inmos®

THE T9000 TRANSPUTER PRODUCTS OVERVIEW MANUAL

First Edition 1991

SGS-THOMSON
MICROELECTRONICS
INMOS is a member of the SGS-THOMSON Microelectronics Group

INMOS Databook series

Transputer Databook

Military and Space Transputer Databook

Transputer Development and iq Systems Databook

Graphics Databook

Transputer Applications Notebook: Architecture and Software

Transputer Applications Notebook: Systems and Performance

The T9000 Transputer Products Overview Manual

Contents overview

Contents

Preface

The T9000 Transputer Products Overview Manual introduces the latest member of the transputer range of microprocessors, the IMS T9000. Transputers are designed to provide extremely high performance in single processor applications and are also designed with hardware and software features for the construction of multiprocessing systems.

Other transputer products include the IMS T225, a 16 bit microprocessor, the 32 bit IMS T425 and the IMS T8xx series, which are 32 bit microprocessors with an on-chip 64 bit floating point processor. Details of these and their support devices can be found in *The Transputer Databook*, which is available as a separate publication. Other transputer related documents, including various application and technical notes, are also available from INMOS.

This manual consists of two parts; an overview section and a set of more detailed documents for the first members of the new product range. Part 1, the overview, introduces the transputer architecture and then the features and benefits of the IMS T9000 family. Part 2 contains preliminary information on the IMS T9000 transputer, the IMS C104 packet routing switch and the IMS C100 system protocol converter. This is advance information and is subject to change.

More detailed documentation on the IMS T9000 family is in preparation. This will include a hardware reference manual, a programmers reference manual, a system networking manual and various application notes. Documentation for systems and software products will also be updated to reflect added support for the IMS T9000. For the latest information, contact your local SGS-THOMSON sales outlet.

Software and hardware examples given in this book are outline design studies and are included to illustrate various ways in which transputers can be used. The examples are not intended to provide accurate application designs.

In addition to transputer products the INMOS product range also includes development systems, systems products and high performance graphics devices. For further information regarding INMOS products please contact your local SGS-THOMSON sales outlet.
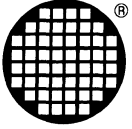
Part 1

Product Family Overview

1   Introducing the INMOS IMS T9000 family

The INMOS IMS T9000 is the latest member of the transputer family. It is designed to provide far higher performance and greatly improved communications facilities.

INMOS has used advanced CMOS technology to integrate a 32-bit integer processor, a 64-bit floating point processor, 16 Kbytes of cache memory, a communications processor and four high bandwidth serial communications links on a single IMS T9000 chip.

The IMS T9000 transputer excels in real-time embedded applications, delivering exceptional single processor performance and scaleable multiprocessor capability.

The IMS T9000 is binary compatible with previous transputers. It extends the transputer range, making it easy to upgrade and complement existing transputer systems. There is extensive, industry standard software support for all members of the transputer family; this includes high level language compilers, systems software (such as real time operating systems) as well as an extensive range of development tools.

1.1   Performance

It is essential that any microprocessor family designed for the embedded system market provides the required performance at low cost.

The transputer family includes a 16 bit processor, a 32 bit range of fast integer and floating point processors and now, the highest performance member of the family, the IMS T9000. These are all designed to make it easy to design low cost, high performance systems.

- **Single processor performance:** the IMS T9000 transputer boasts exceptional single processor performance; the new superscalar CPU is capable of a peak performance of 200 MIPS and 25 MFLOPS.

- **Real-time performance:** the IMS T9000 offers sub-microsecond interrupt response and context switch times, making it ideal for high performance real-time systems.

- **Communications performance:** the four IMS T9000 communication links provide a total of 80 Mbytes/second bidirectional bandwidth.

- **Multiprocessor performance:** the interprocessor communications architecture gives scaleable performance – the ability to increase the performance of a system by adding more processors.

- **Usable performance:** the IMS T9000 implementation makes it easy for compilers to fully exploit the superscalar performance using a range of industry standard programming languages.

- **Price/performance:** the IMS T9000 offers supercomputer performance at an embedded systems price.

1.2   Multiprocessing

For applications that demand performance that single processors cannot provide, the IMS T9000 has complete on-chip support for multiprocessing:

- **Hardware scheduler:** the transputer architecture includes instruction level support for the creation and scheduling of any number of concurrent processes

- **Inter-process communication:** the transputer instruction set includes instructions for communicating between concurrent processes. The same instructions are used to communicate between processes running on a single transputer and between processes running on separate transputers.

- **Inter-processor communication subsystem:** the presence of a dedicated communications processor which operates concurrently with the main processor, makes interprocessor communica-

tions flexible and efficient. The integration of the communications system on-chip makes it easy to write programs for multiprocessor systems

- **System control and monitoring:** all the IMS T9000 transputers in a system can be initialized, loaded with code and monitored for errors through a completely independent communications system.

1.3   Communications support devices

The IMS T9000 transputer is complemented by a range of communications peripherals that extend the communications capabilities of the IMS T9000. The IMS C1XX family ensures that any size of IMS T9000 system can be constructed, connecting first generation and second generation transputers and providing an interface to the outside world.

- **IMS C104 packet routing switch:** the IMS C104 is a complete routing switch on a single chip. The IMS C104 connects 32 links to each other via a 32 by 32 way, non-blocking crossbar switch with sub-microsecond latency. This allows simple, fast communication between IMS T9000 transputers that are not directly connected. Multiple IMS C104s can be connected together to make larger and more complex networks, linking any number of IMS T9000 transputers, or any other devices that use the link protocol.

- **The IMS C100 system protocol convertor:** the IMS C100 system protocol convertor converts between the first generation transputer links and control signals and the new IMS T9000 protocol.The IMS C100 provides an inter-networking solution for transputer systems, allowing networks to be constructed using the optimum mix of transputers to satisfy processing power, communication bandwidth and system cost.

1.4   Software

The success of any microprocessor is determined as much by the quality of its software development tools as by any other feature.

INMOS has over a decade of experience in developing software tools for transputers and for multiprocessing systems. The range of compilers and powerful development tools support all the requirements of software developers.

- **Compatibility:** instruction set compatibility with the first generation transputer family means that the IMS T9000 transputer has inherited a significant range of development and application software.

- **The transputer toolset:** the transputer toolset is a set of development tools for programming, configuring and debugging mixed transputer systems.

  The toolset is available on a variety of host computers including:

  - IBM PC

  - NEC PC

  - VAX

  - Sun 3

  - Sun 4

- **Debugger:** INMOS provides a powerful, interactive debugger for debugging programs running on networks of transputers. This provides full source level debugging with the ability to set breakpoints in any process and on any processor, and then to examine the state of the stopped process as well as the low-level state of the processor.

- **Compilers:** for fast time to market, and to satisfy the diverse programming requirements of different application areas, the toolset can be used with a variety of industry standard compilers, all with major support for multiprocessing.

  The IMS T9000 is supported by a range of compilers including:

    - ANSI C

    - C++

    - Fortran

    - occam

    - Ada

- **System software:** system software support for the IMS T9000 reflects the requirements of the embedded systems marketplace. A range of operating systems and real-time kernels are available for the transputer including:

    - C Executive

    - VRTX

    - CHORUS distributed Unix

This impressive array of development tools, industry standard compilers and system software satisfies the demands of the embedded systems market. It also ensures that the user can benefit from a significant reduction in the critical time to market.

1.5   Applications

The transputer family provides unprecedented price/performance solutions for a wide range of embedded systems applications.

The IMS T9000 transputer has been specifically developed to satisfy the requirements of three segments of the embedded systems market:

- **Imaging:** the imaging market comprises applications that involve the generation, manipulation and transmission of image data. Such applications include:

    - Laser printers

    - Graphics systems

    - Image processing systems

    - Industrial inspection systems

    - Robotics

- **Embedded computing:** the embedded computing market comprises applications that are run within a computer environment and add overall performance and functionality to the computer system. Such applications include:

    - Application accelerators: (graphics, numerical, scientific, DTP)

    - Disk arrays and high performance file servers

    - Databases

    - X terminals

- Supercomputers

- Factory automation

- **Communications:** the embedded communications market can be segmented into two main areas that require high performance microprocessors. These are:

  - Networking: low cost LAN interfacing – FDDI, Ethernet; internetworking systems – bridges, gateways and routers.

  - Packet switching systems.

The IMS T9000 transputer is highly applicable to the communications market due to its integrated architecture combining high performance CPU and communication links with a packet based protocol. The IMS C104 packet routing switch has been designed to support the IMS T9000, and is useful in a range of telecommunications switching applications.

The transputer family provides a range of price/performance solutions for all the above applications.

2 The IMS T9000 transputer

The IMS T9000 is the latest member in the transputer family of high performance microprocessors. It is part of a broad range of 16 and 32 bit microprocessors with compatible instructions sets and interfaces. As well as providing high performance processing, they are designed to be simple to use and enable the construction of low cost systems. Transputers include functions to enable multitasking on a single processor and the building of multiprocessor systems.

2.1 Overview
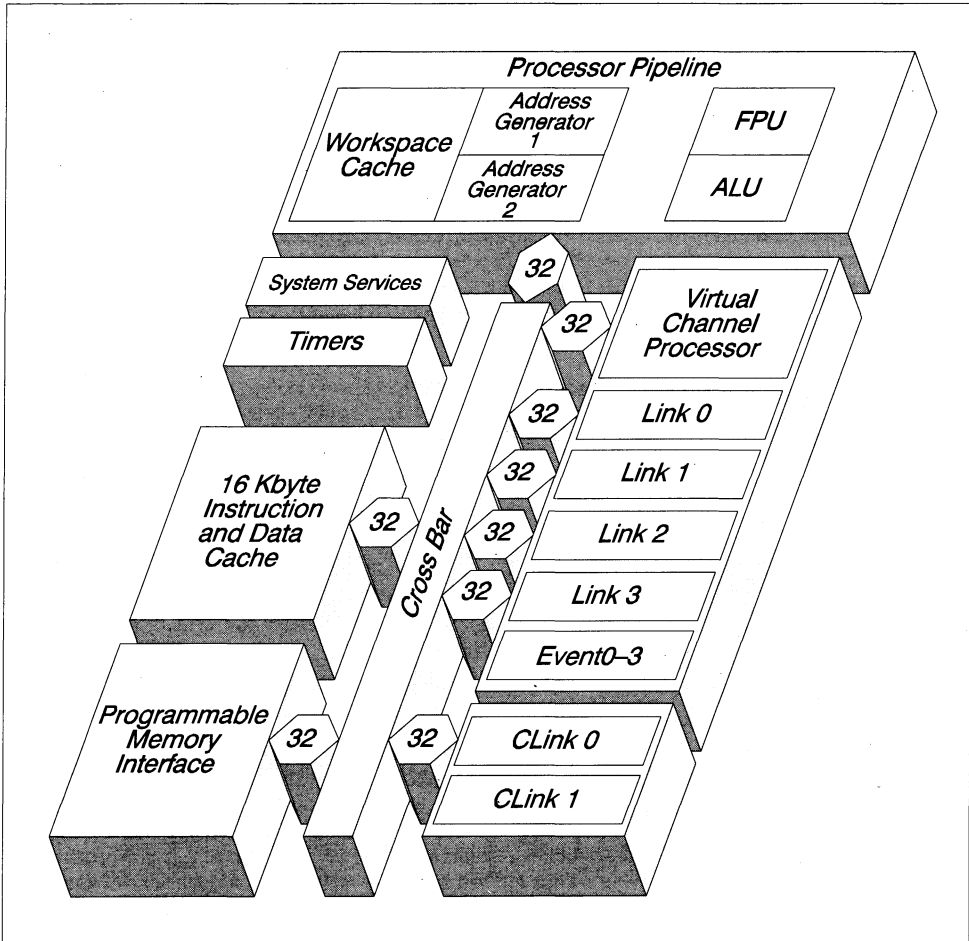
Figure 2.1   Block diagram of IMS T9000

The IMS T9000 integrates a high performance central processing unit (CPU), a 16 Kbyte cache, communications system and other support functions on a single chip. The main functional blocks of the IMS T9000 are shown in figure 2.1. The function of each of these is outlined below, more details will be found in the following sections.

Processor

The IMS T9000 CPU contains a 32 bit arithmetic and logic unit (ALU) and a 64 bit floating point unit (FPU). The FPU operates on 32 and 64 bit floating point numbers as specified by the IEEE 754 standard. The CPU

also includes instructions for byte and half word operations. The CPU uses 32 bit linear addressing and can address up to 4 Gbytes of memory.

The IMS T9000 is binary compatible with previous transputers. In particular it implements the same instruction set as the IMS T805 [1] with many additions.

The instruction set is designed for efficient execution of compiled code and there is a wide range of language compilers available for the transputer including a Plum-Hall validated ANSI C compiler, a validated Ada compiler and Fortran, occam and C + + compilers. These are complemented by a full set of software tools for developing and debugging programs for single transputers and networks of transputers. In addition there are a number of system level software products, such as real time kernels and distributed operating systems.

The transputer includes a hardware kernel for scheduling processes and performing communications. These operations are directly supported in the instruction set.

The IMS T9000 can run code in protected mode. In this mode all memory accesses are made through a memory management unit which checks and translates addresses before using them to address the memory system. Further, only a subset of the full instruction set may be executed, preventing protected code from executing privileged instructions.

There is improved support for error handling over earlier transputers; errors can be trapped and handled independently for each process in addition to the global error handling provided previously.

Hierarchical memory system

The IMS T9000 includes a 16 Kbyte unified cache to provide single cycle access to instructions and data. The cache provides a peak bandwidth of 200 Mwords/s. The CPU also includes another small cache for the most frequently used local variables of a program which provides another 150 Mwords/s of memory bandwidth.

The external memory interface is highly programmable, allowing large memory systems, containing different types of devices, to be built with little or no external logic. There are four independent sets of memory control signals simplifying the use of different device types in the same system. The memory can interface to 8, 16, 32 or 64 bit wide devices. The maximum data transfer rate across the memory interface is 50 Mwords/s.

Communications system

An important issue in multiprocessor system design is the communications architecture. To achieve efficiency and ease of use, communications must be properly integrated into the entire processor architecture.

The transputer hardware and instruction set provides simple and efficient communications between processes and between processors. Both internal and external communications are handled identically, using the same source code and machine instructions.

To support interprocessor communications, there is a complete communications subsystem on chip. This includes four 100 Mbits/s full-duplex, serial communication links each with its own pair of direct memory access (DMA) channels. The links can be directly connected between transputers with no external buffering or other glue logic. The use of serial links simplifies routing of links on a circuit boards and the interconnection of boards in a system. A communications processor, which manages all link communications, operates concurrently with the main CPU so that data transfers do not adversely affect CPU operation.

Two additional links are provided for system control and monitoring. Initialization and booting of the processor can optionally be done through these links.

The communications subsystem also includes four 'Event' channels. As well as acting as interrupt inputs, these can be used, as inputs or outputs, for more general synchronization and signalling.

Multiple internal buses

To support the high degree of concurrent operation on the IMS T9000, and to maintain the high internal data rates required, there are four sets of 32 bit address and data buses internally. These provide multi-port access to the on-chip cache from the various functional units of the IMS T9000.

System services

The system services section provides all the general facilities necessary for the operation of the transputer. This includes the power and ground connections, and the clock input (5 MHz). Other important connections are a capacitor, which is required for the on-chip phase locked loops which generate all the internal high frequency clocks, and the processor speed select pins which can be used to select the frequency of the internal clocks (up to the maximum speed for a particular device). There is also a reset input – however, as the IMS T9000 includes on-chip power-on reset circuitry, external reset logic may not be required in an embedded control application.

2.2 The transputer architecture

An important design decision was that transputers should be programmed in a high level language. The instruction set has, therefore, been designed for simple and efficient compilation.The instructions are all of the same format and chosen to give a compact representation of the operations most frequently occurring in programs.

The CPU of the IMS T9000 contains three registers (**Areg**, **Breg** and **Creg**) used for expression evaluation, which form a hardware stack. Loading a value into the stack pushes **Breg** into **Creg**, and **Areg** into **Breg**, before loading **Areg**. Storing a value from **Areg** pops **Breg** into **Areg**, and **Creg** into **Breg**. Similarly, the FPU includes a three register floating-point evaluation stack. When values are loaded onto, or stored from, the stack the floating-point registers push and pop in the same way as the **Areg**, **Breg** and **Creg** registers. Analysis of a large number of programs, shows that 3 registers provides an effective balance between code compactness and implementation complexity.
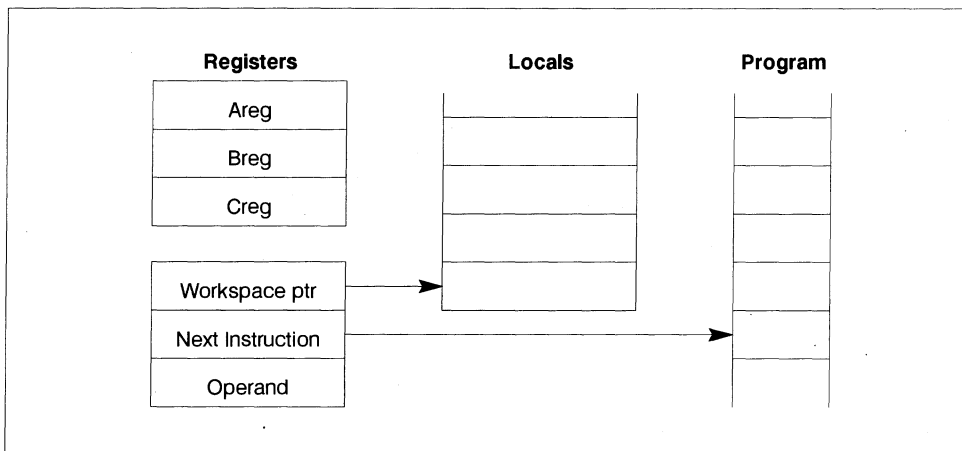
Figure 2.2   Processor registers and memory

The transputer has three other registers used when executing code. These are:

- The instruction pointer which points to the next instruction to be executed.

- The workspace pointer which points to an area of store where local variables are kept. This area is also used as a stack for procedure calls, etc.

- The operand register which is used in the formation of instruction operands.

The addresses of floating-point values are formed on the CPU stack, and values are transferred between the addressed memory locations and the FPU stack under the control of the CPU.

Most transputer functions use the contents of these stacks, and most instructions reference the stacks implicitly. For example the *add* instruction adds the top two values in the CPU stack, leaving the result on the

top of the stack. The use of a stack reduces the need for instructions to specify the location of their operands which reduces the size of instructions and hence of compiled code.

2.3   Support for concurrent processes

Most computers have the ability to effectively run several user tasks or processes concurrently. These processes are created and scheduled by the host operating system. The operating system kernel provides the ability for processes to communicate with the operating system and with each other.

Every transputer includes a hardware kernel with the ability to execute many software processes at the same time, to create new processes rapidly, and to perform communication between processes within a transputer and between processes on different transputers. All of these operations are integrated into the hardware and instruction set of the transputer and are very efficient. Further details of the transputers scheduling mechanism will be found in section 5.

2.4   Pipelined, superscalar implementation

To increase the execution rate of the transputer instruction set, the IMS T9000 is able to issue several instructions per cycle. A superscalar micro-architecture was designed which implements the same high level architecture and instruction set as the IMS T805 but with much higher performance.

Some recent implementations of pipelined and superscalar microprocessors have required very careful programming to obtain the claimed performance. They require that instructions are presented to the pipeline in a sequence that will keep the processor busy. This makes developing effective compilers very difficult, often forcing programmers to resort to assembly code to achieve the required performance. This puts the burden of arranging the correct sequencing of instructions on the programmer, adding to the development time and hence costs of a product.
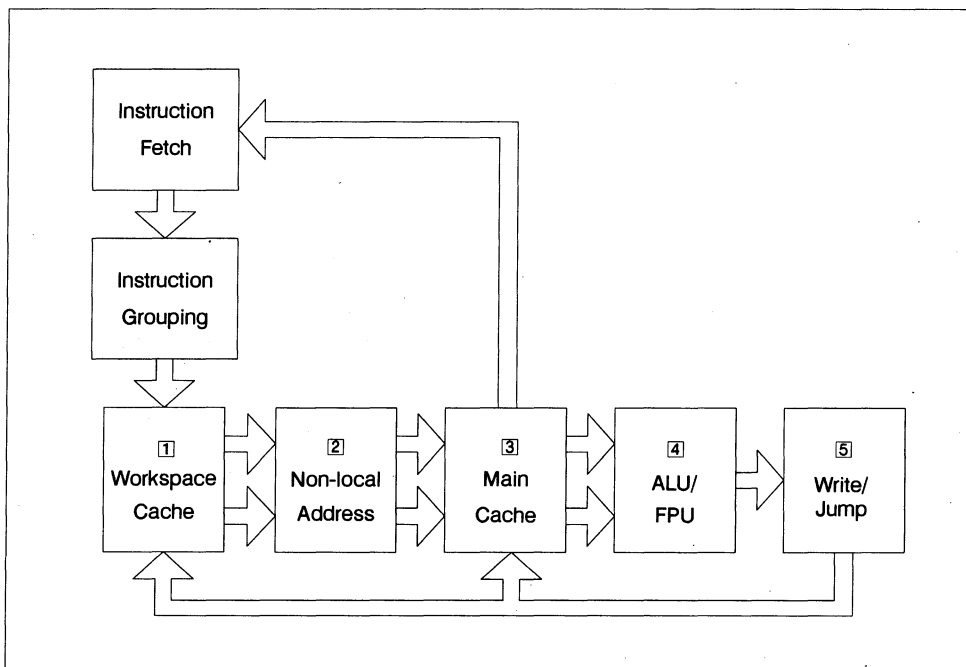
Figure 2.3   Block diagram of grouper and pipeline

The details of the IMS T9000 pipeline are transparent to the programmer. The processor appears to be the simple transputer architecture described above and straightforward code written for that programming model will get nearly the best performance out of the processor. An optimising compiler for the IMS T9000 can, of course, generate more efficient code if the details of the internal architecture are taken into account.

The pipeline

Instructions are executed in a five stage pipeline: the first can fetch two local variables; the second can perform two address calculations, for accessing non-local or subscripted variables; the third stage can load two non-local variables; the next can perform an ALU or FPU operation; and the final stage can do a conditional jump or write.

A conventional pipeline is designed to allow several instructions to be executed simultaneously; different parts of each instruction being handled in different stages of the pipeline. In order to allow multiple instructions to be *issued* per cycle (as well as multiple instructions being executed in each cycle) the IMS T9000 does not simply send a sequence of instructions through the pipeline but has hardware which assembles groups of instructions from the instruction stream. These groups are chosen to make the best use of the available hardware and one group can be sent through the pipeline every cycle. Instructions are put into groups in the order that they arrive at the CPU; dependencies within the group are handled automatically by the pipeline.

The grouper can be thought of as a hardware optimizer; it recognizes commonly occurring code sequences that the processor can execute effectively. The design of the grouping mechanism and the pipeline is based on analysis of the code typically generated by high level language compilers.

An IMS T9000 running at 50 MHz can execute code compiled for the IMS T805 typically 10 times faster than a 20 MHz IMS T805. This means that existing development tools and software can be used to generate code for the IMS T9000. It also means that only a modest amount of work is required to modify compilers to produce code optimized for the IMS T9000.

Grouping of instructions

The grouping of instructions takes advantage of the high degree of concurrency and multiple buses in the processor. For example, both caches are multi-ported and can each support two reads by the CPU simultaneously. This allows two *load local* instructions to go into one group, and the group could also contain two sets of instructions to calculate addresses and fetch non-local variables. These could all be combined with an arithmetic operation such as *add.* More details of the transputer instruction set can be found in [3].

As an example of how the grouper works, consider the assignment and expression evaluation shown below. The code produced is shown along with the number of the pipeline stages in which it is executed.

```
a[i+1] = b[j+15] + c[k+7];
```

ldl	j	1	load local variable j
ldl	b	1	load base address of array b
wsub		2	calculate address of b[j]
ldnl	15	2 3	load value of element b[j+15]
ldl	k	1	
ldl	c	1	
wsub		2	
ldnl	7	2 3	load value of c[k+7]
add		4	add two values on top of stack
ldl	i	1	
ldl	a	1	
wsub		2	
stnl	1	2 5	store into a[i+1]

This code sequence will be executed as three groups (i.e. in 3 cycles) as shown below. The exact contents of each group will depend on the code which precedes and follows this. The first group might contain other instructions from earlier in the instruction stream.

first group	ldl, ldl, wsub, ldnl
second group	ldl, ldl, wsub, ldnl, add
third group	ldl, ldl, wsub, stnl

Since the processor can fetch one word, containing four bytes of instructions and data, in each cycle it is possible to achieve a continuous execution rate of four instructions per cycle (200 MIPS). However, if any of the instructions require more than one cycle to execute, then the instruction fetch mechanism can continue to fetch instructions so that larger groups can be built up. Up to 8 instructions can be put into one group and there may be five groups in the pipeline at any time.

Improvements over IMS T805

In addition to executing several instructions each cycle, the number of cycles required to perform many arithmetic and logical operations has been reduced from previous transputers by adding extra hardware. This, combined with the faster clock speed and new micro-architecture, means a ten-fold increase in speed over the IMS T805.

In addition there is improved support for error handling, and protecting code and data from the errant behavior of a program. The IMS T9000 provides better access to the transputers scheduling mechanism, making it easier to implement software kernels for a particular processing model.

2.5   Hierarchical memory system
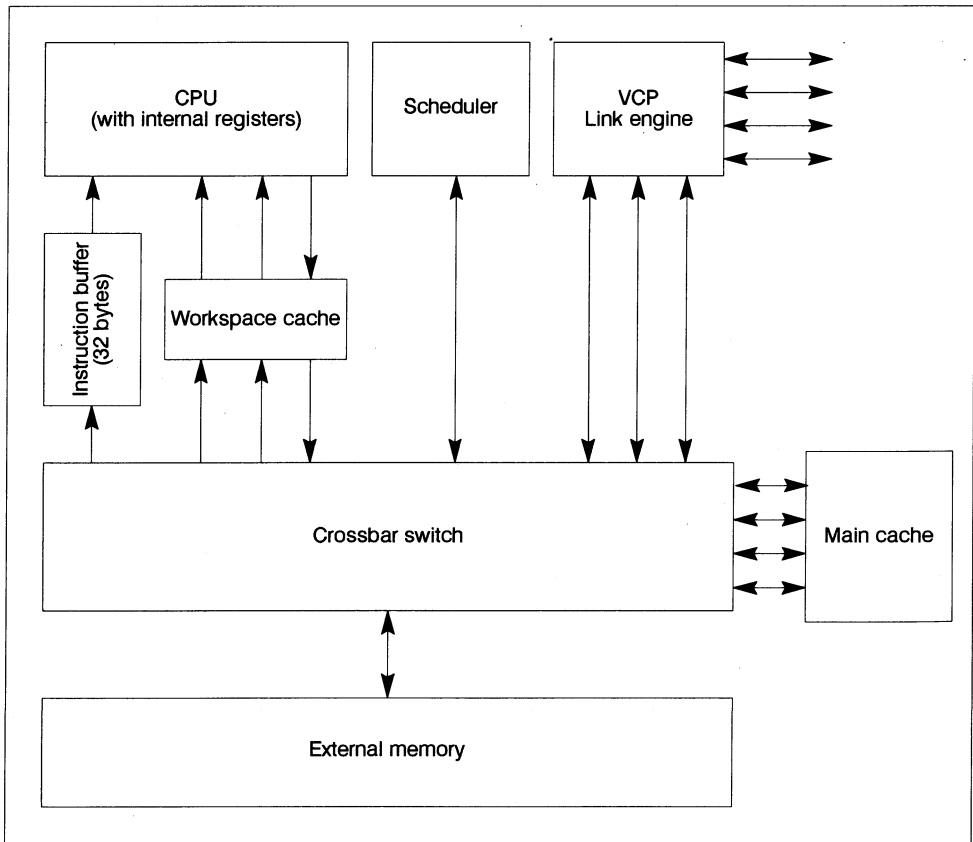
Figure 2.4   IMS T9000 hierarchical memory system

The IMS T9000 has a complete, hierarchical memory system providing fast and efficient access to data and instructions. There are two separate caches on chip, a general purpose unified (code and data) cache and a small cache for local variables.

These caches can provide fast, multi-ported access to data because they are on chip. They also reduce the number and frequency of accesses to external memory, allowing lower cost, slower devices to be used without degrading performance. Finally, because the majority of external memory accesses will be cache refills (and therefore multiple word reads and writes) fast memory access methods, such as page mode, can be used.

2.5.1   Main cache

The main cache consists of four independent banks, each containing 256 lines. Each line holds data from four consecutive words (16 bytes) in memory. An access can be made to every bank on every cycle which, with the multiple internal buses, means there is a very high bandwidth between the cache and different functional units within the IMS T9000.
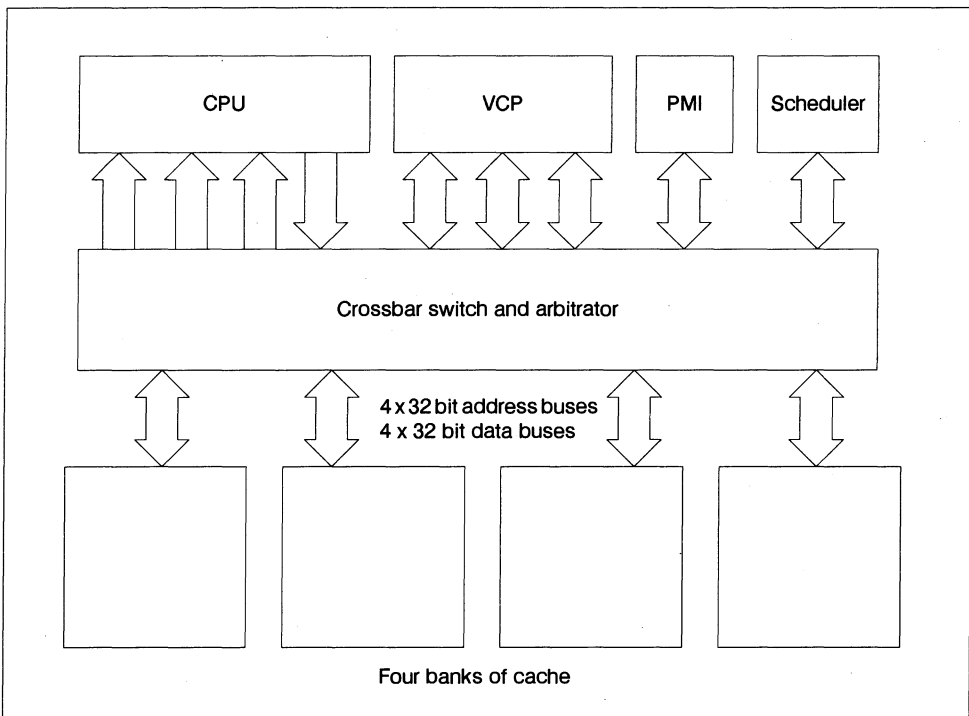
Figure 2.5   Diagram of four banks of cache

The four cache banks are accessed by a number of different functional units in the IMS T9000, some of these units have multiple ports into the cache. To allow four simultaneous reads and writes to take place in each cycle, there are four sets of address and data buses. An arbitrator controls access from the various functional units to the cache banks.

Cache operation

Each of the four banks is addressed by a quarter of the memory space. This division of the address space is done using bits 4 and 5 of the address, the bottom four bits are used to select a byte within a line. Each line consists of: 16 bytes of RAM for the data; 26 bits of associative memory which holds the address of this line

of data; and two control bits to indicate if the line is valid and if it has been modified since it was read in (is 'dirty'). When a memory access is made, the address is checked against the contents of the appropriate bank. If the address is present (and the line is valid) then the access can go ahead, reading or writing the data in a single cycle.

A cache refill engine ensures that there is always one empty line available. Then, if a requested address is not in the cache (a 'cache miss'), the four words containing the data are read from memory into the empty line. The refill engine then has to ensure that a new empty line is created. It does this by choosing a line at random and, if the data has been modified since it was read into the cache, writing it out to memory. The line is then marked as invalid, i.e. empty and available for use. This is known as 'early write-back' as it writes the chosen line out to memory before a cache miss occurs.

The reading and writing of cache lines takes advantage of any fast memory access methods that are available (e.g. 64 bit wide accesses or page mode DRAM).

Use as on-chip RAM

At reset, the cache behaves as 16 Kbytes of normal RAM, enabling the IMS T9000 to be used with no external memory. There may be many applications where a number of transputers are used, each requiring little or no external memory – used in this way the IMS T9000 provides extremely high performance (single cycle memory reads and writes) combined with extremely low cost (possibly no external components except a clock). Starting up in this mode provides compatibility with earlier transputers which have a fixed amount of on-chip RAM. It also makes it possible to test the hardware of a new transputer system as it is known that there is 16 Kbytes of working RAM which can be used by test software.

During the initialization of the IMS T9000 the cache may be programmed to behave as 16 Kbytes of cache, as 16 Kbytes of RAM, or as half cache and half RAM. This can be very useful when certain data or code, e.g. an interrupt handler, must be accessed quickly and in a more deterministic way than a cache provides. The remaining 8 Kbytes of cache will be large enough to achieve high performance.

2.5.2   Workspace cache

The workspace cache can hold a copy of the first 32 words of procedure stack and workspace. It is triple ported, allowing two reads and a write in every cycle. The workspace cache allows local data to be accessed without going outside the CPU, effectively giving zero cycle access and reducing the load on the main cache and external memory. It also means that the pipeline can do four data reads (as well as an instruction fetch) in each cycle: 2 from the local cache and 2 from the main cache.

Because local variables can be accessed quickly, they can be read in the first stage of the pipeline and can then be used for non-local address calculations in the next stage. The workspace cache is write-through; whenever data is written into the local cache it is also written to the main cache. This means there is no overhead for flushing the cache on interrupt or context switch.

The workspace cache is part of the processor pipeline and, in many ways, it is equivalent to the general purpose register set found on other microprocessors, providing fast access to frequently used data. To make use of this architecture, the INMOS ANSI C compiler recognizes the `register` keyword and places those variables lower in the function's workspace so they are more likely to be cached.

Cache operation

The cache is organized as a 32 word circular buffer and is addressed using the bottom five bits of the workspace pointer. As the workspace pointer moves up and down, it rolls around the cache. When the workspace pointer is moved down, on a procedure call for instance, the locations that 'roll into' the cache are marked as invalid and become valid as they are read or written. The first time a variable is read, it is copied from the main cache (and, of course, fetched from main memory if it is not in the main cache). Lines are marked as invalid when they 'roll out' of the cache as the workspace pointer is moved up (e.g. on a return from a procedure call). On a context switch or interrupt, the entire contents of the cache are marked as invalid.

This is illustrated in figure 2.6, where the state of the workspace cache during a procedure call and return sequence is shown. Before the call, the locations in the workspace cache above the workspace pointer

which have been read or written by the program contain valid data. After the call, the workspace pointer moves down – initially the locations which are above the workspace pointer are invalid; as they are accessed by the program they are filled with data and marked as valid. When the procedure returns, the locations which it used will be marked as invalid. As long as the workspace of the called procedure is less than 32 words, some of the workspace of the calling procedure will still be valid after the return. Nested procedure calls, or calls of procedures with a large workspace requirement will cause the workspace pointer to wrap around so that some of the data at the top of the program workspace is no longer in the cache.
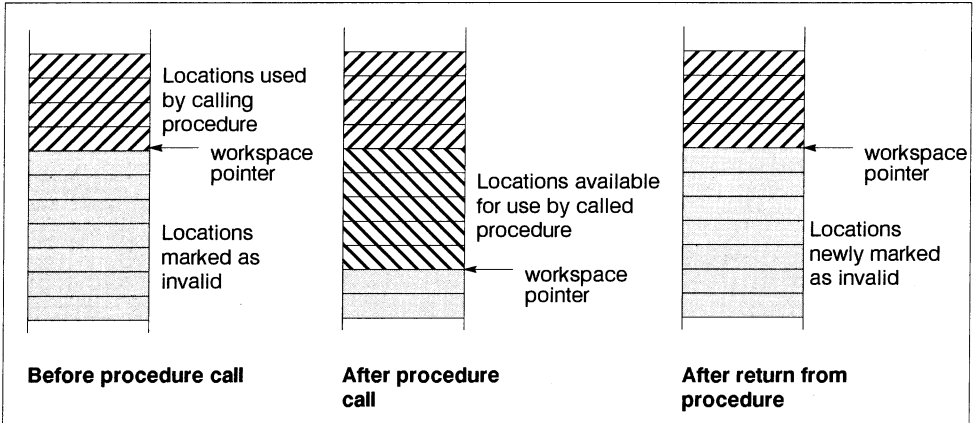
Figure 2.6   Effects of call and return on workspace cache

As the cache is a circular buffer, moving the workspace pointer by 32 or more will cause the pointer into the cache to wrap right round, marking every line as invalid.

3   Simplicity of system design

Many features of the IMS T9000, as with the original transputer range, exist to simplify the user's design
task and to reduce the amount of support hardware and software that is required. This means that designers
can spend more time working on their application and less time worrying about details. Using transputers
can result in smaller, simpler designs, easier system debugging, faster time to market and lower system
cost. Some of these features and their benefits are outlined below.

3.1   Single 5MHz clock input

All transputers, no matter what the processor speed, and all support devices require only a single 5MHz
clock input; on-chip phase locked loops generate all the high frequency internal clocks required for the
processor and links. Because of the asynchronous nature of the link hardware differences in the clock
phase between devices is not important. This means that each processor can have a local clock.

This simplifies system clock generation and distribution, especially where multiple transputers are used.
The use of low frequency signals around a system can be particularly important in electrically noisy environ-
ments such as industrial control systems.

3.2   Programmable memory interface

The first generation of 32 bit transputers have a memory interface which can be programmed to generate
all the timing signals required by a memory system, meaning that little external logic is required to build
a complete system.

The IMS T9000 takes this idea further by providing greater functionality and flexibility. The IMS T9000 pro-
grammable memory interface (PMI) provides complete support for DRAM including multiplexing of row/co-
lumn addresses, refresh, and page mode accesses. It is possible to connect up to 8 Mbytes of 1M x 4
DRAM with no external logic. The amount of memory which can be connected directly is limited only by
capacitive loading; larger amounts of memory will require only the addition of buffering on the address and
data lines.

The IMS T9000 memory interface will automatically exploit any fast access modes for the memory system.
For example if 64 bit wide DRAM is used then an entire cache line can be read in two memory operations.
If page mode DRAM is available, then reads or writes with the same row address will be done using page
mode, greatly reducing the cycle time. This will always be used for cache line reads and writes, where four
consecutive words will be transferred, but it will also work for any set of reads and writes from the same
page.

In addition to supporting fast DRAM, the IMS T9000 will also efficiently interface to other devices, such as
SRAM, ROM or memory mapped peripherals. The PMI on the IMS T9000 divides the address space into
four banks[1]. Each bank provides separate decoding and timing control, generating all the signals needed
for the device types in that bank. The address range, timing, memory type and bus width can be pro-
grammed independently for each bank. There is an additional preset bank for slow, byte-wide ROM. This
is intended for systems where the processor is booted from ROM. Only memory reads can be done from
this bank.

The parameters for the memory interface are programmed into a number of configuration registers. A soft-
ware tool is provided in the transputer development system to simplify the task of designing with the PMI.
This tool can be used interactively to describe the parameters for each memory bank. It then produces an
output file which can be used by other parts of the development system for initializing and loading
transputers. The program also produces timing diagrams and descriptions which can be used in document-
ing the system design.

3.3   Control links and configuration

The IMS T9000 has a pair of control links. One is used for receiving commands and sending status informa-
tion, the other provides a cascade connection so that all devices in a system can be daisy-chained togeth-

1. There is no connection between the four banks in the memory interface and the four banks in the main cache.

er. The control links use the same link protocol as the IMS T9000 data links and provide a control network which is completely independent of the normal data communication network.

The control links have through routing hardware so that the controlling processor (possibly an IMS T9000) appears to have a direct connection to every device in the system.

The control links are kept totally separate from the links used for program communication in a system. A program running on the IMS T9000 cannot send messages down the control links. The separation of control and data links ensures that the control links are completely reliable. For extra reliability, they can be run at a lower bit rate.
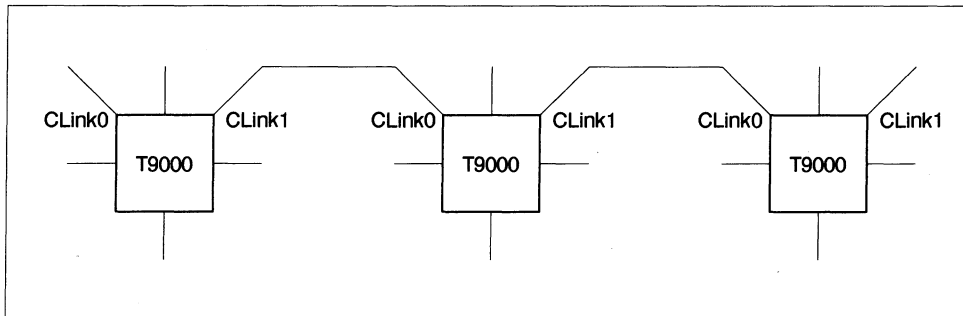
Figure 3.1   Network of control links

The control links provide an independent communication network which can be used to load code, do hardware debugging, monitor a running system for errors and perform diagnostic functions, both for a single IMS T9000 and a network.

Because of the great flexibility of the memory interface and the communications system of the IMS T9000 there are a number of configuration registers that need to be programmed. For all of these, the development tools will program the registers using high level descriptions of the system. For example, as noted above, there is an interactive tool for developing configuration data for the PMI. Similarly, the communication system is set up using high level language descriptions of the software and hardware networks.

There are two ways of programming the configuration registers: by writing to them from a program running on the IMS T9000 itself; or via a control link from the host system. The first method is used when the system is booted from ROM, for example in an embedded system. The second method can be used in a development environment or, in a multi-transputer system, where only one processor is initialized (or 'configured') from ROM and all the others are configured via their control links from that root processor. In both cases the IMS T9000 development tools will generate the data to be programmed into ROM or sent to the control link of a processor.

There are a number of stages of initializing and loading code onto the IMS T9000 after it has been reset. These are known as 'reset levels' and during the initialization process, every IMS T9000 must go through each level from complete reset, to having application code running. Each of these levels can be done from ROM or through a control link.

3.4   Loading and bootstrapping

The transputer can also be bootstrapped in two ways: from code received down a link or from ROM. All INMOS development tools generate programs to be loaded by either method as required during development or in a production system.

There are a number of advantages to the ability to load code from a link. It greatly simplifies the development cycle – there is no need to keep programming new EPROMs with new versions of code (or use an EPROM emulator); it can simply be loaded down a link. It simplifies testing of hardware – a transputer provided with the minimal essential external signals (5 volts, clock, etc) will be guaranteed to work; there is

then16 Kbytes of on-chip RAM in which to load test code. In a multiprocessor system, only the root proces-
sor needs to be booted from ROM – the others can be booted down a link with code contained in that sys-
tem ROM. It is even possible to switch between ROM and link booting, in order to do field testing and diag-
nose faults in an installed system.

3.5   Examples

To show how simple it is to build systems using the IMS T9000, a few example block diagrams are given
here. In the simplest cases these are almost complete circuit diagrams.

The first example (figure 3.2) is a complete working system using the IMS T9000s internal RAM as the sys-
tem memory. The processor boots from ROM which contains the application software. This processor can
communicate with other transputers or peripherals through its data links. It can be set to boot from ROM
or from link for development and test purposes. The full16 Kbytes of on-chip RAM is available for program
workspace.
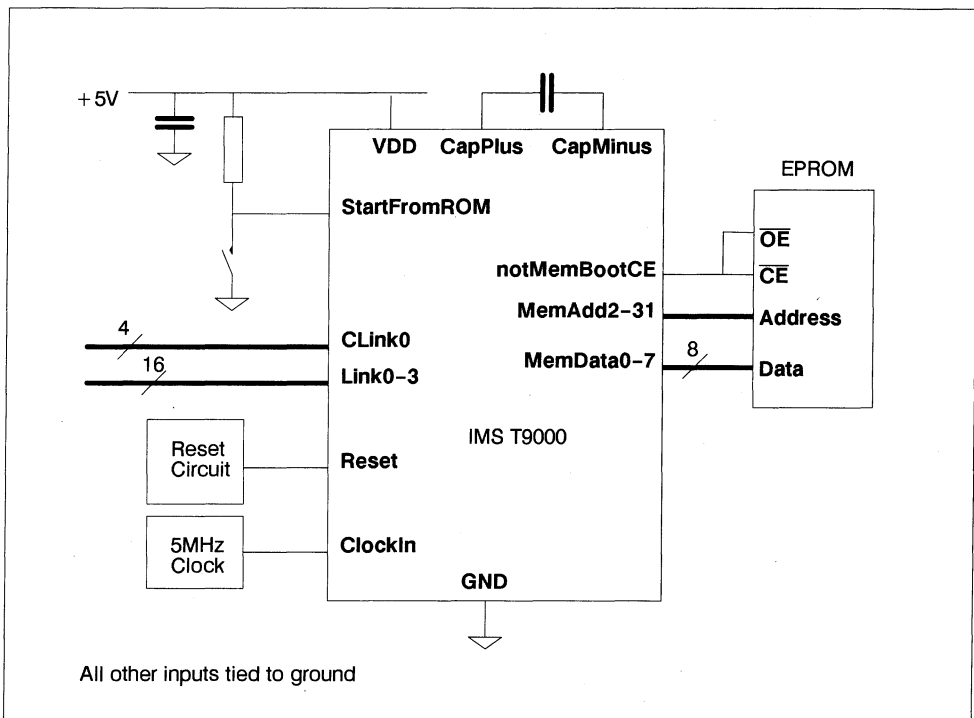
Figure 3.2   Complete IMS T9000 system with EPROM

Figure 3.3 shows how a low cost system can be built using a small amount of SRAM. This could be com-
bined with ROM and peripherals for a low cost embedded application.
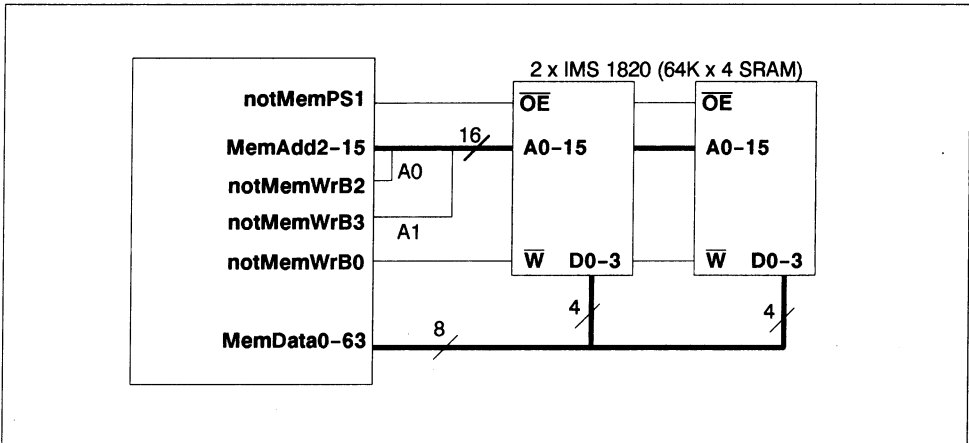
Figure 3.3   Low cost system with 64 Kbyte of SRAM

The third example in figure 3.4 shows how a large amount of DRAM can be connected to the IMS T9000 with no external logic for decoding, control signal generation or buffering.
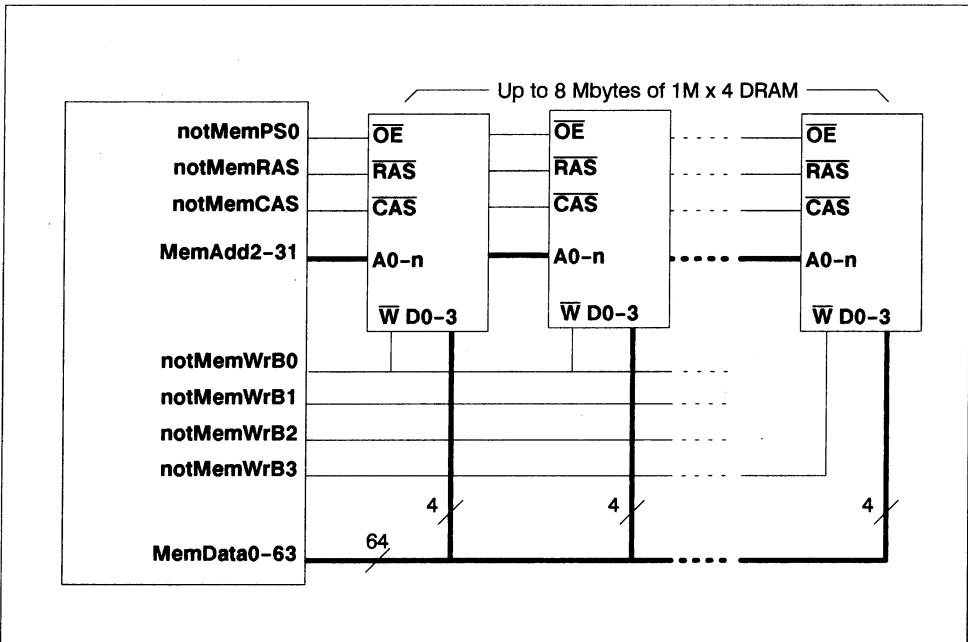
Figure 3.4   High performance system with 8 Mbytes of DRAM.

The memory in each bank is enabled by separate strobe signals so all of the above memory types could be combined on a single IMS T9000.

4   Protection and error handling

The IMS T9000 extends the error handling of earlier transputers to allow error conditions to be easily trapped and handled in software. It can run code in a protected mode where all memory accesses are checked and certain, privileged, instructions cannot be executed.

4.1   Error handling

The first transputers have only a global mechanism for trapping errors; stopping the entire processor when an error is detected. The IMS T9000 extends this to allow a trap handler to be associated with a process to provide more localized error handling. When an error occurs, control is transferred to the trap handler with information about the nature of the error and where it occurred.

The action of a trap handler will, in general, be dependent on the language or operating system being used and will be invisible to the applications programmer. Some languages may include support for user written error handlers. After taking the appropriate action, for example to report or correct the error, the trap handler can return control to the process which caused the error which can then continue execution. Each process can have its own trap handler, or one trap handler can be shared by several (or all) of the processes on the transputer.

To maintain compatibility with earlier transputers, the IMS T9000 can also run processes in a global error mode where the behavior on error is identical to the IMS T805. These two types of processes are known as L-processes and G-processes (for Local and Global error handling) respectively. Both L- and G-processes can be run in parallel, the processor dynamically switching modes as it switches between processes. This allows code compiled for the IMS T805 (which will always be in global error mode) to be run in parallel with code specifically compiled for the IMS T9000.

4.2   Protected mode

The IMS T9000 can also run code in protected mode. This is designed to allow run-time checking of programs written in 'unsafe' languages such as C and also to provide memory management. For example, C allows pointers to data or functions. Without checks for valid pointers, these could contain an illegal memory address such as: another process's data or code; a non word aligned address; or a function pointer which does not point to valid code. As no checks are defined in the language it is important to be able to check such accesses at run time, if needed.

The protection mechanism is intended to support software development and debugging, and programming secure systems. It protects the user's processes or tasks from each other and also protects an operating system kernel, or other run time support, from user code. Although code run in this mode is frequently referred to as a 'protected process', it is not the process which is protected but the rest of the world that is protected from errors in the process.

Protected mode processes

Any L-process can run a piece of code as a protected mode process (or P-process); the processor saves the state of the L-process and starts executing the P-process. The P-process is executed until control is returned to the L-process because of an error, protection violation or some other reason. It is important to realize that P-processes are not scheduled by the transputer's own scheduler – they only run under the control of a supervisor L-process. Any of the instructions or other events that might cause a P-process to be descheduled, will cause control to be returned to the supervisor. The relationship between a P-process and its supervisor is analogous to that between an L-process and its trap-handler. In both cases the processor can be thought of as swapping between the two pieces of code.

Executing illegal instructions

Because control is returned to the supervisor when a P-process attempts to execute a privileged or illegal instruction, it is possible to provide communication and other facilities to a P-process in a controlled way, but one which is invisible to the programmer. For example, the input and output instructions are privileged

so, if a P-process attempts a communication then it will trap to the supervisor L-process. This L-process can examine the state of the P-process and, if the attempted communication is 'legal', perform the communication and return control to the P-process. The P-process will continue as if a normal communication had occurred.

There is also a 'syscall' instruction which can be used by a P-process to explicitly request some action by the supervisor.

Memory management

When running in protected mode, all memory accesses are checked and translated. Each P-process can access four regions of memory. The size and base address of each region can be set, and each can have different protections. Each area can be given permission for code to be executed from it and for data to be written. For example an area of memory containing code would normally be marked with execute permission but write protected.

All addresses generated when the processor is running in protected mode are logical addresses. These are translated to physical addresses by combining the low order bits of the logical address with the high order bits from the control register for that region. The translation and checking is done in parallel with other address generation operations and so imposes no overhead on memory access time.

The IMS T9000's memory management can be used to implement swapping of memory to and from disk and relocation (although it does not support page-based virtual memory). This can be used to implement most operating system kernels. It can also be used for 'stack extension'. All the instructions which move the workspace pointer are checked for a valid address after the operation. If it is found that the workspace address is no longer valid then a trap occurs, the supervisor process can then allocate more memory for the processes stack and restart it.

5   Support for multiprocessing

The requirement for processing performance in embedded systems is continuously increasing as control algorithms become more sophisticated and as systems become more complex. In the long term, the only solution to these ever increasing demands for performance is the use of multiple processors to perform parallel processing.

Transputers are the only microprocessors specifically designed to tackle the problems of building multiprocessor systems.There are advantages other than just performance to using multiple transputers in a system; it allows scaleable systems to be built, where more processors can be added as demand increases, or to provide the optimum balance of price versus performance. The communications facilities of the transputer family can also be used to build distributed systems where, for example, the processors are located near the equipment or components they control and use links to communicate with other processors in the system. In addition, transputers can be used to build high reliability, fault tolerant systems.

Fast interrupt response and process switch

In most embedded applications, there is a need for fast real time response (both to external interrupts and for context switching in multitasking systems). The design of the IMS T9000 processor exploits the presence of the two on-chip caches by having only a small number of registers in the CPU. This means that there is little state to be saved when an interrupt or task switch occurs, so these operations are extremely fast. These types of operations are very efficient on the transputer because of the hardware scheduler.

The register stacks are duplicated so that, when a process running on the IMS T9000 is interrupted, the contents of the stacks do not need to be written to memory. This results in a sub-microsecond interrupt response. Furthermore, the duplication of the register stacks enables floating-point arithmetic to be used in an interrupt routine without any performance penalty.

5.1   The transputer model of concurrency

The model of concurrency and communication implemented by the transputer hardware is based on the ideas of communicating sequential processes. All the features for creating processes and communicating between them are accessible from any high level language for the transputer and are implemented directly by the occam programming language [2].

Processes and channels

Each process can be regarded as a black box with internal state, which can communicate with other processes using communication channels. Each channel is a point to point connection between two processes. One process always inputs from the channel and the other always outputs to it. Communication is synchronized: the first process ready to communicate waits until the second is also ready, then the data is copied from the outputting process to the inputting process and both processes continue.

Each process starts, performs a number of actions and then terminates. An action may be a set of sequential processes performed one after another, as in a conventional programming language, or a set of parallel processes to be performed at the same time as one another. Since a process is itself composed of processes, some of which may be executed in parallel, a process may contain any amount of internal concurrency, and this may change with time as processes start and terminate. Ultimately, all processes are constructed from three primitive processes: assignment; input and output.

Program structure

Figure 5.1 shows an example of a system constructed from three communicating processes. In this case there are separate processes to handle the external hardware (the screen and keyboard) and to execute the main, application, process. This is a modular design – only the hardware handling processes have to be changed if the software is moved to a new environment, the same interface (the data sent and received on channels or 'protocol') can be presented to the application process. The keyboard handler can be interrupt driven, only being scheduled when a character is typed, the interrupts appearing as communications. The input and output processes can provide buffering and other filtering of the data, all of which is invisible

to the main application process, which could even be placed on a separate processor. This use of separate processors need not just be for performance reasons but might be done, for instance, if there are a large number of peripheral devices which could be better handled by a low cost 16 bit transputer. One or more high performance transputers could then be used for the main computing processes.
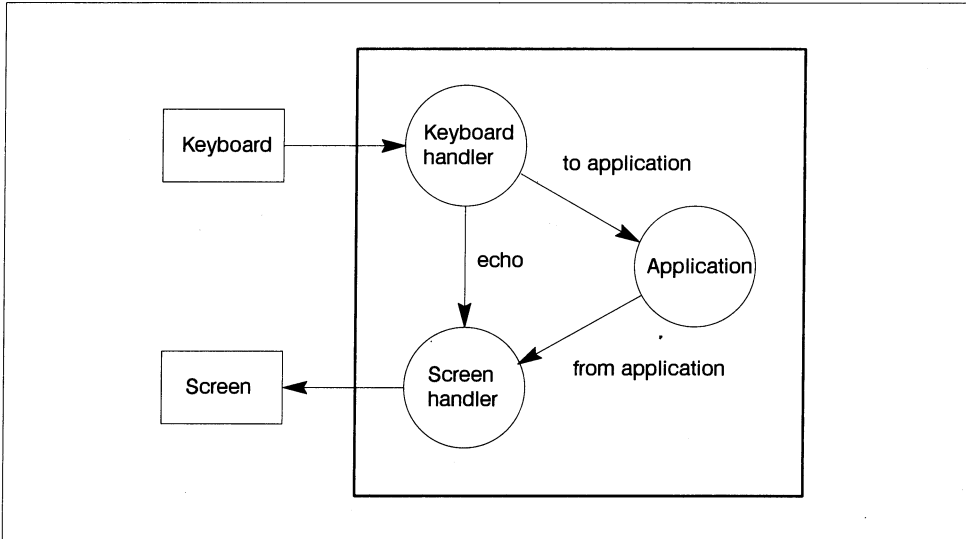
Figure 5.1   Processes and channels

Example

The code for creating parallel processes in C is very simple. For example, if the three processes in the example above are external functions, then the following code is all that is needed to run them in parallel:

```c
#include <stdlib.h>
#include <channel.h>
#include <process.h>

/*
    declare externally defined functions
*/
extern keyboard_handler (Process *p, Channel *to_app, Channel *echo);
extern screen_handler (Process *p, Channel *echo, Channel *from_app);
extern application (Process *p, Channel *to_app, Channel *from_app);

/*
    declare pointers to process and channel data structures
*/
Process *kbd_p, *scrn_p, *appn_p;
Channel *to_app, *from_app, *echo;

/*
    allocate and initialize channel data structures
*/
to_app   = ChanAlloc();
from_app = ChanAlloc();
echo     = ChanAlloc();

/*
```

```
        allocate and initialize the process data structures
*/
kbd_p  = ProcAlloc (keyboard_handler, 0, 2, to_app, echo);
scrn_p = ProcAlloc (screen_handler, 0, 2, echo, from_app);
appn_p = ProcAlloc (application, 0, 2, to_app, from_app);

/*
    now run the three processes in parallel, this call
    will return when all three processes have terminated
*/
ProcPar (kbd_p, scrn_p, appn_p, NULL);
```

A more complete explanation of how parallel programs can be written for the transputer can be found in INMOS Technical Note 68, *"Developing parallel C programs for transputers"* [5].

