Processes are descheduled at the end of their time slice, or shortly thereafter (to allow for completion of the currently executing instruction), and added to the end of the low priority queue. In general, the minimum period of time for time-slicing low priority processes is one millisecond with the expected maximum period being two milliseconds. Given \( n \) low priority processes and no high priority processes, the maximum time a process will wait in the process queue for CPU time is \( 2n-2 \) time slice periods. [Ref 9:p. 51]

D. PROGRAMMING LANGUAGE - OCCAM

Transputers can be programmed in most high level languages, and are designed to ensure that compiled programs will be efficient. Where it is required to exploit concurrency and still to use standard languages, Occam can be used as a harness to link modules written in selected languages. Occam and Transputers were designed together based on Hoare’s CSP. Transputers include special machine instructions and hardware to provide maximum performance and optimum implementations of the Occam model of concurrency and communications. Therefore, to gain the most benefit from the Transputer architecture, the whole system can be programmed in Occam. This provides all the advantages of a high level language, maximum program efficiency and the ability to use the special features of the Transputer. [Ref 9:p. 3]
Many programming languages depend on the existence of the uniformly accessible memory found in conventional single processor computers. As discussed in Chapter I, such an architecture would likely require the existence of a shared bus which itself becomes a bottleneck as the number of processors in a multiprocessor system grows. The aim of Occam is to remove this difficulty by expressing arbitrarily large systems in terms of localized processing and communication. INMOS describes Occam as follows:

The main design objective of Occam was therefore to provide a language which could be directly implemented by a network of processing elements, and could directly express concurrent algorithms. In many respects, Occam is intended as an assembly language for such systems; there is a one-to-one relationship between Occam processes and processing elements, and between Occam channels and links between processing elements. [Ref 2:p. 5]

Occam can further be considered an assembly language in that memory is viewed as a linear object and all resources are assigned at compile time thereby precluding recursion or dynamic resource allocation. The general simplicity of the language is recognized by being named after William of Occam, a 14th century philosopher who is responsible for the adage (known as Occam’s razor): “Entities are not to be multiplied beyond necessity” [Ref. 15:p. 1]. Hoare [Ref. 16:pp. 75-83] echoes this idea by stating, “...There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies. The first is far more difficult...”

Another attractive feature of Occam is that its semantics can be formally stated. Welch [Ref 14:p. 146] describes Occam as a formal logic for the expression, transformation and qualitative analysis of parallel, sequential and nondeterministic systems. By analysis of a language which adheres to formal rules it becomes possible to analyze a program in order to prove that it is, for example, deadlock free. Formal techniques can
also be used to transform a program into different but equivalent forms for either efficiency or readability, or to adapt parallel code to run on a single processor and vice versa [Ref 15:p. 132].

1. Processes

Occam enables an application to be described as a collection of processes which operate concurrently and communicate through channels. In such a description, each Occam process describes the behaviour of one component of the implementation, and each channel describes a connection between components. The design of Occam allows the components and their connections to be implemented in many different ways. This additionally allows the choice of implementation technique to suit available technology, to optimize performance, or to minimize cost [Ref 2:p. 5].

Each process can be regarded as an entity with internal state, which can communicate with other processes using direct communication channels. Processes can be used to duplicate the behaviour of specific entities such as a logic gate or a microprocessor. Processes are finite: each starts, performs a set of instructions then terminates. Processes may be nested within each other much as subprocedures are contained within procedures in other languages.

2. Constructs

a. Sequential: SEQ

A fundamental building block of sequential code is the SEQ statement. Occam code is written sensitive to horizontal indentation of the each line. Blocks are defined, therefore, by their depth of indentation. A block of sequential code is headed by the keyword SEQ and followed by the code indented two spaces as in the example below. The ":=" is the assignment symbol which is one of Occam's three primitives. The other primitives, "!" and "?", are discussed below.
SEQ
  x := a + b
  y := c * d
  z := x / y

b. Communication

A connection formed between two concurrent processes is known as a channel and implemented either internally in the case of the two processes existing on the same processor, or via hardware links in the case of multiple processors. Communications are synchronized and unbuffered. If a channel is used for input in one process and output in another, communication takes place when both processes are ready. Since a process may have concurrency, it may have many input and output channels performing communication at the same time.

Channel input "?" and channel output "!" along with the assignment " := " form Occam's three primitives. An example in Figure 2.9 below shows use of communication constructs in a simple buffer. The process as pictured communicates via two channels: declared "in" and "out". A series of variables assigned to "item" are sequentially taken in and sent out of the process.

```
WHILE TRUE
  SEQ
    inp item
    out ! item
```

Figure 2.9. Single Element Buffer

c. Parallel: PAR

The parallel construct keyword is PAR and has placement rules similar to SEQ. Unlike SEQ, however, each line below PAR in the next indentation will be logically executed in parallel and considered concurrent processes. In the example here the three statements are executed simultaneously. [Ref. 17:pp. 15-17]
The parallel construct is unique to Occam and provides a straightforward way of writing programs which directly reflects the concurrency inherent in real systems. A more involved example using a two stage buffer is shown in Figure 2.10 below. Note the indentation of the SEQ blocks indicating the two executing parallel processes.

Figure 2.10. Two Element Buffer

A priority level can be used with the PAR construct to form high and low priority processes. The PRI PAR construct designates the first process in the following indentation level as high priority, and the rest as low priority. If, as in the example below, there are five processes executed in parallel and it is desired that two be assigned high priority, a combination of PRI PAR and PAR can be used:
Since the PAR construct is more difficult to implement and more demanding of CPU resources it should not be used in situations where SEQ would suffice. Burns [Ref 15:p. 29] states that PAR should be used instead of SEQ when:

- It more accurately reflects the properties of the program,
- It allows the subprocesses to be arranged and thus enables program transformations to be applied,
- It allows channel operations to be executed in an order determined by the dynamics of the program's execution rather than by a predefined and meaningless sequence.

**d. Alternation**

The alternative construct, ALT, waits until one of the listed conditions becomes true, then performs that guard and any subsequent code grouped with it. The ALT construct provides a formal high level language method of handling external and internal events that are usually handled by assembly level programming in conventional microprocessors. A guard under an ALT is usually a channel input but can be other boolean conditions or combinations. In the first example below, two channels are awaiting input and each has a process associated with it. The process associated with the first channel that becomes ready will be executed. The next example shows a boolean condition and combination channel input and boolean as as ALT guards. [Ref 17:pp. 18-19]

\[
\text{ALT} \\
\text{c1 \ ? \ x} \\
\text{process1} \\
\text{c2 \ ? \ y} \\
\text{process2} \\
\text{ALT} \\
(x < y + 5) \ & \ \text{SKIP} \\
\text{process3} \\
\text{c3 \ ? \ z \ & \ continue} \\
\text{process4}
\]

Alternation is entirely nondeterministic in that the choice taken is not predictable since it depends upon actions performed by other independent concurrent processes. Furthermore, if more than one choice in an alternation becomes ready simultaneously, the guard selected is undefined in Occam. In practice, however, the last
defined of the guards becoming ready simultaneously will be selected. This is strictly a
matter of the compiler's implementation of the language and not of the language
definition. In order to force selection of a particular guard when more than one become
ready simultaneously, the PRI ALT can be used to give preferential treatment to the first
guard in the list. [Ref 17:p. 72]

e. Loop and Selection

Loops and conditional selections in Occam are very similar to other high
level languages. Conditions can be any legal expression which evaluates to a boolean
value. The constructs follow the same rules of blocking and indentation as mentioned
above with the SEQ keyword used to head a block. The examples below show a finite
and infinite WHILE loops.

<table>
<thead>
<tr>
<th>WHILE (x - 5) &gt; 0</th>
<th>WHILE TRUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQ</td>
<td>ALT</td>
</tr>
<tr>
<td>x := x - 2</td>
<td>c1 ? x</td>
</tr>
<tr>
<td>i := i + 1</td>
<td>c2 ! x</td>
</tr>
</tbody>
</table>

Selection is accomplished with IF and CASE constructs typical in many
languages. The code associated with the first expression which evaluates to true will be
executed in an IF with multiple choices. To ensure completion of the IF construct in
situations where it is possible none of the conditions are met the last condition should
always be TRUE and its associated code may be a SKIP. For clarity, a boolean constant
"otherwise" can be declared and set to TRUE to replace the keyword as the last choice in
the IF construct:

IF
(x - 5) > 0 -- first condition to be tested
x := x * 2
(x = y) -- next condition to be tested
y := x - 5
otherwise
SKIP -- otherwise assigned TRUE
f. Replication

A replicator is used with SEQ, PAR, ALT or IF constructions to replicate the component process a number of times. This ability allows for creation of multiple code segments, each with potentially different parameters based on the incremented index of the process. For example, a replicator can be used with SEQ to act as a conventional loop or with a PAR to construct an array of concurrent processes P1, P2, ..., Pn.

[Ref 17: pp. 20-21]:

\[
\begin{align*}
\text{SEQ } & i = 0 \text{ FOR } n \\
& x := x \times 2 \\
\text{PAR } & i = 0 \text{ FOR } n \\
& P_i
\end{align*}
\]

3. Configuration

Assignment of processes to specific processors is called configuration. In a sense the configuration section of an Occam program can be considered as the "main" block in other languages in which the working code primarily calls previously defined procedures and passes variables. Link assignments and global variable passing is performed during configuration. The keyword PLACE ... AT is used to assign processor links to processes.

Since each processor executes its code concurrently with other processors, a variation of the PAR construct, PLACED PAR is used. In the example below a network of processors will be assigned the process "process.a" using a replicated PLACED PAR. Note that each process.a is also passed as a parameter the value of its respective "index" which may be used in specializing the actions of each replicated process.

\[
\begin{align*}
\text{PLACED PAR } & \text{index } = 0 \text{ FOR number.of.processors} \\
\text{PROCESSOR } & \text{number.of.processors} \\
& \text{PLACE input.channel AT link0.in} \\
& \text{PLACE output.channel AT link2.out} \\
& \text{process.a (input.channel, output.channel, index)}
\end{align*}
\]
4. Program Development

Considering that parallel programming involves multiple processors, in most cases there will be many ways to perform the same task. Finding the optimum arrangement of processes and priorities will involve experimentation. Another design intent of Occam is that the logical behavior of a program is independent of how the processes are assigned to processors. According to INMOS:

> It is guaranteed that the logical behaviour of a program is not altered by the way in which the processes are mapped onto processors, or by the speed of processing and communication. Consequently a program ultimately intended for a network of Transputers may be compiled, executed and tested on a single computer used for program development. Even if the application uses only a single Transputer, the program can be designed as a set of concurrent processes which could run on a number of Transputers. [Ref 9:p. 18]

This guarantee is useful in using a single Transputer to create complex programs destined for multiple processors. Figure 2.11 shows the equivalence of single and multiprocessor code, and the relationships between internal and external channels.

---

5. Programming Practices

Significant influence of program performance can be achieved by use of priority in parallel constructs. Correct use of prioritization is especially important for most
distributed processes communicating via links and dependent upon prompt data flow for
timely program execution. If a message is transmitted to a Transputer and requires
throughrouting, as in a hypercube topology, it is essential that the Transputer input the
message then output it with minimum delay so that another Transputer somewhere in the
system would not be unnecessarily delayed waiting for the message. [Ref 10:p. 13].

Therefore, all processes which use links (external channels vice internal
channels) should be run at high priority. Processes which perform calculations only or a
minimum of communications should be run at low priority to preclude degrading the
efficiency of those processes performing external communication and subsequently the
efficiency of the network as a whole.

6. Programming Environment

The Occam compiler is packaged by INMOS within the Transputer Develop-
ment System (TDS), which provides compilers, libraries, a debugger, a unique text editor
and other facilities.

Code is entered within a "fold" editor: a text editor in which any number of lines
can be "folded" or replaced with a user defined header line. By using folds to condense
blocks of code, the entire program may always be viewed on just one screen without
having to page-up or page-down, or requiring a multiple window environment to view
different blocks of code simultaneously. A fold header line is identified by three leading
periods followed by the fold title, for example: "... fold title". Folds may be nested to
any level, and if selected wisely, can show the overall structure of a program with the
details hidden within the folds. To see the details the fold is entered and only the
contents of that fold are seen. Optionally, the text surrounding the fold may remain
visible, but will usually be off screen depending upon the size of the fold contents.
The fold editor and compiler work together as there are several specialized types of folds. A PROGRAM fold contains code intended to be booted and run on a network of Transputers external to the development board (described in Chapter III.) located in a desktop personal computer (PC). Executable (EXE) folds contain code for the development board in the PC. Both PROGRAM and EXE folds may contain separately compiled, or SC folds to break up the code into stand alone units which may be compiled and modified separately without affecting others thus reducing recompile time for minor changes. Library (LIB) folds are also available either predefined or created by the user to contain global information which is referenced within appropriate PROGRAM, SC, or EXE folds. Finally, any fold may be temporarily removed from compiler action by nesting it within a COMMENT fold. [Ref. 18:pp. 13-22,40,84-86]

Transputer Development System version D700D was used for this project. Version D included, among other utilities, a debugger for error tracing. The debugger was useful when used on code written for the B004 development board. However, as may be expected, tracing errors in an external network of Transputers is difficult. Complete details of the TDS can be found in [Ref 18].
III. MESSAGE EXCHANGE HARDWARE

The following sections discuss in detail the equipment used in the implementation of the message exchange. A complete diagram showing major component relationships is provided at the end of this chapter.

A. C004 LINK CROSSBAR

1. General

The IMS C004 is a programmable 32-way crossbar switch that supports the INMOS link protocol to provide synchronized communication between similarly equipped components. A crossbar can be thought of as a matrix of switches which permit, in the case of the C004, any one of the 32 inputs to be directly connected to any one of the 32 outputs. Figure 3.1 shows a block diagram of an IMS C004. [Ref 9:pp. 367-368]

Figure 3.1. IMS C004 Block Diagram
This ability to manipulate 32 connections serves to greatly expand the flexibility of the Transputer which typically contains four link engines. (Some specialized Transputers have only two links, and new models are planned with eight). In the normal situation, therefore, a Transputer may be directly connected to a maximum of four other Transputers or link-equipped devices. If the applications requires that more than four Transputers be logically connected, intermediate Transputers must be programmed to pass data along to the desired destination, but only after incurring delays and additional overhead as compared to a direct route. Use of a program controlled crossbar precludes this problem by allowing one (or more) links from the Transputer to be directly connected to as many as 32 different links. Thirty-two Transputers can be completely configured using two IMS C004s [Ref. 19:p. 13]. For applications requiring even greater flexibility, arbitrarily large crossbars can be constructed by cascading multiple C004s as described in [Ref 19:p. 8].

2. Hardware Description

The C004 is packaged in an 84 pin grid array chip and internally organized as a set of 32-to-1 multiplexers. Each multiplexer has associated with it a six bit latch, five bits of which select one of 32 inputs as the source of data for the corresponding output. The sixth bit is used to connect or disconnect the output. When this last bit is set high the link is considered to be active. These latches can be read and written by messages sent on the configuration link via ConfigLinkIn and ConfigLinkOut. Figure 3.2 shows the C004 hardware implementation. [Ref 9:p. 377]

A reset input is also provided to clear all latches so that no connections are active. CapPlus and CapMinus are connections for an external capacitor, and LinkSpeed sets the transfer rate through the crossbar at either ten or 20 Megabits per second.
ClockIn provides input from a five MHz clock used by the C004 for internal component timing. Recall, however, that input communication synchronization is not dependent upon ClockIn or its phase. [Ref 9:pp. 368-372]

Figure 3.2. IMS C004 Hardware Implementation
Although C004 crossbars can be cascaded in multiple connections to any depth without loss of signal, each crossbar does introduce an average 1.75 bit signal delay which arises when the output of each multiplexer is synchronized with an internal high speed clock and regenerated at the output buffer [Ref 19:p. 1]. This delay becomes significant when more than two C004s are connected in series due to the link protocol described in Chapter II. In the T800 Transputer, maximum transmission rates are accomplished by transmitting the acknowledge packet to the sender as soon as the first few bits of the 11 bit data packet is received. Therefore, the sender will be able to send the next data packet without delay since it would have already received the acknowledge for the current data packet. Communications through a series of C004s will cause a signal delay of approximately four to five bits, which in turn causes a gap to occur between consecutive data packets since the sender must wait for the arrival of the acknowledge. A single C004 inserted between two linked Transputers which fully implement overlapped acknowledges causes no reduction in bandwidth [Ref 19:p. 5].

3. Software Commands

Commands and responses are transmitted to the C004 via the ConfigLinkOut and ConfigLinkIn lines from a controlling Transputer. All communications are in sets of one, two or three bytes. Byte values zero through 31 are used to reference a crossbar link number. Bytes values zero through six are used to specify one of seven commands listed in Table 3.1 [Ref 19:p. 3]. If a message of an unspecified configuration is sent to the C004 the effect is undefined.
TABLE 3.1
C004 CROSSBAR CONTROL COMMANDS

<table>
<thead>
<tr>
<th>CONFIGURATION COMMAND [Bytes]</th>
<th>COMMAND FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] [input] [output]</td>
<td>Connects specified input to output (unidirectional).</td>
</tr>
<tr>
<td>[1] [link1] [link2]</td>
<td>Connects link1 to link2 by connecting the input of link1 to the output of link2 and the input of link2 to the output of link1.</td>
</tr>
<tr>
<td>[2] [output]</td>
<td>Enquires which input the output is connected to. The C004 responds with the input link byte designation. The most significant bit of this byte indicates whether or not the link is actually connected (bit set high) or disconnected (bit set low).</td>
</tr>
<tr>
<td>[3]</td>
<td>Essentially a pause command which should be sent immediately after a connection is made to ensure the links are ready to accept data. Necessary only if the connection will be used immediately upon established.</td>
</tr>
<tr>
<td>[4]</td>
<td>Resets all links on the C004 to disconnected status.</td>
</tr>
<tr>
<td>[5] [output]</td>
<td>Specified output is disconnected and held low.</td>
</tr>
<tr>
<td>[6] [link1] [link2]</td>
<td>Disconnects the output of link1 and the output of link2.</td>
</tr>
</tbody>
</table>

B. B012 TRANSPUTER MODULE MOTHERBOARD

The IMS B012 is a module motherboard of standard eurocard design (220 mm by 233.5 mm) which contains a collection of fixed and removable components. Two C004 crossbars (designated IC2 and IC3, discussed in the following section) and a T212 Transputer (IC1) are installed on the motherboard, and 16-pin slots are provided for a maximum of 16 removable Transputer Modules or TRAMS. TRAMS are available in a variety of configurations and sizes, some requiring more than one slot thus reducing the number of TRAMS allowed on the board. According to Watson [Ref. 20:p. 2], the design goals of the module motherboards with crossbars are:

- To be able to build systems with any number of Transputer modules in any combination or type or size;
- To be able to build a variety of different kinds of networks (e.g., arrays, trees, cubes, etc.).
• Enable any number of motherboards to be chained together;
• Make Transputer link connections configurable by software;
• To be able to run test and applications programs on Transputers without first configuring links;
• Provide a standard hardware interface to configuration and applications software;
• Make the Transputer hardware independent of the host system

Two 96-pin edge connectors, P1 and P2, provide connections between the B012 components and other Transputers, as well as power supply to the B012. The TRAM slot and link edge connector arrangements are shown in Figure 3.3.

![Figure 3.3. IMS B012 Slot And Edge Connector Arrangement](image)

1. Motherboard Configuration

Connecting pins for each slot provides power, reset and clock signals to the TRAM and connects the links of the TRAM mounted Transputer to the motherboard. In general, TRAM link 1's and link 2's are connected head to tail in a bidirectional pipe, and
link0’s and link3’s are hardwired to the C004’s. When not all slots are populated with TRAMs, or when TRAMs larger than one slot are installed jumpers must be placed in the unused connectors to maintain connectivity of the pipe. [Ref. 21:p. 2]

Some flexibility is provided by jumper block K1 which can break or rearrange the pipe into other configurations. For example, a four-by-four node matrix can be created by breaking the 16 slot hard-wired pipe into four segments by opening the appropriate K1 jumpers, then completing the mesh using crossbar connections. Instead of leaving the K1 jumpers open as in the mesh, a four-by-four node torus can be created by looping each four-Transputer segment into a ring, and making additional crossbar connections to loop the perpendicular dimensions. Variations in pipe configuration and C004 control can also be made via jumpers on the P2 edge connector. Figure 3.4 shows the 16 slots in a pipe with the variable connections via K1 and P2 identified. A specific pin-out of the K1 jumper block can be found in [Ref 21:p. 36].

Figure 3.4. B012 Configuration Variations Via P2 and K1 Jumpers
Link1 on slot0 is wired to P2 and is called *PipeHead*, and link2 on slot 15 is also wired to P2 and called *PipeTail*. By properly installing the jumpers in unused slots *PipeTail* will then connect to whichever slot is filled last on the B012. By connecting the pipe heads and tails from multiple B012s together, a large pipeline of TRAMs can be created.