The equivalent program in occam would be:

```
CHAN OF BYTE to.app, from.app, echo :
PAR
  keyboard.handler (to.app, echo)
  screen.handler (echo, from.app)
  application (to.app, from.app)
```

Multiprocessor programs
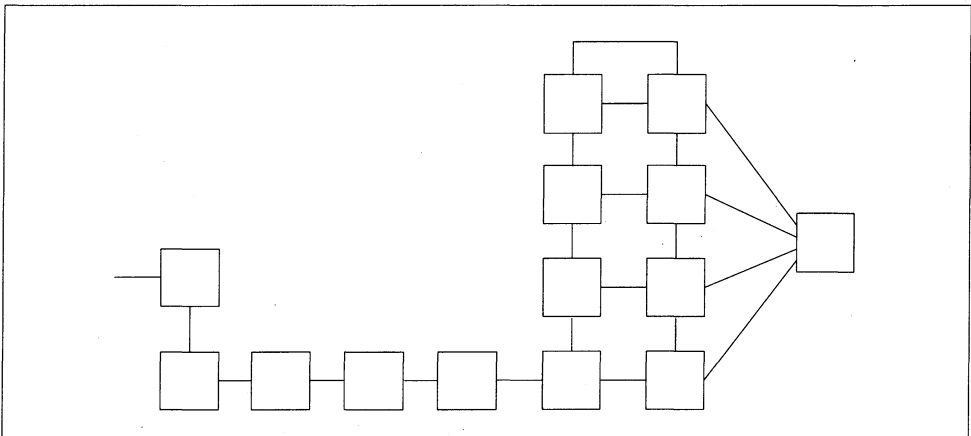
Figure 5.2   Transputers and links

Every transputer implements these concepts of concurrency and communication. As a result, the same model can be used to program an individual transputer or to program a network of transputers. Figure 5.2 shows a typical network of transputers connected by serial links. When a number of processes run on an individual transputer, the processor shares its time between the concurrent processes, and channel communication is implemented by moving data within memory. When this programming model is used to program a network of transputers, each transputer executes the process, or processes, allocated to it.

Communication between processes on different transputers is implemented directly by transputer links. Thus the same program can be implemented on a variety of transputer configurations, with one configura-

tion optimized for cost, another for performance, or another for an appropriate balance of cost and perform-ance as illustrated in figure 5.3.
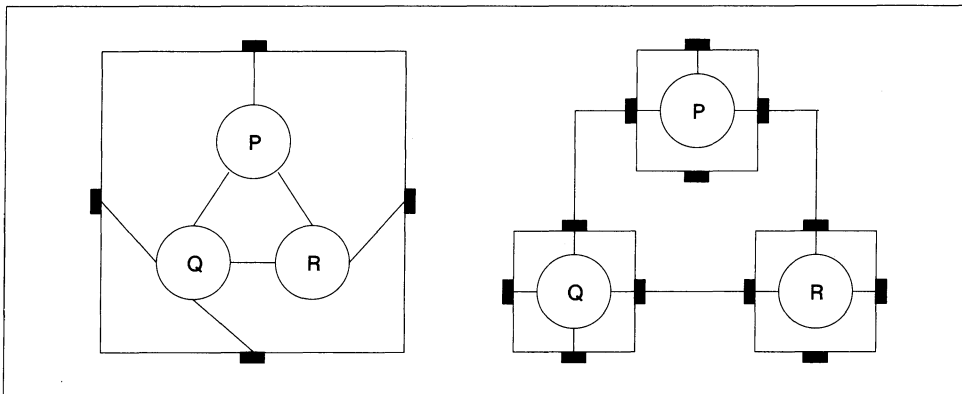
Figure 5.3   Mapping processes onto one or several transputers

5.2   Other models of concurrency

Although the transputer has direct support for concurrent process which communicate via channels, it is possible to use the same features of the transputer to build other types of multiprocessor system or to sup-port different scheduling models. The IMS T9000 includes a number of instructions for manipulating the transputer process queues; these make it simple to write real-time kernels, exploiting the efficient task switching of the transputer architecture. There are also instructions for ensuring that the data in the cache and in memory are consistent. These can be very useful when implementing a shared memory system.

Shared memory

In a shared memory system, a number of processors have some sort of common area of memory which they can all access. This has some advantages over the channel communication model, especially where very large amounts of data need to be shared or moved between processors. The transputer has hardware and software support for shared memory systems.

The PMI has a set of signals for controlling access to the external memory interface by an external device. This is primarily intended for use with a DMA based co-processor. It can also be used, with external arbitra-tion logic, to allow all of the processors in a system to access the shared memory.

Alternatively, there may be a number of blocks of memory that can be switched into the memory map of different processors under software control. These blocks can be used for exchanging data and passing messages between processors. To synchronize the switching of these blocks of memory between proces-sors, the ideal method is to pass messages over the transputer links; as the memory is switched to a pro-cessor's address space, it is sent a message from the previous user of the memory to inform it that it is now the new 'owner' of the memory. This allows large amounts of data to be moved from one processor to another but without the overhead of copying all of it over a link.

In any shared memory system, the use of a cache can be a problem. In the IMS T9000 there are instructions for forcing changed data in the cache to be written out to main memory and for marking data in the cache as invalid so that it will be read from main memory. As the exchange of data is synchronized between pro-cessors, these instructions can be used to make sure that the correct data is in both the main memory and the cache of the processors involved.

It is also possible to mark banks of external memory to be 'un-cacheable'; data from that area of memory will never be put in the cache. This ensures that a number of processors or other devices which make ran-dom reads and writes of that memory will always get the most up to date data. In this case there must still

be some synchronization of the memory accesses to make sure that information is not read by a processor until it has been written; again, this synchronization can be done over the transputer links.

5.3   Hardware scheduler

The IMS T9000 processor includes a hardware scheduler which implements the transputer model of concurrency. In many applications this will remove the need for a software kernel. However, the transputers own scheduling mechanisms can be accessed from software to provide efficient support for the implementation of standard real-time kernels.

At any time, a transputer process may be:

active	–	being executed
	–	on a list waiting to be executed
inactive	–	ready to input
	–	ready to output
	–	waiting until a specified time

The scheduler operates in such a way that inactive processes do not consume any processor time. The active processes waiting to be executed are held on a list of process workspaces. This is implemented using two registers, one of which points to the first process on the list, the other to the last. In figure 5.4, **P** is executing, and **Q**, **R** and **S** are active, awaiting execution.

A process runs until it is unable to proceed because it is waiting to 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 switch times are very small as little state needs to be saved; it is not necessary for the processor to save the evaluation stack on descheduling.
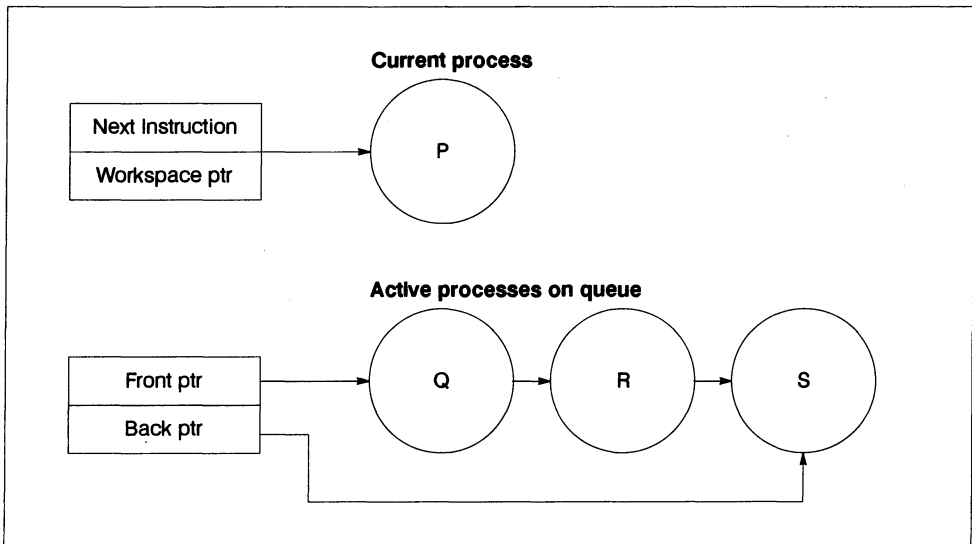
Figure 5.4   Transputer process queue

5.4   Interrupts, events and timers

As well as process scheduling and communications, the scheduling hardware also supports simple handling of interrupts and timers. Any event that a process might need to wait for (whether it be a communication,

an interrupt or a timeout) can be treated in the same way as a communication. For example, an interrupt handler simply has to wait for an input from a special channel which is mapped onto an interrupt ('Event') input. Because inputs are synchronized, that process will not proceed until the 'input' becomes ready, i.e. until there is an interrupt.

This makes interrupts on the transputer very easy to use. An interrupt handler is simply a process like any other waiting on an input from the interrupt 'channel'. This contrasts greatly with the traditional idea of an interrupt handler as something difficult which needs to use special instructions and be written in a very different way from other program code (usually in assembler).

The IMS T9000 has four sets of pins, known as 'Event' channels, which can be used for control and synchronization purposes. Each Event channel can be configured either as an input or an output. As inputs they can be used as interrupts, to cause a fast processor response to a external signals. When an Event channel is configured as an output, the process outputting to it will be descheduled until the external device provides the necessary handshake signal.

The transputer has two timers; one of which 'ticks' every microsecond, the other ticks every 64 microseconds. The current value of the processor timer can be read, or a process can perform a *timer input* in which case it will become ready to execute when a specified time has been reached. Both these uses of the timer are treated as inputs similar to channel communication. If the timer is simply being read then the current timer value is provided immediately; if the process is waiting for a particular time, then it is descheduled until that time.

5.5 Shared resources

The IMS T9000 also provides efficient hardware support for controlling access to a shared resource. This could be a hardware resource (e.g. a printer) or a piece of software running on a particular processor in a network. Each process which wants to use the resource (a 'client') can make a request to the controlling process (the 'server'). This request is done in the form of a channel communication and can, therefore, be done across a network by using transputer links. If the resource is available then the requesting client is given access to it, otherwise it is put on a queue until the resource becomes free. If multiple clients request a resource then they are all automatically queued until it is available.
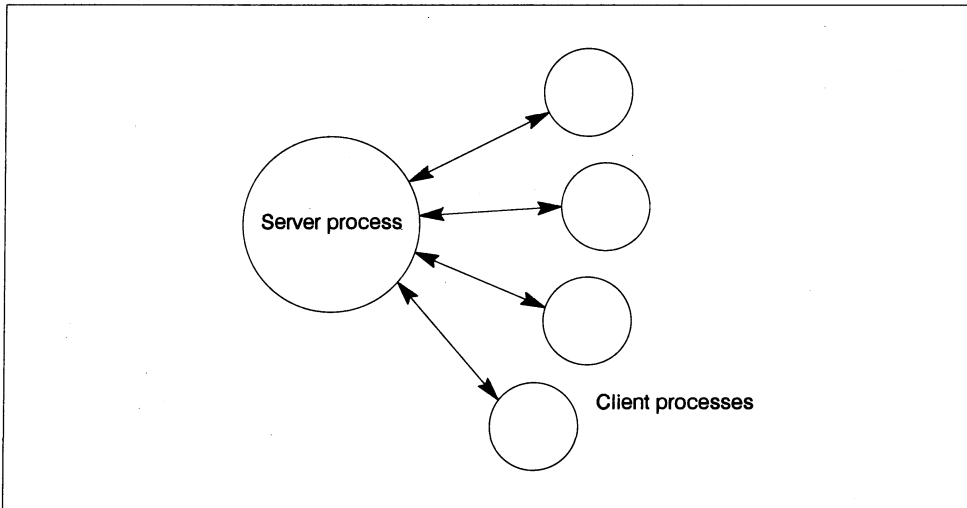
Figure 5.5   Client/server model of resources

The resource mechanism can provide pairs of channels between the server and the processes accessing it. This can be used, for example, to implement remote procedure calls across a transputer system.

6   Communication links

Transputer links provide a simple and regular way of interfacing to peripherals and host systems as well as communicating between transputers. On a single transputer, processes can communicate via channels; the provision of links allows processes on different transputers to communicate in the same way. The IMS C104 routing device enables this communication to take place across a network, even between transputers that are not directly connected.

The same communication model can be used to communicate with peripheral devices or a host system using a link adaptor which converts from the bit-serial protocol of the links to a parallel port.

6.1   Using links between transputers

Transputer links can be used to implement point to point communication between transputers. This allows transputer networks of arbitrary size and topology to be constructed. Point to point links have many advantages over bus based communications in a multiprocessor system:

- There is no contention for the communication mechanism, regardless of the number of processors in the system.

- There is no capacitive load penalty as more processors are added to the system.

- The communications bandwidth does not saturate as more communicating devices are added to the system. Rather, the larger the number of transputers, the greater the total communications bandwidth of the system.

- Because each transputer in a system uses its own local memory, overall memory bandwidth is proportional to the number of transputers in the system. This is in contrast to a large, global memory where the processors must share the available memory bandwidth.

For small systems, the four transputer links on the IMS T9000 can provide complete connection between up to five devices. By using additional message routing devices such as the IMS C104, networks of any size can be built with complete connection between all IMS T9000s. If a system does not need complete connection or the flexibility of routing that the IMS C104 provides, then networks can be built just from directly connected transputers.

6.2   Advantages of using links

The advantages of using links for communication are efficiency, simplicity and hardware independence.

Efficiency

There is a separate DMA controller for every input and every output channel which allows data to be transferred without processor involvement. To exploit this, the transputer deschedules a process which is waiting for a communication to complete, freeing the processor to execute another process. When the communication is complete, the process is rescheduled, providing automatic synchronization with the data transfer.

Simplicity

The communication links are, however, very simple to use. The transputer has simple instructions for performing input and output and these are available to the programmer either as function/procedure calls in a high level language or, in the case of occam, as an integral part of the language. For example, in a C program, to transfer an array of 256 bytes from the array data to a channel c, the following call could be used:

```
ChanOut (c, data, 256);
```

In occam, the same operation could be written as:

```
c ! 256::data
```

This output operation requires four instructions: three to load the address of the channel, the address of the data and the number of bytes, followed by the output instruction itself. It is worthwhile comparing this with the complex code required to do the equivalent transfer on a traditional microprocessor. For example, it would require a DMA controller to be programmed and, in order to allow some degree of multitasking, it would be necessary to set up the interrupt hardware and write an interrupt handler to control the data transfer. All of this is done automatically by the input and output instructions on the transputer.

As a more concrete example, consider the case of a file server running on a host system talking to a program running on the transputer. This would provide the transputer program with all the host operating system facilities such as filing system, terminal i/o etc. At the transputer end, the communication is very simple: a single line of code, as outlined above. At the host end, a lot of complex code (probably written in assembler) is required to handle the data transfer, either programming a DMA controller or polling the status registers of the memory mapped port. In the case of a Unix system, it will also be necessary to write a device driver to interface to the hardware.

Of course, when the communication is between two transputers, then both ends of the communication are equally simple.

Hardware independence

As well as being fast and easy to use, channel communications provide a degree of hardware independence.

The same communication mechanism can be used to communicate between concurrent processes, with peripherals or a host system, and even to handle interrupts. This simplifies the development and testing of code as each process can be functionally tested before being used in the complete system. A good description of program development for transputers can be found in [4].

Furthermore, exactly the same code can be used to communicate between processes on the same transputer (using so called 'soft channels') and to communicate between transputers (using links, or 'hard channels'). Not only is the source code the same, but the same transputer instructions are used – the transputer determines at run time whether it is using a hard or a soft channel. This saves the programmer from having to make decisions about the final hardware implementation while developing and testing code. The IMS T9000 takes this separation of software from hardware one step further than previous transputers.

6.3   IMS T9000 links

On previous transputers the programmer was limited to assigning two channels, one in each direction, to each link. To map a particular piece of software onto a given hardware configuration the programmer has to map processes to processors within the constraints of available connectivity. The problem is illustrated in figure 6.1 where 3 channels are required between two processors, but only a single link connection is available.

One possible solution, and one that is frequently suggested by transputer users, is the addition of more links. However this does not really solve the problems; there is still limited connectivity available. The number of extra links that can be added is limited by VLSI technology. This 'solution' does not address the more general communication problems in networks, such as communication between non-adjacent processors, or combining networks in a simple and regular way.
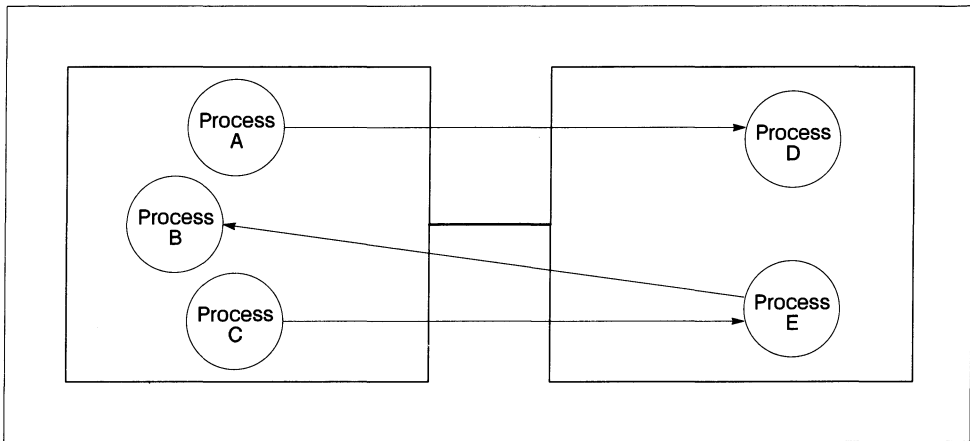
Figure 6.1   Multiple communication channels required between processors

6.3.1   Virtual channels

The solution chosen in the IMS T9000 was to add multiplexing hardware to allow any number of processes to use each link, so physical links can be shared transparently. These channels which share a link are known as 'virtual channels'; they have the same behavior as software channels.

The IMS T9000 has four data communication links, each with a DMA controller and the ability to synchronize with the scheduling of processes. The links and DMA engines are controlled by a separate communications processor, the virtual channel processor (VCP), which works concurrently with the CPU. This supports practically a large number of virtual channels on each link.
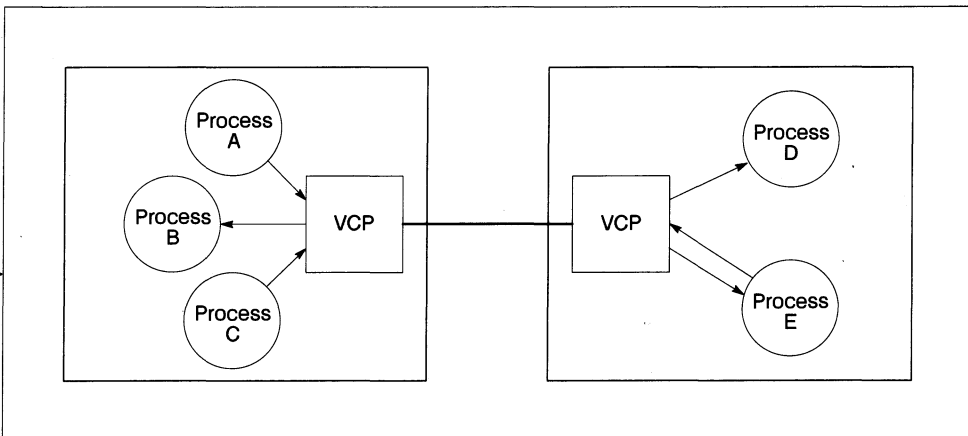
Figure 6.2   Shared links between IMS T9000s

Virtual links

Each message sent across a link is divided into packets. Every packet requires a header to identify its destination process. Packets from different messages are interleaved on the link. There are a number of advantages to this:

- It makes the transputer simpler to use as it separates the software configuration from the hardware. The programmer does not need to limit the number of channels between processors or explicitly allocate channels to links.

- Channels are, generally, not busy all the time therefore the VCP can make better use of hardware resource by keeping the links busy with messages from different channels.

- Messages from different channels can effectively be sent concurrently – the processor does not have to wait for a long message to complete before sending another.

Virtual channels are always created in pairs to form a 'virtual link'; this means there is no need for a return address in packets, the acknowledgements are simply sent back along the other channel of the virtual link.

Sending packets

The IMS T9000 sends the first packet of a message and then waits for an acknowledgement from the receiving processor before sending the next. The process which sent the message cannot proceed until the last packet of the message has been acknowledged. Messages and acknowledgements from other virtual links can be sent while waiting for an acknowledgement on a virtual link. This ensures that a single virtual link cannot monopolize a physical link.
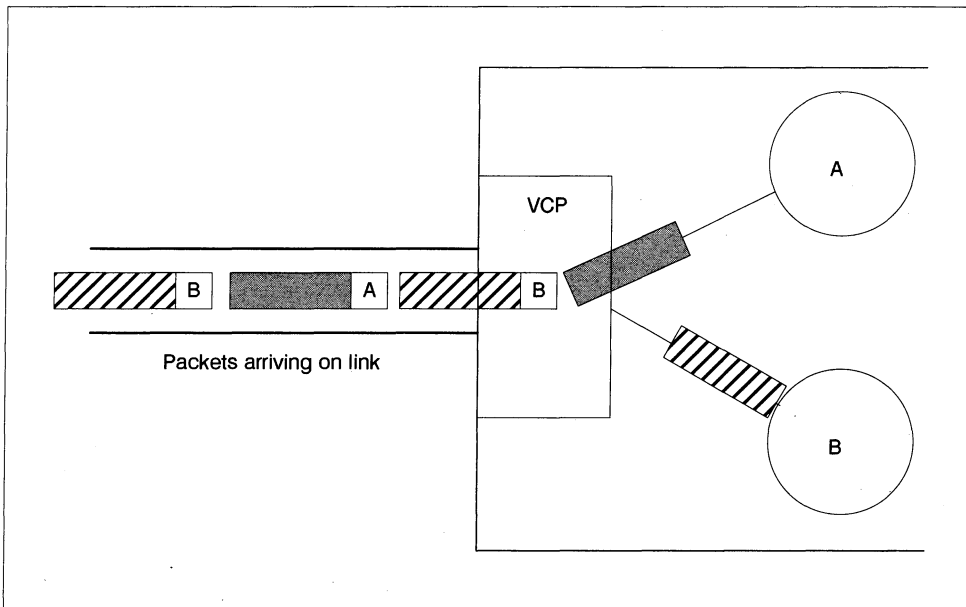
Figure 6.3 Multiple channels sharing a link

Receiving packets

The initial packet of a message is acknowledged if a process has requested a message on that virtual link. The acknowledgement can be sent as soon as the inputting process is identified, as long as the inputter is able to accept another packet. This means that the entire packet does not have to be received before the acknowledgement is sent. In this way the acknowledgement can be received by the transmitter before all of the data packet has been sent and the transmitter can send the next message packet immediately.

The IMS T9000 provides one packet buffer for each virtual link so that each input can be ready to accept an unsolicited packet. This means that other virtual channels sharing a physical link are not delayed if one virtual channel is not ready to input. This buffering of the first packet only takes place if the receiving process is not ready to input, otherwise the data is written directly to the inputting process's workspace. This buffer is not visible to the programmer; all communications are still synchronized at the message level.

The virtual channel processor

The VCP routes messages to and from processes on IMS T9000s. It shares each physical link between any number of processes. It also supports non-local communications by using the IMS C104 to route mes-

sages in a network of transputers. This can provide multiple virtual channels between any two transputers in a network. Requests to send messages are queued by the VCP so that the main CPU is not delayed waiting for packets to be sent.

Implementation

To achieve the speed required to match a faster processor, and to support the virtual channel protocol, a new, simple link standard has been implemented. The original transputer links are referred to as over-sampled (OS) links and use a pair of wires. The IMS T9000 links have four wires for each link (a data and strobe line in each direction) and are known as DS links. All signals are TTL compatible.

The links are asynchronous; the receiving device synchronizes to the incoming data. This simplifies clock distribution within a system, the exact phase or frequency of the clock on a pair of communicating IMS T9000s is not critical. It also means that devices with different processor speeds can communicate.

6.3.2   Levels of link protocol.

As with any communications system, the links can be be described at a number of levels with a hierarchy of protocols. At the highest level a message consists of the data that the user sends down a channel from one process to another. Any type of data or message can be sent in this way. This communication is synchronized; it will not take place until both processes are ready and the two processes will not continue until the message transfer is complete.
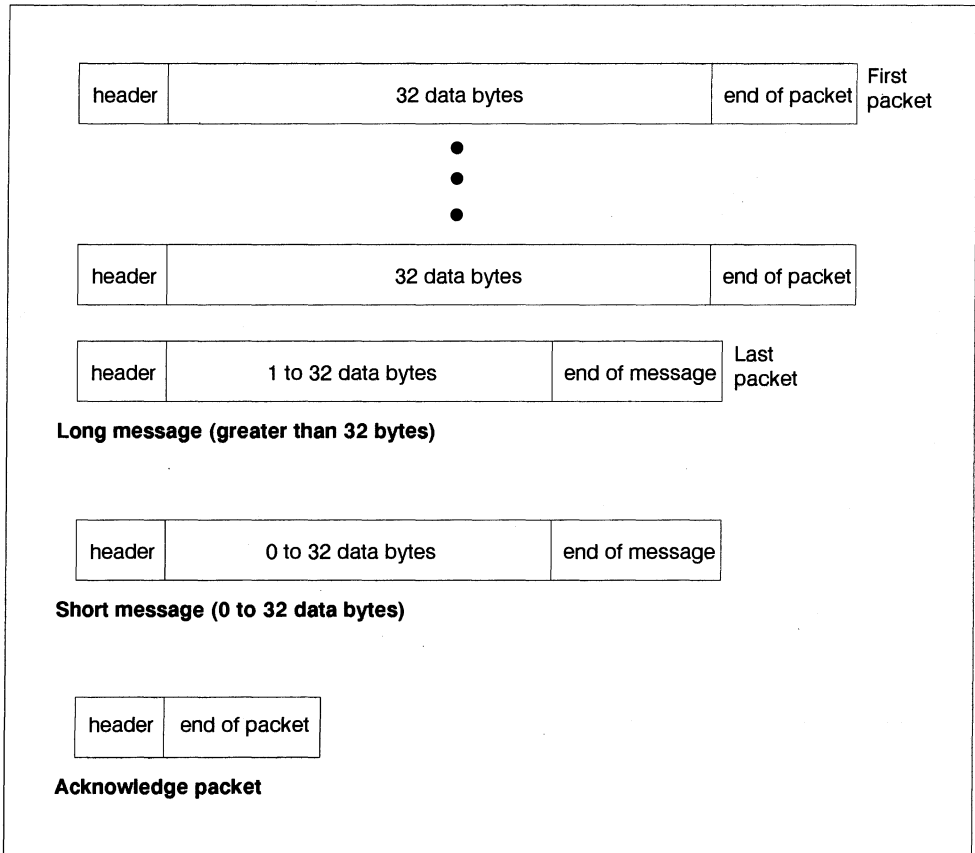
Figure 6.4   High Level protocol

Packet level protocol

In order to transfer a message from one IMS T9000 to another, the virtual channel processor sends it as one or more packets. This allows packets from a number of different channels to be interleaved on the same link. Each packet is acknowledged by the receiving IMS T9000, to maintain synchronized communication and to limit the amount of buffering required.

Every packet has a header defining the destination address followed by the data bytes and, finally, an 'end of packet' or 'end of message' token. See figure 6.4. This simple protocol supports messages of any length; the receiving device knows when each packet and message ends without needing to keep track of the number of bytes received. It also maintains synchronization at the message level.

A packet can contain up to 32 data bytes. If a message is longer than 32 bytes then it is split up into a number of packets all, except the last, terminated by an 'end of packet' token. The last packet of the message, which may contain less than a full 32 bytes, is terminated by an 'end of message' token.

Shorter messages can be sent in a single packet, containing 0 to 32 bytes of data, terminated by the 'end of message' token. With this protocol zero length messages can be sent, allowing efficient synchronization between processors.

Packet acknowledgements are sent as zero length packets terminated with an 'end of packet' token. This type of packet can never occur as part of a message because a zero length data packet must always be the last, and only, packet of a message, and will therefore be terminated by an 'end of message' token.

Token level protocol

In order to support the packet level protocol described above, a lower level protocol is needed for encoding tokens which may contain a data byte or control information. Each token has a parity bit plus a control bit which is used to distinguish between data and control tokens. In addition to the parity and control bits, data tokens contain 8 bits of data and control tokens have two bits to indicate the token type (e.g. 'end of message').
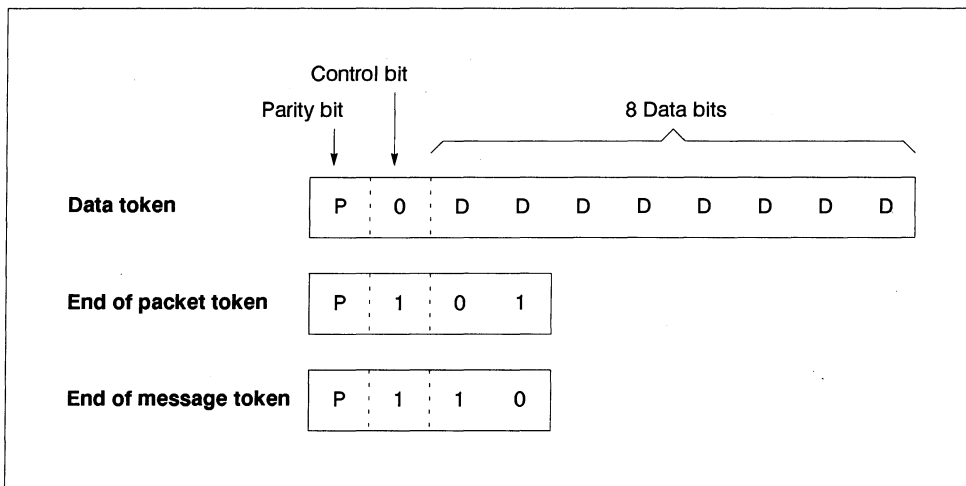
Figure 6.5   Low level protocol

Bit level protocol

At the lowest, hardware, level the signals on the data and strobe lines of a link encode a sequence of bit values. The protocol guarantees that only one of the two wires will have an edge in each bit time. The levels on the data wire give the values of the transmitted bits. The strobe signal changes state whenever the data wire does not. These two signals encode a clock along with the data which makes it easy to asynchronously detect the data at the receiving end.
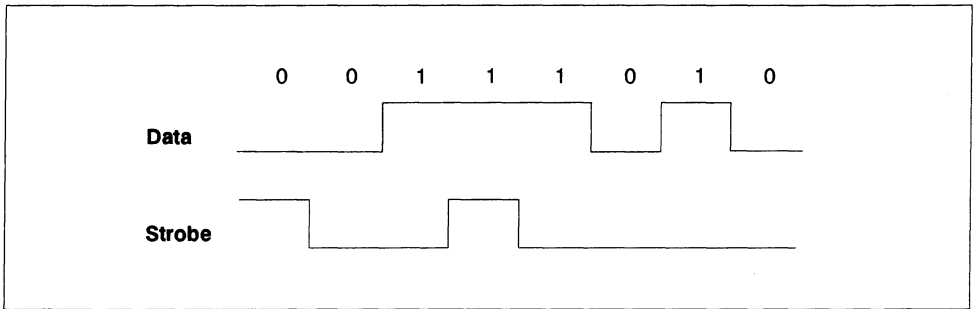
Figure 6.6   Hardware level

The first generation of transputers use a phase locked loop to synthesize a high frequency clock signal which is then used to sample the link data. This is adequate for the data rates involved, but would not easily support the bit rates of 100 Mbits/s and greater used by the IMS T9000.

7 Network communications

The use of INMOS links for directly connecting transputers has already been described. The new link protocol not only simplifies the use of links between processors but also provides hardware support for routing messages across a network.

7.1 Message routing

The VCP (virtual channel processor) on the sending IMS T9000 packetizes messages to be sent over a link and adds a header to each packet to identify the destination process. At the receiving end, the VCP uses the header to send the data in each packet to the intended process. These headers can also be used for routing packets through a communication system connecting a number of IMS T9000s together. This extends the idea of multiple channels on a single hardware link to multiple channels through a communications system; a communications channel can be established between any two processes even if they are running on transputers that are not directly connected. The header still just specifies the destination of the packet; the programmer does not need to know how to route that message to its destination.

Advantages for the programmer

The ability to have channels between any two processes in a network has a number of significant advantages for the programmer. It simplifies the description of multiprocessor systems by separating the hardware architecture from the software configuration. The programmer doesn't need to be concerned with the details of placing channels on links or routing messages through the network. This removes a lot of the problems with placing of processes on processors – the decision now can be made just on the basis of the resources (memory size, etc.) available on each processor without worrying about the available connectivity.

The programming model for networks of IMS T9000 transputers is unchanged from that for the first generation of transputers. There is, however, greater flexibility in configuring software. An important feature is that the hardware and software configurations, and therefore their descriptions, can be kept completely independent. The same hardware, and the same description of that hardware, can be used for many different programs.

Routers

The routing components in a network can be separated from the processing elements. Messages can be passed from one processor, through any number of routing devices, to the destination processor. This creates a temporary path through the routing system for that message so, from the programmers point of view, there still appears to be a single channel directly connecting a process on one transputer with a process on another.
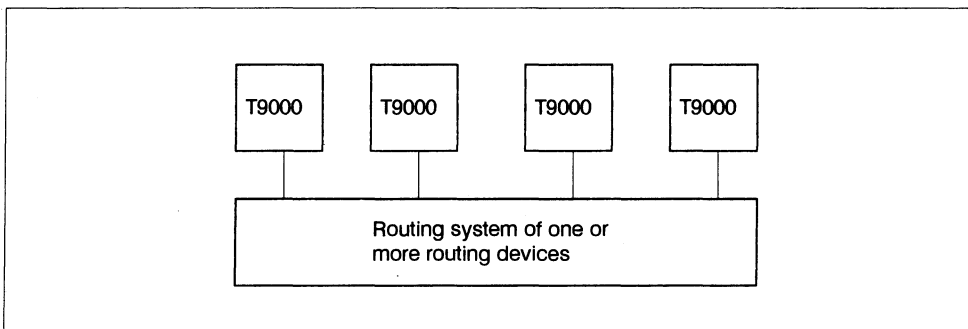
Figure 7.1   A routing system

As a packet arrives on a link, the destination address must be inspected before the outgoing link can be determined. The time before the output link can be determined is therefore proportional to the address

length. Further, the address itself must be transmitted through the network and consumes network band-width. It is therefore important that this address be as short as possible, both to minimize latency and maxi-mize bandwidth.

The router needs to arbitrate between packets which arrive at the same time and have to be sent out of the same link. Ideally, it should start to output the packet as soon as possible; i.e. immediately after the output link is determined, provided that the link is not already in use by another packet. This keeps the latency through the network small, in contrast to a typical packet switching network which uses a 'store and forward' algorithm in which each packet is read into a buffer, the address information is decoded and then the packet is sent out. The delay that would be introduced by this is unacceptable in a transputer net-work. Also the amount of buffering needed would make a VLSI implementation of a large routing switch impractical.

Separating routers and processors

There are a number of advantages to keeping the communications devices and processing elements sepa-rate in a system. Processors can be directly connected where appropriate, which avoids the silicon costs and extra routing delays in a small system that doesn't need to use the routers. Also, the design of the routing devices and processing elements can be optimized for their different roles. For example, the routing component can have a larger number of links than would be possible if the two devices were integrated, because the processor already needs a large number of pins for the memory interface and other functions. Having a routing device with many links means that large network with a small number of routers can be built, hence minimizing cost and latency and maximizing bandwidth. If messages had to flow through the processor, it would increase the pin count, power consumption and packaging costs. This approach also allows the construction of scaleable architectures where the communications throughput and processing power can be balanced.

Parallel networks

Because the new link architecture allows all the virtual channels of a transputer to use a single link, com-plete, system-wide connectivity can be provided by connecting just one link from each transputer to the routing network. This means that the IMS T9000, with its four links, can be connected to several different networks. This can be exploited in a number of ways. For example, two or more networks can be used in parallel to increase bandwidth, to provide a general purpose communications network and an independent monitoring/debugging network, or as a 'user' network running in parallel with a physically separate 'sys-tem' network.

7.2   The IMS C104

An important benefit of the IMS T9000's serial links is that it is easy to implement a full crossbar in VLSI, even with a large number of links. The use of a crossbar allows packets to be passing through all links at the same time, making the best possible use of the available bandwidth.

If the routing logic can be kept simple it can be provided for all the input links in the router. This avoids the need to share the hardware, which would cause extra delays when several packets arrive at the same time. It is also desirable to avoid the need for the large number of packet buffers commonly used in routing sys-tems. The use of small buffers and simple routing hardware allows a single VLSI chip to provide efficient routing between a large number of links.

Wormhole routing

The IMS C104 (figure 7.2) is one of a family of compatible communications support devices for the IMS T9000. It includes a full 32 x 32 non-blocking crossbar switch, enabling messages to be routed from any of its links to any other link. In order to minimize latency, the switch uses 'wormhole routing' – the con-nection through the crossbar is set up as soon as the header has been read. The header and the rest of the packet can start being transmitted from the output link immediately. The path through the switch disap-pears after the 'end of packet/message' token has passed through. This is illustrated in figure 7.3. This method is simple to implement and provides very low latency as the entire packet doesn't have to be read in before the connection is made.
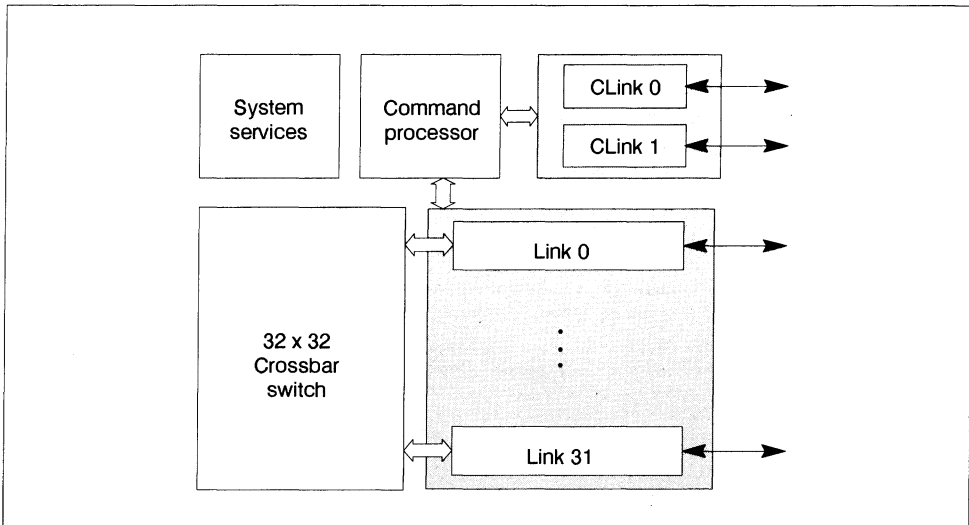
Figure 7.2   Block diagram of IMS C104

Minimizing routing delays

The ability to start outputting a packet while it is still being input can significantly reduce delay, especially in lightly loaded networks. The delay can be further minimized by keeping the headers short and by using fast, simple hardware to determine the link to be used for output. The IMS C104 uses a simple routing algorithm based on interval routing (described in section 7.3.1).
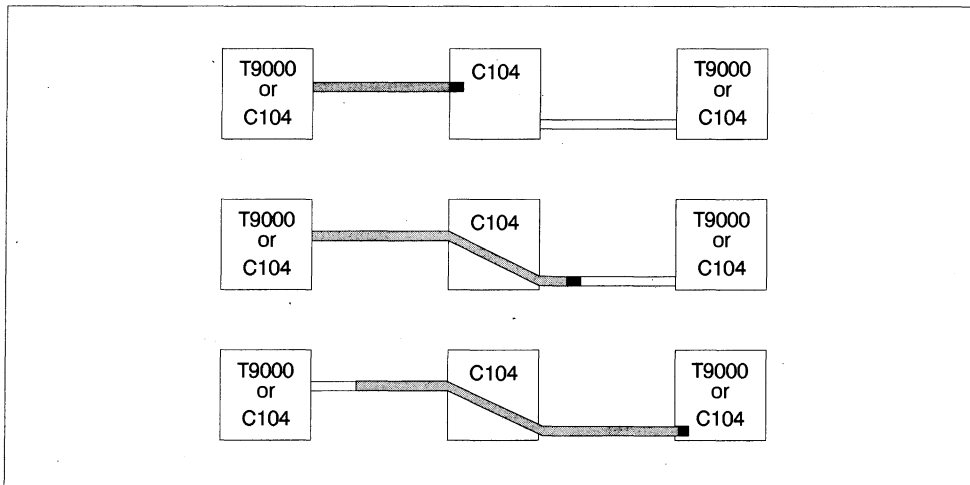
Figure 7.3   Packet passing through IMS C104

Because the route through each IMS C104 disappears as soon as the packet has passed through and the packets from all the channels that pass through a particular link are interleaved, a single virtual channel cannot 'hog' a route through a network. Messages will not be blocked waiting for another message to pass through the system, they will only have to wait for one packet.

Control links

Like the IMS T9000, the IMS C104 has two control links. One link receives control and programming information, the other enables all the devices in a system to be daisy-chained. The routing information for each link of each IMS C104 is programmed, via the control link, from the controlling processor.

7.2.1   Using IMS T9000s with IMS C104s

A single IMS C104 can be used to provide full connectivity between 32 IMS T9000s. It can also be used to connect other compatible communications devices, for example to provide an interface to first generation transputers via a protocol converter, or to peripheral devices via a link adaptor. IMS C104s can also be connected together to build larger switches connecting bigger networks of IMS T9000s.

The IMS C104s that the packets pass through do not need to have information about the complete route to the destination, only which link each packet should be sent out of at each point. Each of the IMS C104s in the network programmed with information that determines which output link should be used for each header value. In this way, each IMS C104 can route packets out of whichever link will send it towards its destination.

Header deletion

An approach that simplifies the construction of networks is to provide two levels of header on each packet. The first header specifies the destination transputer (actually, the output link from the routing network), this header is removed as the packet leaves the routing system. This exposes the second header which tells the VCP in the destination transputer which process (actually, which virtual channel) this packet is for. To support this, the IMS C104 can route packets of any length. Any information after the initial header bytes used by the IMS C104 is just treated as part of the packet, even if it is going to be interpreted as a header elsewhere in the system.The IMS C104 can set any output link to do header deletion, i.e. to remove the routing header from the front of a packet after it been used to make the routing decision. The first part of the remaining data is then treated as a header by the next device that receives the packet.
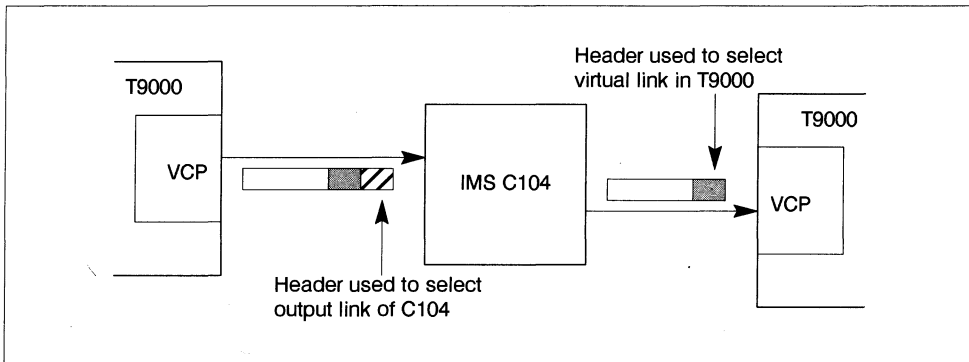
Figure 7.4   Header deletion

As can be seen from figure 7.5, by using separate headers to identify the destination processor and a process within that processor, the labelling of links in a routing network is separated from the labelling of virtual channels within each processor. For instance, if the same 2 byte header were used to do all the routing in a network, then the virtual channels in all the transputers would have to be uniquely labelled with a value in the range 0 to 64K. However, by using two 1 byte headers, all the IMS T9000s can use virtual channel numbers in the range 0 to 255. The first byte of the header will be used by the routing system to ensure that the packets reach the appropriate IMS T9000 before the virtual channel number is decoded.

(a) labeling the system with 2 byte headers

Network of C104s

| T9000 | T9000 | • • • | T9000 |

Virtual
channels: 0–255   256–511   65279–65535

(b) labeling the system with two 1 byte headers

Network of C104s

0   1   255

| T9000 | T9000 | • • • | T9000 |

Virtual
channels: 0–255   0–255   0–255
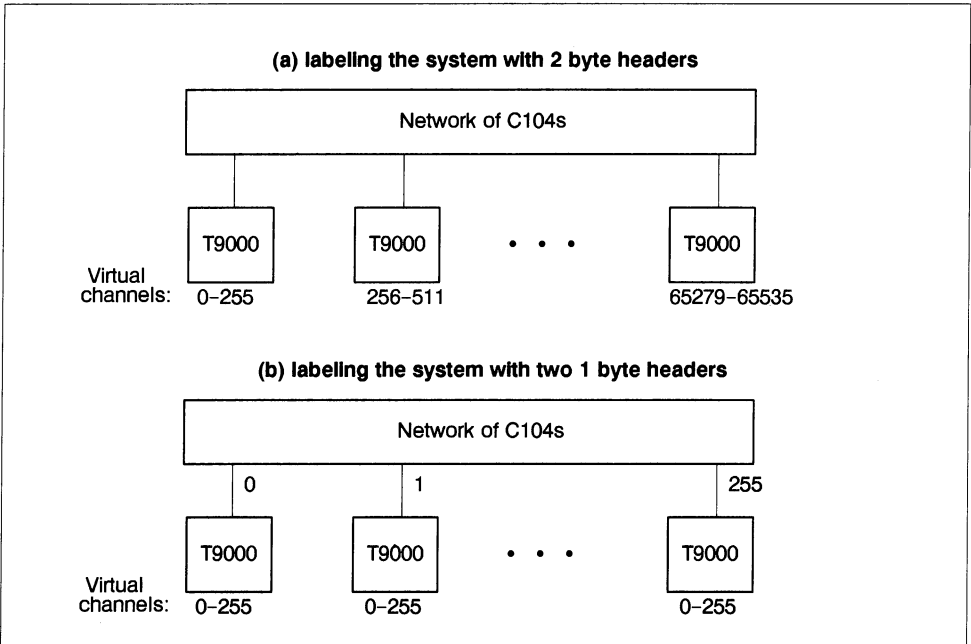
Figure 7.5   Using header deletion to label a network

T9000

sub–network of C104s

used to route packet
through sub–network,
deleted on output.

sub–network of C104s

used to route packet
through sub–network,
deleted on output.

final header used to identify
•   virtual channel on T9000

T9000
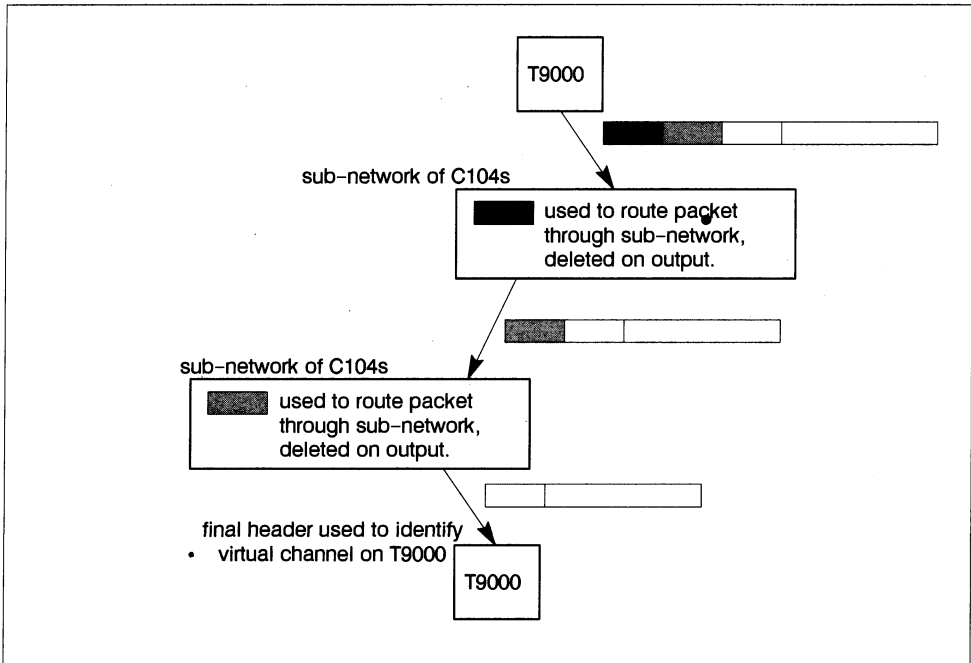
Figure 7.6   Using header deletion to route through sub-networks

The advantages of using header deletion in a network are:

- It separates the headers, and therefore the routing information, for virtual channels from those for the routing network.

- The labelling of the network can be done independently of the application software running on the network.

- There is no limit to the number of virtual channels that can be handled by a system.

Any number of headers can be added to the beginning of a packet so that header deletion can also be used to combine hierarchies of networks as shown in figure 7.6. An extra header is added to route the message through each network. The header at the front of each packet is deleted as it leaves each network to enter a sub-network.

Routing control channels

For very large networks, the usual method of connecting control links, in a chain, might introduce an undesirable delay. In this case, because of the common virtual link protocol, an IMS C104 can be used to route the control links to all the devices in a system more directly, as shown in figure 7.7.
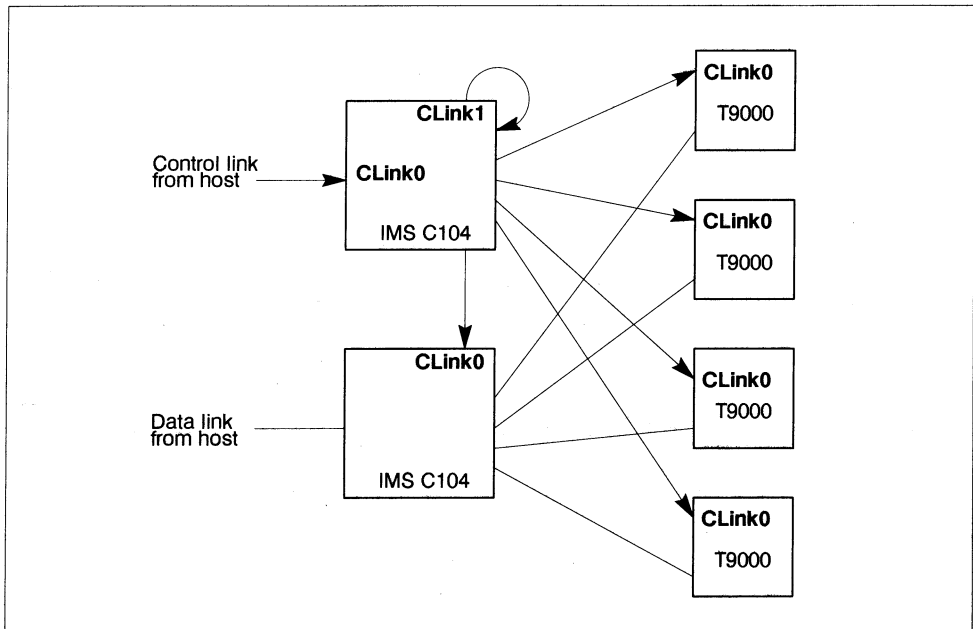
Figure 7.7   Routing control links through an IMS C104

7.3   Routing algorithms

In order to route a message through a network, an algorithm is required which is: complete (ensures that all messages arrive); deadlock free; optimal (packets take the shortest route); scaleable (networks of any size can be built) and simple to implement.

7.3.1 Labelling networks
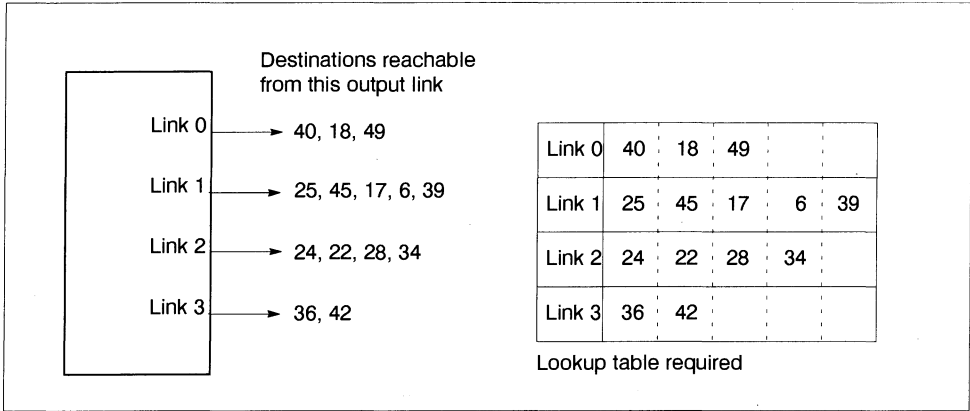
Figure 7.8   Labelling a network

For each routing component there will be a number of destinations which can be reached via each of its output links. Therefore, there needs to be a method of deciding which output link to use for each packet that arrives. The addresses that can be reached through any link will depend on the way the network is labelled. An obvious way of determining which destinations are accessible from each link, is to have a lookup table associated with all the outputs (see figure 7.8). In practice, this is difficult to implement. There must be an upper bound on the lookup table size and it may require a large number of comparisons between the header value and the contents of the table. This is inefficient in silicon area and also potentially slow.
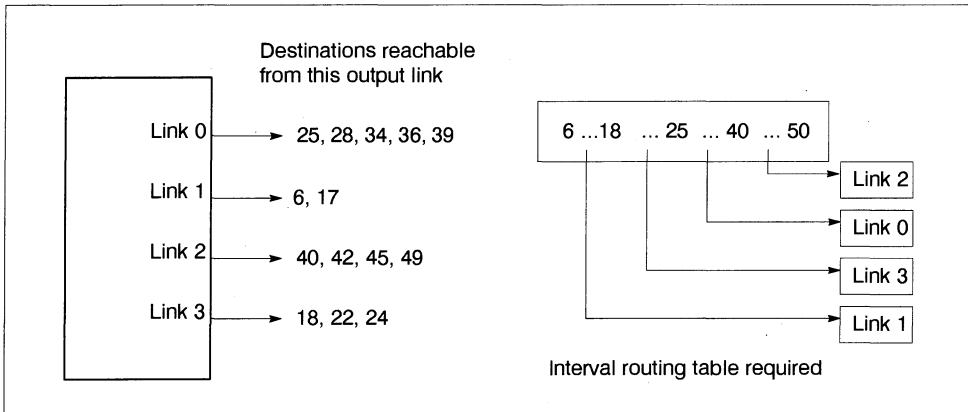
Figure 7.9   Interval labelling

However, a labelling scheme can be chosen for the network such that each output link has a *range* of node addresses that can be reached through it. If it is then ensured that the ranges for each link are non-overlapping, a very simple test is possible. The header just has to be tested to see into which range, or interval, it falls and, hence, which output link to use. For example, in figure 7.9, a header with address n would be tested against each of the four intervals shown below:

Interval	Output link
$6 \leq n < 18$	1
$18 \leq n < 25$	3

$$25 \leq n < 40 \quad \blacklozenge \quad 0$$

$$40 \leq n < 50 \quad \blacklozenge \quad 2$$

The advantages of interval labelling are that:

- It is 'complete' – any network can be labelled.

- It is simple to implement in hardware – it requires little silicon area which means it can be provided for a large number of links as well as keeping costs and power dissipation down.

- Because it is simple, it is also very fast, keeping routing delays to a minimum.

7.3.2   Avoiding deadlock

Deadlock can occur in a network unless the routing algorithm is designed to avoid it. Any program with communicating processes can also deadlock if not designed carefully. It is important here, to distinguish between deadlock as a property of the network and as a property of a program running on the network. A deadlock free network cannot *cause* a program to deadlock (but, of course, neither can it prevent a badly designed program from deadlocking). An essential property of a router in a deadlock free network is that, like a transputer or an IMS C104, it can communicate on all of its links concurrently.

As a simple example consider a network of four nodes (see figure 7.10) with one link in each direction between each node. If the routing algorithm sends all messages clockwise and all nodes start sending to the opposite corner at the same time, every link will become busy and the network will deadlock. It is possible to add buffers to the network, but this will only delay the point at which deadlock occurs. The amount of buffering needed to avoid deadlock is dependent on the network size and the application program running on the network.
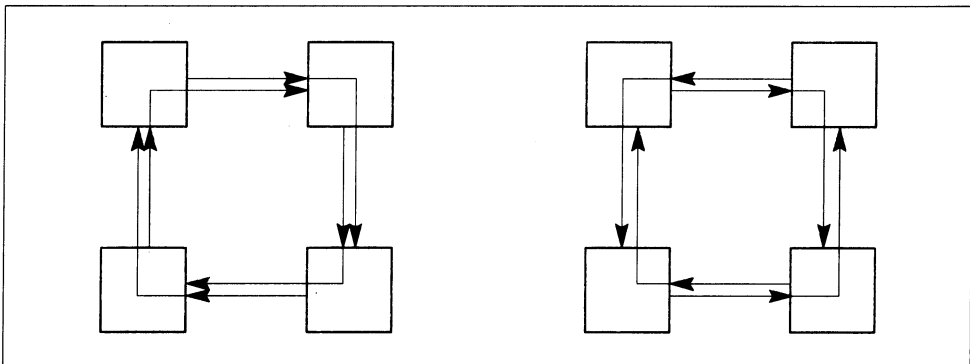
Figure 7.10   Deadlock in a network

In this example, deadlock can easily be avoided by modifying the routing algorithm to send messages in opposite directions from alternate nodes. In this case, each node will only need to send one message in each direction at any time. In this network, buffering can be added just to smooth the flow of data (i.e. to prevent a process having to wait to send a message when the network is busy) but it is not needed to prevent deadlock.

It is possible to use interval labelling to label any network in a deadlock free way. Many regular networks have optimal, deadlock free routing algorithms. Examples are trees, hypercubes and grids. These networks can then be combined, so that any network can be optimally labelled as if constructed from these sub-networks.

8 Other communications devices

To complete the IMS T9000 family, a full range of communications products are planned. These will provide the ability to interface transputers to a range of devices and technologies.

8.1 Mixing transputer types: the IMS C100

The first of these devices is the IMS C100. This allows an IMS T9000 to communicate with a first generation transputer. The two transputer families have different electrical characteristics and data protocol. The IMS C100 converts between the four wire DS links of the IMS T9000 and the two wire OS links of the earlier transputers.

The other conversion done by the IMS C100 is between the IMS T9000 control links and the **Reset**, **Error** and **Analyse** signals used to control the IMS T805 and similar device.