In a standard configuration, slot0 is the first TRAM in the pipe and its link0 and link3 are connected to the C004s. Multiple B012s may be connected in another pipe involving the T212s. However, slot0 can be accessed directly via P2 and K1 instead of the C004s for specialized applications in which the TRAM in slot0 is used to control the T212 or the remaining TRAMs on the B012. Thus, multiple B012s can be chained together connected via the Transputer in slot0 instead of the T212s. Normally, each link0 and link3 of all slots are connected to the two C004s per the arrangement shown in Figure 3.5. [Ref 21:pp. 5-6]

As shown, the link output signals from all the link0's on all slots are connected to the 16 inputs of one C004. The link input signals from all the link3's on all slots are connected to 16 outputs of the same C004. The remaining 16 inputs and 16 outputs of that C004 are connected to the edge connector P1. The other C004 is connected similarly, except that 16 of its inputs are connected to the outputs of all link3's on all slots, and 16 of its outputs are connected to the inputs of all link0's on all slots. The remaining inputs and outputs are connected to P1. [Ref 21:pp. 5-8]
Figure 3.5. Link Organization Between B012 Slots and C004 Crossbars

The B012 link switching organization using C004s does not allow complete freedom to connect any link to any other. [Ref 21:p. 10] The result of this connection scheme does allow any link0 on any module may be routed via the C004s to any link3 on any module in just one programmed connection on each C004. A link between any link0 to another link0, or between a link3 to another link3 (henceforth defined as a "like-link") may also be established, but only with the use of two programmed connections on each C004 and two appropriately selected link cables on edge connector P1.

Edge connector P1 has three columns of 32 pins each: the pins in one column are all ground, another column are all link inputs, and the third are all link outputs. These 32 connections represent half of all available connections on the two crossbars, with the
remainder hardwired to the slots. Therefore, each row of three pins provides a bidirectional link, and the 32 rows can be connected as the user desires to fulfill the needs of the application at hand. The 32 rows are numbered from 0 (at the top) to 31. Figure 3.6 shows the connections routed to P1 and P2. [Ref 21:pp. 9-10,26]
A module motherboard has *Up*, *Down*, and *Subsystem* ports on the P2 connector which provide hierarchical control. A board is able to control a subsystem of other boards by connecting its *Subsystem* port to the *Up* port of the next board. By connecting the *Down* port of the first subsystem board to the *Up* port of the next a daisy chain of boards is established. At any point down the chain, a new subsystem may branch by starting with a *Subsystem* port. This hierarchy is shown in Figure 3.7. B012 architecture also provides flexibility by allowing different sources of reset initiation for different components. For example, if there are n slots on a motherboard, modules in slots one through n may be controlled from either the *Up* port of the board or may be part of a subsystem controlled by the module in slot0. [Ref 20:p. 9]

![Figure 3.7. Transputer Control Hierarchy](image)

2. T212 Transputer Crossbar Controller

A T212 Transputer is used to control the two C004 crossbars on the B012 motherboard. The T212 has four links of which link0 and link3 are connected to the C004s. Link1 and link2 are routed to the P2 edge connector, labelled *ConfigUp* and *ConfigDown* for access to external systems or routing back to other available connections on the B012 such as *PipeHead* and *PipeTail*. Configuration data for the C004 is fed into
one of the T212’s links (*ConfigUp*) from an external Transputer or from one of the TRAMs on the B012. Figure 3.8 shows the link connections from the T212 [*Ref 21:pp. 10-12*]

![Diagram of link connections from T212 to C004-0 and C004-1](image)

**Figure 3.8. T212 C004 Controller Link Organization**

The T212 Transputer follows the description in Chapter II above on the T800 with some key exceptions. The T212 is a 16 bit processor with only two kilobytes of on-chip memory. Although the T212 can address additional memory externally, there are no provisions on the B012 for additional memory. (Another Transputer model, the T222, is pin compatible with the T212 and holds four kilobytes of on-chip memory). Due to the 16 bit architecture a maximum of 62 kilobytes of external memory can be accessed [*Ref 9:p. 318*] via separate 16 bit data and address signal paths.

As a controller for the C004s, the T212 is intended to hold a small program which directs the connections made by the C004s. This program can run independently of external inputs, or can receive direction from other Transputers via *ConfigUp* and *ConfigDown*. By making appropriate jumpers on the P2 edge connector, the T212 can be
included in the pipe formed by the TRAMs on the B012. A standard configuration for a multiple B012 board system is to form a daisy chain of control from a host via ConfigUp, and connecting subsequent ConfigDown to ConfigUp connections. [Ref 21:p. 12]

3. Transputer Modules

INMOS Transputer modules (TRAMS) are designed to form the building blocks of parallel processing systems in which more than the on-chip memory is required. They consist of printed circuit boards in a range of sizes which typically hold a Transputer, some memory, and perhaps some application specific circuitry. A TRAM needs only a five volt power supply, ground and a five MHz clock to operate. These are supplied to the module through pins connecting the TRAM to the motherboard. Other pins carry signals for the Transputer’s links and reset, analyze and error signals. [Ref 21:pp. 12-14]

IMS B401 is the designation of the TRAMs used in this project. Each B401 holds a T800 Transputer and 32 kilobytes of static random access memory and connects to the motherboard using one slot via two rows of eight pins each. Sixteen B401 TRAMs may be placed on one B012 motherboard for a total of 16 Transputers and 576 kilobytes of memory. Figure 3.9 shows the pin arrangement of a B401 TRAM. Full details of this particular TRAM can be found in INMOS product information [Ref. 22].

![Figure 3.9. B401 TRAM Pin Arrangement](image-url)
C. B004 DEVELOPMENT BOARD

1. General

Transputer components form a unique hardware environment which is not immediately compatible with most existing personal computers (PC) or main frames upon which development work is accomplished. The IMS B004 development board was designed to meet these needs by interfacing a Transputer memory with an IBM type personal computer allowing the software developer to edit, compile and test software using the PC as a host. Figure 3.10 shows a block diagram of the B004 development board which fits in a full length eight bit slot of an IBM compatible PC. [Ref. 23]

![Block Diagram of the B004 Development Board](image)

**Figure 3.10. IMS B004 Transputer Development Board Block Diagram**

The B004 contains a T414 Transputer, two megabytes of memory and circuitry to connect with the host computer. This connection is made via one of the Transputer’s four available links and typically through link0. This leaves the remaining three links to connect to additional Transputers in the host PC or in other locations. When the B004 is
used as a host or root Transputer for a larger system, compiled code is booted via the B004 to the other Transputers along link connections. The TDS may prompt the user for the B004 link number being used to boot other Transputers. Once the first link is established, the TDS relies on the configuration information provided in the PROGRAM fold to continue routing compiled code to the assigned Transputers. The order of external Transputers booted follows the order in which they textually appear in the PROGRAM's configuration fold. Each Transputer to be booted is first reset by the host, booted with its assigned compiled code, then used to boot the next Transputer in the system. This process repeats as necessary, following configuration fold link definitions, until all external Transputers are booted.

Backplane connections are provided for link hook-ups and subsystem control (Up, Down, and Subsystem) as discussed for the B012. Jumpers are also attached at the backplane to connect one of the links and the subsystem connection to the host PC. In this case, the host PC acts as the master controller of all Transputers in the network. Additional Transputers daisy chained to the B004 are controlled hierarchically via the Down connection. [Ref 23:p. 10]

B004 interface to the host PC includes access to the PC's resources such as monitor and mass storage. Depending on the compiler used, coded library routines may be available to display information on the monitor or store computed results on the hard disk. Transputer Development System D700 version D was used for this project to edit and compile the Occam program code. Extensive use of library calls to predefined display subroutines was made in order to facilitate monitoring and debugging of code.
2. T414 Transputer

The B004 development board used contained a single T414 Transputer. With few exceptions, most of the information contained in Chapter II above applies to the T414 model. Unlike the T800, the T414 does not include an on-chip floating point unit and only holds two kilobytes of on-chip memory. [Ref 9:p. 179]

T414 link communication protocols are the same as T800, however, the B004 T414 does not send the ACK packet immediately upon receipt of the first few bits of incoming data. Instead, the B004 T414 waits until the entire data packet is received before the ACK is sent. Although the T414 does support the high speed 20 megabits per second data transmission rate, other components on the B004, particularly the PC interface [Ref 23:p. 14], restrict the T414 to ten megabits per second link speed. Since the T414 is used to intercept and test or record communications from the T212 occurring along the link controlling pipe of the message exchange, the entire system data flow rate is maintained at ten megabits per second.

D. MESSAGE EXCHANGE HARDWARE CONFIGURATION

A complete view of the message exchange hardware is shown in Figure 3.11. This diagram shows the relationships between the key components, including the C004 crossbars, crossbar controller, TRAMs, and the B004 used in monitoring the system. Four TRAMs in slots zero through three are shown corresponding to the hardware used for this project. The numbers on the left and right sides of the figure show the connections between the edge connector P1 and the actual link designations on the C004s. (The numbers under "P1 #" are pin row numbers on the edge connector). An arrow accompanies each pair of numbers indicating its communication direction: pointing towards the C004 for LinkIn and away from the C004 for LinkOut.
Figure 3.11. Message Exchange Hardware Configuration
IV. MESSAGE EXCHANGE FUNCTIONAL DESCRIPTION

A. OVERVIEW

Multiple processors connected in some topology can be described as belonging to one of three configuration categories: static, semi-static, and dynamic [Ref. 24:p. 124]. In the static topology, the interprocessor connections are fixed for the duration of the application. Any significant change requires system down time and hardware modification. Semi-static topologies allow for system reconfiguration at synchronized points as the application progresses. For example, one topology may be best for loading data, then once all data is loaded, all processors switch to a second topology better suited for processing the data. A third topology may later be used to output or display the results. The Esprit Project P1085 [Ref 24] discusses a large scale, semi-statically reconfigurable Transputer network which uses crossbars (72 x 72), a specialized control bus, and a three layer operating system.

Most flexible of the three is dynamic configuration in which any processor can be dynamically connected to any other processor upon request while the application is running. Harp [Ref 24:p. 124] states that dynamic switching allows dynamic load balancing, fault tolerance and offers potential benefits for efficient implementation of high level languages. Although the greatest flexibility can be achieved with complete interconnectivity, dynamic configuration also incurs the greatest overhead in system maintenance.

The message exchange program as developed here is intended to provide maximum flexibility in data transfer within a dynamically reconfigurable Transputer network. As explained earlier, each Transputer has four links with which to communicate to similarly
equipped components. If a given Transputer requires communication to more than four components, data must be routed through the intermediaries until it eventually arrives at the intended destination.

The message exchange developed here uses the C004 crossbars on a B012 motherboard to establish direct connections between requesting Transputers. With this hardware, a process using one link of a TRAM can be directly connected to any of the other 31 TRAM links on a fully populated B012 (that is, with 16 TRAMs). Additional code was also added to detect and recover from failed external links between crossbars used in making these connections.

Hill [Ref 19] served as the basis for this program, however, significant modifications were made to complete the partial code provided and adapt it the hardware design of the B012. The example cited was designed around a proposed use of a single C004 and employed all Transputer components as traffic routers for external processors. The program here uses both C004s of the B012 and allows application code to be embedded in each TRAM for production work, with the remaining code serving as a communications kernel which obtains the requested connections status.

A functional description of the message exchange in both normal and link failure operations is discussed below, followed by a detailed explanation of the code in the Controller and TRAM modules. A complete listing of all message exchange code is provided in Appendix A. Code used in program development and testing is provided in Appendix B. Appendix C contains code for a visual display of message exchange activity which was used extensively in system debugging.
B. SYSTEM OPERATIONS

1. Link Control Command Description

Communications on the link control pipe consist of three-byte packets: one byte each for link command, source, and destination [Ref 19:pp. 16-17]. There are six commands issued by either the controller or any of the managers to operate the crossbars and thus dynamically control the network topology and recover from failed links. All commands are predefined in a library of constant definitions made available to the appropriate procedures.

- **REQ** (link request) issued by a Manager (process on a TRAM monitoring link control commands) requesting the creation of a new link for one of its two Agents (processes on a TRAM performing data output) as a source.

- **ACK** (request acknowledge) issued by the Controller (process on T212 link crossbar controller) when a requested link is successfully established, thus informing the Manager the new link is available.

- **REL** (link release) issued by a Manager when an existing link is no longer needed. After communications are completed and the specified path no longer in use, the link resources should be freed for other users.

- **BRK** (release acknowledged) issued by the Controller when the link has been released. This allows the Manager to repeat the cycle and request a new link to the next desired destination as necessary.

- **ABT** (communications abort) issued by the Manager when an attempted outbound communication has exceeded a watchdog timer, thus signalling a communication failure.

- **REI** (link reinitialization) issued by the Controller to synchronize reinitialization of both directions of both ends of the affected link.

2. Link Failure And Recovery Procedures

Under routine circumstances, that is, using only the Occam "!" and "?" primitives for external link communications, a hardware failure of the external link used may cause the Transputers involved not only to lose data, but also to hang entirely causing a need for a hardware reset. Transputer deadlock occurs as soon as a data packet
is sent along a link and its corresponding acknowledge packet is never received. (Or conversely, depending upon when the failure took place, an acknowledge packet is sent and the next data packet never arrives). Although a crossbar can be employed to switch in a "good" connection to replace the failed wire, the Transputers in use are still stuck as before while its link engine is waiting for a transmitted but lost packet which will not be regenerated by either Transputer.

Recovery of a failed link and the Transputers involved, employs Transputer Development System procedures in the reinit library. These library procedures implement input and output processes in lieu of the standard "!" and "?" primitives, and can be made to terminate the process even in the presence of a communication failure. The five library procedures and their associated parameters are as follows: [Ref 9:p. 271]

- PROC InputOrFail.t (CHAN OF ANY c, []BYTE message, TIMER time, VAL INT t, BOOL aborted)
- PROC OutputOrFail.t (CHAN OF ANY c, []BYTE message, TIMER time, VAL INT t, BOOL aborted)
- PROC InputOrFail.c (CHAN OF ANY c, []BYTE message, CHAN OF INT kill, BOOL aborted)
- PROC OutputOrFail.c (CHAN OF ANY c, []BYTE message, CHAN OF INT t, BOOL aborted)
- PROC Reinitialise (CHAN OF ANY c)

Input/OutputOrFail.t procedures take the following parameters: the data channel c which is to be guarded for faults, the actual data variable message, a timer channel time and a delay value t in clock ticks. The procedures return the boolean value aborted equal to true if the communication failed to complete in the allowed time interval t. If the communications are successful, the data is passed and aborted is set to
false. In either case, although the communications may fail, the library procedure always successfully terminates, thus protecting the system from failure. By using a timer to determine successful communication completion, these procedures are used to detect a failure and thus initiate recovery actions. Communication failure detection functions are assigned to the Agent process which is responsible for conducting data output on each TRAM in the message exchange. Figure 4.1 shows a state diagram tracing through the sequence of control commands involved in both normal and fault conditions.

![Link Control Command State Diagram](image)

Figure 4.1. Link Control Command State Diagram

In situations where a watchdog timer is not sufficient to detect communications failure, but where the failure can be detected by other means, the Input/OutputOrFail.c procedures can be used. Instead of reporting a failure based on a timer, this pair of procedures takes an additional external signal, an integer kill, to force the current communications to terminate. Since the data output routines in the Agent processes
detect and report a link failure, the communications input of the User processes use these external "kill" procedures to terminate a failed communication based on link control signals from the appropriate Agent via the link control pipe. By relying upon only the communication source (output) end of the link to detect link failures, timing difficulties and redundant fault signals inherent in the message exchange system are avoided.

Finally, the Reinitialise procedure, which takes as its only parameter the communication channel c involved, actually resets the failed link engine allowing them to be reused. Reinitialise should be called only after both directions of both link engines are terminated (by the above procedures) and no further communication is attempted by them. Before a reset link engine is tasked with conducting communications, the crossbar controlling Transputer should have assigned a working connection to correct the fault. Repeated failures, however, are tolerable and the sequence of events in link recovery repeats until the desired communications are completed successfully using a working connection.

3. Link Control Command Action Summary

Table 4.1 describes the life cycle of the various controlling commands as they are passed among the key modules in the pipe. Origination of each command is straightforward and not flexible, given the requirements of the system. However, termination (removal from pipe) of commands was altered several times during software development. In all cases possible, as soon as the command has travelled along the pipe to complete all its functions, the last station taking action on the command removes it from the pipe. When several components take action on a single command (link reinitialization, for example), the command parameters may be modified allowing the terminal component to remove the command without fear of preventing the flow of needed control information.
<table>
<thead>
<tr>
<th>CMD</th>
<th>CONTROLLER</th>
<th>MANAGER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>With Source Involved</td>
</tr>
</tbody>
</table>
| REQ | If link made: *Link.REQ Consumed, Link.ACK Originates.*
     | If link not made: Link.REQ Recycled. | *Link.REQ Originates* prompted by Agent, which is itself prompted by first byte of data message from User. | Link.REQ passed on. |
| ACK | *Link.REI Originates.*
     | *Link.REI Consumed.* | Link.ACK passed on, source byte set to nil, notify Agent to transmit data. *Link.ACK Consumed* if destination byte already nil. | Link.ACK passed on, destination byte set to nil, notify User to receive data. *Link.ACK Consumed* if source byte already nil. |
| REL | If link broken: *Link.REL Consumed, Link.BRK Originates.*
     | If not broken: Link.REL Recycled. | *Link.REL Originates* based on request from Agent upon successful completion of data transmission. | Link.REL passed on. |
| BRK | *Link.BRK Originates.*
     | *Link.BRK Consumed.* | *Link.BRK Consumed,* notify Agent link is broken. User passes next data block to Agent. | Link.BRK passed on. |
| ABT | Link.ABT passed on.
     | *Link.REI Originates.*
     | *Link.REI Consumed.* | *Link.ABT Originates* based on notification from Agent of failed communication. *Link.ABT Consumed* when command makes complete circuit, ensuring destination informed of failure. | Link.ABT passed on, notify User to abort input of data. Note: destination User notices link failure only when informed via the command pipe. Failure detected by source Agent. |
| REI | *Link.REI Originates.*
     | *Link.REI Consumed.* | Link.REI passed on, source byte set to nil, notify Agent to reinitialize link. *Link.REI Consumed* if destination byte already nil. | Link.REI passed on, destination byte set to nil, notify User to reinitialize link. *Link.REI Consumed* if source byte already nil. |
C. CONTROLLER MODULE

The Controller code executes in a single process which resides entirely on the T212 C004 controller installed on the B012 motherboard. In the configuration established for this project, the T212 is connected in a unidirectional pipe formed by the TRAMs on the motherboard. Link request and status information is circulated in the pipe until received by the Controller, at which time appropriate action is taken to either connect or disconnect a requested data path, or to recover from a reported failed path. The Controller directs the crossbar resources of the B012 ensuring that all requests are fulfilled, if physically possible, and avoiding potential system deadlock caused by partial allocation of resources.

The Controller code consists of a main section which monitors the link control pipe and takes action by calling one or more of its four subprocedures, while also managing connection resources and status. Subprocedure’s functions will be discussed individually followed by an explanation of the main code for the Controller.

1. Subprocedure: reset.or.nil

When the enquire command and the output link designation of interest is sent to a C004, the response is the byte designation of the input link currently connected (see Table 3.1). The left most bit of the response byte (bit seven) indicates that link is active when set high, or inactive (disconnected) when set low. To define one of the 32 possible connections to any given output link, five switches must be set. Their status is indicated by the five lower bits of returned byte (bits four through zero). When a programmed connection is broken, only bit seven is changed (set low); the selection bits are altered only when a new link is established. Bits six and five of the enquire response byte are not used.
Since past connections to a given link are of no interest to this program, the reset.or.nil procedure will return byte.nil if bit seven of the enquire response byte is low, thus indicating that the connection is inactive and free for assignment. If bit seven is high (indicating an active connection), reset.or.nil returns the current connected input link designation by resetting bit seven of the enquire response byte and leaving bits four through zero intact. Setting bit seven low is accomplished using the Occam ">_<" bitwise exclusive-or operator on the hexadecimal value 80 which isolates the left most bit of the C004 enquire response. This subprocedure is repeatedly called by other subprocedures as well as the main code.

2. Subprocedure: c4.cmd

A small code sequence used in communicating with the crossbars is used repeatedly throughout the Controller and is coded as a subprocedure for clarity and to improve memory usage efficiency (recall that only two kilobytes of memory are available on the T212). C4.cmd sends a command followed by zero, one or two parameters (as appropriate for the command: see Table 3.1) to a selected crossbar. The choice of crossbar command to implement is determined by the value of the parameters received. Upon completion, the returned byte is placed in parameter b3. The Occam keyword VAL coupled with other type identifiers, defines the parameters that follow to be constants for the duration of the subprocedure.

PROC c4.cmd (CHAN OF ANY to.x, from.x, VAL BYTE cmd, bl, b2, BYTE b3)

If the command involves a change of link status, c4.cmd code proceeds to immediately query the crossbar and verify the requested status change, for example: that the requested connection is created as desired. The result from the enquiry is then run
through the `reset.or.nil` subprocedure which transforms the crossbar response into a form indicative of the task accomplishment. For example, if a link was to be broken the value returned by `c4.cmd` (via use of `reset.or.nil`) would be `byte.nil` indicating a free link.

3. **Subprocedure: get.current.tie**

   No program defined tables or records are kept by the *Controller* with regard to link connection status. When a connection status must be determined, for example, to verify a link is free prior to making a connection, the C004s are consulted directly. As discussed above, the `enquire` command is used to establish current connections. By sending the command with the byte designation of the output link of interest, the C004 will respond with the byte designation of the currently connected input link. `Reset.or.nil` is then called to strip the high bit from the response if present, or change to `byte.nil` if not.

   Since the five selection bits (four through zero) of the `enquire` response byte change only when a new link connection is ordered or when the entire C004 is reset, current status is always available and accurate. By relying strictly on the hardware switch positions internal to the C004s, potential errors in status table maintenance are completely avoided. Processor effort needed for C004 status enquiry is comparable to status table look-up and maintenance since all *Controller* code is sequential, therefore no competition exists for use of the links between the T212 and the C004’s.

   All subprocedures dealing with the C004s are passed four channels: `to.c0`, `from.c0`, `to.c1`, and `from.c1`, which are the communication paths between the T212 and the C004s. The `get.current.tie` subprocedure is also passed the source or destination link of interest, and it returns either the designation of the current connection, or `byte.nil` if the link of interest is inactive.
PROC get.current.tie (CHAN OF ANY to.c0, from.c0, to.cl, from.cl, VAL BYTE in.link BYTE link.tied.to)

Initially, get.current.tie must determine which C004 to enquire since link0's and link3's are connected differently (see Figure 3.3). This is complicated by the numbering scheme of the C004 links and the slot numbers that are hardwired to them. Therefore, a translation array is used to convert the C004 link designation to a slot number. In this program, link0's are numbered 0 through 15, and link3's are numbered 20 through 35.

It should be recognized that a consistency between the C004 wiring schemes is relied upon to perform these operations. Regarding B012 organization, Figure 3.3 showed that for a given TRAM link0 or link3, the input is routed to one C004 and the output to the other. Although different C004s are involved, the C004 link designation used is the same. This is valuable since the enquire command responds only with the input link connected to the output link in question. There is no enquire to determine what output is connected to a given input. Therefore, the connected output link is assumed to be the same link designation (although different C004s) as the input link. This assumption has proved reliable since all connections are accomplished for both input and output directions in tandem.

4. Subprocedure: make.conn

When the main code directs the establishment of a connection, the make.conn subprocedure is called. All actions necessary to complete the requested connection are contained in this subprocedure. Each routine includes a returned boolean value conn.ok which provides connection verification results. Make.conn is called with the four C004 channels, source and destination bytes, and returns only the boolean conn.ok:
PROC make.conn (CHAN OF ANY to.c0, from.c0, 
    to.cl, from.cl, 
    VAL BYTE in.source, in.dest, 
    BOOL conn.ok)

Three distinct cases must be handled by this procedure: connect any link0 to any link3, connect any link0 to link0, and connect any link3 to link3. The first, link0 to link3, is straightforward since only one connection must be made on each C004 and the procedure is symmetric for each link. Figure 4.2 shows how the crossbars are used to connect any link0 to any link3 on the B012. Note that the link0 and link3 involved can even be of the same Transputer. After enquiring the C004s with the source and destination links of interest, make.conn orders the connection if both links involved are free. Otherwise, the procedure returns false for conn.ok. The remaining two cases, however, are much more complex in that two connections on each C004 must be used as well as an external edge connector jumper. Since like-link cases are similar, an internal subprocedure make.0033 is used.

Figure 4.2. Crossbar Connections To Connect Any Link0 To Any Link3
Although the C004s can provide immediate status of all internal connections, there is no direct way to establish placement of the P1 edge connector jumpers on the B012. Therefore, the user supplies a table listing the combinations of edge connectors to be used to satisfy either link0 to link0 or link3 to link3 requests. Sixteen jumpers can be arranged on the P1 edge connector pins, but not any jumper arrangement will satisfy either type of connection. Due to the design of the B012, only specific combinations may be used to complete a like-link connection. Table 4.2 shows one selection of possible jumper arrangements.

For this project, eight jumpers were selected for link0 to link0 connections and eight for link3 to link3. The user provides values in two byte arrays edge.a and edge.b in which the values for a specified index form a pair of edge connectors defining the location of a jumper. For example, edge.a[3] and edge.b[3] define connections 25 and 26, respectively. This tells make.0033 that these C004 links can be used to jumper across from one C004 to the other to establish a connection between two link0’s.

<table>
<thead>
<tr>
<th>TABLE 4.2</th>
</tr>
</thead>
</table>

**JUMPER PLACEMENT FOR C004 TO C004 CONNECTIONS**

<table>
<thead>
<tr>
<th>Connect P1 Row:</th>
<th>To P1 Row:</th>
<th>Connect P1 Row:</th>
<th>To P1 Row:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>16</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>26</td>
<td>27</td>
</tr>
</tbody>
</table>

The user also provides two integer values: edge0.ct and edge3.ct which are the number of possible jumpers available for each type of connection. The sum of edge0.ct
and \textit{edge3.ct} is the total number of jumpers installed. The first \textit{edge0.ct} listings in the two byte arrays are for link0 to link0 connections, and the subsequent \textit{edge3.ct} pairs are for link3 to link3 connections.

A boolean table, \textit{edge.ok}, maintained globally in the \textit{Controller} module, tracks reported status of each edge connector. When a TRAM reports an edge connector has failed, the corresponding index in \textit{edge.ok} is set to false. Only edge connectors with a corresponding \textit{edge.ok} value of true are used to complete like-links.

With the jumper information provided, the \texttt{make.0033} subprocedure can proceed to establish a connection between two like-links. Figure 4.3 shows the necessary crossbar and jumper connections to accomplish this. As each crossbar connection is made the procedure immediately enquires the status of the assumed newly formed links and checks the returned answer against what is expected. Any deviation in comparing expected from actual status is treated as an incomplete connection and the \texttt{make.conn} procedure returns FALSE for the value \texttt{conn.ok}.

5. Subprocedure: \texttt{break.conn}

Procedure \texttt{break.conn} works similarly to \texttt{make.conn} in that the same three cases of link combinations must be handled, and the two cases involving like-links are handled with an internal subprocedure. Significant difference is that the C004 command to disconnect links, \texttt{c4.disconn.output}, is sent instead of the command to make connections. Enquiries of link status are still conducted and the returned bytes are sent to the \texttt{reset.or.nil} procedure. Successful termination of a link is evidenced by all links involved showing the \texttt{byte.nil} status which shows that the links are now free for their next assignment.
It should be noted again that there are no link status tables to be maintained or updated. In all cases, link status is obtained directly from the C004s involved to ensure accurate assignment and freeing of crossbar connections.

6. Main Controller Code

After variable initialization, the Controller module enters an infinite loop which constantly receives the next link control bytes and takes the appropriate action. Program structure is shown below followed by details of key folds.
SEQ
  ...initialize C004s, clock,
  and edge.ok status array
WHILE TRUE
  ALT
    from.pipe ? token; source; dest
    IF
      token = link.req
        ...action for connection request
      token = link.rel
        ...action for connection release
      token = link.abt
        ...action for failed link
      token = link.ack
        ...action for recycled link acknowledge
      token = link.brk
        ...action for recycled link break
      token = link.rei
        ...action for recycled link reinitialize
      otherwise
        ...action for unrecognized command
        (clock interval reached and
         failed request count limit exceeded)
      ...reset a edge.ok status to TRUE

a. Action For Link Request

Upon receiving a link request, the status of the requested links are
determined using get.current.tie. If the returned status for both source and destination
are byte.nil (indicating free links), the requested connection is established using
make.conn, and a link acknowledge (link.ack) packet is sent to the pipe to inform the
TRAMs involved that a connection is made and data can be transmitted. If a requested
link is currently busy, the link request is recycled through the pipe in hopes that the
resources will be freed for reassignment by the time the request returns. Using the link
control pipe as a waiting queue for failed requests ensures that the request will not be lost
while it is waiting for resources, and also minimizes storage and processing requirements
of the Controller code on the T212.
If the link request could not be fulfilled due to lack of available working "good" edge connectors, a counter is incremented tracking the number of such failed requests. When the req.cnt reaches a preset value, the edge connector management algorithm begins searching for repaired resources. This is explained in more detail below.

b. Action For Link Release

A link release command is a TRAM request that an existing link be freed for new use as needed. A single call to the break.conn terminates the connection and frees the resources. Although there is no resource conflict preventing a link from being disconnected, if the crossbar does not respond with the expected byte.nil status, link.rel will be recycled through the pipe and break.conn will be called again. Upon successful termination of a connection, a link break (link.brk) packet is sent, informing all concerned that the resources are free.

Link request and release commands are the first two choices in the WHILE TRUE loop of the controller due to their frequency. In normal conditions, only these two commands will ever reach the Controller module for action.

c. Action For Link Abort

When a link abort (link.abt) is received, the command is immediately passed on to the pipe since additional action must be taken by other TRAMS. (The abort command is eventually removed by the originating TRAM). Next, the Controller determines the exact connection and its edgelist table index involved via the get.current.tie procedure. To prevent a known bad edge connector from being immediately reused by make.conn, the edge.ok value of the corresponding index is set to false.

With the failed edge connector marked as bad, the Controller continues by sending a link reinitialize (link.rei) to the pipe so that both directions of both links
involved in the aborted communication may be reset. Note, the reinitialize command is sent after the link abort command to ensure that all communication attempts cease prior to completing the link engine reinitialization process. This is required by the TDS link engine recovery procedures. At this point, the existing failed connection is formally broken with `break.conn` and remade with `make.conn` using a different path.

From here, the procedure is similar to that of the link request action above. If the connection cannot be reestablished, a newly generated link requested is placed in the pipe. If the failure to make a new connection is due to lack of edge connector resources, the `req.cnt` counter is incremented. If a new connection is made, a link acknowledge is placed in the pipe to inform the TRAMs involved to attempt communications again.

**d. Action For Link Acknowledge, Break And Reinitialize.**

Under normal circumstances, these commands are removed by the last TRAM which takes action based on their receipt. If a TRAM fails to remove such a command, they will be removed here to prevent the pipe from being flooded with meaningless circulating commands.

An additional section is added to pass noncommands along the pipe. This is used exclusively for testing purposes when specialized test commands were placed in the pipe to monitor specific actions. Such testing or status information, like TRAM code completion, is removed by the B004 monitoring the system.

**e. Edge Connector Status**

Also within the `WHILE TRUE` loop is a block of code which manages the status of the edge connector links. Initially, `Controller` is told by the programmer which connections are hard wired and that their status is "good", that is, the edge connectors as listed in the data library `edgelist` are immediately available for use to establish
connections between like-links of TRAMS. If a TRAM detects and reports a failure as
the message exchange proceeds to make and break links, the Controller marks the edge
connector involved as "bad" and avoids assigning it in the future. This points out a
sharing of responsibilities between the Controller and TRAM processors. An Agent on a
TRAM has no knowledge of the routing assigned when a connection is requested and
acknowledged, however, it will detect a failure if the connection is bad. Conversely, the
Controller which made the requested connection has routing information, but cannot
itself determine if the connection is actually operational without being told by the
requesting TRAM. Similarly, the Controller cannot by itself determine when a failed
connection has been repaired without first assigning it to a TRAM for use. Therefore,
edge connectors previously marked as "bad" are periodically set to "good" so that they
may be assigned and tested. A timer-based trigger is used to prompt the Controller to
reset edge connector status.

Immediately within the WHILE TRUE loop is an ALT which guards both
the pipe input and a boolean expression. The boolean expression evaluates to true when a
predefined time period has elapsed and the number of failed link requests due to lack of
data connector resources has exceeded a predefined set point. Occam keyword AFTER
is used to measure the delay from clock time now which was initialized when the
program began, and reset when this expression evaluates to true. PLUS is a modulo
addition operator which accurately adds clock times since timers recycle from maximum
positive number to zero. The "&" is Occam boolean-and which requires both halves of
the expression to be true for the entire expression to evaluate to true. Under normal
conditions when no data connector faults are reported, the expression never evaluates to
ture since req.cnt never reaches the trip point (currently set at the number of TRAMs in
the system). When many link failures are present, the timer delay prevents the expression
from constantly evaluating to true and thus consuming too much CPU time.
(req.cnt > no.trams) & clock AFTER now PLUS delay

When the combined expression evaluates to true, it signals a need to look for new edge connector resources. If too many edge connectors are marked as bad, the system slows appropriately since the remaining connectors must be used for all like-link connections causing many link requests to be recycled as they wait their turn. If repairs are made to the bad wires, the Controller will not realize the status change unless the connection is reused by the TRAMs. Therefore, when many link requests fail due to lack of resources, the Controller looks for newly repaired edge connectors by periodically resetting their status to "good", thus making them available for use by make.conn as the need arises. If the connection is still bad, the status is reset accordingly. If the connection has been repaired, the status remains good until it fails again.

D. TRAM MODULE

Code on each TRAM is organized into five processes running in parallel and communicating via internal channels. Figure 4.4 shows the assignment of both internal and external channels to the TRAM processes with the configuration level link names written outside the processor box and local channel names written on the inside. Note the division of the data link directions: data output is the responsibility of the Agent process and data input is received by the User process. Although each data link has its own Agent, it was not necessary to separate the User processes between the two links. Since the application code resides in User, the User processes could be combined into one process or connected via an internal channel so that the given application could have simultaneous use of both links. However, User processes were kept separate in this project to allow independent testing of both data links on each TRAM.
Figure 4.4. **TRAM Code Process Organization**

1. **Manager Process**

   Dedicated to managing the data link resources of the message exchange for a single TRAM, the *Manager* is connected via external links to the link control pipe, and via internal channels to both *Agent* and *User* processes on the TRAM. Since the code for the TRAM processes is replicated on each processor, two constants, `slot.desig` and `no.trams`, are passed to the *Manager* along with the channels. *Slot.desig* tells each TRAM which processor it is corresponding to the slot number in which it resides on the B012. The slot number is also used to describe the data links: `link0` of a given TRAM...
equals the slot.desig, and link3 equals slot.desig+20. No.trams is a global constant telling the system how many processors are installed on the B012, thus allowing crossbar connections only to existing processors. Manager procedure call is as follows:

PROC manager (CHAN OF ANY to.pipe, from.pipe, 
mgr.to.agent0, mgr.from.agent0, 
mgr.to.agent3, mgr.from.agent3, 
mgr.from.user0, mgr.from.user3, 
CHAN OF INT mgr.to.user0, mgr.to.user3, 
VAL INT slot.desig, no.trams)

Manager consists of an infinite loop monitoring all channel inputs, and a three stage control pipe output buffer running in parallel with the infinite loop. The buffer itself consists of three parallel processes, each as one stage inputting a three byte link command, source, and destination, then passing it to the next stage. The middle stage is a replicated parallel process allowing for easy increase in buffer size, however, testing showed that minimizing buffer size improved performance, and the smallest reliable size is three stages to allow room for continuous flow along the pipe.

Immediately within the WHILE TRUE loop is an ALT which guards the three possible direct inputs to Manager: from either Agent or from the pipe. Each input is described separately below.

a. Action For Input From Agent

Either Agent communicates the needs of the application program via a single byte along the mgr.from.agent0 or mgr.from.agent3 channels. The byte received, one of several predefined byte constants for internal communication and contained in the intcmds library, is analyzed for further action. If the Agent signals that it is finished using an established link, Manager immediately sends a link release command to the pipe output buffer. Source and destination parameters with the link release are generated by the Manager since it knows which Agent is notifying completion (thus its link designation) and the destination to which it is still attached.
Similarly, if the Agent reports a failed communication the Manager generates a link abort command along with source and destination byte and sends the command to the output buffer. For testing purposes, an additional command was added allowing an Agent to report its completion of data communications, thus allowing performance timers to record processing time.

Finally, if the Manager receives a byte other than one of the predefined commands, the integer translation of that byte is interpreted to be a request for a connection to that link. Manager immediately sends a link request command, source and destination to the output buffer, and records the destination for use in generating subsequent commands.

b. Action For Input From Link Command Pipe

If neither Agent is communicating with the Manager, a three byte link command will be taken from the pipe input when present. The command byte is analyzed to determine action to be taken, if any. A link request or release is simply passed along since it has not yet reached its final destination; the Controller.

When a link acknowledge is received, the Manager determine its relevance to this TRAM by checking the source and destination bytes. If the destination byte matches a local link, that User is informed to expect receipt of data. The destination byte is then set to byte.nil for future reference and removal by a subsequent TRAM. If the source byte is local then the appropriate Agent is informed that a requested link is complete and to begin transmitting. Likewise, the source byte is then set to byte.nil. Finally, both source and destination are checked for content. If both are byte.nil, the entire three byte command is removed from the pipe. Note: this occurs even if both source and destination are local links, that is, the TRAMs link0 and link3 are connected.
Receipt of a link break is relevant to a TRAM only if the source byte is local. (The destination code in the User module which receives data input always monitors its channel, and need not be notified of connection creation or termination.) First, the local variable holding the address of the destination is set to byte.nil to update current status of connections. Secondly, the appropriate Agent is informed that the previous connection has been terminated and a new connection may be requested if desired. If either local link is involved, the command is consumed here, otherwise it is passed along the control pipe.

If a link abort command is received, the command is immediately passed on if neither local link involved. Since aborts are originated by the source, the command is removed by the source upon its return, thus ensuring that both the Controller and the affected destination have seen the abort message. If a local link is the cited destination, the appropriate User is informed of the failed link by passing an integer along the channel used by the InputOrFail.c procedure to trigger a forced termination. A link reinitialization will then be received, which informs the Agent and/or the User involved to reinitialize the affected link engine by executing the Reinitialise procedure. As with the acknowledge command, link bytes corresponding to local links are set to byte.nil before being passed on. If both source and destination are byte.nil, indicating the command has served its purpose, the command is removed from the pipe.

Finally, other bytes passed along for testing or troubleshooting purposes are passed along without action, to be removed by either the Controller or by system monitoring code on the B004.
2. Agent Process

The name *Agent* was chosen for this process to signify its role: to act on behalf of the *User* in obtaining a connection and sending data. Once the *Agent* has instructions from the *User*, the *User* proceeds with the application while the *Agent* waits for the connection, passes the data block or recovers from a failed link.

Like the *Manager*, the *Agent* uses several internal channels to communicate with the other TRAM processes, but only one external channel which outputs data to the destination via crossbars. *Agent* also receives a constant integer denoting its link designation. Header code for *Agent0* is shown; code for *Agent3* is identical after changing the "0" identification to "3".

```
PROC agentO (CHAN OF ANY mgr.to.agentO,
             mgr.from.agentO, agentO.to.userO,
             agentO.from.userO, linkO.out,
             VAL INT link.desig)
```

Upon commencing execution, the *Agent* informs its *User* that it is ready to receive data for outputting. *Agent* then enters an infinite loop which monitors input with a PRI ALT from two sources: *Manager* or its respective *User*. The sequential actions from both code blocks is interleaved by message passing via internal channels. After initialization, the *Agent* sequentially: (1) receives data and destination from *User*, (2) requests a link connection via *Manager*, (3) receives notification of link connection and (4) passes data. If the connection is bad, the *Agent*: (1) informs *Manager* of a problem and recovers, (2) receives notification of link termination, (3) and finally informs *User* that it is ready to receive the next data and destination block. Note that the requested destination address is embedded as the first byte in the data byte received from the *User*.

Use of the reinitialization procedures requires that data be passed as a block of bytes. However, Occam contains numerous retyping operators to convert a collection of bytes to and from other data types such as integers, reals, etc.
In the event of failed communication, the `OutputOrFail.t` procedure returns a boolean true in the `aborted` variable when the watchdog timer expires. This causes program execution to enter an IF construct which: (1) informs the `Manager` of link failure, (2) receives return confirmation then (3) proceeds to reinitialize the failed link. If the communication terminates without failure, the `Manager` is informed of the request that the connection be terminated.

3. User Process

The third member process on each TRAM is the `User`, which completes the relationships and interactions described above. `User` code also contains all application code to be executed in the message exchange. For this analysis, dummy code was inserted to simulate a load on each TRAM processor and the resulting influence on system efficiency. That is, as the `User` module consumes more effort processing data, fewer requests for communications will be issued in a given time period. Conversely, `User` workload influences the performance of the other modules on the TRAM by taking more time slices to complete processing.

`User` header code follows the patterns above with internal channels, one external data input channel, and two constant integers.

```proc
PROC user0 (CHAN OF ANY agent0.to.user0, 
    agent0.from.user0, mgr.from.user0, 
    link0.in, 
    CHAN OF INT mgr.to.user0, 
    VAL INT link.desig, no.trams)
```

`User` code consists of two WHILE TRUE blocks running in parallel. The first block is sequential code which manages data input similar to the output code used in the `Agent`. However, the `User` fault tolerant routines rely on external notification of a fault to terminate a failed communication by using the `InputOrFail.c` procedures. This notification is received from the `Manager` via the integer channel `mgr.to.user`. Accordingly, the
input link must allow for reinitialization with or without data being received along the link at the time of the failure. (Recall that in order to recover from a link fault, both input and output link engines on both ends of communications must be reset). Therefore, depending upon the command received from Manager, the link may be reinitialized at different points in the code.