The IMS C100 provides an inter-networking solution for transputers, allowing transputer systems to be constructed using the optimum mix of devices. The IMS C100 has four modes of operation to enable:

- A single IMS T9000 to work in a network of first generation transputers.

- An existing transputer system to control a sub-system of IMS T9000s.

- An IMS T9000 network to interface to a network of first generation transputers.

- A first generation transputer to emulate an IMS T9000.
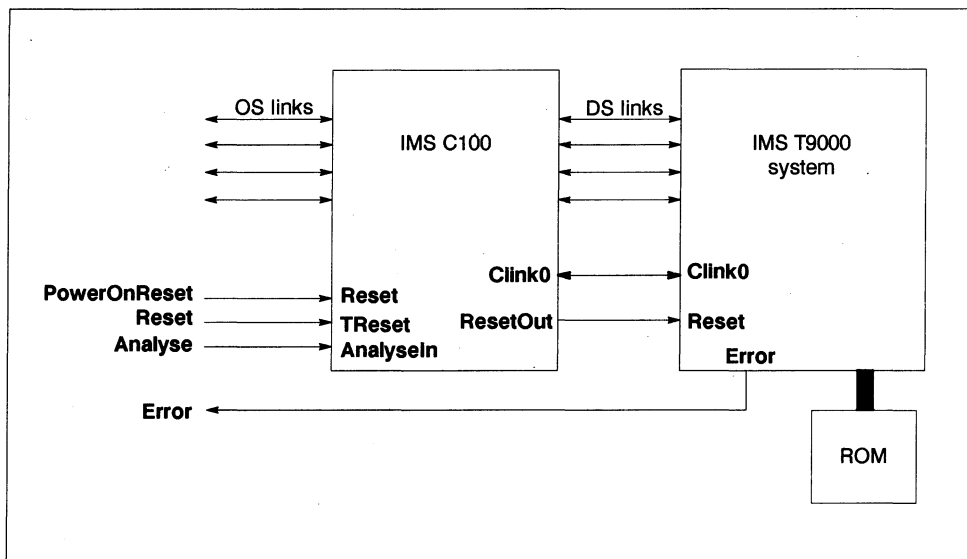
Figure 8.1   IMS C100 used with an IMS T9000

The IMS C100 converts both data and control protocols between the two transputer types and is intended to be used in conjunction with software running on the attached transputers.
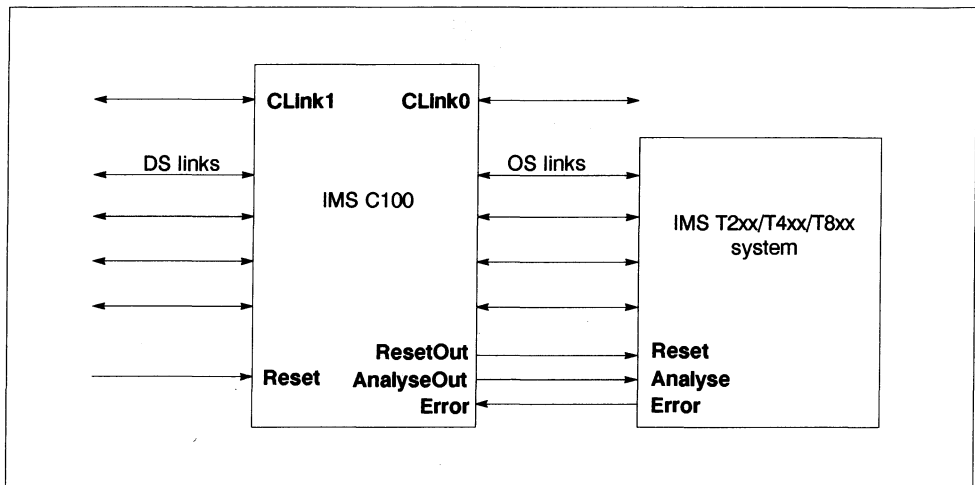
Figure 8.2   IMS C100 used with a first generation transputer

8.2   Interfacing to peripherals and host systems

To complete the family of communications devices, a range of interface devices are being designed. These will convert between the serial link format and a parallel interface, for example. The first of these devices will interface to a microprocessor bus. This will allow the IMS T9000 to communicate with non-transputer systems.

9 Software and systems

INMOS provides a wide range of standard software and hardware products to support development for the transputer. These have been designed to enable users to evaluate transputers and to develop systems easily and within the shortest possible timescales.

Development tools include compilers for languages such as C, Fortran and occam as well as the software needed to test, program and debug systems built from one or many transputers. All the special features of the transputer are available from high level languages (either as part of the language or as library calls).A wide range of software is also available from third-party suppliers including compilers, such as Ada, and real-time kernels, e.g. VRTX and C Executive.

INMOS also supplies a range of modular hardware products. These exploit the ability to build very compact transputer systems (such as an IMS T805 with 4 Mbytes of memory on a board measuring approximately 2.5 cms by 9 cms) to provide a range of small, cost effective 'TRansputer Modules' (TRAMs). These modules can be mounted on a variety of motherboards, which are available for a range of host systems. The motherboards provide an interface to the host development system and can be connected to build larger systems. The standard sizes and interfaces of the modules and motherboards have been adopted by a number of third party developers to extend the range of compatible systems products available to transputer users.

INMOS will continue to support all these standards for the IMS T9000 product family, extending them where necessary to exploit the new features of these products.

More details of the systems and software products currently available for the transputer family can be found in [5].

9.1 Development software

INMOS has a range of development software, running on different hosts, for the transputer family. These tools are aimed mainly at developing code for embedded systems, i.e. not necessarily running under the control of an operating system. It is expected that the end products will either be connected to a host system or will be completely self-contained units.

Software can be developed in standard high level languages using cross-compilers running on a range of host machines. Programs for single transputers can be developed using just conventional programming tools, such as compilers and linkers. All the languages include extensive support, in the form of run-time libraries, for concurrency and communication. It is possible to write a program consisting of many concurrent processes entirely in C (or any other language available for the transputer).

Programs written for multi-transputer systems, or programs written as many sub-programs running in parallel on a single processor, will require the use of extra programming tools. The transputer development system includes tools for preparing a program for execution on a parallel processing system and for debugging such systems. These tools include 'configuration' tools which are used for describing the hardware, mapping processes to transputers and setting up the communications channels. It is possible to boot and load a network with code from the host development system, or from a ROM connected to one of the transputers in the network. Programs can communicate with a 'server' on the host system to get access to host facilities such as i/o. In addition there are tools for debugging a program running on a network of transputers. An outline of some of these transputer specific tools is given below.

All of the programming tools are available for all members of the transputer family and, where appropriate, are used in the same way and provide the same functions for all processor types.

9.1.1 Configuration tools

In discussing IMS T9000 transputer systems, the word 'configuration' is used in two senses. The first is when an IMS T9000 transputer, or an IMS C104, is initialized – at this time a number of internal 'configuration' registers have to be written to program the PMI, the VCP and other subsystems. The process of preparing a program for loading onto a transputer network is also referred to as 'configuration' (and the software

tools used are known as 'configurers'). In this description of the development process, the word 'configuration' is reserved for the latter meaning of software configuration; the setting up of the hardware will be called 'initialization'.

The configuration tools are used to build programs consisting of a number of processes or sub-programs running in parallel on one or more transputers. Input files are used to describe the hardware, the software and a mapping of the software onto the hardware. From these, the configuration tools produce the files which are used to initialize and load the transputer network.

Hardware description

The hardware is described using a Network Description Language (NDL). For each transputer in the system, this specifies the processor type, the amount and types of memory and peripheral devices. It also describes the routing network used, if any, and how the data and control links of all the devices in the system are connected.

The configuration tools use this description to program the PMI and VCP registers of the IMS T9000 and to label the links of any IMS C104s used. The information in this file is also used to create the bootable version of a program to run on the network.

If certain simple rules are followed in the construction and labelling of networks, then the tools can check the descriptions for errors and deadlock freedom. The NDL description can also be checked against the actual hardware.

Software description

The NDL file for a particular system will normally be provided by the hardware vendor or designer. The programmers using the system only need to include a reference to the NDL file in the software configuration file.The NDL description exports the names of the processors and routes in the network for use in the software and mapping description.

The software description has to specify the object code files for each process in the system and the procedure interface (parameters and their types). Optionally, other language dependent attributes can be defined. For example, the size of stack and heap areas for a C program can be specified. The software description must also specify the way that any communication channels are used between processes.

Mapping software to hardware

A mapping of software (processes) onto hardware (transputers) must also be given. The mapping can be as simple as a series of statements of the form: 'place *process* on *processor*' for each process in the program. Any number of processes can be placed on each processor, allowing a program to be initially tested on a single processor before the multi-processor version is tried. The configuration tools automatically work out the mapping of channels onto virtual links. If necessary, for example to access the host system or a particular piece of hardware, the programmer can explicitly map channels onto links or routes through the network.

Configuration languages

To provide a degree of flexibility for the user, there are two 'dialects' of configuration language: a C-like one and an occam-style one. These perform identical functions but each has a different syntax, loosely based on these languages. These configuration languages are used for describing the structure of the software and how it is mapped onto the hardware.

Types of networks

The INMOS development tools support development of programs for:

Networks consisting of IMS T9000 transputers only ('non–routed' networks).

Networks consisting of IMS T9000 transputers and IMS C104 routers ('routed' networks).

Networks consisting of any other transputer types.

The tools do not directly support arbitrary, mixed networks of IMS T9000 transputers and first generation devices. However, it is possible to connect the two types of networks, via an IMS C100, although the code for the two sub-networks has to be developed separately. The two networks can then be loaded from the host, via a common route.

In the case of non-routed networks (of any transputer type) the configuration tools automatically add routing software to the program to provide any communications required between processors which are not directly connected.

A network of IMS T9000 transputers can be loaded with code compiled for an IMS T805 or an IMS T9000. This allows users to write programs for the IMS T9000 even if the compiler used is only available for the IMS T805. It also means that existing, compiled code can be run on an IMS T9000 system.

9.1.2   Initializing and loading a network

Transputer systems can be bootstrapped in two ways; either from ROM or from link. The initialization and initial code loading are done via the control link. This initial boot code then loads the main application code from the data links of the processor.

Levels of initialization

The initialization and loading of code for the IMS T9000 are done in a number of stages. The various levels of initialization can be done either by code running on an IMS T9000 booted from ROM, or from the host system via the control link. In a network, different processors may be initialized to different levels from ROM with the later stages being done via the control link.

Booting a system from link

The 'boot from link' option is normally used during program development or whenever a system needs to be able to run different programs at different times.

In order to load a network from a host system, connections to a single control link and a single data link are required. This data link normally goes directly to an IMS T9000, the rest of the network being loaded via this processor. The development tools generate data files which are used to do all the initialization and loading of code onto the network.

Booting a system from ROM

The development tools can produce a number of different types of ROM. These range in function from performing the (partial) initialization of a single IMS T9000, to booting an entire system.

When booting a system completely from ROM, it is possible to have a single ROM on one processor. This root processor boots from the ROM and then initializes and loads the rest of the network via links; all other transputers in the network being set to boot from link.

9.1.3   Host servers

A server is a program that runs on the host machine to give software, running on an attached transputer system, access to various host facilities such as i/o and disk storage. The server typically loads the executable code onto the transputer network via a link interface. It then waits for requests and data to be sent by the transputer program. These requests generally come from the run-time library, when the program makes calls to standard input and output functions (e.g. printf() in C).

The server allows the development tools running on the host to control the target transputer system in order to reset the system, do any initialization needed and then load a bootable program file. Software running on the host can also use the server to access the transputer system for testing and debugging.

The nature of the connection from the host to the transputer system depends on the type of the host system, but generally provides access to transputer links either directly, via a link adaptor on the host bus, or through some other standard communications system such as Ethernet. In many cases the server software

includes a device driver, which handles the low level details of the hardware interface, plus a set of functions to access the link through the device driver.

9.1.4   Debugging

INMOS provides an interactive symbolic debugger for debugging programs running on networks of transputers. This supports source level debugging of programs which consist of a number of parallel processes running on any number of processors. The user can set breakpoints, inspect the state of processes (including expression evaluation, modification of variables, backtracing procedure calls, etc) as well as examining the low level state of each transputer in the system. A very useful feature is the ability to 'jump' down a communication channel between two processes – this allows the state of two communicating processes to be examined.

The debugger is currently being developed further to make it more powerful and easier to use. Some of the features that will be added are:

- Window based user interface.

- List all processes running in the network.

- Stop processes to examine their state.

- Source level single stepping.

- History tracing (e.g. keeping track of communications events).

- Variable watchpointing.

9.1.5   IMS T805 emulation

An IMS T9000 can be booted from a ROM which performs all initialization and then executes a loader program. The loader then waits for code to arrive on any of the data links. This emulates the behavior of the IMS T805 which, after reset, waits for bootstrap code to arrive on a link. With the addition of an IMS C100 to do protocol conversion, this provides the ability to plug an IMS T9000 directly into an existing transputer network and program it as if it were an IMS T805.

Because the IMS T9000 is binary compatible with, and has the same programming model as, previous transputers the programmer can use existing development tools, source code, libraries and programming techniques.

This compatibility also makes it easy for systems companies to port existing software, such as real-time kernels, compilers and so on, that have already been developed for the current transputer range. This enables an IMS T9000 specific version of these products to be developed very quickly. Additional work can then be done, if necessary, to extend the product to make use of new features of the IMS T9000.

9.2   *i*q Systems products

There is already a wide range of TRAMs and motherboards designed for the first generation of transputers. This includes modules with transputers plus various amounts and types of memory, through to various industry standard interfaces such as SCSI and GPIB. There is a complementary range of motherboards interfacing to hosts such as PC and Sun.

These TRAMs and industry standard motherboards make it easy to develop, prototype and build multiprocessor systems, based on the transputer family.

9.2.1   IMS T9000 products

To support the IMS T9000, a number of products compatible with the existing TRAM definition are being designed. In the longer term a new module standard is being defined to exploit the faster links and other

features of the IMS T9000. In addition the approach to systems design using the IMS T9000 will be somewhat different because of the facilities provided by the virtual links.

The objectives of the new range of systems products are:

- No jumpers or switches on boards – the user can simply plug together modules to build a system and start using it.

- No cables for interconnection within a system – all link interconnections via a backplane and IMS C104s.

- Flexible but fixed network topology – this will be chosen to provide complete connection between all transputers in a system with minimal latency.

- Provision of 3.3V for new high speed, low power components.

- Ability to use existing TRAMs where appropriate.

- Ability to build fault tolerant and 'live insertion' systems.

- Standard interfaces inside and outside the box.

The opportunity will be taken to increase the modularity of the systems components, and to improve the design mechanically and provide better support for peripheral interface connections. To allow flexible interconnection of boards and modules, a backplane architecture is being defined to enable the construction of routed and non-routed systems. These and other changes are being made based on experience and customer feedback.

The new standards will embrace modules, boards and system interconnections. This includes the construction and interconnection of scaleable systems with small (1–10) through medium (up to 64) to large (>256) numbers of processors.

The new module standard will include the provision of a ROM which may be used to provide configuration data for the memory interface, etc. In some cases the processor might boot from this ROM as well. The ROM will also be used to store useful information about the board, such as the module type, serial number, vendor, memory size and speed, and information about other peripheral devices.

The new backplanes will be based on metric standards and will provide a standard backplane interface. This will probably follow the board and connector formats defined by the Futurebus Plus standard.

The transputer links are, naturally, used for connecting between transputers on a single board or within a system. There is also a need for longer connections between systems, for example to support interfacing between a target system and the development host. Two standards are being defined; an electrical connection for distances up to about 10 meters, and a low cost optical fiber interconnect for longer distances.

Compatible development products

Initially, INMOS will provide a TRAM compatible with current standards but containing an IMS T9000. To provide compatibility this will have an IMS C100 to convert the links and control signals. The IMS T9000 will boot from ROM so it can be initialized and then be ready to load code down a data link; it will then appear just like a very fast IMS T805.

This TRAM can be used in an existing development environment to do initial evaluation of code running on the IMS T9000. The program running on this TRAM can communicate but will obviously not have the IMS T9000 advantages of very high speed links and the virtual channel mechanism.

A similar product will be developed, allowing the IMS T9000 to be used in an existing development system, but providing direct connection of the IMS T9000 links. This will enable more realistic multiple IMS T9000 development to be undertaken.

IMS T9000 specific products

INMOS is developing a range of IMS T9000 modules based on the new standard. These will range from simple 'compute only' modules, with a transputer and memory, through to interface modules. These will

provide access to standards such as SCSI, FDDI, etc. Motherboards for these modules will also be supplied, both to the new standard and for popular host computers, such as the IBM PC.

Host interfaces

There is a need to provide a new type of interface to efficiently support communication between a host system and a program running on an IMS T9000 system.

The existence of multiple virtual links into a network can be exploited to simplify the way that software on the transputer accesses host services. This can also be used to provide all transputers with access to the host. The handling of virtual channels on the host could be implemented in hardware for highest performance or software for greatest flexibility and lowest cost. The choice depends on the capabilities of the particular host hardware and operating system, as well as user requirements. For example, the data transfer speed required will be different in a development situation and an accelerator.

The requirements for connecting into the data link network and the control network are quite different. The data links will typically have a relatively small number of virtual channels connecting to the host, but will require very high data rates (especially if the IMS T9000 system is being used an an accelerator or co-processor). There are potentially a very large number of virtual control links but these can run at a lower data rate.

10   References

1 *The transputer databook*, INMOS Limited, 1990

2 *occam 2 reference manual*, Prentice Hall, 1988

3 *The transputer instruction set – A compiler writer's guide*, Prentice Hall, 1988

4 Technical note 5: "Program design for concurrent systems",
   The transputer applications notebook – Systems and performance, INMOS Limited, 1989

5 *The transputer development and iq systems databook*, 2nd Edition, INMOS Limited, 1991

Part 2

Product Family
Preliminary
Information
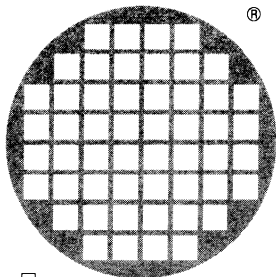
IMS T9000 transputer

Preliminary Information

FEATURES

Instruction set compatible with the IMS T805
Pipelined superscalar micro-architecture
Workspace cache
Programmable memory interface
4 Gbyte physical address space
16 Kbyte instruction and data cache
200 MIPS peak
>70 MIPS sustained
25 MFLOPs peak
>15 MFLOPs sustained
Sub-microsecond interrupt response
Per process error handling
Enhanced support for pre-emptive schedulers
Memory protection and address translation
64 K virtual communication channels
Support for message routing
80 Mbytes/s total bi-directional link bandwidth
Separate control system
Single 5 MHz clock input
50 MHz internal clock
Single 5 V ± 5% power supply
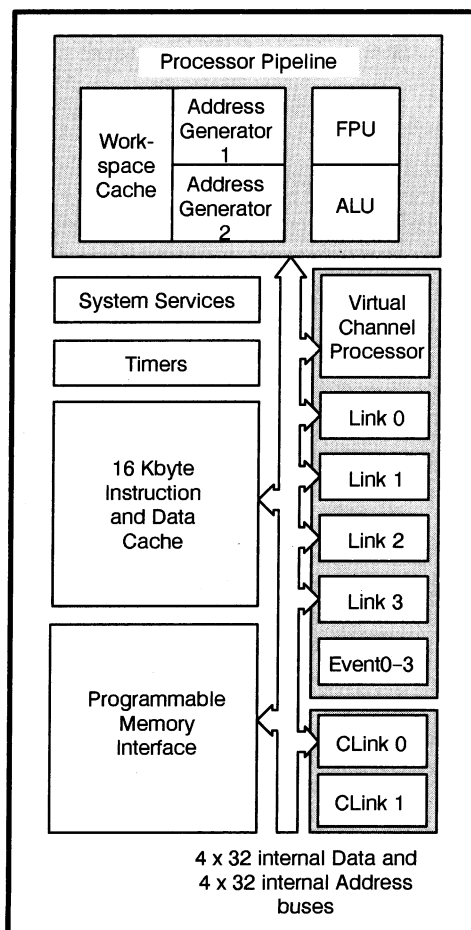
This is preliminary information on a product under development and product details may change.

Processor Pipeline

| Work-space Cache | Address Generator 1 | FPU |
| | Address Generator 2 | ALU |

System Services

Timers

16 Kbyte Instruction and Data Cache

Programmable Memory Interface

Virtual Channel Processor

Link 0

Link 1

Link 2

Link 3

Event0-3

CLink 0

CLink 1

4 x 32 internal Data and
4 x 32 internal Address
buses

1   Introduction

This document contains preliminary hardware information for the IMS T9000 transputer.

The IMS T9000 transputer is a 32–bit CMOS microprocessor designed to be used in applications which require high performance combined with high integration and simplicity of use. It is instruction set compatible with the IMS T805 transputer, with additional support for multiprocessing and real–time applications. Software support for the IMS T9000 transputer includes: ANSI C compilers, ANSI Fortran compilers, and occam compilers, developed and supported by INMOS and third party software companies.

Figure 1.1 shows the major operational units of the IMS T9000 transputer.

The IMS T9000 has a pipelined superscalar architecture, which allows multiple instructions to be executed every processor cycle. Compilers can generate code without considering any details of the pipeline as the hardware organizes the incoming instruction stream into optimum groups of instructions. Other features which contribute to performance are a 16 Kbyte instruction and data cache, a 64–bit floating point unit, and a high bandwidth programmable memory interface. The floating point unit incorporates hardware to perform divide and square root. A separate workspace cache stores 32 locations relative to the workspace pointer to provide zero latency access to local variables. The IMS T9000 has four communication links for fast inter–processor communications.

The 16 Kbyte cache provides a peak bandwidth of 200 Mwords/sec. It can also be programmed to function as 16 Kbyte of on–chip memory, or as 8 Kbyte of on–chip memory and 8 Kbyte of cache. This allows small applications to run with no external memory, and guarantees deterministic code behavior for applications where this is critical.

Transputers provide hardware support for scheduling processes, and this can be used directly by applications written, for example, in C, Fortran or occam. It can also be used to simplify the software implementation of real–time kernels and operating systems. The process model of the IMS T9000 transputer provides per process error handling and debugging support, and allows programs to be run in a protected logical address space. To improve the efficiency of real–time kernels access to the state of the processor has been simplified, and full control over interrupts and timeslicing has been provided.

Communication between processes takes place over channels, and is implemented in hardware. The same machine instructions are used for communication between processes on the same processor as for communication between processes on different IMS T9000 processors. On the IMS T9000, communication between processes on different processors takes place over *virtual* channels. Virtual channels are multiplexed onto each physical link by the virtual channel processor. Communication between IMS T9000 transputers that are not directly connected is achieved by using a separate dynamic routing switch, the IMS C104.

With virtual channels it is not necessary for the programmer to allocate channels to physical links, and the allocation of processes to processors is simplified. The programming of powerful multiprocessor systems is therefore flexible and elegant.

The IMS T9000 has four high bandwidth serial communication links. To support virtual channels and dynamic message switching, and to provide a higher data bandwidth with high data integrity, each physical link consists of four wires, two in each direction, one carrying data and one carrying a strobe. The links are therefore referred to as data–strobe (DS) links. The four DS links support a total bidirectional data bandwidth of 80 Mbytes/sec.

Two separate control links are provided to enable networks of IMS T9000 processors to be controlled and monitored for errors, even during the presence of faults in the normal data communications network. The control links of IMS T9000s and IMS C104s can be daisy chained, and/or connected into a tree by connection to a IMS C104. Whatever the physical connectivity the controlling network forms a logical tree, and a control processor is connected at its root. For small systems (such as a single IMS T9000 transputer) there is no need to use the control links as all necessary functionality can be controlled from software.

The highly integrated programmable memory interface has a 4 Gbyte physical address space, and provides a peak bandwidth of 50 Mwords/sec. Four independent banks of external memory are supported,

and this allows the implementation of mixed memory systems, with support for DRAM, SRAM, EPROM and VRAM. It has a 64-bit data bus, and each bank of memory can be configured to be 8, 16, 32 or 64 bits wide. The full performance of the IMS T9000 can be exploited using relatively low-cost DRAM, and up to 8 Mbytes of DRAM can be connected with no external components.
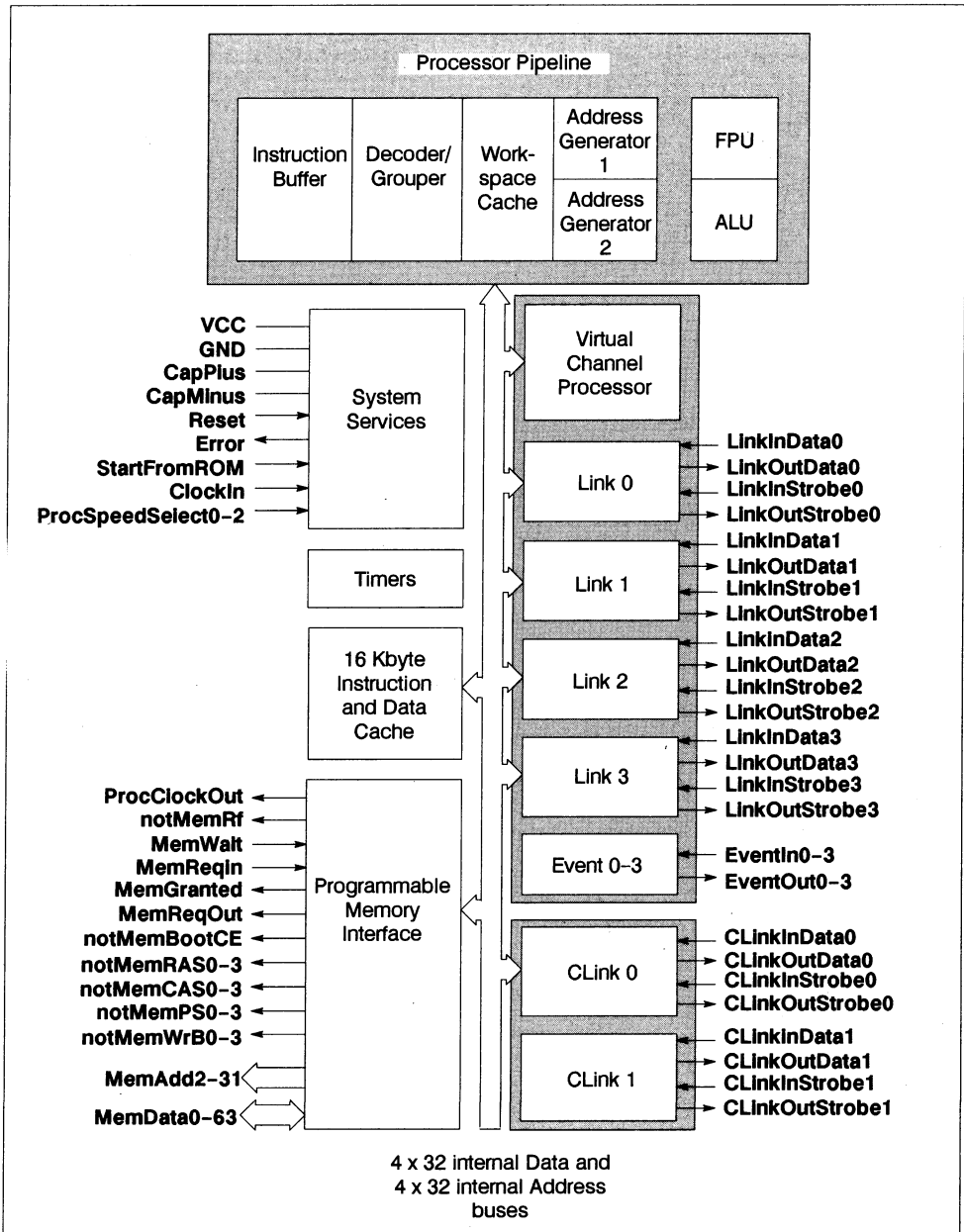
Figure 1.1   IMS T9000 block diagram

2   Preliminary pin designations

Signal names are prefixed by **not** if they are active low, otherwise they are active high.

Pin	In/Out	Function
VCC, GND		Power supply and return
CapPlus, CapMinus		External capacitor for internal clock power supply
ClockIn	in	Input clock
ProcSpeedSelect0-2	in	Processor speed selectors
Reset	in	System reset
StartFromROM	in	Boot from external ROM or from link
Error	out	Error indicator

Table 2.1   IMS T9000 system services

Pin	In/Out	Function
ProcClockOut	out	Processor clock
MemAdd2-31	out	Address bus
MemData0-63	in/out	Data bus
notMemRAS0-3	out	RAS strobes – one per bank
notMemCAS0-3	out	CAS strobes – one per bank
notMemPS0-3	out	Programmable strobes – one per bank
notMemWrB0-3 †	out	Byte–addressing write strobes
MemWait	in	Memory cycle extender
MemReqIn	in	Direct memory access request
MemGranted	out	Direct memory access granted
MemReqOut	out	Processor requires memory bus
notMemBootCE	out	Bootstrap ROM chip enable
notMemRf	out	Dynamic memory refresh indicator

† these pins have different functions depending on the external port sizes

Table 2.2   IMS T9000 programmable memory interface

Pin	In/Out	Function
EventIn0-3	in	Event inputs
EventOut0-3	out	Event outputs

Table 2.3   IMS T9000 event

Pin	In/Out	Function
LinkInData0–3	in	Link input data channels
LinkInStrobe0–3	in	Link input strobes
LinkOutData0–3	out	Link output data channels
LinkOutStrobe0–3	out	Link output strobes
CLinkInData0–1	in	Control link input data channels
CLinkInStrobe0–1	in	Control link input strobes
CLinkOutData0–1	out	Control link output data channels
CLinkOutStrobe0–1	out	Control link output strobes

Table 2.4   IMS T9000 link

3   Processor

The IMS T9000 transputer has a 32–bit pipelined processor. The pipeline consists of 5 stages and, where possible, multiple instructions are combined into a group and passed down the pipeline together. This allows more than one instruction to be executed on each processor cycle. Code can be generated for the IMS T9000 transputer without considering the details of the pipeline. However, optimizing compilers can produce more efficient code if these details are taken into consideration.

Background details of earlier transputers can be found in *Transputer Instruction Set – A Compiler Writers' Guide.* Much of the information in this guide can be directly applied to the IMS T9000 transputer. This preliminary information outlines the implications of the extensions which have been implemented in the IMS T9000 transputer.

3.1   Registers

The design of the IMS T9000 transputer processor exploits the availability of a fast on–chip cache and a workspace cache by having only a small number of registers; six registers are used in the execution of a sequential integer process. The six registers are:

- The workspace pointer which points to an area of store where local variables are kept.

- The instruction pointer which points to the next instruction to be executed.

- The operand register which is used in the formation of instruction operands.

- The **Areg**, **Breg** and **Creg** registers which form an evaluation stack.

Areg, **Breg** and **Creg** are sources and destinations for most arithmetic and logical operations. Loading a value into the stack pushes **Breg** into **Creg**, and **Areg** into **Breg**, before loading **Areg**. Storing a value from **Areg**, pops **Breg** into **Areg** and **Creg** into **Breg**, the value left in **Creg** is undefined.

Expressions are evaluated on the evaluation stack, and instructions refer to the stack implicitly. For example, the *add* instruction adds the top two values in the stack and places the result on the top of the stack. The use of a stack removes the need for instructions to re–specify the location of their operands. No hardware mechanism is provided to detect that more than three values have been loaded onto the stack. It is easy for the compiler to ensure that this never happens.

A separate floating point evaluation stack is provided, consisting of **FPAreg**, **FPBreg**, and **FPCreg**. The floating point evaluation stack behaves in a similar way to the integer evaluation stack.

Any location in memory can be accessed relative to the workspace pointer, enabling the workspace to be of any size. The first 32 words relative to the workspace pointer may be cached by the workspace cache.
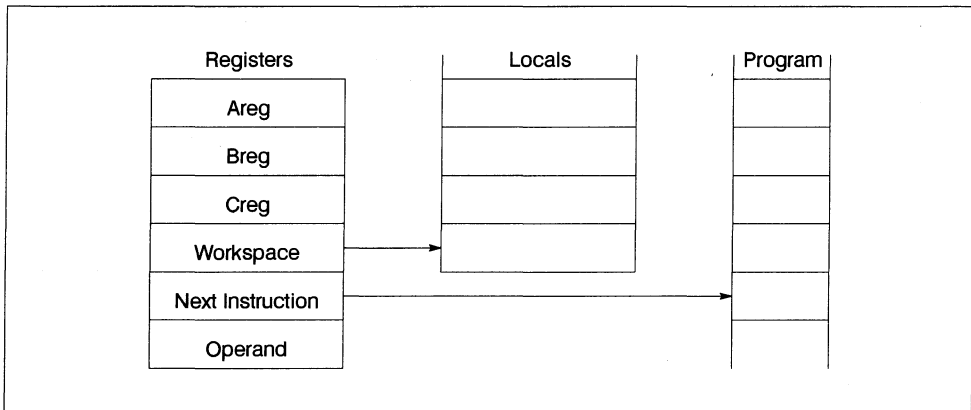
Figure 3.1   Registers used in sequential integer processes

3.2   Processes and concurrency

A process starts, performs a number of actions, and then either stops without completing or terminates complete. Typically, a process is a sequence of instructions. A transputer can run several processes in parallel (concurrently). Processes may be assigned either high or low priority, and there may be any number of each.

The processor has a microcoded scheduler which enables any number of concurrent processes to be executed together, sharing the processor time. This removes the need for a software kernel, although kernels can still be written.

At any time, a process may be

Active	–	Being executed.
	–	Interrupted by a higher priority process.
	–	On a list waiting to be executed.
Inactive	–	Ready to input.
	–	Ready to output.
	–	Waiting until a specified time.

The scheduler operates in such a way that inactive processes do not consume any processor time. Each active high priority process executes in turn until it becomes inactive. The scheduler allocates a portion of the processor's time to each active low–priority process in turn (see section 3.3). Active processes waiting to be executed are held in two linked lists of process workspaces, one of high priority processes and one of low priority processes. Each list is implemented using two registers, one of which points to the first process in the list, the other to the last. In the linked process list shown in figure 3.2, process S is executing and P, Q and R are active, awaiting execution. Only the low priority process queue registers are shown; the high priority process ones behave in a similar manner.
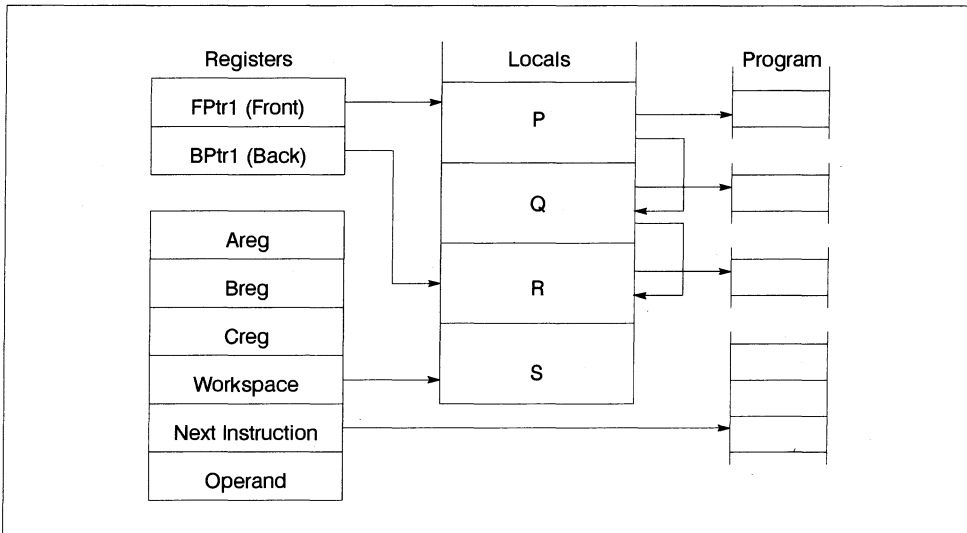
Figure 3.2   Linked process list

Function	High Priority	Low Priority
Pointer to front of active process list	**Fptr0**	**Fptr1**
Pointer to back of active process list	**Bptr0**	**Bptr1**

Table 3.1   Priority queue control registers

Each process runs until it has completed its action or is descheduled whilst waiting (for a communication from another process or transputer, or for a time delay to complete). In order for several processes to operate in parallel, a low-priority process is only permitted to execute for a maximum of two timeslice periods. After this, the machine deschedules the current process at the next timeslicing point, adds it to the end of the low-priority scheduling list and instead executes the next active process. The timeslice period is approximately 1 ms.

There are only certain instructions at which a process may be descheduled. These are known as descheduling points. A process may only be timesliced at certain descheduling points. These are known as timeslicing points. As a result, an expression evaluation can be guaranteed to execute without the process being timesliced part way through.

Whenever a process is unable to proceed, its instruction pointer is saved in the process workspace and the next process taken from the list. Process scheduling pointers are updated by instructions which cause scheduling operations, and pointers or active queues should not be altered directly.

The processor provides a number of special instructions to support the process model, including *start process* and *end process*. When a main process executes a parallel construct, *start process* instructions are used to create the necessary additional concurrent processes. A *start process* instruction creates a new process by adding a new workspace to the end of the scheduling list, enabling the new concurrent process to be executed together with the ones already being executed. When a process is made active it is always added to the end of the list, and thus cannot pre-empt processes already on the same list.

The correct termination of a parallel construct is assured by use of the *end process* instruction. This uses a workspace location as a counter of the parallel construct components which have still to terminate. The counter is initialized to the number of components before the processes are started. Each component ends with an *end process* instruction which decrements and tests the counter. For all but the last component, the counter is non zero and the component is descheduled. For the last component, the counter is zero and the main process continues.

3.3   Priority

The IMS T9000 transputer directly supports two levels of priority. Priority 1 (low priority) processes are executed whenever there are no active priority 0 (high priority) processes.

High priority processes are expected to execute for a short time. If one or more high priority processes are able to proceed, then the first on the queue is selected and executes until it has to wait for a communication, a timer input, or until it completes processing.

If no process at high priority is able to proceed, but one or more processes at low priority are able to proceed, then one is selected. Low priority processes are periodically timesliced to provide an even distribution of processor time between computationally intensive tasks.

If there are **n** low priority processes, then the maximum latency from the time at which a low priority process becomes active to the time when it starts processing is 2 **n** –2 timeslice periods. It is then able to execute for between one and two timeslice periods, less any time taken by high priority processes. This assumes that no process monopolizes the transputer's time; i.e. it has a distribution of timeslicing points.

When the processor is executing a low–priority process and a high–priority process becomes ready to execute, an interrupt occurs. The state of the low–priority process is saved into 'shadow' registers and the high–priority process is executed. When no further high–priority processes are able to run, the state of the interrupted low–priority process is loaded from the shadow registers and the low–priority process is restarted.

Instructions are provided on the IMS T9000 transputer to allow a high–priority process to store the shadow registers to memory and to load them from memory. Instructions are also provided to allow a process to exchange an alternative process queue for either priority process queue. These instructions enable a pre-emptive scheduler to be constructed.

Note that the workspace pointer is always word aligned so that bits 0 and 1 of the **WdescReg** register are free to store the process priority and type. The priority of a process is stored as bit 0 of the **WdescReg** register. For a low priority process this bit is set to 1, for a high priority process to 0.

3.4   Process types

The IMS T9000 transputer schedules two types of process; one is identical to that provided by existing transputers, the other provides additional trap-handling and debugging capabilities.

When running a process of the first type the IMS T9000 transputer implements the same global trap-handling and debugging mechanisms as the IMS T225, IMS T425, IMS T805 and IMS T801 transputers. Processes of this type are therefore referred to as G-processes.

When running a process of the second type the IMS T9000 transputer provides a set of localized, per-process, trap-handling and debugging mechanisms. Processes of this type are therefore referred to as L-processes.

The type of a process is stored as bit 1 of the **WdescReg** register. For a G-process this bit is set to 0, for an L-process it is set to 1. Both types of process may be present on the process queue at the same time, the IMS T9000 dynamically switches to and from emulating the IMS T805.

3.4.1   G-processes: global trap-handling and debugging

The layout of the workspace for a G-process is shown in table 3.2.

Word offset	Location name	Purpose
−1	**p.Iptr**	the instruction pointer of a descheduled process
−2	**p.Link**	the address of the workspace of the next process in scheduling queue
−2	**p.Count**	message length in variable length communication
−3	**p.Pointer**	saved pointer to communication data area
−3	**p.State**	saved alternative state
−3	**p.Length**	length of message received in variable length communication
−4	**p.TLink**	address of the workspace of the next process on the timer queue
−5	**p.Time**	time that a process on a timer list is waiting for

Table 3.2   Word offsets from **Wptr** and names for data locations in a G-process workspace

Note that in some cases, a word offset is shared by more than one location name. This is because the location specified by such an offset is used for a number of different purposes at different times. For example when the **p.Count** slot contains information about the message length, the process is not on a scheduling queue and so the location is not required to contain **p.Link** information.

There are two flags which indicate errors: the **ErrorFlag**, which indicates errors detected within the CPU; and the **FPErrorFlag**, which indicates errors detected within the FPU. The *testerr* instruction sets **Areg** to *false* if the error flag is set, and *true* otherwise. It also clears the error flag. The *stoperr* instruction deschedules the current process if the **ErrorFlag** is set, allowing graceful system degradation when execution of a process gives rise to an error. *stoperr* does not affect the status of the error flag. The *fptesterr* instruction sets **Areg** to *false* if the floating point error flag is set, and to *true* otherwise. The *fptesterr* instruction also clears the floating point error flag. The *fpchkerr* instruction OR's the floating point error flag into the main error flag. This allows floating point errors to be given equal importance to errors on the integer processor.

If the **HaltOnErrorFlag** is set and the **ErrorFlag** is set then the processor will halt and an error message will be output on the control link (**CLink0**).

In G-processes a number of instructions facilitate the implementation of breakpoints. These instructions overload the operation of *j0* . Normally *j0* behaves as a no-op which might cause timeslicing. *Setj0break* enables the breakpointing facilities and causes *j0* to act as a breakpointing instruction. When breakpointing is enabled, *j0* swaps the current **IptrReg** and **Wptr** with an **IptrReg** and **Wptr** stored in memory above **MemStart**. The breakpoint instruction does not cause timeslicing, and preserves the state of the registers.

For further information on G-processes in general refer to Chapters 6 and 7 of *Transputer Instruction Set – A Compiler Writer's Guide*.

3.4.2   L–processes: local error handling and debugging

This is a new process type introduced on the IMS T9000 transputer to allow error conditions to be handled on a per process basis. The layout of the workspace for an L–process is shown in table 3.3.

Word offset	Location name	Purpose
–1	**p.Iptr**	the instruction pointer of a descheduled process
–2	**p.Link**	the address of the workspace of the next process in scheduling queue
–2	**p.Count**	message length in variable length communication
–3	**p.TrapHandler**	trap-handler identity
–4	**p.Pointer**	saved pointer to communication data area
–4	**p.State**	saved alternative state
–4	**p.Length**	length of message received in variable length communication
–5	**p.TLink**	address of the workspace of the next process on the timer queue
–6	**p.Time**	time that a process on a timer list is waiting for

Table 3.3   Word offsets from **Wptr** and names for data locations in a L-process workspace

Each L–process has a trap-handler, a set of error flags, and a set of trap enable bits. Whenever an error is detected the appropriate error flag is set, and depending on the state of the trap enable bits, the trap-handler is invoked. Trap-handlers may be shared between processes of the same priority.

When an L–process is executing the identity of the trap-handler is held in the trap-handler register (**ThReg**). When an L–process is inactive the identity of the trap-handler is held in the process workspace.

If the value of the trap-handler in the workspace of an L–process is *NotProcess.p* this indicates that the null trap-handler will be used. Any process which executes with the null trap-handler ignores any floating point errors and any invalid non–word aligned accesses. Any other error results in the processor halting and an error message being output on the control link **CLink0**.

A trap-handler consists of a trap-handler data structure (THDS) and a process to be executed when an error occurs. The THDS contains: a block of store into which state can be saved when an error occurs; the identity of the trap handler process; and a queue of processes waiting to use the trap handler. The layout of a THDS is shown in table 3.4.

Location name	Purpose
th.Cntl	Control word
th.Wptr	**Wptr** of trap-handler process
th.Iptr	**IptrReg** of trap-handler process
th.Fptr	Front of trap-handler process queue
th.Bptr	Back of trap-handler process queue
th.Eptr	Pointer to instruction causing error
th.eWu	Upper bound for watchpoint
th.eWl	Lower bound for watchpoint
th.sWptr	L-process descriptor
th.sIptr	L-process instruction pointer
th.sAreg	L-process A register
th.sBreg	L-process B register
th.sCreg	L-process C register

Table 3.4   Contents of a trap-handler data structure

The control word is used to control the operation of the trap-handler, and to store the flags and trap enable bits whenever the trap-handler is not being used by an executing process. When an L–process starts to execute (and its trap-handler is not already in use by a process sharing it) its trap-handler is loaded from its workspace into the **ThReg**, the trap control bits in its control word are loaded into the **StatusReg** and the process is allowed to execute.

The trap control bits contained in the trap-handler control word are shown in table 3.5. The control word contains a number of flags which may be set when error conditions occur. A trap enable bit is usually associated with each flag.

Flag name	Function	Bit name	Function
ErrorFlag	T8xx compatible error flag		
FPErrorFlag	T8xx compatible FP error flag	**FPErrorTeBit**	
IntErrorFlag	Error explicitly set or range		
IntOvFlag	Integer overflow or divide by zero	**IntOvTeBit**	
FPInOpFlag	IEEE invalid operation flag	**FPInOpTeBit**	
FPDivByZeroFlag	IEEE divide by zero flag	**FPDivByZeroTeBit**	
FPOvFlag	IEEE overflow flag	**FPOvTeBit**	
FPUndFlag	IEEE underflow flag	**FPUndTeBit**	
FPInexFlag	IEEE inexact result flag	**FPInexTeBit**	
		UnalignTeBit	trap unaligned access
		StepBit	single-stepping enabled
		WtchPntEnbl	watchpoint enabled
		ThInUse	trap-handler in use

Table 3.5   Trap control bits of trap-handler control word

All error conditions are classified into one or more error classes. The state of the trap enable bits determine whether the trap-handler is invoked when an error condition occurs. The error classes supported, the flags they set, and the conditions for them to invoke the trap-handler, are detailed in table 3.6.

Error class	Cause of error	Flags set	Condition for trap to be taken
IntegerError	Error explicitly set or range error.	**ErrorFlag, IntErrorFlag**	Always
IntegerOverflow	An integer overflow or divide by zero.	**ErrorFlag, IntOvFlag**	If null trap-handler or if **IntOvTeBit** set
FPNanOrInfinity	Operations involving NaN or Infinity that would cause the **FPErrorFlag** on the T8 transputer to be set.	**FPErrorFlag, FPInOpFlag** (if a signalling NaN is involved)	If **FPErrorTeBit** set
FPInvalidOp	An IEEE floating point invalid operation.	**FPErrorFlag, FPInOpFlag**	If **FPErrorTeBit** or **FPInOpTeBit** set
FPDivideByZero	An IEEE floating point divide by zero operation.	**FPErrorFlag, FPDivByZeroFlag**	If **FPErrorTeBit** or **FPDivByZeroTeBit** set
FPOverflow	An IEEE floating point overflow operation.	**FPErrorFlag, FPOvFlag**	If **FPErrorTeBit** or **FPOvTeBit** set
FPUnderflow	An IEEE floating point underflow operation.	**FPUndFlag**	If **FPUndTeBit** set
FPInexact	An IEEE floating point inexact operation.	**FPInexFlag**	If **FPInexTeBit** set
Unalign	An operation involving a non-word aligned address.	None	If **UnalignTeBit** set
IllegalInstruction	An illegal instruction.	None	Always

Table 3.6   Error classes

The classes of error that can be generated by each instruction are given in the instruction set definition tables (chapter 11). L–processes can be set up to invoke the trap-handler on precisely the same types of error as would have been detected by a T8xx transputer by setting the **IntOvTeBit** and the **FPErrorTeBit**.

In the T8xx transputer the two least significant bits of an address are ignored by instructions that reference a word. However, L–processes on the IMS T9000 transputer will treat non–word aligned accesses as errors if the **UnAlignTeBit** is set.

The **ThInUse** bit acts as an interlock to prevent more than one process using the same trap-handler. This is achieved by setting the bit in the control word when the trap-handler is entered and clearing it when the trap-handler is exited. Before an L–process is executed the processor checks the **ThInUse** bit in the control word of its trap-handler. If the trap-handler is found to be in use then the L–process is queued onto the trap-handler's process queue. All of the processes on the trap-handler's process queue are dequeued and inserted onto the front of the appropriate process queue when the trap-handler is exited.

When a trap-handler is invoked the integer state of the processor is written to the THDS. The floating point state is restored to the state which was present *before* the operation was performed. (This makes it simple for the trap-handler to compute the correct value to be delivered to an IEEE exception handler.). The floating point and block move state of the processor is not saved by the hardware, and it is left to the trap-handler to save this state as necessary using the *floating point store all (fpstall)* and *store 2D move (stmove2dinit)* instructions. The error flags and trap enable bits are written from the **StatusReg** to the control word of the trap-handler, and the **ThInUse** bit is set. The trap-handler process is then started with codes for the trap cause being returned in **Areg** and **Breg**, and **Creg** containing a pointer to the trap-handler just invoked.

Once a trap-handler has completed it loads any floating point and block move state using the *fpldall* and *move2dinit* instructions respectively, and executes the *trap return* instruction. The **Areg** contains a pointer to the trap-handler, and **Breg** contains a conditional argument to the *trap return* instruction. If the value in **Breg** is zero then the errant process will be descheduled. If it is not zero then the error flags and trap enable bits will be reloaded into the **StatusReg** from the trap-handler control word, the integer state of the processor will be reloaded from the THDS, and the errant process will be allowed to continue.

The current error flags and trap enable bits may be examined by using the *load error flags* instruction, which pushes the error flags and trap enable bits from the **StatusReg** into **Areg**. The error flags and trap enable bits in the **StatusReg** may be set to the value in the **Areg** using the *store error flags* instruction.

Any block of store may be used as a THDS. The block must be initialized by: setting **th.Fptr** to *NotProcess.p*; initializing **th.Wptr** and **th.Iptr** for a suitable process; and setting the **ThInUse** bit of the control word to 0 and the other bits selecting traps as desired.

When running an L–process the IMS T9000 transputer provides support for breakpointing, for single-stepping of instructions, and for a watchpointed region. For this process type the IMS T9000 transputer *always* interprets the *j0* instruction as a breakpoint which causes the trap-handler to be called, with the state of the process being saved as described above.

Single-stepping can be enabled by setting the **StepBit** in the trap-handler control word. If this bit is set, then a trap occurs after execution of a single instruction by an L-process.

A watchpointed region is supported by the upper and lower watchpoint bounds specified in the trap-handler data structure. If the watchpoint enable bit (**WtchPntEnbl**) in the trap-handler control word is set then a write to an address between these bounds causes the trap-handler to be invoked.

3.5   Timers

The transputer has two 32–bit timer clocks which 'tick' periodically. The timers provide accurate process timing, allowing processes to deschedule themselves until a specific 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.

Clock0	Current value of high priority (level 0) process clock
Clock1	Current value of low priority (level 1) process clock
TNextReg0	Indicates time of earliest event on high priority (level 0) timer queue
TNextReg1	Indicates time of earliest event on low priority (level 1) timer queue

Table 3.7   Timer registers

The current value of the processor clock can be read by executing a *load timer* instruction. 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' then the process is descheduled. When the specified time is reached the process is scheduled again.

Figure 3.3 shows two processes waiting on the timer queue, one waiting for time 21, the other for time 31.
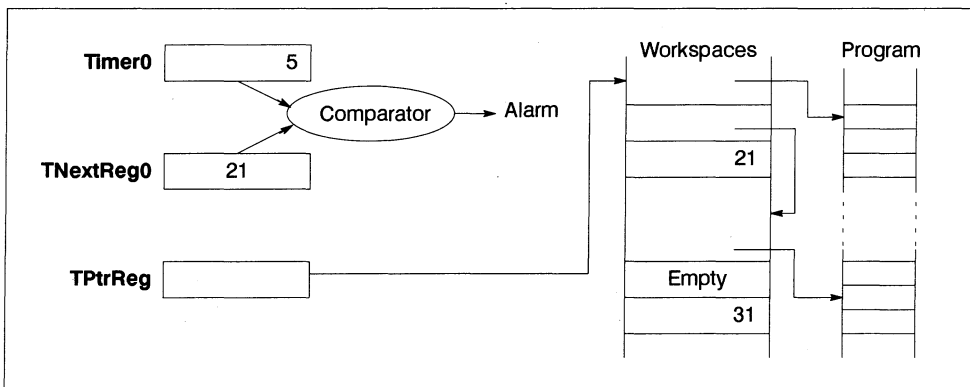
Figure 3.3   Timer registers

3.6   Block move

The block move on the transputer moves any number of bytes from any byte boundary in memory, to any other byte boundary, using the smallest possible number of word read, and word or part–word writes.

3.7   Semaphores

The IMS T9000 transputer provides an efficient implementation of an n–valued semaphore for processes on the same processor. *signal* and *wait* instructions are provided which operate on a data structure which may be located at any address in memory. A semaphore is implemented by a three word data structure. The word locations in the data structure are shown in figure 3.8. The data structure must be initialized with **s.Count** set to n for an n–valued semaphore and with **s.Front** set to *NotProcess.p*.

Location name	Purpose
s.Count	Number of processes which may be granted semaphore
s.Front	Front of waiting queue
s.Back	Back of waiting queue

Table 3.8   Contents of a semaphore data structure

4   Communications, events and resources

Communication between processes may be achieved by means of channels. Channel communication is point–to–point, synchronized, uni-directional and unbuffered. As a result, a channel needs no process queue, no message queue and no message buffer, and so can be implemented very efficiently.

An internal channel between two processes executing on the same transputer is implemented by a single word in memory; an external channel between processes executing on different transputers is implemented by means of point–to–point links. The processor provides a number of operations to support message passing along channels, the most important being *input message (input)* and *output message (output)*.

The *input* and *output* instructions use the address of the channel to determine whether the channel is internal or external. Thus the same instruction sequence can be used for either, allowing a process to be written and compiled without knowledge of where its channels are connected.

Channel communication takes place when both the inputting and outputting processes are ready. Thus, the process which first becomes ready must wait until the second is also ready. A process performs an input or output by loading the evaluation stack with; a pointer to a message, the address of a channel, and a count of the number of bytes to be transferred, and then executing an *input* or *output* instruction. Data is transferred if the other process is ready. If the other process is not ready then the one executing the communications instruction will be descheduled.

4.1   Efficient variable–length communications

Communication using the *input* and *output* instructions requires both communication processes to have knowledge of the length of the message that is to be transferred. To allow the secure and efficient communication of variable–length data, the *vin (variable input)* and *vout (variable output)* instructions may be used instead of *input* and *output*. Variable length communication requires only the outputting process to have knowledge of the length of the message prior to transfer.

When both a *vin* and a *vout* instruction have been executed by processes referring to the same channel, providing the length specified by *vout* does not exceed the length specified by *vin*, data is transferred from the outputting process to the inputting process just the same as if *input* and *output* had been used.

However, in the case where the length specified by *vout* exceeds that specified by *vin*, a –1 is returned in the count location of the workspace of the inputting process, to indicate that an error has occurred in communication.

The *ldcnt (load count)* instruction is provided to enable the inputting process to determine either how much data was transferred during a variable length communication, or whether an error in communication occurred.

4.2   Processor–to–processor communications

The IMS T9000 incorporates a hardware communications processor, called the *Virtual Channel Processor* (VCP), which is able to multiplex any number of *virtual channels* over each physical link. Each message is split into a sequence of packets, and packets from different messages may be interleaved over each physical link. Interleaving packets from different messages allows any number of processes to communicate simultaneously via each physical link. IMS T9000 transputers may be connected directly or via a network of IMS C104 dynamic routing devices. Communication channels can be established between any two processes regardless of where they are physically located, or whether the channels are routed through a network. Thus, programs can be independent of network topology.

In order that packets which are parts of different messages can be distinguished by the VCP of the transputer which receives them, each received packet contains one or two bytes which identify a virtual input channel of the receiving transputer. When a packet is transmitted it may also contain information to route the packet through a packet switching network. The combination of any routing information and the identification of the virtual input channel of the receiving transputer is called the packet header. Every packet of a

message ends with an end–of–packet (EOP) token, except the last packet which ends with an end–of–message (EOM) token.

The maximum length of data in each packet is 32 bytes. All but the last packet of a message contain the maximum amount of data; the last contains the maximum amount of data or less. Each packet has the structure illustrated in figure 4.1. The header bytes (containing routing and channel information) are transmitted first, followed by the data bytes of the packet (if any), followed by the encoded end of packet marker.

header	data bytes	end of packet
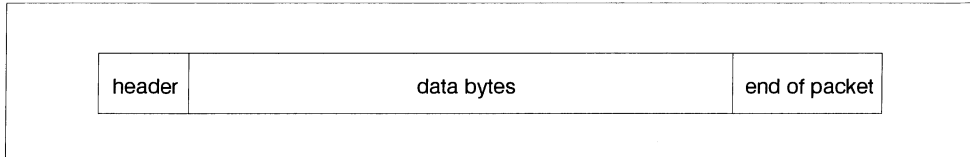
Figure 4.1   Structure of a packet

The VCP distinguishes three types of packet, depending on whether or not there are any bytes of data in the packet, and whether it is terminated with an EOP or an EOM. If the packet is terminated with an EOM token it is the last, possibly the only, packet of a message. If it contains data and is terminated with an EOP token it is part of a message. If it contains no data and is terminated with an EOP token it is taken as the acknowledgement of a previously transmitted packet.

The VCP enforces a high–level protocol on each virtual channel. Each packet of data sent along a virtual channel must be acknowledged before the next is sent to ensure that no data is lost. The last packet must be acknowledged before the outputting process is rescheduled to ensure synchronized communication. Data packets on a virtual channel are acknowledged by the VCP by sending acknowledge packets on another virtual channel back to the VCP which sent them. This acknowledgement is process–to–process (processor–to–processor) and is transparent to intermediate network components.

Virtual channels always occur in pairs between pairs of communicating processors, with one virtual channel in each direction. If a message is being communicated in one direction the virtual channel in the opposite direction is used to return acknowledge packets to the sender. The associated pair of virtual channels is referred to as a *virtual link*. A virtual link can transfer messages in both directions at the same time with data packets and acknowledge packets being interleaved on both of the virtual channels. Because virtual channels are always paired in this way it is not necessary to include source information in the packets. Thus packet headers need only represent their destinations.

Each end of a virtual link is represented by a data structure called a *virtual link control block* (VLCB). A number of instructions are provided on the IMS T9000 for manipulating these data structures. These instructions may be used to establish the links, dynamically alter the connections, activate, deactivate and reset the channels, place channels into resource mode and debug parallel programs.