Application code is running in the second parallel block, which repeats a predefined number of times for testing purposes, each repetition generating and sending a new data block. This includes manipulation of data received, generation of new data, calculations and determination of destination to receive data. For this analysis various routines were used in different test runs to select only like-links or unlike-links, as well as simulating different degrees of load to be processed on the TRAM. A random number generator with a seed based on processor clock time was used to select the next destination when complete randomness was tested. Random numbers were generated until an acceptable link designation was established, which was then sent as a link request.

4. TRAM Code Configuration

To establish priority grouping of the five TRAM processes, the configuration code consists of a PRI PAR with a high priority PAR block nested within. The high priority block includes the processes which are crucial to external channel communication: Manager and Agents. The low priority block for applications contains both User processes. Although this may appear at first glance to be counterproductive, significant gains are made in system performance by ensuring link control functions are conducted expeditiously. With different priorities, application code may wait unnecessarily while requested links are being created and terminated if there are excessive delays along the command pipe and output data links.
Processor configuration code also includes declaration of all internal channels used between the processes. Placement of channel names within parameter listings for each process connects the correct processes together. Also, integer constants are created in the configuration parameters. When assigning link designations to link3 processes, 20 is added to the slot.desig value received by the TRAM from global configuration code.

5. Global Configuration

At the global configuration level, all external links are assigned to processor code, and processor code assigned to processors. As mentioned before, the TRAM code is replicated to each T800 processor on the B012. Code replication is performed in the global configuration section.

Two major blocks of code must be assigned to two different types of processors. Controller code is assigned to the crossbar controlling T212 Transputer, and TRAM code is assigned to one or more T800s on the B012 motherboard. (Code for the monitor code on the B004 and its T414 is handled entirely separate from the message exchange code in an EXE fold). Transputer type is signified by the keyword T2 for T212 Transputer and T8 for a T800. This lets the compiler know which libraries are to be used in creating compiled code for each processor. To assign links, a set of constants is defined in the library links which also contains the constant no.trams. Link assignment matches the processor code parameter with the hardware link engine on each Transputer. Controller code header shown below is preceded by a one-to-one matching of software channel
name with hardware link component with the **PLACE ... AT** construct. The channel names are then matched to the procedure header definition in the local *Controller* code previously defined.

**PLACED PAR**

**PROCESSOR no.trams T2**

PLACE pipe[0] AT link2in :
PLACE pipe[no.trams+1] AT link1out :
PLACE t2.to.c0 AT link0out :
PLACE t2.from.c0 AT link0in :
PLACE t2.to.cl AT link3out :
PLACE t2.from.cl AT link3in :

controller (pipe[0], pipe[no.trams+1],
            t2.to.c0, t2.from.c0,
            t2.to.cl, t2.from.cl, no.trams)

Similarly, the TRAM code parameters are assigned to hardware links, but a replication process is used to assign the same code with slightly different parameters to each T800. This is done by using an array of channels and carefully matching the correct array index with the replicated parameters. For example, to create the pipe each **link1out** of a given index value [slot] is physically connected to the next corresponding **link2in** of index value [slot+1]. This technique informs the software of the actual connections made with the B012 hardware in a single statement instead of having to itemize the links of each processor. Each processor is also given a designation based on the incremented value of *slot*. Processor identification numbers are 100, 200, 300, ... for the T800 in slots zero, one, two, etc. Since *slot* varies for each replication, each T800 processor receives a different constant value for its slot designation and subsequent link designations.
PLACED \text{PAR} \ slot = 0 \ FOR \ (\text{no.trams})

\begin{verbatim}
PROCESSOR \ ((slot+1) \ast 100) \ T8
\hspace{1em} \text{PLACE} \ \text{pipe}[slot] \hspace{1em} \text{AT} \ \text{link1out} \\
\hspace{1em} \text{PLACE} \ \text{pipe}[slot+1] \hspace{1em} \text{AT} \ \text{link2in} \\
\hspace{1em} \text{PLACE} \ \text{link0.from.c1[slot]} \hspace{1em} \text{AT} \ \text{link0in} \\
\hspace{1em} \text{PLACE} \ \text{link0.to.c0[slot]} \hspace{1em} \text{AT} \ \text{link0out} \\
\hspace{1em} \text{PLACE} \ \text{link3.from.c0[slot]} \hspace{1em} \text{AT} \ \text{link3in} \\
\hspace{1em} \text{PLACE} \ \text{link3.to.c1[slot]} \hspace{1em} \text{AT} \ \text{link3out} \\
\end{verbatim}

\text{tram.code} (\text{pipe}[slot], \text{pipe}[slot+1], \text{link0.from.c1[slot]}, \text{link0.to.c0[slot]}, \text{link3.from.c0[slot]}, \text{link3.to.c1[slot]}, \text{slot, no.trams})

As can be seen, all channel names used are declared strictly for the global configuration section, particularly those which are declared as arrays of channels. Correspondence with local channel declarations of the procedures themselves is achieved by correct placement of the parameters listed with the procedure call.
V. MESSAGE EXCHANGE PERFORMANCE EVALUATION

Numerous test runs of the message exchange were conducted with variations in program structure and parameters. In general, each run consisted of 1000 link cycles (link control activity necessary to establish and terminate one link) performed back-to-back by each User. With four TRAMs in the system, a total of 8000 link cycles were completed in each run. The destination is chosen at random for each link cycle. In the unrestricted case, a User chooses from a set of seven possible destinations for each link cycle (excluding itself from the complete set of eight Users). When performance of only like-link or only dislike-link connections was studied, the User chose from a set of three possible destinations. All selections were uniformly distributed across the possible destinations available. Measurements include total time to completion, number of requests, acknowledges, breaks, aborts, and reinitializes needed. For various tests the system was run either loaded (with work simulated in the User process by adding a 10,000 iteration SKIP) or unloaded and thus only performing link control and data passing activity.

In the tables below each measurement is the average of five runs under the same conditions. Individual runs vary due to the random nature of the link destination requests, however, variations between runs are minor. Faults are introduced by providing only one physical crossbar to crossbar edge connector for each like-link connection: one cable for link0 to link0 connections and one cable for link3 to link3 connections. In all fault-test cases the library tables incorrectly state that all 16 edge connector cables are available, therefore, faults are detected when the nonexistent cables are used. The two cables actually connected are the last two in the list, so the previous 14 must be identified.
as faulty before the good cables are found. Additional faults are caused as known failed cables are arbitrarily reset in hopes of finding repaired cables. Also, additional link control commands are needed to accommodate simultaneous like-link requests since only one can be fulfilled at a time with the other request being recycled.

A. DATA TRANSFER VARIATIONS AND SYSTEM EFFICIENCY

1. Data Block Size

Significant variations in system performance occurs due to data block size. Data block size must be defined at compile time due to the requirements of the link failure and recovery library procedures. Once selected, the data block size remains fixed for the duration of program execution. Therefore, if the application program expects to transmit variable length data between nodes, the block size must be chosen to allow the maximum expected data length, with extra bytes in shorter messages being discarded. Test measurements were collected with data block size varying from two to 8192 bytes in factors of two. The upper limit of 8192 bytes was reached due to the memory available on each TRAM (36 kilobytes total of on and off-chip RAM) and the declaration of arrays for both output and input data blocks by each user.

Tables 5.1 and 5.2 provide the results of testing the unloaded system with and without faults, and Tables 5.3 and 5.4 with simulated system load. Time to complete an 8000 request cycle run increases with data block size, but not significantly until block size reaches 64 bytes in the unloaded runs and 128 bytes loaded. The relatively consistent execution times for smaller data block sizes represents the time to route link control information and perform the action required for link operations. Actual processing and data passing performed by Users easily fit within the gaps formed by high priority Agent and Manager processes concentrating on link control.
Raw data measured and listed below includes execution time, number of requests and total link control commands used as averaged over five runs of identical parameters. Calculations which compare number of data bytes take into account the data block size and the number of data blocks passed: 8000 in all runs. Calculations involving link control bytes take into account that each command consists of three bytes and the number of link control commands issued (either requests only or the sum of all six command types, as specified). Calculated figures listed include microsecond per data byte passed, average number of requests needed for the per data byte passed, data transfer rate in kilobytes per second (inverse of microseconds per data byte, adjusted for change in units), a ratio of data bytes to control bytes, and control communications overhead as a percentage of link control bytes passed of the total bytes passed in the system.

The single T212 link controller also represents a limiting factor, particularly when processing a request for a like-link. During the T212’s execution of sequential link manipulation code no additional link control commands are received causing the pipe to fill. As each TRAM’s buffer fills, no more control commands can be submitted or received by the Managers on subsequent TRAMs. This ripple effect continues and prevents Agents and Users from proceeding to the next data transmission. Although it may be possible to implement the T212 code as a collection of parallel processes with one being an input buffer, such constructs consume considerable amounts of memory. With only two kilobytes of on-chip RAM and no off-chip RAM available, every effort was made to minimize T212 code size.
### TABLE 5.1

**SYSTEM PERFORMANCE DATA: UNLOADED, NO FAULTS**

<table>
<thead>
<tr>
<th>Data Block Size</th>
<th>Time (Sec)</th>
<th>Number Of REQs</th>
<th>Total Link Control Cmds</th>
<th>uSec Per Data Byte</th>
<th>REQs Per Data Byte</th>
<th>Data Xfer Rate KB/Sec</th>
<th>Data To Control Ratio</th>
<th>Comms Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3.56</td>
<td>24,266</td>
<td>48,266</td>
<td>222.65</td>
<td>1.517</td>
<td>4.4</td>
<td>0.33</td>
<td>90.05%</td>
</tr>
<tr>
<td>4</td>
<td>3.56</td>
<td>24,287</td>
<td>48,287</td>
<td>111.24</td>
<td>0.759</td>
<td>8.8</td>
<td>0.66</td>
<td>81.91%</td>
</tr>
<tr>
<td>8</td>
<td>3.56</td>
<td>24,269</td>
<td>48,265</td>
<td>55.63</td>
<td>0.379</td>
<td>17.6</td>
<td>1.33</td>
<td>69.35%</td>
</tr>
<tr>
<td>16</td>
<td>3.63</td>
<td>24,788</td>
<td>48,788</td>
<td>28.32</td>
<td>0.194</td>
<td>34.5</td>
<td>2.62</td>
<td>53.35%</td>
</tr>
<tr>
<td>32</td>
<td>3.68</td>
<td>25,329</td>
<td>49,329</td>
<td>14.36</td>
<td>0.099</td>
<td>68.0</td>
<td>5.19</td>
<td>36.63%</td>
</tr>
<tr>
<td>64</td>
<td>3.74</td>
<td>26,322</td>
<td>50,322</td>
<td>7.31</td>
<td>0.051</td>
<td>133.6</td>
<td>10.17</td>
<td>22.77%</td>
</tr>
<tr>
<td>128</td>
<td>3.94</td>
<td>28,468</td>
<td>52,468</td>
<td>3.85</td>
<td>0.028</td>
<td>253.6</td>
<td>19.52</td>
<td>13.32%</td>
</tr>
<tr>
<td>256</td>
<td>4.29</td>
<td>32,286</td>
<td>56,286</td>
<td>2.10</td>
<td>0.016</td>
<td>466.2</td>
<td>36.39</td>
<td>7.62%</td>
</tr>
<tr>
<td>512</td>
<td>5.09</td>
<td>38,330</td>
<td>63,958</td>
<td>1.24</td>
<td>0.009</td>
<td>786.3</td>
<td>64.04</td>
<td>4.47%</td>
</tr>
<tr>
<td>1024</td>
<td>7.29</td>
<td>58,299</td>
<td>82,299</td>
<td>0.89</td>
<td>0.007</td>
<td>1097.1</td>
<td>99.54</td>
<td>2.93%</td>
</tr>
<tr>
<td>2048</td>
<td>11.90</td>
<td>91,136</td>
<td>114,736</td>
<td>0.73</td>
<td>0.006</td>
<td>1344.1</td>
<td>142.80</td>
<td>2.06%</td>
</tr>
<tr>
<td>4096</td>
<td>21.41</td>
<td>157,309</td>
<td>181,309</td>
<td>0.65</td>
<td>0.005</td>
<td>1494.5</td>
<td>180.73</td>
<td>1.63%</td>
</tr>
<tr>
<td>8192</td>
<td>40.44</td>
<td>289,091</td>
<td>313,094</td>
<td>0.62</td>
<td>0.004</td>
<td>1582.8</td>
<td>209.32</td>
<td>1.41%</td>
</tr>
</tbody>
</table>

### TABLE 5.2

**SYSTEM PERFORMANCE DATA: UNLOADED, WITH FAULTS**

<table>
<thead>
<tr>
<th>Data Block Size</th>
<th>Time (Sec)</th>
<th>Number Of REQs</th>
<th>Total Link Control Cmds</th>
<th>uSec Per Data Byte</th>
<th>REQs Per Data Byte</th>
<th>Data Xfer Rate KB/Sec</th>
<th>Data To Control Ratio</th>
<th>Comms Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3.75</td>
<td>24,993</td>
<td>49,044</td>
<td>234.21</td>
<td>1.562</td>
<td>4.2</td>
<td>0.33</td>
<td>90.19%</td>
</tr>
<tr>
<td>4</td>
<td>3.73</td>
<td>25,107</td>
<td>49,154</td>
<td>116.66</td>
<td>0.785</td>
<td>8.4</td>
<td>0.65</td>
<td>82.17%</td>
</tr>
<tr>
<td>8</td>
<td>3.73</td>
<td>24,872</td>
<td>49,121</td>
<td>58.31</td>
<td>0.389</td>
<td>16.7</td>
<td>1.30</td>
<td>69.72%</td>
</tr>
<tr>
<td>16</td>
<td>3.80</td>
<td>25,583</td>
<td>49,631</td>
<td>29.70</td>
<td>0.200</td>
<td>32.9</td>
<td>2.58</td>
<td>53.77%</td>
</tr>
<tr>
<td>32</td>
<td>3.86</td>
<td>26,258</td>
<td>50,306</td>
<td>15.09</td>
<td>0.103</td>
<td>64.7</td>
<td>5.09</td>
<td>37.09%</td>
</tr>
<tr>
<td>64</td>
<td>3.94</td>
<td>27,355</td>
<td>51,437</td>
<td>7.70</td>
<td>0.053</td>
<td>126.8</td>
<td>9.95</td>
<td>23.16%</td>
</tr>
<tr>
<td>128</td>
<td>4.14</td>
<td>29,247</td>
<td>53,441</td>
<td>4.04</td>
<td>0.029</td>
<td>241.4</td>
<td>19.16</td>
<td>13.54%</td>
</tr>
<tr>
<td>256</td>
<td>4.52</td>
<td>33,452</td>
<td>57,511</td>
<td>2.01</td>
<td>0.016</td>
<td>442.1</td>
<td>35.61</td>
<td>7.77%</td>
</tr>
<tr>
<td>512</td>
<td>5.33</td>
<td>41,725</td>
<td>65,786</td>
<td>1.30</td>
<td>0.010</td>
<td>750.4</td>
<td>62.26</td>
<td>4.60%</td>
</tr>
<tr>
<td>1024</td>
<td>7.62</td>
<td>60,629</td>
<td>84,699</td>
<td>0.93</td>
<td>0.007</td>
<td>1050.3</td>
<td>96.72</td>
<td>3.01%</td>
</tr>
<tr>
<td>2048</td>
<td>12.06</td>
<td>93,171</td>
<td>117,245</td>
<td>0.74</td>
<td>0.006</td>
<td>1327.2</td>
<td>139.74</td>
<td>2.10%</td>
</tr>
<tr>
<td>4096</td>
<td>22.08</td>
<td>166,511</td>
<td>190,620</td>
<td>0.67</td>
<td>0.005</td>
<td>1449.1</td>
<td>171.90</td>
<td>1.72%</td>
</tr>
<tr>
<td>8192</td>
<td>42.01</td>
<td>312,722</td>
<td>334,880</td>
<td>0.64</td>
<td>0.005</td>
<td>1523.4</td>
<td>195.70</td>
<td>1.51%</td>
</tr>
</tbody>
</table>
### TABLE 5.3

**SYSTEM PERFORMANCE DATA: LOADED, NO FAULTS**

<table>
<thead>
<tr>
<th>Data Block Size</th>
<th>Time (Sec)</th>
<th>Number Of REQs</th>
<th>Total Link Control Cmds</th>
<th>uSec Per Data Byte</th>
<th>REQs Per Data Byte</th>
<th>Data Xfer Rate KB/Sec</th>
<th>Data To Control Overhead</th>
<th>Comms Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>14.36</td>
<td>8,941</td>
<td>32,941</td>
<td>897.35</td>
<td>0.559</td>
<td>1.1</td>
<td>0.49</td>
<td>86.07%</td>
</tr>
<tr>
<td>4</td>
<td>14.36</td>
<td>8,849</td>
<td>32,849</td>
<td>448.66</td>
<td>0.277</td>
<td>2.2</td>
<td>0.97</td>
<td>75.49%</td>
</tr>
<tr>
<td>8</td>
<td>14.37</td>
<td>8,905</td>
<td>32,905</td>
<td>224.54</td>
<td>0.139</td>
<td>4.3</td>
<td>1.95</td>
<td>60.67%</td>
</tr>
<tr>
<td>16</td>
<td>14.41</td>
<td>9,156</td>
<td>33,156</td>
<td>112.54</td>
<td>0.072</td>
<td>8.7</td>
<td>3.86</td>
<td>43.73%</td>
</tr>
<tr>
<td>32</td>
<td>14.45</td>
<td>9,116</td>
<td>33,116</td>
<td>56.43</td>
<td>0.036</td>
<td>17.3</td>
<td>7.73</td>
<td>27.96%</td>
</tr>
<tr>
<td>64</td>
<td>14.55</td>
<td>9,304</td>
<td>33,304</td>
<td>28.41</td>
<td>0.018</td>
<td>34.4</td>
<td>15.37</td>
<td>16.33%</td>
</tr>
<tr>
<td>128</td>
<td>14.74</td>
<td>9,647</td>
<td>33,647</td>
<td>14.40</td>
<td>0.009</td>
<td>67.8</td>
<td>30.43</td>
<td>8.97%</td>
</tr>
<tr>
<td>256</td>
<td>15.16</td>
<td>10,604</td>
<td>34,604</td>
<td>7.40</td>
<td>0.005</td>
<td>131.9</td>
<td>59.18</td>
<td>4.82%</td>
</tr>
<tr>
<td>512</td>
<td>16.09</td>
<td>14,556</td>
<td>38,556</td>
<td>3.93</td>
<td>0.004</td>
<td>248.6</td>
<td>106.24</td>
<td>2.75%</td>
</tr>
<tr>
<td>1024</td>
<td>17.69</td>
<td>17,180</td>
<td>41,580</td>
<td>2.16</td>
<td>0.002</td>
<td>452.1</td>
<td>197.02</td>
<td>1.50%</td>
</tr>
<tr>
<td>2048</td>
<td>21.70</td>
<td>38,210</td>
<td>62,210</td>
<td>1.32</td>
<td>0.002</td>
<td>737.3</td>
<td>263.37</td>
<td>1.13%</td>
</tr>
<tr>
<td>4096</td>
<td>30.07</td>
<td>85,155</td>
<td>109,155</td>
<td>0.92</td>
<td>0.003</td>
<td>1064.1</td>
<td>300.20</td>
<td>0.99%</td>
</tr>
<tr>
<td>8192</td>
<td>48.01</td>
<td>201,155</td>
<td>225,155</td>
<td>0.73</td>
<td>0.003</td>
<td>1333.1</td>
<td>291.07</td>
<td>1.02%</td>
</tr>
</tbody>
</table>

### TABLE 5.4

**SYSTEM PERFORMANCE DATA: LOADED, WITH FAULTS**

<table>
<thead>
<tr>
<th>Data Block Size</th>
<th>Time (Sec)</th>
<th>Number Of REQs</th>
<th>Total Link Control Cmds</th>
<th>uSec Per Data Byte</th>
<th>REQs Per Data Byte</th>
<th>Data Xfer Rate KB/Sec</th>
<th>Data To Control Overhead</th>
<th>Comms Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>14.38</td>
<td>9,236</td>
<td>33,303</td>
<td>898.88</td>
<td>0.577</td>
<td>1.1</td>
<td>0.48</td>
<td>86.20%</td>
</tr>
<tr>
<td>4</td>
<td>14.39</td>
<td>9,201</td>
<td>33,266</td>
<td>449.58</td>
<td>0.288</td>
<td>2.2</td>
<td>0.96</td>
<td>75.72%</td>
</tr>
<tr>
<td>8</td>
<td>14.40</td>
<td>9,180</td>
<td>33,249</td>
<td>224.93</td>
<td>0.143</td>
<td>4.3</td>
<td>1.92</td>
<td>60.92%</td>
</tr>
<tr>
<td>16</td>
<td>14.43</td>
<td>9,239</td>
<td>33,302</td>
<td>112.74</td>
<td>0.072</td>
<td>8.7</td>
<td>3.84</td>
<td>43.84%</td>
</tr>
<tr>
<td>32</td>
<td>14.48</td>
<td>9,439</td>
<td>33,507</td>
<td>56.54</td>
<td>0.037</td>
<td>17.3</td>
<td>7.64</td>
<td>28.20%</td>
</tr>
<tr>
<td>64</td>
<td>14.57</td>
<td>9,510</td>
<td>33,577</td>
<td>28.45</td>
<td>0.019</td>
<td>34.3</td>
<td>15.25</td>
<td>16.44%</td>
</tr>
<tr>
<td>128</td>
<td>14.77</td>
<td>9,926</td>
<td>34,001</td>
<td>14.4</td>
<td>0.010</td>
<td>67.7</td>
<td>30.12</td>
<td>9.06%</td>
</tr>
<tr>
<td>256</td>
<td>15.20</td>
<td>11,122</td>
<td>35,202</td>
<td>7.42</td>
<td>0.005</td>
<td>131.6</td>
<td>58.18</td>
<td>4.90%</td>
</tr>
<tr>
<td>512</td>
<td>16.14</td>
<td>15,246</td>
<td>39,337</td>
<td>3.94</td>
<td>0.004</td>
<td>247.8</td>
<td>104.13</td>
<td>2.80%</td>
</tr>
<tr>
<td>1024</td>
<td>17.83</td>
<td>19,762</td>
<td>43,254</td>
<td>2.18</td>
<td>0.002</td>
<td>448.8</td>
<td>189.39</td>
<td>1.56%</td>
</tr>
<tr>
<td>2048</td>
<td>21.81</td>
<td>39,840</td>
<td>63,338</td>
<td>1.33</td>
<td>0.002</td>
<td>733.6</td>
<td>258.68</td>
<td>1.15%</td>
</tr>
<tr>
<td>4096</td>
<td>30.44</td>
<td>91,405</td>
<td>115,533</td>
<td>0.93</td>
<td>0.003</td>
<td>1051.3</td>
<td>283.62</td>
<td>1.05%</td>
</tr>
<tr>
<td>8192</td>
<td>49.29</td>
<td>221,866</td>
<td>246,043</td>
<td>0.75</td>
<td>0.003</td>
<td>1298.4</td>
<td>266.36</td>
<td>1.11%</td>
</tr>
</tbody>
</table>
Also of concern is the B004 and T414 monitor Transputer. Excessive work by this Transputer would also have a slowing influence on the link control pipe. Several variations of B004 code were used with widely varying degrees of complexity. Timing differences between the simplest T414 code and the monitor program used were negligible. Also, both input and output buffers were added as part of the T414 code to help smooth its presence on the pipe.

**Figure 5.1** shows the effect of data block size on execution time. All four combinations of load and fault status are shown. Note the minimal difference between the runs with and without faults. With any number of TRAMs in the system, the minimum number of good edge connectors needed is two; one for each type of like-link. Less than one link would result in failure to pass any data between like-links, thus causing affected processes to hang. However, with four TRAMs the minimum needed to obtain fault free transmission is four, two per like-link type. Therefore, the differences in the number of edge connectors in these runs is only a factor of two, and does not significantly stress the system since the simultaneous need for two like-link connections is relatively rare.

As data block size increases beyond 128 bytes the control pipe and T212 are no longer bottlenecks. Sufficient time is spent by each node in creating and passing data that link control commands enter the pipe with minimal restriction. Consequently, links are terminated in a timely manner allowing the next request for those resources to be fulfilled in the first attempt without recycling the link request. Worst case performance occurs with the smallest data block size and unloaded User processes. Since no application work is done by unloaded Users the performance data is used for comparison as a baseline.
Tables 5.1 through 5.4 show increases in data communication rate with block size, but the rate does not double as block size. Although more data is transferred in larger blocks, the system must work harder to establish the requested connections since resources are held longer. Without a quick turnover of resources to the next requesting node, link control information is recycled back into the pipe for a later try. The number of link requests increases dramatically with block size showing the difficulty in achieving the requested connection. Therefore, greater link control overhead reduces system efficiency when block size becomes too large.
One way to show this is to compare data bytes versus control bytes, noting that each control command consists of three bytes. Figure 5.2 shows the data byte-to-control byte ratio and the loss of communication efficiency encountered when data block size becomes very large. In the unloaded tests, as the block size is increased beyond 1024 bytes the data-to-control ratio (and hence, system efficiency) still increases but less rapidly than with block sizes below 1024 bytes. As data block size is increased beyond 4096 bytes, the data-to-control ratio actually decreases. The loaded case shows a similar effect, however, since fewer control communications are needed the decrease in efficiency will take place at higher data block sizes than practical to test.

![Data-To-Control Communication Ratio](image)

**Figure 5.2. Data-To-Control Communication Ratio**
Efficiency comparisons using control and data communications are not the best indicator of performance since each link's DMA engine proceeds with only minimal effort of the system CPUs. Although the use of DMAs is especially beneficial in passing large data blocks, system burden increases when all CPUs in the control pipe are passing small commands. Using the execution time of the unloaded runs with small data block size as a base time gives a better comparison of system overhead resulting from link control operations. As discussed above, no significant change in execution time occurs until block size exceed 64 bytes. Therefore, execution time of small block size runs is predominantly a measure of link control effort of the system. By averaging the execution times for two, four, and eight byte block sizes of the unloaded runs a base execution time for the system is established.

Figure 5.3 shows a ratio of base execution time versus execution time of larger block sizes in both loaded and unloaded runs. Worst case, of course, is small block size on an unloaded system. Control overhead drops off sharply as block size is increased beyond 128 bytes. In the loaded system, overhead starts at about 25% and decreases slowly as block size increases until approaching the unloaded overhead at a about 8% with a 8192 data block size. Control overhead for systems with load less than the test load will plot between the test load and the unloaded system results. Only measurements of runs without faults are shown for clarity.

2. Data Routing Via Multiple Crossbars

Separating each TRAMs data links into individual User processes allowed analysis of the affects of different routing paths. Due to the wiring arrangement between the crossbars and the TRAMs, the methods used to connect like-links is significantly more complex than connecting a link0 to a link3. Additionally, the slowing influence associated with multiple C004 crossbars is a concern in like-link connections.
Three sets of runs were conducted under varying constraints limiting the routing assignments. For each of three data block sizes: 8192, 1024, and 128 bytes, a set of runs were conducted under normal routing (uniformly selected from all seven possible destinations), with only like-link routing and with only dislike-link routing (each uniformly selected from the appropriate subset consisting of three possible destinations). Table 5.5 below shows the results.

As can be seen from the data, the large data block size of 8192 blocks causes an increase in required time when routing is restricted to only like-link (two connections per crossbar) as compared to the case in which routing is restricted to only dislike-links (one connection per crossbar). This result is not as pronounced in the 1024 byte block and...
even less with 128 byte block. Figure 5.4 shows a comparison of the three link restrictions and three data block sizes, normalized to the random link type of each block size.

**TABLE 5.5**

**LINK ROUTING ANALYSIS RESULTS**

<table>
<thead>
<tr>
<th>Parameter Measured</th>
<th>Link Routing Restriction</th>
<th>MESSAGE SIZE: 8192</th>
<th>MESSAGE SIZE: 1024</th>
<th>MESSAGE SIZE: 128</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0 to 3</td>
<td>3 to 0</td>
<td>0 to 0</td>
</tr>
<tr>
<td>Execution Time (seconds)</td>
<td></td>
<td>45.706</td>
<td>48.137</td>
<td>49.023</td>
</tr>
<tr>
<td>Number of Requests</td>
<td></td>
<td>160,311</td>
<td>203,743</td>
<td>216,073</td>
</tr>
<tr>
<td>Total Control Commands</td>
<td></td>
<td>184,311</td>
<td>227,743</td>
<td>239,473</td>
</tr>
<tr>
<td>Time(usec) / Data Byte</td>
<td></td>
<td>0.6974</td>
<td>0.7345</td>
<td>0.7480</td>
</tr>
<tr>
<td>Requests / Data Byte</td>
<td></td>
<td>0.0024</td>
<td>0.0031</td>
<td>0.0033</td>
</tr>
<tr>
<td>Kilobytes Data / Second</td>
<td></td>
<td>1400.3</td>
<td>1329.5</td>
<td>1305.5</td>
</tr>
<tr>
<td>Number of Requests</td>
<td></td>
<td>25,389</td>
<td>25,617</td>
<td>25,427</td>
</tr>
<tr>
<td>Time(usec) / Data Byte</td>
<td></td>
<td>0.0092</td>
<td>0.0094</td>
<td>0.0092</td>
</tr>
<tr>
<td>Kilobytes Data / Second</td>
<td></td>
<td>67.7</td>
<td>67.8</td>
<td>67.3</td>
</tr>
</tbody>
</table>
B. TRAM CODE STRUCTURE

Considerable influence upon performance can result from slight changes in an Occam program structure. Timing and CPU utilization are changed when different priorities are assigned to parallel (PAR) processes. The TRAM code is subject to these considerations and various configurations and their effects are discussed below.

As shown earlier, each TRAM contains five processes executing in parallel and communicating with each other and with other processors (see Figure 3.7). Atkin [Ref 10:p. 12] stressed that efficient code must be structured to separate as much as practical those processes performing communications from those performing calculation.
This is accomplished in the TRAM by placing all application calculation in only the User modules. When data is to be passed to an external destination the data is passed to a high priority process dedicated to outputting that block along a link.

Communications processes should be run in high priority to help ensure that the communications themselves do not become a bottleneck in holding up information flow. Prompt initiation of communications also sets in action the DMA link engines which can then release the processor to perform other work. Given this guidance, it is clear the Manager process must be run at high priority to handle all link control communications and the User processes must be run at low priority to perform application work and data input. Different configurations were tested adjusting these priorities. Performance of the configurations varied widely; some resulted in nonfunctioning programs due to communications starvation. For example, running the User processes in high priority and the Agent processes at low priority resulted in deadlock.

Three configurations were tested and compared under unloaded and loaded conditions using a 1024 data block size. The normal case consisted of the process priority as listed in the presented code. Case I and Case II are listed below:

<table>
<thead>
<tr>
<th>Normal:</th>
<th>Case I:</th>
<th>Case II:</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRI PAR</td>
<td>PRI PAR</td>
<td>PAR -- low</td>
</tr>
<tr>
<td>PAR -- hi</td>
<td>manager () -- hi</td>
<td>manager ()</td>
</tr>
<tr>
<td>manager ()</td>
<td>agent0 () -- low</td>
<td>agent0 ()</td>
</tr>
<tr>
<td>agent0 ()</td>
<td>agent3 ()</td>
<td>agent3 ()</td>
</tr>
<tr>
<td>agent3 ()</td>
<td>user0 ()</td>
<td>user0 ()</td>
</tr>
<tr>
<td>--low</td>
<td>user3 ()</td>
<td>user3 ()</td>
</tr>
<tr>
<td>user0 ()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>user3 ()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Comparisons of the three cases are shown in Figure 5.5. Separate tests were performed with and without simulated load in the User process. All figures are percentages normalized against the appropriately loaded normal case.
C. LINK FAULT RECOVERY PERFORMANCE

When a link fault occurs a minimum of three additional control commands are issued: link.abt, link.rei and link.ack (see Figure 4.1). If a replacement route is not available either due to lack of “good” edge connectors or all routes are busy, a link.req will cycle in the control pipe as in the non-fault case. If other faulty edge connectors are inadvertently assigned (having not yet been discovered as faulty), the abort message chain will repeat until a good route is found or a cycling request is issued.

After the Controller has determined a valid status for all 16 edge connectors the abort message chain should occur once for each new request resulting in an abort due to a newly created fault. However, as the req.cnt increments when the Controller realizes a
resource shortage, previously identified failed edge connectors are reset allowing them to be reassigned in the hopes of locating repaired cables. Two variables control the periodicity of resetting edge connector status: req.cnt upper limit and delay time value. For this project the req.cnt upper limit was set equal to the number of TRAMs in the system (four) and the value for the delay timer was set to 16000 ticks of the low priority clock, or about one second. Use of the counter prevents unnecessarily resetting edge status without need, and the timer minimizes repeated resettings when the need for new resources is great. Both values can be adjusted accordingly to adapt system performance to the application implemented. For example, a shorter time delay would quicken the Controller's ability to find newly repaired edge connectors.

Execution time affects fault recovery actions since as the system operates longer under fault conditions, more previously identified faulty edge connectors will be reset based on delay time. Data block size does not directly affect the degree of additional workload resulting from the presence of link faults. However, in the test cases run the use of larger data blocks increases the total amount of data transferred, thus increasing total execution time and, consequently, the number of fault recovery communications.
### TABLE 5.6
PERFORMANCE ASPECTS OF FAULT DETECTION AND RECOVERY

<table>
<thead>
<tr>
<th>Data Block Size</th>
<th>System Unloaded</th>
<th>System Loaded</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. Of Aborts</td>
<td>Added Total Comms</td>
</tr>
<tr>
<td>2</td>
<td>17.2</td>
<td>778</td>
</tr>
<tr>
<td>4</td>
<td>15.4</td>
<td>867</td>
</tr>
<tr>
<td>8</td>
<td>16.6</td>
<td>856</td>
</tr>
<tr>
<td>16</td>
<td>15.8</td>
<td>843</td>
</tr>
<tr>
<td>32</td>
<td>16.0</td>
<td>977</td>
</tr>
<tr>
<td>64</td>
<td>18.2</td>
<td>1,115</td>
</tr>
<tr>
<td>128</td>
<td>19.4</td>
<td>974</td>
</tr>
<tr>
<td>256</td>
<td>19.4</td>
<td>1,224</td>
</tr>
<tr>
<td>512</td>
<td>20.4</td>
<td>1,828</td>
</tr>
<tr>
<td>1024</td>
<td>23.2</td>
<td>2,400</td>
</tr>
<tr>
<td>2048</td>
<td>24.8</td>
<td>2,509</td>
</tr>
<tr>
<td>4096</td>
<td>36.2</td>
<td>9,310</td>
</tr>
<tr>
<td>8192</td>
<td>52.6</td>
<td>21,786</td>
</tr>
</tbody>
</table>
VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

By using specialized though off-the-shelf hardware, a dynamically reconfigurable network of Transputers can be constructed. The Transputer's unique link hardware makes this single-chip processor very appropriate for this type of system. Efficiency of the message exchange presented here is largely determined by data block size and the application load on each node. System control overhead is minimized by keeping the link control commands small and infrequent. As a dynamically reconfigurable system, this message exchange evaluated the worst case scenario of requesting and establishing a new connection for each data block passed. A semi-static configuration system, which reconfigures the network topology only at synchronized points during application program execution, would improve system efficiency by reducing overhead at the expense of reconfiguration flexibility.

A concept implemented in hardware should have superior performance over the same concept implemented in software. In this case, use of program controlled crossbars to directly connect communicating nodes should perform comparably well or better than a packet routing system which passes data through intermediaries. However, gains made in the direct routing of data are offset by losses incurred in routing crossbar control information. Utilization of a link control pipe to pass control information from node to node diminishes some of the gains achieved from direct data connection. An ideal correction to this drawback would be to pass link control information point-to-point, comparable to the manner in which data is passed. One method for implementing complete crossbar connectivity is described in the Esprit Project [Ref 24] in which all
four links of each Transputer are connected to large crossbars. Thus far in the Esprit Project semi-static reconfiguration is implemented, and use of specialized hardware allows for increased expandability.

Techniques explored in this paper stress the circuit switching approach made possible with readily available hardware, including program controlled crossbar switches. Considerable commercially available hardware also exists employing packet routing methods, most notably hypercube parallel processors. Hypercube topologies maximize performance by symmetrically interconnecting nodes to minimize the distance between any pair of processors, thus reducing packet routing overhead. A hypercube topology contrasts the message exchange developed here by substituting crossbar control overhead with packet routing overhead. A key parameter affecting performance in both topologies is message size, which is determined entirely by the application to be executed. A performance comparison between the two methods should be conducted by executing identical application code in each topology. Since the number of nodes in a hypercube is equal to two raised to the nth power, where n is the number of links available to each node, a 16 node hypercube could be readily constructed using Transputers. This Transputer hypercube can be evaluated against a 16 node message exchange implemented on a fully populated B012 motherboard.

B. RECOMMENDATIONS FOR FOLLOW-ON WORK

Integrated shipboard weapons systems of the complexity found in AEGIS consist of a collection of specialized computers distributed throughout the weapon platform and communicating with each other as necessary to detect, identify, track, display and direct the weapons' fire towards targets. Although the processing power is distributed among a number of processors, the specialization calls for discrete elements of processing power to be assigned to specific tasks. Communications within each group as well as between
groups need not be completely dynamic nor be able to achieve any arbitrary topology. However, some reconfiguration ability would be invaluable as the system posture changes or in response to faults occurring in the system. A posture change of the weapons system could result from refinement of the tactical environment, for example, as the potential source of the current threat is narrowed to subsurface as opposed to airborne, appropriate computing resources can be directed to concentrate accordingly.

Given this dedicated grouping of computing resources, an architecture similar to the B012 can be employed at each specialized station. A collection of Transputers with dynamic reconfiguration could adapt to system load changes as well as the occurrence of local faults. Interconnection between stations can be achieved by using switchable links of the crossbars accessible at the edge connectors. A hierarchy of control would be established between stations using an inter-B012 link control pipe separate from the internal pipe developed here. Again using the B012 as an example, the TRAM in slot zero can readily be disconnected from normal C004 crossbar connections, thus freeing a significant processing resource to direct interstation communication control.

Figure 6.1 shows an arrangement of B012 stations with an additional link control pipe using the slot0 TRAM of each station. Data connections between stations can be hardwired between the remaining crossbar links not used for the local TRAMs. Assignment of these external links can be weighted in favor of the most needed communications routes, with additional links assigned for redundant communications paths thus allowing for interstation link fault tolerance in a similar manner as demonstrated on the single B012 in this project. Interstation link fault tolerance would become especially valuable when considering various battle damage possibilities throughout the ship as well as equipment failures.
Link control communications overhead will always be a factor limiting efficiency, however, if reconfiguration is a rare event then control signals will be minimized as compared to a completely dynamic and therefore worst case system in terms of overhead. Also, if the processing power of a given station must be increased, or if more external links must be added, two or more B012s can be arranged head-to-tail to extend the local link control pipe thus increasing the number of local TRAMS as well as crossbars.
Although Transputers are an extremely capable computer component, the above discussion should not be taken to imply that a complex integrated weapons system can or should be implemented in the off-the-shelf technology as described in this thesis. However, simulations and modelling of system functions and interactions can be implemented as described. Link connections can relay simulated sensor information as input to one station, environmental and real-world simulation to another, weapons response to a third, and so on. Additional stations can output commands to weapons’ control as well as tactical displays and communications. As a modelling tool, Transputers can be used to test and optimize system software and hardware topologies, and to identify and test the specialized hardware necessary to meet the high performance needs of a modern weapons system.
APPENDIX A
MESSAGE EXCHANGE LIBRARIES AND SYSTEM CODE

-- DATA LIBRARIES used in B012 Message Exchange
-- All libraries stored in same directory

---- links - defines Transputer links and # of TRAMS used ----

VAL link0out IS 0 :  -- standard link definitions
VAL link1out IS 1 :
VAL link2out IS 2 :
VAL link3out IS 3 :
VAL link0in  IS 4 :
VAL link1in  IS 5 :
VAL link2in  IS 6 :
VAL link3in  IS 7 :
VAL no.trams IS 4 :  -- number of TRAMS on B012

---- c4cmds - defines C004 crossbar commands ----

VAL c4.input.output IS 0 (BYTE):  
VAL c4.link IS 1 (BYTE):
VAL c4.enquire IS 2 (BYTE):
VAL c4.setup IS 3 (BYTE):
VAL c4.reset IS 4 (BYTE):
VAL c4.disconn.output IS 5 (BYTE):
VAL c4.disconn.link IS 6 (BYTE):

---- ctrlcmds - defines link control pipe commands ----

VAL link.req IS 40 (BYTE):  -- link actions
VAL link.ack IS 41 (BYTE):  -- for control pipe
VAL link.rel IS 42 (BYTE):
VAL link.brk IS 43 (BYTE):
VAL link.abt IS 44 (BYTE):
VAL link.rei IS 45 (BYTE):

---- blcksize - defines data block size for Agents and Users ----

VAL block.size IS INT16(16) :  -- bytes
-- intcmds - defines other internal commands used
-- and link translation table

VAL failed IS 60 (BYTE): -- predefined values
VAL reinit IS 61 (BYTE): -- signal to reset
VAL done.with.link IS 63 (BYTE): -- output is done
VAL all.done IS 65 (BYTE): -- tram done
VAL ready.to.rcv IS 67 (BYTE): -- Agent to User
VAL link.made IS 68 (BYTE): -- Mgr to Agent
VAL link.gone IS 69 (BYTE): -- Mgr to Agent
VAL byte.nil IS 99 (BYTE): -- nil defined
VAL BOOL otherwise IS TRUE: -- for IF stnmts
VAL INT user.failed IS 60 : -- for fault recovery
VAL INT user.reinit IS 61 : -- Mgr to User
VAL INT user.linkmade IS 68 : -- Mgr to User