The VCP of a transputer will send the first packet of a message on a virtual channel to another transputer after the CPU performs an output (*output, outbyte, outword* or *variable output*) instruction. When the VCP of the second transputer receives the packet, it identifies the virtual channel on which the packet was received from the packet header. If a process on the second transputer has performed an input instruction on the channel, the data contained in the packet is stored in the data space of the inputting process. If there is no process ready to receive the first packet then it is placed in an in–store packet buffer associated with the virtual link, which is large enough to hold the 32–byte maximum data length. When the inputting process becomes ready the first packet is copied from the buffer into the data space of the process and an acknowledge packet is sent. This buffering is transparent to the processes because it is never in use when the processes are active. It enables short messages (not longer than 32 bytes) to be sent with only one packet.

In order that the data contained in a buffer is not overwritten, the VCP of a transputer which has sent one packet of a message on a virtual channel to another transputer does not send another packet on that channel until it receives an acknowledgment that a process on the second transputer has become ready to receive the message. When this happens the first packet is copied from the buffer to the data space of the process.

4.3   Virtual link control blocks

Associated with each virtual link are two virtual link control blocks (VLCB's), one in the memory of each transputer connected by the link. These blocks store information to control the operation of the virtual link. Each VLCB is 8 words long and aligned on an 8–word boundary. Table 4.1 shows the information stored in each VLCB. In addition to the 8 word VLCB is a pair of words for resource channels.

VLCB Location	Function	Initialise to:
DataQueueLink	Link to next VLCB with a data packet to send	zero
AckQueueLink	Link to next VLCB with an acknowledge packet to send	zero
OutputWdesc	Workspace descriptor of outputting process	*NotProcess.p*
InputWdesc	Workspace descriptor of inputting process	*NotProcess.p*
OutputLimit	Limiting address from which data may be sent	*NotPointer.p*
InputLimit	Limiting address to which data may be written	*BufferEmpty.p*
HeaderCtrl	Header and control word	*NotHeader.p*
BufferPointer	Pointer to the input packet buffer	*NotPointer.p*

Table 4.1   Content of the VLCB

The physical links are shared by a number of virtual links by threading the control blocks, of the virtual links waiting to use the links, on linked lists. Since each channel of the virtual link may carry both packets of data and acknowledge packets there may be a packet of data and/or an empty acknowledge packet to be sent on a virtual link. Thus the control block contains two queue pointers for threading onto the lists; the **DataQueueLink** and **AckQueueLink** locations. These locations must be initialized to zero.

OutputWdesc and **InputWdesc** store the workspace descriptors of the processes (if any) sending and/or receiving messages on the virtual link. These workspace descriptors must be initialized to *NotProcess.p*.

HeaderCtrl contains a number of bits of control information, and either the header to be included with each packet sent on that virtual link, or the length of the header and an offset to the location in memory where it may be found. Table 4.2 shows the bit fields stored in the most significant byte of the **HeaderCtrl** word.

Bit field	Function		
Header type (2 bits)	0	bytes 0,1 are an offset, byte 2 is a length	
	1	byte 0 is the header	
	2	bytes 0,1 are the header	
	3	bytes 0,1,2 are the header	
Link number (2 bits)	The physical link used by this virtual link		
Input normal	The virtual input channel is operating normally		
Output normal	The virtual output channel is operating normally		

Table 4.2   Bit fields in the most significant byte of **HeaderCtrl**

Headers up to 3 bytes long are held in the VLCB; longer headers are held in a special region of memory. The encoding of short headers within the **HeaderCtrl** word saves a memory access on every packet sent.

4.3.1   Errors

The links can detect disconnection and parity errors. The VCP can detect the following non–attributable errors (errors which cannot be attributed to a particular process): Invalid header; Short non–terminal packet; and Oversize packet. All of these errors cause a serious error to be signalled.

The VCP can also detect a length overrun on an *input*. This is dealt with by recording an invalid message length (–1) in the workspace of the inputting process. The process must recognise and handle the error after it has been rescheduled, which it can do with the *ldcnt* instruction.

4.4   VCP and CPU configuration registers

The VCP and CPU (in common with a number of other sub–systems of the IMS T9000) are controlled via registers in a configuration space. The registers are accessed via the *ldconf* and *stconf* instructions, or via *CPeek* and *CPoke* command messages received along control link **CLink0**. This section describes the functionality of the VCP and CPU to be controlled by bit fields in the associated configuration registers. It also defines the relationship between the addressing of channels and the addressing of store. The complete bit format of each register and the addresses of the registers in the configuration space are **not** included in this preliminary information.

A channel address is the value used to access a channel in a communication instruction. The IMS T9000 channel address space is shown in figure 4.2. The IMS T9000 memory map is shown in figure 4.3.

Event channels are always accessed via channel addresses. The physical links are normally accessed via virtual channels. However, each IMS T9000 physical link may be set to operate in *byte mode* for use in mixed transputer systems (see chapter 14). They are then accessed via the hard channel addresses.

Address		Channel address
	Internal channels	
MemStart		
	Illegal	
MinInvalidChan		
	Virtual channels	
MinInt + 16		#80000040
	Events	
MinInt + 8		#80000020
MinInt + 4	Hard channel inputs	#80000010
MinInt	Hard channel outputs	#80000000

Figure 4.2   IMS T9000 channel address space

4.4.1   MemStart register

The communications instructions operate by treating all channel addresses at or above the **MemStart** register as being internal channel communications – that is between processes executing on the same processor. All channel communications below this address are transferred to the VCP, after checking for illegal addresses.

The *ldmemstartval* instruction can be used to obtain the value of **MemStart**.

Packet buffers may be allocated below **MemStart** in the memory map.

4.4.2   Minimum invalid virtual channel register

There is a range of channel addresses below **MemStart** which do not correspond to valid virtual channels, and which will normally contain virtual link control blocks and headers. The first channel address which corresponds to an invalid virtual channel is held in **MinInvalidChan**.
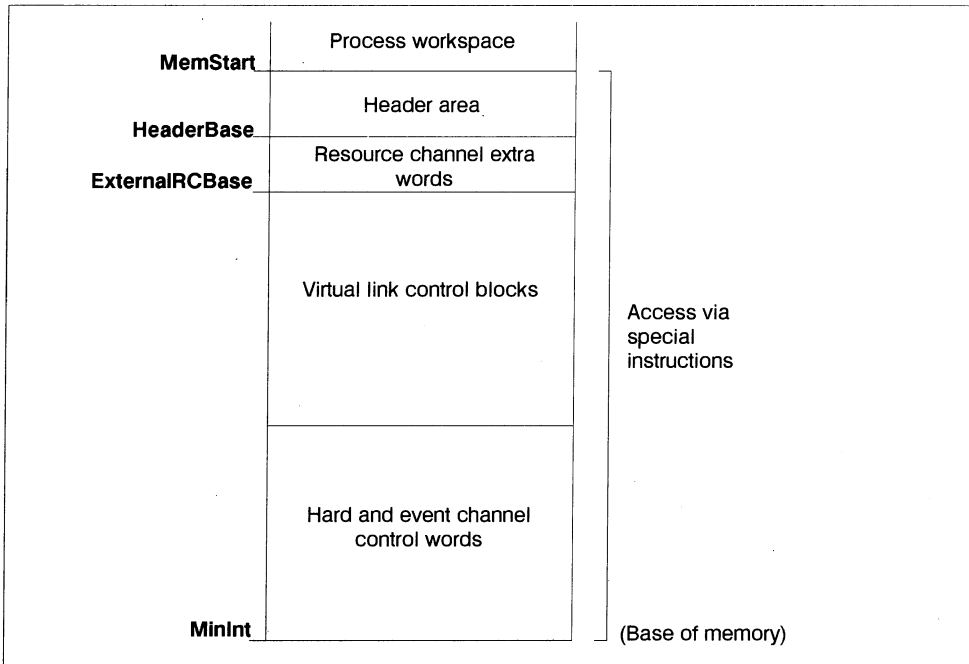
Figure 4.3   IMS T9000 memory map

4.4.3   External resource channel base register

A resource channel is a channel which may be in *normal mode* or *resource channel mode*, plus a two word data structure (see section 4.6 on resources). For local users the extra two words are contiguous with the word used as the channel. For remote users an extra two words are associated with each input virtual channel and Event input. These extra words are allocated together in a block, and the base of the block is defined by the **ExternalRCBase** register.

4.4.4   Header area base register

If the header associated with a virtual channel is longer than three bytes, it is not held in the VLCB associated with that channel, but resides in a separate region of store. The base of this region is defined by the **HeaderBase** register.

4.4.5   Header offset register

The VCP must convert the channel address and header number to the memory address of the VLCB.

The **HeaderOffset0–3** registers are each programmed with an offset which is subtracted from the value contained in the header of a packet which has been input on the associated physical link. The address of the virtual link control block to which a packet is directed is calculated by the VCP hardware using the following formula:

Memory address = vlink.base + ((Header − HeaderOffset) < < vlink.shift)

where for the IMS T9000: vlink.base = **MinInt** + 64, and vlink.shift = 5.

Figure 4.4 shows the mapping of channel addresses and header numbers to the memory address of the VLCB. The example given shows 3 virtual links (6 virtual channels) using 2 words for long headers.
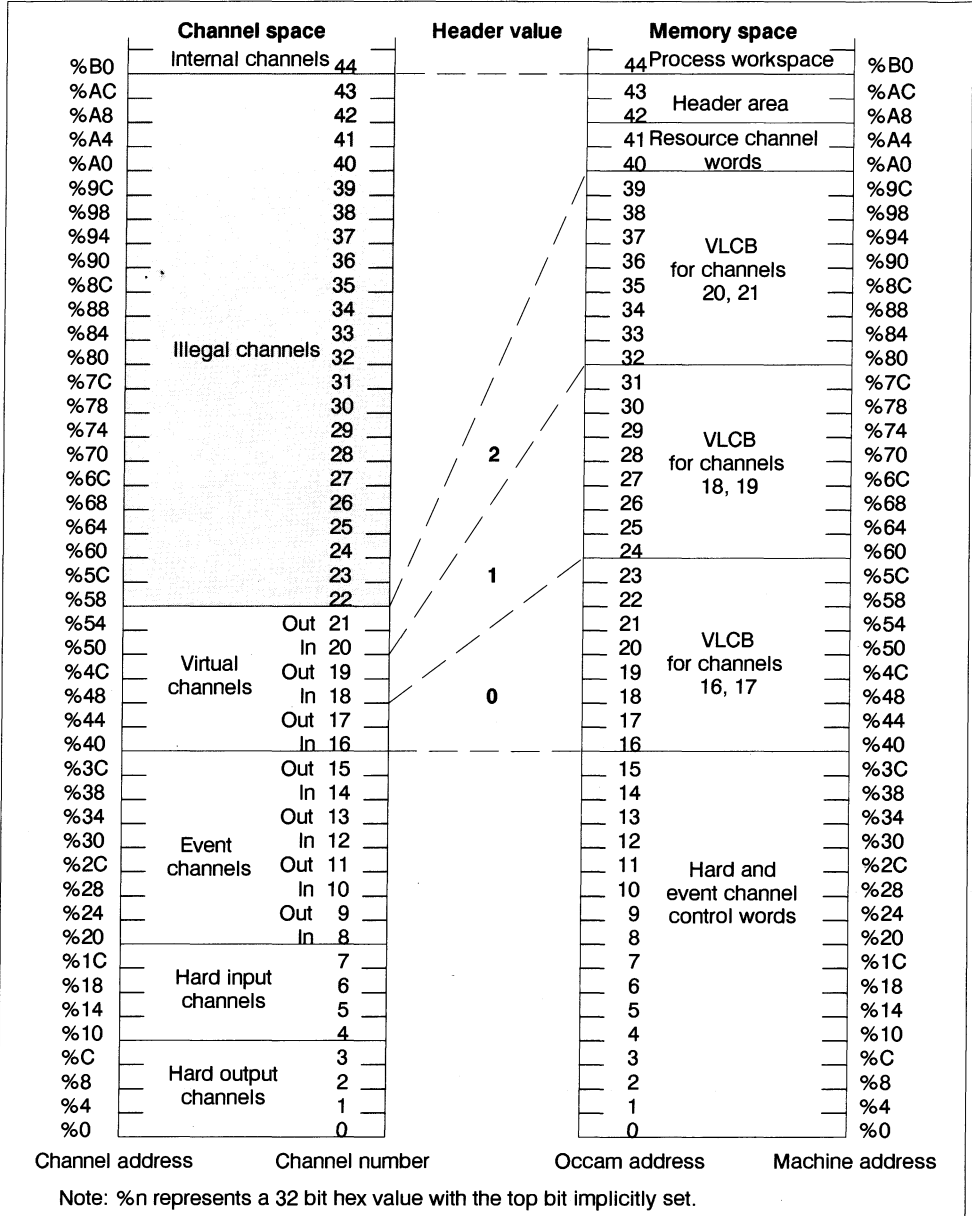
Figure 4.4   Mapping of channel addresses and header numbers to the memory address of the VLCB

4.4.6   Packet header limit registers

The base and limit of packet headers which are acceptable on each physical link are stored in the **Header-Lower0-3** and **HeaderUpper0-3** registers. Out of range headers cause the associated packets to be discarded and, unless the **MaskError** flag (see section 4.4.9) is set, will generate errors. These registers can be used for enhanced system security.

4.4.7   VCP command register

The **VCPCommand** register enables commands to be issued to the VCP. Each bit of the register corresponds to a command, see table 4.3 below. The command is executed when the bit is set. Each write to the register can set only one bit.

Bit	Bit field	Function
0	**Start**	Start the VCP
1	**Stop**	Stop the VCP 'cleanly' so that channel states are preserved. The VCP accepts messages currently in transit but no new messages can be sent.
2	**Reset**	Reset the VCP – stops the VCP and resets the registers to their undefined level 2 state.

Table 4.3   Bit fields in the **VCPCommand** register

4.4.8   VCP status register

The **VCPStatus** register contains information following the occurrence of an error on an input packet (see section 4.3.1 for the types of errors that can occur). Once an error is flagged the packet body is discarded and the following information is returned to the **VCPStatus** register; the header of the packet, the error code, and the link number on which the packet was input. Any subsequent errors are not recorded.

Writes to this register clear the contents regardless of the value written.

4.4.9   VCP link mode register

The **VCPLink0-3Mode** register contains information about the links **Link0-3**.

Bit	Bit field	Function
0	**ByteMode**	Sets the links **Link0-3** to operate in byte mode (see section 14.1).
1	**MaskError**	Masks the error flag for **Link0-3**.
2	**HeaderLength**	Programs the expected length of the incoming packet header (1 or 2 bytes) for each physical link **Link0-3**.

Table 4.4   Bit fields in the **VCPLink0-3Mode** registers

4.4.10   Event mode register

The **EventMode** register contains 4 bits which specify for each event channel **Event0-3** whether it is an input or an output channel (see section 4.5).

4.5   Events

The **EventIn0-3** and **EventOut0-3** pins provide an asynchronous handshake interface between external events and internal processes. Event channels provide process synchronization but cannot transfer any data. Each pair of **EventIn** and **EventOut** pins can act either an input or an output event channel, but not both. This is specified in the **EventMode** register and the 4 pairs of event channels **Event0-3** may be set independently of each other.

Input event channel

When an external event takes an **EventIn** pin high the associated external event channel is made ready to communicate with a process. When both the event channel and the process are ready the processor takes the associated **EventOut** pin high and the process, if waiting, is scheduled. **EventOut** is removed after **EventIn** goes low.

Output event channel

The IMS T9000 asserts the **EventOut** pin to instruct external hardware to perform an action. When both the event channel and the external hardware are ready the external hardware asserts the associated **EventIn** pin and responds to the instruction. **EventIn** should be removed after **EventOut** goes low.

Only one process may use each event channel at any given time. If no process requires an event to occur **EventOut** will never be taken high. Although an **EventIn** triggers the channel on a transition from low to high, it must not be removed before **EventOut** is high. All **EventIn** pins should be low during **Reset**; if not they will be ignored until **Reset** has gone low and returned high. **EventOut** is taken low when **Reset** occurs or when a *resetch* instruction is executed on that channel.
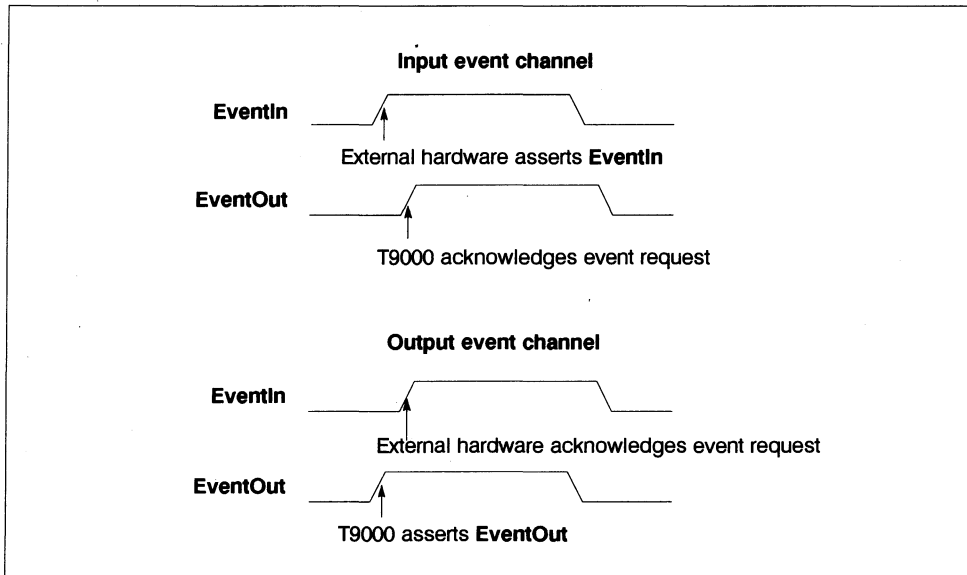
Figure 4.5   Event

4.6   Resources

The IMS T9000 supports the efficient implementation of shared resources. This is done by enabling a number of (user) processes to communicate with a single server process via resource channels. Access to the server process is controlled by a resource data structure which resides on the same transputer as the server process. The resource data structure consists of three words in memory. The word locations in the data structure are shown in table 4.5, and must be initialized with all three words set to *NotProcess.p*. The queue contains resource channels (local and/or remote) on which processes waiting to access the server have performed an output. The flag word contains the identity of a server if one is waiting to service users' requests.

Location name	Purpose
RCqf	Front of queue
RCqb	Back of queue
RCc	Flag

Table 4.5   Contents of a resource data structure

The user processes are connected to the server process via resource channels. A resource channel between a process and a server situated on the same processor is implemented by a three word data structure in memory. A resource channel between a process and a server on a remote machine is implemented by a virtual channel. The user processes communicate to the server process by executing normal output (*output, outbyte, outword* and *variable output*) instructions. The server process is connected to a user process by means of the *grant* instruction; once connected the server process can communicate with the user process by means of input instructions.

The *enable grant* and *disable grant* instructions enable resources to be used in alternative constructs.

5   Memory management

The memory management mechanism in the IMS T9000 transputer is designed to support the development and debugging of programs, to allow the safe execution of programs written in insecure languages, and to support address translation. It also supports the dynamic extension of a calling stack. The mechanism does not support page–based virtual memory.

The memory management mechanism is only invoked for a special type of protected process – known as a P-process. A P-process is run under the control of an ordinary parent transputer process, known as the *supervisor* (or sometimes the *stub*). A P-process is created by the supervisor process executing a *go protected* instruction. This instruction loads the state of the P-process from memory, loads the memory management registers, and starts to execute the P-process.

The P-process may execute only a subset of the IMS T9000 instruction set. The addresses of all memory accesses generated by the P-process are treated as logical addresses; they are checked and translated into physical addresses by hardware. If the P-process attempts to access an illegal address, execute a privileged instruction, or causes an error, control is returned to the supervisor process. Control will also be returned to the supervisor process if the P-process exceeds its timeslice or executes a *system call* instruction. When a trap occurs the P-process's state is saved to memory and the supervisor process is restarted.

5.1   Protection, stack extension, and logical to physical address translation

The memory management mechanism in the IMS T9000 provides for the checking and translation of four independently sized regions of addresses. The P-process may read from any region, but may only write to or execute code out of regions which have the appropriate permissions. All read, write and instruction fetch accesses attempted by the executing P-process are checked. If an illegal access is attempted then the P-process traps back to the supervisor process. It is not normally possible to continue execution of a P-process after an illegal access has been attempted.

In addition to checking the validity of memory accesses, the hardware checks that the location pointed to by the workspace pointer (**WPtr**) is writable. If a *call*, *ajw* or *gajw* instruction causes the workspace pointer to address a non–writeable address then the P-process traps. However, in this case, the supervisor process can restart execution of the P-process after extending the region. In this way it is possible to execute stack extension on demand.

A region may be of size 2^n bytes, with a minimum size of 256 bytes (64 words) and a maximum size of 2^{30} bytes. A region of size 2^n bytes may be translated onto any 2^n byte boundary in the physical address space. The physical addresses associated with the four regions must not overlap. The legal logical addresses within a region either occupy the top 2^n addresses within that region or occupy the bottom 2^n addresses within that region. A consequence of this is that, except for when the maximal sized region (2^{30} bytes) is in use, it is possible to ensure that the addresses 0 and #80000000, which are commonly used as null pointers, do not correspond to legal addresses and so access to such an address is immediately detected as a violation.

5.2   Regions

The logical address space of a P-process is divided into four regions. Each region is sized, assigned access permissions, and has its address accesses translated independently of the others. The two most significant bits of a logical address are used to determine to which region reference is being made. The terms *region 0*, *region 1*, *region 2*, and *region 3* are used to refer to the regions having addresses with the most significant bits set to 00, 01, 10 and 11 respectively.

The legal logical addresses within a region either occupy the top 2^n addresses within that region or occupy the bottom 2^n addresses within that region. The following table shows the legal addresses within each region. The memory mapping for the logical addresses is illustrated in figure 5.1 and table 5.1.
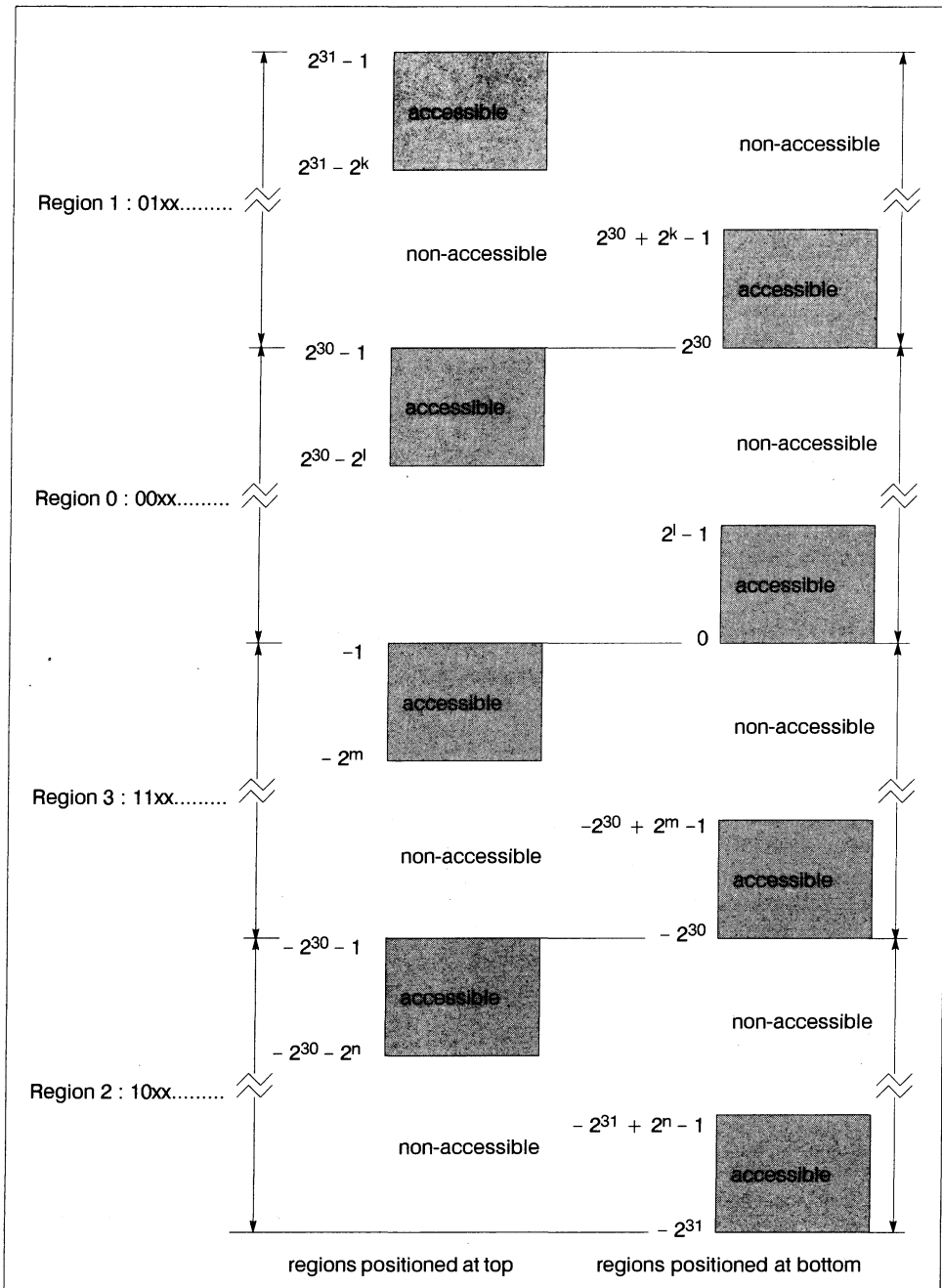
Figure 5.1 Position of region addresses in logical memory space

Region	Size	Positioned from top of region		Positioned from bottom of region	
		Most positive address	Most negative address	Most positive address	Most negative address
0	2^l	$2^{30}-1$	$2^{30}-2^l$	2^l-1	0
1	2^k	$2^{31}-1$	$2^{31}-2^k$	$2^{30}+2^k-1$	2^{30}
2	2^n	$-2^{30}-1$	$-2^{30}-2^n$	$-2^{31}+2^n-1$	2^{31}
3	2^m	-1	-2^m	$-2^{30}+2^m-1$	-2^{30}

Table 5.1   Region addresses

5.3   Region descriptors

A region descriptor defines the size of a region, the position of the logical region, the address translation, and the write and instruction fetch permissions (write-permit and execute-permit respectively) associated with that region.

A region descriptor is a single word. Bit 0 indicates whether writes may be made to the region (1 = write-permit). Bit 1 indicates whether instructions may be fetched from the region (1 = execute-permit). Bit 2 indicates the position of the logical region (1 = top, 0 = bottom). The remaining bits specify the size of the region and the address of the physical region to which the logical region should be relocated. For a region of size 2^n bytes, bit $n-1$ is set to 1. All bits below bit $n-1$ are set to 0 (except for the write-permit, execute-permit and position bits; bits 0, 1 and 2). The remaining high-order bits, bits 31 through n, are used to replace the corresponding bits in the logical address which is being translated.

Note that the minimum region size of 256 bytes implies that bits 2 through 6 of the region descriptor **must** be set to 0.

A region can be set to have zero size by programming its region descriptor with the null descriptor, #8000000. A number of other invalid region descriptors exist, and these should not be used.

An example of a logical to physical address translation which is positioned at the top of region 2 is shown in the following diagram. This region has execute permission but is read-only.
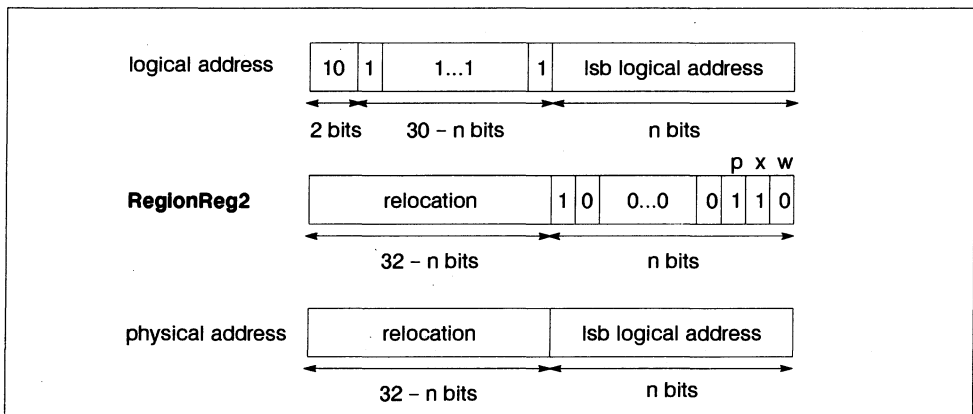
Figure 5.2   Logical to physical address translation

Note that for the logical address to be valid, bits n through to 29 must be 1's if the position bit (bit 2) in the region descriptor is set to 1, and must be 0's if the position bit is 0.

5.4   Machine registers

The IMS T9000 transputer has the following registers related to the operation of memory management for P-processes:

Register	Description
RegionReg0	register descriptor for region 0
RegionReg1	register descriptor for region 1
RegionReg2	register descriptor for region 2
RegionReg3	register descriptor for region 3
PstateReg	pointer to the P-process state vector
WdescStubReg	process descriptor of the supervisor

Table 5.2   Memory management registers

The **RegionRegX** register contains the region descriptor for region X. As described above, the region descriptor defines the size, position, physical address and permissions of a region as a single 32–bit word.

The **PstateReg** register contains a pointer to a block of memory where the state of the executing P-process is to be saved when it traps to its supervisor process.

The **WdescStubReg** register contains the workspace descriptor of the supervisor process which is controlling the execution of the current P-process.

5.5   Debugging

The support provided for debugging a running P-process is an extension of that provided for L-processes, which is described in section 3.4.2. However, whenever the trap-handler would have been invoked for an L-process, for a P-process control is returned to its supervisor process. The supervisor process is responsible for taking any necessary action. Thus, the following debug operations will cause control to be returned to the supervisor process.

- a *j0* instruction acting as a breakpoint

- execution of a single instruction when single-stepping enabled

- a write access to the watchpoint region

6   Main Cache

The IMS T9000 has a 16 Kbyte associative unified write-back (also known as copy-back) cache. That is, memory writes update the cache (if applicable) without necessarily updating memory immediately. Updating of memory occurs when the line that has been changed is discarded from the cache, thus main memory changes on every miss not on every write. A random replacement strategy is used. At power-on the cache behaves as 16 Kbytes of internal memory, so that the IMS T9000 may be used with no external memory. During configuration the cache may be programmed to behave as 16 Kbytes of cache, 16 Kbytes of internal RAM, or 8 Kbytes of cache and 8 Kbytes of internal RAM. The cache has a peak bandwidth of 200 Mwords/s.

The cache is arranged as four independent cache banks, each caching a quarter of the address space. The directory search covers all lines in the cache bank selected. Each bank has 256 lines, with 4 words per line and each line having its own fully-associative tag, see figure 6.1.The banks are selected on address bits **MemAdd4-5**. **MemAdd2-3** determine which word in the line is selected.
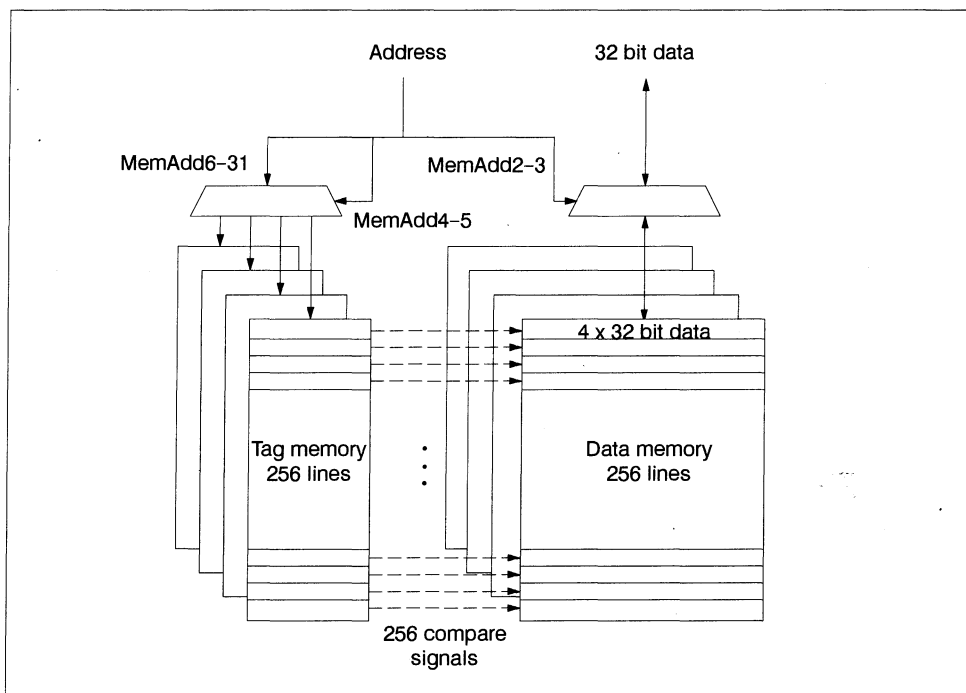
Figure 6.1   Cache operation

An access can be made to each cache bank on every cycle allowing up to four separate accesses to be made to the cache in a single cycle. An arbiter decides which functional unit of the IMS T9000 gains access to each of the cache banks on each cycle. Figure 6.2 shows the major operational units of the cache.

If a physical address requested is missing from the cache, the address is passed to the cache refill engine. The refill engine arbitrates between this and earlier accesses which have misses outstanding. If the missing address is cacheable, the refill engine generates all the addresses needed to refill the associated cache line. The programmable memory interface (PMI) fetches the line requested by the refill engine from external memory. If the address is marked un-cacheable only the missed address is fetched.
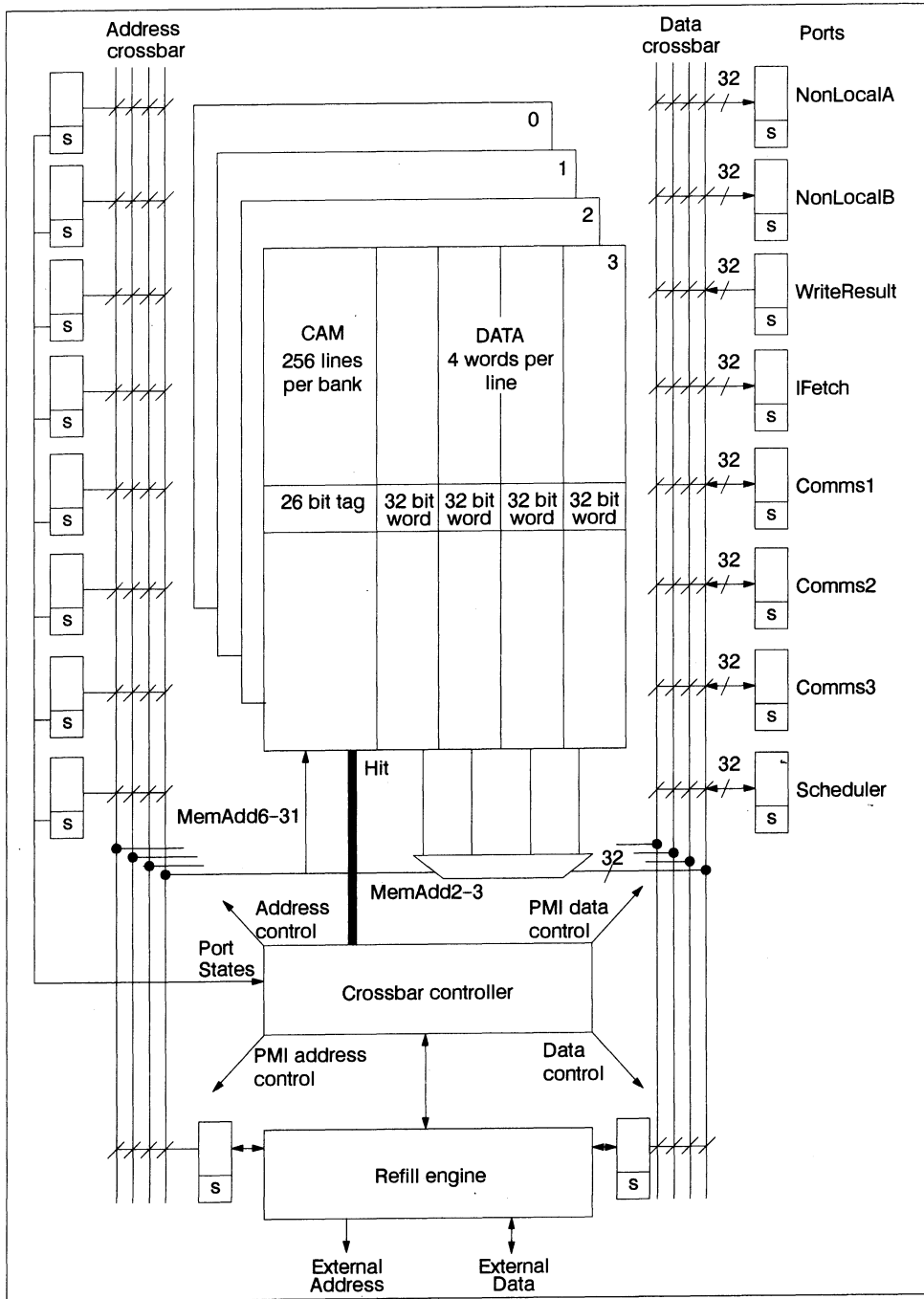
Figure 6.2   Cache block diagram

6.1   Cache instructions

The IMS T9000 provides four instructions to support interfacing the cache to external hardware systems. The flush dirty cache block (*fdcb*) and flush dirty cache line (*fdcl*) instructions write back specified cache lines, and the lines are labeled clean. The invalidate cache block (*icb*) and invalidate cache line (*icl*) instructions invalidate specified cache lines.

6.2   Cache configuration registers

The cache (in common with a number of other sub–systems of the IMS T9000) is controlled via registers in a configuration space. The registers are accessed via the *ldconf* and *stconf* instructions, or via *CPeek* and *CPoke* command messages received along control link **CLink0**. This section describes the functionality of the cache to be controlled by the associated configuration registers. The complete bit format of each register and the addresses of the registers in the configuration space are **not** included in this preliminary information.

The **RAMsize** register defines the amount of RAM which is allocated to be internal RAM. It can be programmed to be 0, 8 or 16 Kbytes. Internal RAM is implemented by locking lines into the cache. The **RAMLineNumber** and **RAMAddress** registers allow the addresses of the locked cache lines to be configured by the user. This enables the RAM to be located anywhere in the processors physical address range.

The **RandomSeed** register allows the random number generators used for cache line replacement to be seeded. A default seed is loaded on reset.

7 Programmable memory interface

The IMS T9000 programmable memory interface (PMI) is designed to support memory subsystems with minimal external support logic. The interface has internal logic to provide decode and timing control functions and can be programmed through the configuration registers as described in section 7.3 below.

The external address space is partitioned into four banks (not to be confused with the four cache banks detailed in chapter 6). This allows the implementation of mixed memory systems, with support for DRAM, SRAM, EPROM and VRAM. The timing of each of the four memory banks can be programmed separately, with a different device type being placed in each bank with no external hardware support. The PMI has a 64 bit data bus, and each bank of memory can be configured to be 8, 16, 32 or 64 bits wide. The PMI directly supports: 8, 16, 32 and 64 bit SRAM; 32 and 64 bit DRAM. All banks programmed to be 64 bit wide memory are defined as cacheable and always transfer a full 64 bit operand to and from external memory to the internal cache, providing fast cache refill. The full performance of the IMS T9000 transputer can be exploited using relatively low-cost DRAM, and up to 8 Mbytes of DRAM can be connected with no external components.

The transputer uses word addressing. 64 bit wide memory is defined as an array of 8 byte words with **MemAdd3-31** selecting an array. No further addressing is performed for 64 bit memory. 32 bit wide memory is defined as an array of 4 byte words with **MemAdd2-31** selecting an array. Each byte of this array is addressable with the byte enable pins **notMemWrB0-3** selecting a byte within the array. 16 bit wide memory is defined as an array of 2 byte words with **MemAdd1-31**selecting an array and **notMemWrB0-1** selecting a byte within the array. 8 bit wide memory is defined as an array of 1 byte words with **MemAdd0-31** selecting an array.

In the following sections a *cycle* is one processor clock cycle and a *phase* is one quarter of the duration of one processor clock cycle.

7.1 Pin functions

7.1.1 ProcClockOut

Output timing signal at rated clock frequency of device.

7.1.2 MemData0-63

The data bus transfers 64, 32, 16 or 8 bit data items depending on the bus width configuration. For 64-bit data items the most significant bit is carried on **MemData63.** For 32 bit data items the most significant bit is **MemData31**. **MemData0-15** transfers 16 bit data items, and **MemData0-7** transfers 8 bit data items.

7.1.3 MemAdd2-31

The address bus may be operated in both multiplexed and non-multiplexed modes. When a bank is configured to contain DRAM, or other multiplexed memory, then the internally generated 32 bit address is multiplexed as row and column addresses through the external address bus. The multiplexing is controlled by the **FormatControl** registers.

7.1.4 notMemWrB0-3

The transputer uses word addressing therefore four byte-write strobes are provided to select one of four bytes addressed by **MemAdd2-31**. For a bank configured to 32, 16 or 8 bits, the lower order address bits are multiplexed onto the unused byte-write pins to give an address bus 30, 31 or 32 bits wide respectively. **notMemWrB0** addresses the least significant byte. All four strobes have the same timing and are only active during write cycles. The timing is controlled by the **WriteStrobe** registers.

The function of the byte enables **notMemWrB0–3** for different bank size configurations is given in table 7.1 below.

	External port size			
	64 bit	**32 bit**	**16 bit**	**8 bit**
notMemWrB3	set active (0)	enables **MemData24–31**	becomes **MemAdd1**	becomes **MemAdd1**
notMemWrB2	set active (0)	enables **MemData16–23**	undefined	becomes **MemAdd0**
notMemWrB1	set active (0)	enables **MemData8–15**	enables **MemData8–15**	undefined
notMemWrB0	set active (0)	enables **MemData0–7**	enables **MemData0–7**	enables **MemData0–7**

Table 7.1   **notMemWrB0–3** pins

7.1.5   notMemRAS0–3

The four programmable RAS strobes are controlled by the **TimingControl** and **RASStrobe** registers. One strobe is allocated to each of the four banks which are decoded on chip. If a bank is programmed to contain DRAM, or other multiplexed memory, then the associated **notMemRAS** pin acts as its RAS strobe by default. For banks which do not contain DRAM the **notMemRAS** pin is available as a general purpose programmable strobe.

7.1.6   notMemCAS0–3

The four programmable CAS strobes are controlled by the **CASStrobe** registers. One strobe is allocated to each of the four banks which are decoded on chip. If a bank is programmed to contain DRAM, or other multiplexed memory, then the associated **notMemCAS** pin acts as its CAS strobe by default. For banks which do not contain DRAM the **notMemCAS** pin is available as a general purpose programmable strobe.

7.1.7   notMemPS0–3

These four additional programmable strobes are controlled by the **ProgStrobe** registers. One strobe is allocated to each of the four banks which are decoded on chip.

7.1.8   MemWait

Wait states can be selected by taking **MemWait** high. **MemWait** is sampled during **RASTime** and **CAS-Time**. **MemWait** retains the state of any strobe during the cycle in which **MemWait** was asserted. **MemWait** suspends the cycle counter and the strobe generation logic until deasserted. When **MemWait** is de-asserted cycles continue as programmed by the configuration registers.

7.1.9   MemReqIn, MemGranted

Direct memory access (DMA) can be requested at any time by driving the asynchronous **MemReqIn** signal high. The address and data buses are tristated after the current memory cycle terminates. If the current memory cycle is part of a cache line write back or fill then the four words of the line are transferred before the buses are tristated.

Strobes are left inactive during the DMA transfer. If a DMA is active for longer than one programmed refresh interval then external logic is responsible for providing refresh.

MemGranted follows the timing of the bus being tristated and can be used to signal to the device request-ing the DMA that it has control of the bus.

Table 7.2 below lists the processor pin state while **MemGranted** is asserted.

Pin Name	Mem Granted State
MemAdd3-31	floating
MemData0-63	floating
notMemWrB0-3	inactive
notMemRAS0-3	inactive
notMemCAS0-3	inactive
notMemPS0-3	inactive
notMemRf	inactive
MemReqOut	active
notMemBootCE	inactive

Table 7.2 **MemGranted** pin states

7.1.10 MemReqOut

The **MemReqOut** pin indicates to external logic that IMS T9000 external bus cycles are pending and execution will stall if a DMA transfer is initiated, or has stalled if a DMA transfer is in progress.

Once a DMA transfer has been granted the IMS T9000 processor can continue to execute out of the internal cache until an access to external memory is required. The **MemReqOut** pin will be taken high and external logic can use this information to interrupt the DMA transfer in progress. The external logic should deassert **MemReqIn** when the memory buses are available for the processor to use.

7.1.11 notMemBootCE

The IMS T9000 has a dedicated area of external memory address space of fixed size and timing. This functions as a fifth bank with fixed decode and timing parameters. This is to provide slow access to configuration/ bootstrap code stored in ROM. **notMemBootCE** is used to access external memory placed in this dedicated address range. This address space can also be used to access code/data which is not bootstrap code if required.

7.1.12 notMemRf

The IMS T9000 can be operated with memory refresh enabled or disabled. The selection is made during memory configuration, when the refresh signal is also determined.

notMemRf indicates that the current cycle is a refresh cycle. It is asserted low at the beginning of the refresh cycle and deasserted high at the end of the refresh cycle.

7.2   External Bus Cycles

The IMS T9000 programmable memory interface is designed to provide efficient support for dynamic memory without compromising support for other devices, such as static memory and I/O devices. This flexibility is provided by allowing the required waveforms to be programmed via the configuration registers described in section 7.3.

Interaction of the PMI with the on-chip cache is highly optimized. In order to support specialized memory types, addresses within 8, 16 or 32 bit memory banks can be specified to be cacheable or non-cacheable Note, 64 bit memory is always defined as cacheable. In addition, each bank can be specified to contain 8/16/32/64 bit wide SRAM, or 32/64 bit wide DRAM memory. The PMI synthesizes the required number of cycles to assemble full words before transferring them to or from the internal cache.

A generic memory interface cycle consists of a number of defined periods, or times, as shown in figure 7.1. This generic memory cycle uses DRAM terminology to clarify the use of the interface in the most complex situations, but can be programmed to provide waveforms for a wide range of other device types. The timing of each of the four memory banks can be programmed separately, with a different device type being placed in each bank with no external hardware support.
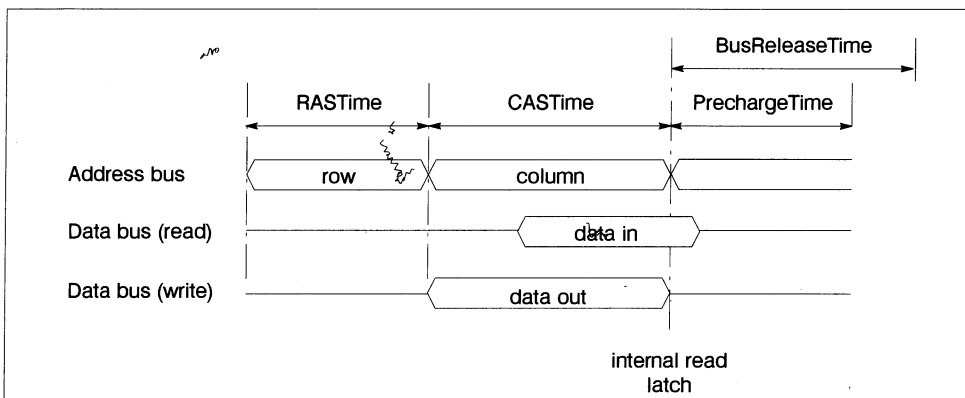
Figure 7.1   Generic memory cycle

The **RASTime** and **CASTime** are consecutive. The **CASTime** is followed by concurrent **Precharge** and **BusRelease** times. Thus, for DRAM, the times are used for RAS, CAS, and precharge respectively. For non-multiplexed addressed memory the **RASTime** is programmed to be zero and there is no RAS time.

If programmed to be non-zero, and page-mode memory is present in a bank, the **RASTime** will only occur if consecutive accesses are not in the same page. The **RASTime** will not commence until the **PrechargeTime** for a previous access to the same bank has completed. During this time the address is multiplexed by the amount specified in the **FormatControl** register so as to output the row address on the address bus. During the **RASTime** a transition can be programmed on the RAS strobe, but not on any other strobe.

During the **CASTime** the programmable strobes and byte-write strobes are active. The address is output on the address bus without being shifted. Write data is valid during **CASTime**. Read data is latched into the interface during the last clock cycle of the **CASTime**.

The **PrechargeTime** and **BusReleaseTime** commence concurrently at the end of the **CASTime**. A **PrechargeTime** will occur to the current bank if:

- the next access is to the same bank but to a different row address.
- the next cycle is to a different bank.

The **BusReleaseTime** runs concurrently with the **PrechargeTime** and will occur if:

- the current cycle is a read and the next cycle is a write.
- the current cycle is a read and the next cycle is a read to a different bank.

The **BusReleaseTime** is provided to allow slow devices to float to a high impedance state.

7.2.1 External DRAM cycles

The IMS T9000 interface has logic to utilize page–mode DRAM. The internal logic determines if page–mode accesses are appropriate and constructs the required waveforms as defined by the **TimingControl** register. For random accesses to dynamic memory the interface will execute a **RASTime**, followed by a **CASTime**, followed by a **PrechargeTime**. Figure 7.2 shows a random access to dynamic memory in bank 0.
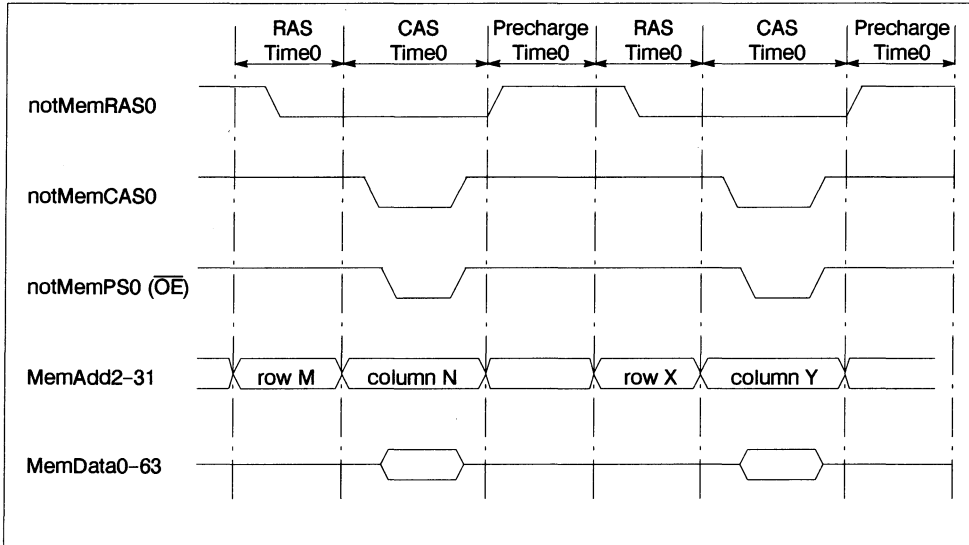
Figure 7.2   Random read access to DRAM from bank 0

For consecutive accesses within the same page in a single bank the row address remains constant and only subsequent column addresses change. To perform a cache line transfer 4 consecutive addresses are transferred, and a **RASTime** sub–cycle is only required for the first transfer across the external data bus. This may be omitted if the previous access to the bank was in the same page. To read a cache line from a 32 bit wide bank of DRAM in bank 2 the PMI will execute a single **RASTime**, followed by four **CASTimes**, followed by a **PrechargeTime**, as shown in figure 7.3.
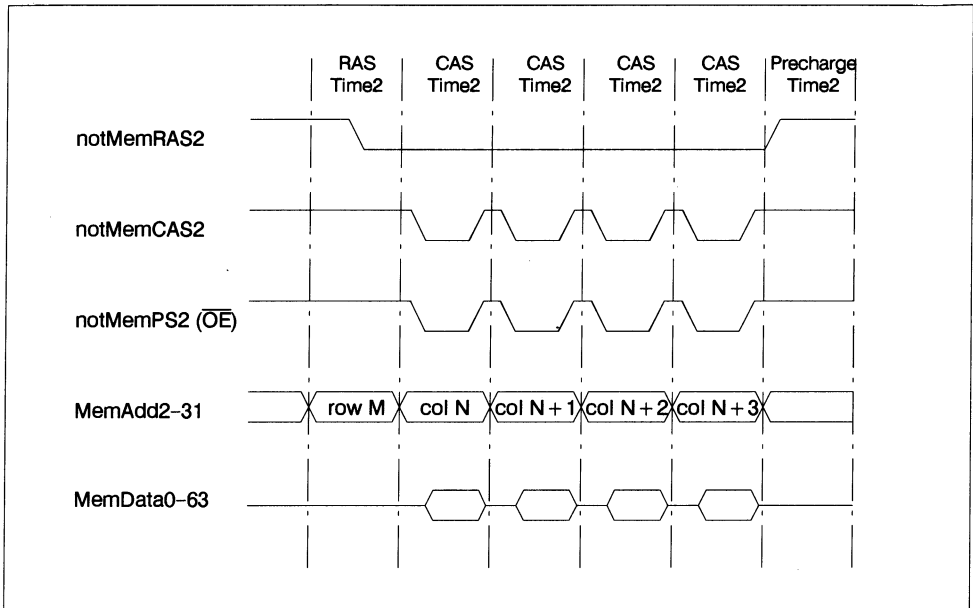
Figure 7.3   Page–mode access to DRAM – 32 bit interface cache refill from bank 2

For a 64 bit wide bank of DRAM the PMI will execute a single **RASTime**, followed by two **CASTime** followed by a **PrechargeTime**, as shown in figure 7.4.
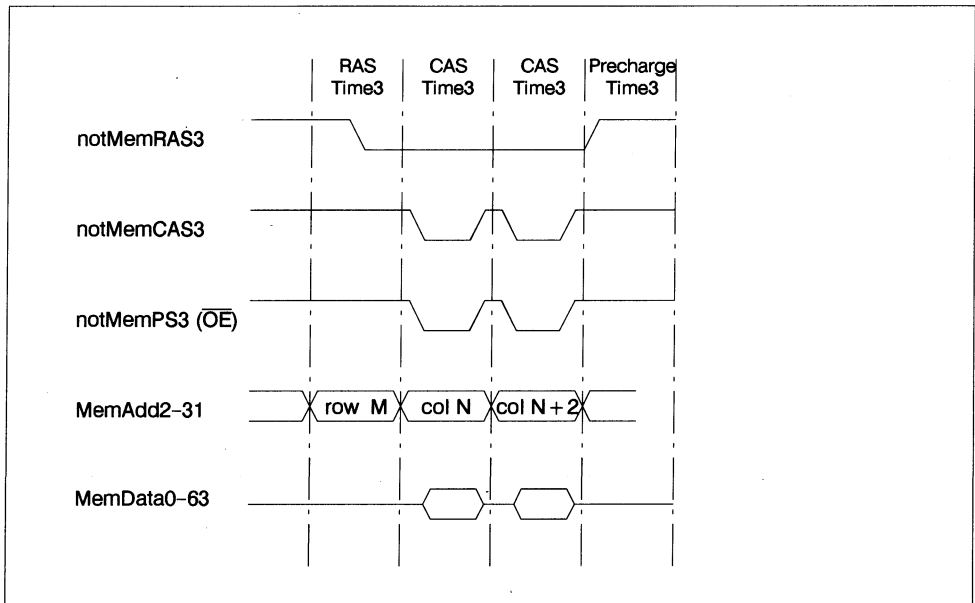
Figure 7.4   Page–mode access to DRAM – 64 bit interface cache refill from bank 3

The IMS T9000 is not limited to performing only cache–line refills in page–mode. As long as the row address remains constant, then the PMI will continually operate in page–mode.