-- array to convert C004 link number to link0 or 3 of Tram slot
-- index 0-31 for c4links 0-31
-- data INT is 0-15 for Trams 0-15 link0
-- 20-35 for Trams 0-15 link3
VAL [32] INT to.slot IS
[ 22, 25, 21, 5, 2, 1, 6, 26,
  15, 8, 35, 31, 12, 32, 28, 11,
  7, 23, 4, 3, 27, 24, 0, 26,
  34, 13, 14, 33, 9, 29, 30, 10] :

-- faulttime - used by Agents for watchdog timer

VAL fault.time IS INT(1024) : -- clock ticks

-- edgedata - defines B012 P1 edge connections wired
-- number of connections suitable for 0 to 0 links
VAL INT edge0.ct IS 8 :
-- number of connections suitable for 3 to 3 links
VAL INT edge3.ct IS 8 :

-- each edge.a[x] is manually patched to edge.b[x]
VAL edge.a IS [ 4(BYTE), 6(BYTE), 31(BYTE), 25(BYTE),
  19(BYTE), 16(BYTE), 8(BYTE), 12(BYTE),
  0(BYTE), 1(BYTE), 29(BYTE), 24(BYTE),
  17(BYTE), 21(BYTE), 10(BYTE), 14(BYTE)] :

VAL edge.b IS [ 5(BYTE), 3(BYTE), 28(BYTE), 26(BYTE),
  22(BYTE), 18(BYTE), 9(BYTE), 15(BYTE),
  2(BYTE), 7(BYTE), 30(BYTE), 27(BYTE),
  23(BYTE), 20(BYTE), 13(BYTE), 11(BYTE)] :
PROC controller (CHAN OF ANY from.pipe, to.pipe, to.cO, from.c0, to.cl, from.cl, VAL INT no.trams)

-- data library calls
#USE "c4cmds.tsr"
#USE "ctrlcmds.tsr"
#USE "intcmds.tsr"
#USE "edgedata.tsr"

VAL delay IS 16000 : -- pause in status reset
BYTE token, source, dest :
BYTE src.conn, dst.conn : -- holds current hookups
BOOL continue, accomplished :

[16]BOOL edge.ok : -- edge status array
INT i, now, edge.index, req.cnt :
TIMER clock : -- timer for edge reset

-- Subroutine to set a link to nil if inactive,
-- or reset high bit to low if active

PROC reset.or.nil (BYTE c4link)

IF
   -- if high bit is set (link is active)
   ((INT(c4link))BITAND(INT(BYTE #80)))=(INT(BYTE #80))
   -- then reset high bit to low for valid link number
   c4link := BYTE((INT(c4link)) >> (INT(BYTE(#80)))))

otherwise
   -- high bit is low therefore link is NOT active
   c4link := byte.nil

: -- end of PROC reset.or.nil
-- Subroutine to handle C004 commands and response

PROC c4.cmd (CHAN OF ANY to.x, fm.x,
            VAL BYTE cmd, b1, b2, BYTE b3)

SEQ
    IF
        cmd = byte.nil
            SKIP
        b1 = byte.nil
            to.x ! cmd; b2
        otherwise
            to.x ! cmd; b1; b2
        -- determine command to use
        -- just do enquire
        -- single parameter command
        -- two parameter command
        to.x ! c4.enquire; b2
        fm.x ? b3
        reset.or.nil(b3)
        -- verify action by enquire
    : -- end of c4.cmd

-- Subroutine to get current connection to
-- requested hookup. All i/o in c4 link designs

PROC get.current.tie (CHAN OF ANY to.c0, from.c0,
                        to.c1, from.c1,
                        VAL BYTE in.link, BYTE link.tied.to)

    -- Note: all Bytes in terms of C004 Link designations
    SEQ
        IF
            in.link = byte.nil
                SKIP
            ((to.slot[INT(in.link)]) < (16)) -- it is a link0
                -- get input link (if any) connected to in.link
                c4.cmd(to.c1, from.c1, byte.nil, byte.nil,
                       in.link, link.tied.to)
            ((to.slot[INT(in.link)]) > (19)) -- it is a link3
                -- get input link (if any) connected to in.link
                c4.cmd(to.c0, from.c0, byte.nil, byte.nil,
                       in.link, link.tied.to)
            otherwise
                SKIP
            : -- end of get.current.tie

104
PROC make.conn (CHAN OF ANY to.c0, from.c0, to.c1, from.c1, VAL BYTE source, dest, BOOL conn.ok)

PROC make.0033 (CHAN OF ANY to.a, fm.a, to.b, fm.b, VAL BYTE src, dst, VAL BOOL link.0toO, BOOL conn03.ok, BYTE in.a, in.b)

BOOL edge.used:
INT index, max.index:
BYTE inputaa, inputbb:

SEQ -- select appropriate index ranges for link
IF
  link.0toO
  SEQ
  index := 0
  max.index := edge0.ct
  otherwise
  SEQ
  index := edge0.ct
  max.index := edge0.ct + edge3.ct

edge.used := TRUE
WHILE (edge.used AND (index < max.index))
  IF
    edge.ok[index] -- edge is still "good"
    SEQ
      c4.cmd(to.a, fm.a, byte.nil, byte.nil, edge.a[index], inputaa)
      c4.cmd(to.a, fm.a, byte.nil, byte.nil, edge.b[index], inputbb)

  IF
    ((inputaa = byte.nil) AND
     (inputbb = byte.nil))
    SEQ -- if edge not currently connected
      edge.used := FALSE
      c4.cmd(to.a, fm.a, c4.input.output, src, edge.a[index], in.a)
      c4.cmd(to.b, fm.b, c4.input.output, edge.b[index], dst, inputbb)
      c4.cmd(to.a, fm.a, c4.output.output,
dst, edge.b[index], in.b)
c4.cmd(to.b, fm.b, c4.input.output, 
    edge.a[index], src, inputaa)

IF
((inputaa = edge.a[index]) AND
(inputbb = edge.b[index])
    conn03.ok := TRUE
otherwise
    conn03.ok := FALSE

otherwise
    index := index + 1 -- busy, try next
otherwise
    index := index + 1 -- get next edge, "bad"
: -- end of make.0033

-- Main Section for PROC make.conn
BYTE linka, linkb, inputa, inputb :
BOOL crosslink.ok :

SEQ
    crosslink.ok := TRUE
    -- make linka link0 of two to be connected if other
    -- is link3. If both link0's 3's, no matter
    IF
        ((to.slot[INT(source)]) < (16))
        SEQ
            linka := source
            linkb := dest
    otherwise
        SEQ
            linkb := source
            linka := dest
    IF
        ((to.slot[INT(linka)]) < (16)) AND
        ((to.slot[INT(linkb)]) > (19))
        SEQ
            c4.cmd(to.c0, from.c0, c4.input.output, 
                linka, linkb, inputa)
            c4.cmd(to.c1, from.c1, c4.input.output, 
                linkb, linka, inputb)
    make.0033(to.c0, from.c0, to.c1, from.c1, linka, 
        linkb, TRUE, crosslink.ok, inputa, inputb)
(((to.slot[INT(linka)]) > (19)) AND
((to.slot[INT(linkb)]) > (19)))
make.0033(to.c1, from.c1, to.c0, from.c0, linka,
    linkb, FALSE, crosslink.ok, inputa, inputb)

otherwise
SKIP

-- verify return input enq bytes match (high bit set)
IF -- links match, connected ok
  (((linka = inputa) AND (linkb = inputb))
    AND crosslink.ok)
  conn.ok := TRUE
otherwise -- links are not made
  conn.ok := FALSE
: -- end of make.conn

-----------------------------------------------
-- Subroutine to brk a connection between two Tram Links
-- (may involve as many as 4 C004 links)
-- verify connection is broken, and return status flag
-----------------------------------------------
PROC break.conn (CHAN OF ANY to.c0, from.c0,
    to.c1, from.c1,
    VAL BYTE source, dest, BOOL broken.ck)

-----------------------------------------------
-- SubSubroutine to handle a 0-0 or 3-3 link
-----------------------------------------------
PROC break.0033 (CHAN OF ANY to.a, fm.a, to.b, fm.b,
    VAL BYTE src, dst,
    BYTE in.a, in.b, BOOL break03.ok)

BYTE edgea, edgeb :
SEQ
-- get edges used
to.a ! c4.enquire; src
fm.a ? edgea
to.a ! c4.enquire; dst
fm.a ? edgeb
-- disconnect the edges
c4.cmd(to.b, fm.b, c4.disconn.output, byte.nil,
    edgea, in.a)
c4.cmd(to.b, fm.b, c4.disconn.output, byte.nil,
    edgeb, in.b)

IF
    ((in.a = byte.nil) AND (in.b = byte.nil))

107
break03.ok := TRUE
otherwise
   break03.ok := FALSE

   c4.cmd(to.a, fm.a, c4.disconn.output, byte.nil, src, in.a)
   c4.cmd(to.a, fm.a, c4.disconn.output, byte.nil, dst, in.b)
: -- end of break.0033

-- Main Section of PROC break.conn
BYTE linka, linkb, inputa, inputb :
BOOL edge.free :

SEQ
   edge.free := TRUE
   IF
      ((to.slot[INT(source)]) < (16))
      SEQ
         linka := source
         linkb := dest
   otherwise
      SEQ
         linkb := source
         linka := dest
   ENDIF

   -- break a tram link 0 to tram link 3
      (((to.slot[INT(linka)]) < (16)) AND
         ((to.slot[INT(linkb)]) > (19)))
      SEQ
         c4.cmd(to.c1, from.c1, c4.disconn.output,
               byte.nil, linka, inputa)
         c4.cmd(to.c0, from.c0, c4.disconn.output,
               byte.nil, linkb, inputb)
      (((to.slot[INT(linka)]) < (16)) AND
         ((to.slot[INT(linkb)]) < (16)))
      break.0033 (to.c1, from.c1, to.c0, from.c0,
               linka, linkb, inputa, inputb, edge.free)
   (((to.slot[INT(linka)]) > (19)) AND
      ((to.slot[INT(linkb)]) > (19)))
      break.0033 (to.c0, from.c0, to.c1, from.c1,
               linka, linkb, inputa, inputb, edge.free)
   otherwise
      SKIP

108
-- verify broken links inactive (high bit off)
IF
-- links verified broken
(((inputa = byte.nil) AND (inputb = byte.nil))
AND edge.free)
broken.ok := TRUE
otherwise
-- links not broken
broken.ok := FALSE
: -- end of break.conn

-- main body of Controller Procedure

SEQ

-- initialization
to.c0 ! c4.reset
to.c1 ! c4.reset
SEQ i = 0 FOR 16
edge.ok[i] := TRUE
edge.index := 0
clock ? now
req.cnt := 0

WHILE TRUE
ALT
from.pipe ? token; source; dest
SEQ
IF token = link.req
SEQ
get.current.tie (to.c0, from.c0, to.c1,
from.c1, source, src.conn)
get.current.tie (to.c0, from.c0, to.c1,
from.c1, dest, dst.conn)

IF
((src.conn = byte.nil) AND
(dst.conn = byte.nil))
-- connections free, make the request
SEQ
make.conn(to.c0, from.c0, to.c1,
from.c1, source, dest, accomplished)
IF
accomplished
SEQ
-- conn made OK, send ACK
root.pipe ! link.ack; source;
dest
otherwise
SEQ
-- for whatever reason, no go
to.pipe ! link.req; source;
dest
req.cnt := req.cnt + 1

otherwise -- repeat request later
to.pipe ! link.req; source; dest

token = link.rel
-- break an existing connection btw 2 Trams
SEQ
break.conn (to.c0, from.c0, to.c1, from.c1,
source, dest, accomplished)
IF
accomplished -- links broken
SEQ
to.pipe ! link.brk; source; dest
otherwise -- links not broken,
-- let link.rel loop again
to.pipe ! link.rel; source; dest

token = link.abt
SEQ
to.pipe ! link.abt, source; dest
-- pass to originator
get.current.tie (to.c0, from.c0, to.c1,
from.c1, source, src.conn)
get.current.tie (to.c0, from.c0, to.c1,
from.c1, dest, dst.conn)
continue := TRUE
i := 0
WHILE ((i < 16) AND continue)
SEQ
IF
(((edge.a[i])=src.conn)
OR ((edge.b[i])=src.conn))
SEQ
edge.ok[i] := FALSE
continue := FALSE
to.pipe ! BYTE(i+220); byte.nil;
byte.nil
otherwise
i := i + 1

to.pipe ! link.rei; source; dest
break.conn (to.c0, from.c0, to.c1, from.c1,
source, dest, accomplished)
make.conn (to.c0, from.c0, to.c1, from.c1,
source, dest, accomplished)
IF
  accomplished
  to.pipe ! link.ack; source; dest
otherwise
  SEQ
    to.pipe ! link.req; source; dest
    req.cnt := req.cnt + 1
  token = link.ack
  -- consume rest of acknowledge packet
  SKIP
  token = link.brk
  -- msg cycled, consume
  SKIP
  token = link.rei
  -- consume rest of acknowledge packet
  SKIP
  otherwise
    -- pass it on for testing and monitoring
    to.pipe ! token; source; dest

(req.cnt>no.trams) & clock ? AFTER now PLUS delay
  SEQ
    clock ? now
    req.cnt := 0
    edge.ok[edge.index] := TRUE
    to.pipe ! BYTE(edge.index+200); byte.nil;
      byte.nil
    edge.index := edge.index + 1
    IF
      edge.index >= (edge0.ct + edge3.ct)
      edge.index := 0
    otherwise
      SKIP
    : -- end of controller code
--- TRAM.CODE - code for each TRAM, performing work & requesting links
---
PROC tram.code (CHAN OF ANY to.pipe, from.pipe,
   link0.in, link0.out, link3.in, link3.out,
   VAL INT slot.desig, no.trams)

--- LINK.MANAGER - maintains pipe local link control
---
PROC link.manager (CHAN OF ANY to.pipe, from.pipe,
   mgr.to.agent0, mgr.from.agent0,
   mgr.to.agent3, mgr.from.agent3,
   mgr.from.user0, mgr.from.user3,
   CHAN OF INT mgr.to.user0, mgr.to.user3,
   VAL INT slot.desig, no.trams)

#USE "c4cmds.tsr"
#USE "ctrlcmds.tsr"
#USE "intcmds.tsr"
BYTE link0, link3 :
BYTE link0.tied.to, link3.tied.to :
BYTE token, byte1, byte2, req.source, req.dest :

-- index corresponds to slot number w/ 0-15 for link0
-- and w/ 20-35 for link3
-- stored value is hardwired C004 link to link0 or 3
VAL link03.desig IS [22, 5, 4, 19, 18, 3,
   6, 16, 9, 28, 31, 15, 12, 25, 26, 8, 99, 99,
   99, 99, 23, 2, 0, 17, 21, 1, 7, 20, 14, 29,
   30, 11, 13, 27, 24, 10]:

VAL INT buffer.size IS 3 :
[buffer.size-1]CHAN OF ANY b:
[buffer.size-2]BYTE t, s, d :
CHAN OF ANY to.buffer :
BYTE t.in, s.in, d.in, t.out, s.out, d.out :

PAR
SEQ
   link0 := BYTE(link03.desig[slot.desig])
   link3 := BYTE(link03.desig[(slot.desig+20)])
   link0.tied.to := byte.nil
   link3.tied.to := byte.nil
WHILE TRUE
  ALT
  mgr.from.agent0 ? token
    IF
      (token = done.with.link)
        to.buffer ! link.rel; link0;
        link0.tied.to
      (token = failed)
        to.buffer ! link.abt; link0;
        link0.tied.to
      (token = all.done)
        to.buffer ! BYTE(slot.desig+100);
        byte.nil; byte.nil
      ((INT(token)) < 36)
        SEQ
          -- token is a link designation
          link0.tied.to :=
          BYTE(link0.desig[INT(token)])
          to.buffer ! link.req; link0;
          link0.tied.to
    otherwise
      SKIP
  mgr.from.agent3 ? token
    IF
      (token = done.with.link)
        to.buffer ! link.rel; link3;
        link3.tied.to
      (token = failed)
        to.buffer ! link.abt; link3;
        link3.tied.to
      (token = all.done)
        to.buffer ! BYTE(slot.desig+120);
        byte.nil; byte.nil
      ((INT(token)) < 36)
        SEQ
          -- token is a link designation
          link3.tied.to :=
          BYTE(link3.desig[INT(token)])
          to.buffer ! link.req; link3;
          link3.tied.to
    otherwise
      SKIP
  from.pipe ? token; byte1; byte2
    IF
      token = link.req
        to.buffer ! token; byte1; byte2
      token = link.rel
        to.buffer ! token; byte1; byte2
token = link.ack
SEQ
IF
-- if local link involved,
-- inform user to xmit
byte2 = link0
SEQ
mgr.to.user0 ! user.linkmade
byte2 := byte.nil
byte2 = link3
SEQ
mgr.to.user3 ! user.linkmade
byte2 := byte.nil
otherwise
SKIP
ENDIF

IF
-- if local link involved,
-- inform user to xmit
bytel = link0
SEQ
mgr.to.agent0 ! link.made
bytel := byte.nil
bytel = link3
SEQ
mgr.to.agent3 ! link.made
bytel := byte.nil
otherwise
SKIP
ENDIF

IF
((bytel=byte.nil)AND(byte2=byte.nil))
SKIP -- consume
otherwise
to.buffer ! token; bytel; byte2
ENDIF

token = link.brk
IF
-- if local links involved,
-- clear user status, consume
bytel = link0
SEQ
link0.tied.to := byte.nil
mgr.to.agent0 ! link.gone
bytel = link3
SEQ
link3.tied.to := byte.nil
mgr.to.agent3 ! link.gone
otherwise
-- pass on if not of local interest
to.buffer ! token; bytel; byte2
ENDIF
token = link.abt
   -- since originated by source,
   -- consume if this is source
SEQ
   IF
      ((byte1 = link0) OR (byte1 = link3))
      SKIP --consume
   otherwise
      to.buffer ! token; byte1; byte2
   IF
      byte2 = link0
      mgr.to.user0 ! user.failed
      byte2 = link3
      mgr.to.user3 ! user.failed
   otherwise
      SKIP

.token = link.rei
SEQ
   IF
      -- if local link involved,
      -- inform user to xmit
      byte1 = link0
      SEQ
         mgr.to.agent0 ! reinit
         byte1 := byte.nil
      byte1 = link3
      SEQ
         mgr.to.agent3 ! reinit
         byte1 := byte.nil
      otherwise
         SKIP
   IF
      -- if local link involved,
      -- inform user to xmit
      byte2 = link0
      SEQ
         mgr.to.user0 ! user.reinit
         byte2 := byte.nil
      byte2 = link3
      SEQ
         mgr.to.user3 ! user.reinit
         byte2 := byte.nil
      otherwise
         SKIP

115
IF 
((bytel=byte.nil)AND(byte2=byte.nil))
  SKIP
otherwise:
  to.buffer ! token; bytel; byte2
otherwise
  -- non-cmd byte, pass it on
  to.buffer ! token; bytel; byte2

PAR
  WHILE TRUE
    SEQ
      to.buffer ? t.in; s.in; d.in
      b[0] ! t.in; s.in; d.in
    PAR
    PAR p = 0 FOR (buffer.size-2)
      WHILE TRUE
        SEQ
          b[p] ? t[p]; s[p]; d[p]
          b[p+1] ! t[p]; s[p]; d[p]
      PAR
      WHILE TRUE
        SEQ
          b[buffer.size-2] ? t.out; s.out; d.out
          to.pipe ! t.out; s.out; d.out
: -- end of manager

-- AGENT0 - handle data msg output for user0
PROC agent0 (CHAN OF ANY mgr.to.agent0, mgr.from.agent0,
  agent0.to.user0,
  agent0.from.user0, link0.out,
  VAL INT link.desig)

#USE reinit
#USE "intcmds.tsr"
#USE "blcksize.tsr"
#USE "faultime.tsr"
VAL block.len IS INT(block.size) :
BYTE byte.in :
  [block.len]BYTE msg :
BOOL continue, aborted :
TIMER time.c :
INT now, check.time :
SEQ
agent0.to.user0 ! ready.to.rcv
continue := TRUE
WHILE continue
  PRI ALT
    mgr.to.agent0 ? byte.in
    IF
      byte.in = link.made
      SEQ
        msg[0] := BYTE(link.desig)
        -- repl dest w/ source
        time.c ? now
        check.time := now PLUS fault.time
        OutputOrFail.t(link0.out,msg,time.c,
                      check.time,aborted)
      IF
        aborted
      SEQ
        mgr.from.agent0 ! failed
        mgr.to.agent0 ? byte.in
        Reinitialise(link0.out)
      otherwise
        mgr.from.agent0 ! done.with.link
        byte.in = link.gone
        agent0.to.user0 ! ready.to.rcv
    otherwise
      SKIP
  otherwise
    SKIP
agent0.from.user0 ? msg
SEQ
  mgr.from.agent0 ! msg[0]
  IF
    msg[0] = all.done
    continue := FALSE
  otherwise
    SKIP
: -- end of agent0

117
-- USER0 - production process using link0

PROC user0 (CHAN OF ANY agent0.to.user0,
            agent0.from.user0, mgr.from.user0, link0.in,
            CHAN OF INT mgr.to.user0,
            VAL INT link.desig, no.trams)

#USE reinit
#USE snglmath
#USE "intcmds.tsr"
#USE "blcksize.tsr"

-- user supplied values
VAL block.len IS INT(block.size) :
[block.len]BYTE data.out, data.in :
BYTE in.byte :
INT req.dest, counter, check.time, now, any.int, seed16:
BOOL continue, aborted, active :
INT32 seed :
REAL32 result :
TIMER time.c, clock :

PRI PAR
  WHILE TRUE
    SEQ
      mgr.to.user0 ? any.int -- trigger communications
      IF
        any.int = user.linkmade
        SEQ
          InputOrFail.c(link0.in, data.in,
                        mgr.to.user0, aborted)
        IF
          aborted
          SEQ
            mgr.to.user0 ? any.int
            IF
              any.int = user.reinit
              Reinitialise(link0.in)
            otherwise
              SKIP
          otherwise
          SKIP -- data rcvd OK, use as desired
      IF
        any.int = user.reinit
        Reinitialise(link0.in)
      otherwise
      SKIP

118
SEQ
IF
    link.desig < no.trams
    SEQ
        continue := TRUE
        counter := 0
    otherwise
    SEQ
        continue := FALSE
        counter := 10000

clock ? seed16
seed := INT32(seed16)
--seed := INT32(link.desig*100)

WHILE counter < 1000
SEQ
    counter := counter + 1
    -- generate data
    SEQ i = 1 FOR (block.len-1)
        data.out[i] := 100(BYTE)
    -- send to any link 0 or 3
    req.dest := link.desig
    WHILE ((req.dest = link.desig) OR
            ((req.dest >= no.trams) AND
             (req.dest < 20)) OR
            (req.dest >= (no.trams+20)))
        -- for link 0-0 ONLY! (commented out otherwise)
        --WHILE ((req.dest=link.desig) OR
        --(req.dest>=no.trams))
        -- put destination address in array at addr 0
    data.out[0] := BYTE(req.dest)
    agent0.to.user0 ? in.byte
    agent0.from.user0 ! data.out

agent0.to.user0 ? in.byte
agent0.from.user0 ! all.done
: -- end of user0
-- AGENT3 - handle data msg output for user3

PROC agent3 (CHAN OF ANY mgr.to.agent3, mgr.from.agent3, agent3.to.user3, agent3.from.user3, link3.out, VAL INT link.desig)

#USE reinit
#USE "intcmdtsr"
#USE "blcksize.tsr"
#USE "faulttime.tsr"
VAL block.len IS INT(block.size) :
BYTE byte.in :
[block.len]BYTE msg :
BOOL continue, aborted :
TIMER time.c :
INT now, check.time :

SEQ
  agent3.to.user3 ! ready.to.rcv
  continue := TRUE
  WHILE continue
    PRI ALT
      mgr.to.agent3 ? byte.in
      IF
        byte.in = link.made
        SEQ
          msg[0] := BYTE(link.desig)
          -- repl dest w/ source
          time.c ? now
          check.time := now PLUS fault.time
          OutputOrFail.t(link3.out,msg,time.c,check.time,aborted)
      IF
        aborted
        SEQ
          mgr.from.agent3 ! failed
          mgr.to.agent3 ? byte.in
          Reinitialise(link3.out)
        otherwise
          mgr.from.agent3 ! done.with.link
      otherwise
        byte.in = link.gone
        agent3.to.user3 ! ready.to.rcv
    otherwise
      SKIP
  120
agent3.from.user3 ? msg
SEQ
mgr.from.agent3 ! msg[0]
IF
msg[0] = all.done
continue := FALSE
otherwise
SKIP

: -- end of agent3

-- USER3 - production process using link3
PROC user3 (CHAN OF ANY agent3.to.user3,
    agent3.from.user3, mgr.from.user3, link3.in,
    CHAN OF INT mgr.to.user3,
    VAL INT link.desig, no.trams)

#USE reinit
#USE snglmath
#USE "intcmds.tsr"
#USE "blksize.tsr"

-- user supplied values
VAL block.len IS INT(block.size) :
[block.len]BYTE data.out, data.in :
BYTE in.byte :
INT req.dest, counter, check.time, now, any.int, seed16:
BOOL continue, aborted, active :
INT32 seed :
REAL32 result :
TIMER time.c, clock :

PRI PAR
WHILE TRUE
SEQ
mgr.to.user3 ? any.int -- trigger communications
IF
any.int = user.linkmade
SEQ
InputOrFail.c(link3.in, data.in,
    mgr.to.user3, aborted)
IF
aborted
SEQ
mgr.to.user3 ? any.int
IF
any.int = user.reinit
Reinitialise(link3.in)
otherwise
SKIP

otherwise
SKIP -- data rcvd OK, use as desired

any.int = user.reinit
Reinitialise(link3.in)

otherwise
SKIP

SEQ
IF
link.desig < (no.trams + 20)
SEQ
continue := TRUE
counter := 0
otherwise
SEQ
continue := FALSE
counter := 10000

clock ? seed16
seed := INT32(seed16)
--seed := INT32(link.desig*100)

WHILE counter < 1000
SEQ
counter := counter + 1
-- generate data
SEQ i = 1 FOR (block.len-1)
data.out[i] := 100(BYTE)

-- send to any link 0 or 3
req.dest := link.desig
WHILE ((req.dest = link.desig) OR ((req.dest >= no.trams) AND (req.dest < 20)) OR (req.dest >= (no.trams+20)))

-- for link 3-3 ONLY! (comment out)
--WHILE ((req.dest = link.desig) OR
--(req.dest < 20) OR
--(req.dest >= (no.trams+20)))

SEQ
result,seed := RAN(seed)
req.dest := INT ROUND (result*31.0(REAL32))

122
-- put destination address in array at addr C
data.out[0] := BYTE(req.dest)

agent3.to.user3 ? in.byte
agent3.from.user3 ! data.out

agent3.to.user3 ? in.byte
agent3.from.user3 ! all.done
: -- end of user3

-- TRAM.CODE - master program for each Tram
-- internal channels between users and manager
CHAN OF ANY
mgr.to.agent0, mgr.from.agent0,
mgr.to.agent3, mgr.from.agent3,
mgr.from.user0, mgr.from.user3,
agent0.to.user0, agent3.to.user3,
agent0.from.user0, agent3.from.user3 :

CHAN OF INT
mgr.to.user0, mgr.to.user3 :

PRI PAR
PAR

link.manager (to.pipe, from.pipe,
mgr.to.agent0, mgr.from.agent0,
mgr.to.agent3, mgr.from.agent3,
mgr.from.user0, mgr.from.user3,
mgr.to.user0, mgr.to.user3,
slot.desig, no.trams)

agent0 (mgr.to.agent0, mgr.from.agent0,
agent0.to.user0, agent0.from.user0,
link0.out, slot.desig)

agent3 (mgr.to.agent3, mgr.from.agent3,
agent3.to.user3, agent3.from.user3,
link3.out, slot.desig+20)

PAR

user0 (agent0.to.user0, agent0.from.user0,
mgr.from.user0, link0.in,
mgr.to.user0, slot.desig, no.trams)

user3 (agent3.to.user3, agent3.from.user3,
mgr.from.user3, link3.in,
mgr.to.user3, slot.desig+20, no.trams)

: -- end of TRAM.CODE
-- Placed Pars (Controller -> T212, TramCode -> T800's)
-- Global Declarations, Constants and Placed Pars

#USE "links.tsr"

-- control info passing pipe btw T212 and Trams
[no.trams+2]CHAN OF ANY pipe :

-- links btw T212 and C004s
CHAN OF ANY t2.to.c0, t2.from.c0,
  t2.to.cl, t2.from.cl :

-- !!!!! make protocol for link0’s & 3’s !!!!!
-- data xfer channels between all Trams (via C004’s)
[no.trams]CHAN OF ANY link0.to.c0, link0.from.cl,
  link3.to.cl, link3.from.c0 :

PLACED PAR
PROCESSOR no.trams T2
  PLACE pipe[0] AT link2in :
  PLACE pipe[no.trams+1] AT link1out :
  PLACE t2.to.c0 AT link0out :
  PLACE t2.from.c0 AT link0in :
  PLACE t2.to.cl AT link3out :
  PLACE t2.from.cl AT link3in :

controller (pipe[0], pipe[no.trams+1],
  t2.to.c0, t2.from.c0,
  t2.to.cl, t2.from.cl, no.trams)

PLACED PAR slot = 0 FOR (no.trams)

PROCESSOR ((slot+1)*100) T8
  PLACE pipe[slot] AT link1out :
  PLACE pipe[slot+1] AT link2in :
  PLACE link0.from.cl[slot] AT link0in :
  PLACE link0.to.c0[slot] AT link0out :
  PLACE link3.from.c0[slot] AT link3in :
  PLACE link3.to.cl[slot] AT link3out :

tram.code (pipe[slot], pipe[slot+1],
  link0.from.cl[slot], link0.to.c0[slot],
  link3.from.c0[slot], link3.to.cl[slot],
  slot, no.trams)
APPENDIX B

MESSAGE EXCHANGE TEST CODE FOR B004

-- TIMER - code for B004 Timer of B012 activity in MSGX
-- Operating on B004 Evaluation Board

#USE userio
#USE "links.tsr"            -- transputer link defns
#USE "blcksize.tsr"        -- data block size for ref

CHAN OF ANY up.in, up.out, echo.to.buffer:

-- place links
PLACE up.in    AT link2in :
PLACE up.out  AT link3out :

PROC timer (CHAN OF INT keyboard,
                CHAN OF ANY screen, up.in, up.out,
                VAL INT no.trams)

    #USE "intcmds.tsr"
    VAL terminate IS 199 (BYTE): -- shuts down buffer procs
    BYTE command.byte, byte1, byte2 :
    BOOL more :
    TIMER clock :
        [37][7]INT counter :  -- various command counters
        [6]INT total, grand :
        [36]INT user.done :
        INT key.in, done.count, sum, sumtot, grandtot :
        INT start, stop, elapsed, slot.temp :

-- main code for B004 monitor - TIMER

SEQ
    -- initialize loop
    more := TRUE
    done.count := 0
    sumtot := 0
    grandtot := 0
    SEQ i = 0 FOR 36
        user.done[i] := 0
    SEQ i = 0 FOR 37
SEQ j = 0 FOR 7
    counter[i][j] := 0
SEQ i = 0 FOR 6
SEQ
    total[i] := 0
    grand[i] := 0
    clock := start
write.int(screen, start, 12)
newline(screen)

WHILE more
PRI ALT
    up.in := command.byte; bytel; byte2
    IF
        (((INT(command.byte)) > (39)) AND
         ((INT(command.byte)) < (46)))
        SEQ
        -- command byte, get next two parameters
        up.out := command.byte; bytel; byte2
        -- update appropriate command counter
        counter[to.slot[INT(bytel)]]
        [(INT(command.byte))-40] :=
        counter[to.slot[INT(bytel)]]
        [(INT(command.byte))-40] + 1
    elseif
        (((INT(command.byte)) > (99)) AND
         ((INT(command.byte)) < (132)))
        -- a tram has sent a notice it is done xmit
        SEQ
        clock := slot.temp
        user.done[(INT(command.byte))-100)] :=
        ((slot.temp-start)*64)/1000
        done.count := done.count + 1
        IF
        done.count = (no.trams * 2)
        SEQ
        write.full.string(screen, "Test Complete!")
        newline(screen)
        clock := stop
        elapsed := ((stop-start)*64)/1000
        write.full.string(screen, "Elapsed Time: ")
        write.int(screen, elapsed, 12)
        write.full.string(screen, " Block Size: ")
        write.int(screen, INT(block.size),12)
    else
        SKIP
    endif
126
otherwise
SEQ SKIP

keyboard ? key.in
SEQ
-- handle keyboard inputs
IF (key.in = 81) OR (key.in = 113) -- "Q": quit
SEQ
more := FALSE
up.out ! terminate; terminate; terminate

(key.in = 80) OR (key.in = 112) -- "P": pause
SEQ
newline(screen)
write.full.string(screen,"press any key.")
keyboard ? key.in

(key.in = 82) OR (key.in = 114)
-- "R": resend a byte
up.out ! BYTE(60)

(key.in = 83) OR (key.in = 115)
-- "S": show summary
SEQ
newline(screen)
newline(screen)
sumtot := 0
grandtot := 0
write.full.string(screen,""
write.full.string(screen, "link.req link.ack link.brk")
write.full.string(screen, "link.abt link.rei ")
newline(screen)
SEQ i = 0 FOR 6
SEQ
grand[i] := 0
total[i] := 0
SEQ i = 0 FOR no.trams
SEQ
write.int(screen,i,6)
write.int(screen,counter[i][0],12)
write.int(screen,counter[i][1],12)
write.int(screen,counter[i][3],12)
write.int(screen,counter[i][4],12)
write.int(screen,counter[i][5],12)
write.int(screen,user.done[i],12)
newline(screen)
SEQ \( j = 0 \) FOR 6

SEQ
\[
\begin{align*}
\text{total}[j] & := \text{total}[j] + \text{counter}[i][j] \\
\text{grand}[j] & := \text{grand}[j] + \text{counter}[i][j] \\
\text{sumtot} & := \text{sumtot} + \text{counter}[i][j] \\
\text{grandtot} & := \text{grandtot} + \text{counter}[i][j]
\end{align*}
\]

write.full.string(screen, "Totals")
write.int(screen, total[0], 12)
write.int(screen, total[1], 12)
write.int(screen, total[2], 12)
write.int(screen, total[3], 12)
write.int(screen, total[4], 12)
write.int(screen, total[5], 12)
write.int(screen, sumtot, 12)
newline(screen)

sumtot := 0
SEQ \( i = 0 \) FOR 6
  total[i] := 0
SEQ \( i = 20 \) FOR no.trams
  SEQ
  \begin{align*}
  \text{write.int}(screen, i, 6) \\
  \text{write.int}(screen, \text{counter}[i][0], 12) \\
  \text{write.int}(screen, \text{counter}[i][1], 12) \\
  \text{write.int}(screen, \text{counter}[i][2], 12) \\
  \text{write.int}(screen, \text{counter}[i][3], 12) \\
  \text{write.int}(screen, \text{counter}[i][4], 12) \\
  \text{write.int}(screen, \text{counter}[i][5], 12) \\
  \text{write.int}(screen, \text{user.done}[i], 12)
  \end{align*}
  newline(screen)
SEQ \( j = 0 \) FOR 6
  SEQ
  \begin{align*}
  \text{total}[j] & := \text{total}[j] + \text{counter}[i][j] \\
  \text{grand}[j] & := \text{grand}[j] + \text{counter}[i][j] \\
  \text{sumtot} & := \text{sumtot} + \text{counter}[i][j] \\
  \text{grandtot} & := \text{grandtot} + \text{counter}[i][j]
  \end{align*}
  write.full.string(screen, "Totals")
write.int(screen, total[0], 12)
write.int(screen, total[1], 12)
write.int(screen, total[2], 12)
write.int(screen, total[3], 12)
write.int(screen, total[4], 12)
write.int(screen, total[5], 12)
write.int(screen, sumtot, 12)
sumtot := 0
newline(screen)
newline(screen)
write.full.string(screen, "Grands")
write.int(screen, grand[0], 12)
write.int(screen, grand[1], 12)
write.int(screen, grand[2], 12)
write.int(screen,grand[4],12)
write.int(screen,grand[5],12)
write.int(screen,grandtot,12)

otherwise -- any key, ignored
  SKIP

: -- end of TIMER

---------------------------------------------------------------------
-- BUFFER -- variable buffer on pipe
---------------------------------------------------------------------
PROC buffer (CHAN OF ANY into.buffer, outof.buffer)

VAL INT buffer.size IS 3:
BYTE t.in, s.in, d.in, t.out, s.out, d.out :
[buffer.size-1]CHAN OF ANY b:
[buffer.size-2]BYTE t, s, d :
[buffer.size-2]BOOL buffer.on :
BOOL first.buffer.on, last.buffer.on :
VAL terminate IS 199 (BYTE):
VAL BOOL otherwise IS TRUE:

PAR
SEQ
  first.buffer.on := TRUE
  WHILE first.buffer.on
  SEQ
    into.buffer ? t.in; s.in; d.in
    b[0] ! t.in; s.in; d.in
    IF
      t.in = terminate
      first.buffer.on := FALSE
    otherwise
      SKIP
  PAR  -- replicated PAR to desired buffer size
    PAR p = 0 FOR (buffer.size-2)
    SEQ
      buffer.on[p] := TRUE
      WHILE buffer.on[p]
      SEQ
        b[p] ? t[p]; s[p]; d[p]
        b[p+1] ! t[p]; s[p]; d[p]
        IF
          t[p] = terminate
          buffer.on[p] := FALSE
        otherwise
          SKIP
SEQ
  last.buffer.on := TRUE
  WHILE last.buffer.on
    SEQ
      b[buffer.size-2] ? t.out; s.out; d.out
      outof.buffer ! t.out; s.out; d.out
      IF
        t.out = terminate
          last.buffer.on := FALSE
        otherwise
          SKIP
    : -- end of buffer

-- BUFFER -- variable buffer on pipe

PROC buffer (CHAN OF ANY into.buffer, outof.buffer)

  VAL INT buffer.size IS 3 :
  BYTE byte.in, byte.out :
    [buffer.size-1]CHAN OF ANY b:
    [buffer.size-2]BYTE byte.hold :
    [buffer.size-2]BOOL buffer.on :
  BOOL first.buffer.on, last.buffer.on :
  VAL terminate IS 199 (BYTE):
  VAL BOOL otherwise IS TRUE:

  PAR
    SEQ
      first.buffer.on := TRUE
      WHILE first.buffer.on
        SEQ
          into.buffer ? byte.in
          b[0] ! byte.in
          IF
            byte.in = terminate
              first.buffer.on := FALSE
            otherwise
              SKIP
        PAR
          p = 0 FOR (buffer.size-2)
          SEQ
            buffer.on[p] := TRUE
            WHILE buffer.on[p]
              SEQ
                b[p] ? byte.hold[p]
                b[p+1] ! byte.hold[p]
                IF
                  byte.hold[p] = terminate
            PAR
              ...
buffer.on[p] := FALSE
otherwise
    SKIP

SEQ
    last.buffer.on := TRUE
    WHILE last.buffer.on
        SEQ
            b[buffer.size-2] ? byte.out
            outof.buffer ! byte.out
            IF
                byte.out = terminate
                last.buffer.on := FALSE
            otherwise
                SKIP
        :-- end of buffer

PAR
    timer(keyboard, screen, up.in, echo.to.buffer, no.trams)
    buffer(echo.to.buffer, up.out)
APPENDIX C
MESSAGE EXCHANGE ACTIVITY DISPLAY

--- STATUSdb - B004 Status of B012 activity in MSGX
--- To display activity of message exchange
---
#USE userio
#USE "links.tsr"
---
CHAN OF ANY up.in, up.out, echo.to.buffer,
    buffer.to.echo, shut.down :
--- place links to intercept link control pipe commands
PLACE up.in AT link2in :
PLACE up.out AT link3out :
---
CHAN OF ANY to.buffer :
VAL INT buffer.size IS 25 : -- number of buffer stages
BYTE byte.in, byte.out, byte.sd :
    [buffer.size-1]CHAN OF ANY b: -- array of channels
    [buffer.size-2]BYTE byte.hold :-- array of command bytes
    [buffer.size-2]BOOL buffer.on :-- array for shutdown

BOOL first.buffer.on, last.buffer.on, shutdown.active :
VAL terminate IS 199 (BYTE) : -- constant for shutdown
VAL BOOL otherwise IS TRUE:

PAR
    -- get next input from B012 or from B004 for shutdown
PAR
SEQ
    shutdown.active := TRUE
    WHILE shutdown.active
    ALT
    shutdown ? byte.sd
    SEQ
    to.buffer ! byte.sd
shutdown.active := FALSE
into.buffer ? byte.sd
to.buffer ! byte.sd

-- place either in first buffer space,
-- terminate process if shutdown
SEQ
  first.buffer.on := TRUE
  WHILE first.buffer.on
  SEQ
    to.buffer ? byte.in
    b[0] ! byte.in
    IF
      byte.in = terminate
      first.buffer.on := FALSE
    otherwise
      SKIP
-- replicated PAR for multi stage buffer.
-- Each terminated upon shutdown
PAR
  PAR p = 0 FOR (buffer.size-2)
  SEQ
    buffer.on[p] := TRUE
    WHILE buffer.on[p]
    SEQ
      b[p] ? byte.hold[p]
      b[p+1] ! byte.hold[p]
      IF
        byte.hold[p] = terminate
        buffer.on[p] := FALSE
      otherwise
        SKIP
-- final buffer stage for delivery to STATUS process
SEQ
  last.buffer.on := TRUE
  WHILE last.buffer.on
  SEQ
    b[buffer.size-2] ? byte.out
    outof.buffer ! byte.out
    IF
      byte.out = terminate
      last.buffer.on := FALSE
    otherwise
      SKIP
: -- end of buffer.in