Figure 7.5 shows an extended page mode cycle from DRAM.
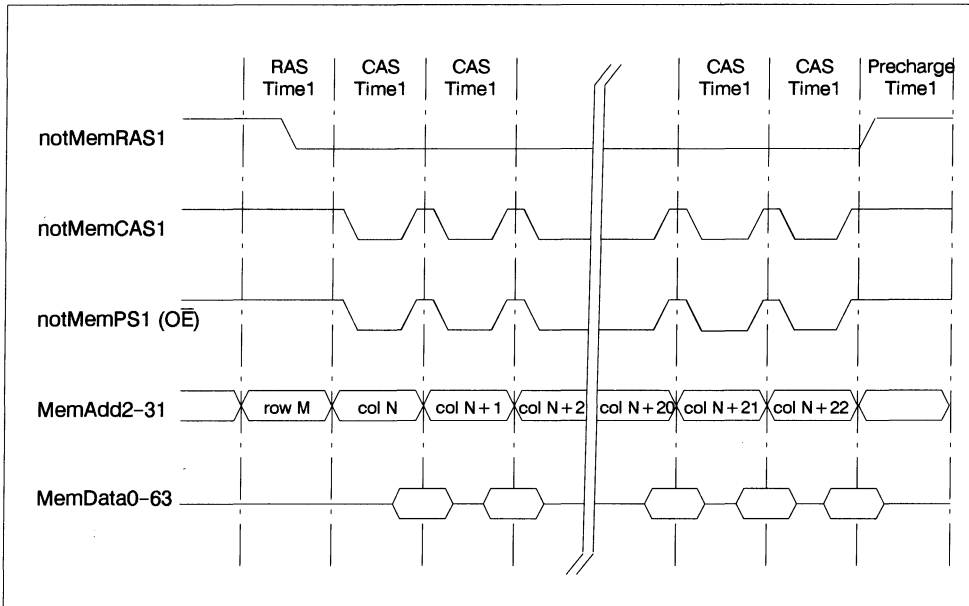
Figure 7.5   Extended page mode access from bank 1

7.2.2   External non–DRAM cycles

The IMS T9000 interface does not explicitly distinguish between a bank which is programmed as dynamic memory and a bank which is not dynamic memory. This is to allow complete flexibility in the use of the strobes and the various timing parameters. The correct mode of access is determined by proper programming of the **TimingControl0–3** register parameters. Some of these parameters are inapplicable to a static memory bank and should be programmed to zero. Static memory cycles can be adequately defined by the **CASTime** parameter. For a cache line read from static memory the **RASTime** is programmed to be zero and no **RAS** sub–cycle occurs. The PMI will execute four **CASTime** cycles for a cache line refill from a 32 bit wide bank of non-DRAM, as shown in figure 7.6.
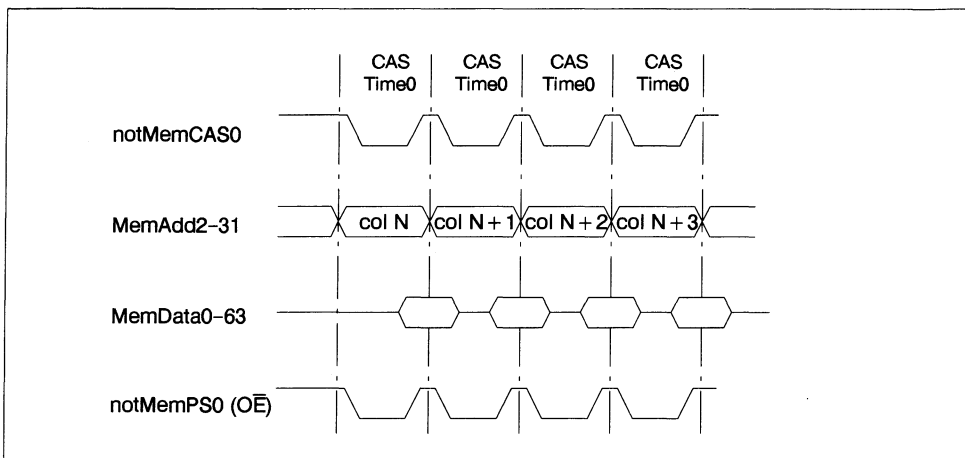
Figure 7.6   32 bit non-DRAM bank 0 cache refill

7.2.3   Bank switching

Precharge Time and **Bus Release Time** allow consecutive cycles to access different banks without the need for any external controlling logic. Figure 7.7 shows switching between SRAM in bank 0 and SRAM in bank1. A **Bus Release Time** is inserted between the two accesses. The CAS, PS and Write strobes are inactive during this time, the RAS strobe is unaffected.
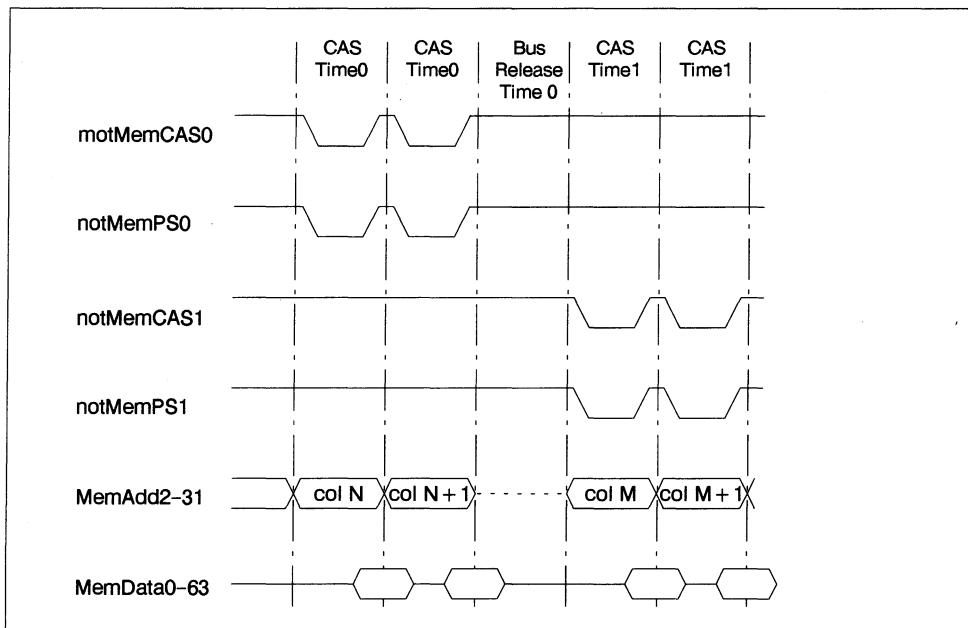
Figure 7.7   SRAM bank 0 to SRAM bank 1 with bus release time

Figure 7.8 shows switching between DRAM in banks 0 and 1. During **PrechargeTime0** the strobes for bank 0 are inactive and the strobes for bank 1 operate as defined by their configuration registers. Access is made to bank 1 whilst bank 0 is precharging. The example shown has the **PrechargeTime** programmed as 2 cycles.
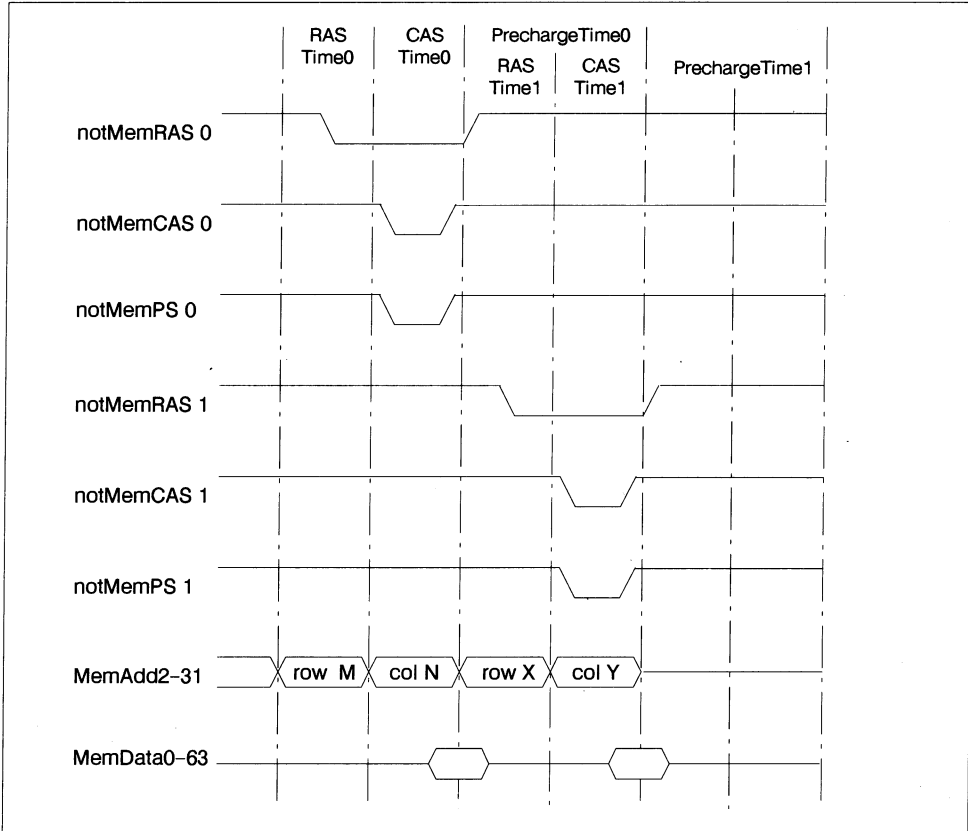
Figure 7.8   DRAM bank 0 to DRAM bank 1 switching, no bus release time

Figure 7.9 shows switching between DRAM in bank 1 and SRAM in bank 2. The example shown has the **PrechargeTime** for bank 1 programmed as 1 cycle and the bank 2 **CASTime** as 2 cycles.
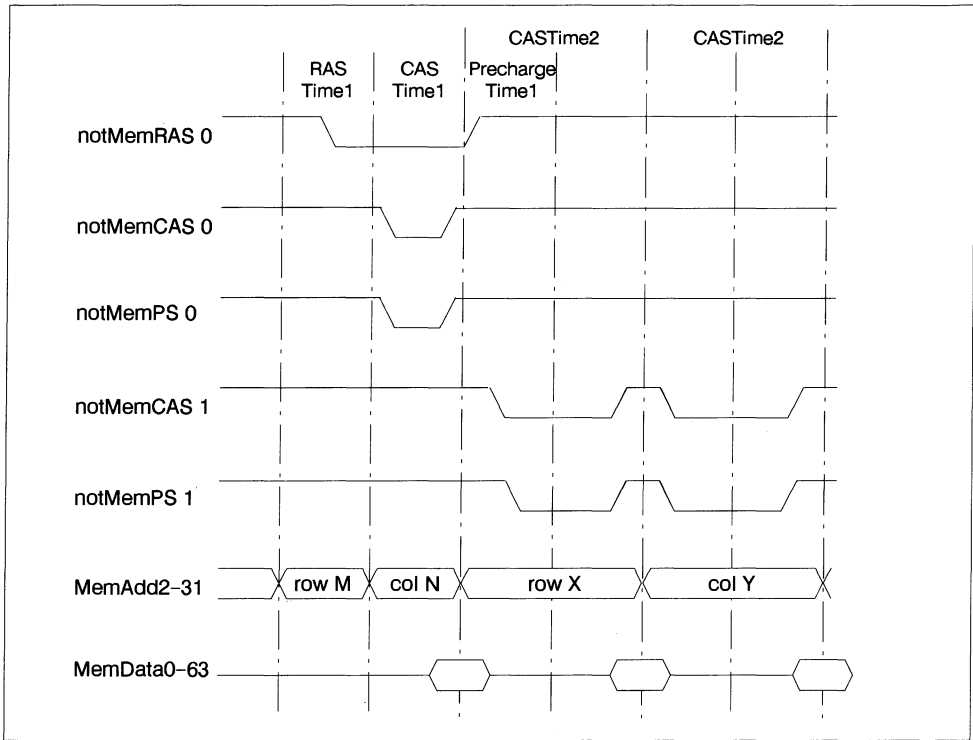
Figure 7.9   DRAM bank 0 to SRAM bank 1 switching, no bus release time

7.3   PMI configuration registers

The PMI (in common with a number of other sub-systems of the IMS T9000) is controlled via a separate configuration address space. The registers in this address space are accessed via the *ldconf* and *stconf* instructions, or via *CPeek* and *CPoke* command messages down **CLink0**. This section describes the functionality of the PMI to be controlled by the associated configuration registers. The complete bit format of each register and the addresses of the registers in the configuration space are **not** included in this preliminary information.

The PMI configuration registers are divided into 2 sets. The *bank address* registers define the structure of the external address space and how it is allocated to the four banks and the *strobe timing* registers define the timing of the strobe edges for the four banks. The function of the registers is to eliminate external decode and timing logic.

7.3.1   Bank address registers

The addresses of operands generated by IMS T9000 internal subsystems are analyzed by the PMI. It uses the values of the configuration registers to establish which bank the address is applicable to and the type of access. The incoming address and the bank address registers are compared. The bits that are not of interest are masked off by the mask address register (**Mask0–3**). This is performed in parallel for all four banks.

Address registers

The **Address0–3** registers define the base address for each of the four banks. The base address must be word aligned.

Mask registers

The **Mask0-3** registers define the bits in the address which should be compared to the address register for the appropriate bank.

> 1 in a given bit position indicates that the corresponding bits should be compared.

> 0 indicates they should be ignored.

If all bits which are to be compared are the same in the presented address and the address register for a bank then the address is a hit on the bank.

RAS bits registers

The **RASBits0-3** registers define the bits in the address which should be compared to the last access to the same bank to determine whether a page hit has occurred. The register contents are only used if the **RAS-Time** in the **TimingControl** register is programmed to be non-zero.

> 1 in a given bit position indicates that the corresponding bits should be compared.

> 0 indicates they should be ignored.

If all bits which are to be compared are the same in the presented address as in the previous access, then the address is a page hit and a **RAS** cycle will not be generated.

Format control registers

The **FormatControl0-3** registers control general aspects of operation of each bank of the PMI.

Bit field	Function	Units
ShiftAmount	Right shift for the on-chip multiplexing for the bank	–
PortSize	Bit width of the bank (8, 16, 32 or 64 bits)	–
CacheMode	Cacheability status of addresses in the bank	–

Table 7.3   Format control register fields

The **ShiftAmount** is defined as the amount of right shift to be applied to the external address field,for the duration of the **RASTime**.The shift amount is only active during the **RASTime**.

The **PortSize** defines the size of the external port that occupies the selected bank.The coding of the bits are defined in table 7.4. The **PortSize** parameter is used by the byte-alignment network to assemble/disassemble data bytes to transfer arbitrary-sized operands to arbitrary-sized ports.

PortSize	Programmed bus widths
00	64 bits
01	32 bits
10	16 bits
11	8 bits

Table 7.4   **PortSize**

The **CacheMode** bit field defines if the bank is occupied by devices whose contents can be transferred to the IMS T9000 internal cache. Note, any bank which is programmed to have 64 bit memory is defined as a cacheable area.

CacheMode	Cacheability
0	Non-cacheable
1	Cacheable

Table 7.5   **CacheMode**

Figure 7.10 illustrates programming of the configuration registers for given bank configurations. Banks 0 and 1 are shown conataining an external bus width of 32 bits, banks 2 and 3 with an external port size of 64 bits. All the banks have a programmed RASTime, except bank 4 which has RASTime set to zero. In the example shown in the figure the base addresses for each of the four banks are as follows:

Bank	Physical address
0	C3C00000 – C3FFFFFF
1	B0300000 – B03FFFFF
2	50E00000 – 50FFFFFF
3	0E1F8000 –0E1FFFFF
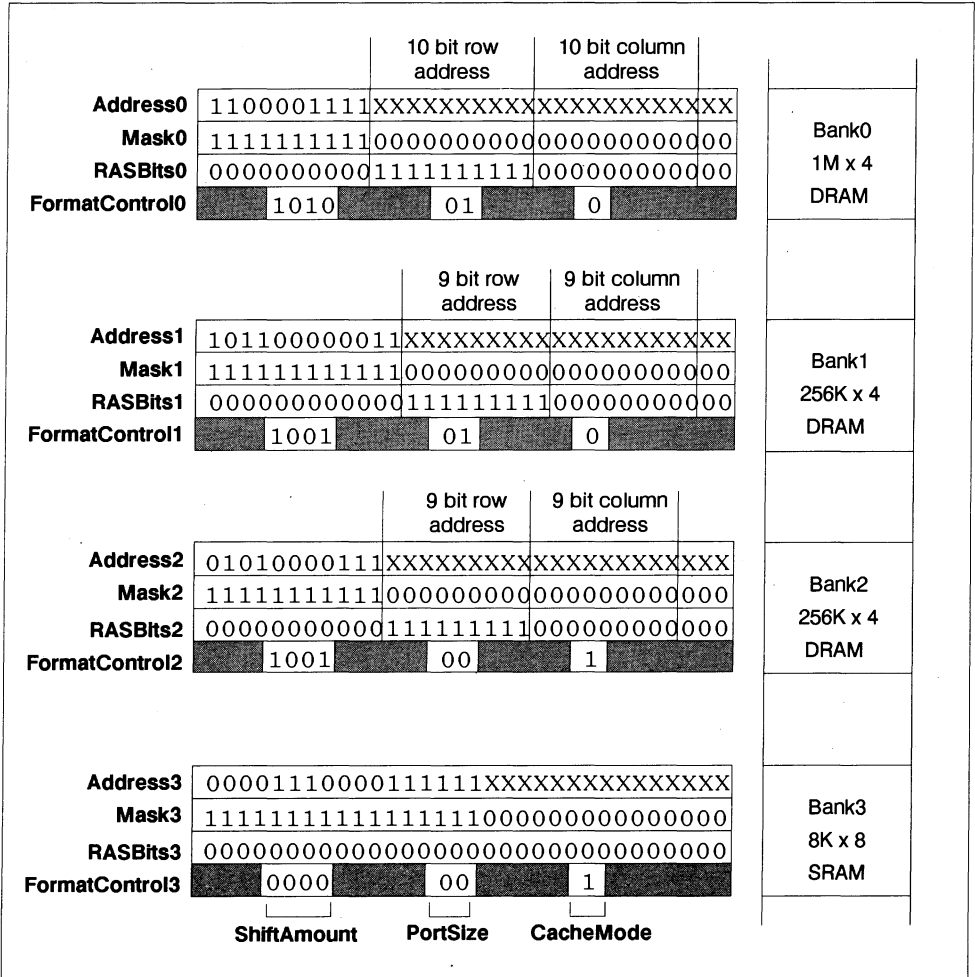
Figure 7.10   Programming page configuration registers

BootSpace allocation

The IMS T9000 includes support for a fifth bank of external memory which is not user programmable. The function of this bank is to provide a configuration/ bootstrap area of external memory. The port size of this bank is hardwired to be a byte wide interface.

7.3.2   Strobe timing registers

The PMI constructs control waveforms with the required timing in the appropriate bank from the contents of the **TimingControl0-3** registers. The internal pipeline structure of the IMS T9000 allows internally pending cycles to be analyzed while the bus is currently in use. The bus control logic can construct the required timing and control waveforms from information about the current bus cycle and the next pending cycle.

Strobe registers

The **RASStrobe0-3** registers, **CASStrobe0-3** registers, **ProgStrobe0-3** registers and the **WriteStrobe0-3** registers all have a common format, as given in table 7.6. The falling (E1) and rising (E2) edges of a waveform are defined to occur during the **CASTime**. During other sub-cycles the programmable strobe pins are held in the inactive state.

Bit field	Function	Units
E1Time	Location of falling edge from **CASTime** start	phases
E2Time	Location of rising edge from **CASTime** start	phases
ActiveCode	Cycle type in which strobe is active	–

Table 7.6   Strobe register fields
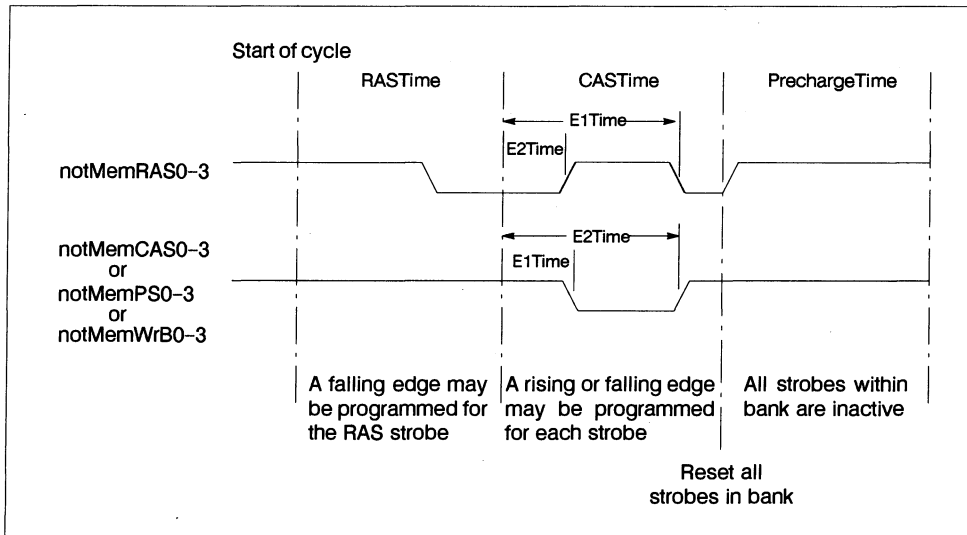
Figure 7.11   Strobe activity within a memory cycle

If DRAM is present in a bank the **E2Time** for the associated **RASStrobe** register is typically programmed to be shorter that the **E1Time** so that the RAS strobe falls during the **RASTime** of the cycle and rises again during the **CASTime** of the cycle.

ActiveCode determines the type of cycle (read or write) during which the strobe will be active. The coding of these bits is indicated below.

ActiveCode	Bus activity
00	Inactive
01	Active during read only
10	Active during write only
11	Active during read and write

Table 7.7   **ActiveCode**

The timing programmed in the **WriteStrobe** register for a bank is used for all four byte–write strobes for writes in that bank.

Timing control registers

The **TimingControl0-3** registers define for each bank timing parameters for the peripheral devices allocated to that memory bank. The parameters defined by the register are shown in table 7.8.

Bit field	Function	Units
RASTime	Duration of RAS sub–cycle	cycles
RASEdgeTime	Delay from start of RAS sub–cycle to falling edge of RAS strobe	phases
CASTime	Duration of CAS sub–cycle	cycles
PrechargeTime	Duration of precharge time	cycles
BusReleaseTime	Duration of bus release time	cycles
WaitEnable	Enables the **MemWait** pin	–

Table 7.8   Timing control register fields

RASTime sets the length of the RAS sub–cycle. If this is programmed to zero then no RAS sub–cycle will occur.

RASEdgeTime sets the delay from the start of the RAS sub–cycle to when the RAS strobe goes low. This is only required if **RASTime** is programmed to be non–zero.

Figure 7.12 gives an example of the programming of the strobe timing registers for a write cycle in bank 1.
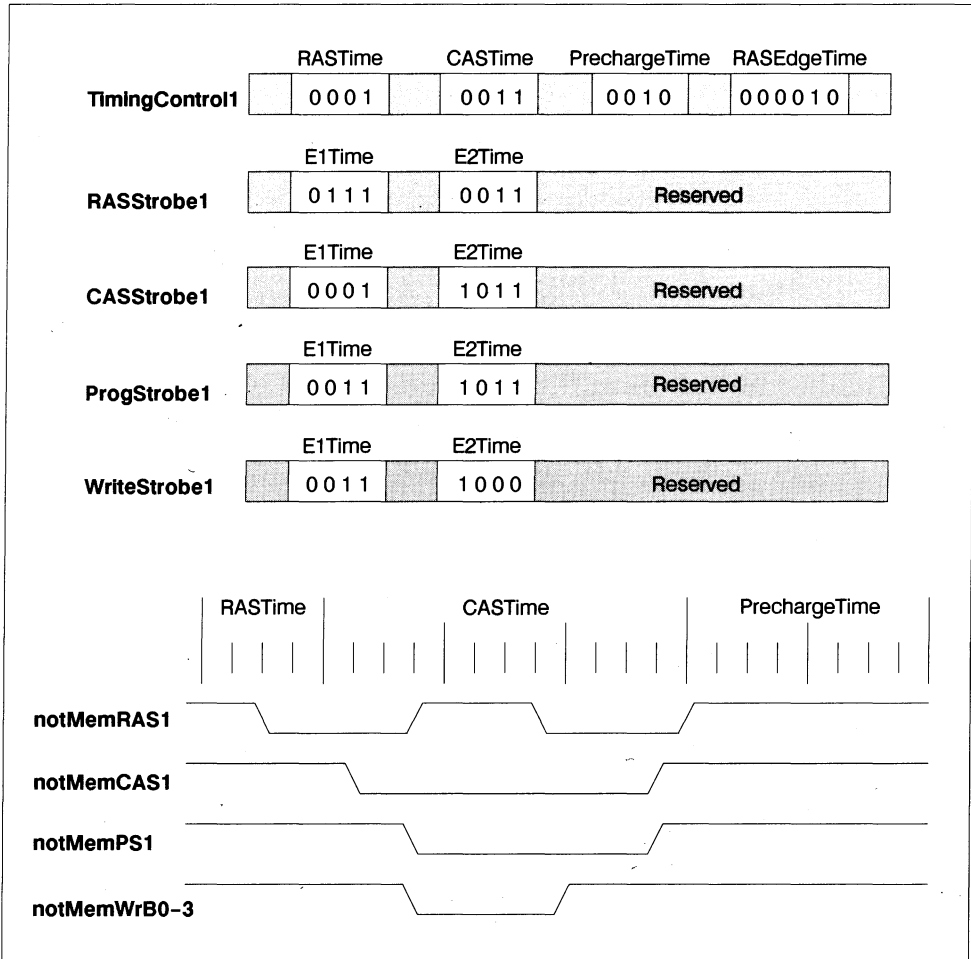
Figure 7.12   Programming of the strobe timing registers for bank 1

Refresh control register

The **RefreshControl** register specifies the banks which require refreshing and the interval between successive refreshes. The refresh timing is also programmed in this register, and is the same for all banks.

The PMI ensures that **CAS** and **RAS** are both high for the required time before every refresh cycle by inserting a **PrechargeTime** in the last bank being accessed and ensuring all **PrechargeTimes** are complete.

The **CASStrobe** is taken low at the beginning of the refresh time. The position of the **RAS** falling edge (**RASedge**) and the time before **RAS** and **CAS** can be taken high again (**RefreshTime**) are programmed. Each of these actions occurs in sequence for each bank. A cycle is inserted between each bank in order to spread current peaks. If no DRAM has been programmed for a bank then no transitions occur on the RAS or CAS strobes. Once all refreshes have occurred a **PrechargeTime** is initiated in all banks and further accesses may occur.

The **RefreshControl** register is loaded during the configuration phase and if the **Refresh Interval** is zero, then no refreshes will take place. The register bit fields are allocated the following functions.

Bit field	Function	Units
RefreshTime	Refresh time	cycles
RASedge	Refresh RAS falling edge	phases
RefreshInterval	Defines DRAM refresh interval	cycles
DRAM0–3	Defines which banks require refresh	–

Table 7.9   Refresh control register fields

Figure 7.13 illustrates programming of the refresh control register. The example shows DRAM programmed in banks 0, 1 and 3, no DRAM programmed in bank 2. After a refresh, a **PrechargeTime** is introduced after each bank has completed in turn. In the example shown the **PrechargeTime** for each bank is programmed (in the strobe timing registers) as 1 cycle. After refresh has completed in bank 3, further accesses may proceed for all banks once any precharge times are complete.
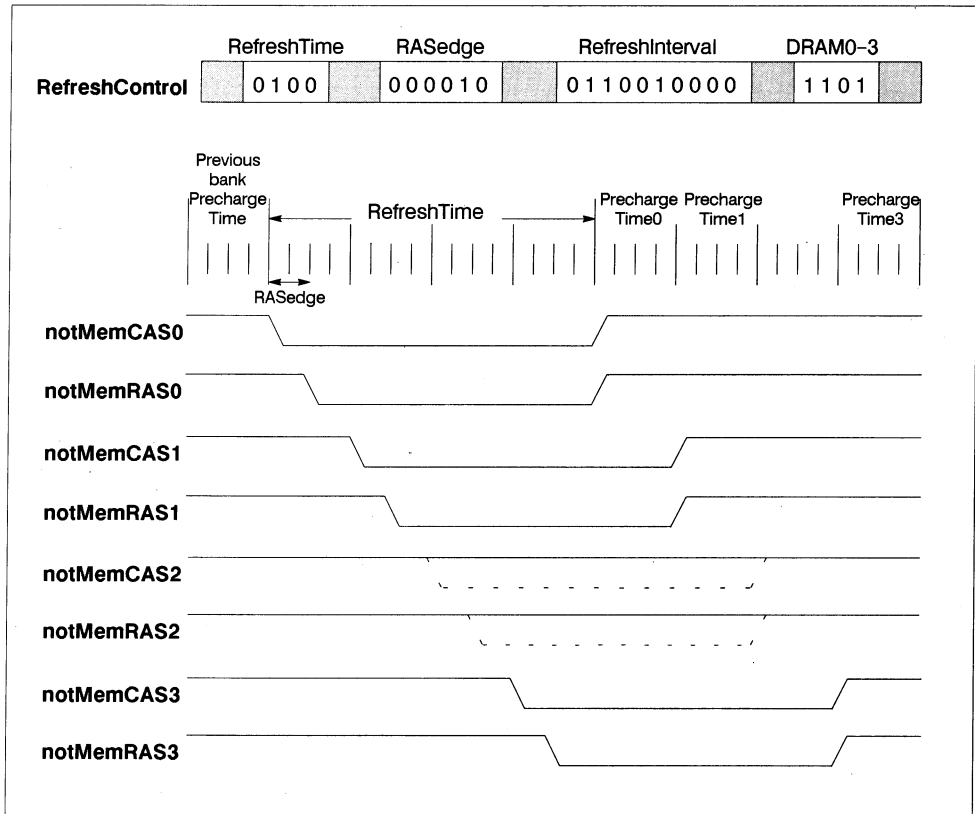
Figure 7.13   Programming of the refresh control register

8   Data/Strobe links

The IMS T9000 has four bidirectional links for normal inter-processor communications, and two additional links which can only be used for control purposes. All of these links use a protocol with two wires in each direction, one for data and one to carry a strobe signal. These links are therefore referred to as data/strobe (DS)links. The DS links are capable of:

- Up to 100 MBaud.

- 80 MBytes/second peak total bidirectional data rate.

- Support for virtual channels and through routing.

The links are TTL compatible and are series matched to 100 ohm transmission lines.

Each DS pair carries tokens and an encoded clock. The tokens can be data or control tokens. Figure 8.1 shows the format of data and control tokens on the data and strobe wires. Data tokens are 10 bits long and contain a parity bit, a flag which is set to 0 to indicate a data token, and 8 bits of data. Control tokens are 4 bits long and contain a parity bit, a flag which is set to 1 to indicate a control token, and 2 bits to indicate the type of control token.
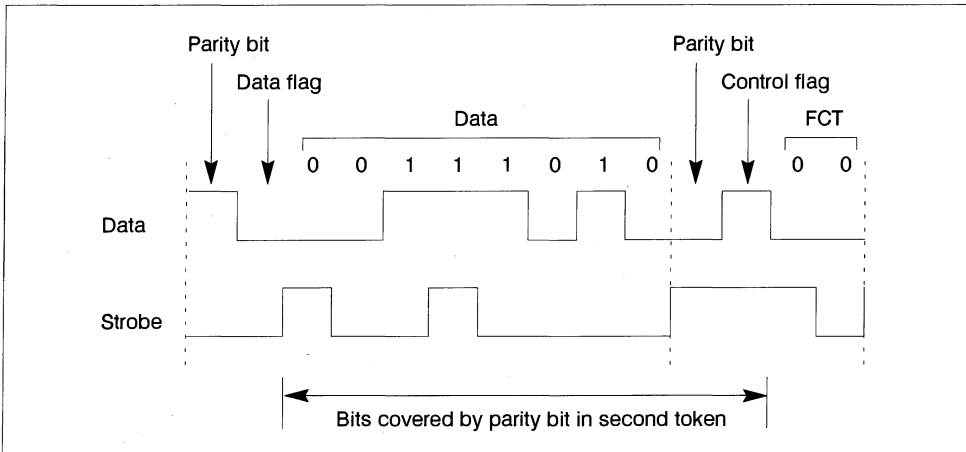
Figure 8.1   Link data format

The parity bit in any token covers the parity of the data or control bits in the previous token, and the data/control flag in the same token, as shown in figure 8.1. This allows single bit errors in the token type flag to be detected. **Odd** parity checking is used. To ensure the immediate detection of errors null tokens are sent in the absence of other tokens. The coding of the control tokens is shown in table 8.1.

Flow control token	FCT	P100
End of packet	EOP	P101
End of message	EOM	P110
Escape token	ESC	P111
Null token	NUL	ESC P100

Table 8.1   Control token codings

8.1   Low-level flow control

The DS link protocol separates the functions of flow control and process synchronization. Flow control is done entirely within the link module and process synchronization is built into a higher-level packet system (see chapter 4).

Token-level flow control is performed in each link module, and the additional flow control tokens used are not visible to the higher-level packet protocol. The token-level flow control mechanism prevents a sender from overrunning the input buffer of a receiving link. Each receiving link input contains a buffer for at least 8 tokens (more buffering than this is in fact provided). Whenever the link input has sufficient buffering available to consume a further 8 tokens a FCT is transmitted on the associated link output, and this FCT gives the sender permission to transmit a further 8 tokens. Once the sender has transmitted a further 8 tokens it waits until it receives another FCT before transmitting any more tokens. The provision of more than 8 tokens of buffering on each link input ensures that in practice the next FCT is received before the previous block of 8 tokens has been fully transmitted, so the token-level flow control does not restrict the maximum bandwidth of the link.

Note that token-level flow control is imposed on a device-to-device basis across each physical link, whereas packet-level flow control is performed on a processor-to-processor basis, and message synchronization is performed on a process-to-process basis.

8.2   Link speeds

The IMS T9000 links can support a range of communication speeds, which are programmed by writing to registers in the configuration space. At reset all links are configured to run at the **BaseSpeed** of 10 Mbits/sec.

Only the transmission speed of a link is programmed as reception is asynchronous. This means that links running at different speeds can be connected, provided that each device is capable of receiving at the speed of the connected transmitter.

The transmission speed of all of the links on a given device are related to the speed of a single on-chip clock. The frequency of this master clock is programmed through the **SpeedMultiply** bit field described in section 8.4. The master frequency is divided down to obtain the transmission frequency for each link. The division factor can be programmed separately for each link via the **SpeedDivide** bit field described in section 8.4. For a given device, with a given programmed master clock frequency, this arrangement allows each link to be run at one of four transmission speeds, as shown in table 8.2.

SpeedMultiply	SpeedDivide				BaseSpeed
	∕ 1	∕ 2	∕ 4	∕ 8	
8	80	40	20	10.0	10
10	100	50	25	12.5	10
12	Reserved	60	30	15.0	10
14	Reserved	70	35	17.5	10
16	Reserved	80	40	20.0	10
18	Reserved	90	45	22.5	10
20	Reserved	100	50	25.0	10

Table 8.2   Link transmission speed in Mbits/sec

8.3   Errors on links

Link inputs detect parity and disconnection conditions as errors. A disconnection error indicates one of two things: either the link has been physically disconnected, or an error has occurred at the other end of the link which has then stopped transmitting. The bit fields **ParityError** and **DiscError** indicate when parity and disconnect errors occur.

The DS links are designed to be highly reliable within a single subsystem and can be operated in one of two environments, determined by the **LocalizeError** bit in each link.

In the majority of applications, the communications system should be regarded as being totally reliable. In this environment errors are considered to be very rare, but are treated as being catastrophic if they do

occur. This environment is the default on power-on reset, with all links having their **LocalizeError** bit set to 0. If an error occurs it will be detected and reported via a message sent along **CLink0**. The CPU and VCP of the IMS T9000 will be halted. Normal practice will then be to reset the subsystem in which the error has occurred and to restart the application.

For some applications, for instance when a disconnect or parity error may be expected during normal operation, an even higher level of reliability is required. This level of fault tolerance is supported by localizing errors to the link on which they occur, by setting the **LocalizeError** bit of the link to 1. In addition a *data link layer* process must be connected to each virtual channel associated with the link. These processes are responsible for establishing and maintaining a higher level flow control, using time-out to detect that a message has not completed, and requesting retransmission. If an error occurs, packets in transit at the time of the error will be discarded or truncated, and the link will be reset without the error being reported via the control link.

For information on *the data link layer* refer to chapter 4 of *'Computer Networks' by Andrew S. Tanenbaum, published by Prentice-Hall International (ISBN: 0-13-166836-6).*

8.4   Link configuration registers

The links (in common with a number of other sub-systems of the IMS T9000) are controlled via a separate configuration address space. The registers in this address space are accessed via the *ldconf* and *stconf* instructions, or via *CPeek* and *CPoke* command messages received along **CLink0**.

Each DS link has three registers, the **LinkMode** register, **LinkCommand** register and **LinkStatus** register.

In addition the configuration space contains the **DSLinkPLL** register which contains the **SpeedMultiply** bit. This takes the 5 MHz input clock and multiplies it by a programmable value to provide the root clock for all the DS links.

The tables below describe the functionality of the DS links to be controlled, and the associated bit fields in the configuration registers.

Bit	Bit field	Function
5:0	SpeedMultiply	Sets DS link master clock to required value (see table 8.2).

Table 8.3   Bit fields in the **DSLinkPLL** register

The **Link0-3Mode** registers power up into a default state and may be reprogrammed before or after the link has been started.

Bit	Bit field	Function
1:0	SpeedDivide	Sets transmit speed of the **Link0-3** (see table 8.2). $00 = /1, 01 = /2, 10 = /4, 11 = /8$
2	SpeedSelect	Sets the **Link0-3** to transmit at the speed determined by the **SpeedDivide** bits as opposed to the base speed of 10 Mbits/s.
3	LocalizeError	When set errors are no longer reported to the control link. Packets in transit at the time of an error will be discarded or truncated.

Table 8.4   Bit fields in the **Link0-3Mode** registers

The **Link0-3Command** registers are write only and contain four bits which when set cause a specific action to be taken by the DS link.

Bit	Bit field	Function
0	ResetLink	Resets the link engine of the Link0–3. The token state is reset, the flow control credit is set to zero, the buffers are marked as empty, and the parity state is reset.
1	StartLink	When a transition from 0 to 1 occurs Link0–3 will be initialized and commence operation.
2	ResetOutput	Sets both outputs of Link0–3 low.
3	WrongParity	The Link0–3 output will generate incorrect parity. This may be used to force a parity error on the transputer at the other end of the Link0–3.

Table 8.5   Bit fields in the **Link0–3Command** registers

The **Link0–3Status** registers are read only and contain six bits which contain information about the state of the DS link.

Bit	Bit field	Function
0	LinkError	Flags that an error has occurred on the Link0–3.
1	LinkStarted	Flags that the output Link0–3 has been started and no errors have been detected.
2	ResetOutputComplete	Flags that ResetOutput has completed on the Link0–3.
3	ParityError	Flags that a parity error has occurred on the Link0–3.
4	DiscError	Flags that a disconnect error has occurred on the Link0–3.
5	TokenReceived	Flags that a token has been seen on the Link0–3 since ResetLink.

Table 8.6   Bit fields in the **Link0–3Status** registers

9   Control links

The control links on all IMS T9000 transputer family products allow a separate control network to be used to assist in error handling and configuring, booting, resetting and analysing processors and other components connected in a system, even in the presence of errors on the data communications links in the network. Many of these functions can also be performed directly by software running on an IMS T9000.

The device has two bidirectional control links; **CLink0** and **CLink1**. They use the same electrical and packet level protocols as the normal data links, and a control link network will generally be connected to one of the data links of a controlling IMS T9000. All communications with the controlling processor are via **CLink0**. **CLink1** is provided to allow IMS T9000 product family components to be connected in a daisy-chain. This allows a simple physical connectivity to be used for the controlling network, as shown in figure 9.1.
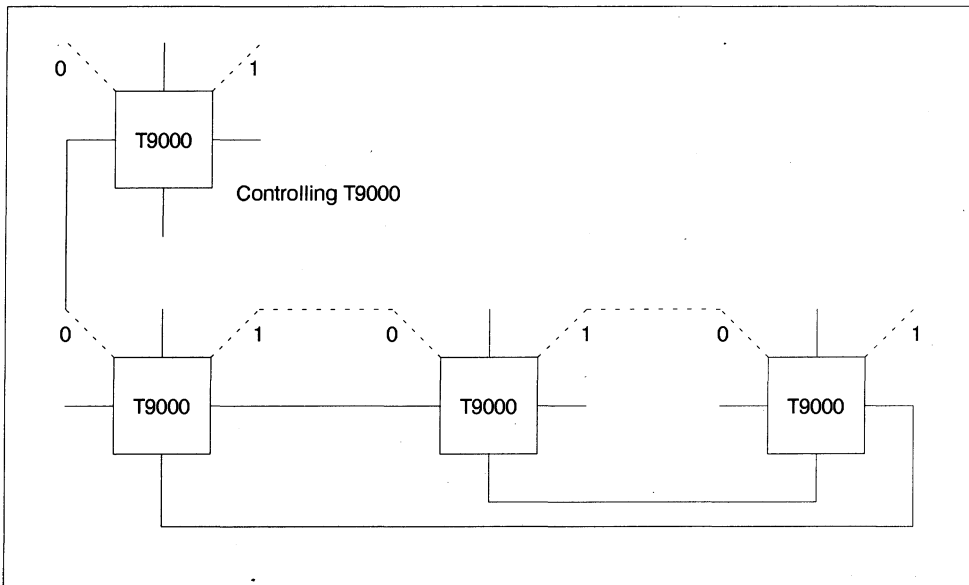
Figure 9.1   A daisy–chained control link network

For large systems IMS C104 dynamic packet routing devices can be used to connect the controlling network as a physical tree. In all cases the controlling network forms a logical tree with each device having a virtual link connected to the control process at the root of the logical tree.

9.1   Initialization

When the network is initialized the first communication with each device programs identity and return addresses to establish the virtual channels between the control process and that device. The identity address determines whether a packet arriving on **CLink0** is for that device, and if not, the message is forwarded along **CLink1** until it reaches its destination.

9.2   Commands

A high level protocol is defined for the controlling network to allow the control process to issue commands to, and receive responses from, devices in the network. Commands are sent as packets with the first byte after the header containing a command code, which may be followed by additional data. The following table details the command codes. Each command is terminated by an EOM token.

Command	Additional data	Function
Start	Return header	Allocates an identity and return header to each node. This must be the first command received following power on reset.
Reset	Level	Resets the processor to the given level (see section 10.1).
Identify	None	Returns the identity and the revision number of the device.
Stop	None	Stops the processor 'cleanly' so that register values are pre-served. Acts like the **Analyse** pin on the T8 transputer.
CPeek	Address	Returns the value stored at the given address in the device config-uration space. If the address is invalid (e.g. does not exist in the programmed external memory map) an invalid status is returned.
CPoke	Address, data	Writes data to the configuration space register at the given address. If the address is invalid an invalid status is returned.
Peek	Address	Returns the word value stored at the given address in the normal address space. If the address is invalid an invalid status is returned.
Poke	Address, data	Writes data to the given address in the normal address space. If the address is invalid an invalid status is returned.
Run	**Wdesc, Iptr**	Causes the processor to start executing with given **Wdesc** and **Iptr**.
Boot	Address, length	Sends the address of the memory into which the boot code is to be written, together with the length of the data to be input.
BootData	Data	This command is followed by 16 bytes of data which are written in 4 byte words to the current value of the boot address. The boot address is incremented after each write.
ReBoot	None	Causes reboot from ROM.
RecoverError	None	This command is used in error recovery on the control links (see section 9.3).
ErrorHandshake	None	Handshakes error message.

Table 9.1   Control link codes

Each command message is acknowledged by an acknowledge packet in the normal manner (see section 4.2). In addition the higher level control protocol requires that all command messages are acknowledged by a response message before the control process can send another command message to the same de-vice. (However, *Reset* and *RecoverError* command messages may be sent to any node at any time to allow the control process to handle error conditions in the network.)

The response message can contain the result of a *Peek* or *Identify* command, or it may be simply a hand-shake code corresponding to the command message. Table 9.2 lists the response messages to each of the command messages. The data parameter 'Status' indicates whether or not there has been an error in per-forming the operation.

Response	Additional Data
StartHandShake	None
ResetHandShake	Status
IdentifyResult	Device type and rev
StopHandShake	Status
CPeekResult	Data, status
CPokeHandShake	Status
PeekResult	Data, status
PokeHandShake	Status
RunHandShake	Status
StartBootHandShake	Status
BootDataHandShake	Status
RecoverHandShake	None
Error	Error code

Table 9.2   Control link responses

9.3   Errors on control links

The control link network is assumed to be designed and connected by the user to achieve very high reliability. The control links should be operated at a low enough speed to ensure this.

If a parity or disconnect error occurs on **CLink1** then an error message is sent to the control process along **CLink0**. If a parity or disconnect error occurs on **CLink0** then an error message cannot be sent to the control process. However, the output of **CLink0** is halted, and this will be detected by the adjacent device, which will report the error to the control process. In this manner all errors on the control link system are reported to the control process.

9.4   Stand alone mode

In a small system, such as a single IMS T9000, in which **CLink0** of each device is not connected, the IMS T9000 can be set to operate in *stand alone mode* by setting a bit in its **Status** register.

In stand alone mode the occurrence of a catastrophic error causes the fifth bank (see page 97) to be re-enabled and the ROM code restarted. A flag is set in the configuration space to indicate that such a restart has occurred. This flag can be accessed by the *testpranal* instruction.

9.5   Link speed

Each control link is powered up running at a standard speed of 10 MHz. This speed can be subsequently changed during configuration by programming the relevant **SpeedDivide** bit field in the configuration space. The speed selection for control links is identical to that of the data DS–links (see section 8.2), and the control links share the master link clock.

9.6   Control link configuration registers

The link module hardware in each control link is identical to that in each data link. An equivalent set of configuration bit fields is provided for each control link, as was described in section 8.4 for the data links.

10   Levels of reset and the configuration space

The term *configuration* is used to refer to the sequence of operations required to take an IMS T9000 transputer network from its power-on state to having an application, or operating system, running. In doing so the state of the network must be taken through a sequence of defined levels or *reset levels*. These are shown in figure 10.1.
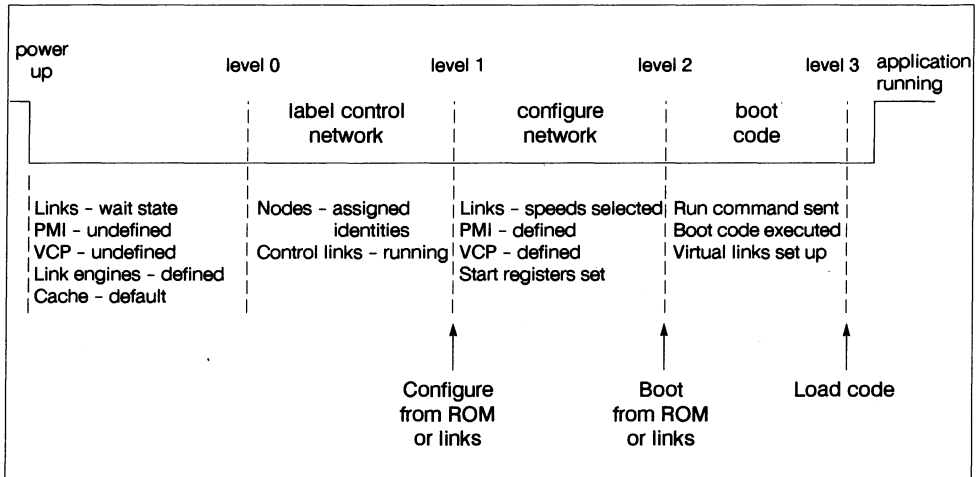
Figure 10.1   Reset levels

10.1   Reset Levels

During configuration the state of a network of IMS T9000 transputers is changed in a sequence of phases. Each phase takes the network from one reset level to the next:

10.1.1   Level 0 – hardware reset

After a hardware reset each IMS T9000 is in the following state:

> The processor is stopped, **Wdesc** is *NotProcess.p* and the scheduler queues are empty.

> The state of the PMI and VCP is not defined, and both are inactive.

> All the (data and control) links are in Wait state with a default speed of 10 MHz. Each link is in **TimesOneMode** and **Halt** is false. The identity and return headers for the control links are undefined.

> The cache is initialized to act as 16 Kbytes of on-chip RAM.

The network can be returned to level 0 by taking all the reset pins in the network high.

10.1.2   Level 1 – labelled control network

The labelling phase moves from level 0 to level 1. In it the identity and return headers are set by a *Start* command message being received on **CLink0**, as described in section 9.2. Level 1 for the network has all identity and return headers configured and all connected control links operational.

In a small system, such as a single IMS T9000 operating in stand alone mode (see section 9.4), the identity and return headers remain undefined. Any error occurring which would normally output an error message on **CLink0** will result in the fifth bank being re-enabled and the ROM code being restarted. Level 1 in this case is considered to have the identity and return headers configured as undefined.

The network can be reset to level 1 by sending a *reset* command message to each IMS T9000. After this reset message the identity and return headers are still valid. All other registers in the configuration space are reset to their level 0 values.

10.1.3   Level 2 – configured network

The configuration phase moves from level 1 to level 2. The state (resident in the configuration space) required to make all subsystems of the IMS T9000 operational, is programmed. If the **StartFromROM** pin was sampled high at the end of the hardware reset then a process will be executed from ROM. This will use the *stconf* instruction to program the configuration space registers. If the **StartFromRom** pin was sampled low then the configuration space will be programmed by *CPoke* command messages received down **CLink0**.

The network can be reset to level 2 by sending a reset command message to each IMS T9000. At this level of reset the application program is stopped (possibly in order to reload and run another one that is configuration compatible) whilst the hardware configuration is unchanged. This level of reset leaves the values in the configuration space of the PMI unaltered and still active.

10.1.4   Level 3 – booted network

The booting phase moves from level 2 to level 3. This phase is responsible for setting up the virtual links for the network using the instructions described in section 4. This is always performed by running code, but this code can either be executed from ROM, or be loaded down the control link using the *Boot* and *BootData* command message.

10.1.5   Loading code

The network is now connected and code can be loaded via the communication links, or executed from ROM.

10.2   Configuration space

A number of subsystems of the IMS T9000 are controlled through a separate address space, the configuration space. These addresses are accessed either by the *ldconf* and *stconf* instructions, or by *CPeek* and *CPoke* command messages received along **CLink0**. The locations accessed via the configuration address space are 32-bit registers for controlling the VCP, cache, PMI, and links. The features controlled via this address space are generally machine specific. Configuration and system code which uses them should be kept self contained to allow easy migration of code to future transputer implementations.

The functionality controlled by most of these registers has been described in earlier sections of this document.

11   Instruction set

The transputer instruction set has been designed for simple and efficient compilation of high–level languages. All instructions have the same format, designed to give a compact representation of the operations occurring most frequently in programs.

Each instruction consists of a single byte divided into two 4–bit parts. The four most significant bits of the byte are a function code and the four least significant bits are a data value.
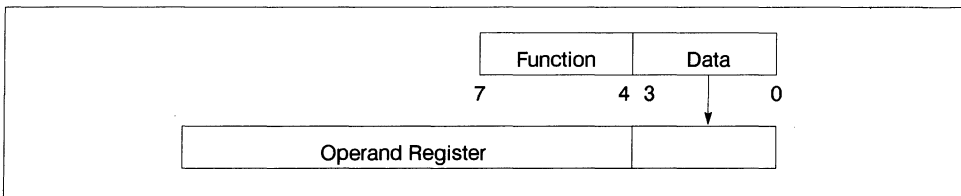
Figure 11.1   Instruction format

11.1   Direct functions

The representation provides for sixteen functions, each with a data value ranging from 0 to 15. Thirteen of these, shown in table 11.1, are used to encode the most important functions.

load constant	*add constant*	*equals constant*
load local	*store local*	*load local pointer*
load non–local	*store non–local*	*load non–local*
jump	*conditional jump*	*call*
adjust workspace		

Table 11.1   Direct functions

The most common operations in a program are the loading of small literal values and the loading and storing of one of a small number of variables. The *load constant* instruction enables values between 0 and 15 to be loaded with a single byte instruction. The *load local* and *store local* instructions access locations in memory relative to the workspace pointer. The first 16 locations can be accessed using a single byte instruction.

The *load non–local* and *store non–local* instructions behave similarly, except that they access locations in memory relative to the **Areg** register. Compact sequences of these instructions allow efficient access to data structures, and provide for simple implementations of the static links or displays used in the implementation of high level programming languages such as occam, Pascal or ADA.

11.2   Prefix functions

Two more function codes allow the operand of any instruction to be extended in length; *prefix* and *negative prefix*.

All instructions are executed by loading the four data bits into the least significant four bits of the operand register, which is then used as the instruction's operand. All instructions, except the prefix instructions, end by clearing the operand register ready for the next instruction.

The *prefix* instruction loads its four data bits into the operand register and then shifts the operand register up four places. The *negative prefix* instruction is similar, except that it complements the operand register before shifting it up. Consequently operands can be extended to any length up to the length of the operand register by a sequence of prefix instructions. In particular, operands in the range –256 to 255 can be represented using one prefix instruction.

The use of prefix instructions has certain beneficial consequences. Firstly, they are decoded and executed in the same way as every other instruction, which simplifies and speeds instruction decoding. Secondly, they simplify language compilation by providing a completely uniform way of allowing any instruction to take an operand of any size. Thirdly, they allow operands to be represented in a form independent of the processor wordlength.

11.3   Indirect functions

The remaining function code, *operate*, causes its operand to be interpreted as an operation on the values held in the evaluation stack. This allows up to 16 such operations to be encoded in a single byte instruction. However, the prefix instructions can be used to extend the operand of an *operate* instruction just like any other. The instruction representation therefore provides for an indefinite number of operations.

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 encodings which require a single prefix operation.

11.4   Efficiency of encoding

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 local* and *add* require just one processor cycle or less with grouping.

The instruction representation gives a more compact representation of high level language programs than more conventional instruction sets. Since a program requires less store to represent it, less of the memory bandwidth is taken up with fetching instructions. Furthermore, as memory is word accessed the processor will receive four instructions for every fetch.

Short instructions also improve the effectiveness of instruction pre–fetch, which in turn improves processor performance. There is a pre–fetch buffer which contains several words, so the processor rarely has to wait for an instruction fetch before proceeding. Since the buffer is transparent on jumps, there is little time penalty when a jump instruction causes the buffer contents to be discarded.

11.5   Interaction of the processor pipeline and the instruction set

The IMS T9000 has a pipelined processor with 5 pipeline stages. Each stage is dedicated to a particular operation, which in the main correspond to individual instructions, although even some of the simple instructions are operated on in more than one pipeline stage.

Stage	Operation	Function
0	Local	Push constants and locals onto the execution stack.
1	Address	Calculate addresses of non–local operands.
2	Read	Read non–local variables.
3	Alu	Stack–based ALU and FPU operations.
4	Conditional Jump/Store	Conditional jump or write results back to memory.

Table 11.2   Pipeline stages

The IMS T9000 treats commonly occurring sequences of instructions as if they were a single 'grouped' operation. The pipelined execution unit is able to execute several groups at the same time. Most groups execute in one cycle, thus delivering an instruction rate well in excess of one instruction per cycle. An example of decoding is shown below:

Program	Mnemonic	Group
x := 0	*ldc 0; stl x*	1st group
y := #24	*pfix 2; ldc 4; stl y*	2nd group
w := x + y	*ldl x; ldl y; add*	3rd group
	stl w	4th group
z := w + (x + y)	*ldl x; ldl y; add*	5th group
	ldl w; add; stl z	6th group
e[0] := a[3] + b[4]	*ldl a; ldnl 3; ldl b; ldnl 4; add*	7th group
	ldl e; stnl 0	8th group
b[j] := a[i]	*ldl i; ldl a; wsub; ldnl 0*	9th group
	ldl j; ldl b; wsub; stnl 0	10th group

Table 11.3   Expression evaluation

Evaluation of expressions sometimes requires use of temporary variables in the workspace, but the number of these can be minimized by careful choice of the evaluation order.

Groups commonly take one cycle at each stage in the pipeline, so that as groups are passed continuously down the pipeline one group is executed per cycle. However, a number of factors may cause a group to take more than one cycle at a given stage in the pipeline. These are enumerated below:

1   **Long ALU/FPU operations:** Most ALU/FPU operations take one cycle; those frequently used instructions which take longer are shown in the table below. The processor cycles column of the instruction set tables detail all instructions which take longer than one cycle.

Operation	Cycles	Notes
prod	2 – 5	
mul	2 – 5	
div	5 – 12	
rem	6 – 13	
lmul	3 – 6	
ldiv	15	
lshr	2	
lshl	2	
crcbyte	4	
crcword	16	
fpadd	2	1
fpsub	2	1
fpmul (single)	2	1
fpmul (double)	3	1
fpdiv (single)	8	1
fpdiv (double)	15	1
fprem (single)	5 – 74	1
fprem (double)	5 – 529	1
fprange (single)	5–10	1
fprange (double)	5–17	1

table continued overleaf

table continued from previous page

Operation	Cycles	Notes
fpsqrt (single)	8	1
fpsqrt (double)	15	1

Notes:

1 These figures assume normalized values, there is a 2 cycle overhead for each denormalized operand or result (except there is no overhead for a denormalized result from *fprem*).

Table 11.4   Speed of ALU/FPU operations

2 **Stack conflicts:** There are occasions when a group will produce a value on the integer or floating point evaluation stack which will then be used by the group. If the following group requires it in an earlier pipeline stage than it is produced in, then the group will have to wait. This occurs mainly with the subscript instructions. Table 11.5 below shows the stages in which values are produced and consumed. If a value is produced and pushed onto the stack in stage n in a particular group, and is consumed in stage m in the following group, then $n - m$ extra cycles will have to be allowed for.

| Instruction | Stage | |
	Consumed	Produced
ldc		0
ldl		0
ldlp		0
mint		0
ldnlp	1	1
All subscript instructions	1	1
ldnl	1	2
load16	1	2
lb	1	2
All ALU and FPU instructions	3	3
cj	4	
All store instructions	4	

Table 11.5   Stages in which instructions operate

3 **Load/store conflicts:** Stores occur in later pipeline stages than loads, so if the load is to the same address as the store, the memory is not yet in the state that the group expects it to be in. When this happens, the second group proceeds until the operand that would have been loaded is actually used, at which point it waits until the data that is to be written has passed it. All writes generate their values at stage 4, which are then consumed in either stages 1 or 3. If it is in stage 3, then there will be no penalty, but there will be a 2 cycle penalty when the value is consumed in stage 1. The load may not occur in the immediately following cycle, but in the subsequent one, in which case any penalty is one cycle less.

4 **Jumps:** A jump causes a pipeline to be (partially) empty while the instruction at the destination address is fetched and decoded. The number of cycles added to the normal time for a group is given in the following table:

Instruction	Cycles
j	2
cj (taken)	4
lend (loop back)	2
lend (terminate)	5
call	3
ret	2

Notes:

1   All these figures assume cache hits, if cache misses occur it may take longer, dependent on the PMI speed.

Table 11.6   Jumps

11.6   Floating point instructions

In the T8xx transputer the basic addition, subtraction, multiplication and division operations are performed by single instructions. Certain less frequently used floating point instructions are selected by a value in register **Areg** (this should be taken into account when allocating registers). A *load constant* instruction *ldc* is used to load register **Areg**; the *floating point entry* instruction *fpentry* then uses this value to select the floating point operation. This pair of instructions is termed a *selector sequence*. Names of operations which use *fpentry* begin with *fpu*.

In the IMS T9000 all FPU operations can be performed by an equivalent single instruction coding, the names of these operations begin with *fp* as oppose to *fpu*. However, the *fpentry* instruction has been retained in order to provide compatibility with the T8.

11.7   Instruction characteristics

Tables 11.10 to 11.44 give the complete set of instructions grouped by function, with tables 11.33 to 11.44 detailing the new IMS T9000 instructions.

The Function Codes table 11.10 gives the basic function code set. Where the operand is less than 16, a single byte encodes the complete instruction. If the operand is greater than 15, one prefix instruction (*pfix*) is required for each additional four bits of the operand. If the operand is negative the first prefix instruction will be *nfix*. Examples of *pfix* and *nfix* coding are given in table 11.7.

Mnemonic		Function code	Memory code
ldc	#3	#4	#43
ldc	#35		
is coded as			
pfix	#3	#2	#23
ldc	#5	#4	#45
ldc	#987		
is coded as			
pfix	#9	#2	#29
pfix	#8	#2	#28
ldc	#7	#4	#47
ldc	−31	(*ldc* #FFFFFFE1)	
is coded as			
nfix	#1	#6	#61
ldc	#1	#4	#41

Table 11.7   *prefix* coding

Tables 11.11 to 11.44 give details of the operation codes. Where an operation code is less than16 (e.g. *add*: operation code **05**), the operation can be stored as a single byte comprising the *operate* function code **F** and the operand (**5** in the example). Where an operation code is greater than15 (e.g. *ladd*: operation code **16**), the *pfix* function code **2** is used to extend the instruction. If the operand code is negative (e.g. *initvlcb:* operation code $\overline{1}6$), the *nfix* function code **6** is used to extend the instruction. These examples are illustrated in table 11.8.

The load device identity (*lddevid*) instruction (table 11.22) pushes the device type identity into the **Areg** register. Each product is allocated a unique group of numbers for use with the *lddevid* instruction. The product identity numbers for the IMS T9000 will start at 60.

In the Floating Point Operation Codes (tables 11.24 to 11.32), a selector sequence code is indicated in brackets in the Operation Code column. This refers to the indirection code, the operand for the *ldc* instruction (see section 11.6).

Mnemonic		Function code	Memory code
add	(*op. code #5)*		#F5
is coded as			
opr	add	#F	#F5
ladd	(*op. code #16)*		#21F6
is coded as			
pfix	#1	#2	#21
opr	#6	#F	#F6
initvlcb	(*op. code #$\overline{1}$6)*		#61F6
is coded as			
nfix	#1	#6	#61
opr	#6	#F	#F6

Table 11.8   *operate* coding

Where applicable the instruction set tables contain a processor cycles column. This refers to the number of cycles taken by an instruction.