PROC statusdb (CHAN OF INT keyboard,
                CHAN OF ANY screen, up.in, up.out, shutdown,
                VAL INT no.trams)
#USE "intcmds.tsr"
#USE "ctrlcmds.tsr"

VAL terminate IS 199 (BYTE):
BYTE command.byte, byte1, byte2, link.byte:
BOOL more, send.quit:

TIMER clock:
[17]INT num.used: -- array for # of links
[37][6]INT counter: -- array for # of commands
[6]INT total, grand: -- totals and subtotals
[36]INT user.done: -- track TRAM USER done

INT key.in, done.count, sum, sumtot, grandtot:
INT num.acks, num.reqs, num.brks, num.abts, num.reis:
INT line.num, xpos.inc, asci.inc, offset:
INT link1, link2, links.used:
INT start, stop, elapsed, delay, now:

----------------------------------------
-- translate input byte from BYTE
-- to english quivielant for display
----------------------------------------

PROC xlate.byte (CHAN OF ANY screen, VAL BYTE x.byte,
VAL INT xpos)

#USE "intcmds.tsr"
#USE "ctrlcmds.tsr"
SEQ
goto.xy(screen,xpos,0)
IF
-- bytes 0-31 passed correspond to C004 link nos.
x.byte = (BYTE 0)
  write.full.string (screen, "-Slot 2 Link 3 ")
x.byte = (BYTE 1)
  write.full.string (screen, "-Slot 5 Link 3 ")
x.byte = (BYTE 2)
  write.full.string (screen, "-Slot 1 Link 3 ")
x.byte = (BYTE 3)
  write.full.string (screen, "-Slot 5 Link 0 ")
x.byte = (BYTE 4)
  write.full.string (screen, "-Slot 2 Link 0 ")
x.byte = (BYTE 5)
  write.full.string (screen, "-Slot 1 Link 0 ")
x.byte = (BYTE 6)
  write.full.string (screen, "-Slot 6 Link 0 ")
x.byte = (BYTE 7)
  write.full.string (screen, "-Slot 6 Link 3 ")
x.byte = (BYTE 8)
  write.full.string (screen, "-Slot 15 Link 0 ")
x.byte = (BYTE 9)
  write.full.string (screen, "-Slot 8 Link 0 ")
x.byte = (BYTE 10)
write.full.string (screen, "-Slot 15 Link 3 ")
x.byte = (BYTE 11)
write.full.string (screen, "-Slot 11 Link 3 ")
x.byte = (BYTE 12)
write.full.string (screen, "-Slot 12 Link 0 ")
x.byte = (BYTE 13)
write.full.string (screen, "-Slot 12 Link 3 ")
x.byte = (BYTE 14)
write.full.string (screen, "-Slot 8 Link 3 ")
x.byte = (BYTE 15)
write.full.string (screen, "-Slot 11 Link 0 ")
x.byte = (BYTE 16)
write.full.string (screen, "-Slot 7 Link 0 ")
x.byte = (BYTE 17)
write.full.string (screen, "-Slot 3 Link 3 ")
x.byte = (BYTE 18)
write.full.string (screen, "-Slot 4 Link 0 ")
x.byte = (BYTE 19)
write.full.string (screen, "-Slot 3 Link 0 ")
x.byte = (BYTE 20)
write.full.string (screen, "-Slot 7 Link 3 ")
x.byte = (BYTE 21)
write.full.string (screen, "-Slot 4 Link 3 ")
x.byte = (BYTE 22)
write.full.string (screen, "-Slot 0 Link 0 ")
x.byte = (BYTE 23)
write.full.string (screen, "-Slot 0 Link 3 ")
x.byte = (BYTE 24)
write.full.string (screen, "-Slot 14 Link 3 ")
x.byte = (BYTE 25)
write.full.string (screen, "-Slot 13 Link 0 ")
x.byte = (BYTE 26)
write.full.string (screen, "-Slot 14 Link 0 ")
x.byte = (BYTE 27)
write.full.string (screen, "-Slot 13 Link 3 ")
x.byte = (BYTE 28)
write.full.string (screen, "-Slot 9 Link 0 ")
x.byte = (BYTE 29)
write.full.string (screen, "-Slot 9 Link 3 ")
x.byte = (BYTE 30)
write.full.string (screen, "-Slot 10 Link 3 ")
x.byte = (BYTE 31)
write.full.string (screen, "-Slot 10 Link 0 ")
-- translation for link control commands and NIL
x.byte = byte.nil
write.full.string (screen, "-Nil Byte ")
x.byte = link.ack
write.full.string (screen, "-link.ack ")
x.byte = link.rel
write.full.string (screen, "-link.rel ")
x.byte = link.req
write.full.string (screen, "-link.req")

x.byte = link.brk
write.full.string (screen, "-link.brk")

x.byte = link.abt
write.full.string (screen, "-link.abt")

x.byte = link.rei
write.full.string (screen, "-link.rei")

-- miscellaneous signals used in control & testing

x.byte = failed
write.full.string (screen, "-failed")

x.byte = reinit
write.full.string (screen, "-reinit")

x.byte = done.with.link
write.full.string (screen, "-done.with.link")

x.byte = all.done
write.full.string (screen, "-all.done")

x.byte = ready.to.rcv
write.full.string (screen, "-ready.to.rcv")

x.byte = link.made
write.full.string (screen, "-link.made")

x.byte = link.gone
write.full.string (screen, "-link.gone")

x.byte = (BYTE 199)
write.full.string (screen, "-terminated echo")

-- display TRAM DONE notices when received
((x.byte)>(BYTE(99))) AND ((x.byte)<(BYTE(136)))
SEQ
  goto.xy(screen,0,17)
  write.full.string (screen, "-Tram")
  write.int(screen,(INT(x.byte))-100,3)
  write.full.string (screen, " done")
  newline(screen)

-- display BAD EDGE notices when received
((x.byte)>(BYTE(199))) AND ((x.byte)<(BYTE(216)))
SEQ
  goto.xy(screen,20,20)
  write.full.string (screen, "-Edge")
  write.int(screen,(INT(x.byte))-200,3)
  write.full.string (screen, " bad")
  newline(screen)

otherwise
-- if unknown byte not successfully translated
SEQ
  goto.xy(screen,20,18)
  write.full.string(screen,"==UNKNOWN BYTE==")
  write.int(screen,INT(x.byte),6)
: -- end of xlate byte
-- PROC handle.screen -- for repeated screen
-- updates for various commands received
-----------------------------------------------
PROC handle.screen (VAL BYTE link.byte,
                      VAL IN: link1, link2, ln1, ln2)

SEQ
  -- place the first byte on screen (source)
  IF
    link1 < 16  -- a link 0
    SEQ
      line.num := ln1
      xpos.inc := 0
  otherwise  -- a link 3
    SEQ
      line.num := ln2
      xpos.inc := 20
  goto.xy(screen, ((link1-xpos.inc)*3)+14, line.num)
  write.char(screen, link.byte)

  -- place the second byte on screen (destination)
  IF
    link2 < 16  -- a link 0
    SEQ
      line.num := ln1
      xpos.inc := 0
  otherwise  -- a link 3
    SEQ
      line.num := ln2
      xpos.inc := 20
  goto.xy(screen, ((link2-xpos.inc)*3)+14, line.num)
  write.char(screen, link.byte)

: -- end of handle.screen

-- main code for B004 monitor
-----------------------------------------------
SEQ
  -- initialize loop
  -- initialize all counters used
  num.acks := 0
  num.reqs := 0
  num.brks := 0
  num.abts := 0
  num.reis := 0
  more := TRUE
  done.count := 0
  links.used := 0
SEQ i = 0 FOR 36
    user.done[i] := 0
delay := 32768
SEQ i = 0 FOR 17
    num.used[i] := 0

-- initialize display format
SEQ i = 0 FOR 20
    newline(screen)
goto.xy(screen, 0, 1)
write.full.string(screen, "Slot Design: A B C D E")
write.full.string(screen, "F G H I J K L M N O P")
newline(screen)
write.full.string(screen, "Slot # ").
SEQ i = 0 FOR 16
    write.int(screen, i, 3)

-- initialize display columns
newline(screen)
newline(screen)
write.full.string(screen, "Link 0 REQ ")
newline(screen)
write.full.string(screen, "Link 3 REQ ")
newline(screen)
write.full.string(screen, "Link 0 ACK ")
newline(screen)
write.full.string(screen, "Link 3 ACK ")
newline(screen)
write.full.string(screen, "Link 0 BRK ")
newline(screen)
write.full.string(screen, "Link 3 BRK ")
newline(screen)
write.full.string(screen, "Link 0 ABT ")
newline(screen)
write.full.string(screen, "Link 3 ABT ")
newline(screen)
write.full.string(screen, "Link 0 REI ")
newline(screen)
write.full.string(screen, "Link 3 REI ")

-- initialize edge status display
goto.xy(screen, 0, 21)
write.full.string(screen, "Edge # ")
SEQ i = 0 FOR 16
    write.int(screen, i, 3)
newline(screen)
goto.xy(screen,13,22)
SEQ i = 0 FOR 16
  write.full.string(screen," + ")

-- initialize # of edges used display
goto.xy(screen,64,0)
write.full.string(screen,"#Used Count")
SEQ i = 1 FOR 16
  SEQ
goto.xy(screen,65,i)
  write.int(screen,i,3)

WHILE more
  SEQ
    up.in ? command.byte -- get next byte in pipe
    IF
      -- take appropriate display and counter action
      -- for each command received from B012
      (((INT(command.byte)) > (39)) AND
        ((INT(command.byte)) < (46)))
      SEQ
        -- command byte, get next two parameters
        up.in ? bytel; byte2
        up.out ! command.byte; byte1; byte2
        counter[to.slot[INT(byte1)]]
        [([INT(command.byte)]-40) :=
        counter[to.slot[INT(byte1)]]
        [([INT(command.byte)]-40) + 1
        xlate.byte(screen,command.byte,0)
        xlate.byte(screen,bytel,15)
        xlate.byte(screen,byte2,35)

-- determine characters used on screen for
-- display of source & dest
IF
  ((INT(byte1)) < 32)
    link1 := to.slot[INT(byte1)]
  otherwise
    link1 := 10

IF
  ((INT(byte2)) < 32)
    link2 := to.slot[INT(byte2)]
  otherwise
    link2 := 30

IF
  link1 < 16
link.byte := BYTE(link1+65)
otherwise
link.byte := BYTE(link1+77)

IF
command.byte = link.req
SEQ
handle.screen(link.byte, link1, link2, 4, 5)
num.reqs := num.reqs + 1
goto.xy(screen, 4, 6)
write.int(screen, num.reqs, 8)

command.byte = link.ack
SEQ
handle.screen(link.byte, link1, link2, 7, 8)
num.acks := num.acks + 1
goto.xy(screen, 4, 9)
write.int(screen, num.acks, 8)
links.used := links.used + 1
num.used[links.used] :=
    num.used[links.used] + 1
goto.xy(screen, 70, links.used)
write.int(screen, num.used[links.used], 6)

command.byte = link.brk
SEQ
handle.screen(link.byte, link1, link2, 10, 11)
handle.screen(45(BYTE), link1, link2, 7, 8)
num.brks := num.brks + 1
goto.xy(screen, 4, 12)
write.int(screen, num.brks, 8)
links.used := links.used - 1

command.byte = link.abt
SEQ
handle.screen(link.byte, link1, link2, 13, 14)
handle.screen(63(BYTE), link1, link2, 7, 8)
links.used := links.used - 1
num.abts := num.abts + 1
goto.xy(screen, 4, 15)
write.int(screen, num.abts, 8)
command.byte = link.rei
SEQ
  handle.screen(link.byte,link1, link2,16,17)
  handle.screen(33(BYTE),link1,link2,7,8)
num.reis := num.reis + 1
goto.xy(screen, 4,18)
write.int(screen,num.abts,8)

otherwise  -- including link.rel's
  SKIP

-- handle various user keystrokes for
-- stepping speed, pause, etc.
clock ? now
PRI ALT
  keyboard ? key.in
  IF
    (key.in = 81) OR (key.in = 113)
    -- "Q": quit
    SEQ
      send.quit := TRUE
      more := FALSE
      shutdown ! terminate
    (key.in = 43)
    -- "+" = faster
      delay := delay / 2
    (key.in = 45)
    -- "-" = slower
      delay := delay * 2
    (key.in = 82) OR (key.in = 114)
    -- "R": resend a byte
      up.out ! BYTE(60)
  otherwise
    send.quit := FALSE
    clock ? AFTER now PLUS (delay * 100)
  SKIP

-- handle TRAM (USER) DONE notices
(((INT(command.byte)) > (99)) AND
(((INT(command.byte)) < (116))
  -- a tram has sent a notice it is done
  -- xmitting on link0&3
SEQ
goto.xy(screen, 13+((INT(command.byte)) - 100)*3), 4)
write.char(screen, 2(BYTE))

((INT(command.byte)) > (119)) AND
((INT(command.byte)) < (136)))
-- a tram has sent a notice it is done
-- xmitting on link0&3
SEQ
  goto.xy(screen, 13+((INT(command.byte)) - 120)*3, 5)
  write.char(screen, 2(BYTE))

-- handle EDGE BAD info from T21x (via input buf)
(((INT(command.byte)) > (199)) AND
((INT(command.byte)) < (216)))
-- an edge status has changed to GOOD
SEQ
  goto.xy(screen, 14+((INT(command.byte)) - 200)*3, 22)
  write.char(screen, 43(BYTE)) -- change to "+

(((INT(command.byte)) > (219)) AND
((INT(command.byte)) < (236)))
-- an edge status has changed to BAD
SEQ
  goto.xy(screen, 14+((INT(command.byte)) - 220)*3, 22)
  write.char(screen, 45(BYTE)) -- change to "-
otherwise
-- handle unknown bytes received (for testing)
SEQ
  IF
    command.byte = terminate
    up.out ! terminate
  otherwise
    xlate.byte(screen, command.byte, 0)

-- terminate all processes in this monitoring program,
-- including all replicated PARs used in buffers
IF
  send.quit = TRUE
  SEQ -- flush queues
    WHILE (command.byte <> terminate)
      up.in ? command.byte
      up.out ! terminate
    otherwise
      SKIP
: -- end of B004 Monitor

142
PROC buffer.out (CHAN OF ANY into.buffer, outof.buffer)

VAL INT buffer.size IS 3:
BYTE byte.in, byte.out:
[buffer.size-1]CHAN OF ANY b:
[buffer.size-2]BYTE byte.hold:
[buffer.size-2]BOOL buffer.on:
BOOL first.buffer.on, last.buffer.on:
VAL terminate IS 199 (BYTE):
VAL BOOL otherwise IS TRUE:

PAR
SEQ
  first.buffer.on := TRUE
  WHILE first.buffer.on
    SEQ
      into.buffer ? byte.in
      b[0] ! byte.in
      IF
        byte.in = terminate
        first.buffer.on := FALSE
        otherwise
        SKIP
    PAR
    PAR p = 0 FOR (buffer.size-2)
    SEQ
      buffer.on[p] := TRUE
      WHILE buffer.on[p]
        SEQ
          b[p] ? byte.hold[p]
          b[p+1] ! byte.hold[p]
          IF
            byte.hold[p] = terminate
            buffer.on[p] := FALSE
            otherwise
            SKIP
      SEQ
      last.buffer.on := TRUE
      WHILE last.buffer.on
        SEQ
          b[buffer.size-2] ? byte.out
          outof.buffer ! byte.out
          IF
            byte.out = terminate
            last.buffer.on := FALSE

143
otherwise
SKIP
: -- end of buffer.out

PAR
  buffer.in(up.in, buffer.to.echo, shut.down)
  statusdb(keyboard, screen, buffer.to.echo,
            echo.to.buffer, shut.down, no.trams)
  buffer.out(echo.to.buffer, up.out)
LIST OF REFERENCES


10. INMOS Technical Note 17, Performance Maximization, by Phil Atkin, Bristol, United Kingdom, March 1987.

11. INMOS Technical Note 18, Connecting INMOS Links, by Michael Rygol and Trevor Watson, Bristol, United Kingdom.

12. INMOS Technical Note 1, Extraordinary Use Of Transputer Links, by Roger Shepherd, Bristol, United Kingdom.

13. INMOS Technical Note 27, Lies, Damned Lies And Benchmarks, by Roger Shepherd and Peter Thompson, Bristol, United Kingdom.


15. Burns, Alan, Programming In Occam 2, University of Bradford, Addison-Wesley Publishing Company, 1988


20. INMOS Technical Note 49, Module Motherboard Architecture, by Trevor Watson, Bristol, United Kingdom.


23. INMOS Technical Note 11, IMS B004 IBM PC Add-In Board, by Stephen Ghee, Bristol, United Kingdom.

<table>
<thead>
<tr>
<th>No.</th>
<th>Distribution List</th>
</tr>
</thead>
</table>
| 1.  | Defense Technical Information Center  
     Cameron Station  
     Alexandria, VA  22304-6145 |
| 2.  | Library, Code 0142  
     Naval Postgraduate School  
     Monterey, CA  93943-5002 |
| 3.  | Department Chairman, Code 52  
     Department of Computer Science  
     Naval Postgraduate School  
     Monterey, CA  93943 |
| 4.  | Dr. Uno R. Kodres, Code 52Kr  
     Department of Computer Science  
     Naval Postgraduate School  
     Monterey, CA  93943 |
| 5.  | Dr. Gregory Buzzard, Code 52Bu  
     Department of Computer Science  
     Naval Postgraduate School  
     Monterey, CA  93943 |
| 6.  | Daniel Green, Code 20F  
     Naval Surface Weapons Center  
     Dahlgren, VA  22449 |
| 7.  | Jerry Gaston, Code N24  
     Naval Surface Weapons Center  
     Dahlgren, VA  22449 |
| 8.  | Captain J. Hood, USN  
     PMS 400B5  
     Naval Sea Systems Command  
     Washington, DC  20362 |
| 9.  | RCA AEGIS Repository  
     RCA Corporation  
     Government System Division  
     Mail STop 127-327  
     Moorestown, NJ  08057 |
<table>
<thead>
<tr>
<th></th>
<th>Name and Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Dr. M. J. Gralia</td>
</tr>
<tr>
<td></td>
<td>Applied Physics Laboratory</td>
</tr>
<tr>
<td></td>
<td>John Hopkins Road</td>
</tr>
<tr>
<td></td>
<td>Laurel, Maryland  20707</td>
</tr>
<tr>
<td>11</td>
<td>Mr. Dana Small, Code 8242</td>
</tr>
<tr>
<td></td>
<td>Naval Ocean Systems Center</td>
</tr>
<tr>
<td></td>
<td>San Diego, CA  92152</td>
</tr>
<tr>
<td>12</td>
<td>Library (Code E33-05)</td>
</tr>
<tr>
<td></td>
<td>Naval Surface Weapons Center</td>
</tr>
<tr>
<td></td>
<td>Dahlgren, VA  22449</td>
</tr>
<tr>
<td>13</td>
<td>Aegis Modeling Laboratory, Code 52</td>
</tr>
<tr>
<td></td>
<td>Department of Computer Science</td>
</tr>
<tr>
<td></td>
<td>Naval Postgraduate School</td>
</tr>
<tr>
<td></td>
<td>Monterey, CA  93943</td>
</tr>
<tr>
<td>14</td>
<td>Mr. Richard Onyett</td>
</tr>
<tr>
<td></td>
<td>INMOS Corporation</td>
</tr>
<tr>
<td></td>
<td>2620 Augustine Drive, Suite 100</td>
</tr>
<tr>
<td></td>
<td>Santa Clara, CA  95054</td>
</tr>
<tr>
<td>15</td>
<td>Lieutenant Winfred P. Pikelis</td>
</tr>
<tr>
<td></td>
<td>Naval Submarine School</td>
</tr>
<tr>
<td></td>
<td>Box 700, Attn: Code 80 SOAC 89080</td>
</tr>
<tr>
<td></td>
<td>Groton, CT  06349-5700</td>
</tr>
<tr>
<td>16</td>
<td>Lieutenant Murat Kilic</td>
</tr>
<tr>
<td></td>
<td>Department of Computer Science</td>
</tr>
<tr>
<td></td>
<td>Naval Postgraduate School</td>
</tr>
<tr>
<td></td>
<td>Monterey, CA  93943</td>
</tr>
<tr>
<td>17</td>
<td>Mr. John Waite</td>
</tr>
<tr>
<td></td>
<td>PMTC Code 1051</td>
</tr>
<tr>
<td></td>
<td>Point Mugu, CA  93042</td>
</tr>
</tbody>
</table>