There are a number of errors that can be trapped. When this occurs, an error code is returned to the trap handler. Any instruction which is not in the instruction set tables is an invalid instruction and is flagged illegal, returning an error code to the trap handler.

The **Note** column of the tables indicates the descheduling and error features of an instruction as described in table 11.9. It also indicates which instructions cannot be used in G–, L– or P–processes.

Ident	Feature
E	Error can be explicitly set
O	Integer overflow / divide by zero error
U	Unaligned memory access to word / half word
M	Invalid memory address for P–process
i	IEEE invalid operation exception
z	IEEE divide by zero exception
o	IEEE overflow exception
u	IEEE underflow exception
x	IEEE inexact exception
t	T800 FPU error exception
I	Interruptible instruction
B	Instruction can cause a breakpoint, G–processes only
T	Timesliceable instruction
P	Instruction not allowed in P–process
G	Instruction not allowed in G–process
L	Instruction not allowed in L–process
D	The instruction is a descheduling point
d	Denormalized operands or results can take 2 processor cycles longer

Table 11.9   Instruction features

Function Code	Memory Code	Mnemonic	Name	Notes
0	0X	j	jump	B,T,D
1	1X	ldlp	load local pointer	
2	2X	pfix	prefix	
3	3X	ldnl	load non–local	M,U
4	4X	ldc	load constant	
5	5X	ldnlp	load non–local pointer	
6	6X	nfix	negative prefix	
7	7X	ldl	load local	M
8	8X	adc	add constant	O
9	9X	call	call	M
A	AX	cj	conditional jump	
B	BX	ajw	adjust workspace	M
C	CX	eqc	equals constant	
D	DX	stl	store local	M
E	EX	stnl	store non–local	M,U
F	FX	opr	operate	

Table 11.10   IMS T9000 function codes

Operation Code	Memory Code	Mnemonic	Processor cycles	Name	Notes
46	24F6	and	1	and	
4B	24FB	or	1	or	
33	23F3	xor	1	exclusive or	
32	23F2	not	1	bitwise not	
41	24F1	shl	1	shift left	
40	24F0	shr	1	shift right	
05	F5	add	1	add	O
0C	FC	sub	1	subtract	O
53	25F3	mul	2 – 5	multiply	O
72	27F2	fmul	3 – 6	fractional multiply	O
2C	22FC	div	5 – 12	divide	O
1F	21FF	rem	6 – 13	remainder	O
09	F9	gt	1	greater than	
04	F4	diff	1	difference	
52	25F2	sum	1	sum	
08	F8	prod	2–5	product	

Table 11.11   IMS T9000 arithmetic/logical operation codes

Operation Code	Memory Code	Mnemonic	Processor cycles	Name	Notes
16	21F6	ladd	1	long add	O
38	23F8	lsub	1	long subtract	O
37	23F7	lsum	1	long sum	
4F	24FF	ldiff	1	long diff	
31	23F1	lmul	3 – 6	long multiply	
1A	21FA	ldiv	15	long divide	O
36	23F6	lshl	2	long shift left	
35	23F5	lshr	2	long shift right	
19	21F9	norm	2 – 3	normalize	

Table 11.12   IMS T9000 long arithmetic operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
00	F0	rev	reverse	
3A	23FA	xword	sign extend to word	
56	25F6	cword	check word	E
1D	21FD	xdble	extend to double	
4C	24FC	csngl	check single	E
42	24F2	mint	minimum integer	
5A	25FA	dup	duplicate top of stack	
79	27F9	pop	pop processor stack	

Table 11.13   IMS T9000 general operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
5B	25FB	move2dinit	initialize data for 2D block move	
5C	25FC	move2dall	2D block copy	M,I
5D	25FD	move2dnonzero	2D block copy non–zero bytes	M,I
5E	25FE	move2dzero	2D block copy zero bytes	M,I

Table 11.14   IMS T9000 2D block move operation codes

Operation Code	Memory Code	Mnemonic	Processor cycles	Name	Notes
74	27F4	crcword	16	calculate crc on word	
75	27F5	crcbyte	4	calculate crc on byte	
76	27F6	bitcnt	8	count bits set in word	
77	27F7	bitrevword	1	reverse bits in word	
78	27F8	bitrevnbits	1	reverse bottom n bits in word	

Table 11.15   IMS T9000 CRC and bit operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
02	F2	bsub	byte subscript	
0A	FA	wsub	word subscript	
81	28F1	wsubdb	form double word subscript	
34	23F4	bcnt	byte count	
3F	23FF	wcnt	word count	
01	F1	lb	load byte	M
3B	23FB	sb	store byte	M
4A	24FA	move	move message	M,I

Table 11.16   IMS T9000 indexing/array operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
22	22F2	ldtimer	load timer	
2B	22FB	tin	timer input	P,I,D
4E	24FE	talt	timer alt start	P
51	25F1	taltwt	timer alt wait	P,I,D
47	24F7	enbt	enable timer	P
2E	22FE	dist	disable timer	P,I

Table 11.17   IMS T9000 timer handling operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
07	F7	in	input message	P,D,I
0B	FB	out	output message	P,D,I
0F	FF	outword	output word	P,D,I
0E	FE	outbyte	output byte	P,D,I
43	24F3	alt	alt start	P
44	24F4	altwt	alt wait	P,D
45	24F5	altend	alt end	P
49	24F9	enbs	enable skip	P
30	23F0	diss	disable skip	P
48	24F8	enbc	enable channel	P
2F	22FF	disc	disable channel	P
12	21F2	resetch	reset channel	P

Table 11.18   IMS T9000 input/output operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
20	22F0	ret	return	M
1B	21FB	ldpi	load pointer to instruction	
3C	23FC	gajw	general adjust workspace	M,U
06	F6	gcall	general call	
21	22F1	lend	loop end	M,T,U, D

Table 11.19   IMS T9000 control operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
0D	FD	startp	start process	P
03	F3	endp	end process	P,D
39	23F9	runp	run process	P
15	21F5	stopp	stop process	P,D
1E	21FE	ldpri	load current priority	

Table 11.20   IMS T9000 scheduling operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
13	21F3	csub0	check subscript from 0	E
4D	24FD	ccnt1	check count from 1	E
29	22F9	testerr	test error false and clear	L,P
10	21F0	seterr	set error	E
55	25F5	stoperr	stop on error	L,P,D
57	25F7	clrhalterr	clear halt–on–error	P
58	25F8	sethalterr	set halt–on–error	P
59	25F9	testhalterr	test halt–on–error	P

Table 11.21   IMS T9000 error handling operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
2A	22FA	testpranal	test processor analyze	P
3E	23FE	saveh	save high priority queue registers	L,P,U
3D	23FD	savel	save low priority queue registers	L,P,U
18	21F8	sthf	store high priority front pointer	L,P
50	25F0	sthb	store high priority back pointer	L,P
1C	21FC	stlf	store low priority front pointer	L,P
17	21F7	stlb	store low priority back pointer	L,P
54	25F4	sttimer	store timer	L,P
17C	2127FC	lddevid	load device identity	
7E	27FE	ldmemstartval	load value of memstart address	P

Table 11.22   IMS T9000 processor initialization operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
B1	2BF1	break	break	B,L,P
B2	2BF2	clrj0break	clear jump 0 break enable flag	P
B3	2BF3	setj0break	set jump 0 break enable flag	P
B4	2BF4	testj0break	test jump 0 break enable flag set	P

Table 11.23 IMS T9000 debugger support codes

Operation Code	Memory Code	Mnemonic	Name	Notes
8E	28FE	fpldnlsn	fp load non-local single	M,U
8A	28FA	fpldnldb	fp load non-local double	M,U
86	28F6	fpldnlsni	fp load non-local indexed single	M,U
82	28F2	fpldnldbi	fp load non-local indexed double	M,U
9F	29FF	fpldzerosn	load zero single	
A0	2AF0	fpldzerodb	load zero double	
AA	2AFA	fpldnladdsn	fp load non local & add single	M,U,i,o,u,x,t
A6	2AF6	fpldnladddb	fp load non local & add double	M,U,i,o,u,x,t
AC	2AFC	fpldnlmulsn	fp load non local & multiply single	M,U,i,o,u,x,t
A8	2AF8	fpldnlmuldb	fp load non local & multiply double	M,U,i,o,u,x,t
88	28F8	fpstnlsn	fp store non-local single	M,U
84	28F4	fpstnldb	fp store non-local double	M,U
9E	29FE	fpstnli32	store non-local int32	M,U

Table 11.24 IMS T9000 floating point load/store operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
AB	2AFB	fpentry	floating point unit entry	
A4	2AF4	fprev	fp reverse	
A3	2AF3	fpdup	fp duplicate	

Table 11.25 IMS T9000 floating point general operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
D0 (22)	2DF0	fprn	set rounding mode to round nearest	
D6 (06)	2DF6	fprz	set rounding mode to round zero	
D4 (04)	2DF4	fprp	set rounding mode to round positive	
D5 (05)	2DF5	fprm	set rounding mode to round minus	

Table 11.26 IMS T9000 floating point rounding operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
83	28F3	fpchkerr	check fp error	E
9C	29FC	fptesterr	test fp error false and clear	
CB (23)	2CFB	fpseterr	set fp error	t
DC (9C)	2DFC	fpclrerr	clear fp error	

Table 11.27   IMS T9000 floating point error operation codes

Operation Code	Memory Code	Mnemonic	Processor cycles	Name	Notes
94	29F4	fpgt	2	fp greater than	i,t
95	29F5	fpeq	2	fp equality	i,t
92	29F2	fpordered	1	fp orderability	i
91	29F1	fpnan	1	fp NaN	i
93	29F3	fpnotfinite	1	fp not finite	i
DE (0E)	2DFE	fpchki32	2	check in range of type int32	i,t
DF (0F)	2DFF	fpchki64	2	check in range of type int64	i,t

Table 11.28   IMS T9000 floating point comparison operation codes

Operation Code	Memory Code	Mnemonic	Processor cycles	Name	Notes
D7 (07)	2DF7	fpr32tor64	2	real32 to real64	i,t,d
D8 (08)	2DF8	fpr64tor32	2	real64 to real32	i,o,u, x,t,d
9D	29FD	fprtoi32	2 – 4	real to int32	i,x,t
96	29F6	fpi32tor32	2 – 4	int32 to real32	M,U,x
98	29F8	fpi32tor64	2	int32 to real64	M,U
9A	29FA	fpb32tor64	2	bit32 to real64	M,U
DD (0D)	2DFD	fpnoround	2	real64 to real32, no round	
A1	2AF1	fpint	2 – 4	round to floating integer	i,x,t

Table 11.29   IMS T9000 floating point conversion operation codes

Operation Code	Memory Code	Mnemonic	Processor cycles		Name	Notes
			Single	Double		
87	28F7	fpadd	2	2	fp add	i,o,u,x,t,d
89	28F9	fpsub	2	2	fp subtract	i,o,u,x,t,d
8B	28FB	fpmul	2	3	fp multiply	i,o,u,x,t,d
8C	28FC	fpdiv	8	15	fp divide	i,z,o,u,x,t,d
DB (0B)	2DFB	fpabs	1	1	fp absolute	i,t
DA (0A)	2DFA	fpexpinc32	2	2	multiply by 2^{32}	i,o,t,d
D9 (09)	2DF9	fpexpdec32	2	2	divide by 2^{32}	i,u,x,t,d
D2 (12)	2DF2	fpmulby2	2	2	multiply by 2.0	i,o,t,d
D1 (11)	2DF1	fpdivby2	2	2	divide by 2.0	i,u,x,t,d

Table 11.30   IMS T9000 floating point arithmetic operation codes

Operation Code	*fpentry* code	Memory Code	Mnemonic	Processor cycles		Name	Notes
				Single	Double		
n/a	01	412AFB	fpusqrtfirst	2	2	floating-point square root first	
n/a	02	422AFB	fpusqrtstep	2	2	floating-point square root step	
n/a	03	432AFB	fpusqrtlast	8	15	floating-point square root last	i,x,t,d
8F	n/a	28FF	fpremfirst	5 – 74	5 – 529	fp remainder	l,i,u,t
90	n/a	29F0	fpremstep	2	2	floating-point remainder step	

Table 11.31   IMS T9000 floating point operation codes which are included for compatibility with the IMS T805

The following tables detail the instructions provided on the IMS T9000 which are additional to the IMS T805 instruction set.

Operation Code	Memory Code	Mnemonic	Processor cycles		Name	Notes
			Single	Double		
CF	2CFF	fprem	5 – 74	5 – 529	fp remainder	l,i,u,t
D3	2DF3	fpsqrt	8	15	fp square root	i,x,t,d
8D	28FD	fprange	5–10	5–17	floating point range reduce	i,u,t
97	29F7	fpge	2	2	fp greater than or equality	i,t
9B	29FB	fplg	2	2	fp less than or greater than	i,t

Table 11.32   IMS T9000 additional floating point operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
CA	2CFA	ls	load 16 bit word	MU
F9	2FF9	lsx	load 16 bit word extended	MU
C8	2CF8	ss	store 16 bit word	MU
F8	2FF8	xsword	sign extend 16 bit word	
FA	2FFA	cs	check 16 bit word signed	E
FB	2FFB	csu	check 16 bit word unsigned	E
C1	2CF1	ssub	16 bit word subscript	
B9	2BF9	lbx	load byte extended	M
B8	2BF8	xbword	extend byte to word	
BA	2BFA	cb	check byte signed	E
BB	2BFB	cbu	check byte unsigned	E

Table 11.33   IMS T9000 part word support operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
$\overline{0}0$	60F0	swapqueue	swap run queue	P
$\overline{0}1$	60F1	swaptimer	swap timer queue	P
$\overline{0}2$	60F2	insertqueue	insert onto front of run queue	P
B0	2BF0	settimeslice	set timeslice enable/ disable	P
$\overline{0}3$	60F3	timeslice	timeslice	T,D
B5	2BF5	ldproc	load process type	P

Table 11.34   IMS T9000 process queue manipulation and timeslicing operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
C2	2CF2	ldth	load trap handler	G,P
$\overline{0}9$	60F9	selth	select trap handler	G,P,D
B6	2BF6	ldflags	load error flags	G
B7	2BF7	stflags	store error flags	G
$\overline{0}A$	60FA	goprot	go protected	G,P
$\overline{0}B$	60FB	tret	trap return	G,P
$\overline{0}8$	60F8	syscall	system call	G

Table 11.35   IMS T9000 trap handler operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
$\overline{0E}$	60FE	fpldall	floating-point load all	M,U
$\overline{0F}$	60FF	fpstall	floating-point store all	M,U
$\overline{10}$	61F0	stmove2dinit	store 2D move initialize data	
$\overline{0C}$	60FC	ldshadow	load shadow registers	P
$\overline{0D}$	60FD	stshadow	store shadow registers	P

Table 11.36   IMS T9000 state storage and retrieval operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
C9	2CF9	chantype	channel type	P,E,U
$\overline{16}$	61F6	initvlcb	initialize virtual link control block	P,E
C3	2CF3	ldchstatus	load channel status	P,E
$\overline{14}$	61F4	readhdr	read header	P,E,U,I
$\overline{17}$	61F7	setchmode	set channel mode	P,E
$\overline{19}$	61F9	swapbfr	swap buffer	P,E,U
$\overline{18}$	61F8	sethdr	set header	P,E,U
$\overline{15}$	61F5	writehdr	write header	P,U,I
$\overline{1A}$	61FA	ldvlcb	load virtual link control block	P,U
$\overline{1B}$	61FB	stvlcb	store virtual link control block	P,U
$\overline{1E}$	61FE	stopch	stop channel	P,E,D
BD	2BFD	readbfr	read buffer	P,E,U
BC	2BFC	insphdr	inspect header	P,E,U

Table 11.37   IMS T9000 channel and virtual link operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
C0	2CF0	ldcnt	load count	P,E
$\overline{1C}$	61FC	vin	variable input	P,I,E,U,D
$\overline{1D}$	61FD	vout	variable output	P,I,E,U,D

Table 11.38   IMS T9000 variable length i/o operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
$\overline{11}$	61F1	grant	grant resource	P,D
$\overline{12}$	61F2	enbg	enable grant	P
$\overline{13}$	61F3	disg	disable grant	P
$\overline{28}$	62F8	ldresptr	load resource queue pointer	P,E,U
$\overline{29}$	62F9	stresptr	store resource queue pointer	P,E,U
$\overline{2A}$	62FA	erdsq	empty resource data structure queue	P,U
$\overline{2B}$	62FB	irdsq	insert resource data structure queue	P
$\overline{2C}$	62FC	mkrc	make resource channel	P,E
$\overline{2D}$	62FD	unmkrc	unmake resource channel	P,E

Table 11.39   IMS T9000 resource channel operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
$\overline{05}$	60F5	wait	wait on semaphore	P,U,O,D
$\overline{04}$	60F4	signal	signal semaphore	P,U,O

Table 11.40   IMS T9000 semaphore operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
BE	2BFE	ldconf	load configuration	P
BF	2BFF	stconf	store configuration	P

Table 11.41   IMS T9000 configuration operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
$\overline{20}$	62F0	fdca	flush dirty cache address	P
$\overline{22}$	62F2	fdcl	flush dirty cache line	P
$\overline{21}$	62F1	ica	invalidate cache address	P
$\overline{23}$	62F3	icl	invalidate cache line	P

Table 11.42   IMS T9000 cache operation codes

Operation Code	Memory Code	Mnemonic	Name	Notes
C4	2CF4	intdis	interrupt disable	P
C5	2CF5	intenb	interrupt enable	P

Table 11.43   IMS T9000 interrupt operation codes

Operation Code	Memory Code	Mnemonic	Processor cycles	Name	Notes
C7	2CF7	cir	2	check in range	E
CC	2CFC	ciru	2	check in range unsigned	E
5F	25F5	gtu	1	unsigned greater than	

Table 11.44   IMS T9000 miscellaneous operation codes

12 Performance

The performance of the IMS T9000 is measured in terms of the number of (internal) processor cycles required to execute the program. The figures here relate to occam programs. For the same function, other languages should achieve approximately the same performance as occam.

The following tables are based on the time it takes to do ALU and FPU operations and it should be noted that many other instructions may be overlapped (see section 11.5).

12.1 Integer operations

These figures are estimates and give the minimum/maximum times for a particular operation.

Operation	Time (cycles)
Names	
variables	
in expressions	0
assigned to or input to	0 to 1
in PROC or FUNCTION call	0
channels	1
Array Variables (1–d)	
constant subscript	0
variable	0 to 1
plus subscript check	3
variable + constant subscript	0 to 1
plus subscript check	4
expression subscript	3
plus subscript check	1
Declarations	
CHAN OF *protocol*	2
[size] CHAN OF *protocol*	3 + 5 * size
PROC	0
Primitives	
assignment	0
input	15 or [5 + *move*]
output	16 or [5 + *move*]
STOP (call error handler)	7
SKIP	0
Arithmetic Operators	
+ –	1
*	2 to 5
/	5 to 12
REM	6 to 13
>> <<	1

table continued overleaf

table continued from previous page

Operation	Time (cycles)
Modulo Arithmetic Operators	
PLUS MINUS	1
TIMES	2 to 5
Boolean Operators	
OR	
first operand true	3 to 4
first operand false	0 to 1
AND	
first operand true	0
first operand false	3
NOT	0 to 1
Comparison Operators	
= < >	1
> <	1
> = < =	1
Bit Operators	
/ \ \ / > < ~	1
Expressions	
constant	0
check if error	1
Timers	
timer input	1
timer AFTER	4 to ∞
ALT (timer)	20 to ∞
ALT guard	7 to ∞
Constructs	
SEQ	0
IF	0
IF guard	4
ALT (non timer)	11 to 17
ALT guard	7 to 16
PAR	$20 * branches - 6$
WHILE	$4 + 3 * loops$
Procedures and Function	
call and return	6 to 8
scalar parameter	0 to 1
array parameter	2

table continued from previous page

Operation	Time (cycles)
Replicators	
replicated SEQ	(1 to 3) + 3 * count
replicated IF	(4 to 6) + 3 * count
replicated ALT	(13 to 23) + (13 to 22) * count
replicated timer ALT	(2 to ∞) + (13 to ∞) * count
replicated PAR	10 + 27 * count
range check on any of above	2

Table 12.1   Integer Performance

12.2   Floating point operations

All references to REAL32 or REAL64 operands within programs compiled for the IMS T9000 normally produce the following performance figures.

Operation	REAL32 Time (cycles)	REAL64 Time (cycles)	Notes
Names			
variables			
in expressions	0	0 to 1	
assigned to	0 to 1	1 to 2	
input to	1	1	
in PROC or FUNCTION call	0	0	
Arithmetic Operators			
+ −	2	2	1
*	2	3	1
/	8	15	1
SQRT	8	15	1
REM	5 to 74	5 to 529	1, 2
Comparison Operators			
= < >	2	2	
> <	2	2	
> = < =	2	2	
Conversions			
REAL32 to −		2	
REAL64 to −	2		
INT32 to −	2 to 4	2	
INT64 to −	12	8	
To INT32 from −	4 to 6	4 to 6	
To INT64 from −	11	11	

Notes:

1   These figures assume normalized values, there is a 2 cycle overhead for each denormalized operand or result (except there is no overhead for a denormalized result from *fprem*).

2   Typical value for REAL32 is 5 to 11; for REAL64 is 5 to 18, longer times are extremely rare.

Table 12.2   Floating point performance

12.3   Predefines

Operation	Time (cycles)
LONGADD	1
LONGSUM	1
LONGSUB	1
LONGDIFF	1
LONGPROD	3 to 6
LONGDIV	15
SHIFTRIGHT	2
SHIFTLEFT	2
NORMALISE	2 to 3
ASHIFTRIGHT	3
ASHIFTLEFT	4
ROTATERIGHT	3
ROTATELEFT	3
FRACMUL	3 to 6
BITCOUNT	8
CRCBYTE	4
CRCWORD	16
BITREVNBIT	1
BITREVWORD	1

Table 12.3   Predefines

13   Compatibility with the IMS T805

13.1   Binary code compatibility

Existing binary code compiled for the IMS T805 will run on the IMS T9000, even though the IMS T9000 has been designed to provide a significant performance improvement compared to the IMS T805, and includes a number of major functional enhancements. This has been achieved by the provision of a special process type (the G–process, see section 3.4) which retains compatibility with IMS T805 binary code.

Prior to any IMS T805 compatible code being loaded, a preamble must be executed on the IMS T9000. This preamble programs the behavior of the subsystems of the IMS T9000 (including its memory interface and links) and emulates the boot time behaviour of the IMS T805. That is, it allows code to be booted down the data links of the IMS T9000 in the same way as for the IMS T805. In situations where the IMS T9000 is required to act as a direct replacement for the IMS T805 this preamble will usually be performed by code executing out of ROM. The preamble is invisible to IMS T805 compatible code when it is run.

In order to provide a simple hardware interface between first generation T2/T4/T8–family components and T9000–family components a systems converter, the IMS C100, has been designed. The IMS C100 performs protocol conversions between T2/T4/T8–family oversampled links (OS–links) and T9000–family links (DS–links). It also converts the **Reset**, **Analyse** and **Error** signals of the T2/T4/T8–family components to appropriate command messages on IMS T9000 control links.

If configured IMS T805 binary compatible code is to be run on an IMS T9000, connected via an IMS C100 to a network of T2/T4/T8–family components, then byte mode must be set via the VCP configuration registers (see section 4.4.9, page 75). No virtual channels may be used. Code on both the IMS T9000 and any connected T2/T4/T8–family transputers will run unmodified.

It is possible to make use of virtual channels in this situation by reconfiguring the IMS T805 binary compatible code, and thereby changing its external channel addresses. To make use of virtual channels across links to T2/T4/T8–family components it is necessary to add protocol conversion software to those T2/T4/T8–family transputers directly connected to the IMS T9000 via IMS C100 systems converters.

13.2   Source level compatibility

The code which must be generated for both new process types (L–processes and P–processes, see section 3.4) is still instruction set compatible with the IMS T805 transputer. A number of additional instructions and in–store data structures have been provided to support new features, and some in–store data structures have been modified. Existing compilers for the IMS T805 transputer will be able to make use of the new capabilities of the IMS T9000 without significant modification. Existing source code will be able to be recompiled directly through a modified compiler, with the exception that language specific additions for allocating processes to processors and channels to hard links may need to be modified in order to exploit virtual channels.

13.3   Compatibility issues

The following are the only known exceptions to the compatibility of IMS T805 binary code described above. All are unlikely to occur in compiled binary code, but may be present in assembler code for certain systems software, such as real–time kernels:

1   The IMS T9000 stores all of its interrupted state in internal registers, whereas the IMS T805 stores some of its interrupted state in memory locations. Code which directly references these memory locations on the IMS T805 will need to be modified to run on the IMS T9000.

2   If a block move instruction on a T2/T4/T8 transputer is interrupted, when it is restarted the last word accessed prior to interrupt is accessed again. This second access will not occur on the IMS T9000. Code which modifies the interrupted state of a process on a T2/T4/T8 transputer to prevent the second access will need to be modified to run on an IMS T9000.

3   The floating point unit on the IMS T9000 does not always return the same type of NaN as the T2/T4/T8 transputer, even when running a G–process. Most code written for the T2/T4/T8–transputer will not make use of the distinction between signalling and non–signalling NaNs and will function identically on T2/T4/T8 and IMS T9000 transputers.

4   The transfer of messages across OS–links is not synchronized. This allows, for example, two four byte messages to be sent and for them to be received as a single 8 byte message on the receiving T2/T4/T8 transputer. This is not consistent with the communication of messages between processes on the same processor. In this case the transfer of messages is synchronized.

On the IMS T9000 the communication of messages across DS–links is also synchronized. Code which makes use of the lack of message synchronization across OS–links will have to be modified to run on networks of IMS T9000 transputers.

5   Storing a value from **Areg**, pops **Breg** into **Areg** and **Creg** into **Breg**, the value left in **Creg** is undefined. On T2/T4/T8 transputers the value in **Creg** actually remains unchanged. Although this was never specified, this fact was used by some users. It should be noted that this is not guaranteed to be the case on the IMS T9000.

The published behavior of the IMS T805 is defined in the *IMS T805 transputer datasheet (document number 42 1440 00)* and in *The Transputer Instruction Set – A Compiler Writer's Guide (document number 72 TRN 119 05)*.

14   Mixed T9000 and T2/T4/T8 systems

14.1   Byte mode

Each IMS T9000 data link (**Link0–3**) may be set to operate either in virtual channel mode (as described in section 4.2) or in byte mode by setting the associated bit in the **VCPLink0–3Mode** registers. The IMS T9000 links can be set independently of each other, enabling each IMS T9000 to be connected to several different networks.

In byte mode, the IMS T9000 links are designed to operate in conjunction with the IMS C100 system protocol converter, which converts to the oversampled links (OS–links) used on earlier (T2/T4/T8) transputer products. The IMS C100 uses a mode of operation in which it converts packetized messages on the IMS T9000 links to and from byte–streams on the OS–links.

> **IMS T9000 to T2/T4/T8** – The IMS C100 buffers a packet, which it acknowledges as soon as it has room to buffer another. This is repeated until it receives a packet terminated with an EOM, which it acknowledges when the last byte is acknowledged on the OS–link.
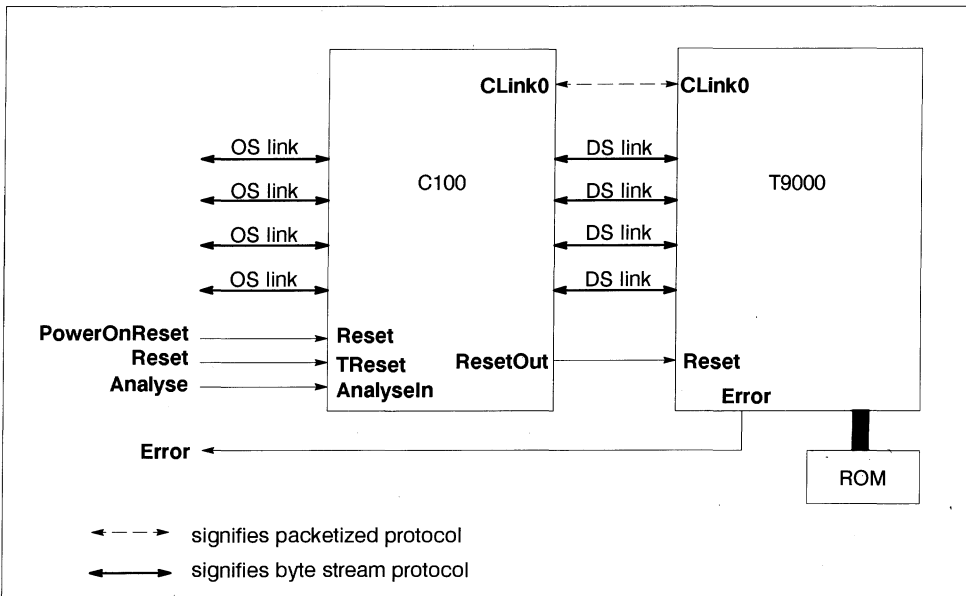
Figure 14.1   Converting an IMS T9000 transputer for use in a T2/T4/T8 series network

> **T2/T4/T8 to IMS T9000** – The IMS T9000 sends information about how much data it wishes to input on a separate virtual channel to the IMS C100. The IMS C100 then forms packets to send to the IMS T9000. The IMS C100 informs the IMS T9000 when an unsolicited byte arrives, to enable `alternative` to work. This is done by sending a zero–length message to the IMS T9000.
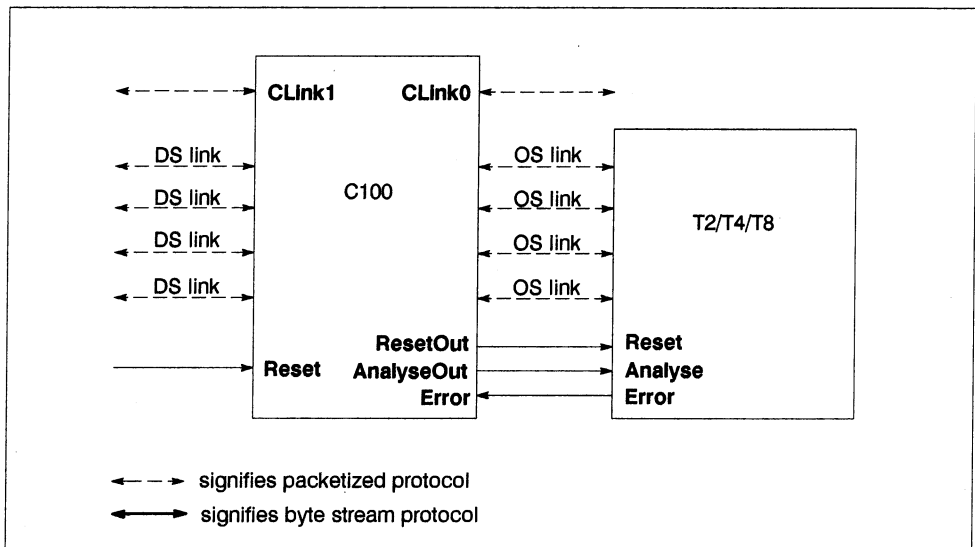
Figure 14.2   Converting a T2/T4/T8 series transputer for use in an H-series network

The IMS C100 can respond to both data bytes and acknowledges on the OS-links immediately by buffering data from the IMS T9000 and holding a count of the input length, thus maintaining full bandwidth along them.

Note that two IMS T9000 links in byte mode will not work correctly if connected directly together.

Refer to the *IMS C100 System Protocol Converter Preliminary Information* for further information on enabling IMS T9000s to interface with earlier transputer products and vice versa.

15 Package specifications

The IMS T9000 is available in a 208 pin ceramic quad flat pack package which conforms to JEDEC specifications. It is a cavity down package with the dimensions and thermal characteristics detailed below.

15.1 208 pin ceramic quad flat pack package dimensions
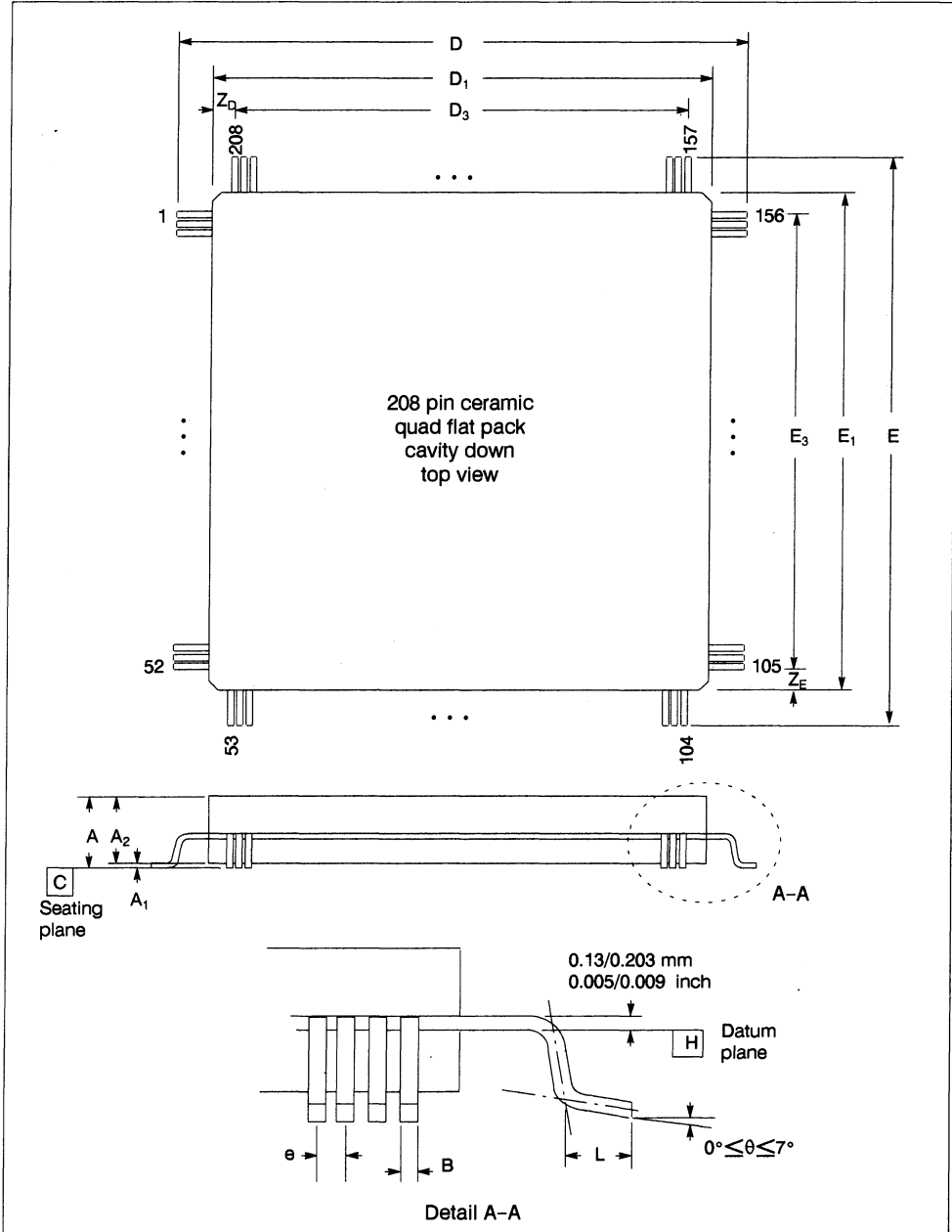
Figure 15.1   208 pin ceramic quad flat pack dimensions

Symbol	Millimeters		Inches		Notes
	Min	Max	Min	Max	
A		4.07		0.160	
A_1	0.25		0.010		
A_2	3.17	3.67	0.125	0.144	
D	31.65	32.15	1.246	1.266	1
D_1	27.90	28.10	1.098	1.106	2
D_3	25.50 Ref		1.004 Ref		
Z_D	1.25 Ref		0.049 Ref		
E	31.65	32.15	1.246	1.266	1
E_1	27.90	28.10	1.098	1.106	2
E_3	25.50 Ref		1.004 Ref		
Z_E	1.25 Ref		0.049 Ref		
L	0.65	0.95	0.026	0.037	
e	0.50 Basic		0.020 Basic		
B	0.15	0.25	0.006	0.010	

Notes

1 To be determined at seating plane C.

2 To be determined at seating plane H.

Table 15.1   208 pin ceramic quad flat pack dimensions

15.2   208 pin ceramic quad flat pack thermal characteristics

The junction to case thermal resistance (θ_{JC}) of the package is given below.

Symbol	Parameter	Min	Nom	Max	Units
θ_{JC}	Junction to case thermal resistance			1	°C/W

Table 15.2   Thermal characteristics

Technical notes describing the thermal behavior of specific package and heat sink combinations will be issued subsequent to this preliminary information.

16   Thermal management

The following section describes the relationship between the thermal resistance, temperature and power dissipation of the device.

The peak operating temperature T_J of the chip is:

$$T_J = T_A + \theta_{JA} * P_D$$

where T_A is the external ambient temperature in °C, θ_{JA} is the junction-to-ambient thermal resistance in °C/W, and P_D is the peak power dissipated by the chip. The maximum junction temperature T_J for an IMS T9000 to operate at the specified maximum operating frequency is 100°C. Derating curves of maximum operating frequency against power dissipation will be included in the final datasheet.

θ_{JA} for the package is dependent on air flow and heat sink:

$$\theta_{JA} = \theta_{JC} + \theta_{CA}$$

where θ_{CA} is the case–to–ambient thermal resistance and θ_{JC} is the junction-to-case thermal resistance which is given in the table 15.2.

A reasonable operating ambient temperature range (T_A) can be achieved using a heat sink and/or forced air flow cooling. A heat sink and ambient air flow cooling can be used to reduce the case to ambient thermal resistance (θ_{CA}) and hence the junction to ambient thermal resistance (θ_{JA}), thus increasing T_A. The design of a heat sink will need to be determined by the system designer taking into account both thermal perform- ance requirements and size requirements.

Power considerations

The internal power dissipation of the IMS T9000 depends on **Vcc**, and is substantially independent of tem- perature. It is dependent on operating frequency and program execution. The typical peak internal power dissipation (P_{INT}) for an IMS T9000 operating at 50 MHz is 3W.

The total power dissipation of the IMS T9000 is dependent on operating frequency, program execution, external memory configuration, and output pin loading.

The total peak power dissipation P_D of the chip is:

$$P_D = P_{INT} + P_{PMI}$$

The peak power dissipation of the PMI (P_{PMI}) can be determined for a given memory configuration from the following equation:

$$P_{PMI} = V_{CC}^2 * ((n_{pA} * C_{pinA} * f_A) + (n_{pS} * C_{pinS} * f_S) + (n_{pD} * C_{pinD} * f_D))$$

where,

> n_p is the total number of active (address/ strobe/ data) pins
> C_{pin} is the actual capacitance per (address/ strobe/ data) pin
> f is the effective operating frequency per (address/ strobe/ data) pin

The maximum allowable capacitances that can be connected to each class of pins are:

Symbol	Parameter	Max	Units
C_{pinA}	Capacitance per address pin	250	pF
C_{pinS}	Capacitance per strobe pin	60	pF
C_{pinD}	Capacitance per data pin	60	pF
$n_{pA} * C_{pinA}$	Total address bus capacitance	2500	pF
$n_{pS} * C_{pinS}$	Total strobe pins capacitance	500	pF
$n_{pD} * C_{pinD}$	Total data bus capacitance	5000	pF

Table 16.1   Capacitance specifications
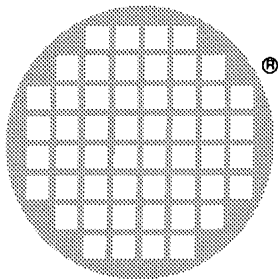
IMS C104
packet routing
switch

Preliminary Information

FEATURES

32 way programmable packet router
100 Mbits/s serial bi-directional links
640 Mbytes/s bandwidth
Concurrent processing of packets
High rate of packet processing
    – up to 200 M packets/s
Less than 1 μ second packet latency
Non-blocking crossbar
Separate control system
Wormhole interval routing algorithm
Cascadable to any depth
No loss of signal integrity
Partitioning
Grouped adaptive routing
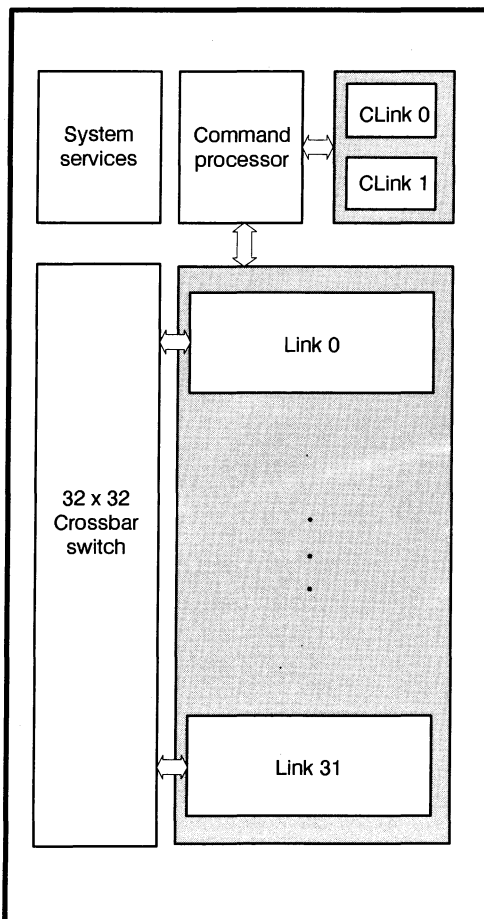
This is preliminary information on a product under development and product details may change.

1   Introduction.

This document contains preliminary information for the IMS C104 packet routing switch. The IMS C104 is part of a new product family based around the IMS T9000 transputer.

The IMS C104 is a complete, low latency, packet routing switch on a single chip. It connects 32 high band-width serial communication links to each other via a 32 by 32 way non-blocking crossbar switch, enabling messages to be routed from any of its links to any other link. The links operate concurrently and the transfer of a packet between one pair of links does not affect the data rate for another packet passing between a second pair of links. Each link can operate at up to 100 MBits/s, providing a bidirectional bandwidth of 20 MBytes/s, with the IMS C104 supporting a rate of packet processing of up to 200 M packets/s.

The IMS C104 allows communication between IMS T9000 transputers that are not directly connected. A single IMS C104 can be used to connect up to 32 IMS T9000 transputers. The IMS C104 can also be con-nected to other IMS C104s to make larger and more complex switching networks, linking any number of IMS T9000 transputers, link adaptors, and any other devices that use the link protocol. Another member of the IMS T9000 product family, the IMS C101 flexible link adaptor, will allow links to be interfaced to periph-eral buses and devices.

The IMS C104 enables networks to be built which effectively emulate a direct connection between each of the devices in the system. In the absence of any contention for a link output, the packet latency will be less than 1μ second.

A message on a IMS C104 communication system is transmitted as a sequence of packets. To ensure that packets which are parts of different messages can be routed, each packet contains a header. The IMS C104 uses the header of each packet arriving to determine the link to be used to output the packet. Anything after the header is treated as the packet body until the packet terminator is received. This enables the IMS C104 to transmit packets of arbitrary length.

In most packet switching networks complete packets are stored internally, decoded, and then routed to the destination node. This causes relatively long delays due to high latency at each node. To overcome this limitation, the IMS C104 uses *wormhole routing*, in which the routing decision is taken as soon as the routing information, which is contained in the packet header, has been input. Therefore the packet header can be received, and the routing decision taken, before the whole packet has been transmitted by the source. A packet may be passing through several nodes at any one time. Thus, latency is minimized and transmission can be continuous.

The term *wormhole routing* comes from the analogy of a worm crawling through soil, creating a hole that closes again behind its tail. Wormhole routing is invisible as far as the senders and receivers of packets are concerned, its only effect is to minimize the latency in message transmission.

The routing algorithm which makes the routing decision is called *interval labeling*, which is complete, dead-lock free, inexpensive and fast. Each destination in a network is labeled with a number, and this number is used as the destination address in a packet header. Each link in a routing switch is labeled with an interval of possible header values, and only packets whose header value falls within that interval are output via that link.

The IMS C104 is controlled and programmed via a control link. The IMS C104 has two separate control links, one for receiving commands and one to provide daisy chaining. The control links enable networks of IMS T9000 transputers and IMS C104s to be controlled and monitored for errors. The control links can be con-nected into a daisy chain or tree, with a controlling processor, such as an IMS T9000, at the root.

The IMS C104 contains a hardware mechanism to allow independently programmed networks to be con-nected together. It also has additional circuitry to reduce the impact of message congestion on worst-case latency and bandwidth, in heavily loaded networks.

A set of tools will be available to support the configuration of IMS T9000 systems. The tools will provide support in the configuration and initialization of networks consisting of IMS T9000 processors and IMS C104 routing switches. These tools will be contained as a standard part of the *Version 3 toolsets* for C, occam and FORTRAN.

2 Overview

2.1 Communication on IMS T9000 transputers

The IMS C104 is part of a new product family based around the IMS T9000 transputer. Communication between processes on one IMS T9000 transputer takes place over software channels. Communication between processes on different processors takes place over *virtual channels*. Multiple virtual channels are multiplexed onto each physical link by a communications processor within the IMS T9000. The links use a protocol which supports virtual channels and *dynamic message routing*, and provides a high data bandwidth.

Each message is split into a sequence of packets, and packets from different messages may be interleaved over each physical link. Interleaving packets from different messages allows any number of processes to communicate simultaneously via each physical link. Communication channels can be established between any two processes regardless of where they are physically located, or whether the channels are routed through a network. Thus, programs can be independent of network topology.

In order that packets which are parts of different messages can be distinguished by the transputer which receives them, each packet contains a one or two byte header which identifies a virtual input channel of the receiving transputer. The packet header is also used to route the packet through a network. Bytes following the header are treated as the data section of the packet until a packet termination token is received. A packet termination token is either an EOP (end of packet) token or an EOM (end of message) token.

The maximum length of data in each packet which the IMS T9000 can transmit is 32 bytes. All but the last packet of a message contains the maximum amount of data; the last contains the maximum amount of data or less.

The communications processor within the IMS T9000 enforces a high-level protocol on each virtual channel. To maintain synchronized communication, and to ensure that no data is lost, each packet of data sent along a virtual channel must be acknowledged before the next is sent. The last packet must be acknowledged before the outputting process is rescheduled. Data packets on a virtual channel are acknowledged by the communications processor by sending acknowledge packets on another virtual channel back to the processor which sent them. Acknowledge packets are packets containing no data and which are always terminated by an EOP token. This acknowledgement is process-to-process and is transparent to intermediate network components.

Virtual channels always occur in pairs between pairs of communicating processors, with one virtual channel in each direction. If a message is being communicated in one direction the virtual channel in the opposite direction is used to return acknowledge packets to the sender. The associated pair of virtual channels is referred to as a *virtual link*. A virtual link can transfer messages in both directions at the same time with data packets and acknowledge packets being interleaved on both of the virtual channels. Because virtual channels are always paired in this way it is not necessary to include source information in the packets. Thus packet headers need only represent their destinations.

The IMS C104 allows communication between IMS T9000 transputers that are not directly connected.

3 Operation of IMS C104 networks

A single IMS C104 can be used to connect up to 32 IMS T9000 transputers that are not directly connected to each other. The IMS C104 can also be connected to other IMS C104s to make larger and more complex switching networks, linking any number of IMS T9000 transputers, link adaptors, and any other devices that use the link protocol.

The IMS C104 uses a 1 or 2 byte header of each packet arriving, to determine the link to be used to output the packet. The output link taken is independent of the input link on which the packet arrives. Bytes following the header are treated as the data section of the packet until a packet termination token is received. This enables the IMS C104 to transmit packets of **arbitrary** length.

An IMS C104 network consists of one or more IMS C104 routing devices connected together by bi-directional links. Each device is called a node of the network. Some links of the network are connected to the exterior of the network, to transputers or to another network. These links are called terminal links.

In order to support the efficient routing of packets through a network the IMS C104 implements a complete routing algorithm in hardware. The component parts of the algorithm are described in the following sections.

3.1 Wormhole routing

In most packet-switching networks each routing switch inputs the whole of a packet, decodes the routing information, and then forwards the packet to the next node. This is undesirable in transputer networks because it requires storage for packets in each routing switch and it causes long delays between the output of a packet and its reception.

The IMS C104 uses *wormhole routing* (figure 3.1) in which the routing decision is taken as soon as the header of the packet has been input. If the output link is free, the header is output and the rest of the packet is sent directly from input to output without being stored. If the output link is not free the packet is buffered. The packet header, in passing through a network of IMS C104s, creates a temporary circuit through which the data flows. As the end of the packet is pulled through, the circuit vanishes. The wormhole analogy is based on the comparison with a worm crawling through sandy soil, which creates a hole that closes again behind its tail.

The implications of wormhole routing are that a packet can be passing through several IMS C104s at the same time, and the head of the packet may be received by the destination before the whole packet has been transmitted by the source. Thus latency is minimized and transmission can be continuous.

Wormhole routing is invisible as far as the senders and receivers of packets are concerned. Its major effect is to minimize the latency in the message transmission.
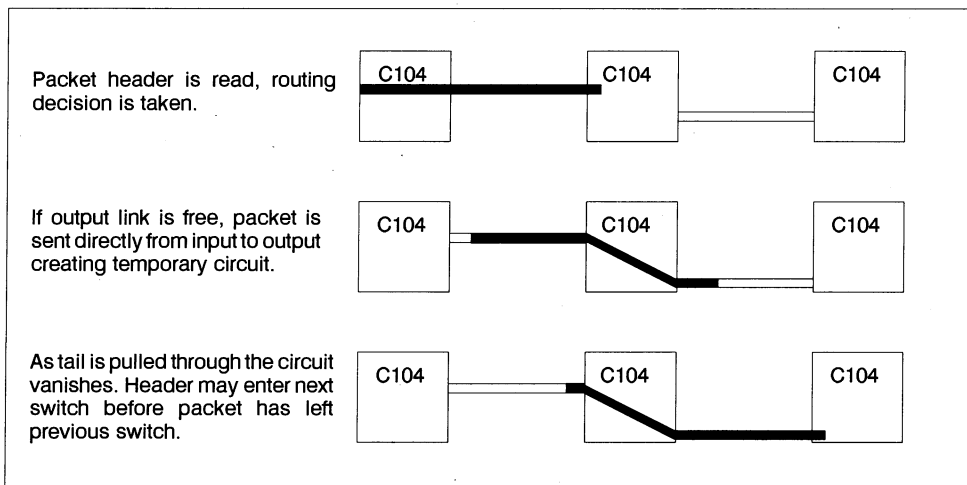
Figure 3.1   Wormhole routing

3.2 Interval labeling

Wormhole routing requires a routing strategy to decide which link a packet should be output from. The IMS C104 uses a routing scheme called *interval labeling*, whereby each output link of an IMS C104 is assigned a range, or interval, of labels. This interval contains the number of all the terminal nodes (i.e. IMS T9000 transputer, gateway to another network, peripheral chip, etc) which are accessible via that link. Each terminal link of a network has an associated interval of labels. On entering a network the packet header contains a label. The label determines which link the packet is to be output to and hence must occur within the interval associated with the destination link.

As the packet arrives at an IMS C104 the selection of the outgoing link is made by comparing the header value with the set of intervals, as in the example shown in figure 3.2. The intervals are contiguous and non-overlapping and assigned so that each header value can only belong to one of the intervals. The output link associated with the interval in which the header value lies is the one selected. In the example the incoming header contains the value 154, which lies between 145 and 186, so the packet is output along link 8.
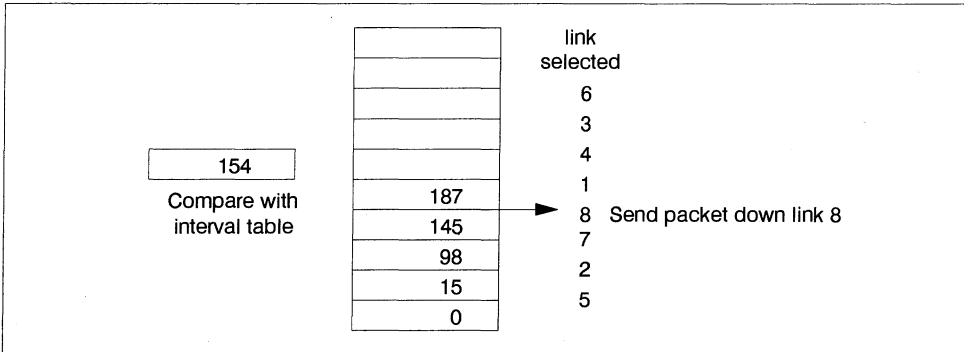
Figure 3.2   Interval labeling

Figure 3.3 gives an example of interval routing for a network of two IMS C104's and six IMS T9000 transputers showing one virtual link per transputer. The example shows six virtual channels, one to each transputer, labeled 0 to 5. The interval contains the labels of all virtual channels accessible via that link. The interval notation [3,6) is read as meaning that the header value must be greater than or equal to 3 and less than 6. If the progress of a packet with the header value 4 is followed from IMS $T9000_1$ then it is evident that it passes through both IMS C104s before leaving on the link to IMS $T9000_4$.
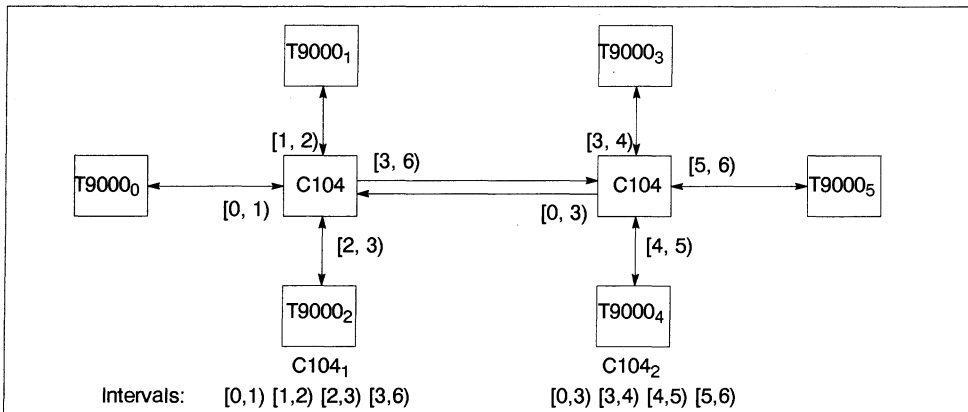
Figure 3.3   Interval routing

It is possible to label all the major network topologies such that packets follow an optimal route through the network, and such that the network is deadlock free. Optimal, deadlock free labelings, which will be provided to customers, are available for grids, hypercubes, trees and various multi–stage networks. A few topologies, such as rings, cannot be labeled in an optimal deadlock free manner. Although they can be labeled so that they are deadlock free, this is at the expense of not using one or more of the links, so that the labeling is not optimal. Optimal deadlock free labelings exist if one or more additional links are used.

Interval routing ensures that each packet takes the shortest route with low control overhead, and that all packets reach their destinations. It is independent of network topology and the output link selected is independent of the input link used. The transfer of a packet between one pair of links does not affect the data rate for another packet passing between a second pair of links. The hardware required to implement interval routing is simple, enabling many routing decisions to be made concurrently, thus providing a high rate of packet processing.

3.3   Modular composition of networks

To assist in the modular composition of routing networks the IMS C104 contains a hardware mechanism to implement *header deletion.* Header deletion mode is where each link output of the IMS C104 can be programmed to delete the header of a packet before transmitting the remainder of the packet.

The benefits achieved by header deletion are:

1   Simplified labeling of systems, by separating out the task of labeling networks from that of identifying virtual channels on IMS T9000 transputers.

2   Removal of the limit of a maximum of 64K virtual channels per system.

3   Hierarchical composition of networks.

Figure 3.4 illustrates how header deletion is used to simplify the labeling of systems by separating out the task of labeling networks from that of identifying virtual channels on IMS T9000 transputers. Figure 3.4(a) shows a system of 256 IMS T9000 transputers connected by a network of IMS C104s. All of the link inputs in the system are programmed to receive 2 byte headers. The IMS C104 interval routing tables and IMS T9000 headers (stored in the IMS T9000) are programmed to support 256 virtual channels connected to each IMS T9000 transputer, with the header values allocated as shown in figure 3.4(a).

Figure 3.4(b) shows the same system but with all the link inputs in the system programmed to receive 1 byte headers, and with the terminal links of the IMS C104 network programmed to delete headers. Note that the IMS T9000 transputer and the IMS C104 can both be configured to accept headers which are 1 or 2 bytes long. A packet is now transmitted with a header consisting of two 1 byte sub–headers. It should be noted that as far as the IMS C104 is concerned the packet has just one header, any subsequent sub–headers are treated as part of the data body of the packet. The first 1 byte sub–header routes the packet across the network to the terminal link which the packet is to be sent out of; the terminal links being numbered from 0 to 255 as shown. This header is deleted as the packet leaves a terminal link of the network. The second 1 byte sub–header is then exposed, and is interpreted by the destination IMS T9000 transputer to identify the target virtual channel.
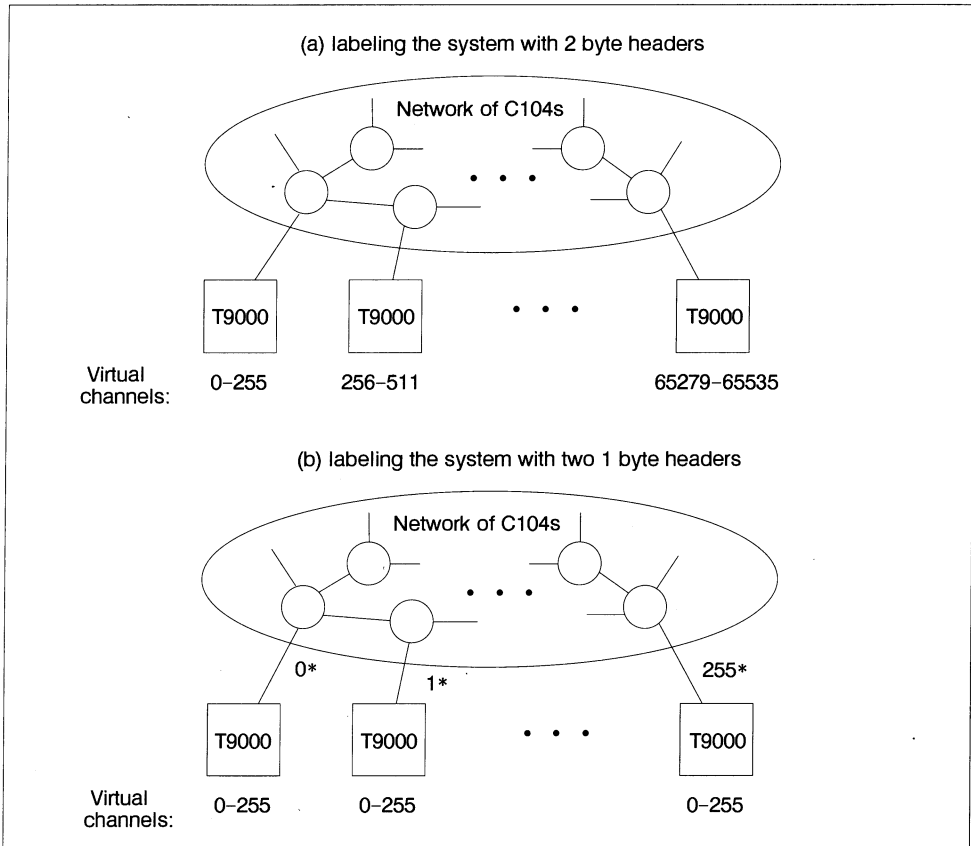
Figure 3.4   Header deletion used to separate network labeling and virtual channel identification.

In this manner header deletion allows network routing information to be separated out from the identification of virtual channels on IMS T9000 transputers. A first header is used to route the packet across a network to a terminal link, and a second header is used to identify a virtual channel within the destination transputer. The use of two 1 byte headers also decreases latency.

The total number of virtual channels in the system shown in figure 3.4 has not been increased, as headers are still 2 bytes long in total. However, the total number of virtual channels in the system can now be increased by programming the links on the IMS T9000 transputers to accept 2 byte headers (whilst the IMS C104s still accept 1 byte headers).

In this case a packet is transmitted with a header consisting of a 1 byte sub-header and a second 2 byte sub-header. As before, the first 1 byte sub-header routes the packet across the network and is deleted as the packet leaves a terminal link of the network. Thus exposing the second 2 byte sub-header which allows 64K separate virtual channels to be identified on the destination IMS T9000 transputer. Header deletion thereby removes the limit of 64K virtual channels in a total system, and replaces it with the less constraining limit of 64K virtual channels on each IMS T9000 transputer.

Header deletion also allows networks to be connected together, as shown in figure 3.5. In this example a packet is routed through two networks and then to a virtual channel on an IMS T9000 transputer. All of the terminal links of the two networks are set to header deletion mode. Figure 3.5 shows the header as it is routed through the network. The header of the packet in this case is made up of three concatenated sub-headers. The first sub-header routes the packet across the first network and is deleted as the packet leaves

the terminal link of the network. The second sub-header routes the packet across the second network in the same way. Finally the third header is exposed to identify the destination virtual channel on the IMS T9000 transputer.

In the case in which each IMS C104 is treated as a separate network and has its link outputs set to header deletion mode, packets can be explicitly steered across a network. This is at the expense of having 1 byte of header for each IMS C104 traversed.
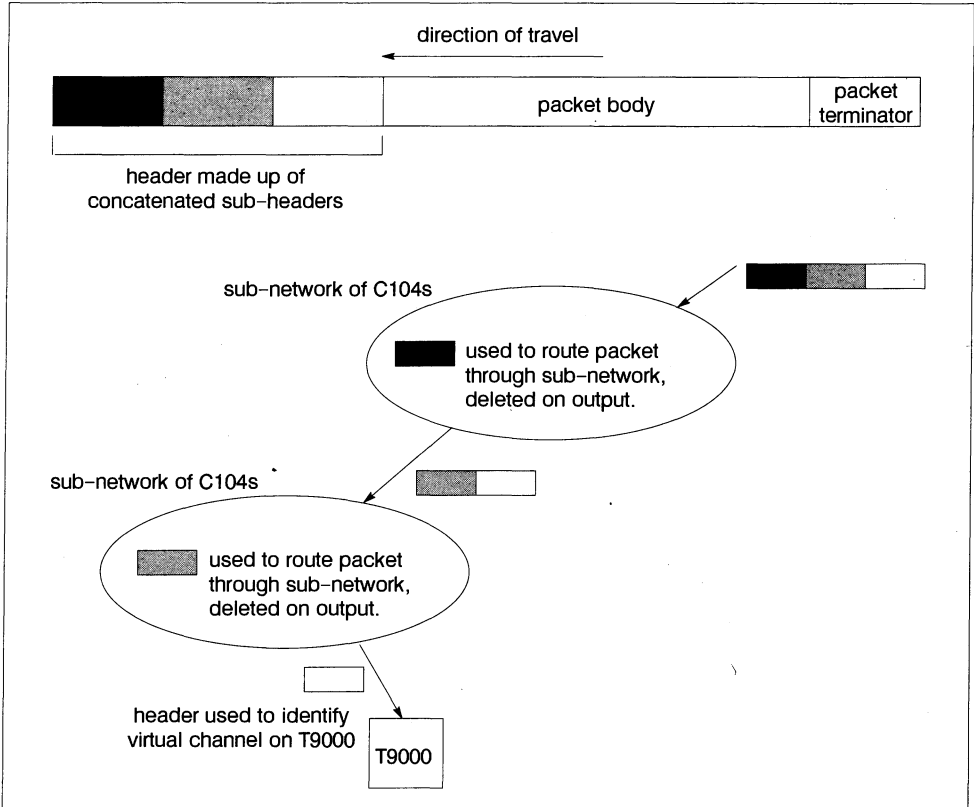
Figure 3.5   Hierarchical composition of networks using header deletion

A major advantage of extending the capabilities of the IMS C104, through header deletion, is that headers can be minimized for small systems, thus optimizing network latency and network bandwidth, whilst still enabling more complex, larger, systems to be constructed efficiently.

3.4   Use of parallel networks

System wide communication can be provided by connecting each transputer to a single routing network via one or more of its links. However, as each transputer has several links it can be connected to several different networks. These can be completely distinct networks, or simply logical sub-networks of one net-work of IMS C104s. The use of multiple networks can provide the following:

- Higher available processor to processor bandwidth.

- Separate networks for different priority messages. The link protocol does not provide any support for associating a priority with a packet. This can be supported by providing a separate network for each required message priority.

- Separate networks for identified concurrent data streams in a system designed for a specific application.

3.5   Hot spot avoidance

The routing algorithms described so far provide efficient deadlock free communications and allow a wide range of networks to be constructed from a standard router. Packets are delivered at high speed and low latency provided that there are no collisions between packets travelling through any single link.

Unfortunately, in any sparse communication network, some communications patterns cannot be realized without collisions. A link over which an excessive amount of communication is required to take place at any instant is referred to as a *hot spot* in the network, and results in packets being stalled for an unpredictable length of time.

To eliminate network hot spots, the IMS C104 can optionally implement a two phase routing algorithm. This involves every packet being first sent to a randomly chosen intermediate destination; from the intermediate destination it is forwarded to its final destination. This algorithm, referred to as *Universal Routing,* is designed to maximize capacity and minimize delay under conditions of heavy load. (This has been proven by simulations and theory. Refer to *'A scheme for fast parallel communication' SIAM J. of Computing, 11 (1982) 350–361).* It trades this off against best case performance in an empty network.

To implement two phase routing each packet must have a 'random' header prepended to it as it enters the randomizing network, which indicates its intermediate destination. This is implemented on the IMS C104 by enabling each input link to be programmed into a *random header* generation mode. In this mode the input link adds a random header to the front of each packet that it receives. The random header is generated from within a programmed range. The IMS C104 then treats this random header as the header of the packet, (the destination header is now treated as part of the data body of the packet), and routes the packet accordingly. The packet is routed on through the network until it reaches its random intermediate destination where the first phase of routing terminates.

Each IMS C104 link recognizes a range of *portal* values. The portal values set the random phase routing interval. This interval is compared with each arriving header. Any packet with a header within this interval will be recognized by the IMS C104; the random header will be deleted; and the header that is exposed is used to route the packet through the network to its final destination.

Note that the deletion of the random header associated with universal routing is different to that of the operation of header deletion mode, as described in section 3.3 above. Header deletion mode deletes headers as the packet is sent along a link output, whereas header deletion associated with universal routing occurs when the random header of the packet input into the IMS C104 is recognized to be within the portal range.

In order to ensure that deadlock does not occur the two phases of routing must use completely separate links. This is achieved by assigning destination headers and random headers from distinct intervals. All links in the network must be considered to be either *destination* or *random* links. The intervals associated with a given link on a IMS C104 must be a sub–interval of the destination or random header range as appropriate.

Effectively this scheme provides two separate networks; one for the randomizing phase and one for the destination phase. The combination will be deadlock free if the separate networks are deadlock free.

Universal routing can be beneficially applied to a wide variety of network topologies, including hypercubes and arrays. There are a small number of network topologies where universal routing is not always beneficial, as it can prevent highly optimal routings through the network being utilized.

4   Control of the IMS C104

The IMS C104 is controlled and programmed via the control links (see chapter 5). Messages sent to the IMS C104 allow its configuration registers to be set and read. The registers can be accessed via *CPeek* and *CPoke* command messages sent along the control links and control the interval selector, the random number generator and the links.

4.1   Programmable parameters

Interval routing is achieved in the IMS C104 by interval selector units. An interval selector performs the routing decision for each packet. It consists of 35 base and limit comparators (see figure 4.1). Each comparator is connected to a pair of registers, except the lowest whose base is fixed at zero. Each register is connected to the limit of one comparator and the base of the next comparator, except the top register which is connected to the limit of the top comparator only. These registers must be programmed with a set of unsigned 16 bit values ascending from zero, thus the intervals are non−overlapping and each header value can only belong to one of the intervals. This sets the interval for each link. Any link can be assigned to any interval. The output of each comparator is connected to a register (**SelectLinkn**). The **SelectLinkn** register contains the number of the associated output link. The contents are sent to the address gate if the packet header is greater than or equal to the base and less than the limit value of the adjoining comparator.

The interval selector reads in the value of the header and the pre−programmed comparators determine the corresponding link address for output. Once the path through the crossbar is set the tokens are passed through until an EOP or EOM terminator token is detected.

Each link input of a IMS C104 can be set to random header generation mode by the **Randomize** flag. In random header generation mode the random header generator produces a header which is added in front of the existing header and is used to route the packet to a random node, thus implementing the universal routing algorithm.

The lower limit and range of the random number generator must be programmed into the **RandomBase** and **RandomRange** registers.

Associated with each interval is a flag, held in the **Discard0−34** bit field, which indicates which of the intervals is the portal. If the input header is indicated as belonging to a portal interval (i.e. the random header has reached its random intermediate destination) the 'Discard' signal is sent to the header buffer telling it to discard the header. In this case the output of the ladder of comparators is not sent to the crossbar and the next 1 or 2 bytes of data (dependent on the **HeaderLength** flag) is taken as the new header and is again processed using the interval labeling algorithm.

If the header is not flagged as the portal by the **Discard0−34** bits the 'No' signal is sent to the address gate, which then allows the address which is produced from the ladder of comparators to be sent out to the crossbar. If none of the flags **Discard0−34** is set, the portal mechanism is disabled.
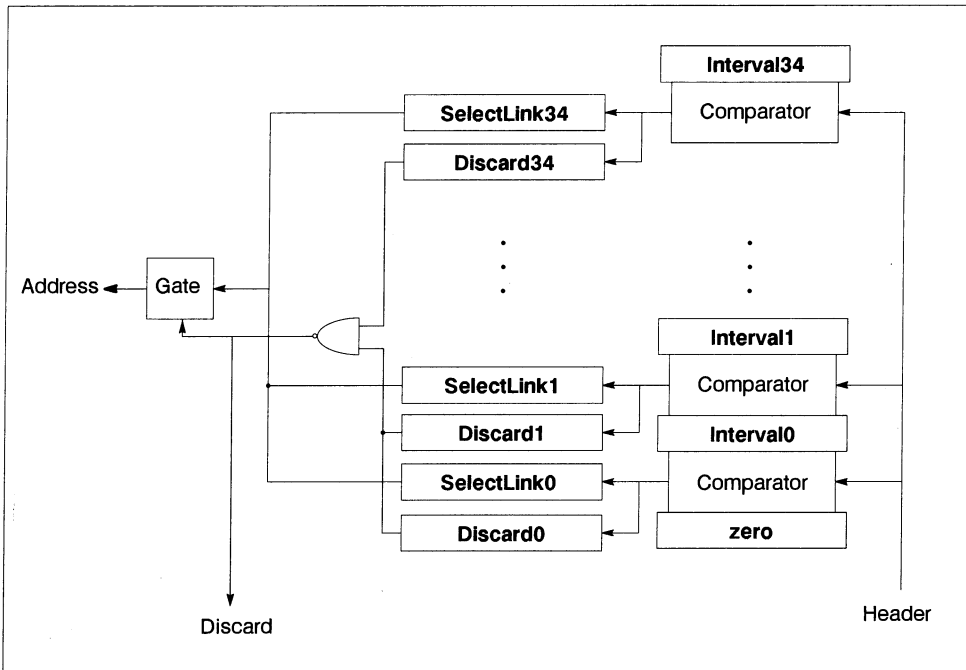
Figure 4.1   Interval selector registers

Each link can be set to input 1 or 2 byte headers. This is determined by the **HeaderLength0-31** flag in the configuration registers which are set after power on. It allows headers to be minimized for small systems, thus optimizing network latency and network bandwidth, whilst also enabling large homogeneous systems to be constructed. Heterogeneous and hierarchical systems can be implemented using hierarchical labeling and header deletion (which is implemented by setting the **DeleteHeader0-31** flag for a given link).

4.1.1   Partitioning

All the parameters described above are programmable on a per link basis. This enables an IMS C104 to be used as part of two or more different networks. For example, a single IMS C104 could be used for access to both a data network and a control network (see figure 4.2).

Partitioning provides economy in small systems, where using an IMS C104 solely for the control network is not desired.

Network 1
C104 used in a data network

Network 2
C104 used in a control network

T9000

[1, 2)

[3, 6)

T9000   C104   T9000

[0, 1)

[2, 3)

T9000

[10, 14)

[0, 9)   C104

[9, 10)

C104

Single C104 used between 2 networks

Interval table for links 4, 5 and 6

Interval	SelectLink
14	4
10	5
9	6
0	–

Network 2

[9, 10)

Link6   Link5

[0, 9)

Link4

[10, 14)

C104

Link0

[1, 2)

Link1   Link3

[0, 1)   [3, 6)

Link2

Network 1   [2, 3)

Interval table for links 0, 1, 2 and 3

Interval	SelectLink
6	3
3	2
2	0
1	1
0	–

Figure 4.2   Using partitioning to enable one IMS C104 to be used by two different networks

4.1.2 Grouped adaptive routing

The IMS C104 can implement *grouped adaptive routing*. Sets of consecutive numbered links can be configured to be grouped, so that a packet routed to any link in the set would be sent down any free link of the set. This achieves improved network performance in terms of both latency and throughput.

Figure 4.3 gives an example of grouped adaptive routing. Consider a message routed from $C104_1$, via $C104_2$, to $T9000_1$. On entering $C104_2$ the header specifies that the message is to be output down **Link6** to $T9000_1$. If **Link6** is already in use, the message will automatically be routed down **Link5**, **Link7** or **Link8**, dependent on which link is available first. The links can be configured in groups by setting the **Group0-31** bit fields. Each bit corresponds to a link and can be set to 'Start' to begin a group and 'Continue' to be included in a group, as shown in figure 4.3.

Settings in **Group0-31** bit field for $C104_2$

0	Start	⎫
1	Continue	⎬ Grouped
2	Continue	⎪
3	Continue	⎭
4	Start	
5	Start	⎫
6	Continue	⎬ Grouped
7	Continue	⎪
8	Continue	⎭
9	Start	⎫ Grouped
10	Continue	⎭
11	Start	
.	.	
.	.	
31	Start	
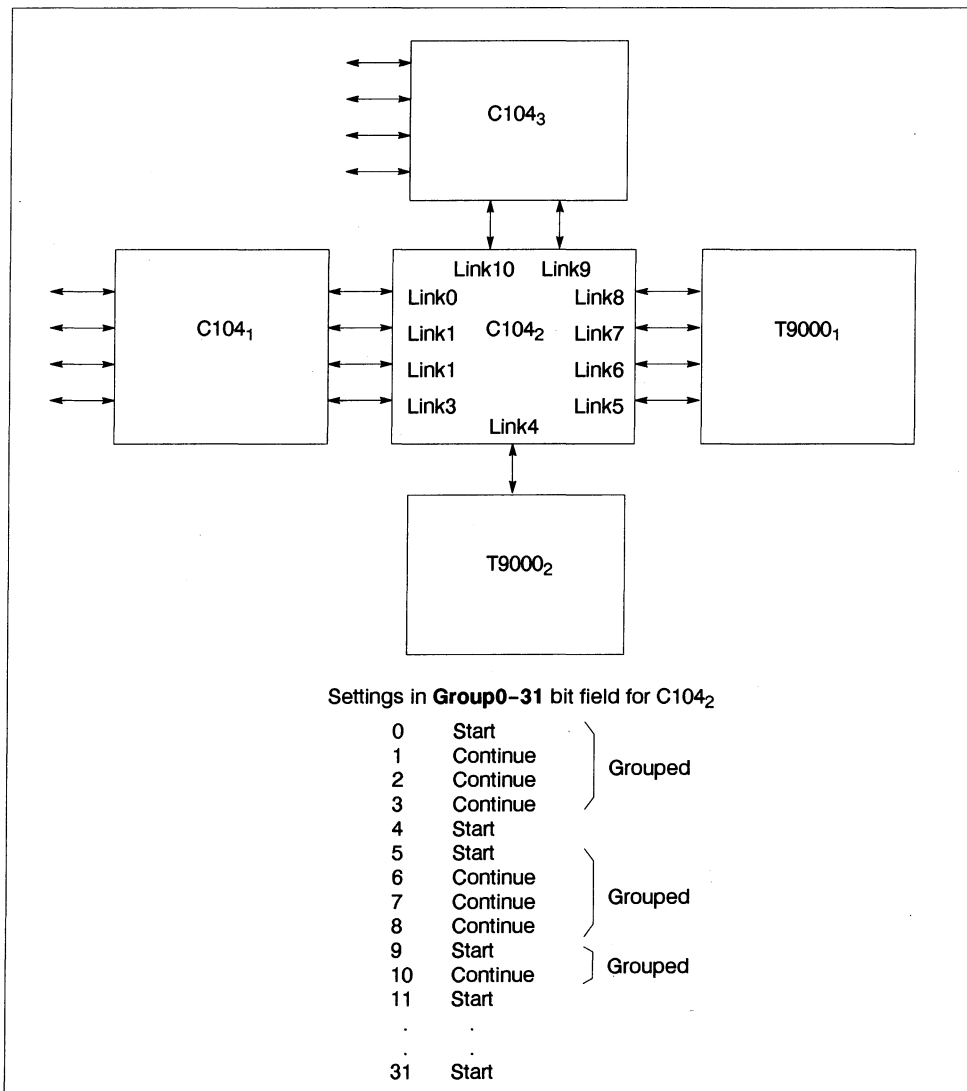
Figure 4.3   Grouped adaptive routing

4.2   Registers

All the parameters described above are loaded into the appropriate registers by the command processor in response to commands received on the control link (see section 5.1). The parameters must be supplied before the device can operate.

The functionality controlled by these registers is described below. The complete bit format of each register and the addresses of the registers are **not** included in this preliminary information.

Bit field	Function
HeaderLength	Sets the header length to 1 or 2 bytes
Randomize	Sets a given link input to random header generation mode
DeleteHeader	Sets a given link output to delete header mode

Table 4.1   Bit fields in the link configuration registers per link

Bit field	Function
Interval0–34	Sets the intervals for each link
SelectLink0–34	Indicates the associated link from which the packet is to be output
Discard0–34	Indicates which of the intervals is the portal .

Table 4.2   Interval selector registers per link

Bit field	Function
RandomSeed	Start of 16 bit pseudo–random sequence
RandomBase	Base level of random number
RandomRange	Range of random number

Table 4.3   Bit fields in the random number generator registers per link

Bit field	Function
Group	Each bit can be set to 'start of group' or 'continuation of group'.

Table 4.4   Bit field to set grouped adaptive routing per link

5 Control links

The control links on the IMS C104 allow a separate control network to be used to assist in configuring, error handling and resetting of components connected in a system, even in the presence of errors on the data communications links in the network.

The IMS C104 has two bidirectional control links; **CLink0** and **CLink1**. They use the same electrical and packet level protocols as the communication data/strobe (DS) links (refer to chapter 6). Thus, an IMS C104 can be connected by its control link to a data DS link of a controlling IMS T9000 transputer and the IMS T9000 can issue commands to the IMS C104.

All communications with the controlling processor are via **CLink0.** The IMS C104 is programmed via commands along **Clink0**. **CLink1** provides a daisy–chain link, allowing a simple physical connectivity to be used for controlling networks.

The control links can be connected into a daisy chain or tree, with a controlling processor at the root. Figure 5.1 shows daisy–chained IMS C104's connected to one of the data DS links of a controlling IMS T9000 transputer, each IMS C104 has 32 data DS links but is shown as having just 5 links for clarity.
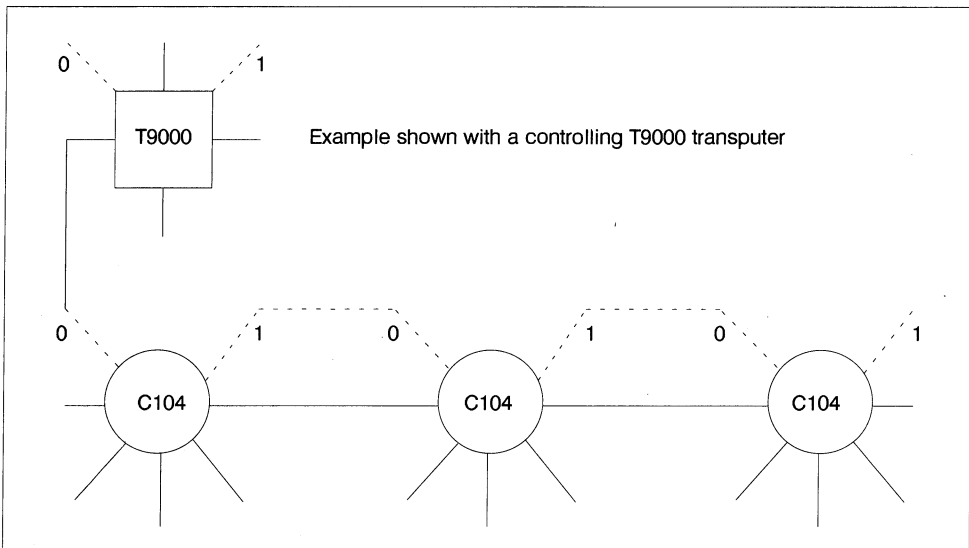
Figure 5.1 A daisy–chained control link network

Figure 5.2 gives an example of a daisy–chained control link network in which the IMS C104 is used to route control link packets from the control processor to the application network. In this example the controlled application network consists of IMS T9000 transputers, and three data DS links of the IMS C104 are connected to the control links of the application network to provide fan–out of the controlling system.

This provides a separate network of virtual channels between the root processor and the individual nodes of the application network. The control network is in effect a root node with a single virtual link to each node of the application network.

In order to establish the virtual channels between the root and each node, an identity and return address must be given to each node. The identity address is used to establish whether or not a packet arriving on **CLink0** is for that node and if not the message is forwarded down **CLink1** until it reaches its destination. Any output must be prefixed by the return header in order to identify the node of origin to the controlling process and to route the message through the IMS C104.

CLink1 is connected back to the IMS C104 by data DS link (**Link0**), and used to route messages back to the control processor.
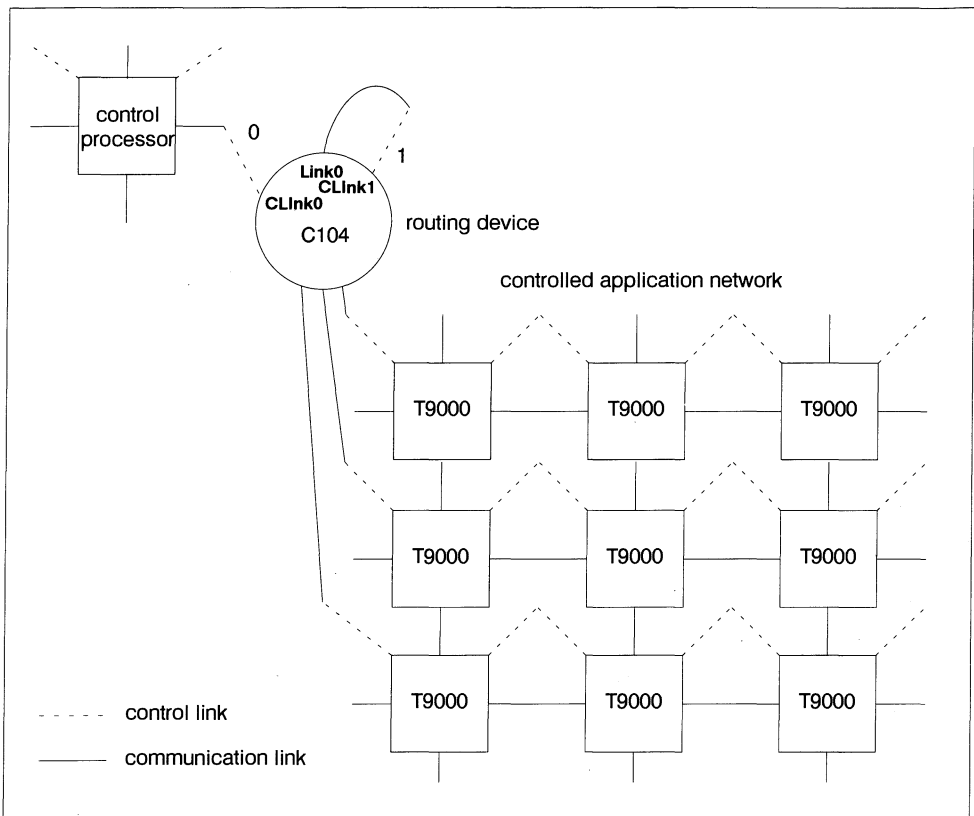
Figure 5.2   An IMS C104 providing fan–out.

5.1   Commands

A high level protocol is defined for the controlling network to allow the devices to issue commands to, and receive responses from, other devices in the network. Commands are sent as packets with the first byte after the header containing a command code, which may be followed by additional data. The following table details the command codes. Each command is terminated by an EOM token.

Command	Additional data	Function
Start	Return header	Allocates an identity and return header to each node. This must be the first command received following power on reset.
Reset	Level	Resets the IMS C104 to the given level (see chapter 7).
Identify	None	Returns the identity and the revision number of the device.
RecoverError	None	This command is used in error recovery on control system failure.
CPeek	Address	Returns the value stored at the given address in the device configuration space. If the address is invalid an invalid status is returned.
CPoke	Address, data	Writes data to the configuration space at the given address. If the address is invalid an invalid status is returned.
ErrorHandshake	None	Handshakes error message.

Table 5.1   Control link codes

Each command message is acknowledged by an acknowledge packet which is a packet containing no data and terminated by an EOP token. In addition the higher level control protocol requires that all command messages are acknowledged by a response message, in order to avoid deadlock, before the control process can send another command message to the same device. (However, *Start, Reset* and *Recover-Error* command messages may be sent to any node at any time to allow the control process to handle error conditions in the network.)

The response message can contain the result of a *CPeek* or *Identify* command, or it may be simply a handshake code corresponding to the command message. Each message is preceded by the return header and followed by an EOM token. Table 5.2 lists the response messages to each of the command messages. The data parameter 'Status' indicates whether or not there has been an error in performing the operation.

Response	Additional data
StartHandShake	None
ResetHandShake	Status
IdentifyResult	Device type and rev
RecoverHandShake	None
CPeekResult	Data, status
CPokeHandShake	Status
Error	Error code

Table 5.2   Control link responses

The error code indicates the cause of error as either;

- packet too short – for instance if the header length was set at 2 bytes and a packet consisting of a1 byte header and a terminator code was received then an error would occur.

- header out of range – if the header value received was not within the range of the interval selector.

- link error

- control link error – protocol

- control link error – command code

All the error codes must be handshaken from the root with the *ErrorHandShake* command.

5.2   Link speeds

After power–on the control links run at a default speed of 10 MHz; this can be changed by means of *CPokes*. The speed selection for control links is identical to that of the data DS links (see section 6.2), and the control links share the same master clock.

5.3   Control link configuration registers

The link module hardware in each control link is identical to that in each data link. An equivalent set of configuration bit fields is provided for each control link, as for the data links (see section 6.4).

6   Data/Strobe links

The IMS C104 has 32 links used for routing, and two control links which are used for monitoring and control purposes only. All of these links use a protocol with two wires in each direction, one for data and one to carry a strobe signal and are referred to as data/strobe (DS) links.

The links are TTL compatible and are series matched to 100 ohm transmission lines.

Each DS pair carries tokens and an encoded clock. The tokens can be data or control tokens. Figure 6.1 shows the format of data and control tokens on the data and strobe wires. Data tokens are 10 bits long and consist of a parity bit, a flag which is set to 0 to indicate a data token, and 8 bits of data. Control tokens are 4 bits long and consist of a parity bit, a flag which is set to 1 to indicate a control token, and 2 bits to indicate the type of control token.
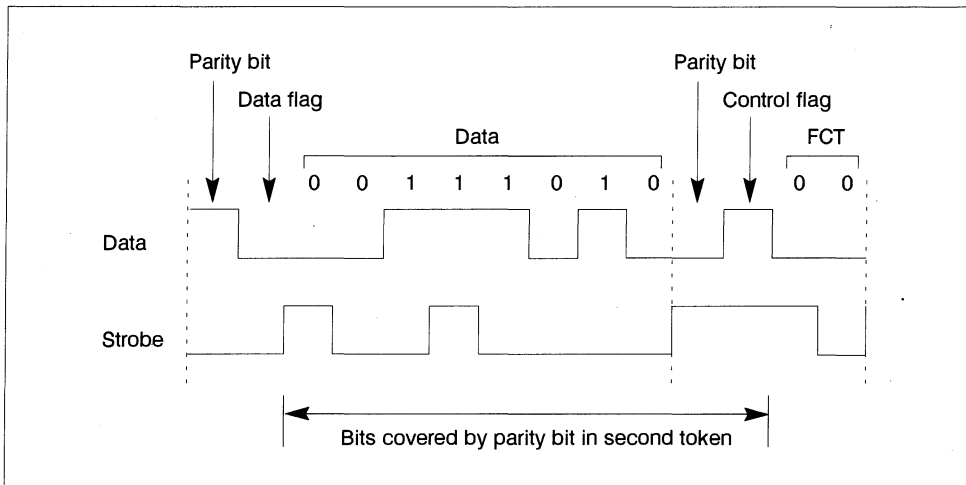
Figure 6.1   Link data format

The parity bit in any token covers the parity of the data or control bits in the previous token, and the data/control flag in the same token, as shown in figure 6.1. This allows single bit errors in the token type flag to be detected. **Odd** parity checking is used. To ensure the immediate detection of errors null tokens are sent in the absence of other tokens. The coding of the control tokens is shown in table 6.1.

Flow control token	FCT	P100
End of packet	EOP	P101
End of message	EOM	P110
Escape token	ESC	P111
Null token	NUL	ESC P100

Table 6.1   Control token codings

6.1   Low-level flow control

The DS link protocol separates the functions of flow control and process synchronization. Flow control is done entirely within the link module and process synchronization is built into a higher-level packet system.

Token-level flow control is performed in each link module, and the additional flow control tokens used are not visible to the higher-level packet protocol. The token-level flow control mechanism prevents a sender from overrunning the input buffer of a receiving link. Each receiving link input contains a buffer for at least 8 tokens (more buffering than this is in fact provided). Whenever the link input has sufficient buffering available to consume a further 8 tokens a FCT is transmitted on the associated link output, and this FCT gives the sender permission to transmit a further 8 tokens. Once the sender has transmitted a further 8 tokens it waits until it receives another FCT before transmitting any more tokens. The provision of more than 8 tokens of buffering on each link input ensures that in practice the next FCT is received before the previous block of 8 tokens has been fully transmitted, so the token-level flow control does not restrict the maximum bandwidth of the link.

6.2   Link speeds

The IMS C104 links can support a range of communication speeds, which are programmed by writing to registers using the *CPoke* command via control link **CLink0**. At reset all links are configured to run at the **BaseSpeed** of 10 Mbits/sec.

Only the transmission speed of a link is programmed as reception is asynchronous. This means that links running at different speeds can be connected, provided that each device is capable of receiving at the speed of the connected transmitter.

The transmission speed of all of the links on a given device are related to the speed of a single on-chip clock. The frequency of this master clock is programmed through the **SpeedMultiply** bit field described in section 6.4. The master frequency is divided down to obtain the transmission frequency for each link. The division factor can be programmed separately for each link via the **SpeedDivide** bit field described in section 6.4. For a given device, with a given programmed master clock frequency, this arrangement allows each link to be run at one of four transmission speeds, as shown in table 6.2.

SpeedMultiply	SpeedDivide				BaseSpeed
	/1	/2	/4	/8	
8	80	40	20	10.0	10
10	100	50	25	12.5	10
12	Reserved	60	30	15.0	10
14	Reserved	70	35	17.5	10
16	Reserved	80	40	20.0	10
18	Reserved	90	45	22.5	10
20	Reserved	100	50	25.0	10

Table 6.2   Link transmission speed in Mbits/sec

6.3   Errors on links

Link inputs detect parity and disconnection conditions as errors. A disconnection error indicates one of two things: either the link has been physically disconnected, or an error has occurred at the other end of the link which has then stopped transmitting. The bit fields **ParityError** and **DiscError** indicate when parity and disconnect errors occur.

The DS links are designed to be highly reliable within a single subsystem and can be operated in one of two environments, determined by the **LocalizeError** bit in each link.

In the majority of applications, the communications system should be regarded as being totally reliable. In this environment errors are considered to be very rare, but are treated as being catastrophic if they do occur. This environment is the default on power–on reset, with all links having their **LocalizeError** bit set to 0. If an error occurs it will be detected and reported via a message sent along **CLink0**. Normal practice will then be to reset the subsystem in which the error has occurred and to restart the application.

For some applications, for instance when a disconnect or parity error may be expected during normal oper- ation, an even higher level of reliability is required. This level of fault tolerance is supported by localizing errors to the link on which they occur, by setting the **LocalizeError** bit of the link to 1. In addition a *data link layer* process must be connected to each virtual channel associated with the link. These processes are responsible for establishing and maintaining a higher level flow control, using time-out to detect that a message has not completed, and requesting retransmission. If an error occurs, packets in transit at the time of the error will be discarded or truncated.

For information on *the data link layer* refer to chapter 4 of *'Computer Networks' by Andrew S. Tanenbaum, published by Prentice–Hall International (ISBN: 0–13–166836–6).*

6.4   Link configuration registers

The links are controlled via registers accessed via the control link (see chapter 4).

Each link has three registers, the **LinkMode** register, **LinkCommand** register and **LinkStatus** register.

In addition the configuration space contains the **DSLinkPLL** register which contains the **SpeedMultiply** bit. This takes the 5 MHz input clock and multiplies it by a programmable value to provide the root clock for all the DS links.

The tables below describe the functionality of the DS links to be controlled, and the associated bit fields in the configuration registers.

Bit	Bit field	Function
5:0	SpeedMultiply	Sets DS link master clock to required value (see table 6.2).

Table 6.3   Bit fields in the **DSLinkPLL** register

The **Link0–3Mode** registers power up into a default state and may be re–programmed before or after the link has been started.

Bit	Bit field	Function
1:0	SpeedDivide	Sets transmit speed of the **Link0–3** (see table 6.2). 00 = /1, 01 = /2, 10 = /4, 11 = /8
2	SpeedSelect	Sets the **Link0–3** to transmit at the speed determined by the **SpeedDi- vide** bits as opposed to the base speed of 10 Mbits/s.
3	LocalizeError	Packets in transit at the time of an error will be discarded or truncated. When set false communication on the link stops until the link is reset.

Table 6.4   Bit fields in the **Link0–3Mode** registers

The **Link0–3Command** registers are write only and contain four bits which when set cause a specific action to be taken by the DS link.

Bit	Bit field	Function
0	ResetLink	Resets the link engine of the **Link0-3**. The token state is reset, the flow control credit is set to zero, the buffers are marked as empty, and the parity state is reset.
1	StartLink	When a transition from 0 to 1 occurs **Link0-3** will be initialized and commence operation.
2	ResetOutput	Sets both outputs of **Link0-3** low.
3	WrongParity	The **Link0-3** output will generate incorrect parity. This may be used to force a parity error on the transputer at the other end of the **Link0-3**.

Table 6.5   Bit fields in the **Link0-3Command** registers

The **Link0-3Status** registers are read only and contain six bits which contain information about the state of the DS link.

Bit	Bit field	Function
0	LinkError	Flags that an error has occurred on the **Link0-3**.
1	LinkStarted	Flags that the output **Link0-3** has been started and no errors have been detected.
2	ResetOutputCom-plete	Flags that **ResetOutput** has completed on the **Link0-3**.
3	ParityError	Flags that a parity error has occurred on the **Link0-3**.
4	DiscError	Flags that a disconnect error has occurred on the **Link0-3**.
5	TokenReceived	Flags that a token has been seen on the **Link0-3** since **ResetLink**.

Table 6.6   Bit fields in the **Link0-3Status** registers

7   Levels of reset

The IMS C104 can be reset to a given level using the *Reset* command or **Reset** pin. The different levels of reset are described below.

A reset results in any packets currently being routed within the IMS C104 being lost, except for a *Reset3* command which has no effect on the IMS C104.

7.1   Level 0 – hardware reset

The network can be returned to level 0 by taking all the **Reset** pins in the network high.

After a hardware reset each IMS C104 is in the following state:

All the (data and control) links are in Wait state with a default speed of 10 MHz. The identity and return headers for the control links are undefined. All registers are undefined and contain their default values. The packet processors are inactive.

7.2   Level 1 – labelled control network

The network can be reset to level 1 by sending a *Reset1* command message to each IMS C104.

This level of reset leaves the identity and return headers unaltered and all connected control links remain operational. All the data links are in Wait state with a default speed of 10 MHz. All registers are reset to their level 0 default values. All data in the IMS C104 is lost.

7.3   Level 2 – configured network

The network can be reset to level 2 by sending a *Reset2* command message to each IMS C104.

At this level of reset the identity and return headers are unaltered and register contents are unaffected. All data in the IMS C104 is lost. The data links are reset and returned to the Wait state. The packet processors are deactivated.

7.4   Level 3

Reset level 3 is invalid on the IMS C104. If a *Reset3* command message is received from an IMS T9000 transputer it is handshaken with status set to false.

8 Software

8.1 IMS T9000 configuration tools

A set of tools is available to support the configuration of IMS T9000 systems. The tools will, among other things, provide support for the configuration and initialization of networks consisting of IMS T9000 processors and IMS C104 routing switches.

The tools will be able to set the attributes of each device in the network by sending initialization data down the control link, and will set the processors into a state ready to receive an application down the data links.

A Network Description Language (NDL) is used to describe networks of devices and the labeling of IMS C104s, and will allow the specification of values for all the attributes of a device.

The Network Description Language will support the following:

- declaration of processors, IMS C104 routing chips and their interconnections.

- specification of attributes for IMS C104 routing chips; including interval settings, header deletion and randomization characteristics.

- the construction of the control system, including chains of devices plus a predefined method of using the IMS C104 as a fan-out. It is possible to calculate the IMS C104 attributes (including interval values) for such devices used in the control system.

- desired message routing paths.

From the NDL file the initialization tools produce a file containing the network initialization data. This data is sent down the control link to the network.

Once the network has been initialized, programs are built and loaded to the network in the same way as for T2/T4/T8-series processors.

9   Preliminary pin designations

Pin	In/Out	Function
VCC, GND		Power supply and return
CapPlus, CapMinus		External capacitor for internal clock power supply
ClockIn	in	Input clock
Reset	in	System reset

Table 9.1   IMS C104 system services

Pin	In/Out	Function
LinkInData0–31	in	Link input data channels
LinkInStrobe0–31	in	Link input strobes
LinkOutData0–31	out	Link output data channels
LinkOutStrobe0–31	out	Link output strobes
CLinkInData0–1	in	Control link input data channel
CLinkInStrobe0–1	in	Control link input strobe
CLinkOutData0–1	out	Control link output data channel
CLinkOutStrobe0–1	out	Control link output strobe

Table 9.2   IMS C104 links
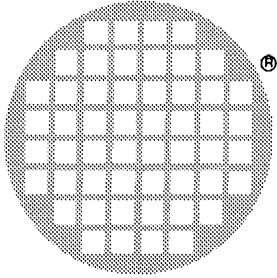
IMS C100
system protocol
converter

inmos®

Preliminary Information

FEATURES

Communicates between T2xx/T4xx/T8xx and T9000 transputers

T2/T4/T8–series and T9–series data link protocol

T2/T4/T8–series and T9–series control protocol

Converts data and control protocols

Four modes of operation:

Mode 0: Enables a single T9–series transputer to be used in a T2/T4/T8–series network

Mode 1: Enables a T2/T4/T8–series transputer system to use a T9–series subsystem

Mode 2: Enables a T9–series transputer system to use an existing T2/T4/T8–series subsystem without modifications to the T2/T4/T8 software

Mode 3: Enables a T9–series transputer system to use an existing T2/T4/T8–series subsystem, and enables a T2/T4/T8–series transputer to emulate a T9–series transputer

This is preliminary information on a product under development and product details may change.

CLink 0	Control conversion unit	CLink 1
OS Link 0	Data conversion unit	DS Link 0
OS Link 1	Data conversion unit	DS Link 1
OS Link 2	Data conversion unit	DS Link 2
OS Link 3	Data conversion unit	DS Link 3

System services
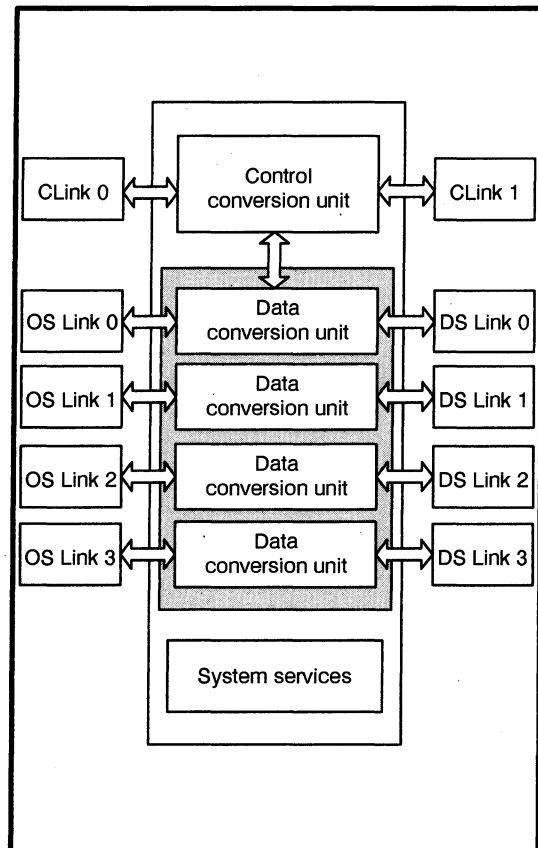
1   Introduction

This document contains preliminary information for the IMS C100 system protocol converter.

The IMS C100 is part of a new product family based around the IMS T9000 transputer, referred to as the 'T9-series'. The current family of T2xx/T4xx/T8xx transputers are referred to as 'T2/T4/T8-series'.

T9-series transputers are binary compatible with T2/T4/T8-series transputers. However T9-series transputers have different physical links and data protocols than T2/T4/T8-series transputers. The IMS C100 is a system protocol converter which converts between these protocols. It allows mixed systems, consisting of both T9-series and T2/T4/T8-series transputers, to be constructed.

T2/T4/T8-series transputer links consist of two wires, one in each direction, and use an asynchronous bit-serial (byte-stream) protocol. Each bit received is sampled five times and hence the links are referred to as oversampled (OS) links. Each link provides a pair of channels, one in each direction and can operate at up to 20 MBits/sec, providing a bidirectional bandwidth of 2.4 MBytes/sec.

T9-series transputer links consist of four wires, two in each direction, one carrying data and one carrying a strobe. The links are therefore referred to as data-strobe (DS) links. Each link can operate at up to 100 MBits/sec, providing a bidirectional bandwidth of 20 MBytes/sec. The DS link protocol supports *virtual channels* and *dynamic message routing,* and provides a high data bandwidth.

T2/T4/T8-series transputers are controlled by means of **Reset**, **Analyse** and **Error** pins on each device and are inspected and booted by means of a special protocol on their links. On T9-series transputers this is achieved by special links, called control links.

The IMS C100 provides an inter-networking solution for transputer systems, allowing systems to be constructed using the optimum mix of transputers, for processing power, communication bandwidth and system cost.

The IMS C100 converts both data and control protocols of T9-series transputer systems to those of T2/T4/T8-series, and vice versa. It is intended to be used in conjunction with software running on either T9-series or T2/T4/T8-series transputers and can operate in one of four modes.

This document describes the operation of the IMS C100 in detail, and summarizes the background information necessary to understand the full implications of each mode of operation.

2 IMS C100 modes of operation

This chapter describes the modes of operation of the IMS C100 and gives examples of its use in each mode. For a complete understanding of the implications of this chapter consult chapters 3 and 4, which describe the link and control protocols of T2/T4/T8–series and T9–series components, and how the IMS C100 converts between these protocols.

The four modes of operation of the IMS C100 are listed below:

Mode 0: enables a T9–series transputer with ROM, from which the transputer boots, to emulate a T2/T4/T8–series transputer.

Mode 1: enables a T2/T4/T8–series system to use a T9–series subsystem.

Mode 2: enables a T9–series system to use an existing T2/T4/T8–series subsystem without any modification to the existing T2/T4/T8–series software.

Mode 3: enables a T9–series system to use an optimum T2/T4/T8–series subsystem and enables a T2/T4/T8–series transputer to emulate a T9–series transputer.

2.1 Mode pins

The IMS C100 has two mode pins (**Mode0–1**) which must be set at power–on. These pins determine which type of conversion is to be performed between the data links, which system interface is regarded as master, and whether **OSLink0** has special initial behavior. In modes 2 and 3 **OSLink0** is usurped to generate the pre–boot protocol of the T2/T4/T8–series transputer until the transputer is booted (refer to section 4.3.3 for further information). Table 2.1 details the mode settings.

The OS link protocol synchronizes the communications of each byte of data, and hence the term *byte-stream* protocol has been adopted. DS links use a high level packet protocol and hence the term *packetized* protocol has been adopted. Each IMS T9000 transputer DS link may be set to operate in virtual channel mode or in byte mode (see section 3.2.1). The IMS T9000 DS links operating in byte mode, in conjunction with an IMS C100, convert the DS links to the byte stream protocol.

Mode1	Mode0	Mode	Conversion type	System master	OSLink0
Low	Low	0	Byte–stream	**Reset, Analyse, Error**	Not special
Low	High	1	Packetized	Control link 0	Not special
High	Low	2	Byte–stream	Control link 0	Special
High	High	3	Packetized	Control link 0	Special

Table 2.1 **Mode0–1** pins

The behavior of the IMS C100 is undefined if the mode pins are changed after reset.

2.2   Mode 0: Enables a single T9-series transputer to be used in a T2/T4/T8-series network

The purpose of this mode is to allow a single IMS T9000 transputer to operate as a fast IMS T805.

Connect control link **CLink0** of the IMS C100 to **CLink0** of the IMS T9000 transputer, connect the four data DS links (**DSLink0–3**) of the IMS C100 to the four data links of the IMS T9000, and set the IMS C100 into mode 0, as shown in figure 2.1. The combination of the IMS C100 and the IMS T9000 transputer has **Reset**, **Analyse** and **Error** pins.
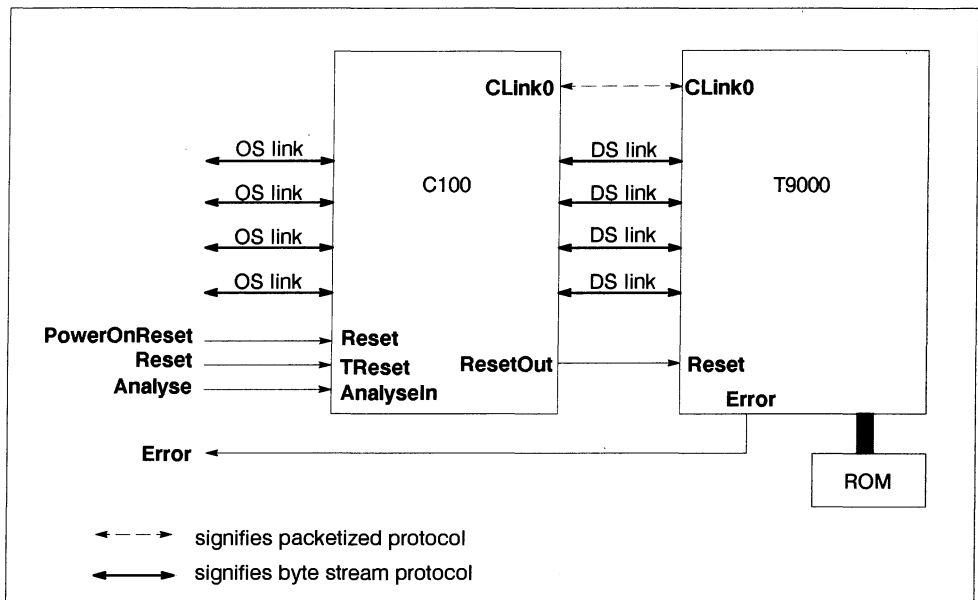

Figure 2.1   Mode 0 – converting an IMS T9000 transputer for use in a T2/T4/T8–series network

The **StartFromROM** pin on the IMS T9000 transputer must be set high so that the IMS T9000 transputer boots from ROM. The ROM software configures the IMS T9000, and sets the IMS T9000 data links into byte mode, so that they interact with the IMS C100 DS links operating in byte-stream conversion mode to generate the T2/T4/T8–series transputer protocol on the OS links of the IMS C100. The software, which will be supplied to customers, also emulates the pre-boot protocol of T2/T4/T8–series transputers.

The **TReset** pin indicates transputer reset of the connected T2/T4/T8–series transputer. If the **TReset** pin of the IMS C100 is asserted with **AnalyseIn** low, the IMS C100 is reset, and the signal is reproduced on **ResetOut**, which causes the IMS T9000 to be reset also. When the **Reset** pin on the IMS T9000 goes low execution is restarted from ROM.

The **Reset** pin is provided in this case for systems which separate power–on reset from transputer reset. When the **Reset** pin is asserted it always causes a reset of both the IMS C100 and the attached IMS T9000 (by being reproduced on **ResetOut**).

The **TReset** and **AnalyseIn** signals are used in this mode only and are ignored in modes 1, 2 and 3.

2.3 Mode 1: Enables a T2/T4/T8-series system to use a T9-series subsystem

The purpose of this mode is to allow a T9-series subsystem to be connected to, and controlled from, a T2/T4/T8-series network.

Communication is in the packetized protocol, and software must be run on the T2/T4/T8-series system to interface the packetized protocol, and to control the T9-series subsystem.

To enable a T2/T4/T8-series system to use an T9-series subsystem set the IMS C100 to mode 1, and connect one or more OS links from the T2/T4/T8-series system to the OS data links of the IMS C100. Since T9-series systems are controlled entirely via links this enables T9-series subsystems to be configured, booted, reset and analyzed from a T2/T4/T8-series system. An example network is shown in figure 2.2. The **RAE** signals to the T2/T4/T8-series network are shown by the dotted line. The IMS C004 programmable link switch has 32 links, of which only six are shown in this example.

Note that, by 'looping back' through the control links of the IMS C100, the T2/T4/T8-series system obtains full control of the device. Note, however, that the IMS C100 must be given its identity before any of the devices in the T9-series subsystem.
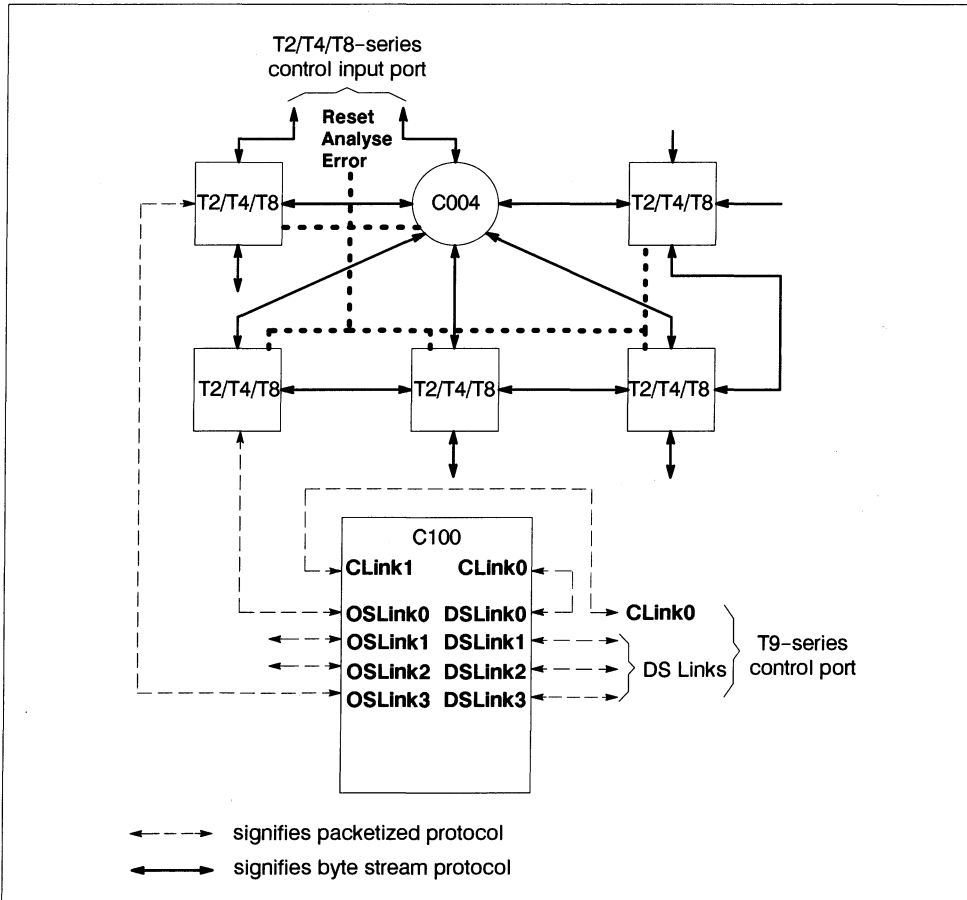

Figure 2.2   Mode 1 – T2/T4/T8-series system using T9-series subsystems

2.4   Mode 2: Enables a T9–series system to use an existing T2/T4/T8–series subsystem

The purpose of this mode is to allow T9–series systems to use an existing T2/T4/T8–series subsystem, without having to change either the hardware or the software of the T2/T4/T8–series subsystem. For example, a SCSI TRAM purchased as a functional subsystem from a third party supplier (including both hardware and the associated software drivers) can be used unmodified as a subsystem to a T9–series system. Thus this mode protects users existing investment in transputer–based equipment.

Figure 2.3 shows how a T2/T4/T8–series control port can be provided using an IMS C100 in mode 2. Each IMS C104 packet routing switch has 32 data links, of which only seven are shown in this example. Note that the data DS links of the IMS C100 must be connected directly to IMS T9000 data links set into byte mode, and **cannot** be connected to an IMS C104 packet routing switch.

The T2/T4/T8–series subsystem is controlled via **CLink0** of the IMS C100. After power–on, commands sent along **CLink0** are converted to the appropriate T2/T4/T8–series byte sequences which are sent along **OSLink0** of the IMS C100. This allows the memory of transputers in the T2/T4/T8–series subsystem to be peeked and poked, and for it to be booted.

Assertion of the **AnalyseOut** and **ResetOut** pins results in the **Reset** and **Analyse** pins of the connected T2/T4/T8–series transputer being asserted, enabling it to be stopped and analyzed.
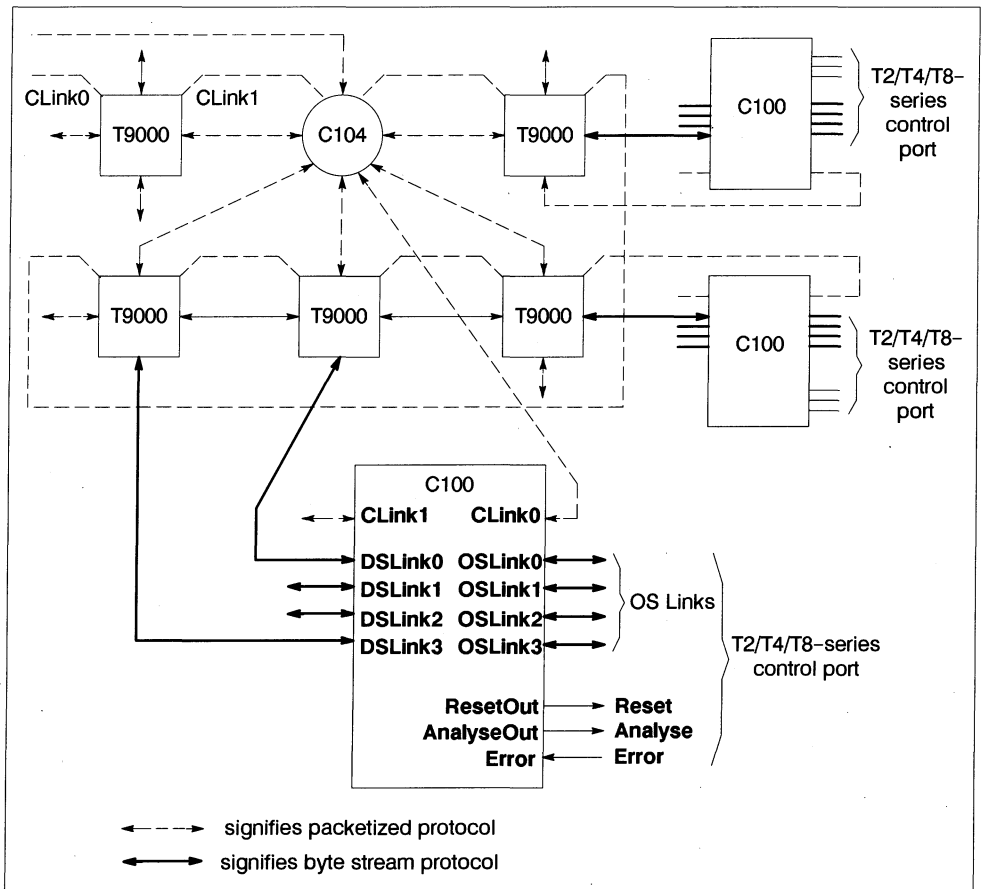
Figure 2.3   Mode 2 – T9–series system using existing T2/T4/T8–series subsystems

2.5   Mode 3: Enables a T9-series system to use a T2/T4/T8-series subsystem

The purpose of this mode is to allow T9-series systems to be built which use T2/T4/T8-series subsystems, enabling systems to be built using the optimum mix of transputers with regard to cost and performance.

Communication is in the packetized protocol. Thus the data DS links of the IMS C100 can be connected directly to an IMS C104 packet routing switch, as in figure 2.4.

Software to interface to the packetized protocol must be run on all T2/T4/T8-series links connected to the IMS C100.
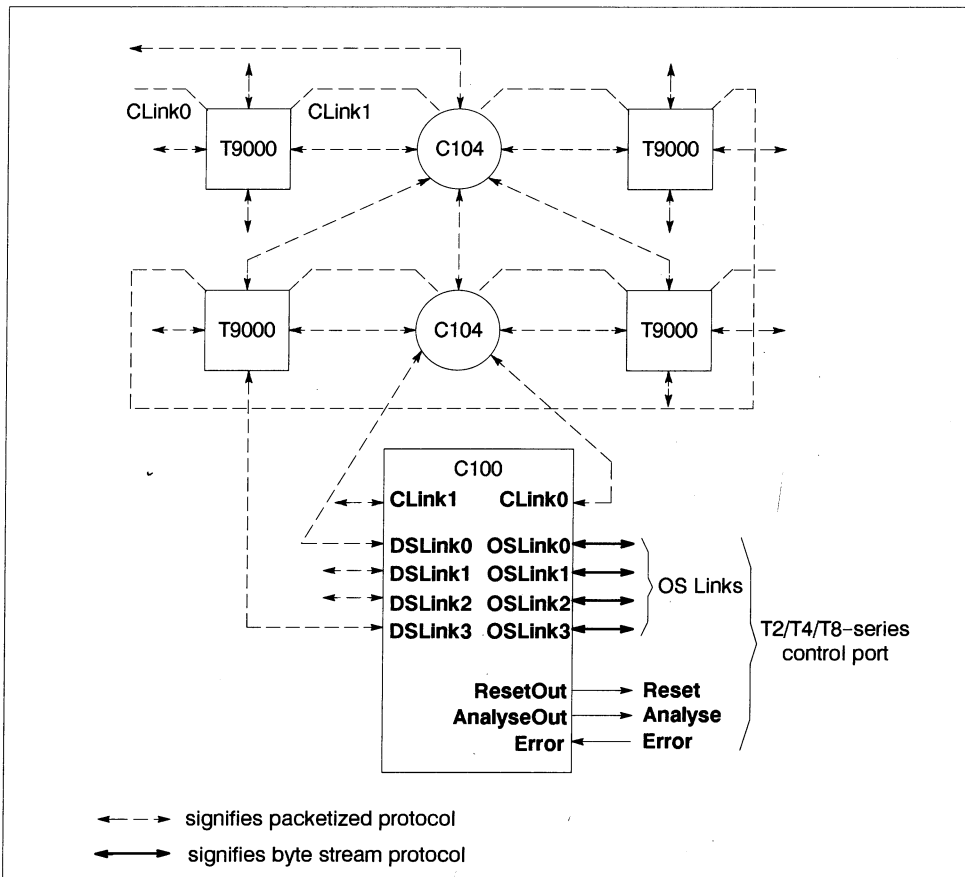
Figure 2.4   Mode 3 – T9-series system using optimum T2/T4/T8-series subsystems

The T2/T4/T8-series subsystem is controlled via **CLink0** of the IMS C100. Messages into **CLink0** of the IMS C100 cause individual links to be reset, and the **ResetOut** and **AnalyseOut** pins to be toggled. Assertion of the **AnalyseOut** and **ResetOut** pins results in the **Reset** and **Analyse** pins of the connected T2/T4/T8-series transputer being asserted, enabling it to be stopped and analyzed.

An error from within the IMS C100 and a signal on the **Error** pin both cause an *Error* message to be sent from **CLink0**.

The IMS C100 operating in mode 3 enables a T2/T4/T8–series transputer to emulate an IMS T9000 transputer. This is achieved by connecting the **ResetOut**, **AnalyseOut**, and **Error** pins of the IMS C100 to the **Reset**, **Analyse**, and **Error** pins of the T2/T4/T8–series transputer and setting the IMS C100 into mode 3. This combination of the IMS C100 and the T2/T4/T8–series transputer has a control link 0 (**CLink0**), and a control link 1 (**CLink1**) for daisy–chaining. Figure 14.2 shows a T2/T4/T8–series transputer being converted to an IMS T9000 interface, with the T2/T4/T8–series transputer being booted from a link. Software must be run on the T2/T4/T8–series transputer to convert the OS links to the packetized protocol.

Figure 2.5   Mode 3– converting a T2/T4/T8–series transputer for use in a T9–series network

3   Link protocols and link protocol conversion

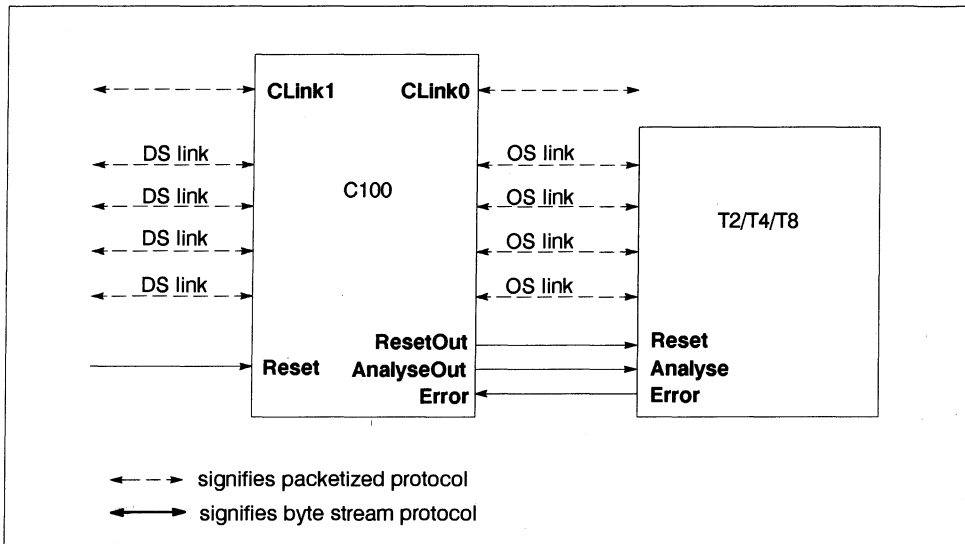This chapter describes the different link protocols used on T2/T4/T8–series and T9–series components, and the two types of conversion between the link protocols that the IMS C100 supports.

3.1   T2/T4/T8 series oversampled links

T2/T4/T8–series transputer links consist of two wires, one in each direction, and use an asynchronous bit-serial protocol. Link inputs are sampled five times in each bit period, and hence the links are referred to as oversampled (OS) links.

Messages are transmitted as a sequence of single byte communications, each of which must be acknowledged. The acknowledge packets are used both to signal reception of the data bytes and to maintain flow control.

A link provides a pair of channels, one input and one output channel. Every byte of data sent on an output channel is acknowledged on the input channel of the same link, thus each signal line carries both data and control information.

Each data byte is transmitted as a high start bit followed by another high bit followed by eight data bits followed by a low stop bit, as shown in figure 3.1. The least significant bit of data is transmitted first. After transmitting a data byte the sender waits for the acknowledge, which consists of a high start bit followed by a zero bit. The acknowledge signifies that the receiving link is able to receive another byte.

The receiving transputer can send an acknowledge as soon as the data has been identified (provided there is sufficient buffer space for another data byte, and that an inputting process is ready to receive the data byte) so that communications can be continuous.

The link protocol synchronizes the communications of each byte of data, and hence the term *byte–stream* protocol has been adopted. As the protocol supports the transmission of an arbitrary sequence of bytes transputers of different word lengths can be connected together.
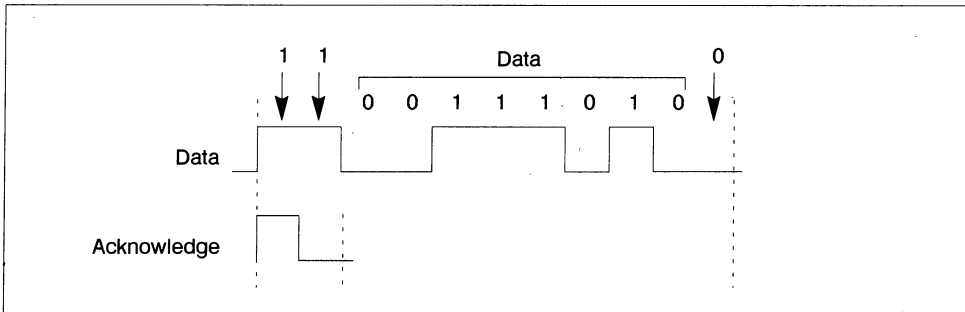
Figure 3.1   OS link data and acknowledge formats

The T2/T4/T8–series transputer family includes link adaptor devices, the IMS C011 and IMS C012, which enable OS links to interface with non–transputer devices.

3.2   T9 series data/strobe links

T9–series transputer links consist of four wires, two in each direction, one for data and one to carry a strobe signal. These links are therefore referred to as data/strobe (DS) links.

Communication between processes on one IMS T9000 transputer takes place over software channels. Communication between processes on different processors takes place over virtual channels. Virtual channels are multiplexed onto each physical link by a communications processor within the IMS T9000.

The data links support a physical link protocol to support virtual channels and dynamic message routing, and to provide a high data bandwidth.

Each message is split into a sequence of packets, each of which has the structure shown in figure 3.2. Packets from different messages may be interleaved over each physical link. Interleaving packets from different messages allows any number of processes to communicate simultaneously via each physical link. Communication channels can be established between any two processes regardless of where they are physically located, or whether the channels are routed through a network. Thus, programs can be independent of network topology.

header	packet body	packet terminator

Figure 3.2   Structure of a packet on DS links

In order that packets which are parts of different messages can be distinguished by the transputer which receives them, each packet contains a one or two byte header which identifies a virtual input channel of the receiving transputer. 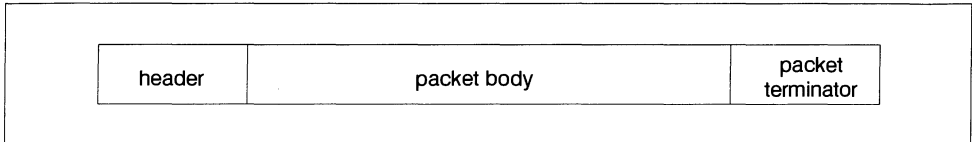The packet header is also used to route the packet through a network. Bytes following the header are treated as the data section of the packet until a packet termination token is received. A packet termination token is either an EOP (end of packet) token or an EOM (end of message) token.

The maximum length of data in each packet which the IMS T9000 can transmit is 32 bytes. All but the last packet of a message contains the maximum amount of data; the last contains the maximum amount of data or less.

The communications processor within the IMS T9000 enforces a high-level protocol on each virtual channel. To maintain synchronized communication, and to ensure that no data is lost, each packet of data sent along a virtual channel must be acknowledged before the next is sent. The last packet must be acknowledged before the outputting process is rescheduled. Data packets on a virtual channel are acknowledged by the communications processor by sending acknowledge packets on another virtual channel back to the processor which sent them. Acknowledge packets are packets containing no data and which are always terminated by an EOP token. The acknowledge packets perform packet-level flow-control and process synchronization.

Virtual channels always occur in pairs between pairs of communicating processors, with one virtual channel in each direction. If a message is being communicated in one direction the virtual channel in the opposite direction is used to return acknowledge packets to the sender. The associated pair of virtual channels is referred to as a *virtual link*. A virtual link can transfer messages in both directions at the same time with data packets and acknowledge packets being interleaved on both of the virtual channels. Because virtual channels are always paired in this way it is not necessary to include source information in the packets. Thus packet headers need only represent their destinations.

Figure 3.3 shows the format of data and control tokens on the data and strobe wires. Data tokens are 10 bits long and contain a parity bit, a flag which is set to 0 to indicate the presence of a data token, and 8 bits of data. Control tokens are 4 bits long and contain a parity bit, a flag which is set to 1 to indicate the presence of a control token, and 2 bits to indicate the type of control token.
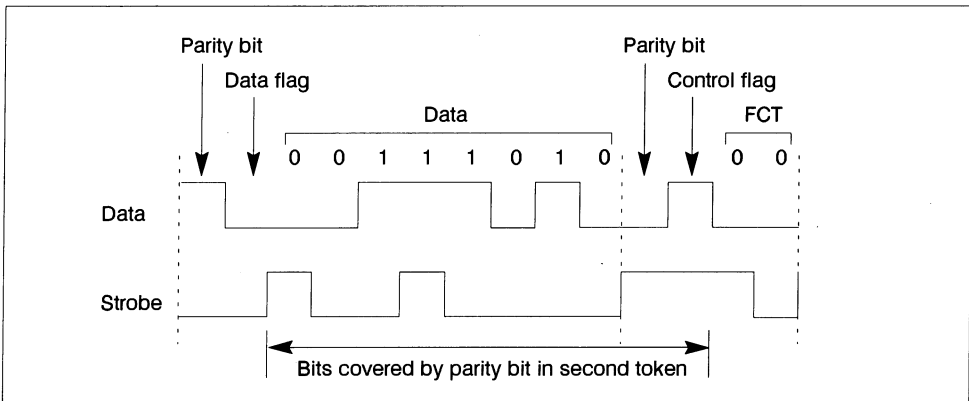
Figure 3.3   DS link data format

The parity bit in any token covers the parity of the data or control bits in the previous token, and the data/control flag in the same token, as shown in figure 3.3. This allows single bit errors in the token type flag to be detected. **Odd** parity checking is used. To ensure the immediate detection of errors null tokens are sent in the absence of other tokens. The coding of the control tokens is shown in table 3.1.

Flow control token	FCT	P100
End of packet	EOP	P101
End of message	EOM	P110
Escape token	ESC	P111
Null token	NUL	ESC P100

Table 3.1   Control token codings

The DS-link protocol separates the functions of flow control and process synchronization. Token-level flow control is performed in each link module, and the additional flow control tokens used are not visible to the higher-level packet protocol. The token-level flow control mechanism prevents a sender from overrunning the input buffer of a receiving link.

Each receiving link input contains a buffer for at least 8 tokens (more buffering than this is in fact provided). Whenever the link input has sufficient buffering available to consume a further 8 tokens (consisting of data and EOP or EOM tokens) a FCT is transmitted on the associated link output, and this FCT gives the sender permission to transmit a further 8 tokens. Once the sender has transmitted a further 8 tokens it waits until it receives another FCT before transmitting any more tokens. The provision of more than 8 tokens of buffering on each link input ensures that in practice the next FCT is received before the previous block of 8 tokens has been fully transmitted, so that the token-level flow control does not restrict the maximum bandwidth of the link.

DS links use a high level packet protocol and hence the term *packetized* protocol has been adopted.

3.2.1   Byte mode

Each IMS T9000 data DS link may be set to operate either in virtual channel mode or in byte mode. Byte mode is provided to allow IMS T9000 DS links to communicate with OS links carrying the byte-stream protocol via an IMS C100. The mode is set for each IMS T9000 link **Link0-3** by the **ByteMode0-3** bit fields in the configuration registers (as described in the *IMS T9000 Preliminary Information*). Setting the IMS T9000 links independently of each other, enables each IMS T9000 transputer to be connected to several different networks.

3.3   Data protocol conversion

The IMS C100 is able to convert between the OS and DS link protocols in two ways:

Byte stream conversion: This is where the message/ packet level is removed from the DS links.

The DS link of a connected T9–series transputer is set to byte mode. A pair of channels is then supported from the T9–series transputer, through the IMS C100, to a T2/T4/T8–series transputer. Software on the T2/T4/T8–series transputer sees the channels as being identical to that through a normal OS link. No modification to the T2/T4/T8–series transputer software is needed.

Packetized conversion: This is where the message/ packet level is added to the OS links.

A process must be run on the connected T2/T4/T8–series transputer to impose a software packet protocol onto the OS link. This is converted to the hardware supported packet protocol on the DS link by the IMS C100.

The IMS C100 data DS and OS links are paired, and all pairs perform one or other type of conversion, depending on the mode. In modes 1 and 3, all four link pairs convert the packetized protocol; in modes 0 and 2, all four convert the byte–stream protocol.

The two types of conversion are described in more detail below. Each pair of data links functions in the same way and the following sections describe the action of one pair in each of the two types.

3.3.1   Byte–stream conversion

The OS link of the IMS C100, operating in byte–stream mode, is identical to an OS link on a T2/T4/T8–series component. No modification to software running on a connected T2/T4/T8–series transputer is needed.

The DS link of the connected T9–series transputer must be set in *byte mode* and connected to the DS link of the IMS C100. The IMS C100 cannot be directly connected to an IMS C104 when this type of conversion is being used. The IMS T9000 DS link set in byte mode is able to send and receive single bytes. Software on the T9–series transputer will send and receive messages normally, via a pair of channels.

A special protocol is used between the IMS C100 and the T9–series transputer. This protocol is invisible to the user, and is described here for completeness. Data is transferred along the DS link in the form of packets each with a single byte header. Each packet is terminated with either an EOP or EOM token.

The IMS C100 interprets packets from the T9–series transputer as indicated in table 3.2. Note: the DS links of an IMS T9000 transputer which have been set into byte mode generate this protocol automatically.

Header	Data	Terminator	Interpretation	Notes
0	32 bytes	EOP	Part of message	
0	1–32 bytes	EOM	End of message	
0	none	EOP	Acknowledgement	
1	1–4 bytes	EOM	Input count	1
1	none	EOM	Reset link	2

Notes

1   The IMS C100 knows the length of a message from the IMS T9000 to the T2/T4/T8–series transputer as this is indicated by an EOM token. In order for it to know the length of a message from the T2/T4/T8–series to the IMS T9000 transputer the IMS T9000 must tell it explicitly. The 'input count' packet contains the count of the data bytes to be transferred from the OS link to the DS link of the IMS C100.

2   The 'reset link' packet is sent whenever an IMS T9000 link in byte mode is reset. Its effect is to cause the reset of the associated OS link (see Reset chapter 7).

Table 3.2   Packets from IMS T9000 to IMS C100

The IMS C100 can respond to both data bytes and acknowledges on the OS links immediately by buffering data from the IMS T9000 and holding a count of the input length, thus maintaining full bandwidth.

The IMS C100 sends packets along the DS link, as shown in table 3.3. Note: DS links of an IMS T9000 transputer which have been set into byte mode accept this protocol automatically.

Header	Data	Terminator	Interpretation	Notes
0	32 bytes	EOP	Part of message	
0	1–32 bytes	EOM	End of message	
0	none	EOP	Acknowledgement	1
0	none	EOM	Unsolicited byte	2

Notes

1   The acknowledgement packet is sent when the IMS C100 is ready to receive more data.

2   If a byte is received from the OS link whilst the output count is zero, the count is effectively reduced to –1 and an unsolicited packet is sent.

Table 3.3   Packets from IMS C100 to IMS T9000

3.3.2   Packetized conversion

This conversion type allows software on a connected T2/T4/T8–series transputer to use virtual channels to communicate with processes in the connected T9–series system. The IMS C100 can be directly connected to an IMS C104 when this type of conversion is being used.

With packetized conversion the DS links of the IMS C100 are operationally identical to the DS links of T9–series transputers.

Software must be run on the connected T2/T4/T8–series transputer to:

1   Packetize the messages output from the T2/T4/T8–series transputer, according to the protocol described below.

2   Interpret the packetized messages arriving on the T2/T4/T8–series transputer.

The IMS C100 converts packets between the software supported protocol on the OS link and the hardware supported packetized protocol on the DS link.

The length of packets in T9–series DS links is indicated by terminator codes EOP or EOM. In order for the IMS C100 to determine the length of the packet on OS links, the length must be given explicitly as an unsigned 8 bit value ('count byte') at the start of the packet.
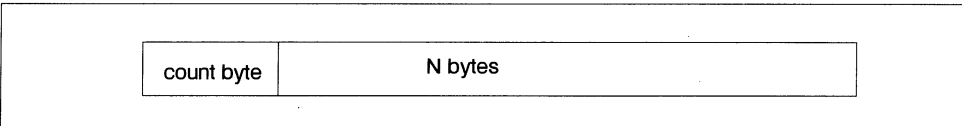
Figure 3.4   Structure of a software supported packet on OS links

The first byte of a (software supported) packet on an OS link is defined to be a count byte as in figure 3.5.
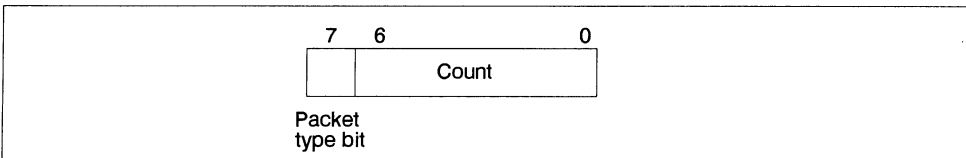
Figure 3.5   Structure of a count byte in an OS link packet.

If the packet type bit in the count byte is 0 then the packet is equivalent to a DS link packet terminated by an EOP token. If the packet type bit is 1 then the packet is equivalent to a DS link packet terminated by an EOM token.

Received on OS link of IMS C100	Packet type bit in count byte	Transmitted on DS link of IMS C100
(Count byte) (N bytes)	0	(N bytes) EOP
(Count byte) (N bytes)	1	(N bytes) EOM

Table 3.4   Packets from T2/T4/T8 to IMS C100

The packet level of the DS link protocol is represented in the OS link protocol for different representations of a packet containing N bytes, including the header, see figure 3.6.
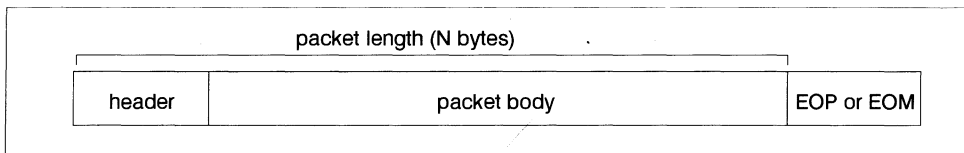
Figure 3.6   Structure of a packet on DS links

Packets received on the DS link, from the connected T9–series component, which are less than 7 bytes have zero bytes added, making 8 bytes in all including the count byte. This improves the efficiency of software running on T2/T4/T8–series transputers. The extra bytes added must be discarded by the T2/T4/T8–series transputer inputting software.

Note that these transformations are independent of details of the structure of the packet, such as the header length.

4   Control protocols and control protocol conversion

This chapter describes the different control protocols used on T2/T4/T8-series and T9-series components, and the two types of control protocol conversion that the IMS C100 supports.

4.1   T2/T4/T8 type control

T2/T4/T8-series transputers are controlled by means of **Reset**, **Analyse** and **Error** pins (RAE) on each device, and are inspected and booted by means of a special protocol across the transputer links (T2/T4/T8-type control).

The falling edge of **Reset** initializes the transputer, triggers the memory configuration sequence and starts the bootstrap routine.

The processor and the OS links start after reset. The transputer obeys a bootstrap program which can either be in off-chip ROM or can be received from one of the links.

A software error, such as arithmetic overflow, array bounds violation or divide by zero, causes an error flag to be set in the transputer processor. The flag is directly connected to the **Error** pin. Both the flag and the pin can be ignored, or the transputer stopped. Stopping the transputer on an error means that the error cannot cause further corruption.

As well as containing the error in this way it is possible to determine the state of the transputer and its memory at the time the error occurred.

If **Analyse** is taken high when the transputer is running, the transputer will halt at the next descheduling point. From **Analyse** being asserted, the processor will halt within three time slice periods plus the time taken for any high priority process to complete. **Reset** may then be asserted. When **Reset** comes low again the transputer will be in its reset state, but the registers contain information on the state of the machine when it was halted by the assertion of **Analyse**, permitting analysis of the halted machine.

Input links will continue with outstanding transfers. Output links will not make another access to memory for data but will transmit only those bytes already in the link buffer. Providing there is no delay in link acknowledgement, the links will be inactive within a few microseconds of the transputer halting.

4.2   T9 type control

T9-series transputers are controlled by a pair of control links, **CLink0-1**, on each device (T9-type control).

The control links on all T9-series transputer family products allow a separate control network to be used to assist in configuring, booting, error handling, resetting and analysing processors and other components connected in a system, even in the presence of errors on the data communications links in the network. Many of these functions can also be performed directly by software running on an IMS T9000 transputer.

Each IMS T9000 transputer has two bidirectional control links, **CLink0** and **CLink1**, which use the same electrical and packet level protocols as the DS data links. **CLink0** will be connected via a control link network to one of the data links of a controlling IMS T9000 transputer, or to a different host via a link adaptor. All communications with the controlling processor are via **CLink0**. **CLink1** is provided to allow T9-series product family components to be connected in a daisy-chain. This allows a simple physical connectivity to be used for the controlling network, as shown in figure 4.1.

The controlling network provides each device with a virtual link connected to the control process.

When the network is initialized the first communication with each device programs identity and return addresses to establish the virtual link between the control process and that device. The identity address determines whether a packet arriving on **CLink0** is for that device, and if not, the packet is forwarded along **CLink1** until it reaches its destination.
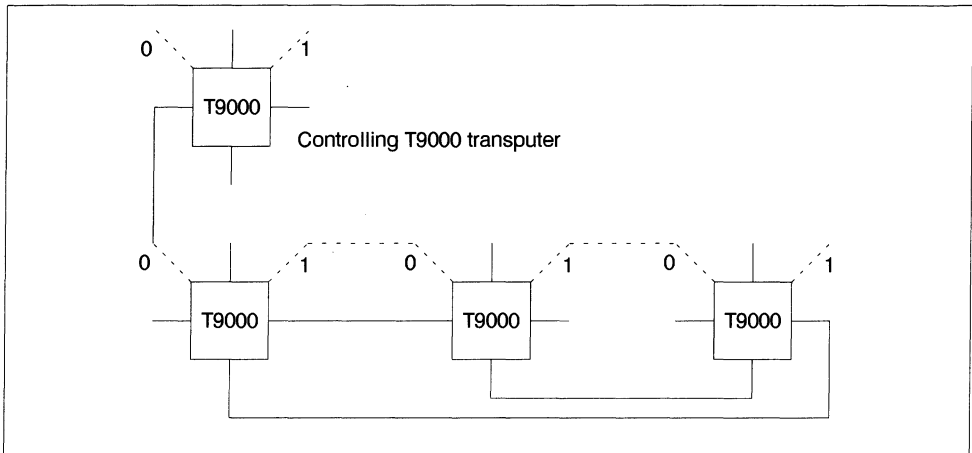
Figure 4.1   A daisy–chained control link network

A high level protocol is defined for the controlling network to allow the control process to issue commands to, and receive responses from, devices in the network. Commands are sent as normal packets but with the first byte after the header containing a command code, which may be followed by additional data.

4.3   Control protocol conversion

To enable T2/T4/T8–series subsystems to be easily incorporated into T9–series transputer systems (and vice versa) the IMS C100 converts between the two control systems described above. The subsystem to be connected can be controlled either through reset, analyse and error signals, or through control link **CLink0** of the IMS C100:

> **RAE master:** The **TReset** and **AnalyseIn** pins of the IMS C100 act as master and errors are reported by the **Error** pin of the connected IMS T9000 transputer.
>
> The IMS T9000 transputer is booted from ROM. The ROM code sets the IMS T9000 DS links in byte mode and emulates the boot time behaviour of the T2/T4/T8–series transputer. That is, it allows code to be booted down the data links of the IMS T9000 transputer in the same way as for T2/T4/T8–series transputers.
>
> **CLink0 master:** The **CLink0** of the IMS C100 acts as master. Commands received on **CLink0** are converted either to signals on the **Reset** and **Analyse** pins, or into T2/T4/T8–series *Boot*, *Peek* and *Poke* messages transmitted along **OSLink0**. Signals on the **Error** pin are converted to *Error messages* transmitted along **OSLink0**.

The mode determines which conversion is to be carried out. The IMS C100 has two control links, one for issuing and receiving commands (**CLink0**) and one for daisy–chaining (**CLink1**). In mode 0 **CLink0** generates commands, and in modes 1–3 it is receptive to commands.

The two types of conversion are described in more detail below.

4.3.1   RAE master control (mode 0)

In mode 0 T2/T4/T8–type control is the master control and the IMS C100 translates **Reset** and **Analyse** pin signals from the T2/T4/T8–series transputer to control link commands in T9–series transputers.

In this mode the IMS C100 cannot receive commands from an IMS T9000 transputer, but can issue commands via **CLink0**. In effect the IMS C100 acts like the root process in a network of IMS T9000 transputers. The commands generated are the same as those received by an IMS T9000.

Figure 4.2 illustrates mode 0 in which the IMS C100 converts a T2/T4/T8–series interface to an T9–series interface by translating the **Reset** and **Analyse** pin signals from the T2/T4/T8–series transputer into commands sent via **CLink0** to the IMS T9000 transputer.

Note in this mode **CLink0** of the IMS C100 is connected to **CLink0** of the IMS T9000 transputer. The standard connection of control links is to connect **CLink0** to **CLink1.**
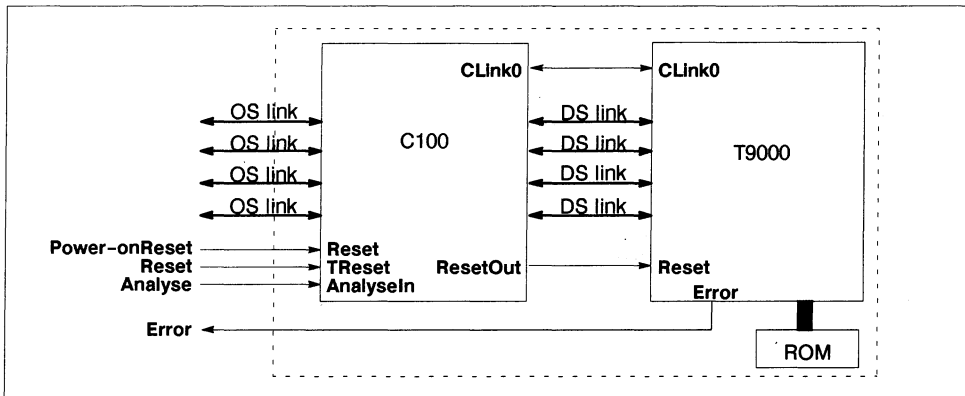
Figure 4.2   RAE master control mode 0

The IMS C100 sends commands as packets. Each message is preceded by the return header and followed by an EOM token. The first byte after the header contains a command code, which may be followed by additional data. The first bit of the command code indicates whether the packet is a command/error or a handshake. Table 4.1 outlines the command codes that can be sent from an IMS C100 in mode 0 to a connected IMS T9000 transputer. It also describes the effect of the commands on a connected IMS T9000.

Command	Additional data	Function
Start	Return header	Programs the IMS T9000's **CLink0** by allocating an identity and return header.
Reset	Level	Resets the IMS T9000 to the given level.
Stop	None	Stops the processor 'cleanly' so that register values are preserved. Acts like the **Analyse** pin on the T8 transputer.
Reboot	None	Causes reboot from ROM

Table 4.1   Commands sent from the IMS C100 in mode 0 to an IMS T9000

Table 4.2 lists the response messages received by the IMS C100. The response message is a handshake code corresponding to the command message. The data parameter 'Status' indicates whether or not there has been an error in performing the operation. Status can be 1 if the command was in some way incorrect or inappropriate, and consequently not obeyed, or 0 otherwise.

Response	Additional data
StartHandShake	None
ResetHandShake	Status
StopHandShake	Status
ReBootHandShake	Status
Error	Error code

Table 4.2   Response messages received by the IMS C100 in mode 0

After power–on reset a *Start* command sent from the IMS C100, via **CLink0**, provides the return header to program **CLink0** of the attached IMS T9000 transputer. The IMS T9000 returns a *StartHandShake* which programs the identity of the IMS C100. This forms the virtual link between the root (IMS C100) and the node (IMS T9000).

The IMS C100 sends *Reset*, *Stop*, *Reboot* commands via **CLink0** as a response to pins going high in given sequences, as described below.

If the **Reset** pin goes high then the IMS C100 will send a *Reset* command (reset level :1) to the attached IMS T9000 (this resets all registers, stops the PMI, VCP and CPU but retains the control links identity and return header). The IMS C100 will receive the *ResetHandShake* and the **Reset** pin will be taken low. A *ReBoot* command will then be sent to the IMS T9000.

Reboot

The *Reboot* command will cause the attached IMS T9000 to boot from ROM using a **Wptr** and **Iptr** from a fixed location in ROM. The ROM code, configures the IMS T9000, sets the links into byte mode, starts them, and then emulates the T2/T4/T8–series pre-boot protocol.

Analyse

In response to the **AnalyseIn** pin being asserted the IMS C100 will send a *Stop* command from **CLink0** to the IMS T9000. The *Stop* command causes the processor to be stopped whilst preserving register values.

When the **TReset** pin is asserted a *Reset* command (reset level 3 - to stop the CPU) is sent. The *Reset* command is followed by a *ResetHandshake* from the IMS T9000. When both **TReset** and **AnalyseIn** are deasserted the IMS C100 sends a *Reboot* command. This restarts the ROM code. If this code executes a *testpranal* instruction it can take special action to assist the debugger before it repeats the above pre-boot sequence.

Error

If an error occurs on the IMS T9000, this is signalled by the **Error** pin. It also causes an *Error* message to be sent from **CLink0** of the IMS T9000, which is received by **CLink0** of the IMS C100 and ignored.

4.3.2   CLink0 master control (modes 1, 2 and 3)

In modes 1, 2 and 3, T9–type control is the master control. In these modes **CLink0** and **CLink1** act as a daisy chain (see figure 4.3) with **CLink0** saving the header of the first packet it receives, and only inputting subsequent packets with the same header. Packets with a different header are relayed out of **CLink1**. All packets received on **CLink1** are relayed out of **CLink0**. There is a fair arbiter to deal with the case that the IMS C100 needs to send a packet at the same time as a packet arrives on **CLink1**. Note this is identical to the daisy-chaining behavior of the IMS T9000 (as described in the *IMS T9000 Preliminary Information*).

All packets received on **CLink0** with the same header as the first packet received are input by the IMS C100 and decoded as either acknowledge packets (which allow further messages to be sent by the IMS C100), or as messages. Messages are further subdivided into commands and handshakes. A handshake indicates that a previously sent error message has been received, and so another can be sent.
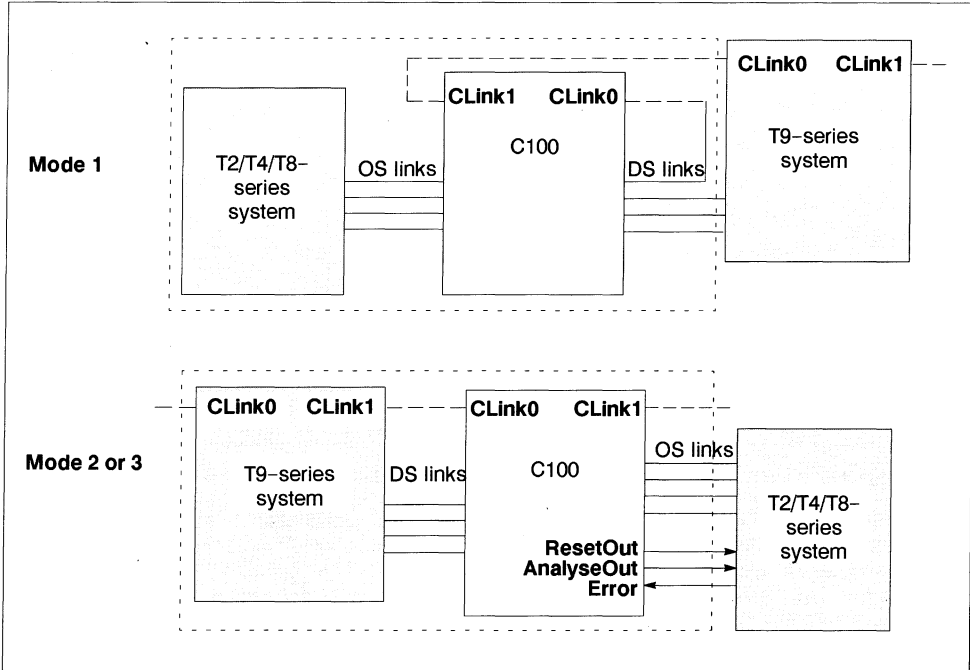

Figure 4.3   **CLink0** master control modes 1, 2 and 3

Table 4.3 details the commands which can be sent from the T9–series control processor to the IMS C100. Each command is terminated by an EOM token.

The commands recognized by the IMS C100 are the same as those received by an IMS T9000 transputer. However the execution of commands is adapted to the appropriate T2/T4/T8–series behavior.

Command	Additional data	Function
Start	Return header	Programs the C100's **CLink0** by allocating an identity and return header.
Reset	Level	Resets the IMS C100 and consequently the connected T2/T4/T8-series transputer.
Identify	None	Returns the identity and the revision number of the device.
Stop	None	Asserts the **AnalyseOut** pin, resulting in the **Analyse** pin on the connected T2/T4/T8-series transputer being asserted.
CPeek	Address	Returns the value stored at the given address in the device configuration space. If the address is invalid an invalid status is returned.
CPoke	Address, data	Writes data to the configuration space at the given address. If the address is invalid an invalid status is returned.
Peek16	Address	Peeks from connected 16 bit transputer.
Peek32	Address	Peeks from connected 32 bit transputer.
Poke16	Address, data	Pokes to connected 16 bit transputer.
Poke32	Address, data	Pokes to connected 32 bit transputer.
Run16	**Wdesc, Iptr**	Ignored and handshake sent with invalid status.
Run32	**Wdesc, Iptr**	Ignored and handshake sent with invalid status.
Boot16	Address, length	Starts boot sequence (16 bit words).
Boot32	Address, length	Starts boot sequence (32 bit words).
BootData	Data	Continues the boot sequence.
ReBoot	None	Causes reboot from ROM.
RecoverError	None	Used in error recovery on control system failure.
ErrorHandshake	None	Handshakes error message.

Table 4.3   Commands received by the IMS C100 in modes 1, 2 and 3 from an IMS T9000

Each command message is acknowledged by an acknowledge packet which is a packet containing no data and terminated by an EOP token. In addition the higher level control protocol requires that all command messages are acknowledged by a response message before the control process can send another command message to the same device, so appropriate responses must be generated by the IMS C100 in this mode of operation. (However, *Start*, *Reset* and *RecoverError* command messages may be sent to any node at any time to allow the control process to handle error conditions in the network.)

The response message can contain the result of a *Peek* or *Identify* command, or it may be simply a handshake code corresponding to the command message. Each message is preceded by the return header and followed by an EOM token. Table 4.4 lists the response messages to each of the command messages. The data parameter 'Status' indicates whether or not there has been an error in performing the operation. Status can be 1 if the command was in some way incorrect or inappropriate, and consequently not obeyed, or 0 otherwise.

Commands sent which cannot be converted to T2/T4/T8-series actions, or commands which are illegal in certain states, are handshaken with status set to 1.

Response	Additional data
StartHandShake	None
ResetHandShake	Status
IdentifyResult	Device type and rev
StopHandShake	Status
CPeekResult	Data, status
CPokeHandShake	Status
Peek16Result	Data, status
Peek32Result	Data, status
Poke16HandShake	Status
Poke32HandShake	Status
Run16HandShake	Status
Run32HandShake	Status
StartBoot16HandShake	Status
StartBoot32HandShake	Status
BootDataHandShake	Status
ReBootHandShake	Status
RecoverHandShake	None
Error	Error code

Table 4.4   Messages sent by the IMS C100 in modes1, 2 and 3

The IMS C100 error codes are listed in table 4.5.

Error code	Cause of error
0	Parity or disconnect error on **CLIink1**
1	Protocol error on **CLIink0** e.g. bad command length, extra acknowledge
2	Unrecognized command code on **CLIink0**
3	Signal on **Error** pin
4	Parity or disconnect error on **DSLIink0**
5	Parity or disconnect error on **DSLIink1**
6	Parity or disconnect error on **DSLIink2**
7	Parity or disconnect error on **DSLIink3**
8	Overlong packet on **OSLink0**
9	Overlong packet on **OSLink1**
10	Overlong packet on **OSLink2**
11	Overlong packet on **OSLink3**
12	Invalid count **OSLink0** (i.e. count = 0)
13	Invalid count **OSLink1**
14	Invalid count **OSLink2**
15	Invalid count **OSLink3**

Table 4.5   Error codes

All the error codes must be handshaken from the root processor with the *ErrorHandShake* command.

Errors are reported by sending an error message with the corresponding code, or, in the case of an error on **CLink0**, by causing a disconnection. Software at the root processor can then take appropriate action.

There are four control link commands which correspond to the special protocol of an unbooted T2/T4/T8– series transputer. These cause messages to be generated from **OSLink0** in certain modes of operation of the IMS C100 for which the behavior of **OSLink0** is defined to be special (see section 4.3.3). The assumed length (N) of addresses and data can be either 16 or 32 bits depending on whether a command is being sent to a 16 bit or 32 bit transputer.

Peek

On receipt of a *PeekN* command, and the associated peek address, the IMS C100 sends from **OSLink0** the following sequence of bytes:

1(BYTE);address[0];...address[N]

When the last byte has been sent and acknowledged the IMS C100 awaits an associated response. A *PeekNResult* is returned with the returned bytes as data and a status byte of 0.

If the communication does not complete (for example if there is no transputer connected), the peeking process will not receive a *PeekNResult* and can time-out, and, if required, reset the IMS C100 with a *Reset* command.

Poke

On receipt of a *PokeN* command, and the associated poke address and data, the IMS C100 sends from **OSLink0** the following sequence of bytes:

0(BYTE);address[0];...address[N];data[0];...data[N]

The command is acknowledged immediately, and if and when the last byte of the above communication is acknowledged, a *PokeNHandshake* is returned with a status of 0.

If the communication does not complete (for example because there is no transputer connected after all), the poking process will not receive a *PokeNHandshake* and can time-out, and reset the IMS C100.

Boot

On receipt of a *BootN* command, and the associated boot address and length byte, the IMS C100 sends the length byte from **OSLink0** and discards the address.

The *BootN* command is acknowledged immediately, and if and when the length byte is acknowledged by a connected transputer, a *BootNHandShake* response is sent with a status of 0.

The value of the length byte is kept by the IMS C100 as a count, and that number of bytes are then received by **CLink0**, as a sequence of *BootData* messages. The bytes are sent out on the OS link. Each arriving *BootData* message is acknowledged immediately, but not handshaken until all its data bytes have been sent and acknowledged on the OS link. Once all bytes have been sent and acknowledged a *BootData-HandShake* is sent with a status byte of 0.

After a *BootN* command has been received, the **booting** flag is set, and any further *PeekN*, *PokeN* or *BootN* commands are invalid. Once the number of bytes as allowed for in the count of the *BootN* command have been received, the **booting** flag is unset, and the **booted** flag is set; *BootData* commands are then also invalid. All such invalid commands are acknowledged and handshaken immediately, but with a status byte of 1. No other action is taken. The **booting** and **booted** flags are reset by any *Reset* command.

Reset

On receipt of a *Reset* command on **CLink0**, the IMS C100 asserts its **ResetOut** pin. This pin is deasserted by a *CPoke* command. Whilst the **ResetOut** pin is asserted *Stop, PeekN, PokeN, BootN, BootData* and *Run* commands are invalid and will be handshaken immediately with a status of 1.

Analyse

On receipt of a *Stop* command on **CLink0**, the IMS C100 asserts its **AnalyseOut** pin. This pin is deasserted by a *CPoke* command. Whilst the **AnalyseOut** pin is asserted *PeekN, PokeN, BootN, BootData* and *Run* commands are invalid and will be handshaken immediately with a status of 1.
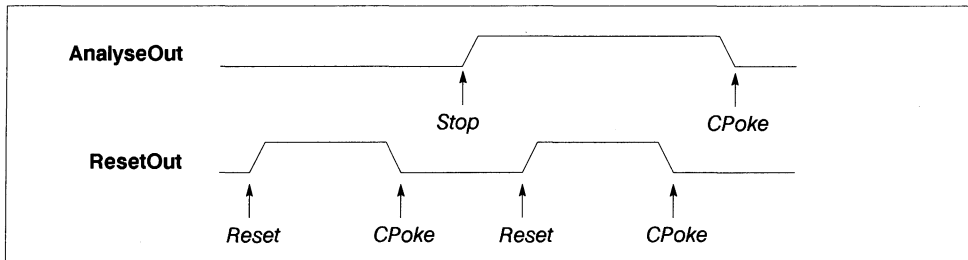
Figure 4.4   Resetting and analyzing in modes 1, 2 and 3

4.3.3   OS Link 0 special function

In modes 2 and 3, the control links are the system master, and the default assumption is that at least **OSLink0** is connected to an unbooted T2/T4/T8–series transputer. Commands to peek, poke and boot the T2/T4/T8–series transputer, arriving down **CLink0**, are converted to T2/T4/T8–series protocol and sent down **OSLink0**. **OSLink0** is usurped to generate the pre-boot protocol of the T2/T4/T8–series transputer until the transputer is booted.

This default is controlled by the **booted** flag which is set automatically by the booting sequence (*BootN, BootData*) or *Reboot,* and reset by the *Reset* command.

5   Links

5.1   Data links

The IMS C100 has eight data links. **OSLink0-3** are oversampled links and **DSLink0-3** are data/strobe links. Each OS link is paired with a DS link, for data protocol conversion. All pairs of links perform one or other type of conversion, depending on mode.

Each pair of links is joined by a conversion unit. **OSLink0** can be diverted into the control conversion unit by a switch which is controlled by the **booted** flag. Refer to section 4.3.3 for description of **OSLink0** special function in modes 2 and 3.

The OS and DS links are TTL compatible.

5.1.1   Data link speeds

There are four pins to set the operating speed of the links. **OSLinknSpecial** pins set the operating speed of the OS links and **DSLinknSpecial** pins set the default speed of the DS links.

OSLink0-3 support a communication speed of 10 Mbits/sec. In addition they can be used at 20 Mbits/sec which is determined by the **OSLinknSpecial** pin. Links are not synchronized with **ClockIn** and are insensitive to their phases. Thus links from independently clocked systems may communicate, providing only that the clocks are nominally identical and within specification.

The **OSLink0Special** pin enables the speed of **OSLink0** to be set independently of OS links 1, 2, 3 and the **DSLink0Special** pin enables the default speed of **DSLink0** to be set independently of DS links 1, 2, 3 (see table 5.1).

The OS link speeds must only be set at power-on. If these pins are changed after power-on the IMS C100 is not guaranteed to function correctly until it has been reset.

OSLink0Special	OSLink123Special	OS link speed
0	0	10 MHz
1	1	20 MHz
DSLink0Special	**DSLink123Special**	**DS default link speed**
0	0	25 MHz
1	1	50 MHz

Table 5.1   **LinkSpecial** pins

5.1.2   DS links in modes 1, 2 and 3

The IMS C100 DS links can support a range of communication speeds, which are programmed by writing to registers in the configuration space using the *CPoke* command via **CLink0**. At reset all data DS links run at the default speed determined by the **DSLink0Special** and **DSLink123Special** pins.

Only the transmission speed of a DS link is programmed as reception is asynchronous. This means that DS links running at different speeds can be connected, provided that each device is capable of receiving at the speed of the connected transmitter.

The transmission speeds of all of the DS links (data and control links) on a given device are related to the speed of a single on-chip clock. The frequency of this master clock is programmed through the **SpeedMultiply** bit field. The master frequency is divided down to obtain the transmission frequency for each DS link. The division factor can be programmed separately for each DS link via the **SpeedDivide** bit field. For a given device, with a given programmed master clock frequency, this arrangement allows each DS link to

be run at one of four transmission speeds, as shown in table 8.2. The **BaseSpeed** is the default speed of 10 MHz.

SpeedMultiply	SpeedDivide				BaseSpeed
	/1	/2	/4	/8	
8	80	40	20	10.0	10
10	100	50	25	12.5	10
12	Reserved	60	30	15.0	10
14	Reserved	70	35	17.5	10
16	Reserved	80	40	20.0	10
18	Reserved	90	45	22.5	10
20	Reserved	100	50	25.0	10

Table 5.2   DS link transmission speed in Mbits/sec

Errors on DS links

DS link inputs detect parity and disconnection conditions as errors. A disconnection error indicates one of two things: either the DS link has been physically disconnected, or an error has occurred at the other end of the DS link which has then stopped transmitting. The bit fields **ParityError** and **DiscError** indicate when parity and disconnect errors occur.

The DS links are designed to be highly reliable within a single subsystem and can be operated in one of two environments, determined by the **LocalizeError** bit in each link.

In the majority of applications, the communications system should be regarded as being totally reliable. In this environment errors are considered to be very rare, but are treated as being catastrophic if they do occur. This environment is the default on power–on reset, with all links having their **LocalizeError** bit set to 0. If an error occurs it will be detected and reported via a message sent along **CLink0**. Normal practice will then be to reset the subsystem in which the error has occurred and to restart the application.

For some applications, for instance when a disconnect or parity error may be expected during normal operation, an even higher level of reliability is required. This level of fault tolerance is supported by localizing errors to the link on which they occur, by setting the **LocalizeError** bit of the link to 1. In addition a *data link layer* process must be connected to each virtual channel associated with the link. These processes are responsible for establishing and maintaining a higher level flow control, using time-out to detect that a message has not completed, and requesting retransmission. If an error occurs, packets in transit at the time of the error will be discarded or truncated, and the link will be reset without the error being reported via the control link.

For information on *the data link layer* refer to chapter 4 of *'Computer Networks' by Andrew S. Tanenbaum, published by Prentice–Hall International (ISBN: 0–13–166836–6).*

5.2   Control links

The IMS C100 has two bidirectional control links; **CLink0** and **CLink1**. They use the same electrical and packet level protocols as the DS links (refer to section 3.2).

All communications with the controlling processor are via **CLink0**. **CLink1** provides a daisy–chain link, allowing a simple physical connectivity to be used for controlling networks.

The behavior of **CLink0** depends on the mode as detailed in section 4.3.

5.2.1   Control link speeds

After power–on the control links run at a default speed of 10 MHz; this can be changed by means of *CPokes.*

6   Configuration

6.1   Configuration space

The IMS C100 can be controlled via the configuration address space. These addresses are accessed by *CPeek* and *CPoke* command messages received along **CLink0**.

The configuration bus can be used to reset any individual DS link or any link pair in packetized conversion mode, modes 1 and 3.

Table 6.1 gives the configuration space map.

Address	Function	Reset Value	Notes
#1001	IMS C100 Device type		Also used by the *Identify* command
#1002	IMS C100 Device type and rev		
#1003	IMS C100 **Command/Status** word	see table 6.2 and table 6.3	Write to command word, read from status word
#1005	IMS C100 **DSLinkPLL**	see table 8.3	
#8001	**DSLink0Mode**		
#8101	**DSLink1Mode**		
#8201	**DSLink2Mode**		
#8301	**DSLink3Mode**		
#8002	**DSLink0Command**		
#8102	**DSLink1Command**		
#8202	**DSLink2Command**		
#8302	**DSLink3Command**		
#8003	**DSLink0Status**	see Reset chapter 7	
#8103	**DSLink1Status**		
#8203	**DSLink2Status**		
#8303	**DSLink3Status**		
#FD01	**CLink0Mode**		
#FE01	**CLink1Mode**		
#FD02	**CLink0Command**		
#FE02	**CLink1Command**		
#FD03	**CLink0Status**	see Reset chapter 7	
#FE03	**CLink1Status**	Depends on mode	

Table 6.1   Configuration space map

The IMS C100 **Command** and **Status** words have the structure shown below.

Bit	Function
4	Link pair 0 reset
5	Link pair 1 reset
6	Link pair 2 reset
7	Link pair 3 reset
30	End **Reset**
31	End **Analyse**

Table 6.2   **Command** word

Bit	Status of pin
16	**Mode0**
17	**Mode1**
18	**DSLink0Special**
19	**DSLink123Special**
20	**OSLink0Special**
21	**OSLink123Special**

Table 6.3   **Status** word

A bit set in the **Command** word effects the indicated function. The command word is write only.
A bit set in the **Status** word indicates the current status. The status word is read only.

6.2   Data DS link configuration registers

Each DS link has three registers, the **DSLinkMode** register, **DSLinkCommand** register and **DSLinkStatus** register.

In addition the configuration space contains the **DSLinkPLL** register which contains the **SpeedMultiply** bit. This takes the 5 MHz input clock and multiplies it by a programmable value to provide the root clock for all the DS links.

The tables below describe the functionality of the DS links to be controlled, and the associated bit fields in the configuration registers.

Bit	Bit field	Function
5:0	**SpeedMultiply**	Sets DS link master clock to required value (see table 8.2).

Table 6.4   Bit fields in the **DSLinkPLL** register

The **DSLink0-3Mode** registers power up into a default state and may be reprogrammed before or after the link has been started.

Bit	Bit field	Function
1:0	**SpeedDivide**	Sets transmit speed of the **DSLink** (see table 8.2). 00 = /1, 01 = /2, 10 = /4, 11 = /8
2	**SpeedSelect**	Sets the **DSLink** to transmit at the speed determined by the **SpeedDivide** bits as opposed to the base speed of 10 Mbits/s.
3	**LocalizeError**	When set errors are no longer reported to the control link. Packets in transit at the time of an error will be discarded or truncated.

Table 6.5   Bit fields in the **DSLink0-3Mode** registers

The **DSLink0-3Command** registers are write only and contain four bits which when set cause a specific action to be taken by the DS link.

Bit	Bit field	Function
0	**ResetLink**	Resets the link engine of the **DSLink**. The token state is reset, the flow control credit is set to zero, the buffers are marked as empty, and the parity state is reset.
1	**StartLink**	When a transition from 0 to 1 occurs the **DSLink** will be initialized and commence operation.
2	**ResetOutput**	Sets both outputs of the **DSLink** low.
3	**WrongParity**	The **DSLink** output will generate incorrect parity. This may be used to force a parity error on a transputer at the other end of the **DSLink**.

Table 6.6   Bit fields in the **DSLink0-3Command** registers

The **DSLink0-3Status** registers are read only and contain six bits which contain information about the state of the DS link.

Bit	Bit field	Function
0	**LinkError**	Flags that an error has occurred on the **DSLink**.
1	**LinkStarted**	Flags that the output **DSLink** has been started and no errors have been detected.
2	**ResetOutputComplete**	Flags that **ResetOutput** has completed on the **DSLink**.
3	**ParityError**	Flags that a parity error has occurred on the **DSLink**.
4	**DiscError**	Flags that a disconnect error has occurred on the **DSLink**.
5	**TokenReceived**	Flags that a token has been seen on the **DSLink** since **ResetLink**.

Table 6.7   Bit fields in the **DSLink0-3Status** registers

6.3   Control link configuration registers

The link module hardware in each control link is identical to that in each data DS link. An equivalent set of configuration bit fields is provided for each control link, as for the data DS links.

7 Levels of reset

The IMS C100 can be reset to a given level using the *Reset* command or **Reset** pin. The different levels of reset are described below.

7.1 Resetting links

There are two basic mechanisms for resetting links. One applies in the byte–stream conversion mode, in which case a specific packet received from an attached IMS T9000 or similar causes the OS link of the pair and the internal state of the data conversion unit to be reset. The other mechanism is the configuration bus, which can be used to reset any individual DS link or any link pair, see section 6.1.

7.2 Level 0 – hardware reset

In all modes the IMS C100 is reset by asserting the **Reset** pin high. The **ResetOut** pin follows the **Reset** pin. In mode 0 the IMS C100 is also reset by a similar transition on **TReset**, providing **AnalyseIn** is low.

After a hardware reset has been deasserted each IMS C100 is in the following state:

All the links are in Wait state, with the data links operating at their default speed set by the **LinkSpecial** pins and the control links operating at their default speed of 10 MHz. The identity and return headers for the control links are undefined. All registers contain their default values. All buffers are cleared; all latched error signals are cleared; and the **AnalyseOut** pin is taken low.

7.3 Level 1 – labelled control network

The network can be reset to level 1 by sending a *Reset1* command message to each IMS C100.

This level of reset leaves the identity and return headers unaltered and all connected control links remain operational. All the data links are in Wait state and operate at the default speed set by the **LinkSpecial** pins. All registers are reset to their default values. All buffers are cleared.

The **ResetOut** pin is set high.

7.4 Level 2 – configured network

The network can be reset to level 2 by sending a *Reset2* command message to each IMS C100.

At this level of reset the identity and return headers are unaltered and register contents are unaffected. All buffers are cleared. The data links are reset and returned to the Wait state.

The **ResetOut** pin is set high.

7.5 Level 3

If a *Reset3* command message is received, for example from an IMS T9000 transputer, it is handshaken with a status of 0.

The **ResetOut** pin is set high.

8   Software

8.1   Toolsets

The IMS Dx2xx toolsets refers to the C, occam and FORTRAN toolsets written in C and supporting T2/T4/T8-series transputer networks.

A set of C, occam and FORTRAN toolsets is also available which incorporate T9-series transputer support. The tools provide support in the configuration and initialization of T9-series networks. The tools set the attributes of each device in the T9-series network by sending initialization data down the control link, and set the processors into a state ready to receive an application down the data DS links. A Network Description Language (NDL) is used to describe networks of devices and allows the specification of values for all the attributes of a device. From the NDL file the initialization tools produce a file containing the network initialization data. This data is sent down the control link to the network. Once the network has been initialized, programs can be built and loaded to the network in the same way as for T2/T4/T8-series processors.

The IMS T9000 configuration tools do not directly support the configuration of mixed T9-series and T2/T4/T8-series systems. Systems made up of T9-series networks and T2/T4/T8-series networks connected together via IMS C100s can be configured, with each network being configured and loaded separately using the appropriate toolset. The user is able to specify (in the NDL file) the edges of an T9-series network which communicate with a T2/T4/T8-series network.

9 Pin designations

Pin	In/Out	Function
VCC, GND		Power supply and return
CapPlus, CapMinus		External capacitor for internal clock power supply
ClockIn	in	5 MHz input clock
ClockOut0–1	out	Internally generated high speed clock output.

Table 9.1   IMS C100 system services

Pin	In/Out	Function
Reset	in	System reset
ResetOut	out	Asserts the **Reset** pin on any connected T9–series or T2/T4/T8–series device.
TReset	in	Mode 0 T2/T4/T8–series transputer reset
Error	in	Modes 1–3 error indicator – message sent from **CLink0**
AnalyseIn	in	Mode 0 error analysis
AnalyseOut	out	Mode 1–3 error analysis – message received on **CLink0**
Mode0–1	in	Mode of operation

Table 9.2   IMS C100 control unit

Pin	In/Out	Function
OSLinkIn0–3	in	OS link input data channels
OSLinkOut0–3	out	OS link output data channels
DSLinkInData0–3	in	DS link input data channels
DSLinkInStrobe0–3	in	DS link input strobes
DSLinkOutData0–3	out	DS link output data channels
DSLinkOutStrobe0–3	out	DS link output strobes
CLinkInData0–1	in	Control link input data channels
CLinkInStrobe0–1	in	Control link input strobes
CLinkOutData0–1	out	Control link output data channels
CLinkOutStrobe0–1	out	Control link output strobes
OSLink0Special	in	OS link 0 speed selection
OSLink123Special	in	OS link 1, 2, 3 speed selection
DSLink0Special	in	DS link 0 speed selection
DSLink123Special	in	DS link 1, 2, 3 speed selection

Table 9.3   IMS C100 links